CHAPTER 7

The Extension C API

This chapter is about teaching SQLite new tricks. The previous chapter dealt with generic database work; this chapter is about being creative. The latter half of the API—the Extension API—offers three basic ways to extend (or customize) SQLite, through the creation of user-defined *functions*, *aggregates*, and *collation sequences*.

User-defined functions are SQL functions that map to some implementation that you write. They are callable from within SQL. For example, you could create a function `hello_newman()` that returns the string `'Hello Jerry'` and, once it is registered, call it from SQL as follows:

```
sqlite > select hello_newman() as reply;
reply
------------------
'Hello Jerry'
```

This is a special version of the SQLite command-line program I customized myself that includes `hello_newman()`. When I get bored, I call it occasionally.

Aggregates are a special form of function and work in much the same way except that they operate on sets of records and return aggregated values or expressions computed over a particular column in the set. Or they may be computed from multiple columns. Standard aggregates that are built into SQLite include `SUM()`, `COUNT()`, and `AVG()`, for example.

Collations are methods of comparing and sorting text. That is, given two strings, a Collation would decide which one is greater, or whether the two are equal. The default Collation in SQLite is `BINARY`—it compares strings byte by byte using `memcmp()`. While this collation happens to work well for words in the English language (and other languages that use UTF-8 encoding), it doesn't necessarily work so well for other languages that use different encodings. This is why SQLite version 3 introduced user-defined collations.

This chapter covers each of these three user-defined extension facilities and their associated API functions. As you will see, when combined with other facilities such as triggers and conflict resolution, the user-defined extensions API can prove to be a powerful tool for creating nontrivial features that extend SQLite.

All of the code in this chapter is included in examples that you can obtain online from the Apress website (`www.apress.com`). The examples for this particular chapter are located in the *ch7* subdirectory of the examples folder. I will specifically point out all the source files corresponding to the examples as they are introduced in the text. Some of the error-checking code in the figures has been taken out for the sake of clarity.

Also, many examples in this chapter use simple convenience functions from a common library I wrote that helps simplify the examples. I will point out these functions and explain what they do as they are introduced in the text. The common code is located in the *common* subdirectory of the examples folder, and must first be built before the examples are built. Please refer to the README file in the examples folder for details on building the common library.

Finally, to clear up any confusion, the terms *storage class* and *data type* are interchangeable. You know that SQLite stores data internally using five different data types, or storage classes. These classes are `INTEGER`, `FLOAT`, `TEXT`, `BLOB`, and `NULL`, as described in Chapter 4. Throughout the chapter, I try to use the term that best fits the context. When speaking generally, I use the term *data type*, as it seems the most appropriate. When speaking about how SQLite handles a specific value internally, I find that the term *storage class* seems to work better. In either case, you can equate either term as describing a specific form or representation of data.

The API

The basic method for implementing functions, aggregates, and collations consists of implementing a callback function and then registering it in your program. Once registered, they can be used from SQL. Functions and aggregates use the same registration function and similar callback functions. Collation sequences, while using a separate registration function, are almost identical and can be thought of as a particular class of user-defined function.

The lifetime of user-defined aggregates, functions, and collations is transient. They are registered on a connection-by-connection basis; they are *not* stored in the database. It is up to you to ensure that your application loads your custom extensions and registers them on each connection. Sometimes you may find yourself thinking of your extensions in terms of stored procedures and completely forget about the fact that they don't actually reside in the database. Extensions exist in libraries, not in databases, and are restricted to the life of your program. If you don't grasp this concept, you may do something like try to create a trigger (which is stored in the database) that references your extensions (which aren't stored in the database) and then turn around and try to call it from the SQLite shell or from an application that knows nothing of your extensions. In this case, you will find that either it triggers an error or nothing happens. Extensions require your program to register them on *each* and *every* connection; otherwise, that connection's SQL syntax will know nothing about them.

Registering Functions

That said, you can register functions and aggregates on a connection using the `sqlite3_create_function()`, which is declared as follows:

```
int sqlite3_create_function(
  sqlite3 *cnx,              /* connection handle                  */
  const char *zFunctionName, /* function/aggregate name in SQL     */
  int nArg,                  /* number of arguments. -1 = unlimited. */
  int eTextRep,              /* encoding (UTF8, 16, etc.)          */
  void *pUserData,           /* application data, passed to callback */
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

```
int sqlite3_create_function16(
  sqlite3 *cnx,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void *pUserData,
  void (*xFunc)(sqlite3_context*, int args, sqlite3_value**),
  void (*xStep)(sqlite3_context*, int args, sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

As in the previous chapter, I have included both the UTF-8 and UTF-16 versions of this function to underscore the fact that the SQLite API has functions that support both encodings. However, from now on, I will only refer to the UTF-8 versions for the sake of brevity.

The arguments for `sqlite3_create_function()` are defined as follows:

- `cnx`: The connection handle. Functions and aggregates are connection specific. They must be registered on the connection as it is opened in order to be used.
- `zFunctionName`: The function name as it will be called in SQL.
- `nArg`: The number of arguments required by the function. SQLite will enforce this as the exact number of arguments that must be provided to the function. However, if you specify -1 for this value, SQLite will allow a variable number of arguments.
- `eTextRep`: The encoding. Allowable values are `SQLITE_UTF8`, `SQLITE_UTF16`, `SQLITE_UTF16BE`, `SQLITE_UTF16LE`, and `SQLITE_ANY`. `SQLITE_ANY` is for a single function implementation that can negotiate any kind of text representation provided.
- `pUserData`: Application-specific data. This data is made available through the callback functions specified in `xFunc`, `xStep`, and `xFinal`. Unlike other functions, which receive the data as a void pointer in the argument list, these functions must use a special API function to obtain this data
- `xFunc`: The function callback. This is the actual implementation of the SQL function. Functions only provide this callback and leave the `xStep` and `xFinal` function pointers `NULL`. These latter two callbacks are exclusively for aggregate implementations.
- `xStep`: The aggregate step function. Each time SQLite processes a row in an aggregated result set, it calls `xStep` to allow the aggregate to process the relevant field value(s) of that row and include the result in its aggregate computation.
- `xFinal`: The aggregate finalize function. When all rows have been processed, SQLite calls this function to allow the aggregate to conclude its processing, which usually consists of computing the final value and optionally cleaning up.

With regard to encoding, you can register multiple versions of the same function differing only in the encoding (`eTextRep` argument) and SQLite will automatically select the best version of the function for the case in hand.

The Step Function

The function and aggregate step functions are identical, and are declared as

```
void fn(sqlite3_context* ctx, int nargs, sqlite3_value** values)
```

The ctx argument is the function/aggregate context. It holds state for a particular function call instance and is the means through which you obtain the application data (pUserData) argument provided in sqlite3_create_function(). The user data is obtained from the context using sqlite3_user_data(), which is declared as follows:

```
void *sqlite3_user_data(sqlite3_context*);
```

For functions, this data is shared among all calls to like functions, so it is not really unique to a particular instance of function call. That is, the same pUserData is passed or shared among all instances of a given function. Aggregates, however, can allocate their own state for each particular instance using sqlite3_aggregate_context(), declared as follows:

```
void *sqlite3_aggregate_context(sqlite3_context*, int nBytes);
```

The first time this routine is called for a particular aggregate, a chunk of memory of size nBytes is allocated, zeroed, and associated with that context. On subsequent calls with the same context (for the same aggregate instance), this allocated memory is returned. Using this, aggregates have a way to store state in between calls in order to accumulate data. When the aggregate completes the final() callback, the memory is automatically freed by SQLite.

■ **Note** One of the things you will see throughout the API is the use of user data in void pointers. Since many parts of the API involve callback functions, these simply serve as a convenience to help you maintain state when implementing said callback functions.

The nargs argument of the callback function contains the number of arguments passed to the function.

Return Values

The values argument is an array of SQLite value structures that are handles to the actual argument values. The actual data for these values is obtained using the family of sqlite3_value_*xxx*() functions, which have the following form:

```
xxx sqlite3_value_xxx(sqlite3_value* value);
```

where *xxx* is the C data type to be returned from the value argument. If you read Chapter 6, you may be thinking that these functions have a striking resemblance to the sqlite3_column_*xxx*() family of functions—and you'd be right. They work in exactly the same way, even down to the difference between the way scalar and array values are obtained. The functions for obtaining scalar values are as follows:

```
int sqlite3_value_int(sqlite3_value*);
long long int sqlite3_value_int64(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
```

The functions used to obtain array values are as follows:

```
int sqlite3_value_bytes(sqlite3_value*);
const void *sqlite3_value_blob(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
```

The `sqlite3_value_bytes()` function returns the amount of data in the value buffer for a `BLOB`. The `sqlite3_value_blob()` function returns a pointer to that data. Using the size and the pointer, you can then copy out the data. For example, if the first argument in your function was a `BLOB`, and you wanted to make a copy, you would do something like this:

```
int len;
void* data;

len = sqlite3_value_bytes(values[0]);
data = malloc(len);
memcpy(data, sqlite3_value_blob(values[0]), len);
```

Just as the `sqlite3_column_`*`xxx`*`()` functions have `sqlite_column_type()` to provide the column types, the `sqlite3_value_`*`xxx`*`()` functions likewise have `sqlite3_value_type()`, which works in the same way. It is declared as follows:

```
int sqlite3_value_type(sqlite3_value*);
```

This function returns one of the following values, which correspond to SQLite's internal, storage classes (data types), defined as

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
```

This covers the basic workings of the function/aggregate interface. We can now dive into some examples and even a few practical applications. The Collation interface is so similar that rather than previewing it here, we will just address it in full after we cover functions and aggregates.

Functions

Let's start with a trivial example, one that is very easy to follow. Let's implement `hello_newman()`, but with a slight twist. I want to include using function arguments in the example, so `hello_newman()` will take one argument: the name of the person addressing him. It will thus work as follows:

```
sqlite > select hello_newman('Jerry') as reply;
reply
------------------
Hello Jerry

sqlite > select hello_newman('Kramer') as reply;
reply
------------------
Hello Kramer

sqlite > select hello_newman('George') as reply;
reply
------------------
Hello George
```

The basic program (taken from `hello_newman.c`) is shown in Listing 7-1. Some of the error-checking code has been removed for clarity.

Listing 7-1. *The hello_newman() Test Program*

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db;

    sqlite3_open("test.db", &db);
    sqlite3_create_function( db, "hello_newman", 1, SQLITE_UTF8, NULL,
                             hello_newman, NULL, NULL);

    /* Log SQL as it is executed. */
    log_sql(db,1);

    /* Call function with one text argument. */
    fprintf(stdout, "Calling with one argument.\n");
    print_sql_result(db, "select hello_newman('Jerry')");

    /* Call function with two arguments. This will fail as we registered the
    ** function as taking only one argument. */
    fprintf(stdout, "\nCalling with two arguments.\n");
    print_sql_result(db, "select hello_newman ('Jerry', 'Elaine')");

    /* Call function with no arguments. This will fail too */
    fprintf(stdout, "\nCalling with no arguments.\n");
    print_sql_result(db, "select hello_newman()");

    /* Done */
    sqlite3_close(db);

    return 0;
}
```

This program connects to the database, registers the function, and then calls it three times. The callback function is implemented as follows:

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    const char *msg;

    /* Generate Newman's reply */
    msg = sqlite3_mprintf("Hello %s", sqlite3_value_text(values[0]));

    /* Set the return value. Have sqlite clean up msg w/ sqlite_free(). */
    sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

Running the program yields the following output:

```
Calling with one argument.
  TRACE: select hello_newman('Jerry')
hello_newman('Jerry')
---------------------
Hello Jerry

Calling with two arguments.
execute() Error: wrong number of arguments to function hello_newman()

Calling with no arguments.
execute() Error: wrong number of arguments to function hello_newman()
```

The first call consists of just one argument. Since the example registered the function as taking exactly one argument, it succeeds. The second call attempts to use two arguments, which fails. The third call uses no arguments, and also fails.

And there you have it: a SQLite extension function. This does bring up a few new functions that I have not addressed yet. The first few functions are specific to the common code. The function `log_sql()` simply calls `sqlite3_trace()`, passing in a tracing function that prefixes the traced SQL with the word *TRACE*. I use this so you can see what is happening in the example as the SQL is executed. The second function is `print_sql_result()`, which has the following declaration:

```
int print_sql_result(sqlite3 *db, const char* sql, ...)
```

This is a simple wrapper around `sqlite3_prepare()` and friends that executes a SQL statement and prints the results.

Return Values

This example introduces a new SQLite function: `sqlite3_result_text()`.This function is just one of a family of `sqlite_result_xxx()` functions used to set return values for user-defined functions and aggregates. The scalar functions are as follows:

```
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
```

These functions simply take a (second) argument of the type specified in the function name and set it as the return value of the function. The array functions are

```
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
```

These functions take array data and set that data as the return value for the function. They have the general form

```
void sqlite3_result_xxx(
    sqlite3_context *ctx,   /* function context */
    const xxx* value,       /* array value */
    int len,                /* array length */
    void(*free)(void*));    /* array cleanup function */
```

Here *xxx* is the particular array type—`void` for `BLOB`s or `char` for `TEXT`. Again, if you read Chapter 6, you may find these functions suspiciously similar to the `sqlite3_bind_xxx()` functions, which they are. Setting a return value for a function is, for all intents and purposes, identical to binding a value to a parameterized SQL statement. You could even perhaps refer to it as "binding a return value."

Arrays and Cleanup Handlers

Just as in binding array values to SQL statements, these functions work in the same way as `sqlite3_bind_xxx()`. They require a pointer to the array, the length (or size) of the array, and a function pointer to a cleanup function. This cleanup function pointer can be assigned the same predefined meanings as in `sqlite3_bind_xxx()`:

```
  #define SQLITE_STATIC      ((void(*)(void *))0)
  #define SQLITE_TRANSIENT   ((void(*)(void *))-1)
```

`SQLITE_STATIC` means that the array memory resides in unmanaged space, and therefore SQLite should not attempt to clean it up. `SQLITE_TRANSIENT` tells SQLite that the array data is subject to change, and therefore SQLite needs to make its own copy. In this case, SQLite will copy the data and free it after the function returns. The third option is to pass in an actual function pointer to a cleanup function of the form

```
void cleanup(void*);
```

In this case, SQLite will call the cleanup function after the user-defined function completes. This is the method used in the previous example: I used sqlite3_mprintf() to generate Newman's reply, which allocated a string on the heap. I then passed in sqlite3_free() as the cleanup function to sqlite3_result_text(), which could free the string memory when the extension function completed.

Error Conditions

Sometimes functions encounter errors, in which case the return value should be set appropriately. This is what sqlite3_result_error() is for. It is declared as follows:

```
void sqlite3_result_error(
    sqlite3_context *ctx, /* the function context */
    const char *msg,      /* the error message */
    int len);             /* length of the error message */
```

This tells SQLite that there was an error, the details of which are contained in msg. SQLite will abort the command that called the function and set the error message to the value contained in msg.

Returning Input Values

Sometimes, you may want to pass back an argument as the return value in the exact same form. Rather than your having to determine the argument's type, extract its value with the corresponding sqlite3_column_*xxx*() function, and then turn right around and set the return value with the appropriate sqlite3_result_*xxx*() function, the API offers sqlite3_result_value() so you can do all of this in one fell swoop. It is declared as follows:

```
void sqlite3_result_value(
    sqlite3_context *ctx,   /* the function context */
    sqlite3_value* value);   /* the argument value   */
```

For example, say you wanted to create a function echo() that spits back its first argument. The implementation is as follows:

```
void echo(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    sqlite3_result_value(ctx, values[0]);
}
```

It would work from SQL as follows:

```
sqlite > select echo('Hello Jerry') as reply;
reply
------------------
Hello Jerry
```

Even better, echo() works equally well with arguments of any storage class, and returns them accordingly:

```
sqlite> select echo(3.14) as reply, typeof(echo(3.14)) as type;
reply       type
----------  ----------
3.14        real
sqlite> select echo(X'0128') as reply, typeof(echo(X'0128')) as type;
reply       type
----------  ----------
(?         blob
sqlite> select echo(NULL) as reply, typeof(echo(NULL)) as type;
reply       type
----------  ----------
            null
```

A Complete Example

Let's move on to a complete example that uses most of the API functions described earlier. This will give you a better feel for all that is available to you in one place. We will implement a function called `function()`, which takes a variable number of arguments. It will print the argument types and return an integer value. If, however, the first argument is the string value `'fail'`, it will call the error handler, returning an error message.

The main function, shown in Listing 7-2, is much the same as the previous example (the complete example is located in `func.c`). Again, some of the error handling has been excised for clarity.

Listing 7-2. *The main Function*

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    const char* sql;

    sqlite3_open("test.db", &db);
    sqlite3_create_function( db, "function", -1, SQLITE_UTF8, NULL,
                             function, NULL, NULL);

    /* Turn on SQL logging */
    log_sql(db, 1);

    /* Call function with one text argument. */
    execute(db, "select function(1)");

    /* Call function with several arguments of various types. */
    execute(db, "select function(1, 2.71828)");
    /* Call function with variable arguments, the first argument's value
    ** being 'fail'. This will trigger the function to call
    ** sqlite3_result_error(). */
    execute(db, "select function('fail', 1, 2.71828, 'three', X'0004', NULL)");

    /* Done */
    sqlite3_close(db);

    return 0;
}
```

Note the -1 value for the nArg argument in sqlite3_create_function(). This corresponds to the number of required arguments that must be provided to the function. Using -1 tells SQLite that the function accepts a variable number of arguments. Note that there is a new convenience function here called execute(). This is just a thin wrapper around sqlite3_exec() that executes a SQL command and reports any errors to standard error, if any. It was listed in Chapter 6 and is part of the common library. The function's implementation is shown in Listing 7-3.

Listing 7-3. *A Vanilla User-Defined Function*

```
void function(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    int i; const char *msg;

    fprintf(stdout, "function() : Called with %i arguments\n", nargs);

    for(i=0; i < nargs; i++) {
        fprintf( stdout, "    arg %i: value=%-7s type=%i\n", i,
                 sqlite3_value_text(values[i]),
                 sqlite3_value_type(values[i]));
    }

    if(strcmp(sqlite3_value_text(values[0]), "fail") == 0) {
        msg = "function() : Failing because you told me to.";
        sqlite3_result_error(ctx, msg, strlen(msg));
        fprintf(stdout, "\n");
        return;
    }

    fprintf(stdout, "\n");
    sqlite3_result_int(ctx, 0);
}
```

Running the program produces the following output:

```
  TRACE: select function(1)
function() : Called with 1 arguments
    arg 0: value=1       type=1

  TRACE: select function(1, 2.71828)
function() : Called with 2 arguments
    arg 0: value=1       type=1
    arg 1: value=2.71828 type=2

  TRACE: select function('fail', 1, 2.71828, 'three', X'0004', NULL)
function() : Called with 6 arguments
    arg 0: value=fail    type=3
    arg 1: value=1       type=1
    arg 2: value=2.71828 type=2
    arg 3: value=three   type=3
    arg 4: value=        type=4
    arg 5: value=(null)  type=5
execute() Error: function() : Failing because you told me to.
```

First, we can see that SQLite classifies the values provided as arguments into five possible data types or storage classes. These possible types correspond to SQLite's five internal data types, which, again, are defined as

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
```

Each call passes in an increasing number of arguments and types to illustrate this. In the final call to `function()`, one of each storage class is passed in as an argument, and the function's output confirms that SQLite has typed them accordingly. Note that the `X'0004'` is an example of SQLite's string representation for binary data. The `X` (which is case insensitive) followed by hexadecimal values delimited by single quotes denotes binary data (a `BLOB` representation).

The first thing to notice in this example is that the function API provides a way to determine the number of arguments passed to a function, their respective storage classes, and their values.

Second, the function API provides a way to fetch argument values according to their storage class, if so desired. Since all storages classes have respective text representations—even binary data—you can always count on being able to get function arguments in text form, regardless of their internal representation.

Finally, just as you can receive values in their native storage class, you can also return them that way. One special case of return value is the error return value, which is always in text form, and causes the SQL command calling it to fail.

A Practical Application

Now let's move on to a real application of user-defined functions. In the introduction I made the claim that when combined with other SQLite features such as triggers, user-defined extensions can be quite a powerful tool. By design, SQLite uses manifest typing, which in many cases is very useful and convenient. But what if there are some critical columns in a table for which you just have to have strict typing? You might have an integer column that must always hold only integer values. No inserts or updates should be allowed to put any other kind of data type into this column.

In this example (`strict_typing.c`), we will combine user-defined functions with triggers to implement strict typing in SQLite. Furthermore, we will be able to apply this strict typing to entire tables, if so desired, and for an unlimited number of data types (that is, even for data types that SQLite knows nothing about). All that is required is that we implement validation functions for each respective data type that is to be enforced under the strict typing policy.

To start with, say we have a table `episodes`, defined as follows:

```
create table episodes (
    id integer primary key,
    cid integer not null default 0, -- main character id
    name text,                      -- episode name
    rating float not null default 0 -- scale 1-10 );
```

The `cid` column is the main character ID, which corresponds to the `id` column in the `character` table, which is defined as follows:

```
create table characters (
    id integer primary key, -- character id
    name text               -- character name );
```

We can infer from these tables that the `episodes.cid` column should always be an integer. Furthermore, we want to enforce this in our application. We know that in order to do this, we have to be able to control `INSERT` and `UPDATE` operations that target this column.

Validation Triggers

This is where triggers come in. We will define two triggers to do this, shown in Listing 7-4.

Listing 7-4. *Validation Triggers*

```
/-- INSERT trigger
CREATE TRIGGER episodes_insert_cid_typecheck_tr
BEFORE INSERT ON episodes
BEGIN
   SELECT CASE
     WHEN(SELECT validate_int(new.cid) != 1)
     THEN RAISE(ABORT, 'invalid int value for episodes.cid ')
   END;
END;
```

```
CREATE TRIGGER episodes_update_cid_typecheck_tr
BEFORE UPDATE OF cid ON episodes
FOR EACH ROW BEGIN
  SELECT CASE
    WHEN(SELECT validate_int(new.cid) != 1)
    THEN RAISE(ABORT, 'invalid int value for episodes.cid ')
  END;
END;
```

Thus, each `UPDATE` and `INSERT` on the `episodes` table will fire a *before* trigger, which will take the proposed `cid` value being submitted and call `validate_int()`—a user-defined function that simply tests whether a value (provided as the single argument) is in fact an integer. If the value is an integer, `validate_int()` returns true (1); otherwise, it returns false (0). The trigger evaluates the returned value in the `WHEN` clause and if it is not true, the trigger raises an abort. Therefore, only integer values can pass through the trigger and therefore all `INSERT`s and `UPDATE`s are only allowed to provide integer values to `episodes.cid`, giving us strict typing for this column.

■ **Note** This implementation makes an assumption you should be aware of. It assumes that `INSERT` statements always provide a value for the column to be validated. In this example, it assumes that all `INSERT` statements will always provide a value for `cid`. If one is not provided, the trigger will fail. To get around this, simply provide a default value for the column in the table definition so that a suitable value will be provided for `INSERT` statements that don't provide a `cid` value. If you notice, the definition for `cid` in this example is `cid integer not null default 0`. Another thing to remember is that while `INSERT` triggers can only be defined for entire records, `UPDATE` triggers can be defined for specific columns, so they do not have this problem.

The Validation Function

The `validate_int()` function is quite simple. It is shown in Listing 7-5.

Listing 7-5. *The validate_int() Validation Function*

```
void validate_int_udf(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    sqlite3 *db;
    const char *value;
    char *tmp;

    db    = (sqlite3*)sqlite3_user_data(ctx);
    value = sqlite3_value_text(values[0]);

    /* Assume NULL values for type-checked columns are not allowed */
    if(value == NULL) {
        sqlite3_result_int(ctx, 0);
        return;
    }
    /* Validate */
    tmp = NULL;
    strtol(value, &tmp, 0);

    if(*tmp != '\0') {
        /* Value does not conform to type */
        sqlite3_result_int(ctx, 0);
        return;
    }

    /* If we got this far, the value is valid. */
    sqlite3_result_int(ctx, 1);
}
```

It just uses `strtol()` to look for errors in the conversion. The actual implementation is arbitrary. You can create whatever criteria you want to validate an integer value. Using this function as a template, you can create user-defined functions for any data type—`float`, `double`, `varchar`, `date`, `timestamp`, whatever you like. They only differ in how they test the input value for validity. This example uses three general data type tests for integer, float, and text values. It then registers them under various SQL function names:

```
/* Type Validation: Called to validate text. */
sqlite3_create_function( db, "validate_text", 1, SQLITE_UTF8, db,
                         validate_text_udf, NULL, NULL);

/* Type Validation: Called to validate integer. */
sqlite3_create_function( db, "validate_int", 1, SQLITE_UTF8, db,
                         validate_int_udf, NULL, NULL);

/* Type Validation: Called to validate long integer. Same UDF as above,
 * different SQL function name. */
sqlite3_create_function( db, "validate_long", 1, SQLITE_UTF8, db,
                         validate_int_udf, NULL, NULL);

/* Type Validation: Called to validate double. */
sqlite3_create_function( db, "validate_double", 1, SQLITE_UTF8, db,
                         validate_double_udf, NULL, NULL);

/* Type Validation: Called to validate float. Same UDF as above,
 * different SQL function name. */
sqlite3_create_function( db, "validate_float", 1, SQLITE_UTF8, db,
                         validate_double_udf, NULL,
```

Both the `validate_int()` and `validate_long()` SQL functions map to the same `validate_int_udf()` callback function. Their validity test is the same, so why write two callback functions to do the same thing? Likewise, the `validate_float()` and `validate_double()` SQL functions call the same `validate_double_udf()` callback function to validate their respective values.

Installation Functions

So far, so good. Before creating a test program, let's take the example a step further. Instead of having to write and maintain our own triggers for each column we want to control, let's write another user-defined function to do it for us. Actually, we need two such functions: one to create the triggers and one to remove them (in case we don't want type checking any more on a given column). The callback function to add the triggers is called `add_strict_type_check_udf()`. Its corresponding SQL function is called `add_strict_type_check()`. The callback function is implemented as shown in Listing 7-6.

Listing 7-6. *The strict_type_check() User-Defined Function*

```
void add_strict_type_check_udf( sqlite3_context* ctx, int nargs,
                                sqlite3_value** values )
{
    sqlite3 *db; sqlite3_stmt *stmt;
    int rc;
    const char *table, *column, *sql, *tmp, *tail, *err;

    db=(sqlite3*)sqlite3_user_data(ctx);

    table  = sqlite3_value_text(values[0]);
    column = sqlite3_value_text(values[1]);

    /* If the column name is an asterisk */
    if(strncmp(column,"*",1) == 0) {

        /* Install type checking on all columns */

        sql = "pragma table_info(%s)";
        tmp = sqlite3_mprintf(sql, table);

        rc = sqlite3_prepare(db, tmp, strlen(tmp), &stmt, &tail);
        sqlite3_free((void*)tmp);

        if(rc != SQLITE_OK) {
            err = sqlite3_errmsg(db);
            sqlite3_result_error(ctx, err, strlen(err));
        }

        rc = sqlite3_step(stmt);

        while(rc == SQLITE_ROW) {
            /* If not primary key */
            if(sqlite3_column_int(stmt, 5) != 1) {
                column = sqlite3_column_text(stmt, 1);
                install_type_trigger(db, ctx, table, column);
            }
            rc = sqlite3_step(stmt);
        }

        sqlite3_finalize(stmt);
    }
    else {

        /* Column name specified, just install on that one column. */
        if(install_type_trigger(db, ctx, table, column) != 0) {
            return;
        }
    }

    sqlite3_result_int(ctx, 0);
}
```

This function installs triggers on either all columns or on a single column specified by `column`. If the value passed in column is an asterisk (*), it installs the triggers on all columns in the table (with the exception of the primary key column), using `PRAGMA table_info()` to obtain column information. Whether one or many columns, the real work is ultimately passed on to a helper function `install_type_trigger()`, which does the trigger installation. It is shown in Listing 7-7.

Listing 7-7. *The Trigger Installation Helper Function*

```
int install_type_trigger( sqlite3* db, sqlite3_context* ctx,
                          const char* table, const char* column )
{
    int rc;
    char buf[256];
    char *tmp;
    const char *type, *sql, *emsg;
    char* err;

    /* Get the column's declared type */
    type = column_type(db, table, column);

    /* Check to see if corresponding validation function exists */

    sql = "select validate_%s(null)";
    tmp = sqlite3_mprintf(sql, type);
    rc = sqlite3_exec(db, tmp, NULL, NULL, &err);
    sqlite3_free(tmp);

    if(rc != SQLITE_OK && err != NULL) {
        emsg = "No validator exists for column type";
        sqlite3_result_error(ctx, emsg, strlen(emsg));
        sqlite3_free((void*)type);
        sqlite3_free(err);
        return 1;
    }

    /* Create INSERT trigger */
    sql = "CREATE TRIGGER %s_insert_%s_typecheck_tr \n"
          "BEFORE INSERT ON %s \n"
          "BEGIN \n"
          "   SELECT CASE \n"
          "     WHEN(SELECT validate_%s(new.%s) != 1) \n"
          "     THEN RAISE(ABORT, 'invalid %s value for %s.%s') \n"
          "   END; \n"
          "END;";

    tmp = sqlite3_mprintf(sql, table, column, table, type,
                          column, type, table, column);

    rc = sqlite3_exec(db, tmp, NULL, NULL, &err);
    sqlite3_free(tmp);

    if(rc != SQLITE_OK && err != NULL) {
        strncpy(&buf[0], err, 255);
        buf[256] = '\0';
        sqlite3_result_error(ctx, &buf[0], strlen(&buf[0]));
        sqlite3_free((void*)type);

        return 1;
    }

    /* Create UPDATE trigger */

    sql = "CREATE TRIGGER %s_update_%s_typecheck_tr \n"
          "BEFORE UPDATE OF %s ON %s \n"
          "FOR EACH ROW BEGIN \n"
          "  SELECT CASE \n"
          "    WHEN(SELECT validate_%s(new.%s) != 1) \n"
          "    THEN RAISE(ABORT, 'invalid %s value for %s.%s') \n"
          "  END; \n"
          "END;";

    tmp = sqlite3_mprintf(sql, table, column, column, table,
                          type, column, type, table, column);

    rc = sqlite3_exec(db, tmp, NULL, NULL, &err);
    sqlite3_free(tmp);
    sqlite3_free((void*)type);
    if(rc != SQLITE_OK && err != NULL) {
        strncpy(&buf[0], err, 255);
        buf[256] = '\0';
        sqlite3_result_error(ctx, &buf[0], strlen(&buf[0]));
        sqlite3_free(err);

        return 1;
    }

    return 0;
}
```

As stated, this function does all the real work. First, it gets the column's declared type through another user-defined function `column_type()`, which in turn just prepares a query with the given column and uses `sqlite3_column_decltype()` to fetch its declared type. With the type, `install_type_trigger()` assumes that all validation functions will follow the naming convention `validate_`*`xxx`*`()`, where *xxx* is the data type to be validated. If, for example, the column's declared type is `int`, `install_type_trigger()` tests for the existence of the respective validation function by executing the query

```
select validate_int(1);
```

Exactly one of two things can happen here: the query will fail or succeed. It can only fail if the function does not exist. It doesn't matter if the argument 1 is of the correct type; even if the `validate_int()` function exists, it will return false (0) and the query will still succeed as the query constitutes as a valid SQL statement. That is, `sqlite3_exec()` will return `SQLITE_OK` if `validate_int()` exists. Otherwise, it will return an error. That is all it needs—a test for existence that answers the question: has a `validate_`*`xxx`*`()` user-defined function been registered? Yes or no? If the validation function exists, `install_type_trigger()` proceeds to build the trigger statements (as illustrated earlier) and runs them.

Likewise, the user-defined function to remove the triggers is called `drop_strict_type_check_udf()`, and works in exactly the same way, except that it executes SQL to drop the triggers. It too uses a helper function to do most of the work.

The Test Program

Putting it all together, strict typing can now be applied to a column using something like

```
select add_strict_type_check('episodes', 'cid');
```

It can be imposed on the entire table using an asterisk for the column argument:

```
select add_strict_type_check('episodes', '*');
```

Similarly, it can be removed with either of the following:

```
select remove_strict_type_check('episodes', 'cid');
select remove_strict_type_check('characters', '*');
```

As a part of the implementation, there is also a handy function with which to get a column's declared type:

```
select column_type('episodes', 'rating') as type;
```

It returns the declared type for the column as defined in the `CREATE TABLE` definition. The example in `strict_typing.c` includes all of this code and presents a working model. The complete test program is implemented in Listing 7-8.

Listing 7-8. *The strict_typing Program*

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db; char *sql;

    rc = sqlite3_open("test.db", &db);

    /* I. Register SQL functions -------------------------------------------*/

    /* Generates and installs validation triggers. */
    rc = sqlite3_create_function( db, "add_strict_type_check", 2, SQLITE_UTF8, db,
                                  add_strict_type_check_udf, NULL, NULL);

    /* Removes validation triggers. */
    sqlite3_create_function( db, "drop_strict_type_check", 2, SQLITE_UTF8, db,
                             drop_strict_type_check_udf, NULL, NULL);

    /* Convenience function for pulling a column's type from sqlite_master.
     * It is a fine-grained 'PRAGMA table_info()'.*/
    sqlite3_create_function( db, "column_type", 2, SQLITE_UTF8, db,
                             column_type_udf, NULL, NULL);

    /* Type Validation: Called to validate text values. */
    sqlite3_create_function( db, "validate_text", 1, SQLITE_UTF8, db,
                             validate_text_udf, NULL, NULL);

    /* Type Validation: Called to validate integer values. */
    sqlite3_create_function( db, "validate_int", 1, SQLITE_UTF8, db,
                             validate_int_udf, NULL, NULL);

    /* Type Validation: Called to validate long integer values. It uses
    ** the same C UDF as above, only a different SQL function name. */
    sqlite3_create_function( db, "validate_long", 1, SQLITE_UTF8, db,
                             validate_int_udf, NULL, NULL);

    /* Type Validation: Called to validate double values. */
    sqlite3_create_function( db, "validate_double", 1, SQLITE_UTF8, db,
                             validate_double_udf, NULL, NULL);
    /* Type Validation: Called to validate float values. It uses the
    ** same as above, and has only a different SQL function name.  */
    sqlite3_create_function( db, "validate_float", 1, SQLITE_UTF8, db,
                             validate_double_udf, NULL, NULL);

/* II. Test SQL functions ---------------------------------------------------*/

    /* Turn on SQL tracing */
    log_sql(db, 1);

    /* Add strict type checks to all columns in table episodes */
    printf("1. Add strict typing:\n");
    execute(db, "select add_strict_type_check('episodes', '*')");
    printf("\n");

    /* Insert a record with valid values */
    printf("2. Insert valid values -- should succeed:\n");
    execute(db, "insert into episodes (name,cid,rating) "
                "values ('Fusilli Jerry', 1, 9.5)");
    printf("\n");

    /* Update with invalid values */
    printf("3. Update with invalid values -- should fail:\n");
    execute(db, "update episodes set cid = 'text'");
    execute(db, "update episodes set rating = 'text'");
    printf("\n");

    /* Remove static type checks for table episodes */
    printf("4. Remove strict type checks on table episodes\n");
    execute(db, "select drop_strict_type_check('episodes', '*')");
    printf("\n");

    /* Insert a record with invalid values */
    printf("5. Update rating with non-float value -- should succeed:\n");
    execute(db, "update episodes set rating = 'Two thumbs up'");
    printf("\n");

    printf("6. Select records, show update results:\n");
    print_sql_result(db, "select * from episodes");
    printf("\n");

    /* Test column_type() function:*/
    printf("7. Test column_type() UDF\n");
    sql = "select column_type('episodes', 'id') as 'id',\n"
          "         column_type('episodes', 'name')  as 'name',\n"
          "         column_type('episodes', 'cid')  as 'cid',\n"
          "         column_type('episodes', 'rating') as 'rating'";
    print_sql_result(db, sql);
    printf("\n");

    sqlite3_close(db);

    return 0;
}
```

Results

This program produces the following output:

```
1. Add strict typing:
  TRACE: select add_strict_type_check('episodes', '*')
  TRACE: pragma table_info(episodes)
  TRACE: select validate_int(null)
  TRACE: CREATE TRIGGER episodes_insert_cid_typecheck_tr
BEFORE INSERT ON episodes
BEGIN
   SELECT CASE
     WHEN(SELECT validate_int(new.cid) != 1)
     THEN RAISE(ABORT, 'invalid int value for episodes.cid')
   END;
END;
  TRACE: CREATE TRIGGER episodes_update_cid_typecheck_tr
BEFORE UPDATE OF cid ON episodes
FOR EACH ROW BEGIN
  SELECT CASE
    WHEN(SELECT validate_int(new.cid) != 1)
    THEN RAISE(ABORT, 'invalid int value for episodes.cid')
  END;
END;
  TRACE: select validate_text(null)
  TRACE: CREATE TRIGGER episodes_insert_name_typecheck_tr
BEFORE INSERT ON episodes
BEGIN
   SELECT CASE
     WHEN(SELECT validate_text(new.name) != 1)
     THEN RAISE(ABORT, 'invalid text value for episodes.name')
   END;
END;
  TRACE: CREATE TRIGGER episodes_update_name_typecheck_tr
BEFORE UPDATE OF name ON episodes
```

```
FOR EACH ROW BEGIN
  SELECT CASE
    WHEN(SELECT validate_text(new.name) != 1)
    THEN RAISE(ABORT, 'invalid text value for episodes.name')
  END;
END;
  TRACE: select validate_float(null)
  TRACE: CREATE TRIGGER episodes_insert_rating_typecheck_tr
BEFORE INSERT ON episodes
BEGIN
   SELECT CASE
     WHEN(SELECT validate_float(new.rating) != 1)
     THEN RAISE(ABORT, 'invalid float value for episodes.rating')
   END;
END;
  TRACE: CREATE TRIGGER episodes_update_rating_typecheck_tr
BEFORE UPDATE OF rating ON episodes
FOR EACH ROW BEGIN
  SELECT CASE
    WHEN(SELECT validate_float(new.rating) != 1)
    THEN RAISE(ABORT, 'invalid float value for episodes.rating')
  END;
END;

2. Insert valid values -- should succeed:
  TRACE: insert into episodes (name,cid,rating) values ('Fusilli Jerry', 1, 9.5)

3. Update with invalid values -- should fail:
  TRACE: update episodes set cid = 'text'
execute() : Error 19 : invalid int value for episodes.cid
  TRACE: update episodes set rating = 'text'
execute() : Error 19 : invalid float value for episodes.rating

4. Remove strict type checks on table episodes
  TRACE: select drop_strict_type_check('episodes', '*')
  TRACE: pragma table_info(episodes)
  TRACE: DROP TRIGGER episodes_insert_cid_typecheck_tr
  TRACE: DROP TRIGGER episodes_update_cid_typecheck_tr
  TRACE: DROP TRIGGER episodes_insert_name_typecheck_tr
  TRACE: DROP TRIGGER episodes_update_name_typecheck_tr
  TRACE: DROP TRIGGER episodes_insert_rating_typecheck_tr
  TRACE: DROP TRIGGER episodes_update_rating_typecheck_tr

5. Update rating with non-float value -- should succeed:
  TRACE: update episodes set rating = 'Two thumbs up'
```

```
6. Select records, show update results:
  TRACE: select * from episodes
id cid name             rating
-- --- ---------------- -------------
1  1   Chocolate Babka  Two thumbs up
2  2   Mackinaw Peaches Two thumbs up
3  3   Soup Nazi        Two thumbs up
4  4   Pig Man          Two thumbs up
5  1   Fusilli Jerry    Two thumbs up

7. Test column_type() UDF
  TRACE: select column_type('episodes', 'id') as 'id',
         column_type('episodes', 'name')  as 'name',
         column_type('episodes', 'cid')  as 'cid',
         column_type('episodes', 'rating') as 'rating'
id      name cid rating
------- ---- --- ------
integer text int float
```

Step 1 adds static typing constraints on the table, at which point we see a barrage of CREATE TRIGGER statements in which there is one INSERT and one UPDATE trigger for each column. Step 2 simply inserts a valid record. The absence of an error confirms that this operation completed successfully. Step 3 attempts to update with invalid values by providing text values for the integer cid and float rating columns in two consecutive updates. It is rebuffed both times. Steps 2 and 3 indicate that both of the triggers (INSERT and UPDATE) and their respective validation functions (float, text, and int) are working as intended. Step 4 removes the strict typing constraints on all columns, whereupon we see a parade of DROP TRIGGER statements. Step 5 then attempts what step 3 couldn't manage—sticking text values in a non-text column—and succeeds. Step 6 shows off step 5's handiwork. And finally, step 7 demonstrates the utility function column_type() for good measure.

To sum up, there are certainly other ways to go about implementing strict typing. This example is simplistic at best. But it does illustrate the mileage you can get out of SQLite's user-defined functions. At first glance, I never suspected that I could do something like strict typing until I sat down and started working with the version 3 API. Better yet, I've seen a number of clever ideas on various blogs and on the SQLite mailing list on how to use similar techniques and other useful features.

Aggregates

After the function workout, you should be pretty well versed on the extension process in general. Aggregates and collation sequences work so similarly to functions that you will have no trouble with the remaining parts of this chapter. You are nearing the end of the C API.

Aggregates are only slightly more involved than functions. Whereas you only had to implement one callback function to do the work in user-defined functions, you have to implement both a step function to compute the ongoing value as well as a finalizing function to finish everything off and clean up. The general process is shown in Figure 7-1. It still isn't hard, though. Like functions, one good example will be all you need to get the gist of it.

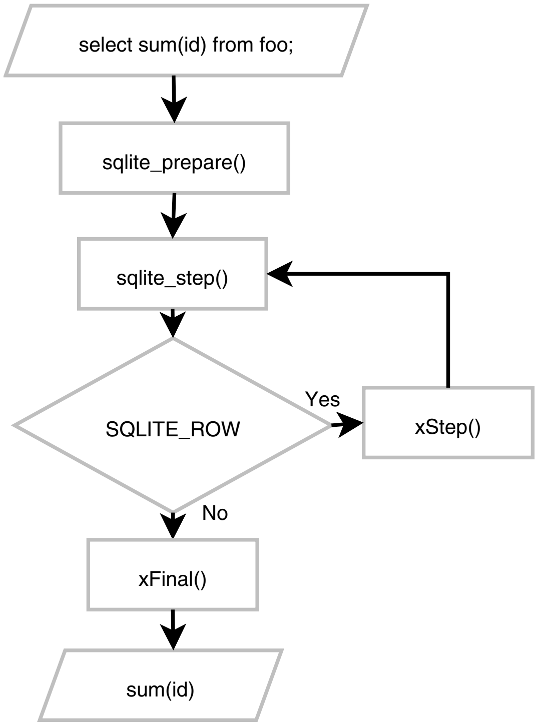

Figure 7-1. *Query processing with aggregates*

Registration Function

As mentioned before, aggregates and functions use the same registration function, `sqlite3_create_function()`, which is listed here again for convenience:

```
int sqlite3_create_function(
  sqlite3 cnx*,               /* connection handle                      */
  const char *zFunctionName,  /* function/aggregate name in SQL         */
  int nArg,                   /* number of arguments. -1 = unlimited. */
  int eTextRep,               /* encoding (UTF8, 16, etc.)              */
  void *pUserData,            /* application data, passed to callback */
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

You do exactly the same thing here as you do with functions, but rather than proving a callback for `xFunc`, you leave it `NULL` and provide functions for `xStep` and `xFinal`.

A Practical Example

The best way to illustrate implementing aggregates is by example. Here we will implement the aggregate SUM(). Yes, SQLite already has this aggregate built in, but SUM() is simple and to the point. To avoid collision, ours will be called sum_int(). Also, this example is a little watered down as it sums only integers. But there is little value in getting caught up in *what* we are aggregating—you can do that later. The goal here is *how* to aggregate. The example program looks almost identical to that of the func.c example, and is shown in Listing 7-9 (taken from aggregate.c).

Listing 7-9. *The sum_int() Test Program*

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db; char *sql;

    sqlite3_open("test.db", &db);

    /* Register aggregate. */
    fprintf(stdout, "Registering aggregate sum_int()\n");

    /* Turn SQL tracing on. */
    log_sql(db, 1);

    /* Register aggregate. */
    sqlite3_create_function( db, "sum_int", 1, SQLITE_UTF8, db,
                             NULL, step, finalize);

    /* Test. */
    fprintf(stdout, "\nRunning query: \n");
    sql = "select sum_int(id) from aggregate";
    print_sql_result(db, sql);

    sqlite3_close(db);

    return 0;
}
```

Things to note here are that our step function is called step(), and our finalizing function is called finalize(). Also, this aggregate is registered as taking only one argument.

The Step Function

The step() function is shown in Listing 7-10.

Listing 7-10. *The sum_int() Step Function*

```
void step(sqlite3_context* ctx, int ncols, sqlite3_value** values)
{
    sum* s;
    int x;

    s = (sum*)sqlite3_aggregate_context(ctx, sizeof(sum));

    if(sqlite3_aggregate_count(ctx) == 1) {
        s->x = 0;
    }

    x = sqlite3_value_int(values[0]);
    s->x += x;

    fprintf(stdout, "step()     : value=%i, total=%i\n", x, s->x);
}
```

The value `sum` is a struct that is specific to this example and is defined as follows:

```
typedef struct {
    int x;
} sum;
```

It's just a glorified integer. This structure serves as our state between aggregate iterations (calls to the step function).

The Aggregate Context

The first order of business is to retrieve the structure for the given aggregate instance. This is done using `sqlite3_aggregate_context()`. As mentioned earlier, the first call to this function allocates the data for the given context and subsequent calls retrieve it. The structure memory is automatically freed when the aggregate completes (after `finalize()` is called). In the `step()` function, the value to be aggregated is retrieved from the first argument using `sqlite3_value_int()`. This value is then added to the `x` member of the `sum` struct, which stores the intermediate summation.

The Finalize Function

Each record in the materialized result set triggers a call to `step()`. After the last record is processed, SQLite calls `finalize()`, which is implemented as shown in Listing 7-11.

Listing 7-11. *The sum_int() Finalize Function*

```
void finalize(sqlite3_context* ctx)
{
    sum* s;
    s = (sum*)sqlite3_aggregate_context(ctx, sizeof(sum));
    sqlite3_result_int(ctx, s->x);
    fprintf(stdout, "finalize() : total=%i\n\n", s->x);
}
```

This function retrieves the `sum` struct and sets the aggregate's return value to the value stored in the struct's `x` member using `sqlite3_result_int()`. The `finalize()` function basically works just like a user-defined function callback.

Results

The program produces the following output:

```
Dropping table aggregate, if exists.
  TRACE: select count(*) from sqlite_master where name='aggregate'
  TRACE: drop table aggregate
Creating table aggregate.
  TRACE: create table aggregate(id integer)
Populating table aggregate.
  TRACE: insert into aggregate values (1)
  TRACE: insert into aggregate values (2)
  TRACE: insert into aggregate values (3)

Registering aggregate sum_int()

Running query:
  TRACE: select sum_int(id) from aggregate
step()     : value=1, total=1
step()     : value=2, total=3
step()     : value=3, total=6
finalize() : total=6

sum_int(id)
-----------
6
```

I included some output from the setup code in this example (which creates and populates the aggregate table) to better illustrate how this example works. As you can see, the `SELECT` statement pulls back a total of three records, which results in three calls to `step()`. After the last step function comes `finalize()`, which provides the value returned by `sum_int()`. In this case the value is 6.

And that is the concept of aggregates in a nutshell. They are so similar to functions that there isn't much to talk about once you are familiar with functions.

Collations

As stated before, the purpose of Collations is to sort strings. But before starting down that path, it is important to recall that SQLite's manifest typing scheme allows varying data types to coexist in the same column. For example, consider the following SQL (located in `collate1.sql`):

```
.headers on
.m col
create table foo(x);
insert into foo values (1);
insert into foo values (2.71828);
insert into foo values ('three');
insert into foo values (X'0004');
insert into foo values (null);

select quote(x), typeof(x) from foo;
```

Feeding it to the SQLite CLP will produce the following:

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)    typeof(x)
----------  ----------
1           integer
2.71828     real
'three'     text
X'0004'     blob
NULL        null
```

You've got every one of SQLite's native storage classes sitting in column `x`. Naturally, the question arises as to what happens when you try to sort this column. That is, before we can talk about how to sort strings, we need to review how different data types are sorted first.

When SQLite sorts a column in a result set (when it uses a comparison operator like `<` or `>=` on the values within that column), the first thing it does is arrange the column's values according to storage class. Then within each storage class, it sorts the values according to methods specific to that class. Storages classes are sorted in the following order, from first to last:

1. `NULL` values
2. `INTEGER` and `REAL` values
3. `TEXT` values
4. `BLOB` values

Now let's modify the `SELECT` statement in the preceding example to include an `ORDER BY` clause such that the SQL is as follows:

```
select quote(x), typeof(x) from foo order by x;
```

Rerunning the query (located in `collate2.sql`) confirms this ordering:

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)    typeof(x)
----------  ----------
NULL        null
1           integer
2.71828     real
'three'     text
X'0004'     blob
```

`NULL`s are first, `INTEGER` and `REAL` next (in numerical order), followed by `TEXT`, and then `BLOB`s last.

SQLite employs specific sorting methods for each storage class. `NULL`s are obvious—there is no sort order. Numeric values are sorted numerically—integers and floats alike are compared based on their respective quantitative values. `BLOB`s are sorted by binary value. Text, finally, is where Collations come in.

Collation Defined

Collation is the method by which strings are compared. According to Wikipedia,[1] collation is defined as

> *In library and information science and computer science, collation is the assembly of written information into a standard order. In common usage, this is called alphabetization, though collation is not limited to ordering letters of the alphabet. Collating lists of words or names into alphabetical order is the basis of most office filing systems, library catalogues, and books of reference.*

According to IBM's glossary of Unicode terms, collation is defined as

> *Text comparison using language-sensitive rules as opposed to bitwise comparison of numeric character codes.*

In general, collation has to do with the arranging of strings and the arranging of characters. Collation methods usually employ *collation sequences*. A collation sequence is just a list of characters, each of which is assigned some numerical value to denote its position in the list. The list in turn is used to dictate how characters are ordered. Given any two characters, the Collation (list) can resolve which would come first, or if they are the same.

How Collation Works

Normally, a collation method compares two strings by breaking them down and comparing their respective characters using a Collation. They do this by lining the strings up and comparing characters from left to right (Figure 7-2). For example, the strings `'jerry'` and `'jello'` would be compared by first comparing the first letter in each string (both *j* in this case). If one letter's

1. See `http://en.wikipedia.org/wiki/Collation`.

numerical value according to the collation sequence is larger than the other, the comparison is over: the string with the larger letter is deemed greater and no further comparison is required. If, on the other hand, the letters have the same value, then the process continues on to the second letter in each string, and to the third, and so on until some difference is found (or no difference is found, in which case the strings are considered equal). If one string runs out of letters before the other, then it is up to the collation method to determine what this means.

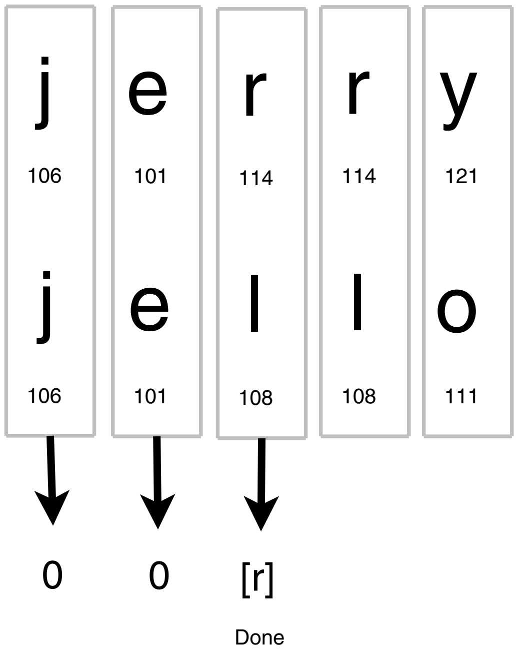

Figure 7-2. *Binary collation*

In this example, the Collation is the ASCII character set. Each character's numeric ASCII value in the figure is displayed underneath it. Using the collation method just described, string comparison continues to the third character, whereupon `'jerry'` wins out, as it has an `r` with the value 114 whereas `'jello'` has the inferior `'l'` with the value 108. According to this collation method using the ASCII Collation, `'jerry'` > `'jello'`.

Had `'jerry'` been `'Jerry'`, then `'jello'` would have won out, as big `J`'s little 74 pales to little `j`'s big 106, so the comparison would have been resolved on the first character—the prize going to `'jello'`. In short, `'jello'` > `'Jerry'`.

However, to continue the example, we could use a different Collation wherein uppercase values are assigned the same numbers as their lowercase counterparts (a case-insensitive collation sequence). Then the comparison between `'Jerry'` and `'jello'` would go back to `'Jerry'` as in this new Collation `J` and `j` are the same, pulling the rug out from under `'jello'`'s lowercase `j` superiority.

But enough about Collations. You get the picture. A sequence is a sequence. Ultimately, it's not the collation sequence that matters, but the collation method. Collation methods don't have to use sequences. They can do whatever they want. A collation method is called a method for a reason; it can do whatever it wants, even if its chosen method may be madness.

Odds are that you are really confused by now and are just ready for a summary. So here it is. Think of collation methods as the way that strings are compared. Think of collation sequences as the way characters are compared. Ultimately, all that matters in SQLite is how strings are compared, and the two terms refer to just that. Therefore, the two terms *collation method* and *collation sequence* should just be thought of as string comparison. If you read *collation*, think string comparison. If you read *Collation*, think string comparison. All in all, just think string comparison. That is the way the API is geared. When you create a custom Collation, SQLite hands it two strings, *a* and *b*, and the Collation returns whether string *a* is less than, equal to, or greater than string *b*. That's it, plain and simple.

Standard Collation Types

SQLite comes with a single built-in Collation named `BINARY`, which is implemented applying the `memcmp()` routine from the standard C library to the two strings to be compared. The `BINARY` collation happens to work well for English text. For other languages or locales, alternate Collation may be preferred. SQLite also offers an alternate collation called `NOCASE`, which does not distinguish between cases, and another called `REVERSE`, which as you may have guessed compares everything backwards (according to binary value).

Collations are applied by associating them with columns in a table or an index definition, or by specifying them in a query. For example, to create a case-insensitive column `bar` in `foo`, you would define `foo` as follows:

```
create table foo (bar text COLLATE NOCASE, baz text);
```

To create a `REVERSE` index on `baz`, you would do something like this:

```
create index bar_rev_idx on foo (baz) COLLATE REVERSE;
```

From that point on, whenever SQLite deals with a `bar` it will use the `NOCASE` collation. When it uses an index for `baz`, it will follow the `REVERSE` collation.

If you prefer not to attach a Collation to a database object, but would rather specify it as needed on a query-by-query basis, you can specify them directly in queries, as in the following:

```
select * from foo order by bar COLLATE NOCASE;
```

At the time of this writing, there are plans under way to include standard `CAST()` syntax to allow the collation of an expression to be defined.

A Simple Example

To jump into collations, let's begin with a simple example to get the big picture. Let's implement a collation called `POLITICAL`. `POLITICAL` will take two strings and decide which one is greater depending on the day. One Mondays, `'Jello'` might be greater than `'jerry'`. On Tuesdays, it may be a different story entirely.

Collations are registered with SQLite through the `sqlite3_create_collation()` function, which is declared as follows:

```
int sqlite3_create_collation(
  sqlite3* db,          /* database handle */
  const char *zName,    /* collation name in SQL */
  int pref16,           /* encoding         */
  void* pUserData,      /* application data */
  int(*xCompare)(void*,int,const void*,int,const void*)
);
```

It looks pretty similar to `sqlite3_create_function()`. There is the standard database handle, the collation name as it will be addressed in SQL (similar to the function/aggregate name), and the encoding, which pertains to the format of the comparison strings when passed to the comparison function.

■ **Note** Just as in aggregates and functions, separate collation comparison functions can be registered under the same collation name for each of the UTF-8, UTF-16LE, and UTF-16BE encodings. SQLite will automatically select the best comparison function for the given collation based on the strings to be compared.

Finally, the application data pointer is another API mainstay. The pointer provided there is passed on to the callback function. Basically, `sqlite3_create_collation()` is a stripped-down version of `sqlite3_create_function()`, which takes exactly two arguments (both of which are text) and returns an integer value. It also has the aggregate callback function pointers stripped out.

The Compare Function

The `xCompare` argument is really the only new thing here. It points to the actual comparison function that will resolve the text values. The comparison function must have the form

```
int compare( void* data,        /* application data */
             int len1,          /* length of string 1 */
             const void* str1,  /* string 1 */
             int len2,          /* length of string 2 */
             const void* str2)  /* string 2 */
```

The function should return negative, zero, or positive if the first string is less than, equal to, or greater than the second string, respectively.

As stated, the political collation is a bit wishy-washy. Its implementation is shown in Listing 7-12.

Listing 7-12. *The Political Collation Function*

```
int political_collation( void* data, int l1, const void* s1,
                                     int l2, const void* s2 )
{
    int result, opinion; struct tm* t; time_t rt;
    /* Get the unpolitical comparison result */
    result = strcmp(s1,s2);

    /* Get the date and time */
    time(&rt);
    t = localtime(&rt);

    /* Form an opinion: is s1 really < or = to s2 ? */
    switch(t->tm_wday) {
        case 0: /* Monday yes    */
            opinion = result;
            break;
        case 1: /* Tuesday no     */
            opinion = -result;
            break;
        case 2: /* Wednesday bigger is better */
            opinion = l1 >= l2 ? -1:1;
            break;
        case 3: /* Thursday strongly no   */
            opinion = -100;
            break;
        case 4: /* Friday strongly yes    */
            opinion = 100;
            break;
        case 5: /* Saturday golf, everything's the same */
            opinion = 0;
            break;
        default: /* Sunday - Meet the Press, opinion changes
                    every minute */
            opinion = 2*sin(t->tm_min*180);
    }

    /* Now change it on a whim */
    opinion = rand()-(RAND_MAX/2) > 0 ? -1:1;

    return opinion;
}
```

The Test Program

All that remains is to illustrate it in a program. The example program (`political.c`) implementation is shown in Listing 7-13.

Listing 7-13. *The Political Collation Test Program*

```
int main(int argc, char **argv)
{
    char *sql; sqlite3 *db; int rc;

    /* For forming more consistent political opinions. */
    srand((unsigned)time(NULL));

    sqlite3_open("test.db", &db);

    /* Create issues table, add records. */
    setup(db);

    /* Register Collation. */
    fprintf(stdout, "1. Register political Collation\n\n");
    sqlite3_create_collation( db, "POLITICAL",
                              SQLITE_UTF8, db,
                              political_collation );

    /* Turn SQL logging on. */
    log_sql(db, 1);

    /* Test default collation. */
    fprintf(stdout, "2. Select records using default collation.\n");
    sql = "select * from issues order by issue";
    print_sql_result(db, sql);

    /* Test Political collation. */
    fprintf(stdout, "\nSelect records using political collation. \n");
    sql = "select * from issues order by issue collate POLITICAL";
    print_sql_result(db, sql);

    fprintf(stdout, "\nSelect again using political collation. \n");
    print_sql_result(db, sql);

    /* Done. */
    sqlite3_close(db);

    return 0;
}
```

Results

Running the program yields the following results:

```
1. Register political Collation

2. Select records using default collation.
  TRACE: select * from issues order by issue
issue
-----------------
Defense
Deficit
Economy
Education
Environment
Foreign Policy
Health Care
National Security
Social Security
Unemployment

Select records using political collation.
  TRACE: select * from issues order by issue collate POLITICAL
issue
-----------------
Health Care
Foreign Policy
National Security
Economy
Social Security
Unemployment
Environment
Defense
Education
Deficit

Select again using political collation.
  TRACE: select * from issues order by issue collate POLITICAL
issue
-----------------
National Security
Defense
Unemployment
Deficit
Foreign Policy
Education
Social Security
```

```
Health Care
Environment
Economy
```

Step 1 registers the political collation sequence. Step 2 selects records using SQLite's default binary collation, which returns records in alphabetical order. Step 3 selects records twice using the political Collation, obtaining different results each time. As the results indicate, we are right on the mark. In fact, the more times you select using the `POLITICAL` collation, the more you are apt to find that the results continually change. Politics can be very complicated.

Collation on Demand

SQLite provides a way to defer collation registration until it is actually needed. So if you are not sure your application is going to need something like the `POLITICAL` collation, you can use the `sqlite3_collation_needed()` function to defer its registration to the last possible moment (which incidentally seems befitting of this particular collation). You simply provide `sqlite3_collation_needed()` with a callback function, which SQLite can rely on to register unknown collation sequences as needed, given their name. It is a callback for a callback, so to speak. Better yet, it's like saying to SQLite, "Here, if you are ever asked to use a Collation that you don't recognize, call this function, and it will register the unknown sequence, then you can continue your work."

The `sqlite3_collation_needed()` function is declared as follows:

```
int sqlite3_collation_needed(
  sqlite3* db,    /* connection handle  */
  void* data,     /* application data   */
  void(*crf)(void*,sqlite3*,int eTextRep,const char*)
);
```

The `crf` argument (the collation registration function as I call it) points to the function that will register the unknown Collation. For clarity, it is defined as follows:

```
 void crf( void* data,    /* application data */
           sqlite3* db,   /* database handle */
           int eTextRep,  /* encoding */
           const char*)   /* collation name */
```

The `db` and `data` arguments of `crf()` are the values passed into the first and second arguments of `sqlite3_collation_needed()`, respectively. To make it all clear, say we want to defer all collation registration. A possible implementation that anticipates the `POLITCAL` Collation and uses some default binary comparison function for all other unknown Collations would be implemented as shown in Listing 7-14.

Listing 7-14. *Collation Registration Function*

```
void crf( void* data, sqlite3* db,
          int eTextRep, const char* cname)
{
    if(strcmp(collation_name, "POLITICAL") == 0) {
        /* Political collation has not been registered and is now needed */
        sqlite3_create_collation( db, "POLITICAL",
                                  SQLITE_UTF8, db,
                                  political_collation );
    } else {
        /* Punt: Use some default comparison function this collation. */
        sqlite3_create_collation( db, collation_name,
                                  SQLITE_UTF8, db,
                                   default_collation );
    }
}
```

Thus `POLITCAL` will get registered only if it is ever referenced. Likewise, if SQLite encounters something like

```
select * from issues order by issue COLLATE EMOTIONAL;
```

our `crf()` handler will just install a generic comparison function `default_collation()` (which I just made up) as the comparison function for `EMOTIONAL` collations. Bottom line: the `sqlite3_collation_needed()` handler's job is to take a Collation name and register a Collation to associate with it.

A Practical Application

We will conclude this chapter with an application of Collations that may prove a little more useful than the previous example (if not more consistent). We will combine what we've learned from functions to create a collation that helps out with date formats. Now, it turns out that ISO dates by virtue of their format happen to sort both lexicographically and chronologically. That is, if you sort a set of ISO dates alphabetically, they will end up in the correct chronological order. As a result, ISO dates tend to work just fine in SQLite.

Let's say you that have some data that you want to pass back and forth between SQLite and an Oracle database. And in your data you have to deal with Oracle style dates. Oracle dates, of the form `DD-MONTH-YY`, don't simultaneously sort lexicographically and chronologically. It's one or the other, but not both.

And while it's possible to get Oracle to use ISO or ANSI dates, rather than bumbling with converting back and forth, you would rather just have SQLite be able to sort Oracle dates as well as it does ISO dates. This is a perfect job for a user-defined Collation.

The Comparison Function

Rather than using a string compare, we parse the date format, breaking it down into its individual parts, and sort them numerically. The comparison function is simple, as you can see in Listing 7-15.

Listing 7-15. *Oracle Date Collation Function*

```
int oracle_date_collation( void* data,
                           int len1, const void* arg1,
                           int len2, const void* arg2 )
{
    int len;
    date d1;
    date d2;

    strncpy(&zDate2[0], arg2, len);
    zDate2[len] = '\0';

    /* Convert dates to date struct */
    oracle_date_str_to_struct(arg1, &d1);
    oracle_date_str_to_struct(arg2, &d2);

    fprintf(stdout, "collate_fn() : date1=%s, date2=%s\n", zDate1, zDate2);

    /* Compare structs */

    if(d1.year < d2.year) {
        return -1;
    }
    else if(d1.year > d2.year) {
        return 1;
    }

    /* If this far, years are equal. */

    if(d1.month < d2.month) {
        return -1;
    }
    else if(d1.month > d2.month) {
        return 1;
    }

    /* If this far, months are equal. */

    if(d1.day < d2.day) {
        return -1;
    }
    else if(d1.day > d2.day) {
        return 1;
    }
    /* If this far, dates are equal. */

    return 0;
}
```

Date Parsing

This function uses another function, called `oracle_date_str_to_struct()`, to parse the dates and populate a generic date struct. This function is implemented as shown in Listing 7-16.

Listing 7-16. *The Oracle Date Parsing Function*

```
int oracle_date_str_to_struct(const char* value, date* d)
{
    const char* date, *tmp;
    char *start, *end, zDay[3], zMonth[4], zYear[3];

    date = get_date(value);

    if(date == NULL) {
        fprintf(stderr, "Invalid date\n");
        return -1;
    }

    /* Find first '-' */
    start = strchr(date,'-');

    /* Find last '-' */
    end   = strchr(start+1,'-');

    /* Extract day part, convert to int*/
    strncpy(zDay, date,2);
    zDay[2] = '\0';
    d->day = atoi(zDay);

    /* Extract month part, convert to int*/
    strncpy(zMonth, start+1,3);
    zMonth[3] = 0;
    tmp = uppercase(zMonth);
    d->month = month_num(tmp);
    free((void*)tmp);

    /* Extract year part, convert to int*/
    strncpy(zYear, end+1,2);
    zYear[2] = '\0';
    d->year = atoi(zYear);
    free((void*)date);

    return 0;
}
```

This function uses another function, `get_date()`, to extract the minimum part of the string, which makes up a complete Oracle date format. It uses Perl Compatible Regular Expressions to do this. Using a regular expression to define the date format makes it easier to tweak without having to do any additional coding. Using `get_date()` to extract a date value from a string along with a flexible regex allows us some flexibility to deal with sloppy dates, if needed. The `get_date()` function is defined in Listing 7-17.

Listing 7-17. *The get_date() Function*

```
#define ORACLE_DATE_REGEX "[0-9]{1,2}-[a-zA-Z]{3,3}-[0-9]{2,2}";

const char* get_date(const char* value)
{
    pcre *re;
    const char *error, *pattern;
    int erroffset;
    int ovector[3];
    int value_length;
    int rc, substring_length;
    char* result, *substring_start;

    pattern = ORACLE_DATE_REGEX;

    re = pcre_compile(
        pattern,              /* the pattern */
        0,                    /* default options */
        &error,               /* for error message */
        &erroffset,           /* for error offset */
        NULL);                /* use default character tables */

    /* Compilation failed */
    if (re == NULL) {
        return NULL;
    }

    value_length = (int)strlen(value);

    rc = pcre_exec(
        re,           /* the compiled pattern */
        NULL,         /* no extra data - we didn't study the pattern */
        value,        /* the value string */
        value_length, /* the length of the value */
        0,            /* start at offset 0 in the value */
        0,            /* default options */
        ovector,      /* output vector for substring information */
        3);           /* number of elements in the output vector */

    if (rc < 0) {
        /* Match error */
        return NULL;
    }

    substring_start = (char*)value + ovector[0];
    substring_length = ovector[1] - ovector[0];
    result = malloc(substring_length);
    strncpy(result, substring_start, substring_length);
    result[substring_length] = '\0';

    return result;
}
```

The Test Program

All three of the above functions work together to collate Oracle dates in chronological order. Our example program is shown in Listing 7-18.

Listing 7-18. *The Oracle Collation Test Program*

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    char *sql;

    sqlite3_open("test.db", &db);

    /* Install oracle related date functions. */
    install_date_functions(db);

    /* Register Collation. */
    fprintf(stdout, "Registering collation sequence oracle_date\n");
    sqlite3_create_collation( db, "oracle_date",
                              SQLITE_UTF8, db,
                              oracle_date_collation );

    /* Create dates table, add records. */
    setup(db);

    /* Turn SQL logging on. */
    log_sql(db, 1);
    /* Test default collation. */
    fprintf(stdout, "Select records. Use default collation.\n");
    sql = "select * from dates order by date";
    print_sql_result(db, sql);

    /* Test Oracle collation. */
    fprintf(stdout, "\nSelect records. Use Oracle data collation. \n");
    sql = "select * from dates order by date collate oracle_date";
    print_sql_result(db, sql);

    /* Done. */
    sqlite3_close(db);

    return 0;
}
```

Results

Running the program yields the following output:

```
Registering collation sequence oracle_date
Select records. Use default collation.
  TRACE: select * from dates order by date
id date
-- ----------
4  1-APR-05
8  1-AUG-05
12 1-DEC-05
2  1-FEB-05
1  1-JAN-05
7  1-JUL-05
6  1-JUN-05
3  1-MAR-05
5  1-MAY-05
11 1-NOV-05
10 1-OCT-05
9  1-SEP-05

Select records. Use Oracle data collation.
  TRACE: select * from dates order by date collate oracle_date
collate_fn() : date1=1-DEC-05, date2=1-NOV-05
collate_fn() : date1=1-OCT-05, date2=1-SEP-05
collate_fn() : date1=1-NOV-05, date2=1-SEP-05
collate_fn() : date1=1-NOV-05, date2=1-OCT-05
collate_fn() : date1=1-AUG-05, date2=1-JUL-05
collate_fn() : date1=1-JUN-05, date2=1-MAY-05
collate_fn() : date1=1-JUL-05, date2=1-MAY-05
collate_fn() : date1=1-JUL-05, date2=1-JUN-05
collate_fn() : date1=1-SEP-05, date2=1-MAY-05
collate_fn() : date1=1-SEP-05, date2=1-JUN-05
collate_fn() : date1=1-SEP-05, date2=1-JUL-05
collate_fn() : date1=1-SEP-05, date2=1-AUG-05
collate_fn() : date1=1-APR-05, date2=1-MAR-05
collate_fn() : date1=1-FEB-05, date2=1-JAN-05
collate_fn() : date1=1-MAR-05, date2=1-JAN-05
collate_fn() : date1=1-MAR-05, date2=1-FEB-05
collate_fn() : date1=1-MAY-05, date2=1-JAN-05
collate_fn() : date1=1-MAY-05, date2=1-FEB-05
collate_fn() : date1=1-MAY-05, date2=1-MAR-05
collate_fn() : date1=1-MAY-05, date2=1-APR-05
id date
-- ----------
1  1-JAN-05
2  1-FEB-05
3  1-MAR-05
4  1-APR-05
5  1-MAY-05
6  1-JUN-05
7  1-JUL-05
8  1-AUG-05
9  1-SEP-05
10 1-OCT-05
11 1-NOV-05
12 1-DEC-05
```

So there you have it. If I were really serious about all this, I would implement strict type checking for columns that had Oracle-style dates. I would implement a `validate_oradate()` SQL function that calls `get_date()` to determine if the date is legitimate. I would then declare all related columns in my tables as `oradate`. Included in their declaration would be the `oracle_date` collation. For example:

```
create table log ( id autoincrement,
                   date oradate COLLATE oracle_date CHECK(get_date(date)),
                   entry text );
```

Notice that I use a `CHECK` constraint for strict typing in this case. It works just as easily (if not more easily) than the trigger functions used earlier for checking a single column (while the trigger approach can be easier for checking all columns in an entire table). Working together, these user-defined functions ensure that the `oracle_date` collation is always presented with properly formatted dates to compare.

Summary

As I hope some of the examples in this chapter demonstrate, user-defined functions, aggregates, and Collations can be surprisingly useful. And while it certainly helps that SQLite is an open source library, meaning that you are free to dig in and modify it to your heart's content, the extensions part of the C API goes a long way in providing a friendly, easy-to-user interface that makes it possible to implement a wide range of powerful extensions and customizations, especially when combined with other features already present in SQLite.

In the next chapter, you will see how many extension languages take advantage of this. They use the C API to make it possible to implement user-defined functions, aggregates, and collations in different languages.