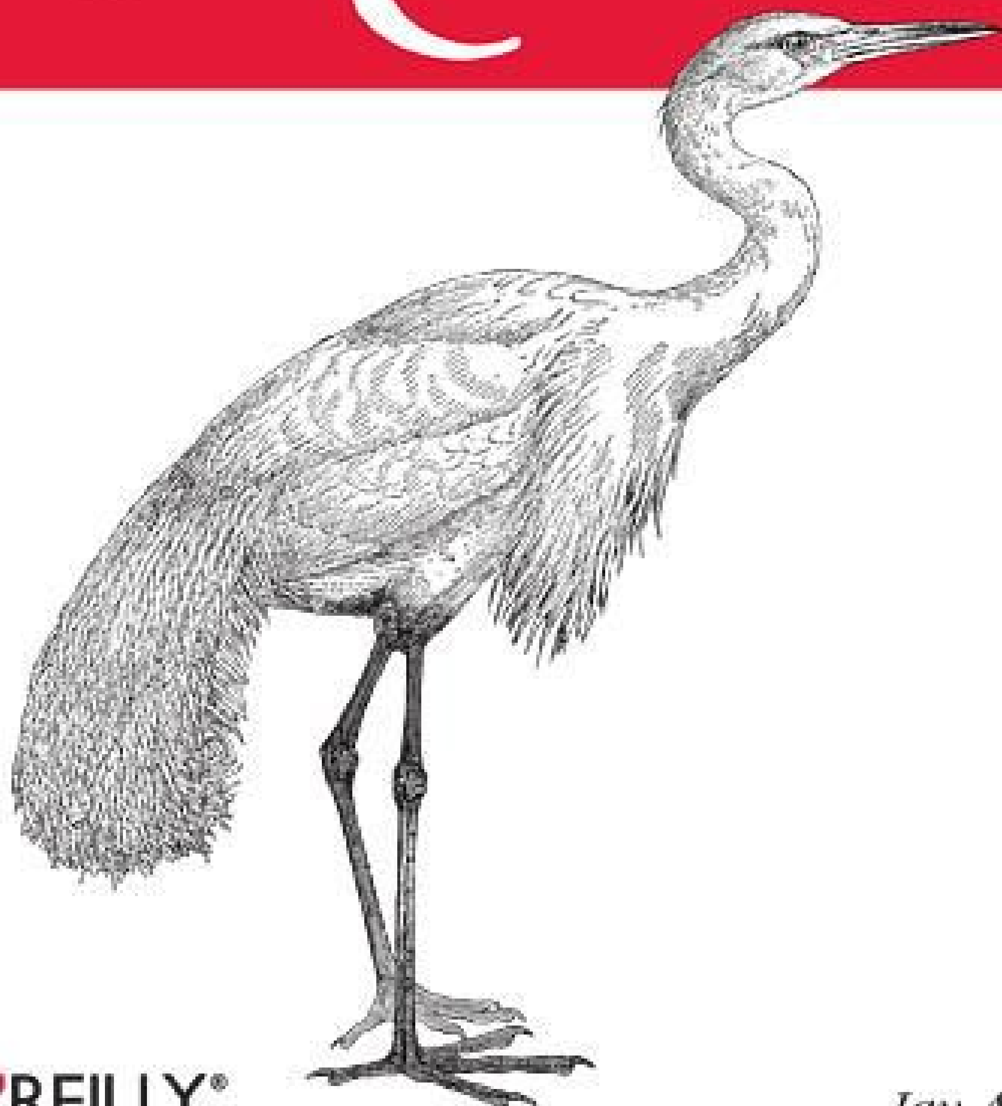
Using

SQLite



O'REILLY®

Jay A. Kreibich

Table of Contents

Chapter 9. SQL Functions and Extensions **1**
Section 9.1. Scalar Functions 2
Section 9.2. Aggregate Functions 14
Section 9.3. Collation Functions 20
Section 9.4. SQLite Extensions 24

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 9

SQL Functions and Extensions

SQLite allows a developer to expand the SQL environment by creating custom SQL functions. Although these functions are used in SQL statements, the code to implement the function is written in C.

SQLite supports three kinds of custom functions. Simple *scalar functions* are the first type. These take some set of parameters and return a single value. An example would be the built-in function `abs()`, which takes a single numeric parameter and returns the absolute value of that number.

The second type of function is an *aggregate function*, or *aggregator*. These are SQL functions, such as `sum()` or `avg()`, that are used in conjunction with `GROUP BY` clauses to summarize or otherwise aggregate a series of values together into a final result.

The last type of custom function is a *collation*. Collations are used to define a custom sort orders for an index or an `ORDER BY` clause. Conceptually, collation functions are quite simple: they take two text values and return a greater than, less than, or equal status. In practice, collations can become quite complex, especially when dealing with natural language strings.

This chapter will also take a look at how to package up a set of custom features into an SQLite *extension*. Extensions are a standard way to package custom functions, aggregations, collations, virtual tables (see Chapter 10), or any other custom feature. Extensions are a handy and standardized way to bundle up sets of related functions or customizations into SQL function libraries.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Extensions can be statically linked into an application, or they can be built into *loadable extensions*. Loadable extensions act as "plug-ins" for the SQLite library. Loadable extensions are a particularly useful way to load your custom functions into `sqlite3`, providing the ability to test queries or debug problems in the same SQL environment that is found in your application.

The source code to the examples found in this chapter can be found in the book download. Downloading the source will make it much easier to build the examples and try them out. See "Example Code Download" on page xvi for more information on where to find the source code.

Scalar Functions

Licensed by
Jere Sandidge
1113004

The structure and purpose of SQL scalar functions are similar to C functions or traditional mathematical functions. The caller provides a series of function parameters and the function computes and returns a value. Sometimes these functions are purely functional (in the mathematical sense), in that they compute the result based purely off the parameters with no outside influences. In other cases, the functions are more procedural in nature, and are called to invoke specific side effects.

The body of a function can do pretty much anything you want, including calling out to other libraries. For example, you could write a function that allows SQLite to send email or query the status of a web server all through SQL functions. Your code can also interact with the database and run its own queries.

Although scalar functions can take multiple parameters, they can only return a single value, such as an integer or a string. Functions cannot return rows (a series of values), nor can they return a result set, with rows and columns.

Scalar functions can still be used to process sets of data, however. Consider this SQL statement:

```
SELECT format( name ) FROM employees;
```

In this query, the scalar function `format()` is applied to every row in the result set. This is done by calling the scalar function over and over for each row, as each row is computed. Even though the `format()` function is only referenced once in this SQL statement, when the query is executed, it can result in many different invocations of the function, allowing it to process each value from the name column.

Registering Functions

To create a custom SQL function, you must bind an SQL function name to a C function pointer. The C function acts as a callback. Any time the SQL engine needs to invoke the named SQL function, the registered C function pointer is called. This provides a way for an SQL statement to call a C function you have written.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

These functions allow you to create and bind an SQL function name to a C function pointer:

```
int sqlite3_create_function(   sqlite3 *db, const char *func_name,
                               int num_param, int text_rep, void *udp,
                               func_ptr, step_func, final_func )
int sqlite3_create_function16( sqlite3 *db, const void *func_name,
                               int num_param, int text_rep, void *udp,
                               func_ptr, step_func, final_func )
```

Creates a new SQL function within a database connection. The first parameter is the database connection. The second parameter is the name of the function as either a UTF-8 or UTF-16 encoded string. The third parameter is the number of expected parameters to the SQL function. If this value is negative, the number of expected parameters is variable or undefined. Fourth is the expected representation for text values passed into the function, and can be one of `SQLITE_UTF8`, `SQLITE_UTF16`, `SQLITE_UTF16BE`, `SQLITE_UTF16LE`, or `SQLITE_ANY`. This is followed by a user-data pointer.

The last three parameters are all function pointers. We will look at the specific prototypes for these function pointers later. To register and create a scalar function, only the first function pointer is used. The other two function pointers are used to register aggregate functions and should be set to NULL when defining a scalar function.

SQLite allows SQL function names to be overloaded based off both the number of parameters and the text representation. This allows multiple C functions to be associated with the same SQL function name. You can use this overloading capability to register different C implementations of the same SQL function. This might be useful to efficiently handle different text encodings, or to provide different behaviors, depending on the number of parameters.

You are not required to register multiple text encodings. When the SQLite library needs to make a function call, it will attempt to find a registered function with a matching text representation. If it cannot find an exact match, it will convert any text values and call one of the other available functions. The value `SQLITE_ANY` indicates that the function is willing to accept text values in any possible encoding.

You can update or redefine a function by simply reregistering it with a different function pointer. To delete a function, call `sqlite3_create_function_xxx()` with the same name, parameter count, and text representation, but pass in NULL for all of the function pointers. Unfortunately, there is no way to find out if a function name is registered or not, outside of keeping track yourself. That means there is no way to tell the difference between a create action and a redefine action.

It is permissible to create a new function at any time. There are limits on when you can change or delete a function, however. If the database connection has any prepared statements that are currently being executed (`sqlite3_step()` has been called at least

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

once, but `sqlite3_reset()` has not), you cannot redefine or delete a custom function, you can only create a new one. Any attempt to redefine or delete a function will return `SQLITE_BUSY`.

If there are no statements currently being executed, you may redefine or delete a custom function, but doing so invalidates all the currently prepared statements (just as any schema change does). If the statements were prepared with `sqlite3_prepare_v2()`, they will automatically reprepare themselves next time they're used. If they were prepared with an original version of `sqlite3_prepare()`, any use of the statement will return an `SQLITE_SCHEMA` error.

The actual C function you need to write looks like this:

```
void custom_scalar_function( sqlite3_context *ctx,
                             int num_values, sqlite3_value **values )
```

This is the prototype of the C function used to implement a custom scalar SQL function. The first parameter is an `sqlite3_context` structure, which can be used to access the user-data pointer as well as set the function result. The second parameter is the number of parameter values present in the third parameter. The third parameter is an array of `sqlite3_value` pointers.

The second and third parameters (`int num_values, sqlite3_value **values`) work together in a very similar fashion to the traditional C main parameters (`int argc, char **argv`).

In a threaded application, it may be possible for different threads to call into your function at the same time. As such, user-defined functions should be thread-safe.

Most user-defined functions follow a pretty standard pattern. First, you'll want to examine the `sqlite3_value` parameters to verify their types and extract their values. You can also extract the user-data pointer passed into `sqlite3_create_function_xxx()`. Your code can then perform whatever calculation or procedure is required. Finally, you can set the return value of the function or return an error condition.

Extracting Parameters

SQL function parameters are passed into your C function as an array of `sqlite3_value` structures. Each of these structures holds one parameter value.

To extract working C values from the `sqlite3_value` structures, you need to call one of the `sqlite3_value_xxx()` functions. These functions are extremely similar to the `sqlite3_column_xxx()` functions in use and design. The only major difference is that these functions take a single `sqlite3_value` pointer, rather than a prepared statement and a column index.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Like their column counterparts, the value functions will attempt to automatically convert the value into whatever datatype is requested. The conversion process and rules are the same as those used by the `sqlite3_column_xxx()` functions. See Table 7-1 for more details.

`const void* sqlite3_value_blob( sqlite3_value *value )`
: Extracts and returns a pointer to a BLOB.

`double sqlite3_value_double( sqlite3_value *value )`
: Extracts and returns a double-precision floating point value.

`int sqlite3_value_int( sqlite3_value *value )`
: Extracts and returns a 32-bit signed integer value. The returned value will be clipped (without warning) if the parameter value contains an integer value that cannot be represented with only 32 bits.

`sqlite3_int64 sqlite3_value_int64( sqlite3_value *value )`
: Extracts and returns a 64-bit signed integer value.

`const unsigned char* sqlite3_value_text( sqlite3_value *value )`
: Extracts and returns a UTF-8 encoded text value. The value will always be null-terminated. Note that the returned `char` pointer is unsigned and will likely require a cast. The pointer may also be NULL if a type conversion was required.

```
const void* sqlite3_value_text16(   sqlite3_value *value )
const void* sqlite3_value_text16be( sqlite3_value *value )
const void* sqlite3_value_text16le( sqlite3_value *value )
```
: Extracts and returns a UTF-16 encoded string. The first function returns a string in the native byte ordering of the machine. The other two functions will return a string that is always encoded in big-endian or little-endian. The value will always be null-terminated. The pointer may also be NULL if a type conversion was required.

There are also a number of helper functions to query the native datatype of a value, as well as query the size of any returned buffers.

`int sqlite3_value_type( sqlite3_value *value )`
: Returns the native datatype of the value. The return value can be one of `SQLITE_BLOB`, `SQLITE_INTEGER`, `SQLITE_FLOAT`, `SQLITE_TEXT`, or `SQLITE_NULL`. This value can change or become invalid if a type conversion takes place.

`int sqlite3_value_numeric_type( sqlite3_value *value )`
: This function attempts to convert a value into a numeric type (either `SQLITE_FLOAT` or `SQLITE_INTEGER`). If the conversion can be done without loss of data, then the conversion is made and the datatype of the new value is returned. If a conversion cannot be done, the value will not be converted and the original datatype of the value will be returned. This can be any value that is returned by `sqlite3_value_type()`.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The main difference between this function and simply calling `sqlite3_value_double()` or `sqlite3_value_int()`, is that the conversion will only take place if it is meaningful and will not result in lost data. For example, `sqlite3_value_double()` will convert a NULL into the value 0.0, while this function will not. Similarly, `sqlite3_value_int()` will convert the first part of the string `'123xyz'` into the integer 123, ignoring the trailing `'xyz'`. This function will not, however, because no sense can be made of the trailing `'xyz'` in a numeric context.

`int sqlite3_value_bytes( sqlite3_value *value )`
: Returns the number of bytes in a BLOB or in a UTF-8 encoded string. If returning the size of a text value, the size will include the null-terminator.

`int sqlite3_value_bytes16( sqlite3_value *value )`
: Returns the number of bytes in a UTF-16 encoded string, including the null-terminator.

As with the `sqlite3_column_xxx()` functions, any returned pointers can become invalid if another `sqlite3_value_xxx()` call is made against the same `sqlite3_value` structure. Similarly, data conversions can take place on text datatypes when calling `sqlite3_value_bytes()` or `sqlite3_value_bytes16()`. In general, you should follow the same rules and practices as you would with the `sqlite3_column_xxx()` functions. See "Result Columns" on page 127 for more details.

In addition to the SQL function parameters, the `sqlite3_context` parameter also carries useful information. These functions can be used to extract either the database connection or the user-data pointer that was used to create the function.

`void* sqlite3_user_data( sqlite3_context *ctx )`
: Extracts the user-data pointer that was passed into `sqlite3_create_function_xxx()` when the function was registered. Be aware that this pointer is shared across all invocations of this function within this database connection.

`sqlite3* sqlite3_context_db_handle( sqlite3_context *ctx )`
: Returns the database connection that was used to register this function.

The database connection returned by `sqlite3_context_db_handle()` can be used by the function to run queries or otherwise interact with the database.

Returning Results and Errors

Once a function has extracted and verified its parameters, it can set about its work. When a result has been computed, that result needs to be passed back to the SQLite engine. This is done by using one of the `sqlite3_result_xxx()` functions. These functions set a result value in the function's `sqlite3_context` structure.

Setting a result value is the only way your function can communicate back to the SQLite engine about the success or failure of the function call. The C function itself has a `void` return type, so any result or error has to be passed back through the context structure. Normally, one of the `sqlite3_result_xxx()` functions is called just prior to

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

calling `return` within your C function, but it is permissible to set a new result multiple times throughout the function. Only the last result will be returned, however.

The `sqlite3_result_xxx()` functions are extremely similar to the `sqlite3_bind_xxx()` functions in use and design. The main difference is that these functions take an `sqlite3_context` structure, rather than a prepared statement and parameter index. A function can only return one result, so any call to an `sqlite3_result_xxx()` function will override the value set by a previous call.

```
void sqlite3_result_blob( sqlite3_context* ctx,
                          const void *data, int data_len, mem_callback )
```

Encodes a data buffer as a BLOB result.

```
void sqlite3_result_double( sqlite3_context *ctx, double data )
```

Encodes a 64-bit floating-point value as a result.

```
void sqlite3_result_int( sqlite3_context *ctx, int data )
```

Encodes a 32-bit signed integer as a result.

```
void sqlite3_result_int64( sqlite3_context *ctx, sqlite3_int64 data )
```

Encodes a 64-bit signed integer as a result.

```
void sqlite3_result_null( sqlite3_context *ctx )
```

Encodes an SQL NULL as a result.

```
void sqlite3_result_text( sqlite3_context *ctx,
                          const char *data, int data_len, mem_callback )
```

Encodes a UTF-8 encoded string as a result.

```
void sqlite3_result_text16(   sqlite3_context *ctx,
                              const void *data, int data_len, mem_callback )
void sqlite3_result_text16be( sqlite3_context *ctx,
                              const void *data, int data_len, mem_callback )
void sqlite3_result_text16le( sqlite3_context *ctx,
                              const void *data, int data_len, mem_callback )
```

Encodes a UTF-16 encoded string as a result. The first function is used for a string in the native byte format, while the last two functions are used for strings that are explicitly encoded as big-endian or little-endian, respectively.

```
void sqlite3_result_zeroblob( sqlite3_context *ctx, int length )
```

Encodes a BLOB as a result. The BLOB will contain the number of bytes specified, and each byte will all be set to zero (`0x00`).

```
void sqlite3_result_value( sqlite3_context *ctx, sqlite3_value *result_value )
```

Encodes an `sqlite3_value` as a result. A copy of the value is made, so there is no need to worry about keeping the `sqlite3_value` parameter stable between this call and when your function actually exits.

This function accepts both protected and unprotected value objects. You can pass one of the `sqlite3_value` parameters to this function if you wish to return one of the SQL function input parameters. You can also pass a value obtained from a call to `sqlite3_column_value()`.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Setting a BLOB or text value requires the same type of memory management as the equivalent `sqlite3_bind_xxx()` functions. The last parameter of these functions is a callback pointer that will properly free and release the given data buffer. You can pass a reference to `sqlite3_free()` directly (assuming the data buffers were allocated with `sqlite3_malloc()`), or you can write your own memory manager (or wrapper). You can also pass in one of the `SQLITE_TRANSIENT` or `SQLITE_STATIC` flags. See "Binding Values" on page 135 for specifics on how these flags can be used.

In addition to encoding specific datatypes, you can also return an error status. This can be used to indicate a usage problem (such as an incorrect number of parameters) or an environment problem, such as running out of memory. Returning an error code will result in SQLite aborting the current SQL statement and returning the error back to the application via the return code of `sqlite3_step()` or one of the convenience functions.

```
void sqlite3_result_error(   sqlite3_context *ctx,
                             const char *msg, int msg_size )
void sqlite3_result_error16( sqlite3_context *ctx,
                             const void *msg, int msg_size )
```

Sets the error code to `SQLITE_ERROR` and sets the error message to the provided UTF-8 or UTF-16 encoded string. An internal copy of the string is made, so the application can free or modify the string as soon as this function returns. The last parameter indicates the size of the message in bytes. If the string is null-terminated and the last parameter is negative, the string size is automatically computed.

```
void sqlite3_result_error_toobig( sqlite3_context *ctx )
```

Indicates the function could not process a text or BLOB value due to its size.

```
void sqlite3_result_error_nomem( sqlite3_context *ctx )
```

Indicates the function could not complete because it was unable to allocate required memory. This specialized function is designed to operate without allocating any additional memory. If you encounter a memory allocation error, simply call this function and have your function return.

```
void sqlite3_result_error_code( sqlite3_context *ctx, int code )
```

Sets a specific SQLite error code. Does not set or modify the error message.

It is possible to return both a custom error message and a specific error code. First, call `sqlite3_result_error()` (or `sqlite3_result_error16()`) to set the error message. This will also set the error code to `SQLITE_ERROR`. If you want a different error code, you can call `sqlite3_result_error_code()` to override the generic error code with something more specific, leaving the error message untouched. Just be aware that `sqlite3_result_error()` will always set the error code to `SQLITE_ERROR`, so you must set the error message before you set a specific error code.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Example

Here is a simple example that exposes the SQLite C API function `sqlite3_limit()` to the SQL environment as the SQL function `sql_limit()`. This function is used to adjust various limits associated with the database connection, such as the maximum number of columns in a result set or the maximum size of a BLOB value.

Here's a quick introduction to the C function `sqlite3_limit()`, which can be used to adjust the soft limits of the SQLite environment:

`int sqlite3_limit( sqlite3 *db, int limit_type, int limit_value )`
: For the given database connection, this sets the limit referenced by the second parameter to the value provided in the third parameter. The old limit is returned. If the new value is negative, the limit value will remain unchanged. This can be used to probe an existing limit. The soft limit cannot be raised above the hard limit, which is set at compile time.

For more specific details on `sqlite3_limit()`, see `sqlite3_limit()` in Appendix G. You don't need a full understanding of how this API call works to understand these examples.

Although the `sqlite3_limit()` function makes a good example, it might not be the kind of thing you'd want to expose to the SQL language in a real-world application. In practice, exposing this C API call to the SQL level brings up some security concerns. Anyone that can issue arbitrary SQL calls would have the capability of altering the SQLite soft limits. This could be used for some types of denial-of-service attacks by raising or lowering the limits to their extremes.

sql_set_limit

In order to call the `sqlite3_limit()` function, we need to determine the `limit_type` and `value` parameters. This will require an SQL function that takes two parameters. The first parameter will be the limit type, expressed as a text constant. The second parameter will be the new limit. The SQL function can be called like this to set a new expression-depth limit:

```
SELECT sql_limit( 'EXPR_DEPTH', 400 );
```

The C function that implements the SQL function `sql_limit()` has four main parts. The first task is to verify that the first SQL function parameter (passed in as `values[0]`) is a text value. If it is, the function extracts the text to the `str` pointer:

```
static void sql_set_limit( sqlite3_context *ctx, int
                                                 num_values, sqlite3_value **values )
{
    sqlite3      *db = sqlite3_context_db_handle( ctx );
    const char   *str = NULL;
    int           limit = -1, val = -1, result = -1;
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    /* verify the first param is a string and extract pointer */
    if ( sqlite3_value_type( values[0] ) == SQLITE_TEXT ) {
        str = (const char*) sqlite3_value_text( values[0] );
    } else {
        sqlite3_result_error( ctx, "sql_limit(): wrong parameter type", -1 );
        return;
    }
```

Next, the function verifies that the second SQL parameter (`values[1]`) is an integer value, and extracts it into the `val` variable:

```
    /* verify the second parameter is an integer and extract value */
    if ( sqlite3_value_type( values[1] ) == SQLITE_INTEGER ) {
        val = sqlite3_value_int( values[1] );
    } else {
        sqlite3_result_error( ctx, "sql_limit(): wrong parameter type", -1 );
        return;
    }
```

Although our SQL function uses a text value to indicate which limit we would like to modify, the C function `sqlite3_limit()` requires a predefined integer value. We need to decode the `str` text value into an integer `limit` value. I'll show the code to `decode_limit_str()` in just a bit:

```
    /* translate string into integer limit */
    limit = decode_limit_str( str );
    if ( limit == -1 ) {
        sqlite3_result_error( ctx, "sql_limit(): unknown limit type", -1 );
        return;
    }
```

After verifying our two SQL function parameters, extracting their values, and translating the text limit indicator into a proper integer value, we finally call `sqlite3_limit()`. The result is set as the result value of the SQL function and the function returns:

```
    /* call sqlite3_limit(), return result */
    result = sqlite3_limit( db, limit, val );
    sqlite3_result_int( ctx, result );
    return;
}
```

The `decode_limit_str()` function is very simple, and simply looks for a predefined set of text values:

```
int decode_limit_str( const char *str )
{
    if ( str == NULL ) return -1;
    if ( !strcmp( str, "LENGTH"          ) ) return SQLITE_LIMIT_LENGTH;
    if ( !strcmp( str, "SQL_LENGTH"      ) ) return SQLITE_LIMIT_SQL_LENGTH;
    if ( !strcmp( str, "COLUMN"          ) ) return SQLITE_LIMIT_COLUMN;
    if ( !strcmp( str, "EXPR_DEPTH"      ) ) return SQLITE_LIMIT_EXPR_DEPTH;
    if ( !strcmp( str, "COMPOUND_SELECT" ) ) return SQLITE_LIMIT_COMPOUND_SELECT;
    if ( !strcmp( str, "VDBE_OP"         ) ) return SQLITE_LIMIT_VDBE_OP;
    if ( !strcmp( str, "FUNCTION_ARG"    ) ) return SQLITE_LIMIT_FUNCTION_ARG;
    if ( !strcmp( str, "ATTACHED"        ) ) return SQLITE_LIMIT_ATTACHED;
    if ( !strcmp( str, "LIKE_LENGTH"     ) ) return SQLITE_LIMIT_LIKE_PATTERN_LENGTH;
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    if ( !strcmp( str, "VARIABLE_NUMBER" ) ) return SQLITE_LIMIT_VARIABLE_NUMBER;
    if ( !strcmp( str, "TRIGGER_DEPTH"   ) ) return SQLITE_LIMIT_TRIGGER_DEPTH;
    return -1;
}
```

With these two functions in place, we can create the `sql_limit()` SQL function by registering the `sql_set_limit()` C function pointer.

```
sqlite3_create_function( db, "sql_limit", 2, SQLITE_UTF8,
                         NULL, sql_set_limit, NULL, NULL );
```

The parameters for this function include the database connection (`db`), the name of the SQL function (`sql_limit`), the required number of parameters (`2`), the expected text encoding (UTF-8), the user-data pointer (NULL), and finally the C function pointer that implements this function (`sql_set_limit`). The last two parameters are only used when creating aggregate functions, and are set to NULL.

Once the SQL function has been created, we can now manipulate the limits of our SQLite environment by issuing SQL commands. Here are some examples of what the `sql_limit()` SQL function might look like if we integrated it into the `sqlite3` tool (we'll see how to do this using a loadable extension later in the chapter).

First, we can look up the current `COLUMN` limit by passing a new limit value of `-1`:

```
sqlite> SELECT sql_limit( 'COLUMN', -1 );
2000
```

We verify the function works correctly by setting the maximum column limit to two, and then generating a result with three columns. The previous limit value is returned when we set the new value:

```
sqlite> SELECT sql_limit( 'COLUMN', 2 );
2000
sqlite> SELECT 1, 2, 3;
Error: too many columns in result set
```

We see from the error that the soft limit is correctly set, meaning our function is working.

One thing you might be wondering about is parameter value count. Although the `sql_set_limit()` function carefully checks the types of the parameters, it doesn't actually verify that `num_values` is equal to two. In this case, it doesn't have to, since it was registered with `sqlite3_create_function()` with a required parameter count of two. SQLite will not even call our `sql_set_limit()` function unless we have exactly two parameters:

```
sqlite> SELECT sql_limit( 'COLUMN', 2000, 'extra' );
Error: wrong number of arguments to function sql_limit()
```

SQLite sees the wrong number of parameters and generates an error for us. This means that as long as a function is registered correctly, SQLite will do some of our value checking for us.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

sql_get_limit

While having a fixed parameter count simplifies the verification code, it might be useful to provide a single-parameter version that can be used to look up the current value. This can be done a few different ways. First, we can define a second C function called `sql_get_limit()`. This function would be the same as `sql_set_limit()`, but with the second block of code removed:

```
/* remove this block of code from a copy of   */
/* sql_set_limit() to produce sql_get_limit() */
if ( sqlite3_value_type( values[1] ) == SQLITE_INTEGER ) {
    val = sqlite3_value_int( values[1] );
} else {
    sqlite3_result_error( ctx, "sql_limit(): wrong parameter type", -1 );
    return;
}
```

With this code removed, the function will never decode the second SQL function parameter. Since `val` is initialized to –1, this effectively makes every call a query call. We register each of these functions separately:

```
sqlite3_create_function( db, "sql_limit", 1,
        SQLITE_UTF8, NULL, sql_get_limit, NULL, NULL );
sqlite3_create_function( db, "sql_limit", 2,
        SQLITE_UTF8, NULL, sql_set_limit, NULL, NULL );
```

This dual registration overloads the SQL function name `sql_limit()`. Overloading is allowed because the two calls to `sqlite3_create_function()` have a different number of required parameters. If the SQL function `sql_limit()` is called with one parameter, then the C function `sql_get_limit()` is called. If two parameters are provided to the SQL function, then the C function `sql_set_limit()` is called.

sql_getset_limit

Although the two C functions `sql_get_limit()` and `sql_set_limit()` provide the correct functionality, the majority of their code is the same. Rather than having two functions, it might be simpler to combine these two functions into one function that can deal with either one or two parameters, and is capable of both getting or setting a limit value.

This combine `sql_getset_limit()` function can be created by taking the original `sql_set_limit()` function and modifying the second section. Rather than eliminating it, as we did to create `sql_get_limit()`, we'll simply wrap it in an `if` statement, so the second section (which extracts the second SQL function parameter) is only run if we have two parameters:

```
/* verify the second parameter is an integer and extract value */
if ( num_values == 2 ) {
    if ( sqlite3_value_type( values[1] ) == SQLITE_INTEGER ) {
        val = sqlite3_value_int( values[1] );
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        } else {
            sqlite3_result_error( ctx, "sql_limit(): wrong parameter type", -1 );
            return;
        }
    }
```

We register the same `sql_getset_limit()` C function under both parameter counts:

```
sqlite3_create_function( db, "sql_limit", 1,
        SQLITE_UTF8, NULL, sql_getset_limit, NULL, NULL );
sqlite3_create_function( db, "sql_limit", 2,
        SQLITE_UTF8, NULL, sql_getset_limit, NULL, NULL );
```

For this specific task, this is likely the best choice. SQLite will verify the SQL function `sql_limit()` has exactly one or two parameters before calling our C function, which can easily deal with either one of those two cases.

sql_getset_var_limit

If for some reason you don't like the idea of registering the same function twice, we could also have SQLite ignore the parameter count and call our function no matter what. This leaves verification of a valid parameter count up to us. To do that, we'd start with the `sql_getset_limit()` function and change it to `sql_getset_var_limit()`, by adding this block at the top of the function:

```
    if ( ( num_values < 1 )||( num_values > 2 ) ) {
        sqlite3_result_error( ctx, "sql_limit(): bad parameter count", -1 );
        return;
    }
```

We register just one version. By passing a required parameter count of `-1`, we're telling the SQLite engine that we're willing to accept any number of parameters:

```
sqlite3_create_function( db, "sql_limit", -1, SQLITE_UTF8,
        NULL, sql_getset_var_limit, NULL, NULL );
```

Although this works, the `sql_getset_limit()` version is still my preferred version. The registration makes it clear which versions of the function are considered valid, and the function code is reasonably clear and compact.

Completely free-form parameter counts are usually used by items like the built-in function `coalesce()`. The `coalesce()` function will take any number of parameters (greater than one) and return the first non-NULL value in the list. Since you might pass anywhere from two to a dozen or more parameters, it is impractical to register each possible configuration, and is better to just allow the function to do its own parameter management.

On the other hand, something like `sql_getset_limit()` can really only accept two configurations: one parameter or two. In that case, I find it easier to explicitly register both parameter counts and allow SQLite to do my parameter verification for me.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Aggregate Functions

Aggregate functions are used to collapse values from a grouping of rows into a single result value. This can be done with a whole table, as is common with the aggregate function `count(*)`, or it can be done with groupings of rows from a `GROUP BY` clause, as is common with something like `avg()` or `sum()`. Aggregate functions are used to summarize, or aggregate, all of the individual row values into some single representative value.

Defining Aggregates

SQL aggregate functions are created using the same `sqlite3_create_function_xxx()` function that is used to create scalar functions (See "Scalar Functions" on page 182). When defining a scalar function, you pass in a C function pointer in the sixth parameter and set the seventh and eighth parameter to NULL. When defining an aggregate function, the sixth parameter is set to NULL (the scalar function pointer) and the seventh and eighth parameters are used to pass in two C function pointers.

The first C function is a "step" function. It is called once for each row in an aggregate group. It acts similarly to an scalar function, except that it does not return a result (it may return an error, however).

The second C function is a "finalize" function. Once all the SQL rows have been stepped over, the finalize function is called to compute and set the final result. The finalize function doesn't take any SQL parameters, but it is responsible for setting the result value.

The two C functions work together to implement the SQL aggregate function. Consider the built-in `avg()` aggregate, which computes the numeric average of all the rows in a column. Each call to the step function extracts an SQL value for that row and updates both a running total and a row count. The finalize function divides the total by the row count and sets the result value of the aggregate function.

The C functions used to implement an aggregate are defined like this:

```
void user_aggregate_step( sqlite3_context *ctx,
                          int num_values, sqlite3_value **values )
```

The prototype of a user-defined aggregate step function. This function is called once for each row of an aggregate calculation. The prototype is the same as a scalar function and all of the parameters have similar meaning. The step function should not set a result value with `sqlite3_result_xxx()`, but it may set an error.

```
void user_aggregate_finalize( sqlite3_context *ctx )
```

The prototype of a user-defined aggregate finalize function. This function is called once, at the end of an aggregation, to make the final calculation and set the result. This function should set a result value or error condition.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Most of the rules about SQL function overloading that apply to scalar functions also apply to aggregate functions. More than one set of C functions can be registered under the same SQL function name if different parameter counts or text encodings are used. This is less commonly used with aggregates, however, as most aggregate functions are numeric-based and the majority of aggregates take only one parameter.

It is also possible to register both scalar and aggregate functions under the same name, as long as the parameter counts are different. For example, the built-in `min()` and `max()` SQL functions are available as both scalar functions (with two parameters) and aggregate functions (with one parameter).

Step and finalize functions can be mixed and matched—they don't always need to be unique pairs. For example, the built-in `sum()` and `avg()` aggregates both use the same step function, since both aggregates need to compute a running total. The only difference between these aggregates is the finalize function. The finalize function for `sum()` simply returns the grand total, while the finalize function for `avg()` first divides the total by the row count.

Aggregate Context

Aggregate functions typically need to carry around a lot of state. For example, the built-in `avg()` aggregate needs to keep track of the running total, as well as the number of rows processed. Each call to the step function, as well as the finalize function, needs access to some shared block of memory that holds all the state values.

Although aggregate functions can call `sqlite3_user_data()` or `sqlite3_context_db_handle()`, you can't use the user-data pointer to store aggregate state data. The user-data pointer is shared by all instances of a given aggregate function. If more than one instance of the aggregate function is active at the same time (for example, an SQL query that averages more than one column), each instance of the aggregate needs a private copy of the aggregate state data, or the different aggregate calculations will get intermixed.

Thankfully, there is an easy solution. Because almost every aggregate function requires some kind of state data, SQLite allows you to attach a data-block to each specific aggregate instance.

`void* sqlite3_aggregate_context( sqlite3_context *ctx, int bytes )`

> This function can be called inside an aggregate step function or finalize function. The first parameter is the `sqlite3_context` structure passed into the step or finalize function. The second parameter represents a number of bytes.
>
> The first time this function is called within a specific aggregate instance, the function will allocate an appropriately sized block of memory, zero it out, and attach it to the aggregate context before returning a pointer. This function will return the same block of memory in subsequent invocations of the step and finalize functions. The memory block is automatically deallocated when the aggregate goes out of scope.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Using this API call, you can have the SQLite engine automatically allocate and release your aggregate state data on a per-instance basis. This allows multiple instances of your aggregate function to be active simultaneously without any extra work on your part.

Typically, one of the first things a step or finalize function will do is call `sqlite3_aggregate_context()`. For example, consider this oversimplified version of sum:

```
void simple_sum_step( sqlite3_context *ctx, int num_values, sqlite3_value **values )
{
    double *total = (double*)sqlite3_aggregate_context( ctx, sizeof( double ) );
    *total += sqlite3_value_double( values[0] );
}

void simple_sum_final( sqlite3_context *ctx )
{
    double *total = (double*)sqlite3_aggregate_context( ctx, sizeof( double ) );
    sqlite3_result_double( ctx, *total );
}

/* ...inside an initialization function... */
    sqlite3_create_function( db, "simple_sum", 1, SQLITE_UTF8, NULL,
            NULL, simple_sum_step, simple_sum_final );
```

In this case, we're only allocating enough memory to hold a double-precision floating-point value. Most aggregate functions will allocate a C struct with whatever fields are required to compute the aggregate, but everything works the same way. The first time `simple_sum_step()` is called, the call to `sqlite3_aggregate_context()` will allocate enough memory to hold a double and zero it out. Subsequent calls to `simple_sum_step()` that are part of the same aggregation calculation (have the same `sqlite3_context`) will have the same block of memory returned, as will `simple_sum_final()`.

Because `sqlite3_aggregate_context()` may need to allocate memory, it is also a good idea to make sure the returned value is not NULL. The above code, in both the step and finalize functions, should really look something like this:

```
    double *total = (double*)sqlite3_aggregate_context( ctx, sizeof( double ) );
    if ( total == NULL ) {
        sqlite3_result_error_nomem( ctx );
        return;
    }
```

The only caution with `sqlite3_aggregate_context()` is in properly dealing with data structure initialization. Because the context data structure is silently allocated and zeroed out on the first call, there is no obvious way to tell the difference between a newly allocated structure, and one that was allocated in a previous call to your step function.

If the default all-zero state of a newly allocated context is not appropriate, and you need to somehow initialize the aggregate context, you'll need to include some type of initialization flag. For example:

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
typedef struct agg_state_s {
    int    init_flag;
    /* other fields used by aggregate... */
} agg_state;
```

The aggregate functions can use this flag to determine if it needs to initialize the aggregate context data or not:

```
agg_state *st = (agg_state*)sqlite3_aggregate_context( ctx, sizeof( agg_state ) );
/* ...return nonmem error if st == NULL... */
if ( st->init_flag == 0 ) {
    st->init_flag = 1;
    /* ...initialize the rest of agg_state... */
}
```

Since the structure is zeroed out when it is first allocated, your initialization flag will be zero on the very first call. As long as you set the flag to something else when you initialize the rest of the data structure, you'll always know if you're dealing with a new allocation that needs to be initialized or an existing allocation that has already been initialized.

Be sure to check the initialization flag in both the step function and the finalize function. There are cases when the finalize function may be called without first calling the step function, and the finalize function needs to properly deal with those cases.

Example

As a more in-depth example, let's look at a *weighted average* aggregate. Although most aggregates take only one parameter, our `wtavg()` aggregate will take two. The first parameter will be whatever numeric value we're trying to average, while the second, optional parameter will be a weighting for this row. If a row has a weight of two, its value will be considered to be twice as important as a row with a weighting of only one. A weighted average is taken by summing the product of the values and weights, and dividing by the sum of the weights.

To put things in SQL terms, if our `wtavg()` function is used like this:

```
SELECT wtavg( data, weight ) FROM ...
```

It should produce results that are similar to this:

```
SELECT ( sum( data * weight ) / sum( weight ) ) FROM ...
```

The main difference is that our `wtavg()` function should be a bit more intelligent about handling invalid weight values (such as a NULL) and assign them a weight value of 1.0.

To keep track of the total data values and the total weight values, we need to define an aggregate context data structure. This will hold the state data for our aggregate. The only place this structure is referenced is the two aggregate functions, so there is no need to put it in a separate header file. It can be defined in the code right along with the two functions:

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
typedef struct wt_avg_state_s {
   double   total_data;  /* sum of (data * weight) values */
   double   total_wt;    /* sum of weight values */
} wt_avg_state;
```

Since the default initialization state of zero is exactly what we want, we don't need a separate initialization flag within the data structure.

In this example, I've made the second aggregate function parameter (the weight value) optional. If only one parameter is provided, all the weights are assumed to be one, resulting in a traditional average. This will still be different than the built-in `avg()` function, however. SQLite's built-in `avg()` function follows the SQL standard in regard to typing and NULL handling, which might not be what you first assume. (See `avg()` in Appendix E for more details). Our `wtavg()` is a bit simpler. In addition to always returning a double (even if the result could be expressed as an integer), it simply ignores any values that can't easily be translated into a number.

First, the step function. This processes each row, adding up the value-weight products, as well as the total weight value:

```
void wt_avg_step( sqlite3_context *ctx, int num_values, sqlite3_value **values )
{
    double        row_wt = 1.0;
    int           type;
    wt_avg_state  *st = (wt_avg_state*)sqlite3_aggregate_context( ctx,
                                            sizeof( wt_avg_state ) );
    if ( st == NULL ) {
        sqlite3_result_error_nomem( ctx );
        return;
    }

    /* Extract weight, if we have a weight and it looks like a number */
    if ( num_values == 2 ) {
        type = sqlite3_value_numeric_type( values[1] );
        if ( ( type == SQLITE_FLOAT )||( type == SQLITE_INTEGER ) ) {
            row_wt = sqlite3_value_double( values[1] );
        }
    }

    /* Extract data, if we were given something that looks like a number. */
    type = sqlite3_value_numeric_type( values[0] );
    if ( ( type == SQLITE_FLOAT )||( type == SQLITE_INTEGER ) ) {
        st->total_data += row_wt * sqlite3_value_double( values[0] );
        st->total_wt   += row_wt;
    }
}
```

Our step function uses `sqlite3_value_numeric_type()` to try to convert the parameter values into a numeric type without loss. If the conversion is possible, we always convert the values to a double-precision floating-point, just to keep things simple. This approach means the function will work properly with text representations of numbers (such as the string `'153'`), but will ignore other datatypes and other strings.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In this case, the function does not report an error, it just ignores the value. If the weight cannot be converted, it is assumed to be one. If the data value cannot be converted, the row is skipped.

Once we have our totals, we need to compute the final answer and return the result. This is done in the finalize function, which is pretty simple. The main thing we need to worry about is the possibility of dividing by zero:

```
void wt_avg_final( sqlite3_context *ctx )
{
    double        result = 0.0;
    wt_avg_state   *st = (wt_avg_state*)sqlite3_aggregate_context( ctx,
                                    sizeof( wt_avg_state ) );
    if ( st == NULL ) {
        sqlite3_result_error_nomem( ctx );
        return;
    }

    if ( st->total_wt != 0.0 ) {
        result = st->total_data / st->total_wt;
    }
    sqlite3_result_double( ctx, result );
}
```

To use our aggregate, our application code needs to register these two functions with a database connection using `sqlite3_create_function()`. Since the `wtavg()` aggregate is designed to take either one or two parameters, we'll register it twice:

```
    sqlite3_create_function( db, "wtavg", 1, SQLITE_UTF8, NULL,
            NULL, wt_avg_step, wt_avg_final );
    sqlite3_create_function( db, "wtavg", 2, SQLITE_UTF8, NULL,
            NULL, wt_avg_step, wt_avg_final );
```

Here are some example queries, as seen from the `sqlite3` command shell. This assumes we've integrated our custom aggregate into the `sqlite3` code (an example of the different ways to do this is given later in the chapter):

```
sqlite> SELECT class, value, weight FROM t;

class       value       weight
----------  ----------  ----------
1           3.4         1.0
1           6.4         2.3
1           4.3         0.9
2           3.4         1.4
3           2.7         1.1
3           2.5         1.1
```

First, we can try things with only one parameter. This will use the default 1.0 weight for each row, resulting in a traditional average calculation:

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
sqlite> SELECT class, wtavg( value ) AS wtavg, avg( value ) AS avg
   ...>   FROM t GROUP BY 1;

class       wtavg       avg
----------  ----------  ----------
1           4.7         4.7
2           3.4         3.4
3           2.6         2.6
```

And finally, here is an example of the full weighted-average calculation:

```
sqlite> SELECT class, wtavg( value, weight ) AS wtavg, avg( value ) AS avg
   ...>   FROM t GROUP BY 1;

class       wtavg             avg
----------  ----------------  ----------
1           5.23571428571428  4.7
2           3.4               3.4
3           2.6               2.6
```

In the case of `class=1`, we see a clear difference, where the heavily weighted 6.4 draws the average higher. For `class=2`, there is only one value, so the weighted and unweighted averages are the same (the value itself). In the case of `class=3`, the weights are the same for all values, so again, the average is the same as an unweighted average.

Collation Functions

Collations are used to sort text values. They can be used with `ORDER BY` or `GROUP BY` clauses, or for defining indexes. You can also assign a collation to a table column, so that any index or ordering operation applied to that column will automatically use a specific collation. Above everything else, SQLite will always sort by datatype. NULLs will always come first, followed by a mix of integer and floating-point numeric values in their natural sort order. After the numbers come text values, followed by BLOBs.

Most types have a clearly defined sort order. NULL types have no values, so they cannot be sorted. Numeric types use their natural numeric ordering, and BLOBs are always sorted using binary comparisons. Where things get interesting is when it comes to text values.

The default collation is known as the `BINARY` collation. The `BINARY` collation sorts individual bytes using a simple numeric comparison of the underlying character encoding. The `BINARY` collation is also used for BLOBs.

In addition to the default `BINARY` collation, SQLite includes a built-in `NOCASE` and `RTRIM` collation that can be used with text values. The `NOCASE` collation ignores character case for the purposes of sorting 7-bit ASCII, and would consider the expression `'A' == 'a'` to be true. It does not, however, consider `'Ä' == 'ä'` to be true, nor does it consider `'Ä' == 'A'` to be true, as the representations of these characters are outside of the ASCII standard. The `RTRIM` collation (right-trim) is similar to the default `BINARY` collation, only it ignores trailing whitespace (that is, whitespace on the right side of the value).

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

While these built-in collations offer some basic options, there are times when complex sort ordering is required. This is especially true when you get into Unicode representations of languages that cannot be represented with a simple 7-bit ASCII encoding. You may also need a specialized sorting function that sorts by whole words or groups of characters if you're storing something other than natural language text. For example, if you were storing gene sequences as text data, you might require a custom sorting function for that data.

User-defined collation functions allow the developer to define a new collation by registering a comparison function. Once registered, this function is used to compare strings as part of any sorting process. By defining the basic comparison operator, you essentially define the behavior of the whole collation.

Registering a Collation

To define a custom collation, an application needs to register a comparison function under a collation name. Anytime the database engine needs to sort something under that collation, it uses the comparison function to define the required ordering. You will need to reregister the collation with each database connection that requires it.

There are three API calls that can be used to register a collation comparison function:

```
int sqlite3_create_collation(   sqlite3 *db, const char *name,
                                int text_rep, void *udp, comp_func )
int sqlite3_create_collation16( sqlite3 *db, const void *name,
                                int text_rep, void *udp, comp_func )
```

Registers a collation comparison function with a database connection. The first parameter is the database connection. The second parameter is the name of the custom collation encoded as a UTF-8 or UTF-16 string. The third parameter is the string encoding the comparison function expects, and can be one of `SQLITE_UTF8`, `SQLITE_UTF16`, `SQLITE_UTF16BE`, `SQLITE_UTF16LE`, or `SQLITE_UTF16_ALIGNED` (native UTF-16 that is 16-bit memory aligned). The fourth parameter is a generic user-data pointer that is passed to your comparison function. The last parameter is a function pointer to your comparison function (the prototype of this function is given below).

You can unregister a collation by passing a NULL function pointer in under the same name and text encoding as it was originally registered.

```
int sqlite3_create_collation_v2( sqlite3 *db, const char *name,
                                 int text_rep, void *udp, comp_func,
                                 dest_func )
```

This function is the same as `sqlite3_create_collation()`, with one additional parameter. The additional sixth parameter is an optional function pointer referencing a clean-up function that is called when the collation is destroyed (the prototype of this function is given below). This allows the collation to release any resources associated with the collation (such as the user-data pointer). A NULL function pointer can be passed in if no destroy function is required.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

A collation is destroyed when the database connection is closed, when a replacement collation is registered, or when the collation name is cleared by binding a NULL comparison function pointer.

The collation name is case-insensitive. SQLite allows multiple C sorting functions to be registered under the same name, so long as they take different text representations. If more than one comparison function is available under the same name, SQLite will pick the one that requires the least amount of conversion. If you do register more than one function under the same name, their logical sorting behavior should be the same.

The format of the user-defined function pointers is given below.

```
int user_defined_collation_compare( void* udp,
                                    int lenA, const void *strA,
                                    int lenB, const void *strB )
```

This is the function type of a user-defined collation comparison function. The first parameter is the user-data pointer passed into `sqlite3_create_collation_xxx()` as the fourth parameter. The parameters that follow pass in the length and buffer pointers for two strings. The strings will be in whatever encoding was defined by the register function. You cannot assume the strings are null-terminated.

The return value should be negative if string A is less than string B (that is, A sorts before B), 0 if the strings are considered equal, and positive if string A is greater than B (A sorts after B). In essence, the return value is the ordering of A minus B.

```
void user_defined_collation_destroy( void *udp )
```

This is the function type of the user-defined collation destroy function. The only parameter is the user-data pointer passed in as the fourth parameter to `sqlite3_create_collation_v2()`.

Although collation functions have access to a user-data pointer, they don't have an `sqlite3_context` pointer. That means there is no way to communicate an error back to the SQLite engine. As such, if you have a complex collation function, you should try to eliminate as many error sources as you can. Specifically, that means it is a good idea to pre-allocate any working buffers you might need, as there is no way to abort a comparison if your memory allocations fail. Since the collation function is really just a simple comparison, it is expected to work and provide an answer every time.

Collations can also be dynamically registered on demand. See `sqlite3_collation_needed()` in Appendix G for more details.

Collation Example

Here is a simple example of a user-defined collation. In this example, we're defining a `STRINGNUM` collation that can be used to sort string representations of numeric values.

Unless they're the same length, string representations of numbers often sort in odd ways. For example, using standard text sorting rules, the string `'485'` will sort before the string `'73'` because the character `'4'` sorts before the character `'7'`, just as the

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

character 'D' sorts before the character 'G'. To be clear, these are text strings made up of characters that represent numeric digits, not actual numbers.

The collation attempts to convert these strings into a numeric representation and then use that numeric value for sorting. Using this collation, the string '485' will sort after '73'. To keep things simple, we're only going to deal with integer values:

```
int col_str_num( void *udp,
        int lenA, const void *strA,
        int lenB, const void *strB )
{
    int valA = col_str_num_atoi_n( (const char*)strA, lenA );
    int valB = col_str_num_atoi_n( (const char*)strB, lenB );

    return valA - valB;
}

static int col_str_num_atoi_n( const char *str, int len )
{
    int total = 0, i;
    for ( i = 0; i < len; i++ ) {
        if ( ! isdigit( str[i] ) ) {
            break;
        }
        total *= 10;
        total += digittoint( str[i] );
    }
    return total;
}
```

The collation attempts to convert each string into an integer value using our custom `col_str_num_atoi_n()` function, and then compares the numeric results. The `col_str_num_atoi_n()` function is very similar to the C standard `atoi()` function, with the prime difference that it takes a maximum length parameter. That is required in this case, since the strings passed into our collations may not be null-terminated.

We would register this collation with SQLite like this:

```
sqlite3_create_collation( db, "STRINGNUM", SQLITE_UTF8, NULL, col_str_num );
```

Because the standard C function `isdigit()` is not Unicode aware, our collation sort function will only work with strings that are limited to 7-bit ASCII.

We can then have SQL that looks like this:

```
sqlite> CREATE TABLE t ( s TEXT );
sqlite> INSERT INTO t VALUES ( '485' );
sqlite> INSERT INTO t VALUES ( '73' );
sqlite> SELECT s FROM t ORDER BY s;
485
73
sqlite> SELECT s FROM t ORDER BY s COLLATE STRINGNUM;
73
485
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

It would also be possible to permanently associate our collation with a specific table column by including the collation in the table definition. See `CREATE TABLE` in Appendix C for more details.

SQLite Extensions

While custom functions are a very powerful feature, they can also introduce undesired dependencies between database files and custom SQLite environments. If a database uses a custom collation in a table definition or a custom function in a view definition, then that database can't be opened by any application (including `sqlite3`) that does not have all the proper custom functions defined.

This normally isn't a big deal for a custom-built application with just a few custom features. You can simply build all the custom code directly into your application. Anytime a database file is created or opened by your application, you can create the appropriate function bindings and make your custom function definitions available for use by the database files.

Where things get tricky is if you need to open your database files in a general purpose application, like the `sqlite3` command-line shell, or one of the third-party GUI database managers. Without some way of bringing your custom functions and features with you, your only choice is to splice your custom-feature code into the source of whatever utilities you require, and build site-specific versions that support your SQL environment. That's not very practical in most cases—especially if the source code to the utility is unavailable.

The solution is to build your custom content as an *extension*. Extensions come in two flavors: static and loadable (dynamic). The difference is in how the extension is built and linked into your main application. The same source can be used to build both a static extension and a loadable extension.

Static extensions can be built and linked directly into an application, not unlike a static C library. Loadable extensions act as external libraries, or "plug-ins," to the SQLite engine. If you build your extension as an external loadable extension, you can load the extension into (almost) any SQLite environment, making your custom functions and SQL environment available to `sqlite3` or any other database manager.

In both cases, extensions are a handy way to package a set of related functions into one deployable unit. This is particularly useful if you're writing an SQL support library that is used by a large number of applications, or if you're writing an SQLite interface to an existing library. Structuring your code as an extension also provides a standard way to distribute a set of custom functions to other SQLite users. By providing your code as an extension, each developer can choose to build and integrate the extension to best suit their needs, without having to worry about the format or design of the extension code.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Even if you plan on statically linking all of your custom function and feature code directly into your application, there is still great value in packaging your code as an extension. By writing an extension, you don't lose the ability to build and statically link your extension directly into your application, but you gain the ability to build an external loadable module.

Having your extended environment available as a loadable module allows you to recreate your application's SQL environment in the `sqlite3` command-line tool, or any other general purpose database manager. This opens up the ability to interactively examine your database files in order to design and test queries, debug problems, and track down customer support issues. This alone is a strong reason to consider writing all your custom functions as loadable extensions, even if you never plan on releasing or distributing the standalone loadable extensions.

Extension Architecture

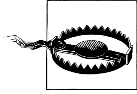
Extensions are nothing more than a style of packaging your code. The SQLite API calls used to register and create custom function handlers, aggregate handlers, collations, or other custom features are completely unchanged in an extension. The only difference is in the initialization process that creates and binds your custom C functions to a database connection. The build process is also slightly different, depending if you want to build a statically linked extension or a dynamically loadable extension, but both types of extensions can be built from the same source code.

Extension architecture focuses on getting dynamically loadable extensions to operate correctly across multiple platforms. The biggest challenge for the dynamic extension architecture is making sure the loadable extension is provided access to the SQLite API. Without getting into a lot of details about how the runtime linker works on different operating systems, the basic issue is that code compiled into an extension and loaded at runtime may not be able to resolve link dependencies from the loadable extension back into the application where the SQLite library sits.

To avoid this problem, when an extension is initialized it is passed a large data structure that contains a C function pointer to every function in the SQLite API. Rather than calling the SQLite functions directly, an extension will dereference the required function pointer and use that. This provides a means to resolve any calls into the SQLite library without depending on the linker. While this isn't fully required for a static extension, the mechanism works equally well with both static and dynamic extensions.

Thankfully, the details of how this big data structure works are all well hidden from the developer by using an alternate header file and a few preprocessor macros. These macros completely hide the whole linker and function pointer issue, but with one limitation: all the extension code that makes calls into the SQLite API must be in the same file, along with the extension initialization function. That code may call out to other files and other libraries, just as long as that "other code" doesn't make any direct calls to *any* `sqlite3_xxx()` function.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



For an SQLite extension to work correctly, every function that interacts with the SQLite library must be in the same C source file as the initialization function.

In practice, this is rarely a significant limitation. Keeping your custom SQLite extensions in their own files, out of your application code, is a natural way to organize your code. Most SQLite extensions are a few hundred lines or less, especially if they are simply acting as a glue layer between SQLite and some other library. This can make them large, but usually not so large they become unmanageable as a single file.

Extension Design

To write an extension, we need to use the extension header file. Rather than using the more common *sqlite.h* file, an extension uses the *sqlite3ext.h* file:

```
#include "sqlite3ext.h"
SQLITE_EXTENSION_INIT1;  /* required by SQLite extension header */
```

The SQLite extension header defines two macros. The first of these is `SQLITE_EXTENSION_INIT1`, and should be referenced at the top of the C file that holds the extension source. This macro defines a file-scoped variable that holds a pointer to the large API structure.

Each extension needs to define an *entry point*. This acts as an initialization function for the extension. The entry point function looks like this:

```
int ext_entry_point( sqlite3 *db, char **error,
                     const sqlite3_api_routines *api )
```

This is the prototype of an extension entry point. The first parameter is the database connection that is loading this extension. The second parameter can be used to pass back a reference to an error message, should the extension be unable to properly initialize itself. The last parameter is used to convey a block of function pointers to assist in the linking process. We'll see how this is used in a moment.

This function is called by the SQLite engine when it loads a static or dynamic extension. Typically, this function will create and register any custom functions or other custom extensions with the database connection.

The entry point has two main jobs. The first job is to finish the initialization process by calling the second extension macro. This should be done as the first bit of code in the entry point (the macro expands into a line of code, so if you're working in pure C you will need to put any function-scope variables before the initialization macro). It *must* be done before any `sqlite3_xxx()` calls are made, or the application will crash:

```
int ext_init( sqlite3 *db, char **error, const sqlite3_api_routines *api )
{
    /* local variable definitions */
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    SQLITE_EXTENSION_INIT2(api);
    /* ... */
}
```

This macro is the only time you should need to directly reference the `api` parameter. Once the entry function has finished the extension API initialization, it can proceed with its second main job, which is registering any and all custom functions or features provided by this extension.

Unlike a lot of functions, the name of the entry point function is somewhat important. When a dynamic extension is loaded, SQLite needs to ask the runtime linker to return a function pointer to the entry point function. In order to do this, the name of the entry point needs to be known.

As we'll see when we look at the dynamic load functions, by default SQLite will look for an entry point named `sqlite3_extension_init()`. In theory, this is a good function name to use, since it will allow a dynamic extension to be loaded even if all you know is the filename.

Although the same application can load multiple dynamic extensions, even if they have the same entry point name, that is not true about statically linked extensions. If you need to statically link more than one extension into your application, the entry points must have unique names or the linker won't be able to properly link in the extensions.

As a result, it is customary to name the entry point something that is unique to the extension, but fairly easy to document and remember. The entry point often shares the same name as the extension itself, possibly with an `_init` suffix. The example extension we'll be looking at is named `sql_trig`, so good choices for the entry point would be `sql_trig()` or `sql_trig_init()`.

Example Extension: sql_trig

For our example extension, we will be creating a pair of SQL functions that expose some simple trigonometric functions from the standard C math library. Since this is just an example, we'll only be creating two SQL functions, but you could use the same basic technique to build SQL functions for nearly every function in the standard math library.

The first half of our *sql_trig.c* source file contains the two functions we will be defining in our example extension. The functions themselves are fairly simple, extracting one double-precision floating-point number, converting from degrees to radians, and then returning the result from the math library. I've also shown the top of the file with the required `#include` statements and initialization macros:

```
/* sql_trig.c */

#include "sqlite3ext.h"
SQLITE_EXTENSION_INIT1;
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
#include <stdlib.h>

/* this bit is required to get M_PI out of MS headers */
#if defined( _WIN32 )
#define _USE_MATH_DEFINES
#endif /* _WIN32 */

#include <math.h>

static void sql_trig_sin( sqlite3_context *ctx, int num_values, sqlite3_value **values )
{
    double a = sqlite3_value_double( values[0] );
    a = ( a / 180.0 ) * M_PI;   /* convert from degrees to radians */
    sqlite3_result_double( ctx, sin( a ) );
}

static void sql_trig_cos( sqlite3_context *ctx, int num_values, sqlite3_value **values )
{
    double a = sqlite3_value_double( values[0] );
    a = ( a / 180.0 ) * M_PI;   /* convert from degrees to radians */
    sqlite3_result_double( ctx, cos( a ) );
}
```

You'll notice these are declared as `static` functions. Making them `static` hides them from the linker, eliminating any possible name conflicts between this extension and other extensions. As long as the extension entry point is in the same file (which, as we've already discussed, is required for other reasons), the entry point will still be able to properly register these functions. Declaring these functions static is not strictly required, but doing so is a good practice and can eliminate linking conflicts.

We then need to define our entry point. Here is the second part of the *sql_trig.c* file:

```
int sql_trig_init( sqlite3 *db, char **error, const sqlite3_api_routines *api )
{
    SQLITE_EXTENSION_INIT2(api);

    sqlite3_create_function( db, "sin", 1,
            SQLITE_UTF8, NULL, sql_sin, NULL, NULL );
    sqlite3_create_function( db, "cos", 1,
            SQLITE_UTF8, NULL, sql_cos, NULL, NULL );

    return SQLITE_OK;
}
```

This entry point function should *not* be declared `static`. Both the static linker (in the case of a static extension) and the dynamic linker (in the case of a loadable extension) need to be able to find the entry point function for the extension to work correctly. Making the function `static` would hide the function from the linker.

These two blocks of code make up our entire *sql_trig.c* source file. Let's look at how to build that file as either a static extension or a dynamically loadable extension.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Building and Integrating Static Extensions

To statically link an extension into an application, you can simply build the extension source file into the application, just like any other *.c* file. If your application code was contained in the file *application.c*, you could build and link our example `sql_trig` extension using the commands shown here.

In the case of most Linux, Unix, and Mac OS X systems, our trig example requires that we explicitly link in the math library (`libm`). In some cases, the standard C library (`libc`) is also required. Windows includes the math functions in the standard runtime libraries, so linking in the math library is not required.

Unix and Mac OS X systems (with math lib):

```
$ gcc -o application application.c sqlite3.c sql_trig.c -lm
```

Windows systems, using the Visual Studio compiler:

```
> cl /Feapplication application.c sqlite3.c sql_trig.c
```

These commands should produce an executable named `application` (or `application.exe` under Windows).

Just linking the code together doesn't magically make it integrate into SQLite. You still need to make SQLite aware of the extension so that the SQLite library can initialize the extension correctly:

`int sqlite3_auto_extension( entry_point_function );`
: Registers an extension entry point function with the SQLite library. Once this is done, SQLite will automatically call an extension's entry point function for every database connection that is opened. The only parameter is a function pointer to the entry point.

  This function only works with statically linked extensions and does not work with dynamically loadable extensions. This function can be called as many times as is necessary to register as many unique entry points as are required.

This function is called by an application, typically right after calling `sqlite3_initialize()`. Once an extension's entry point is registered with the SQLite library, SQLite will initialize the extension for each and every database connection it opens or creates. This effectively makes your extension available to all database connections managed by your application.

The only odd thing about `sqlite3_auto_extension()` is the declaration of the entry point function. The auto extension API call declares the function pointer to have a type of `void entry_point( void )`. That defines a function that takes no parameters and returns no value. As we've already seen, the actual extension entry point has a slightly more complex prototype.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The code that actually calls the extension first casts the provided function pointer to the correct type, so the fact that the types don't match is only an issue for setting the pointer. Extensions typically don't have header files, since the entry point function would typically be the only thing in a header. To get everything working, you can either provide the proper prototype for the entry point and then cast back to what the API is expecting, or you can simply declare the function prototype incorrectly, and let the linker match things up. Pure C doesn't type-check function parameters when it links, so this will work, even if it isn't the most elegant approach.

Here's what the proper prototype with a cast might look like in our application code:

```
/* declare the (correct) function prototype manually  */
int sql_trig_init( sqlite3 *db, char **error, const sqlite3_api_routines *api );

/* ... */
    sqlite3_auto_extension( (void(*)(void))sql_trig_init ); /* needs cast */
/* ... */
```

Or, if you're working in pure C, you can just declare a different prototype:

```
/* declare the (wrong) function prototype manually */
void sql_trig_init(void);

/* ... */
    sqlite3_auto_extension( sql_trig_init );
/* ... */
```

As long as the actual `sql_trig_init()` function is in a different file, this will compile and link correctly, resulting in the desired behavior.

If you want a quick practical example of how to add a static extension to an existing application, we can add our `sql_trig` extension to the `sqlite3` shell with a minimum number of changes. We'll need our *sql_trig.c* file, which contains the two SQL trig functions, plus the `sql_trig_init()` entry function. We'll also need the *shell.c* source code for the `sqlite3` command-line application.

First, we need to add some initialization hooks into the `sqlite3` source. Make a copy of the *shell.c* file as *shell_trig.c*. Open your new copy and search for the phrase "`int main(`" to quickly locate the starting point of the application. Right before the main function, in global file scope, add a prototype for our `sql_trig_init()` entry point:

```
/* ... */
void sql_trig_init(void);  /* insert this line */

int main(int argc, char **argv){
/* ... */
```

Then, inside the existing `main()` function, search for a call to "`open_db(`" to find a good spot to insert our code. Right before the small block of code (and comments) that contains the first call to `open_db()`, add this line:

```
sqlite3_auto_extension( sql_trig_init );
```


Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

With those two edits, you can save and close the *shell_trig.c* file. We can then recompile our modified *shell_trig.c* source into a custom `sqlite3trig` utility that has our extension built into it.

Unix/Linux and Mac OS X:

```
$ gcc -o sqlite3trig sqlite3.c shell_trig.c sql_trig.c -lm
```

Windows:

```
> cl /Fesqlite3trig sqlite3.c shell_trig.c sql_trig.c
```

Our new `sqlite3trig` application now has our extension built directly into it, and our functions are accessible from any database that is opened with our modified `sqlite3trig` utility:

```
$ ./sqlite3trig
SQLite version 3.X.XX
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT sin( 30 );
0.5
sqlite> SELECT cos( 30 );
0.866025403784439
```

Although we had to modify the source, the modifications were fairly small. While we needed to modify and recompile the main application (`sqlite3trig`, in this case) to integrate the extension, you can see how easy it would be to add additional extensions.

Using Loadable Extensions

Dynamic extensions are loaded on demand. An application can be built without any knowledge or understanding of a specific extension, but can still load it when requested to do so. This means you can add new extensions without having to rebuild or recompile an application.

Loadable extension files are basically shared libraries in whatever format is appropriate for the platform. Loadable extensions package compiled code into a format that the operating system can load and link into your application at runtime. Table 9-1 provides a summary of the appropriate file formats on different platforms.

Table 9-1. Summary of loadable extension file format

Platform	File type	Default file extension
Linux and most Unix	Shared object file	*.so*
Mac OS X	Dynamic library	*.dylib*
Windows	Dynamically linked library	*.DLL*

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Loadable extensions are not supported on all platforms. Loadable extensions depend on the operating system having a well-supported runtime linker, and not all handheld and embedded devices offer this level of support. In general, if a platform supports some type of dynamic or shared library for application use, there is a reasonable chance the loadable extension interface will be available. If the platform does not support dynamic or shared libraries, you may be limited to statically linked extensions. However, in most embedded environments this isn't a major limitation.

Although the file formats and extensions are platform dependent, it is not uncommon to pick a custom file extension that is used across all your supported platforms. Using a common file extension is not required, but it can keep the cross-platform C or SQL code that is responsible for loading the extensions a bit simpler. Like database files, there is no official extension for an SQLite loadable extension, but *.sqlite3ext* is sometimes used. That's what I'll use in our examples.

Building Loadable Extensions

Generally, building a loadable extension is just like building a dynamic or shared library. The code must first be compiled into an object file (a *.o* or *.obj* file) and that file must be packaged into a shared or dynamic library. The process of building the object file is exactly the same for both static and dynamic libraries. You can build the object file directly with one of these commands.

Mac OS X and Unix/Linux:

```
$ gcc -c sql_trig.c
```

Windows:

```
> cl /c sql_trig.c
```

Once you have the object file, that needs to be converted into a dynamic or shared library using the linker. The commands for that are a bit more platform dependent.

First, the Unix and Linux command, which builds a shared object file and links in the standard math library:

```
$ ld -shared -o sql_trig.sqlite3ext sql_trig.o -lm
```

Mac OS X, which uses dynamic libraries, rather than shared object files:

```
$ ld -dylib -o sql_trig.sqlite3ext sql_trig.o -lm
```

And finally, Windows, where we need to build a DLL file. In this case, we need to tell the linker which symbols we want exported. For an extension, only the entry point needs to be exported, so we just include that on the command-line:

```
> link /dll /out:sql_trig.sqlite3ext /export:sql_trig_init sql_trig.obj
```

You can test out your dynamic extension in `sqlite3` using the `.load` command. The command takes two parameters. The first is the filename of your loadable extension, and the second is the name of the entry point function:

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
$ sqlite3
SQLite version 3.X.XX
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT sin( 60 );
Error: no such function: sin
sqlite> .load sql_trig.sqlite3ext sql_trig_init
sqlite> SELECT sin( 60 );
0.866025403784439
```

As you can see, when we first start `sqlite3`, it has no awareness of our extension or the SQL functions it contains. The `.load` command is used to dynamically load the extension. Once loaded, our custom trig functions are available without any need to recompile or rebuild the `sqlite3` utility.

Loadable Extension Security

There are some minor security concerns associated with loadable extensions. Because an extension might contain just about any code, a loadable extension might be used to override application values for the SQLite environment. In specific, if an application uses an authorization function to protect against certain types of queries or modifications, a loadable extension could clear the authorization callback function, eliminating any authorization step (see `sqlite3_set_authorizer()` in Appendix G for more details).

To prevent this, and other possible issues, an application must explicitly enable the ability to load external extensions. This has to be done each time a database connection is established.

`int sqlite3_enable_load_extension( sqlite3 *db, int onoff )`

Enables or disables the ability to load dynamic extensions. *Loadable extensions are off by default.*

The first parameter is the database connection to set. The second parameter should be true (nonzero) to enable extensions, or false (zero) to disable them. This function always returns `SQLITE_OK`.

Most general purpose applications, including the `sqlite3` shell, automatically enable loadable extensions for every database they open. If your application will support loadable extensions, you will need to enable this as well. Extension loading needs to be enabled for each database connection, every time the database connection is opened.

Loading Loadable Extensions

There are two ways to load an extension. One is through a C API call, and one is through an SQL function that calls down into the same code as the C API function. In both cases, you provide a filename and, optionally, the name of the entry point function.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
int sqlite3_load_extension( sqlite3 *db, const char *ext_name,
                            const char *entry_point, char **error )
```

Attempts to load a loadable extension and associate it to the given database connection. The first parameter is the database connection to associate with this extension. The second parameter is the filename of the extension. The third parameter is the name of the entry point function. If the entry point name is NULL, the entry point `sqlite3_extension_init` is used. The fourth parameter is used to pass back an error message if something goes wrong. This string buffer should be released with `sqlite3_free()`. This last parameter is optional and can be set to NULL.

This will return either `SQLITE_OK`, to indicate the extension was loaded and the initialization function was successfully called, or it may return `SQLITE_ERROR` to indicate something went wrong. If an error condition is returned, there may or may not be a valid error string.

This function is typically called as soon as a database connection is opened, before any statements are prepared. Although it is legal to call `sqlite3_load_extension()` at any time, any API calls made by the extension entry point and initialization function are subject to standard restrictions. In specific, that means any calls to `sqlite3_create_function()` made by the extension entry point function will fail to redefine or delete a function if there are any executing SQL statements.

The other way to load a loadable extension is with the built-in SQL function `load_extension()`.

```
load_extension( 'ext_name' )
load_extension( 'ext_name', 'entry_point' )
```

This SQL function loads the extension with the given filename. If an entry point name is given, that is used as the initialization function. If not, the name `sqlite3_extension_init` will be used.

This function is similar to the C `sqlite3_load_extension()` call, with one major limitation. Because this is an SQL function, when it is called there will be, by definition, an SQL statement executing when the extension is loaded. That means that any extension loaded with the `load_extension()` SQL function will be completely unable to redefine or delete a custom function, including the specialized set of `like()` functions.

To avoid this problem while testing your loadable extensions in the `sqlite3` shell, use the `.load` command. This provides direct access to the C API call, allowing you to get around the limitations in the SQL function. See `.load` in Appendix B for more details.

No matter which mechanism you use to load a loadable extension, you'll need to do it for each database connection your application opens. Unlike the `sqlite3_auto_extension()` function, there is no automatic way to import a set of loadable extensions for each and every database.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The only way to completely unload a loadable extension is to close the database connection.

Multiple Entry Points

Although most extensions have only a single entry point function, there is nothing that says this must be true. It is perfectly acceptable to define multiple entry points in a single extension—just make sure they each call `SQLITE_EXTENSION_INIT2()`.

Multiple entry points can be used to control the number of imported functions. For example, if you have a very large extension that defines a significant number of functions in several different subcategories, you would likely define one main entry point that imports every extension, aggregation, collation, and other features with one call. You could also define an entry point for each subcategory of functionality, or one entry point for all the functions, one for all the collations, etc. You might also define one entry point to bind UTF-8 functions, and another for UTF-16.

No matter how you want to mix and match things, this allows an extension user to import just the functionality they need. There is no danger in redefining a function from two different entry points (assuming all of the entry points register similar functions in similar ways), so different entry points can register overlapping sets of functions without concern.

Even if your extension is not large and doesn't really justify multiple entry points, a second one can still be handy. Some extensions define a "clear" entry point, for example, `sql_trig_clear()`. This would typically be very similar to the `_init()` entry point function, but rather than binding all the function pointers into a database connection, it would bind all NULL pointers. This effectively "unloads" the extension from the SQL environment—or at least removes all the functions it created. The extension file may still be in memory, but the SQL functions are no longer available to that database connection. The only thing to remember about a `_clear()` entry point is that it cannot be called while an SQL statement is being executed, because of the redefine/delete rules for functions like `sqlite3_create_function()`. This also means you cannot call a `_clear()` entry point using the SQL function `load_extension()`.

Chapter Summary

Custom functions, aggregations, and collations can be an extremely powerful means to extend and expand the SQL environment to fit your own needs and designs. Extensions make for a relatively painless way to modularize and compartmentalize those custom features. This makes the extension code easier to test, support, and distribute.

In the next chapter, we'll look at one of the more powerful customizations in SQLite: virtual tables. Virtual tables allow a developer to merge the SQLite environment to just about any data source. Like other custom features, the easiest way to write a virtual table is to use an extension.

Chapter 9. SQL Functions and Extensions. Using SQLite, ISBN: 9781449394592
Prepared for jts@nmrx.com, Jere Sandidge
Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Copyright © 2010 Jay Kreibich. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.