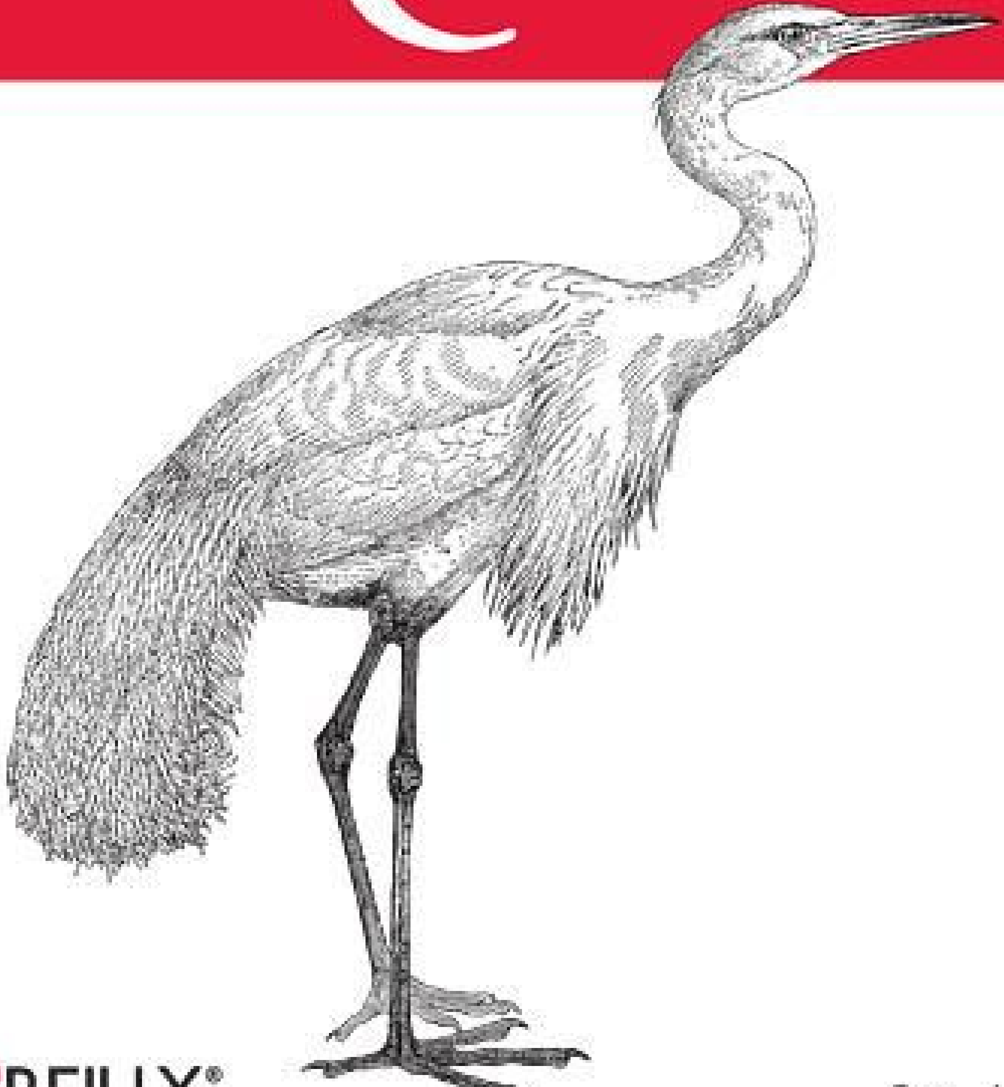
Using

SQLite



O'REILLY®

Jay A. Kreibich

Table of Contents

CHAPTER 10

Virtual Tables and Modules

This chapter looks at how to use and write virtual tables. A *virtual table* is a custom extension to SQLite that allows a developer to define the structure and contents of a table through code. To the database engine, a virtual table looks like any other table—a virtual table can be queried, updated, and manipulated using the same SQL commands that are used on other tables. The main difference is where the table data comes from. When processing a normal table, the SQLite library might access a database file to retrieve row and column values. In the case of a virtual table, the SQLite library calls in to your code to perform these functions and return data values. Your functions, in turn, can gather or calculate the requested data from whatever sources you want.

Developing a virtual table implementation, which is known as an SQLite *module*, is a fairly advanced feature. This chapter should give you a good idea of what virtual tables are capable of doing and the basics of how to write your own module. We'll be walking through the code for two different modules. The first is a fairly simple one that exposes some internal SQLite data as a table. The second example will allow read-only access to standard Apache *httpd* server logs.

This chapter should provide a solid starting point. However, if you find yourself having to write a more robust module, you may need to dig a bit deeper into the development documentation, found at *http://www.sqlite.org/vtab.html*. I would also suggest having a look at the source code to some of the other modules out there (including those that ship with SQLite) to get a better idea of how the advanced features work. Modules are complex enough that sometimes it is easier to modify an existing module, rather than implementing everything from the ground up.

Like the last chapter, this type of advanced development is usually best learned hands-on. To help with this, the full source to both examples is available in the book download. See "Example Code Download" on page xvi for more details.

Introduction to Modules

Virtual tables are typically used to link the SQLite database engine to an alternate data source. There are two general categories of virtual tables: internal and external. There aren't any implementation differences between the categories, they just provide a rough way to define a module's functionality.

Internal Modules

Internal modules are self-contained within the database. That is, the virtual table acts as a fancy front-end to more traditional database tables that are created and maintained by the virtual table module. These back-end tables are sometimes known as *shadow tables*. Most importantly, all the data used by the module is still stored within the database file. These types of modules typically provide a specialized type of indexing or lookup feature that is not well suited to the native database indexes. Internal virtual tables may require multiple shadow tables to efficiently operate.

The two largest modules included in the SQLite distribution (FTS3 and R*Trees) are both internal style modules. Both of these modules create and configure several standard tables to store and index the data they've been asked to maintain.

Generally, internal modules are used to improve or extend the data manipulation facilities of a database. In most cases, an internal virtual table isn't doing anything an SQL developer couldn't do on their own, the module is just making it easier or faster (or both). Internal modules often play the role of an abstract "smart view" that offers highly optimized access patterns to specific types of data or specific structures of data. Both the Full Text Search module and the R*Tree module are prime examples of modules that provide highly specialized searches on specific types and structures of data.

External Modules

The other major category of modules are external modules. These are modules that interface with some type of external data source. That data source might be something as simple as an external file. For example, a module could expose a CSV file or Excel file as an SQL table within the database. Pretty much any structured file can be exposed this way. An external module can also be used to present other data sources to the SQLite database engine. You could actually write an SQLite module that exposed tables from a MySQL database to the SQLite database engine. Or, for something a bit more unusual, have the query `SELECT ip FROM dns WHERE hostname = 'www.oreilly.com'` go out and process a DNS request. External modules can get quite exotic.

In the case of external modules, it is important to understand that the data is not imported or copied into the SQLite database. Rather than loading the data into standard tables and allowing you to access it from there, an external module acts as a real-time translator between the SQLite data structures and whatever external data source you wish to access. Modules will typically reflect changes to their data source in real time.

Of course, you can use an external module as an importer by copying the data from a virtual table to a standard table with an `INSERT...SELECT` statement. If the module has full read/write support, you can even use it as an exporter by copying data from the database into the virtual table. By using this technique, I've seen cases of SQLite being used as a "universal translator" for several different external data formats. By writing a virtual table module that can speak to each file format, you can easily and quickly move data between supported formats.

Example Modules

To help explain how modules work, we're going to work through two examples. The first example is a very simple internal module that exposes the output of the `PRAGMA database_list` command as a full-blown table. This allows you to run `SELECT` queries (including `WHERE` constraints) against the current database list. Although this module is read-only and extremely simple, it should serve as a good first introduction to the module system.

The second example is a bit more in-depth. We'll be looking at building an external module that exposes Apache *httpd* server logs to the database engine. This allows a webmaster to run SQL queries directly against a logfile (including the active logfile) without having to first import the data into a traditional database table.

SQL for Anything

As we'll see with the webserver logfile example, developing an external SQLite module can be an easy way to provide generic search-and-report services to arbitrary data formats. In the case of webserver logs, many server administrators have a stash of scripts and utilities they use for logfile analysis. While these can work quite well for clearly defined tasks, such scripts often require significant code modifications to alter search parameters or report formats. This can make custom scripts difficult to modify and somewhat inflexible.

Webserver log analysis is a common enough problem that there are some extremely powerful general purpose packages available for download. Some of these packages are quite robust and impressive, but to use them effectively requires understanding and experience with the package and the tools that it provides.

With the external data module, you can simply attach the SQLite engine directly to your logfiles, making the logs appear as a big (and constantly updating) table. Once this is done, you have the whole power of the relational database engine at your disposal. Best of all, the queries and searches are all defined in SQL, a language that many web administrators already know. Report generation becomes a snap and, when combined with the `sqlite3` command-line utility, the module will enable real-time interaction with the live log data. This allows a system administrator faced with a security or performance incident to quickly formulate and execute arbitrary searches and summary reports interactively, in a language and environment they're already comfortable using.

This is one of the more compelling uses of virtual tables. While there are many instances of applications that can take advantage of the custom index formats and improved search features offered by some virtual tables, the true magic happens with external modules. The ability to integrate any regular data source into a full SQL environment makes for an extremely powerful and enabling tool, especially in cases where there is a need to directly interact with the data in real time.

The next time you're thinking about clobbering together some scripts to scan or filter a structured data source, ask yourself how hard it would be to write an SQLite module instead. Modules can definitely be tricky to write, but once you have a working module, you also have the full power of the SQL language at your hands.

Module API

The virtual table API is one of the more advanced SQLite APIs. In specific, it does very little hand-holding and will often fool those that make assumptions. The functions you need to write are often required to do a very specific set of operations. If you fail to do any one of those, or forget to initialize a data structure field, the result might very well be a bus error or segmentation fault.

That street goes two ways, however. While the SQLite core expects you to do your job, it does a very good job of always doing its job in a very predictable and documented way. Most of this code operates fairly deeply in the SQLite core, and SQLite does a solid job of protecting your code against odd user behavior. For example, none of the example code checks for NULL parameter values, as you can be sure SQLite will never allow a NULL database pointer (or some equally critical parameter) to be passed into your function.

Implementing a virtual table module is a bit like developing an aggregate function, only a lot more complex. You must write a series of functions that, taken together, define the behavior of the module. This block of functions is then registered under a module name.

```
int sqlite3_create_module( sqlite3 *db, const char *name,
                           const sqlite3_module *module, void *udp )
```

Creates and registers a virtual table module with a database connection. The second parameter is the name of the module. The third parameter is a block of function pointers that implements the virtual table. This pointer must remain valid until the SQLite library is shut down. The final parameter is a generic user-data pointer that is passed to some of the module functions.

```
int sqlite3_create_module_v2( sqlite3 *db, const char *name,
                              const sqlite3_module *p, void *udp,
                              destroy_callback )
```

The v2 version of this function is identical to the original function, except for an additional fifth parameter. This version adds a destroy callback of the form `void callback(void *udp)`. This function can be used to release or otherwise clean up the user-data pointer, and is called when the entire module is unloaded. This is done when the database is shut down, or when a new module with the same name is registered in place of this one. The destroy function pointer is optional, and can be set to NULL.

Function pointers are passed in through an `sqlite3_module` structure. The main reason for this is that there are nearly 20 functions that define a virtual table. All but a few of those functions are mandatory.

A module defines a specific type of virtual table. Once a module has been successfully registered, an actual table instance of that type must be created using the SQL command `CREATE VIRTUAL TABLE`. A single database may have multiple instances of the same type of virtual table. A single database may also have different types of virtual tables, just as long as all the modules are properly registered.

The syntax for the `CREATE VIRTUAL TABLE` command looks something like this:

```
CREATE VIRTUAL TABLE table_name USING module_name( arg1, arg2, ... )
```

A virtual table is named, just like any other table. To define the table, you must provide the module name and any arguments the module requires. The argument block is optional, and the exact meaning of the arguments is up to the individual module implementations. It is the responsibility of the module to define the actual structure (column names and types) of the table. The arguments have no predefined structure and do not need to be valid SQL expressions or column definitions. Each argument is passed as a literal text value to the module, with only the leading and trailing whitespace trimmed. Everything else, including whitespace within the argument, is passed as a single text value.

Here is a quick overview of the different functions that are defined by `sqlite3_module` structure. When we look at the example modules, we'll go back through these one at a time in much more detail. The module functions are divided up into three rough groups. The first set of functions operate on table instances. The second set includes the functions that scan a table and return data values. The last group of functions deals with implementing transaction control. To implement a virtual table module, you will need to write a C function that performs each of these tasks.

Functions that deal with individual table instances include:

`xCreate()`
: Required. Called when a virtual table instance is first created with the `CREATE VIRTUAL TABLE` command.

`xConnect()`
: Required, but frequently the same as `xCreate()`. Very similar to `xCreate()`, this is called when a database with an existing virtual table instance is loaded. Called once for each table instance.

`xDisconnect()`
: Required. Called when a database containing a virtual table instance is detached or closed. Called once for each table instance.

`xDestroy()`
: Required, but frequently the same as `xDisconnect()`. Very similar to `xDisconnect()`, this is called when a virtual table instance is destroyed with the `DROP TABLE` command.

`xBestIndex()`
: Required. Called, sometimes several times, when the database engine is preparing an SQL statement that involves a virtual table. This function is used to determine how to best optimize searches and queries made against the table. This information helps the optimizer understand how to get the best performance out of the table.

`xUpdate()`
: Optional. Called to modify (`INSERT`, `UPDATE`, or `DELETE`) a table row. If this function is not defined, the virtual table will be read-only.

`xFindFunction()`
: Optional. Called when preparing an SQL statement that uses virtual table values as parameters to an SQL function. This function allows the module to override the default implementation of any SQL function. This is typically used in conjunction with the SQL functions `like()` or `match()` to define module-specific versions of these functions (and, from that, module-specific versions of the `LIKE` and `MATCH` SQL expressions).

`xRename()`
: Required. Called when a virtual table is renamed using the `ALTER TABLE...RENAME` command.

The second group of functions deals with processing table scans. These functions operate on a *table cursor*, which holds all of the state information required to perform a table scan. As the database engine scans a table and steps through each individual row, the cursor is responsible for keeping track of which row is being processed.

A single virtual table instance may be involved in more than one table scan at a time. To function correctly, the module must keep all state information in the table cursor, and cannot use user-data pointers or static variables. Consider, for example, a virtual table instance that is self-joined, and must have more than one scan active at the same time.

Cursor functions include:

`xOpen()`
: Required. Called to create and initialize a table cursor.

`xClose()`
: Required. Called to shut down and release a table cursor.

`xFilter()`
: Required. Called to initiate a table scan and provide information about any specific conditions put on this particular table scan. Conditions typically come from `WHERE` constraints on the query. The `xFilter()` function is designed to work in conjunction with `xBestIndex()` to allow a virtual table to pre-filter as many rows as it can. After readying the module for a table scan, `xFilter()` should also look up the first row. This may be called more than once between `xOpen()` and `xClose()`.

`xNext()`
: Required. Called to advance a table cursor to the next row.

`xEof()`
: Required. Called to see if a table cursor has reached the end of the table or not. EOF is traditional shorthand for *end-of-file*. This function is always called right after a call to `xFilter()` or `xNext()`.

`xRowid()`
: Required. Called to extract the virtual `ROWID` of the current row.

`xColumn()`
: Required. Called to extract a column value for the current row. Normally called multiple times per row.

Finally, we have the transaction control functions. These allow external data sources to take part in the transaction control process, and include:

`xBegin()`
: Optional. Called when a transaction is started.

`xSync()`
: Optional. Called to start committing a transaction.

`xCommit()`
: Optional. Called to finalize a database transaction.

`xRollback()`
: Optional. Called to roll back a database transaction.

If this sounds confusing, don't give up just yet. As we start to work through the code examples, we will go back through each function and take a closer look at all the details.

You may be surprised to see that the transactional functions are optional. The reason for this is that internal modules don't need or require their own transactional control. When an internal module modifies any standard table in response to a virtual table operation, the normal transactional engine is already protecting those changes and updates. Additionally, external read-only modules don't require transactional control because they aren't driving any modifications to their external data sources. The only type of module that really needs to implement transactional control are those that provide transaction-safe read/write support to external data sources.

Simple Example: dblist Module

The first example takes the output of the `PRAGMA database_list` command and presents it as a table. Since the output from the `PRAGMA` command is already in the same structure as a table, this conversion is fairly simple. The main reason for doing this is to use the full `SELECT` syntax, including `WHERE` conditions, against the virtual table. This is not possible with the `PRAGMA` command.

The `PRAGMA database_list` command normally returns three columns: `seq`, `name`, and `file`. The `seq` column is a sequence value that indicates which "slot" the database is attached to. The `name` column is the logical name of the database, such as `main` or `temp`, or whatever name was given to the `ATTACH DATABASE` command. (See `ATTACH DATABASE` in Appendix C). The `file` column displays the full path to the database file, if such a file exists. In-memory databases, for example, do not have any associated filenames.

To keep things simple, the module uses the `seq` value as our virtual `ROWID` value. The `seq` value is an integer value and is unique across all of the active databases, so it serves this purpose quite well.

Create and Connect

The first set of functions we'll be looking at are used to create or connect a virtual table instance. The functions you need to provide are:

```
int xCreate( sqlite3 *db, void *udp,
             int argc, char **argv,
             sqlite3_vtab **vtab, char **errMsg )
```

Required. This function is called by SQLite in response to a `CREATE VIRTUAL TABLE` command. This function creates a new instance of a virtual table and initializes any required data structures and database objects.

The first parameter is the database connection where the table needs to be created. The second parameter is the user-data pointer passed into `sqlite3_create_module()`. The third and fourth parameters pass in a set of creation arguments. The fifth parameter is a reference to a virtual table (`vtab`) structure pointer. Your function must allocate and return one of these structures. The final parameter is a reference to an error message. This allows you to pass back a custom error message if something goes wrong.

If everything works as planned, this function returns `SQLITE_OK`. If the return code is anything other than `SQLITE_OK`, the `vtab` structure should *not* be allocated.

Every module is passed at least three arguments. The variable argv[0] will always contain the name of the module used to create the virtual table. This allows the same xCreate() function to be used with similar modules. The logical name of the database (main, temp, etc.) is passed in as argc[1], and argv[2] contains the user-provided table name. Any additional arguments given to the CREATE VIRTUAL TABLE statement will be passed in, starting with argv[3], as text values.

```
int xConnect( sqlite3 *db, void *udp,
              int argc, char **argv,
              sqlite3_vtab **vtab, char **errMsg )
```

Required. The format and parameters of this function are identical to xCreate(). The main difference is that xCreate() is only called when a virtual table instance is first created. xConnect(), on the other hand, is called any time a database is opened. The function still needs to allocate and return a vtab structure, but it should not need to initialize any database objects.

If no object creation step is required, many modules use the same function for both xCreate() and xConnect().

These functions bring a virtual table instance into being. For each virtual table, only one of these functions will be called over the lifetime of a database connection.

The create and connect functions have two major tasks. First, they must allocate an sqlite3_vtab structure and pass that back to the SQLite engine. Second, they must define the table structure with a call to sqlite3_declare_vtab(). The xCreate() call must also create and initialize any storage, be it shadow tables, external files, or whatever is required by the module design. The order of these tasks is not important, so long as all of the tasks are accomplished before the xCreate() or xConnect() function returns.

Allocate the vtab structure

The xCreate() and xConnect() functions are responsible for allocating and passing back an sqlite3_vtab structure. That structure looks like this:

```
struct sqlite3_vtab {
    const sqlite3_module  *pModule;  /* module used by table */
    int                   nRef;      /* SQLite internal use only */
    char                  *zErrMsg;  /* Return error message */
};
```

The module is also (eventually) responsible for deallocating this structure, so you can technically use whatever memory management routines you want. However, for maximum compatibility, it is strongly suggested that modules use sqlite3_malloc(). This will allows the module to run in any SQLite environment.

The only field of interest in the sqlite3_vtab structure is the zErrMsg field. This field allows a client to pass a custom error message back to the SQLite core if any of the functions (other than xCreate() or xConnect()) return an error code. The xCreate() and xConnect() functions return any potential error message through their sixth parameter, since they are unable to allocate and pass back a vtab structure (including the zErrMsg pointer) unless the call to xCreate() or xConnect() was successful. The xCreate() and xConnect() functions initialize the vtab error message pointer to NULL after allocating the vtab structure.

Typically, a virtual table needs to carry around a lot of state. In many systems, this is done with some kind of user-data pointer or other generic pointer. While the sqlite3_create_module() function does allow you to pass in a user-data pointer, that pointer is only made available to the xCreate() and xConnect() functions. Additionally, the same user-data pointer is provided to every table instance managed by a given module, so it isn't a good place to keep instance-specific data.

The standard way of providing instance-specific state data is to wrap and extend the sqlite3_vtab structure. For example, our dblist module will define a custom vtab data structure that looks like this:

```
typedef struct dblist_vtab_s {
    sqlite3_vtab         vtab;  /* this must go first */
    sqlite3              *db;   /* module-specific fields then follow */
} dblist_vtab;
```

By defining a custom data structure, the module can effectively extend the standard sqlite3_vtab structure with its own data. This will only work if the sqlite3_vtab structure is the first field, however. It must also be a vanilla C struct, and not a C++ class or some other managed object. Also, note that the vtab field is a full instance of the sqlite3_vtab structure, and not a pointer. That is, the custom vtab structure "contains-a" sqlite3_vtab structure, and does not "references-a" sqlite3_vtab structure.

It might seem a bit ugly to append module instance data to the sqlite3_vtab structure in this fashion, but this is how the virtual table interface is designed to work. In fact, this is the whole reason why the xCreate() and xConnect() functions are responsible for allocating the memory required by the sqlite3_vtab structure. By having the module allocating the memory, it can purposely overallocate the structure for its own means.

In the case of the dblist module, the only additional parameter the module requires is the database connection. Most modules require a lot more information. In specific, the xCreate() and xConnect() functions are the only time the module code will have access to the user-data pointer, the database connection pointer (the sqlite3 pointer), the database name, or the virtual table name. If the module needs access to these later, it needs to stash copies of this data in the vtab structure. The easiest way to make copies of these strings is with sqlite3_mprintf(). (See sqlite3_mprintf() in Appendix G.)

Most internal modules will need to make a copy of the database connection, the name of the database that contains the virtual module, and the virtual table name. Not only is this information required to create any shadow tables (which happens in xCreate()), but this information is also required to prepare any internal SQL statements (typically in xOpen()), as well as deal with xRename() calls. One of the most common bugs in module design is to assume there is only one database attached to the database connection, and that the virtual table is living in the `main` database. Be sure to test your module when virtual tables are created and manipulated in other databases that have been opened with `ATTACH DATABASE`.

The `dblist` module doesn't have any shadow tables, and the data we need to return is specific to the database connection, not any specific database. The module still needs to keep a copy of the database connection, so that it can prepare the `PRAGMA` statement, but that's it. As a result, the `dblist_vtab` structure is much simpler than most internal structures.

Define the table structure

The other major responsibility of the `xCreate()` and `xConnect()` functions is to define the structure of the virtual table.

`int sqlite3_declare_vtab( sqlite3 *db, const char *sql )`
: This function is used to declare the format of a virtual table. This function may only be called from inside a user-defined `xCreate()` or `xConnect()` function. The first parameter is the database connection passed into `xCreate()` or `xConnect()`. The second parameter is a string that should contain a single, properly formed `CREATE TABLE` statement.

Although the module must provide a table name in the `CREATE TABLE` statement, the table name (and database name, if provided) is ignored. The given name does not need to be the name of the virtual table instance. In addition, any constraints, default values, or key definitions within the table definition are also ignored—this includes any definition of an `INTEGER PRIMARY KEY` as a `ROWID` alias. The only parts of the `CREATE TABLE` statement that really matters are the column names and column types. Everything else is up to the virtual table module to enforce.

Like standard tables, virtual tables have an implied `ROWID` column that must be unique across all of the rows in the virtual table. Most of the virtual table operations reference rows by their `ROWID`, so a module will need some way to keep track of that value or generate a unique `ROWID` key value for every row the virtual table manages.

The `dblist` virtual table definition is quite simple, reflecting the same structure as the `PRAGMA database_list` output. Since the table structure is also completely static, the code can just define the SQL statement as a static string:

```
static const char *dblist_sql =
"CREATE TABLE dblist ( seq INTEGER, name TEXT, file TEXT );";

/* ... */
    sqlite3_declare_vtab( db, dblist_sql );
```

Depending on the design requirements, a module might need to dynamically build the table definition based off the user-provided `CREATE VIRTUAL TABLE` arguments.

It was already decided that the dblist example will simply use the seq values returned by the PRAGMA command as the source of both the seq output column and the ROWID values. Virtual tables have no implicit way of aliasing a standard column to the ROWID column, but a module is free to do this explicitly in the code.

It makes sense for a virtual table to define its own structure, rather than having it defined directly by the CREATE VIRTUAL TABLE statement. This allows the application to adapt to fit its own needs, and tends to greatly simplify the CREATE VIRTUAL TABLE statements. There is one drawback, however, in that if you want to look up the structure of a virtual table, you cannot simply look in the sqlite_master system table. Each virtual table instance will have an entry in this table, but the only thing you'll find there is the original CREATE VIRTUAL TABLE statement. If you want to look up the column names and types of a virtual table instance, you'll need to use the command PRAGMA table_info(*table_name*). This will provide a full list of all the column names and types in a table, even for a virtual table. See table_info in Appendix F for more details.

Storage initialization

If a virtual table module manages its own storage, the xCreate() function needs to allocate and initialize the required storage structure. In the case of an internal module that uses shadow tables, the module will need to create the appropriate tables. Only the xCreate() function needs to do this. The next time the database is opened, xConnect(), and not xCreate(), will be called. The xConnect() function may want to verify the correct shadow tables exist in the correct database, but it should not create them.

If you're writing an internal module that uses shadow tables, it is customary to name the shadow tables after the virtual table. In most cases you'll also want to be sure to create the shadow tables in the same database as the virtual table. For example, if your module requires three shadow tables per virtual table instance, such as Data, IndexA, and IndexB, a typical way to create the tables within your xCreate() function would be something like this (see sqlite3_mprintf() in Appendix G for details on the %w format):

```
sql_cmd1 = sqlite3_mprintf(
        "CREATE TABLE \"%w\".\"%w_Data\"   (...)", argv[1], argv[2],... );
sql_cmd2 = sqlite3_mprintf(
        "CREATE TABLE \"%w\".\"%w_IndexA\" (...)", argv[1], argv[2],... );
sql_cmd3 = sqlite3_mprintf(
        "CREATE TABLE \"%w\".\"%w_IndexB\" (...)", argv[1], argv[2],... );
```

This format will properly create per-instance shadow tables in the same database as the virtual table. The double quotes also ensure you can handle nonstandard identifier names. You should use a similar format (with a fully quoted database name and table name) in every SQL statement your module may generate.

If you're writing an external module that manages its own files, or something similar, you should try to follow some similar convention. Just remember *not* to use the database name (argv[1]) in your naming convention, as this can change, depending on how the database was open, or attached to the current database connection.

Create/connect dblist example

Although the `dblist` module could be considered an internal module, the module does not manage storage for any of the data it uses. This means there is no requirement to create shadow tables. This allows the module to use the same function for both the `xCreate()` and `xConnect()` function pointers.

Here is the full `dblist` create and connect function:

```
static int dblist_connect( sqlite3 *db, void *udp, int argc,
        const char *const *argv, sqlite3_vtab **vtab, char **errmsg )
{
    dblist_vtab    *v = NULL;

    *vtab = NULL;
    *errmsg = NULL;
    if ( argc != 3 )  return SQLITE_ERROR;
    if ( sqlite3_declare_vtab( db, dblist_sql ) != SQLITE_OK ) {
        return SQLITE_ERROR;
    }

    v = sqlite3_malloc( sizeof( dblist_vtab ) ); /* alloc our custom vtab */
    *vtab = (sqlite3_vtab*)v;
    if ( v == NULL ) return SQLITE_NOMEM;

    v->db = db;                                  /* stash this for later */
    (*vtab)->zErrMsg = NULL;                     /* initalize this */
    return SQLITE_OK;
}
```

The create/connect function walks through the required steps point by point. We verify the argument count (in this case, only allowing the standard three arguments), define the table structure, and finally allocate and initialize our custom `vtab` structure. Remember that you should not pass back an allocated `vtab` structure unless you're returning an `SQLITE_OK` status.

Disconnect and Destroy

Not surprisingly, the `xCreate()` and `xConnect()` functions each have their own counterparts:

`int xDisconnect( sqlite3_vtab *vtab )`
: Required. This is the counterpart to `xConnect()`. It is called every time a database that contains a virtual table is detached or closed. This function should clean up any process resources used by the virtual table implementation and release the `vtab` data structure.

`int xDestroy( sqlite3_vtab *vtab )`
: Required. This is the counterpart to `xCreate()`, and is called in response to a `DROP TABLE` command. If an internal module has created any shadow tables to store module data, this function should call `DROP TABLE` on those tables. As with

`xDisconnect()`, this function should also release any process resources and release the virtual table structure.

Many modules that do not manage their own storage use the same function for `xDisconnect()` and `xDestroy()`.

As with the `xCreate()` and `xConnect()` functions, only one of these functions will be called within the context of a given database connection. Both functions should release the memory allocated to the `vtab` pointer. The `xDestroy()` function should also delete, drop, or deallocate any storage used by the virtual table. Make sure you use fully qualified and quoted database and table names.

The `dblist` version of this function—which covers both `xDisconnect()` and `xDestroy()`—is very simple:

```
static int dblist_disconnect( sqlite3_vtab *vtab )
{
    sqlite3_free( vtab );
    return SQLITE_OK;
}
```

The code frees the `vtab` memory, and that's about it.

Query Optimization

Virtual tables present a challenge for the query optimizer. In order to optimize `SELECT` statements and choose the most efficient query plan, the optimizer must weigh a number of factors. In addition to understanding the constraints on the query (such as `WHERE` conditions), optimization also requires some understanding of how large a table is, what columns are indexed, and how the table can be sorted.

There is no automatic way for the query optimizer to deduce this information from a virtual table. Virtual tables cannot have traditional indexes, and if the internal virtual table implements a fancy custom indexing system, the optimizer has no way of knowing about it or how to best take advantage of it. While every query could perform a full table scan on a virtual table, that largely defeats the usefulness of many internal modules that are specifically designed to provide an optimized type of lookup.

The solution is to allow the query optimizer to ask the virtual table module questions about the cost and performance of different kinds of lookups. This is done through the `xBestIndex()` function:

`int xBestIndex( sqlite3_vtab *vtab, sqlite3_index_info *idxinfo )`

Required. When an SQL statement that references a virtual table is prepared, the query optimizer calls this function to gather information about the structure and capabilities of the virtual table. The optimizer is basically asking the virtual table a series of questions about the most efficient access patterns, indexing abilities, and natural ordering provided by the module. This function may be called several times when a statement is prepared.

Communication between the query optimizer and the virtual table is done through the `sqlite3_index_info` structure. This data structure contains an input and output section. The SQLite library fills out the input section (input to your function), essentially asking a series of questions. You can fill out the output section of the structure, providing answers and expense weightings to the optimizer.

If `xBestIndex()` sounds complicated, that's because it is. The good news is that if you ignore the optimizer, it will revert to a full table scan for all queries and perform any constraint checking on its own. In the case of the `dblist` module, we're going to take the easy way out, and more or less ignore the optimizer:

```
static int dblist_bestindex( sqlite3_vtab *vtab, sqlite3_index_info *info )
{
    return SQLITE_OK;
}
```

Given how simple the module is, and the fact that it will never return more than 30 rows (there is an internal limit on the number of attached databases), this is a fair trade off between performance and keeping the code simple. Even with fairly large datasets, SQLite does a pretty good job at processing full table scans with surprising speed.

Of course, not all modules—especially internal ones—can get away with this. Most larger modules should try to provide an intelligent response to the optimizer's questions. To see how this is done, we'll take a more in-depth look at this function later on in the chapter. See "Best Index and Filter" on page 262.

Custom Functions

As much as possible, a good module will attempt to make virtual table instances look and act exactly like standard tables. Functions like `xBestIndex()` help enforce that abstraction, so that virtual tables can interact with the optimizer to correctly produce more efficient lookups—especially in the case of an internal module trying to provide a better or faster indexing method.

There are a few other cases when SQLite needs a bit of help to hide the virtual table abstraction from the database user and other parts of the SQLite engine. SQL function calls and expression processing is one such area.

The `xFindFunction()` allows a module to override an existing function and provide its own implementation. In most cases, this is not needed (or even recommended). The major exceptions are the SQL functions `like()` and `match()`, which are used to implement the SQL expressions `LIKE` and `MATCH` (see Appendix D for more details).

A text-search engine is likely to override the `like()` and `match()` functions to provide an implementation that can directly access the search string and base its index optimization off the provided arguments. Without the ability to override these functions, it would be very difficult to optimize text searches, as the standard algorithm would require a full stable scan, extracting each row value and doing an external comparison.

`int xFindFunction( sqlite3_vtab *vtab, int arg, const char *func_name,`
`                   custom_function_ref, void **udp_ref)`

Optional. This function allows a module to override an existing function. It is called when preparing an SQL statement that uses a virtual table column as the first parameter in an SQL function (or the second, in the case of `like()`, `glob()`, `match()`, or `regexp()`). The first parameter is the `vtab` structure for this table instance. The second parameter indicates how many parameters are being passed to the SQL function, and the third parameter holds the name of the function. The fourth parameter is a reference to a scalar function pointer (see "Scalar Functions" on page 182), and the fifth parameter is a reference to a user-data pointer.

Using data from the first three parameters, a virtual table module needs to decide if it wants to override the existing function or not. If the module does not want to override the function, it should simply return zero. If the module does want to provide a custom function, it needs to set the function pointer reference (the fourth parameter) to the scalar function pointer of its choice and set the user-data pointer reference to a user-data pointer. The new function (and user-data pointer) will be called in the same context as the original function.

Most modules will not need to implement this function, and those that do should only need to override a few key functions. The `dblist` module does not provide an implementation.

Table Rename

Many modules, especially internal modules, key specific information off the name of the virtual table. This means that if the name of the virtual table is changed, the module needs to update any references to that name. This is done with the `xRename()` function.

`int xRename( sqlite3_vtab *vtab, const char *new_name )`

Required. This function is called in response to the SQL command `ALTER TABLE...RENAME`. The first parameter is the table instance being renamed, and the second parameter is the new table name.

In the case of an internal module, the most likely course of action is to rename any shadow tables to match the new name. Doing this properly will require knowing the original table name, as well as the database (`main`, `temp`, etc.), that was passed into `xCreate()` or `xConnect()`.

In the case of an external module, this function can usually just return `SQLITE_OK`, unless the table name has significance to some external data mapping.

As with any virtual table function that deals with table names, the module needs to properly qualify any SQL operation with a full database and table name, both properly quoted.

The dblist module has a very short xRename() function:

```
static int dblist_rename( sqlite3_vtab *vtab, const char *newname )
{
    return SQLITE_OK;
}
```

The dblist module does not use the table name for anything, so it can safely do nothing.

Opening and Closing Table Cursors

We will now look at the process of scanning a table and retrieving the rows and column values from the virtual table. This is done by opening a table cursor. The cursor holds all the state data required for the table scan, including SQL statements, file handles, and other data structures. After the cursor is created, it is used to step over each row in the virtual table and extract any required column values. When the module indicates that no more rows are available, the cursor is either reset or released. Virtual table cursors can only move forward through the rows of a table, but they can be reset back to the beginning for a new table scan.

A cursor is created using the xOpen() function and released with the xClose() function. Like the vtab structure, it is the responsibility of the module to allocate an sqlite3_vtab_cursor structure and return it back to the SQLite engine.

`int xOpen( sqlite3_vtab *vtab, sqlite3_vtab_cursor **cursor )`
: Required. This function must allocate, initialize, and return a cursor.

`int xClose( sqlite3_vtab_cursor *cursor )`
: Required. This function must clean up and release the cursor structure. Basically, it should undo everything done by xOpen().

The native sqlite3_vtab_cursor structure is fairly minimal, and looks like this:

```
struct sqlite3_vtab_cursor {
    sqlite3_vtab    *pVtab;  /* pointer to table instance */
};
```

As with the sqlite3_vtab structure, a module is expected to extend this structure with whatever data the module requires. The custom dblist cursor looks like this:

```
typedef struct dblist_cursor_s {
    sqlite3_vtab_cursor   cur;    /* this must go first */
    sqlite3_stmt          *stmt;  /* PRAGMA database_list statement */
    int                   eof;    /* EOF flag */
} dblist_cursor;
```

For the dblist module, the only cursor-specific data that is needed is an SQLite statement pointer and an EOF flag. The flag is used to indicate when the module has reached the end of the PRAGMA database_list output.

Outside of allocating the dblist_cursor, the only other task the dblist xOpen() function needs to do is prepare the PRAGMA SQL statement:

```
static int dblist_open( sqlite3_vtab *vtab, sqlite3_vtab_cursor **cur )
{
    dblist_vtab    *v = (dblist_vtab*)vtab;
    dblist_cursor  *c = NULL;
    int            rc = 0;

    c = sqlite3_malloc( sizeof( dblist_cursor ) );
    *cur = (sqlite3_vtab_cursor*)c;
    if ( c == NULL ) return SQLITE_NOMEM;

    rc = sqlite3_prepare_v2( v->db, "PRAGMA database_list", -1, &c->stmt, NULL );
    if ( rc != SQLITE_OK ) {
        *cur = NULL;
        sqlite3_free( c );
        return rc;
    }
    return SQLITE_OK;
}
```

As with xCreate() and xConnect(), no sqlite3_vtab_cursor should be allocated or passed back unless an SQLITE_OK is returned. There is no need to initialize the pVtab field of the cursor—SQLite will take care of that for us.

The dblist version of xClose() is very simple. The module must make sure the prepared statement is finalized before releasing the cursor structure:

```
static int dblist_close( sqlite3_vtab_cursor *cur )
{
    sqlite3_finalize( ((dblist_cursor*)cur)->stmt );
    sqlite3_free( cur );
    return SQLITE_OK;
}
```

You may be wondering why the module puts the statement pointer into the cursor. This requires the module to reprepare the PRAGMA statement for each cursor. Wouldn't it make more sense to put the statement pointer in the vtab structure? That way it could be prepared only once, and then reused for each cursor.

At first, that looks like an attractive option. It would be more efficient and, in most cases, work just fine—right up to the point were SQLite needs to create more than one cursor on the same table instance at the same time. Since the module depends on the statement structure to keep track of the position in the virtual table data (that is, the output of PRAGMA database_list), the module design needs each cursor to have its own statement. The easiest way to do this is simply to prepare and store the statement with the cursor, binding the statement lifetime to the cursor lifetime.

Filtering Rows

The `xFilter()` function works in conjunction with the `xBestIndex()` function, providing the SQLite query engine a means to communicate any specific constraints or conditions put on the query. The `xBestIndex()` function is used by the query optimizer to ask the module questions about different lookup patterns or limits. Once SQLite decides what to do, the `xFilter()` function is used to tell the module which plan of action is being taken for this particular scan.

```
int xFilter( sqlite3_vtab_cursor *cursor,
             int idx_num, const char *idx_str,
             int argc, sqlite3_value **argv )
```

Required. This function is used to reset a cursor and initiate a new table scan. SQLite will communicate any constraints that have been placed on the current cursor. The module may choose to skip over any rows that do not meet these constraints. All of the parameters are determined by actions taken by the `xBestIndex()` function. The first row of data must also be fetched.

The idea is to allow the module to "pre-filter" as many rows as it can. Each time the SQLite library asks the module to advanced the table cursor to the next row, the module can use the information provided to the `xFilter()` function to skip over any rows that do not meet the stated criteria for this table scan.

The `xBestIndex()` and `xFilter()` functions can also work together to specify a specific row ordering. Normally, SQLite makes no assumptions about the order of the rows returned by a virtual table, but `xBestIndex()` can be used indicate the ability to support one or more specific orderings. If one of those orderings is passed into `xFilter()`, the table is required to return rows in the specified order.

To get any use out of the `xFilter()` function, a module must also have a fully implemented `xBestIndex()` function. The `xBestIndex()` function sets up the data that is passed to the `xFilter()` function. Most of the data passed into `xFilter()` has no specific meaning to SQLite, it is simply based off code agreements between `xBestIndex()` and `xFilter()`.

Implementing all this can be quite cumbersome. Thankfully, as with `xBestIndex()`, it is perfectly valid for a module to ignore the filtering system. If a user query applies a set of conditions on the rows it wants returned from a virtual table, but the module does not filter those out, the SQLite engine will be sure to do it for us. This greatly simplifies the module code, but with the trade-off that any operation against that module turns into a full table scan.

Full table scans may be acceptable for many types of external modules, but if you're developing a customized index system, you have little choice but to tackle writing robust `xBestIndex()` and `xFilter()` functions. To get a better idea on how to do this, see "Best Index and Filter" on page 262.

Even if the actual filtering process is ignored, the `xFilter()` function is still required to do two important tasks. First, it must reset the cursor and prepare it for a new table scan. Second, `xFilter()` is responsible for fetching the first row of output data. Since the `dblist` module doesn't utilize the filtering system, these are pretty much the only things the `xFilter()` function ends up doing:

```
static int dblist_filter( sqlite3_vtab_cursor *cur,
            int idxnum, const char *idxstr,
            int argc, sqlite3_value **value )
{
    dblist_cursor  *c = (dblist_cursor*)cur;
    int            rc = 0;

    rc = sqlite3_reset( c->stmt );     /* start a new scan */
    if ( rc != SQLITE_OK ) return rc;
    c->eof = 0;                        /* clear EOF flag */

    dblist_get_row( c );               /* fetch first row */
    return SQLITE_OK;
}
```

Although the `dblist` module does not utilize the `xBestIndex()` data, there are still important things to do. The `xFilter()` function must first reset the statement. This "rewinds" the pragma statement, putting our cursor at the head of the table. There are situations where `sqlite3_reset()` may be called on a freshly prepared (or freshly reset) statement, but that is not a problem. There are other calling sequences which may require `xFilter()` to reset the statement.

Because both `xFilter()` and the `xNext()` function (which we'll look at next) need to fetch row data, we've broken that out into its own function:

```
static int dblist_get_row( dblist_cursor *c )
{
    int  rc;

    if ( c->eof ) return SQLITE_OK;
    rc = sqlite3_step( c->stmt );
    if ( rc == SQLITE_ROW ) return SQLITE_OK;     /* we have a valid row */

    sqlite3_reset( c->stmt );
    c->eof = 1;
    return ( rc == SQLITE_DONE ? SQLITE_OK : rc );  /* DONE -> OK */
}
```

The main thing this function does is call `sqlite3_step()` on the cursor's SQL statement. If the module gets a valid row of data (`SQLITE_ROW`), everything is good. If the module gets anything else (including `SQLITE_DONE`) it considers the scan done. In that case, the module resets the statement before returning `SQLITE_OK` (if it got to the end of the table) or the error. Although the module could wait for `xFilter()` to reset the statement or `xClose()` to finalize it, it is best to reset the statement as soon as we know we've reached the end of the available data.

Extracting and Returning Data

We now, finally, get to the core of any virtual table implementation. Once the module has a valid cursor, it needs to be able to advanced that cursor over the virtual table data and return column values. This core set of four functions is used to do just that:

`int xNext( sqlite3_vtab_cursor *cursor )`

Required. This function is used to advance the cursor to the next row. When the SQLite engine no longer needs data from the current row, this is called to advance the virtual table scan to the next row. If a virtual table is already on the last row of the table when `xNext()` is called, it should *not* return an error.

Note that `xNext()` truly is a "next" function and not a "get row" function. It is not called to fetch the first row of data. The first row of data should be fetched and made available by the `xFilter()` function.

If the module is filtering rows via `xBestIndex()` and `xFilter()`, it is legitimate for `xNext()` to skip over any rows in the virtual table that do not meet the conditions put forth to `xFilter()`. Additionally, if `xBestIndex()` indicated an ability to return the data in a specific order, `xNext()` is obligated to do so. Otherwise, `xNext()` may return rows in any order it wishes, so long as they are all returned.

`int xEof( sqlite3_vtab_cursor *cursor )`

Required. This function is used to determine if the virtual table has reached the end of the table. Every call to `xFilter()` and `xNext()` will immediately be followed by a call to `xEof()`. If the previous call to `xNext()` advanced the cursor past the end of the table, `xEof()` should return a true (nonzero) value, indicating that the end of the table has been reached. If the cursor still points to a valid table row, `xEof()` should return false (zero).

`xEof()` is also called right after `xFilter()`. If a table is empty or will return no rows under the conditions defined by `xFilter()`, then `xEof()` needs to return true at this time.

There is no guarantee `xNext()` will keep being called until `xEof()` returns true. The query may decide to terminate the table scan at any time.

`int xRowid( sqlite3_vtab_cursor *cursor, sqlite_int64 *rowid )`

Required. This function is used to retrieve the `ROWID` value of the current row. The `ROWID` value should be passed back through the `rowid` reference provided as the second parameter.

`int xColumn( sqlite3_vtab_cursor *cursor, sqlite3_context *ctx, int cidx )`

Required. This function is used to extract column values from the cursor's current row. The parameters include the virtual table cursor, an `sqlite3_context` structure, and a column index. Values should be returned using the `sqlite3_context` and the `sqlite3_result_xxx()` functions. The column index is zero-based, so the first column defined in the virtual table definition will have a column index of zero. This function is typically called multiple times between calls to `xNext()`.

The first two functions, xNext() and xEof(), are used to advance a cursor through the virtual table data. A cursor can only be advanced through the data, it cannot be asked to back up, save for a full reset back to the beginning of the table. Unless xBest Index() and xFilter() agreed on a specific filtering or ordering, xNext() is under no obligation to present the data in a specific order. The only requirement is that continuous calls to xNext() will eventually visit each row exactly once.

At each row, xRowid() and xColumn() can be used to extract values from the current row. xRowid() is used to extract the virtual ROWID value, while xColumn() is used to extract values from all the other columns. While a cursor is at a specific row, the xRowid() and xColumn() functions may be called any number of times in any order.

Since the dblist module depends on executing the PRAGMA statement to return data, most of these functions are extremely simple. For example, the dblist xNext() function calls the dblist_get_row() function, which in turn calls sqlite3_step() on the cursor's statement:

```
static int dblist_next( sqlite3_vtab_cursor *cur )
{
    return dblist_get_row( (dblist_cursor*)cur );
}
```

The dblist xEof() function returns the cursor EOF flag. This flag is set by dblist_get_row() when the module reaches the end of the PRAGMA database_list data. The flag is simply returned:

```
static int dblist_eof( sqlite3_vtab_cursor *cur )
{
    return ((dblist_cursor*)cur)->eof;
}
```

The data extraction functions for the dblist module are also extremely simple. The dblist module uses the seq column from the PRAGMA database_list output as its virtual ROWID. This means that it can return the value of the seq column as our ROWID. As it happens, the seq column is the first column, so it has an index of zero:

```
static int dblist_rowid( sqlite3_vtab_cursor *cur, sqlite3_int64 *rowid )
{
    *rowid = sqlite3_column_int64( ((dblist_cursor*)cur)->stmt, 0 );
    return SQLITE_OK;
}
```

The xColumn() function is nearly as simple. Since there is a one-to-one mapping between the output columns of the PRAGMA statement and the dblist virtual table columns, the module can extract values directly from the PRAGMA output and pass them back as column values for our virtual table:

```
static int dblist_column( sqlite3_vtab_cursor *cur, sqlite3_context *ctx, int cidx )
{
    dblist_cursor   *c = (dblist_cursor*)cur;
    sqlite3_result_value( ctx, sqlite3_column_value( c->stmt, cidx ) );
```

```
        return SQLITE_OK;
    }
```

In most cases, these functions would be considerably more complex than what the dblist module has here. The fact that the dblist module depends on only a single SQL command to return all of the required data makes the design of these functions quite simple—even more so, since the output of the SQL command exactly matches the data format we need.

To get a better idea of what a more typical module might look like, have a look at the implementation of these functions in the other example module ("Advanced Example: weblog Module" on page 246).

Virtual Table Modifications

As with any other table, modules support the ability to make modifications to a virtual table using the standard `INSERT`, `UPDATE`, and `DELETE` commands. All three operations are supported by the `xUpdate()` function. This is a table-level function that operates on a table instance, not a cursor.

```
int xUpdate( sqlite3_vtab *vtab,
             int argc, sqlite3_value **argv,
             sqlite_int64 *rowid )
```

Optional. This call is used to support all virtual table modifications. It will be called in response to any `INSERT`, `UPDATE`, or `DELETE` command. The first parameter is the table instance. The second and third parameters pass in a series of database values. The fourth parameter is a reference to a `ROWID` value, and is used to pass back the newly defined `ROWID` when a new row is inserted.

The `argv` parameter will have a valid `sqlite3_value` structure for each argument, although some of those values may have the type `SQLITE_NULL`. Rows are always inserted or updated as whole sets. Even if the SQL `UPDATE` command only updates a single column of a row, the `xUpdate()` command will always be provided with a value for every column in a row.

If only a single argument is provided, this is a `DELETE` request. The sole argument (`argv[0]`) will be an `SQLITE_INTEGER` that holds the `ROWID` of the row that needs to be deleted.

In all other cases, exactly *n*+2 arguments will be provided, where *n* is the number of columns, including `HIDDEN` ones (see "Create and Connect" on page 248) in the table definition. The first argument (`argv[0]`) is used to refer to existing `ROWID` values, while the second (`argv[1]`) is used to refer to new `ROWID` values. These two arguments will be followed by a value for each column in a row, starting with `argv[2]`. Essentially, the arguments `argv[1]` through `argv[n+1]` represent a whole set of row values starting with the implied `ROWID` column followed by all of the declared columns.

If `argv[0]` has the type `SQLITE_NULL`, this is an `INSERT` request. If the `INSERT` statement provided an explicit `ROWID` value, that value will be in `argv[1]` as an `SQLITE_INTEGER`. The module should verify the `ROWID` is appropriate and unique before using it. If no explicit `ROWID` value was given, `argv[1]` will have a type of `SQLITE_NULL`. In this case, the module should assign an unused `ROWID` value and pass it back via the `rowid` reference pointer in the `xUpdate()` parameters.

If `argv[0]` has the type `SQLITE_INTEGER`, this is an `UPDATE`. In this case, both `argv[0]` and `argv[1]` will be `SQLITE_INTEGER` types with `ROWID` values. The existing row indicated in `argv[0]` should be updated with the values supplied in `argv[1]` through `argv[n+1]`. In most cases, `argv[0]` will be the same as `argv[1]`, indicating no change in the `ROWID` value. However, if the `UPDATE` statement includes an explicit update to the `ROWID` column, it may be the case that the first two arguments do not match. In that case, the row indicated by `argv[0]` should have its `ROWID` value changed to `argv[1]`. In either case, all the other columns should be updated with the additional arguments.

It is the module's responsibility to enforce any constraints or typing requirements on the incoming data. If the data is invalid or otherwise cannot be inserted or updated into the virtual table, `xUpdate()` should return an appropriate error, such as `SQLITE_CONSTRAINT` (constraint violation).

There may be times when a modification (including a DELETE) happens while an active cursor is positioned at the row in question. The module design must be able to handle this situation.

The `xUpdate()` function is optional. If no implementation is provided, all virtual table instances provided by this module will be read-only. That is the case with our dblist module, so there is no implementation for `xUpdate()`.

Cursor Sequence

Most cursor functions have very specific tasks. Some of these, like `xEof()`, are typically very small, while others, like `xNext()`, can be quite complex. Regardless of size or complexity, they all need to work together to perform the proper tasks and maintain the state of the `sqlite3_vtab_cursor` structure, including any extensions your module might require.

In order to maintain the proper state, it is important to understand which functions can be called, and when. Figure 10-1 provides a sequence map of when cursor functions can be called. Your module needs to be prepared to properly deal with any of these transitions.

Some of the call sequences can catch people by surprise, such as having `xClose()` called before `xEof()` returns true. This might happen if an SQL query has a `LIMIT` clause. Also, it is possible for `xRowid()` to be called multiple times between calls to `xNext()`. Similarly, `xColumn()` may be called multiple times with the same column index between calls to xNext(). It is also possible that neither xRowid() nor xColumn() (nor both) may be called at all between calls to xNext().

In addition to cursor functions, some table-level functions may also be called throughout this sequence. In specific, xUpdate() may be called at any time, possibly altering the row a cursor is currently processing. Generally, this happens by having an update statement open a cursor, find the row it is looking to modify, and then call xUpdate() outside of the cursor context.

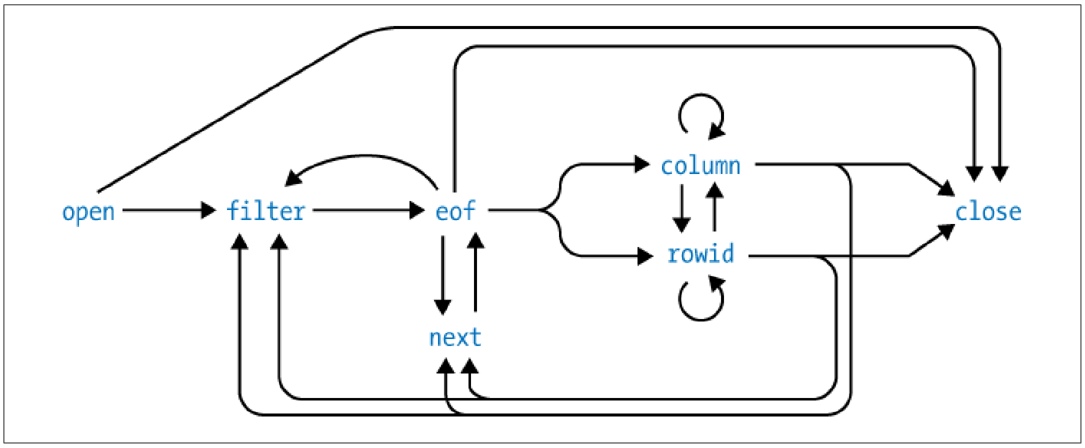


Figure 10-1. The lifespan of a virtual table cursor. This shows the possible calling sequences for the cursor functions of a virtual table module.

It can be tricky to test your module and confirm that everything is working properly. The only advice I can offer is to test your module with a known and relatively small set of data, running it through as many query types as possible. Try to include different variations of GROUP BY, ORDER BY, and any number of join operations (including self-joins). When you're first starting to write a module, it might also help to put simple printf() or other debug statements at the top of each function. This will assist in understanding the call patterns.

Transaction Control

Like any other database element, virtual tables are expected to be aware of database transactions and support them appropriately. This is done through four optional functions. These functions are table-level functions, not cursor-level functions.

int xBegin(sqlite3_vtab *vtab)
: Optional. Indicates the start of a transaction involving the virtual table. Any return code other than SQLITE_OK will cause the transaction to fail.

int xSync(sqlite3_vtab *vtab)
: Optional. Indicates the start of a transactional commit that involves the virtual table. Any return code other than SQLITE_OK will cause the whole transaction to automatically be rolled back.

`int xCommit( sqlite3_vtab *vtab )`
: Optional. Indicates the finalization of a transactional commit that involves the virtual table. The return code is ignored—if `xSync()` succeeded, this function must succeed.

`int xRollback( sqlite3_vtab *vtab )`
: Optional. Indicates that a transaction involving the virtual table is being rolled back. The module should revert its state to whatever state it was in prior to the call to `xBegin()`. The return code is ignored.

These functions are optional and are normally only required by external modules that provide write capabilities to external data sources. Internal modules that record their data into standard tables are covered by the existing transaction engine (which will automatically begin, commit, or roll back under the control of the user SQL session). Modules that are limited to read-only functionality do not need transactional control, since they are not making any modifications.

Internal modules (modules that store all their data in shadow database tables) do *not* need to implement transaction control functions. The existing, built-in transaction system will automatically be applied to any changes made to standard database tables.

If you do need to support your own transactions, it is important to keep the program flow in mind. `xBegin()` will always be the first function to be called.[*] Typically, there will be calls to `xUpdate()` followed by a two-step sequence of calls to `xSync()` and `xCommit()` to close and commit the transaction. Once `xBegin()` has been called, it is also possible to get a call to `xRollback()` to roll the transaction back. The `xRollback()` function can also be called after `xSync()` (but before `xCommit()`) if the sync step fails.

Full transactions do not nest, and virtual tables do not support save-points. Once a call to `xBegin()` has been made, you will not get another one until either `xCommit()` or `xRollback()` has been called.

In keeping with the ACID properties of a transaction, any modifications made between calls to `xBegin()` and `xSync()` should only be visible to this virtual table instance in this database connection (that is, this `vtab` structure). This can be done by delaying any writes or modifications to external data sources until the `xSync()` function is called, or by somehow locking the data source to ensure other module instances (or other applications) cannot modify or access the data. If the data is written out in `xSync()`, the data source still needs to be locked until a call to `xCommit()` or `xRollback()` is made. If `xSync()` returns `SQLITE_OK`, it is assumed that any call to `xCommit()` will succeed, so you want to try to make your modifications in `xSync()` and verify and release them in `xCommit()`.

[*] In theory. Currently, calls are made directly to `xSync()` and `xCommit()` following the call to `xCreate()`. It isn't clear if this is considered a bug or not, so this behavior may change in future versions of SQLite.

Proper transactional control is extremely hard, and making your transactions fully atomic, consistent, isolated, and durable is no small task. Most external modules that attempt to implement transactions do so by locking the external data source. You still need to support some type of rollback ability, but exclusive access eliminates any isolation concerns.

Since the `dblist` module is read-only, it does not need to provide any transactional functions.

Register the Module

Now that we've had a look at all the functions required to define a module, we need to register them. As we've already seen, this is done with the `sqlite3_create_module()` function. To register the module, we need to fill out an `sqlite3_module` structure and pass that to the create function.

You may have noticed that all of our module functions were marked static. This is because the module was written as an extension (see the section "SQLite Extensions" on page 204). By structuring the code that way, we can easily build our virtual table module into an application, or we can create a dynamic extension.

Here is the initialization function for our extension:

```
static sqlite3_module dblist_mod = {
    1,                   /* iVersion        */
    dblist_connect,      /* xCreate()       */
    dblist_connect,      /* xConnect()      */
    dblist_bestindex,    /* xBestIndex()    */
    dblist_disconnect,   /* xDisconnect()   */
    dblist_disconnect,   /* xDestroy()      */
    dblist_open,         /* xOpen()         */
    dblist_close,        /* xClose()        */
    dblist_filter,       /* xFilter()       */
    dblist_next,         /* xNext()         */
    dblist_eof,          /* xEof()          */
    dblist_column,       /* xColumn()       */
    dblist_rowid,        /* xRowid()        */
    NULL,                /* xUpdate()       */
    NULL,                /* xBegin()        */
    NULL,                /* xSync()         */
    NULL,                /* xCommit()       */
    NULL,                /* xRollback()     */
    NULL,                /* xFindFunction() */
    dblist_rename        /* xRename()       */
};

int dblist_init( sqlite3 *db, char **error, const sqlite3_api_routines *api )
{
    int   rc;
    SQLITE_EXTENSION_INIT2(api);
    /* register module */
    rc = sqlite3_create_module( db, "dblist", &dblist_mod, NULL );
    if ( rc != SQLITE_OK ) {
        return rc;
    }

    /* automatically create an instance of the virtual table */
    rc = sqlite3_exec( db,
        "CREATE VIRTUAL TABLE temp.sql_database_list USING dblist",
        NULL, NULL, NULL  );
    return rc;
}
```

The most important thing to notice is that the `sqlite3_module` structure is given a static allocation. The SQLite library does not make a copy of this structure when the module is registered, so the `sqlite3_module` structure must remain valid for the duration of the database connection. In this case, we use a file-level global that is statically initialized with all the correct values.

The extension entry point function is a bit unique, in that it not only defines the module, but it also goes ahead and creates an instance of a dblist virtual table. Normally, an extension initialization function wouldn't (and shouldn't) do something like this, but in this case it makes sense. Like any other table, virtual tables are typically bound to a specific database. But the active database list we get from the `PRAGMA database_list` command is a function of the current state of the database connection (and all attached databases), and isn't really specific to a single database. If you were to create a dblist table in multiple databases that were all attached to the same database connection, they would all return the same data. It is the database connection (and not a specific database) that is the real source of the data.

So, in the somewhat unique case of the dblist module, we only need one instance of the virtual table per database connection. Ideally, it would always be there, no matter which databases are attached. It would also be best if the table wasn't "left behind" in a database file after that database was closed or detached. Not only would this tie the database file to our module, it is also unnecessary since a dblist table instance doesn't have any state beyond the database connection.

To meet all these needs, the module goes ahead and just makes a single instance of the table in the temporary database. Every database connection has a temporary database, and it is always named `temp`. This makes the table instance easy to find. Creating it in the `temp` database also keeps the table instance out of any "real" database files, and ties the lifetime of the table to the lifetime of the database connection. All in all, it is a perfect, though somewhat unusual, fit for this specific module.

The end result is that if you load the dblist extension, it will not only register the dblist module, it will also create an instance of the dblist virtual table at `temp.sql_database_list`. System tables in SQLite have the prefix `sqlite_`, but those names are reserved and the extension cannot create a table with that prefix. The name `sql_database_list` gets the idea across, however.

Example Usage

After all that work, what do we get? First, lets have a look at what the `PRAGMA database_list` does by itself. Here is some example output:

```
sqlite> PRAGMA database_list;
seq        name       file
---------- ---------- ----------------------------
0          main       /Users/jak/sqlite/db1.sqlite3
1          temp
2          memory
3          two        /Users/jak/sqlite/db2.sqlite3
```

In this case, I ran the `sqlite3` utility with the file `db1.sqlite3`, created an empty temporary table (so the `temp` database shows up), attached an in-memory database as `memory`, and finally attached a second database file as `two`.

Now let's load our module extension and see what we get:

```
sqlite> .load dblist.sqlite3ext dblist_init
sqlite> SELECT * FROM sqlite3_database_list;
seq        name       file
---------- ---------- ----------------------------
0          main       /Users/jak/sqlite/db1.sqlite3
1          temp
2          memory
3          two        /Users/jak/sqlite/db2.sqlite3
```

And we get...the exact same thing! Actually, that's a good thing—that was the whole point. The key thing is that, unlike the `PRAGMA` command, we can do this:

```
sqlite> SELECT * FROM sqlite3_database_list WHERE file == '';
seq        name       file
---------- ---------- ----------------------------
1          temp
2          memory
```

This shows us all of the databases that have no associated filename. You'll note that `PRAGMA database_list` (and hence the dblist module) returns an empty string, and not a NULL, for a database that does not have an associated database file.

Perhaps most useful, we can also make queries like this to figure out what the logical database name is for a particular database file:

```
sqlite> SELECT name FROM sqlite3_database_list
   ...>   WHERE file LIKE '%db2.sqlite3';
name
----------
two
```

I'd be the first to admit this isn't exactly ground-breaking work. Parsing the direct output of `PRAGMA database_list` isn't that big of a deal—for a program or for a human. The main point of this example wasn't to show the full power of virtual tables, but to give us a problem to work with where we could focus on the functions and interface required by the virtual table system, rather than focusing on the complex code required to implement it.

Now that you've seen the module interface and have a basic idea of how things work, we're going to shift our focus to something a bit more practical, and a bit more complex.

Advanced Example: weblog Module

Now that we've seen a very simple example of a virtual table module, you should have some idea of how they work. Although our dblist module was a good introduction to how virtual tables operate, it isn't a very representative example.

To provide a more advanced and realistic example, we're going to look at a second example module. This module is known as *weblog*, and is designed to parse Apache *httpd* server logs and present them to the database engine as a virtual table. It will parse the default Apache *combine* or *common* logfile formats, or any other logfile that matches this format. Apache logfiles are cross-platform and reasonably common. Many people have access to logfiles with a decent amount of interesting data, allowing this example to be a bit more hands-on.

Be warned that some of the code explanations will be a bit brief. Although the functions are larger, much of the code involves rather basic housekeeping-type tasks, such as string scanning. Rather than focus on these parts, most of the descriptions will focus on how the code interacts with the SQLite library. Many of the housekeeping details will be, as they say, left as an exercise for the reader.

The weblog module is designed as an external read-only module. The module gets all of its data directly from a web server logfile, making it dependent on external resources to provide data. The module does not let you modify those data sources, however.

A weblog virtual table would be created with an SQL command like this:

```
CREATE VIRTUAL TABLE current USING weblog( /var/log/httpd/access.log );
```

Notice that the filename has no single quotes and is not a string literal. Table parameters include everything between the commas (of which we have none, since there is only one argument), so if you need to reference a file with spaces, you can do something like this:

```
CREATE VIRTUAL TABLE log USING weblog( /var/log/httpd/access log file.txt );
```

The first example will create a table instance `current`, and bind it to the data found in the `/var/log/httpd/access.log` file. The second example will bind the SQL table `log` to the file `access log file.txt` in the same directory.

Briefly, the Apache *common* log format contains seven fields. The first field is the IP address of the client. In rare situations this might be a hostname, but most servers are configured to simply record the IP address in dot-decimal format. The second field is a legacy ident field. Most web servers do not support this, and record only a single dash.

The third field records the username, if given. If not given, this field is also recorded as a single dash. The fourth field is a timestamp, surrounded by square brackets (`[ ]`). The fifth is the first line of the HTTP request, in double quotes. This contains the HTTP operation (such as `GET` or `POST`) as well as the URL. In the sixth column is the HTTP result code (e.g., 200 for OK, 404 for missing resource), with the number of payload bytes returned in the seventh field.

The *combine* file format adds two more fields. The eighth field is the referrer header, which contains a URL. The ninth field is the user-agent header, also in double quotes.

Count	Logfile field	Meaning
1	Client Address	IP or hostname of HTTP client
2	Ident	Legacy field, not used
3	Username	Client-provided username
4	Timestamp	Time of transaction
5	HTTP Request	HTTP operation and URL
6	Result Code	Result status of HTTP request
7	Bytes	Payload bytes
8	Referrer	URL of referrer page
9	User Agent	Client software identifier

The weblog module is designed to read the *combine* file format. However, if given a *common* logfile that lacks the last two fields, these extra fields will simply be NULL.

Although the logfile has seven or nine columns, the weblog virtual table will have more than nine columns. The virtual table adds a number of additional columns that present the same data in different ways.

For example, the IP address will be returned in one column as a text value that holds the traditional dotted notation. Another column will provide a raw integer representation. The text column is easier for humans to understand, but the integer column allows for faster searches, especially over ranges. The underlying data is the same: the two columns just return the data in different formats. Similarly, the timestamp column can return the string value from the logfile, or it can return separate integer values for the year, month, day, etc.

If this were a fully supported SQLite extension, it would likely include more than just the weblog module. Ideally, it would also include a number of utility functions, such as a function that converted text values containing dot-decimal IP addresses to and from integer values. (Then again, if this were a fully supported module, it would include decent error messages and other polish that this example lacks. I'm trying to keep the line counts as small as possible.) Some of these functions would reduce the need for extra columns, since you could just convert the data using SQL, but there are still times when having the extra columns is extremely useful.

Create and Connect

Since the weblog module is an external module, there isn't any data to initialize. This means that, like the dblist, we can use the same function for both `xCreate()` and `xConnect()`.

Before we get into the function, let's have a quick look at our augmented vtab structure. Since this module does not use the table name for anything, the only data we need to keep around is the logfile filename:

```
typedef struct weblog_vtab_s {
    sqlite3_vtab   vtab;
    char           *filename;
} weblog_vtab;
```

The weblog create/connect function is a bit longer than the dblist version, but still fairly easy to follow. First, it verifies that we have exactly four arguments. Remember that the first three arguments are always the module name, the database name, and the table name. The fourth argument is the first user-provided argument, which in this case is the log filename. The function tries to open that file for read-only access, just to verify the file is there and can be opened it for reading. This test isn't foolproof, but it is a nice check. The module then allocates the vtab structure, stashes a copy of the filename, and declares the table definition:

```
static int weblog_connect( sqlite3 *db, void *udp, int argc,
        const char *const *argv, sqlite3_vtab **vtab, char **errmsg )
{
    weblog_vtab  *v = NULL;
    const char   *filename = argv[3];
    FILE         *ftest;

    if ( argc != 4 ) return SQLITE_ERROR;

    *vtab = NULL;
    *errmsg = NULL;

    /* test to see if filename is valid */
    ftest = fopen( filename, "r" );
    if ( ftest == NULL ) return SQLITE_ERROR;
    fclose( ftest );

    /* allocate structure and set data */
    v = sqlite3_malloc( sizeof( weblog_vtab ) );
    if ( v == NULL ) return SQLITE_NOMEM;
    ((sqlite3_vtab*)v)->zErrMsg = NULL; /* need to init this */

    v->filename = sqlite3_mprintf( "%s", filename );
    if ( v->filename == NULL ) {
        sqlite3_free( v );
        return SQLITE_NOMEM;
    }
    v->db = db;
    sqlite3_declare_vtab( db, weblog_sql );
    *vtab = (sqlite3_vtab*)v;
    return SQLITE_OK;
}
```

The table definition contains 20 columns total. The first 9 map directly to the fields within the logfile, while the extra 11 columns provide different representations of the same data. The last column represents the whole line of the logfile, without modifications:

```
const static char *weblog_sql =
"    CREATE TABLE weblog (           "
"        ip_str       TEXT,          "  /*  0 */
"        login        TEXT HIDDEN,   "  /*  1 */
"        user         TEXT,          "  /*  2 */
"        time_str     TEXT,          "  /*  3 */
"        req          TEXT,          "  /*  4 */
"        result       INTEGER,       "  /*  5 */
"        bytes        INTEGER,       "  /*  6 */
"        ref          TEXT,          "  /*  7 */
"        agent        TEXT,          "  /*  8 */
#define TABLE_COLS_SCAN                 9
"        ip_int       INTEGER,       "  /*  9 */
"        time_day     INTEGER,       "  /* 10 */
"        time_mon_s   TEXT,          "  /* 11 */
"        time_mon     INTEGER,       "  /* 12 */
"        time_year    INTEGER,       "  /* 13 */
"        time_hour    INTEGER,       "  /* 14 */
"        time_min     INTEGER,       "  /* 15 */
"        time_sec     INTEGER,       "  /* 16 */
"        req_op       TEXT,          "  /* 17 */
"        req_url      TEXT,          "  /* 18 */
"        line         TEXT HIDDEN    "  /* 19 */
"     );                             ";
#define TABLE_COLS                     20
```

You may have noticed a few of the columns have the keyword `HIDDEN`. This keyword is only valid for virtual table definitions. Any column marked `HIDDEN` will not be returned by `SELECT * FROM...` style queries. You can explicitly request the column, but it is not returned by default. This is very similar in behavior to the `ROWID` column found in standard tables. In our case, we've marked the `login` and `line` columns as `HIDDEN`. The `login` column almost never contains valid data, while the `line` column is redundant (and large). The columns are there if you need them, but in most cases people aren't interested in seeing them. To keep the general output cleaner, I've chosen to hide them.

Disconnect and Destroy

As with `xConnect()` and `xCreate()`, the weblog `xDisconnect()` and `xDestroy()` functions share the same implementation:

```
static int weblog_disconnect( sqlite3_vtab *vtab )
{
    sqlite3_free( ((weblog_vtab*)vtab)->filename );
    sqlite3_free( vtab );
    return SQLITE_OK;
}
```

Free up the memory used for the filename, free up the memory used by the vtab structure, and return. Simple and easy.

Other Table Functions

The last set of table-level functions includes `xBestIndex()`, `xFindFunction()`, `xRename()`, and `xUpdate()`, as well as the four transactional functions, `xBegin()`, `xSync()`, `xCommit()`, and `xRollback()`. The `xFindFunction()` is optional, and the weblog module has no use for it, so there is no implementation of this function. Since this is a read-only module, same is true of `xUpdate()`. Similarly, the transactional functions are also optional and not required for read-only modules. For table-level functions, that leaves only `xRename()` and `xBestIndex()`.

The `xRename()` function is required, but since the module makes no use of the virtual table instance name, it is basically a no-op:

```
static int weblog_rename( sqlite3_vtab *vtab, const char *newname )
{
    return SQLITE_OK;
}
```

In the case of the weblog module, once you set the name of the external logfile when creating a virtual table, there is no way to alter it, other than dropping and re-creating the table.

The last function, `xBestIndex()`, is required, but it isn't actually returning any useful data:

```
static int weblog_bestindex( sqlite3_vtab *vtab, sqlite3_index_info *info )
{
    return SQLITE_OK;
}
```

Since the module has no indexing system, it can't offer any optimized search patterns. The logfile is always scanned start to finish anyway, so every query is a full table scan.

Open and Close

We can now move on to the cursor functions. The first thing to look at is the weblog cursor structure. The weblog cursor is a bit more complex than the dblist example, as it needs to read and scan the data values from the logfile.

There are three basic sections to this structure. The first is the base `sqlite3_vtab_cursor` structure. As always, this must come first, and must be a full instance of the structure:

```
#define LINESIZE 4096

typedef struct weblog_cursor_s {
    sqlite3_vtab_cursor   cur;              /* this must be first */

    FILE          *fptr;                    /* used to scan file */
    sqlite_int64  row;                      /* current row count (ROWID) */
    int           eof;                      /* EOF flag */

    /* per-line info */
    char          line[LINESIZE];           /* line buffer */
    int           line_len;                 /* length of data in buffer */
    int           line_ptrs_valid;          /* flag for scan data */
    char          *(line_ptrs[TABLE_COLS]); /* array of pointers */
    int           line_size[TABLE_COLS];    /* length of data for each pointer */
} weblog_cursor;
```

The second block deals with the data we need to scan the logfile. The weblog module uses the standard C library `f` functions (such as `fopen()`) to open and scan the logfile. Each weblog cursor needs a unique `FILE` pointer, just as each dblist cursor required a unique statement structure. The module uses the `FILE` structure to keep track of its location within the file, so each cursor needs its own unique `FILE` structure. The cursor needs to keep track of the number of lines it has read from the file, as this value is used as the `ROWID`. Finally, the cursor needs an EOF flag to indicate when it has reached the end of the file.

Having a unique `FILE` pointer for each cursor means the module needs to reopen the file for each table scan. In the case of the weblog module, this is actually an advantage, as each table scan will reassociate itself with the correct file. This can be important in a web server environment, where logfiles may roll frequently.

The third section of the `weblog_cursor` structure holds everything the cursor needs to know about the current line. The cursor has a buffer to hold the text and length of the current line. There are also a series of pointers and length counters that are used to scan the line. Since scanning the line is fairly expensive, and must be done all at once, the module delays scanning the line until it's sure the data is needed. Once scanned, the module will keep the scan data around until it reads a new line. To keep track of when a line has been scanned, the cursor contains a "valid" flag.

As we go through the rest of the module functions, you'll see how these fields are used.

You might be thinking that a 4 KB line buffer seems a bit large, but frequently it is not enough. CGI scripts that use extensive query strings can generate very long logfile lines. Another issue is that many referrer URLs, especially those from search engines, can be extremely large. While most lines are only a hundred characters or so, it is best if the module can try to deal with the longer ones as well. Even with a 4 KB buffer, you'll need to properly deal with potential buffer overflows.

Now that we've seen what the cursor looks like, let's have a look at how it is opened and created. When the module needs to create a new cursor, it will first attempt to open the correct logfile. Assuming that succeeds, it will allocate the cursor structure and initialize the basic data:

```
static int weblog_open( sqlite3_vtab *vtab, sqlite3_vtab_cursor **cur )
{
    weblog_vtab     *v = (weblog_vtab*)vtab;
    weblog_cursor   *c;
    FILE            *fptr;

    *cur = NULL;

    fptr = fopen( v->filename, "r" );
    if ( fptr == NULL ) return SQLITE_ERROR;

    c = sqlite3_malloc( sizeof( weblog_cursor ) );
    if ( c == NULL ) {
        fclose( fptr );
        return SQLITE_NOMEM;
    }

    c->fptr = fptr;
    *cur = (sqlite3_vtab_cursor*)c;
    return SQLITE_OK;
}
```

The open function doesn't need to initialize the line data, as this will all be reset when we read the first line from the data file.

The `xClose()` function is relatively simple:

```
static int weblog_close( sqlite3_vtab_cursor *cur )
{
    if ( ((weblog_cursor*)cur)->fptr != NULL ) {
        fclose( ((weblog_cursor*)cur)->fptr );
    }
    sqlite3_free( cur );
    return SQLITE_OK;
}
```

Close the file, release the memory.

Filter

Since the weblog module chooses to ignore the `xBestIndex()` function, it largely ignores `xFilter()` as well. The file is reset to the beginning, just to be sure, and the module reads the first line of data:

```
static int weblog_filter( sqlite3_vtab_cursor *cur,
        int idxnum, const char *idxstr,
        int argc, sqlite3_value **value )
{
    weblog_cursor   *c = (weblog_cursor*)cur;

    fseek( c->fptr, 0, SEEK_SET );
    c->row = 0;
    c->eof = 0;
    return weblog_get_line( (weblog_cursor*)cur );
}
```

The `weblog_get_line()` function reads in a single line from the logfile and copies it into our line buffer. It also verifies that it got a full line. If it didn't get a full line, the function keeps reading (but discards the input) to make sure the file location is left at the beginning of the next valid line. We can reduce how often this happens by making the line buffer bigger, but no matter how big we make the buffer, it is always a good idea to make sure a whole line is consumed, even if the tail is discarded:

```
static int weblog_get_line( weblog_cursor *c )
{
    char   *cptr;
    int    rc = SQLITE_OK;

    c->row++;                         /* advance row (line) counter */
    c->line_ptrs_valid = 0;           /* reset scan flag */
    cptr = fgets( c->line, LINESIZE, c->fptr );
    if ( cptr == NULL ) {  /* found the end of the file/error */
        if ( feof( c->fptr ) ) {
            c->eof = 1;
        } else {
            rc = -1;
        }
        return rc;
    }
    /* find end of buffer and make sure it is the end a line... */
    cptr = c->line + strlen( c->line ) - 1;       /* find end of string */
    if ( ( *cptr != '\n' )&&( *cptr != '\r' ) ) { /* overflow? */
        char   buf[1024], *bufptr;
        /* ... if so, keep reading */
        while ( 1 ) {
            bufptr = fgets( buf, sizeof( buf ), c->fptr );
            if ( bufptr == NULL ) {  /* found the end of the file/error */
                if ( feof( c->fptr ) ) {
                    c->eof = 1;
                } else {
                    rc = -1;
                }
                break;
            }
            bufptr = &buf[ strlen( buf ) - 1 ];
            if ( ( *bufptr == '\n' )||( *bufptr == '\r' ) ) {
                break;               /* found the end of this line */
            }
        }
    }

    while ( ( *cptr == '\n' )||( *cptr == '\r' ) ) {
        *cptr-- = '\0';   /* trim new line characters off end of line */
    }
    c->line_len = ( cptr - c->line ) + 1;
    return rc;
}
```

Besides reading a full line, this function also resets the scan flag (to indicate the line buffer has not had the individual fields scanned) and adds one (1) to the line count. At the end, the function also trims off any trailing newline or carriage return characters.

Rows and Columns

We only have a few functions left. In specific, the module only needs to define the two row-handling functions, `xNext()` and `xEof()`. We also need the two column functions, `xRowid()` and `xColumn()`.

Three of these four functions are quite simple. The `xNext()` function can call `weblog_get_line()`, just as the `xFilter()` function did. The `xEof()` and `xRowid()` functions return or pass back values that have already been calculated elsewhere:

```
static int weblog_next( sqlite3_vtab_cursor *cur )
{
    return weblog_get_line( (weblog_cursor*)cur );
}

static int weblog_eof( sqlite3_vtab_cursor *cur )
{
    return ((weblog_cursor*)cur)->eof;
}

static int weblog_rowid( sqlite3_vtab_cursor *cur, sqlite3_int64 *rowid )
{
    *rowid = ((weblog_cursor*)cur)->row;
    return SQLITE_OK;
}
```

The interesting function is the `xColumn()` function. If you'll recall, in addition to the line buffer, the `weblog_cursor` structure also had an array of character pointers and length values. Each of these pointers and lengths corresponds to a column value in the defined table format. Before the module can extract those values, it needs to scan the input line and mark all the columns by setting the pointer and length values.

Using a length value means the module doesn't need to insert termination characters into the original string buffer. That's good, since several of the fields overlap. Using terminating characters would require making private copies of these data fields. In the end, a length value is quite useful anyway, as most of SQLite's value-handling routines utilize length values.

The function that sets up all these pointers and length calculations is `weblog_scan line()`. We'll work our way through this section by section. At the top are, of course, the variable definitions. The `start` and `end` pointers will be used to scan the line buffer, while the `next` value keeps track of the terminating character for the current field:

```
static int weblog_scanline( weblog_cursor *c )
{
    char   *start = c->line, *end = NULL, next = ' ';
    int    i;

    /* clear pointers */
    for ( i = 0; i < TABLE_COLS; i++ ) {
        c->line_ptrs[i] = NULL;
        c->line_size[i] = -1;
    }
```

With the variables declared, the first order of business is to reset all of the column pointers and sizes.

Next, the scan function loops over the native data fields in the line. This scans up to nine fields from the line buffer. These fields correspond to all the primary fields in a *combine* format logfile. If the logfile is a *common* format file (with only seven fields) or if the line buffer was clipped off, fewer fields are scanned. Any fields that are not properly scanned will eventually end up returning NULL SQL values:

```
    /* process actual fields */
    for ( i = 0; i < TABLE_COLS_SCAN; i++ ) {
        next = ' ';
        while ( *start == ' ' )  start++;     /* trim whitespace */
        if (*start == '\0' )  break;          /* found the end */
        if (*start == '"' ) {
            next = '"';  /* if we started with a quote, end with one */
            start++;
        }
        else if (*start == '[' ) {
            next = ']';  /* if we started with a bracket, end with one */
            start++;
        }
        end = strchr( start, next );    /* find end of this field */
        if ( end == NULL ) {            /* found the end of the line */
            int     len = strlen ( start );
            end = start + len;          /* end now points to '\0' */
        }
        c->line_ptrs[i] = start;        /* record start */
        c->line_size[i] = end - start;  /* record length */
        while ( ( *end != ' ' )&&( *end != '\0' ) )  end++;  /* find end */
        start = end;
    }
```

This loop attempts to scan one field at a time. The first half of the loop figures out the ending character of the field. In most cases it is a space, but it can also be a double-quote or square bracket. Once it knows what it's looking for, the string is scanned for the next end marker. If the marker isn't found, the rest of the string is used.

When this loop exits, the code has attempted to set up the first nine column pointers. These make up the native fields of the logfile. The next step is to set up pointers and lengths for the additional 11 columns that represent subfields and alternate representations. The first additional value is the IP address, returned as an integer. This function doesn't do data conversions, so a direct copy of pointer and length from the first column can be made:

```
/* process special fields */
/* ip_int - just copy */
c->line_ptrs[9] = c->line_ptrs[0];
c->line_size[9] = c->line_size[0];
```

Next, all of the date field pointers and lengths are set up. This section of code makes some blatant assumptions about the format of the timestamp, but there isn't much choice. The code could scan the individual fields, but it would still be forced to make assumptions about the ordering of the fields. In the end, it is easiest to just assume the format is consistent and hardcode the field lengths. This example ignores the time zone information:

```
/* assumes: "DD/MMM/YYYY:HH:MM:SS zone" */
/*     idx:  012345678901234567890...   */
if (( c->line_ptrs[3] != NULL )&&( c->line_size[3] >= 20 )) {
    start = c->line_ptrs[3];
    c->line_ptrs[10] = &start[0];    c->line_size[10] = 2;
    c->line_ptrs[11] = &start[3];    c->line_size[11] = 3;
    c->line_ptrs[12] = &start[3];    c->line_size[12] = 3;
    c->line_ptrs[13] = &start[7];    c->line_size[13] = 4;
    c->line_ptrs[14] = &start[12];   c->line_size[14] = 2;
    c->line_ptrs[15] = &start[15];   c->line_size[15] = 2;
    c->line_ptrs[16] = &start[18];   c->line_size[16] = 2;
}
```

After the date fields, the next step is to extract the HTTP operation and URL. These are extracted as the first two subfields of the HTTP Request log field. The code plays some games to be sure it doesn't accidentally pass a NULL pointer into `strchr()`, but otherwise it just finds the first two spaces and considers those to be the ending of the two fields it is trying to extract:

```
/* req_op, req_url */
start = c->line_ptrs[4];
end = ( start == NULL ? NULL : strchr( start, ' ' ) );
if ( end != NULL ) {
    c->line_ptrs[17] = start;
    c->line_size[17] = end - start;
    start = end + 1;
}
end = ( start == NULL ? NULL : strchr( start, ' ' ) );
if ( end != NULL ) {
    c->line_ptrs[18] = start;
    c->line_size[18] = end - start;
}
```

The final column represents the full contents of the line buffer. We also need to set the valid flag to indicate the field pointers are valid and ready for use:

```
        /* line */
        c->line_ptrs[19] = c->line;
        c->line_size[19] = c->line_len;

        c->line_ptrs_valid = 1;
        return SQLITE_OK;
    }
```

Once this function has been called, all the fields that could be scanned will have a valid pointer and length value. With the data scanned, this and subsequent calls to `xColumn()` can use the relevant values to pass back their database values. Let's return to looking at `xColumn()`.

The first thing the `xColumn()` code does is making sure the line has already been scanned. If not, the code calls `weblog_scanline()` to set up all the field pointers:

```
    static int weblog_column( sqlite3_vtab_cursor *cur, sqlite3_context *ctx, int cidx )
    {
        weblog_cursor    *c = (weblog_cursor*)cur;

        if ( c->line_ptrs_valid == 0 ) {
            weblog_scanline( c );        /* scan line, if required */
        }
        if ( c->line_size[cidx] < 0 ) {   /* field not scanned and set */
            sqlite3_result_null( ctx );
            return SQLITE_OK;
        }
```

Next, if the requested column doesn't have a valid set of values, the module passes back an SQL NULL for the column.

The code then processes columns with specific conversion needs. Any column that needs special processing or conversion will be caught by this switch statement. The first specialized column is the integer version of the IP address. This block of code converts each octet of the IP address into an integer value. The only issue is that all integer values within SQLite are signed, so the code needs to be careful about constructing the value into a 64-bit integer. For maximum compatibility, it avoids using shift operations:

```
        switch( cidx ) {
        case 9: { /* convert IP address string to signed 64 bit integer */
            int           i;
            sqlite_int64   v = 0;
            char          *start = c->line_ptrs[cidx], *end, *oct[4];

            for ( i = 0; i < 4; i++ ) {
                oct[i] = start;
                end = ( start == NULL ? NULL : strchr( start, '.' ) );
                if ( end != NULL ) {
                    start = end + 1;
                }
            }
            v += ( oct[3] == NULL ? 0 : atoi( oct[3] ) ); v *= 256;
            v += ( oct[2] == NULL ? 0 : atoi( oct[2] ) ); v *= 256;
            v += ( oct[1] == NULL ? 0 : atoi( oct[1] ) ); v *= 256;
            v += ( oct[0] == NULL ? 0 : atoi( oct[0] ) );
            sqlite3_result_int64( ctx, v );
            return SQLITE_OK;
        }
```

The next specialized column is one of the two month fields. In the logfile, the month value is given as a three-character abbreviation. One column returns this original text value, while another returns a numeric value. To convert from the abbreviation to the numeric value, the code simply looks for constants in the month string. If it can't find a match, the code breaks out. As we'll see, if the code breaks out it will eventually end up returning the text value:

```
    case 12: {
        int m = 0;
             if ( strncmp( c->line_ptrs[cidx], "Jan", 3 ) == 0 ) m =  1;
        else if ( strncmp( c->line_ptrs[cidx], "Feb", 3 ) == 0 ) m =  2;
        else if ( strncmp( c->line_ptrs[cidx], "Mar", 3 ) == 0 ) m =  3;
        else if ( strncmp( c->line_ptrs[cidx], "Apr", 3 ) == 0 ) m =  4;
        else if ( strncmp( c->line_ptrs[cidx], "May", 3 ) == 0 ) m =  5;
        else if ( strncmp( c->line_ptrs[cidx], "Jun", 3 ) == 0 ) m =  6;
        else if ( strncmp( c->line_ptrs[cidx], "Jul", 3 ) == 0 ) m =  7;
        else if ( strncmp( c->line_ptrs[cidx], "Aug", 3 ) == 0 ) m =  8;
        else if ( strncmp( c->line_ptrs[cidx], "Sep", 3 ) == 0 ) m =  9;
        else if ( strncmp( c->line_ptrs[cidx], "Oct", 3 ) == 0 ) m = 10;
        else if ( strncmp( c->line_ptrs[cidx], "Nov", 3 ) == 0 ) m = 11;
        else if ( strncmp( c->line_ptrs[cidx], "Dec", 3 ) == 0 ) m = 12;
        else break;    /* give up, return text */
        sqlite3_result_int( ctx, m );
        return SQLITE_OK;
    }
```

There are a number of additional columns (including some of the "native" ones) that are returned as integers. None of these columns require special processing, other than the string-to-integer conversion. The standard `atoi()` function is used for this conversion. Although the string pointers are not null-terminated, the `atoi()` function will automatically return once it encounters a non-numeric character. Since all of these fields are bound by spaces or other characters, this works out exactly the way we want:

```
    case 5:    /* result code */
    case 6:    /* bytes transfered */
    case 10:   /* day-of-month */
    case 13:   /* year */
    case 14:   /* hour */
    case 15:   /* minute */
    case 16:   /* second */
        sqlite3_result_int( ctx, atoi( c->line_ptrs[cidx] ) );
        return SQLITE_OK;
    default:
        break;
    }
```

```
        sqlite3_result_text( ctx, c->line_ptrs[cidx],
                                  c->line_size[cidx], SQLITE_STATIC );
        return SQLITE_OK;
    }
```

Finally, any field that did not require special processing is returned as a text value. Although the line buffer will be overwritten when the next line is read, the data pointer passed into `sqlite3_result_text()` only needs to stay valid until the next call to `xNext()`. This allows the module to use the `SQLITE_STATIC` flag.

With that, we've defined all the required functions for our weblog module.

Register the Module

Now that we've seen how all the module functions are implemented, the last thing to do is register the weblog module as part of the extension initialization function:

```
static sqlite3_module weblog_mod = {
    1,                  /* iVersion        */
    weblog_connect,     /* xCreate()       */
    weblog_connect,     /* xConnect()      */
    weblog_bestindex,   /* xBestIndex()    */
    weblog_disconnect,  /* xDisconnect()   */
    weblog_disconnect,  /* xDestroy()      */
    weblog_open,        /* xOpen()         */
    weblog_close,       /* xClose()        */
    weblog_filter,      /* xFilter()       */
    weblog_next,        /* xNext()         */
    weblog_eof,         /* xEof()          */
    weblog_column,      /* xColumn()       */
    weblog_rowid,       /* xRowid()        */
    NULL,               /* xUpdate()       */
    NULL,               /* xBegin()        */
    NULL,               /* xSync()         */
    NULL,               /* xCommit()       */
    NULL,               /* xRollback()     */
    NULL,               /* xFindFunction() */
    weblog_rename       /* xRename()       */
};

int weblog_init( sqlite3 *db, char **error, const sqlite3_api_routines *api )
{
    SQLITE_EXTENSION_INIT2(api);
    return sqlite3_create_module( db, "weblog", &weblog_mod, NULL );
}
```

Since there is no attempt to create an instance of a weblog table, this initialization function is a bit simpler than the previous dblist example.

Example Usage

Now that we've worked through the whole example, let's see what the code can do. Here are a few different examples that show off the power of the weblog module.

While doing these types of queries is not a big deal for people that are comfortable with SQL, realize that we can run all of these queries without having to first import the logfile data. Not only does that make the whole end-to-end process much faster, it means we can run these types of queries against active, "up to the second" logfiles.

To show off how this module works, the server administrators of *http://oreilly.com/* were nice enough to provide me with some of their logfiles. The file referred to as *oreilly.com_access.log* is an Apache *combine* logfile with 100,000 lines of data. Once compiled and built into a loadable module, we can import the weblog module and create a virtual table that is bound to this file using these commands:

```
sqlite> .load weblog.sqlite3ext weblog_init
sqlite> CREATE VIRTUAL TABLE log USING weblog( oreilly.com_access.log );
```

We then issue queries to look at different aspects of the file. For example, if we want to know what the most common URL is, we run a query like this:

```
sqlite> SELECT count(*) AS Count, req_url AS URL FROM log
   ...>   GROUP BY 2 ORDER BY 1 DESC LIMIT 8;

Count  URL
-----  ----------------------------------------
2490   /images/oreilly/button_cart.gif
2480   /images/oreilly/button_acct.gif
2442   /styles/all.css
2348   /images/oreilly/888-line.gif
2233   /styles/chrome.css
2206   /favicon.ico
1975   /styles/home2.css
1941   /images/oreilly/satisfaction-icons.gif
```

It is fairly common to see `favicon.ico` very near the top, along with any site-wide CSS and image files. In the case of smaller sites that have a lot less traffic, it isn't uncommon for the most requested URL to be `/robots.txt`, which is used by search engines.

We can also see what the most expensive items on the website are, in terms of bytes moved:

```
sqlite> SELECT sum(bytes) AS Bytes, count(*) AS Count, req_url AS URL
   ...>   FROM log WHERE result = 200 GROUP BY 3 ORDER BY 1 DESC LIMIT 8;

Bytes     Count  URL
--------  -----  ---------------------------------------------
46502163  1137   /images/oreilly/mac_os_x_snow_leopard-148.jpg
40780252  695    /
37171328  2384   /styles/all.css
35403200  2180   /styles/chrome.css
31728906  781    /catalog/assets/pwr/engine/js/full.js
31180460  494    /catalog/9780596510046/index.html
21573756  88     /windows/archive/PearPC.html
21560154  3      /catalog/dphotohdbk/chapter/ch03.pdf
```

We see that some of these items are not that large, but are requested frequently. Other items have only a small number of requests, but are big enough to make a noticeable contribution to the total number of served bytes.

Here is one final example. This shows what IP addresses are downloading the most number of unique items. Since this is from live data, I've altered the IP addresses:

```
sqlite> SELECT count(*) AS Uniq, sum(sub_count) AS Ttl,
   ...>        sum(sub_bytes) AS TtlBytes, sub_ip AS IP
   ...>   FROM (SELECT count(*) AS sub_count, sum(bytes) AS sub_bytes,
   ...>                ip_str AS sub_ip FROM log GROUP BY 3, req_url)
   ...>   GROUP BY 4 ORDER BY 1 DESC LIMIT 8;

Uniq  Ttl   TtlBytes    IP
----  ----  ----------  ------------------
1295  1295  31790418    10.5.69.83
282   334   13571771    10.170.13.97
234   302   4234382     10.155.7.28
213   215   3089112     10.155.7.77
163   176   2550477     10.155.7.29
159   161   4279779     10.195.137.175
153   154   2292407     10.23.146.198
135   171   2272949     10.155.7.71
```

For each IP address, the first column is the number of unique URLs requested, while the second column is the total number of requests. The second column should always be greater than or equal to the first column. The third column is the total number of bytes, followed by the (altered) IP address in question. Exactly how this query works is left as an exercise for the reader.

There are countless other queries we could run. For anyone that has ever imported log data into an SQL database and played around with it, none of this is particularly inspiring. But consider this for a moment: the query time for the first two of these examples is a bit less than five seconds on an economy desktop system that is several years old. The third query was a bit closer to eight seconds.

Five seconds to scan a 100,000-row table might not be blazingly fast, but remember that those five seconds are the grand total for everything, including data "import." Using the virtual table module allows us to go from a raw logfile with 100,000 lines to a query answer in just that amount of time—no data staging, no format conversions, no data imports. That's important, since importing involves a lot of I/O and can be a slow process. For example, importing the same file into a standard SQLite table by more traditional means takes nearly a minute and that doesn't even include any queries!

Now consider that we enable all this functionality with less than 400 lines of C code. Accessing the original data, rather than importing it into standard tables, allows the end-to-end data analysis process to be much faster, and allows you to query the data, as it is recorded by the web server, in real time. As an added bonus, the virtual table can also be used as an importer, by using the `CREATE TABLE... AS` or `INSERT... SELECT` SQL commands.

If you find yourself faced with the task of writing a script to analyze, search, or summarize some structured source of data, you might consider writing an SQLite module instead. A basic, read-only module is a fairly minor project, and once you've got that in place you have the complete power of the SQLite database engine at your disposal (plus an added data importer!). That makes it easy to write, test, and tune whatever queries you need in just a few lines of SQL.

Best Index and Filter

Let's take a closer look at the `xBestIndex()` and `xFilter()` functions. Both of our example modules were fairly simple and didn't use them, but proper use of these functions is critical for internal modules that implement some types of high-performance indexing system.

Purpose and Need

By default, the only way to get data out of a table—virtual or otherwise—is to do a full table scan. This can be quite expensive, especially if the table is large and the query is trying to extract a small number of rows.

Standard tables have ways of boosting retrieval speeds, such as using indexes. The query optimizer can use other hints found in a standard table definition, such as knowing which columns are unique or have other constraints on them.

Virtual tables lack these features. You cannot create an index on a virtual table, and the query optimizer has no knowledge of the structure or format of a virtual table, other than the column names. The only known constraint on a virtual table is that each virtual row must have a unique, integer `ROWID`.

Without any additional information, it is very difficult to optimize a query that involves a virtual table. This is true for both the query planner and the virtual table itself. For the best performance, the query optimizer needs to understand what types of lookups the virtual table is best suited to doing. Conversely, the virtual table module needs to understand the nature of the user query, including any constraints, so that it can use any internal indexes or lookup optimizations to the best of its ability.

The purpose of the `xBestIndex()` and `xFilter()` functions is to bridge this gap. When an SQL statement is prepared, the query optimizer may call `xBestIndex()` several times, presenting several different query possibilities. This allows the module to formulate its own query plan and pass back an approximate cost metric to the query optimizer. The query optimizer will use this information to pick a specific query plan.

When the query statement is executed, the SQLite library uses `xFilter()` to communicate back to the module which query plan was actually chosen. The module can use this information to optimize its internal data lookups, as well as skip over any rows that are not relevant to the query at hand. This allows a virtual table to implement more targeted data lookups and retrievals, not unlike an index on a traditional table.

xBestIndex()

If you'll recall, the `xBestIndex()` function is a table-level function that looks like this:

```
int xBestIndex( sqlite3_vtab *vtab, sqlite3_index_info *idxinfo );
```

The whole key to this function is the `sqlite3_index_info` structure. This structure is divided into two sections. The first section provides a series of inputs to your function, allowing SQLite to propose a query plan to the module. The input section should be treated as read-only.

The second section is the output section. A module uses this second section to communicate back to the query optimizer information about which constraints the virtual table is prepared to enforce, and how expensive the proposed query might be. The module is also given a chance to associate an internal query plan or other data to this particular proposal. The query optimizer will then use this data to select a specific query plan.

The input section consists of two size values and two arrays. The `nConstraint` integer indicates how many elements are in the `aConstraint[]` array. Similarly, the `nOrderBy[]` integer indicates how many elements are in the `aOrderBy[]` array:

```
struct sqlite3_index_info {
    /**** Inputs ****/
    int      nConstraint;             /* Number of entries in aConstraint */
    struct sqlite3_index_constraint {
        int            iColumn;       /* Column on lefthand side of constraint */
        unsigned char  op;            /* Constraint operator */
        unsigned char  usable;        /* True if this constraint is usable */
        int            iTermOffset;   /* Used internal - xBestIndex should ignore */
    } *aConstraint;                   /* Table of WHERE clause constraints */

    int      nOrderBy;                /* Number of terms in the ORDER BY clause */
    struct sqlite3_index_orderby {
        int            iColumn;       /* Column number */
        unsigned char  desc;          /* True for DESC.  False for ASC. */
    } *aOrderBy;                      /* The ORDER BY clause */
```

The `aConstraint[]` array communicates a series of simple constraints that a query may put on the virtual table. Each array element defines one query constraint by passing values for a column index (`aConstraint[i].iColumn`) and a constraint operator (`aConstraint[i].op`). The column index refers to the columns of the virtual table, with a zero signifying the first column. An index of `-1` indicates the constraint is being applied to the virtual `ROWID` column.

The specific constraint operator is indicated with one of these constants. The referenced column (the column index) is always assumed to be on the lefthand side. These are the only operators that can be optimized by a virtual table:

```
SQLITE_INDEX_CONSTRAINT_EQ      /* COL =  Expression */
SQLITE_INDEX_CONSTRAINT_GT      /* COL >  Expression */
SQLITE_INDEX_CONSTRAINT_LE      /* COL <= Expression */
SQLITE_INDEX_CONSTRAINT_LT      /* COL <  Expression */
SQLITE_INDEX_CONSTRAINT_GE      /* COL >= Expression */
SQLITE_INDEX_CONSTRAINT_MATCH   /* COL MATCH Expression */
```

For example, if one of the `aConstraint` elements had the values:

```
aConstraint[i].iColumn = -1;
aConstraint[i].op      = SQLITE_INDEX_CONSTRAINT_LE;
```

That would roughly translate to a `WHERE` clause of:

```
...WHERE ROWID <= ?
```

The parameter on the right side of the expression may change from query to query, but will remain constant for the any given table scan, just as if it were a statement parameter with a bound value.

Each `aConstraint[]` element also contains a `usable` element. Some constraints may not be usable by the optimizer due to joins or other external conditions put on the query. Your code should only pay attention to those constraints where the usable field is nonzero.

The second array of the input section, `aOrderBy[]`, communicates a set of requested `ORDER BY` sortings (it may also be generated by columns in a `GROUP BY` clause). Each ordering element is defined by a column index and a direction (ascending or descending). The column indexes work the same way, with defined columns starting at 0 and -1 referring to the `ROWID`. The ordering elements should be treated as a series of `ORDER BY` arguments, with the whole data set being sorted by the first ordering, then subsets of equal values being sorted by the second ordering, and so on.

The output section contains the data that is passed back to the SQLite optimizer. It consists of a constraint array and a set of values. The `aConstraintUsage[]` array will always be the same size as the `aConstraint[]` array (that is, will always have `nConstraint` elements). SQLite will always zero out the memory used by the output section. This is why it is safe to ignore the structure in simplified implementations of `xBestIndex()`—the structure is basically preset to an answer of, "this module cannot optimize anything." In that case, every virtual table query will require a full table scan:

```
        /**** Outputs ****/
        struct sqlite3_index_constraint_usage {
            int             argvIndex;  /* If >0,constraint is part of argv to xFilter */
            unsigned char   omit;       /* Do not code a test for this constraint */
        } *aConstraintUsage;

        int      idxNum;                /* Number used to identify the index */
        char     *idxStr;               /* Application-defined string */
        int      needToFreeIdxStr;      /* Free idxStr using sqlite3_free() if true */
        int      orderByConsumed;       /* True if output is already ordered */
        double   estimatedCost;         /* Estimated cost of using this index */
    };
```

If a module is able to optimize some part of the query, this is indicated to the query optimizer by modifying the output section of the `sqlite3_index_info` structure to indicate what query optimizations the module is willing and capable of performing.

Each element of the `aConstraintUsage[]` array corresponds to the same ordered element of the `aConstraint[]` array. For each constraint described in an `aConstraint[]` element, the corresponding `aConstraintUsage[]` element is used to describe how the module wants the constraint applied.

The `argvIndex` value is used to indicate to SQLite that you want the expression value of this constraint (that is, the value on the righthand side of the constraint) to be passed to the `xFilter()` function as one of the `argv` parameters. The `argvIndex` values are used to determine the sequence of each expression value. Any `aConstraintUsage[]` element can be assigned any index value, just as long as the set of assigned index values starts with one and has no gaps. No `aConstraintUsage[]` elements should share the same nonzero `argvIndex` value. If the default `argvIndex` value of zero is returned, the expression value is not made available to the `xFilter()` function. Exactly how this is used will make more sense when we look more closely at `xFilter()`.

The `omit` field of an `aConstraintUsage[]` element is used to indicate to the SQLite library that the virtual table module will take the responsibility to enforce this constraint and that SQLite can omit the verification process for this constraint. By default, SQLite verifies the constraints of every row returned by a virtual table (e.g., every row `xNext()` stops at). Setting the `omit` field will cause SQLite to skip the verification process for this constraint.

Following the constraint array is a series of fields. The first three fields are used to communicate to the `xFilter()` function. The fields `idxNum` and `idxStr` can be used by the module however it wishes. The SQLite engine makes no use of these fields, other than to pass them back to `xFilter()`. The third field, `needToFreeIdxStr`, is a flag that indicates to the SQLite library that the memory pointed to by `idxStr` has been dynamically allocated by `sqlite3_malloc()`, and the SQLite library should free that memory with `sqlite3_free()` if the library decides it is no longer required.

This flag is needed to prevent memory leaks. Remember that `xBestIndex()` may be called several times as part of the prepare process for an SQL statement. The module will usually pass back a unique `idxStr` value for each proposed query plan. Only one of these `idxStr` values will be passed to `xFilter()`, however, and the rest must be discarded. That means that any string (or other memory block) you provide to `idxStr` needs to either be static memory, or the memory needs to be allocated with `sqlite3_malloc()` and the `needToFreeIdxStr` flag needs to be set. This allows the SQLite library to properly clean up any unused `idxStr` allocations.

The `orderByConsumed` field is used to indicate that the module is able to return the data presorted in the order defined by the `aOrderBy` array. This is an all-or-nothing flag. If three `aOrderBy` elements are given, but the module can only sort the output by the first column, it must return a false value.

Finally, the `estimatedCost` field is used to communicate a cost value back to the SQLite library. If this is an external module, this number should approximate the total number of disk accesses required to return all rows that meet the specified constraints. If this is an internal module, it can be an approximation of the number of `sqlite3_step()` and `sqlite3_column_xxx()` calls. In situations where a full table scan is required, it can estimate the number of rows in the virtual table. The exact measurement is not extremely meaningful, other than the relative values between different calls to `xBestIndex()`.

xFilter()

The `xFilter()` function provides a way for the SQLite library to notify the module, within the context of a specific table cursor, exactly what constraints and ordering should be applied to the next table scan. Recall that the `xFilter()` prototype looks like this:

```
int xFilter( sqlite3_vtab_cursor *cursor,
        int idxNum, const char *idxStr,
        int argc, sqlite3_value **argv )
```

The first argument is the table cursor that requires these constraints. The `idxNum` and `idxStr` values are the same values that were passed back by the module in a prior call to `xBestIndex()`. These mean whatever the module wants, just as long as the code in `xBestIndex()` and `xFilter()` agrees on what they are and what the values represent.

Finally, the last two arguments are derived from the `aConstraintUsage[].argvIndex` values passed back by the module. The `argv` parameter is an array of `sqlite3_value` structures, while the `argc` parameter indicates how many elements are in the `argv` array.

Going back to our prior example, consider an `sqlite3_index_info` structure with an `aConstraint[i]` element, where `iColumn=-1` and `op=SQLITE_INDEX_CONSTRAINT_LE` (indicating a constraint of `ROWID >= ?`). If the module's `xBestIndex()` function set `aConstraintUsage[i].argIndex` to a value of 1, the `argv[0]` value passed into `xFilter()` will have the value found on the righthand side of the expression.

Notice that the argument indexes between `xBestIndex()` and `xFilter()` are off by one. Because `sqlite3_index_info` considers an `aConstraintUsage[].argvIndex` value of 0 to indicate an invalid index, the `argvIndex` values start at 1. The actual `argv` indexes will all be one less, however, as they start at 0.

Using the `idxNum`, `idxStr`, and `argv` values, it is the responsibility of `xFilter()` to configure this table cursor to provide the correct constraints and ordering that were promised by the corresponding `sqlite3_index_info` block.

Typical Usage

The design of xBestIndex() and xFilter() functions is strongly focused on optimizing internal style modules. These are modules that are going to use one or more SQL statements that operate over a set of internal tables to produce the virtual table data. This is similar to how the dblist module works, but normally involves more complex SQL commands.

A module is free to do whatever it wants with the idxNum and idxStr values, but most internal modules use them to pass off pre-built SQL command strings. Each time xBestIndex() is called, the module tries to figure out how it would service the query constraints and ordering constraints, by adding conditions, constraints, and ORDER BY parameters to the internal SQL statements used to generate the virtual table data. The xBestIndex() function marks the constraints it can use and builds the required SQL command strings, complete with statement parameters. These SQL commands are passed back with the idxStr value. The idxNum can be used to pass back a string length, or some other index value or bit flags or whatever the module wants. The argvIndex values of the aConstraintUsage elements are set to the corresponding statement parameter index values. In essence, the xBestFilter() function will build the SQL command strings that query the virtual table data in such a way that the required constraints and ordering are already "baked in" to the behind-the-scenes queries.

When xFilter() is called, the idxStr value will have the relevant SQL command strings for that query configuration. The SQL command strings can then be prepared, and the constraint expressions pass in via the argv array, and can be bound to any statement parameters. The xFilter() function starts to step through the prepared statements, generating the first row. Like the dblist internal module, subsequent calls to xNext() continue to step through any internal statements, returning additional rows.

As long as xBestIndex() can derive a reasonable set of SQL command strings that are capable of expressing the required internal query (or queries), this is all reasonably straightforward. If necessary, multiple SQL command strings can be passed into xFilter() by defining them one after another in a large string, and using the tail parameter of sqlite3_prepare_xxx() to prepare multiple statements, one after another.

Things can be difficult when dealing with external modules. Very often external modules can't define complex query conditions or sort ordering with a simple string. Although the idxStr pointer can be used to pass in some type of data structure, it can be difficult to encode all the constraint information. This is one of the reasons why many modules, and especially external modules, forego the use of xBestIndex() and xFilter(), and just depend on full table scans for all operations. Full table scans might be slower, but they still work.

That might sound bad, but remember that even on a standard table with a standard index, you typically don't start to see really good returns on using the index unless a constraint and appropriate index are able to eliminate 80% or better of the rows. Spending a lot of time to build a constraint handler that only filters out a small percentage of rows is normally a losing proposition. While that can be the whole point of internal modules, the primary goal of most external modules is to simply provide data connectivity. If you're working on an external module, get the basic data translation working first, and then worry about possibly implementing more efficient lookup patterns.

Wrap-Up

If you've made it through the whole chapter, you should have a pretty good idea of what the virtual table system can do and how it works. A well-written internal module can bring a whole new feature set to the database, allowing an application to shape and manipulate data structures that SQLite might not otherwise be able to store or process in an efficient manner. External modules can provide a data conduit, providing both speed and flexibility in the use of external data sources. They can also double as powerful data importers.

That power comes at a cost, however. Without getting into filesystems and storage engines, a module is the most complex bit of extension code most people will ever write for SQLite. A good module depends on a solid design that will properly track all of the necessary states and perform all of the required functions.

To avoid frustration, it is often a good idea to start with the simple base cases and expand your code and design to cover the more complex situations.