CHAPTER 8

Language Extensions

SQLite's native language is C. It is written in C and has its own C API. The open source community, however, has provided many extensions for SQLite that make it accessible to many programming languages and libraries such as Perl, Python, Ruby, Java, Qt, and ODBC. In many cases there are multiple extensions to choose from for a given language, developed by different people to meet different needs.

Many extensions conform to various API standards. One of the SQLite Perl extensions, for example, follows the Perl DBI—Perl's standard interface for database access. Similarly, one of the Python extensions conforms to Python's DB API specification, as does at least one of the Java extensions to JDBC. Regardless of their particular APIs, internally all extensions work on top of the same SQLite C API, which has its own unique features and design. And to some degree, all extensions reflect this design. Some extensions provide both a standard interface that conforms to their particular API standard as well as an alternative interface that better reflects the design of the SQLite API. Therefore, in such cases you can choose which interface works best for you based on your requirements.

In general, all extensions have the same comparative anatomy regardless of their particular API. They follow a similar pattern. Once you understand this pattern, every interface will start to look similar in many respects. There is some kind of connection object representing a single connection to the database, and some kind of cursor object representing a SQL query from which you can both execute commands and iterate over results. Conceptually, it is quite simple. Internally, you are looking at a `sqlite3` structure and a `sqlite3_stmt` structure, and from reading Chapter 5 you should have a good feel for how these work. The same rules apply for language extensions as for the C API.

This chapter covers language extensions for six popular languages: Perl, Python, Ruby, Java, Tcl, and PHP. The intent is to provide you with a convenient introduction with which to quickly get started using SQLite in a variety of different languages. Coverage for each language follows a common outline composed of four topics that pertain to both general database work and specific SQLite features:

- Connecting to databases
- Executing queries
- Using bound parameters
- Implementing user-defined functions and aggregates

Together these topics constitute the vast majority of what you will ever use in any particular SQLite extension. Not every extension supports all of these topics. For instance, some don't offer bound parameters. Conversely, some extensions offer unique features that are outside of the topics covered here. The aim here is to provide a consistent, straightforward process with which to easily get started using SQLite in a wide variety of languages. Once you get started with any extension, it is usually easy to pick up any special features available in that extension.

Where appropriate, this chapter addresses how different extensions use various parts of the C API in order to help you understand what is going on under the hood. If you have not read Chapter 5, I strongly recommend that you do so. No matter what language you program in, it will almost always be helpful to understand how SQLite works in order to make the most of it and thus write good code. It will help you select the most suitable query methods, understand the scope of transactions, and know how to anticipate, deal with, or avoid locks, among other things.

This chapter is not a language reference. I assume that you know how to program and are familiar with your particular language of interest. SQLite is first and foremost a programmer's database, and that is the assumption I've made here—you are a programmer and I need not cover what you already know.

Selecting an Extension

In some cases multiple interfaces are available for some languages. In such cases (with the exception of Python), I cover the interface that best fits the following criteria:

- Support for SQLite 3
- Good documentation
- Stability
- Portability

Despite these criteria, there were multiple candidates in some cases that met all the qualifications. In such cases the tiebreaker was more a matter of personal preference than anything else. But the purpose here is not to favor a particular extension. It is to teach concepts so that you can easily pick any extension in any language and quickly put it and SQLite to good use.

And the reasons I chose a particular extension may not be the reasons you would. There are quite a few things to consider when you select a particular extension. You will have requirements that must be satisfied when choosing an extension. Some of the things you might want to take into consideration include the following:

- **License compatibility**: The extension's license can directly affect how you can use it. If you are writing code for a commercial product, you definitely need to take this into consideration.
- **Data type mapping**: Another important point is how the extension maps SQLite's storage classes to the language's native types. Are all values returned as text? If not, is there an easy way to determine the mappings? Do you have any control over how the mapping is done?

- **Query methods**: Three different query methods are supported in the C API. Which one does the extension use? Ideally it supports all three. You are already familiar with the pros and cons of each approach, and it's nice to have all three methods in different situations.
- **API coverage**: How well does the extension cover other areas that don't easily map to many standard database interfaces? For example, does it allow you to call the SQL trace function, or operational security functions? Are these something you need?
- **Linkage and distribution**: How easy is it to use the extension with a particular version of SQLite? Does it include a version of SQLite in the distribution? Does it use a shared library or is it statically linked to a particular version? What if you need to upgrade SQLite; how easily can you do this with the extension?

There are other considerations as well, such as community support, mailing list activity, maintainer support and responsiveness, regression testing, code quality... the list goes on. All of these things may play an important role in your decision to use a particular extension. Only you can answer these questions for yourself.

The SQLite Wiki has a page that provides an exhaustive list of language interfaces, located at `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`. All of the interfaces covered here were taken from this list. The source code for all of the examples in this chapter is located in the *ch8* directory of the examples zip file, available at the Apress website.

Perl

There are two SQLite extensions for Perl. The one covered here is written by Matt Sergeant, and can be found on CPAN (`http://search.cpan.org/~msergeant/`). This extension follows very closely the Perl DBI standard; therefore, if you understand DBI, you will have no trouble using the extension. The full documentation for Perl DBI can be obtained by typing `man DBI` on POSIX systems, or online at `http://search.cpan.org/search?module=DBI`.

Installation

The current version supporting SQLite 3 at the time of this writing is `DBD-SQLite-1.11`. You can install it by using CPAN or by manually building the package from source. The prerequisites for the SQLite DBD module are of course Perl, a C compiler, as well as the DBI module. To install `DBD::SQLite` using CPAN, invoke the CPAN shell from the command line as follows:

```
mike@linux # cpan
cpan> install DBD::SQLite
```

To install from source in Linux/Unix, change to a temporary directory and do the following:

```
mike@linux # wget http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/
DBD-SQLite-1.11.tar.gz

... lots of wget gibberish ...
```

```
mike@linux # tar xzvf DBD-SQLite-1.11.tar.gz

... lots of tar gibberish ...

mike@linux # cd DBD-SQLite-1.11
mike@linux DBD-SQLite-1.11 # perl Makefile.PL

... lots of perl gibberish ...

Checking installed SQLite version...
Looks good
mike@linux DBD-SQLite-1.11 # make install

... more perl gibberish ...
```

The SQLite Perl extension includes its own copy of the SQLite 3, so there is no need to compile and install SQLite beforehand. SQLite is embedded in the extension.

Connecting

To ensure that you have the SQLite driver installed, you can use the DBI `available_drivers()` within Perl:

```
print "Drivers: " . join(", ", DBI->available_drivers()), "\n";
```

You connect to a database using the standard `DBI::connect` function, as follows:

```
use DBI;
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });
$dbh->disconnect;
```

The second and third arguments correspond to the driver name and database name. The third and fourth correspond to username and password, which are not applicable to SQLite.

The function returns a database handle object representing the database connection. Internally, this corresponds to a single `sqlite3` structure. You can create in-memory databases by passing `:memory:` for the name of the database. In this case, all tables and database objects will then reside in memory for the duration of the session, and will be destroyed when the connection is closed. You close the database using the database handle's `disconnect` method.

Query Processing

Queries are performed using the standard DBI interface as well. You can use the `prepare()`, `execute()`, `fetch()`, or `selectrow()` functions of the database connection handle. All of these are fully documented in the DBI man page. An example of using queries is shown in Listing 8-1.

Listing 8-1. *Executing Queries in Perl*

```
use DBI;
# Connect to database
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# Prepare the statment
my $sth = $dbh->prepare("SELECT name FROM foods LIMIT 3");

# Execute
print "\nArray:\n\n";
$sth->execute;

# Iterate over results and print
while($row = $sth->fetchrow_arrayref) {
    print @$row[0] . "\n";
}

# Do the same thing, this time using hashref
print "\nHash:\n\n";
$sth->execute;
while($row = $sth->fetchrow_hashref) {
    print @$row{'name'} . "\n";
}

# Finalize the statement
$sth->finish;

#Disconnect

$dbh->disconnect;
```

Internally, the `prepare()` method corresponds to the `sqlite3_prepare()` method. Similarly, `execute()` calls the first `sqlite3_step()` method. It automatically figures out if there is data to be returned. If there is no data (e.g., non-`SELECT` statements), it automatically finalizes the query. The `fetchrow_array()`, `fetchrow_hashref()`, and `fetch()` methods call `sqlite3_step()` as well, returning a single row in the result set, if one is available.

Additionally, you can run non-`SELECT` statements in one step using the database object's `do()` method. Listing 8-2 illustrates this using an in-memory database.

Listing 8-2. *The do() Method*

```
use DBI;
my $dbh = DBI->connect("dbi:SQLite:dbname=:memory:", "", "", { RaiseError => 1 });
$dbh->do("CREATE TABLE cast (name)");
$dbh->do("INSERT INTO cast VALUES ('Elaine')");
$dbh->do("INSERT INTO cast VALUES ('Jerry')");
$dbh->do("INSERT INTO cast VALUES ('Kramer')");
```

```
$dbh->do("INSERT INTO cast VALUES ('George')");
$dbh->do("INSERT INTO cast VALUES ('Newman')");

my $sth = $dbh->prepare("SELECT * FROM cast");
$sth->execute;

while($row = $sth->fetch) {
    print join(", ", @$row), "\n";
}
$sth->finish;
$dbh->disconnect;
```

The statement handle object's `finish()` method will call `sqlite3_finalize()` on the query object, if it has not already done so. This program produces the following output:

```
Elaine
Jerry
Kramer
George
Newman
```

Parameter Binding

Parameter binding follows the method defined in the DBI specification. While SQLite supports both positional and named parameters, the Perl interface uses only positional parameters, as illustrated in Listing 8-3.

Listing 8-3. *Parameter Binding in Perl*

```
use DBI;

my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

my $sth = $dbh->prepare("SELECT * FROM foods where name like :1");
$sth->execute('C%');

while($row = $sth->fetchrow_hashref) {
    print @$row{'name'} . "\n";
}

$sth->finish;
$dbh->disconnect;
```

The `execute()` method takes the parameter values defined in `prepare()`.

User-Defined Functions and Aggregates

User-defined functions and aggregates are implemented using private methods of the driver. They follow closely to their counterparts in the SQLite C API. Functions are registered using the following private method:

```
$dbh->func( $name, $argc, $func_ref, "create_function" )
```

Here `$name` specifies the SQL function name, `$argc` the number of arguments, and `$func_ref` a reference to the Perl function that provides the implementation. Listing 8-4 illustrates an implementation of `hello_newman()` in Perl.

Listing 8-4. *hello_newman() in Perl*

```
use DBI;

sub hello_newman {
    return "Hello Jerry";
}

# Connect
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# Register function
$dbh->func('hello_newman', 0, \&hello_newman, 'create_function');

# Call it
print $dbh->selectrow_arrayref("SELECT hello_newman()")->[0] . "\n";

$dbh->disconnect;
```

If you provide -1 as the number of arguments, then the function will accept a variable number of arguments. Listing 8-5 has Newman replying to each of the names supplied as arguments. If no arguments are provided, he simply replies "Hello Jerry."

Listing 8-5. *hello_newman() in Perl with Variable Arguments*

```
use DBI;

sub hello_newman {
    if (@_ == 0) {
        return "Hello Jerry";
    }

    return "Hello " . join ", ", @_;
}
```

```
# Connect
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# Register function
$dbh->func('hello_newman', -1, \&hello_newman, 'create_function');

# Call it
print $dbh->selectrow_arrayref("SELECT hello_newman()")->[0] . "\n";
print $dbh->selectrow_arrayref(
   "SELECT hello_newman('Elaine', 'Jerry')")->[0] . "\n";
print $dbh->selectrow_arrayref(
  "SELECT hello_newman('Elaine', 'Jerry', 'George')")->[0] . "\n";

$dbh->disconnect;
```

The `hello_newman()` function is passed all the arguments provided in the SQL statement in the `@_` variable. This example produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

Aggregates

User-defined aggregates are implemented as packages. Each package implements a step function and a finalize function. The step function is called for each row that is selected. The first value supplied to the function is the context, which is a persistent value maintained between calls to the function. The remaining values are the arguments to the aggregate function supplied in the SQL statement. The finalize function is provided the same context as the step function. The following example implements a simple `SUM` aggregate called `perlsum`. The aggregate is implemented in a package called `perlsum.pm`, shown in Listing 8-6.

Listing 8-6. *The perlsum() Aggregate*

```
package perlsum;

sub new { bless [], shift; }

sub step {
    my ( $self, $value ) = @_;
    @$self[0] += $value;
}
```

```
sub finalize {
    my $self = $_[0];
    return @$self[0];
}

sub init {
    $dbh = shift;
    $dbh->func( "perlsum", 1, "perlsum", "create_aggregate" );
}

1;
```

The `init` function is used to register the aggregate in the database connection. Listing 8-7 is a simple program illustrating its use.

Listing 8-7. *A perlsum() Test Program*

```
use DBI;
use perlsum;

# Connect
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

perlsum::init($dbh);

# Call it
print $dbh->selectrow_arrayref("SELECT perlsum(id) from foods")->[0] . "\n";

$dbh->disconnect;
```

There are two other private methods in the driver. One is `last_insert_rowid()`, used for obtaining the last primary key value generated by an autoincrement key in an `INSERT` statement, and the other is for setting the busy timeout. These are demonstrated in Listing 8-8.

Listing 8-8. *The last_insert_rowid() and busy_timeout() Methods*

```
use DBI;
use perlsum;

# Connect
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# Set timeout to 5 seconds
print $dbh->func(5, 'busy_timeout') . "\n";
print $dbh->func('busy_timeout') . "\n";
```

```
$dbh->do("INSERT INTO foods (type_id, name) values (9, 'Junior Mints')");

# Print the last generated autoincrement key value
print $dbh->func('last_insert_rowid') . "\n";

$dbh->disconnect;
```

Python

There are two SQLite wrappers for Python: PySQLite and APSW. PySQLite was one of the first SQLite extensions for Python. Currently, it is a Python DB API 2–compliant extension that supports both SQLite version 2 and version 3 databases. It follows the Python database API specification very closely and includes many additional features as well.

APSW, short for "Another Python SQLite Wrapper," was built specifically to take advantage of Python's newer language features such as iterators, which weren't around when the Python Database API specification was drafted. It uses these language features to provide a simpler way to work with SQLite, often resulting in cleaner, more concise code. Unlike PySQLite, APSW only supports SQLite 3 databases and does not conform to the Python DB API specification, nor does it have any intention of ever doing so.

The reason I cover both extensions is because APSW capitalizes on new language features (iterators and generators) that make for a simpler and more powerful database interface. To be fair, PySQLite version 2, written after APSW, supports these language features as well. One of the other arguments for APSW is its small footprint and simplicity. Which extension you choose depends on your requirements and/or personal preference. Both extensions compile and run on Windows and POSIX systems alike.

PySQLite

I wrote the original version of PySQLite and started the project in 2002, which then was written using the SQLite 2.x API. Gerhard Häring has since taken over the project and completely rewritten it for PySQLite version 2, which supports SQLite version 3. Project information for PySQLite can be found at `http://pysqlite.org`. Both the source code and native Windows binaries are available for download.

Installation

Windows users are encouraged to use the provided binaries rather than building them by hand. On POSIX systems, you build and install PySQLite using Python's `distutils` package. The prerequisites are a C compiler, Python 2.3 or later, and SQLite 3.1 or later. After unpacking the source code, compile the code with the following command:

```
python setup.py build
```

`distutils` will normally be able to figure out where everything is on your system. If not, you will get an error, in which case you need to edit the `setup.cfg` file in the main directory to point out where things are. If everything goes well, you should see "Creating library..." in the output, indicating a successful build. Next install the library:

```
python setup.py install
```

To test the install, change to another directory and do the following:

```
mike@linux # python
>>> from pysqlite2 import test
>>> test.test()
..................................................................................
............
----------------------------------------------------------------------
Ran 101 tests in 0.182s
```

You should not see any errors. If you do, report them to the PySQLite bug tracker at `http://initd.org/tracker/pysqlite`.

For Windows systems, simply download the binary distribution and run the installer. The only prerequisite in this case is that SQLite 3.2 or higher be installed on your system.

Connecting

The PySQLite package is in `pysqlite2`. The DB API module is located in `dbapi2`. You connect to a database using the module's `connect()` function, passing in the relative name or complete file path of the database file you wish to open, as follows:

```
from pysqlite2 import dbapi2 as sqlite

db = sqlite.connect("foods.db")
```

The usual rules apply: you can use in-memory databases by passing in the name `:memory:` as the database name; new databases are created for new files; and so forth.

Query Processing

Query execution is done according to the Python DB API Specification 2. You create a cursor object with which to run the query and obtain results. Internally, a cursor object holds a `sqlite3_stmt` handle. Listing 8-9 illustrates executing a basic query.

Listing 8-9. *A Basic Query in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite

# Connect
con = sqlite.connect("foods.db")

# Prepare a statement and execute. This calls sqlite3_prepare() and sqlite3_step()
cur = con.cursor()
cur.execute('SELECT * FROM foods LIMIT 10')
```

```
# Iterate over results, print the name field (row[2])
row = cur.fetchone()
while row:
    print row[2]
    # Get next row
    row = cur.fetchone()

cur.close()
con.close()
```

Running this code produces the following output:

```
Bagels
Bagels, raisin
Bavarian Cream Pie
Bear Claws
Black and White cookies
Bread (with nuts)
Butterfingers
Carrot Cake
Chips Ahoy Cookies
Chocolate Bobka
```

Query compilation and the first step of execution are performed together in `Cursor.execute()`. It calls `sqlite3_prepare()` followed by `sqlite3_step()`. For modifying queries, this completes the query (short of finalizing the statement handle). For `SELECT` statements, it fetches the first row of the result. The `close()` method finalizes the statement handle.

PySQLite 2 supports iterator-style result sets, similar to APSW. For example, Listing 8-9 could be rewritten as shown in Listing 8-10.

Listing 8-10. *An Iterator-Style Query*

```
from pysqlite2 import dbapi2 as sqlite

con = sqlite.connect("foods.db")
cur = con.cursor()
cur.execute('SELECT * FROM foods LIMIT 10')

for row in cur:
    print row[2]
```

Parameter Binding

PySQLite supports parameter binding by both position and name. You do this by using `Cursor.execute()`. Compilation, binding, and the first step of execution are performed in the one call to `execute()`. You specify positional parameters by passing a tuple as the second argument

to `execute()`. You specify named parameters by passing a dictionary rather than a tuple. Listing 8-11 illustrates both forms of parameter binding.

Listing 8-11. *Parameter Binding in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite

con = sqlite.connect("foods.db")
cur = con.cursor()
cur.execute('INSERT INTO episodes (name) VALUES (?)', ('Soup Nazi'))
cur.close()

cur = con.cursor()
cur.execute('INSERT INTO episodes (name) VALUES (:name)', {'name':'Soup Nazi'})
cur.close()
```

This model does not support reuse of compiled queries (`sqlite3_reset()`). To do this, use `Cursor.executemany()`. While the DB API specifies that `executemany` should take a list of parameters, PySQLite extends it to accommodate iterators and generators as well. For example, the canonical form is shown in Listing 8-12.

Listing 8-12. *The executemany() Method*

```
from pysqlite2 import dbapi2 as sqlite

con = sqlite.connect("foods.db")
cur = con.cursor()

episodes = ['Soup Nazi', 'The Fusilli Jerry']
cur.executemany('INSERT INTO episodes (name) VALUES (?)', episodes)
cur.close()
con.commit()
```

But you could just as easily use a generator, as shown in Listing 8-13.

Listing 8-13. *The executemany() Method with a Generator*

```
from pysqlite2 import dbapi2 as sqlite

con = sqlite.connect("foods.db")
cur = con.cursor()

def episode_generator():
    episodes = ['Soup Nazi', 'The Fusilli Jerry']
    for episode in episodes:
        yield (episode,)

cur.executemany('INSERT INTO episodes (name) VALUES (?)', episode_generator())
cur.close()
con.commit()
```

In both Listing 8-12 and Listing 8-13, the query is compiled only once and reused for each item in the sequence or iterator. This improves overall performance when you are running a large batch of identical queries.

TRANSACTION HANDLING IN PYSQLITE

It is important to notice the `con.commit()` lines in the previous examples. Don't forget them if you use PySQLite. If you don't include them, then all transactions in the examples will be rolled back. This behavior differs from both the SQLite C API and other extensions, and it might surprise you. By default, SQLite runs in autocommit mode. This means that if you don't explicitly start a transaction, then every statement is automatically run in its own transaction. Thus, if you execute a successful `INSERT`, the record is inserted, end of story. PySQLite, however, starts and finishes transactions behind the scenes, using its own logic to determine when to start and commit transactions based on what kind of SQL you execute. Before passing your SQL statement to SQLite, PySQLite analyzes it. If you issue an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE`, PySQLite will implicitly start a transaction (issue a `BEGIN`). Then, if you issue any other kind of command following it, PySQLite will automatically issue a `COMMIT` before executing that command. Therefore, if you issue an `INSERT` statement, and then close the connection, your `INSERT` will never make it to the database unless you manually commit it, or strangely enough—run a `SELECT` statement after it. Furthermore, this transaction logic places an additional constraint on your code in that you cannot use `ON CONFLICT ROLLBACK`; otherwise it interferes with PySQLite's monitoring of your transactions.

However, it is possible to turn off this behavior and restore the default SQLite behavior. To do so, you must explicitly turn autocommit mode back on by setting `isolation_level` of the database connection to `None`, as follows:

```
from pysqlite2 import dbapi2 as sqlite

# Turn on autocommit mode
con = sqlite.connect("foods.db", isolation_level=None)
```

User-Defined Functions and Aggregates

You register user-defined functions using `create_function()`, defined as follows:

```
con.create_function(name, args, pyfunc)
```

Here `name` is the SQL name of the function, `args` the number of arguments accepted by the function, and `pyfync` the Python function that implements the SQL function. Listing 8-14 shows a Python implementation of `hello_newman()` using PySQLite.

Listing 8-14. *hello_newman() in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite

def hello_newman():
    return 'Hello Jerry'
```

```
con = sqlite.connect(":memory:")
con.create_function("hello_newman", 0, hello_newman)
cur = con.cursor()

cur.execute("select hello_newman()")
print cur.fetchone()[0]
```

If you use -1 as the number of arguments, then the function will accept a variable number of arguments. A rendition accepting multiple arguments is illustrated in Listing 8-15.

Listing 8-15. *hello_newman() in PySQLite with Variable Arguments*

```
import string
from pysqlite2 import dbapi2 as sqlite

def hello_newman(*args):
    if len(args) > 0:
        return 'Hello %s' % string.join(args, ', ')
    return 'Hello Jerry'

con = sqlite.connect(":memory:")
con.create_function("hello_newman", -1, hello_newman)
cur = con.cursor()

cur.execute("select hello_newman()")
print cur.fetchone()[0]

cur.execute("select hello_newman('Elaine')")
print cur.fetchone()[0]

cur.execute("select hello_newman('Elaine', 'Jerry')")
print cur.fetchone()[0]

cur.execute("select hello_newman('Elaine', 'Jerry', 'George')")
print cur.fetchone()[0]
```

This produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

Aggregates

PySQLite implements user-defined aggregates as Python classes. These classes must implement `step()` and `finalize()` methods. You register aggregates using `Connection::create_aggregate()`, which takes three arguments: the function's SQL name, the number of arguments, and the class name that implements the aggregate. Listing 8-16 implements a simple `SUM` aggregate called `pysum`.

Listing 8-16. *The pysum() Aggregate in PySQLite*

```
from pysqlite2 import dbapi2 as sqlite

class pysum:
    def init(self):
        self.sum = 0

    def step(self, value):
        self.sum += value

    def finalize(self):
        return self.sum

con = sqlite.connect("foods.db")
con.create_aggregate("pysum", 1, pysum)
cur = con.cursor()
cur.execute("select pysum(id) from foods")
print cur.fetchone()[0]
```

APSW

APSW is written and maintained by Roger Binns. Detailed information on APSW can be found at `www.rogerbinns.com/apsw.html`. Among the other information you will find there is a longer, more detailed list of differences between APSW and PySQLite.

Installation

Like PySQLite, building APSW requires that you have a working C compiler installed on your system, Python, and SQLite version 3. Similarly, once the source is unpacked, building with Python on POSIX systems can be performed by a single command.

```
python setup.py install
```

Building with MinGW on Windows is about as simple:

```
python setup.py build --compile=mingw32 install
```

This creates a single extension—`apsw.so` (Linux/Mac) or `apsw.pyd` (Windows)—which can either be installed in the Python site packages directory or simply dropped into the same directory as the scripts that use it.

Connecting

You create a database connection by instantiating a `Connection` object, passing it the name of the database file as the constructor's sole argument:

```
connection=apsw.Connection("foods.db")
```

All of the normal connection rules apply—the name follows the conventions for filenames on the system; a database name of `:memory:` creates an in-memory database; and so forth.

Query Processing

You execute commands using `Cursor` objects, which are obtained from `Connection.cursor()`. These are very similar to cursors in PySQLite. They are thin wrappers over SQLite statement handles. `Cursor.execute()` can be used as an iterator with which to traverse records in the result set:

```
import apsw

connection=apsw.Connection("foods.db")
cursor=connection.cursor()

for row in cursor.execute("SELECT * FROM foods LIMIT 10"):
    print row[2]
```

You can use a more traditional loop to obtain records through `Cursor.next()`, which returns a tuple containing the fields of the next record.

APSW automatically converts SQLite internal storage classes to similar Python types. Listing 8-17 illustrates this mapping.

Listing 8-17. *Type Mapping in APSW*

```
import apsw
connection=apsw.Connection(":memory:")
cursor=connection.cursor()

c = connection.cursor()
c.execute('create table types(i int, f float, t text, b blob, n)')
c.execute("insert into types values (1, 1.1, '1.1', X'01', NULL)")

for row in cursor.execute('SELECT * FROM types'):
    d = cursor.getdescription()
    i = 0
    for field in row:
        print "%s %5s %-6s %-10s" % ( d[i][0],
                                      d[i][1],
                                      str(field),
                                      type(field) )
        i += 1
```

This produces the following output:

```
i   int  1      <type 'int'>
f float  1.1    <type 'float'>
t  text  1.1    <type 'str'>
b  blob         <type 'buffer'>
n  None  None   <type 'NoneType'>
```

User-Defined Functions and Aggregates

You create user-defined functions using `Connection.createscalarfunction()`, which takes two arguments: the SQL function name and the Python function name. The number of arguments is always variable. Listing 8-18 illustrates an example of `hello_newman()` using APSW.

Listing 8-18. *hello_newman() in APSW*

```
import apsw, string

connection=apsw.Connection(":memory:")

def hello_newman(*args):
    if len(args) > 0:
        return 'Hello %s' % string.join(args, ', ')
    return 'Hello Jerry'

connection.createscalarfunction("hello_newman", hello_newman)

c = connection.cursor()
print c.execute("select hello_newman()").next()[0]
print c.execute("select hello_newman('Elaine')").next()[0]
print c.execute("select hello_newman('Elaine', 'Jerry')").next()[0]
print c.execute("select hello_newman('Elaine', 'Jerry', 'George')").next()[0]
```

Aggregates

You register aggregates using a factory function that specifies the step and finalize functions. The best way to demonstrate this is by example. Consider the example shown in Listing 8-19.

Listing 8-19. *The pysum() Aggregate in APSW*

```
import apsw, string

connection=apsw.Connection("foods.db")

def step(context, *args):
    context['value'] += args[0]
```

```
def finalize(context):
    return context['value']

def pysum():
    return ({'value' : 0}, step, finalize)

connection.createaggregatefunction("pysum", pysum)

c = connection.cursor()
print c.execute("select pysum(id) from foods").next()[0]
```

The `pysum` function is used to define the step and finalize functions as well as initialize a context. The context here is a dictionary with one entry called `'value'`, whose initial value is 0. The step function adds to this value the first argument passed to the function. The finalize function simply returns the value built up over the step function iterations.

Ruby

The Ruby extension was written by Jamis Buck of 37 Signals. Detailed information on the extension as well as the source code can be obtained at `http://rubyforge.org/projects/sqlite-ruby`. There are extensions for both SQLite 2 and 3 databases. The complete documentation for the SQLite 3 extension can be found at `http://sqlite-ruby.rubyforge.org/sqlite3/` whereas SQLite 2 is at `http://sqlite-ruby.rubyforge.org/sqlite/`.

Installation

The extension can be built in two ways: with or without SWIG. The Ruby configuration script will automatically figure out if SWIG is available on your system. You should build the extension with SWIG (the C extension), as that implementation is more stable.

At the time of this writing, the current version of `sqlite-ruby` is 1.1.0. To build from source, fetch the tarball from `http://rubyforge.org/projects/sqlite-ruby`. On POSIX systems, simply unpack the tarball and run three setup commands:

```
mike@linux $ tar xjvf sqlite3-ruby-1.1.0.tar.bz2
mike@linux $ cd sqlite3-ruby-1.1.0
mike@linux $ ruby setup.rb config
mike@linux $ ruby setup.rb setup
mike@linux $ ruby setup.rb install
```

If you have Ruby gems installed, you can get Ruby to do everything for you in one step:

```
mike@linux $ gem install --remote sqlite3-ruby
```

Connecting

To load the SQLite extension, you must import the `sqlite3` module, using either `load` or `require`, as follows:

```
require 'sqlite3'
```

You connect to a database by instantiating a `SQLite3::Database` object, passing in the name of the database file. By default, columns in result sets are accessible by their ordinal. However, they can be accessed by column name by setting `Database::results_as_hash` to `true`:

```
require 'sqlite3'
db = SQLite3::Database.new("foods.db")
db.results_as_hash = true
```

Query Processing

The Ruby extension follows the SQLite API quite closely. It offers both prepared queries and wrapped queries.

Prepared queries are performed via `Database::prepare()`, which passes back a `Statement` object, which holds a `sqlite3_stmt` structure. You execute the query using `Statement`'s `execute` method, which produces a `ResultSet` object. You can pass the `Statement` a block, in which case it will yield the `ResultSet` object to the block. If you don't use a block, then the `Statement` will provide the `ResultSet` object as a return value. `ResultSet` is used to iterate over the returned rows. Internally, it uses `sqlite3_step()`. You can get at the rows in `ResultSet` either through a block using the `each()` iterator, or by using a conventional loop with the `next()` method, which returns the next record in the form of an array, or `nil` if it has reached the end of the set. Listing 8-20 illustrates prepared queries in Ruby.

Listing 8-20. *Prepared Queries in Ruby*

```
#!/usr/bin/env ruby

require 'sqlite3'

db = SQLite3::Database.new('/tmp/foods.db')

stmt = db.prepare('SELECT name FROM episodes')

stmt.execute do | result |
  result.each do | row |
    puts row[0]
  end
end

result = stmt.execute()
result.each do | row |
  puts row[0]
end

stmt.close()
```

It is important to call `Statement.close()` when you are done to finalize the query.

As a `Statement` object holds a `sqlite3_stmt` structure internally, each subsequent call to `execute` thus reuses the same query, avoiding the need to recompile the query.

Parameter Binding

The Ruby extension supports both positional and named parameter binding. You bind parameters using `Statement::bind_param()` and/or `Statement::bind_params()`. `bind_param()` has the following form:

```
bind_param(param, value)
```

If `param` is a `Fixnum`, then it represents the position (index) of the parameter. Otherwise, it is used as the name of the parameter. `bind_params()` takes a variable number of arguments. If the first argument is a hash, then it uses it to map parameter names to values. Otherwise, it uses each argument as a positional parameter. Listing 8-21 illustrates both forms of parameter binding.

Listing 8-21. *Parameter Binding in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

# Named paramters

stmt = db.prepare('SELECT * FROM foods where name like :name')
stmt.bind_param(':name', '%Peach%')

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# Positional paramters

stmt = db.prepare('SELECT * FROM foods where name like ? OR type_id = ?')
stmt.bind_params('%Bobka%', 1)

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# Free read lock
stmt.close()
```

If you don't need to use parameters, a shorter way to process queries is using `Database::query()`, which cuts out the `Statement` object and just returns a `ResultSet`, as shown in Listing 8-22.

Listing 8-22. *Using the Database::query() Method in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

result = db.query('SELECT * FROM foods limit 10')
result.each do |row|
  puts row['name']
end

result.reset()

while row = result.next
  puts row['name']
end
result.close()
```

Like `Statement` objects, `Result` objects are also thin wrappers over statement handles, and therefore represent compiled queries. They can be rerun with a call to `reset`, which calls `sqlite3_reset()` internally and reexecutes the query. Unlike `Statement` objects however, they cannot be used for bound parameters.

Other, even shorter, query methods include `Database::get_first_row()`, which returns the first row of a query, and `Database::get_first_value()`, which returns the first column of the first row of a query.

User-Defined Functions and Aggregates

User-defined functions are implemented using `Database::create_function()`, which has the following form:

```
create_function( name, args, text_rep=Constants::TextRep::ANY ) {|func, *args| ...}
```

Here `name` is the name of the SQL function, `args` the number of arguments (`-1` is variable), and `text_rep` corresponds to the UTF encoding. Values are `UTF8`, `UTF16LE`, `UTF16BE`, `UTF16`, and `ANY`. Finally, the function implementation is defined in the block. Listing 8-23 illustrates a Ruby implementation of `hello_newman()`.

Listing 8-23. *hello_newman() in Ruby*

```
require 'sqlite3'

db = SQLite3::Database.new(':memory:')

db.create_function('hello_newman', -1 ) do |func, *args|
  if args.length == 0
    func.result = 'Hello Jerry'
```

```
  else
    func.result = 'Hello %s' % [args.join(', ')]
  end
end

puts db.get_first_value("SELECT hello_newman()")
puts db.get_first_value("SELECT hello_newman('Elaine')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry', 'George')")
```

This program produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

Aggregates

There are two ways to implement aggregates. One uses blocks, and the other a class. The block approach has embedded in it both the step function and the finalize function. Listing 8-24 illustrates this approach.

Listing 8-24. *The rubysum() Aggregate*

```
require 'sqlite3'

db = SQLite3::Database.new(':memory:')

db.create_aggregate( "rubysum", 1 ) do
    step do |func, value|
      func[ :total ] ||= 0
      func[ :total ] += value.to_i
    end

    finalize do |func|
      func.set_result( func[ :total ] || 0 )
    end
  end

puts db.get_first_value( "SELECT rubysum(id) FROM episodes " )
```

The class approach is similar to PySQLite and Perl, in which you create an aggregate class that has a step and a finalize method. You register the class using `Database::create_aggregate_handler()`. The class implements methods that provide all the needed information to register and run the aggregate. Two class methods and two instance methods are required. The class methods are

- `name()`: This specifies the name of the SQL function. The handler must implement this message.
- `arity()`: This is optional, and specifies the arity (number of arguments) of the function. If the handler does not respond to it, the function will have an arity of -1.

The instance methods are the step function and finalize function. Listing 8-25 illustrates this approach.

Listing 8-25. *A Class Implementation of rubysum()*

```
class RubySumAggregateHandler
  def self.arity; 1; end
  def self.name; "rubysum"; end

  def initialize
    @total = 0
  end

  def step( ctx, name )
    @total += ( name ? name.length : 0 )
  end

  def finalize( ctx )
    ctx.set_result( @total )
  end
end

db.create_aggregate_handler(RubySumAggregateHandler)
puts db.get_first_value("SELECT rubysum(id) FROM episodes")
```

Java

Several Java extensions are available for SQLite. The one covered here is written by Christian Werner, who wrote the SQLite ODBC driver as well. It includes both a JDBC driver and a native JNI extension, which closely shadows the SQLite C API. The extension can be downloaded from `www.ch-werner.de/javasqlite/overview-summary.html#jdbc_driver`.

The main class in the JNI extension is `SQLiteDatabase`. Most of its methods are implemented using callbacks that reference the following interfaces:

- SQLite.Callback
- SQLite.Function
- SQLite.Authorizer
- SQLite.Trace
- SQLite.ProgressHandler

The SQLite.Callback interface is used to process result sets through row handlers, as well as column and type information. SQLite.Authorizer is a thin wrapper over the SQLite C API function sqlite3_set_authorizer(), with which you can intercept database events before they happen. SQLite.Trace is used to view SQL statements as they are compiled, and wraps sqlite3_trace(). SQLite.ProgressHandler wraps sqlite3_progress_handler(), which is used to issue progress events after a specified number of VDBE instructions have been processed.

Installation

The current version requires JDK 1.1 or higher. The extension uses GNU Autoconf, so building and installing requires only three steps:

```
mike@linux $ javasqlite-20050608.tar.gz
mike@linux $ ./configure
mike@linux $ make
mike@linux $ make install
```

The configure script will look for SQLite and the JDK in several default locations. However, to explicitly specify where to look for SQLite and the JDK, use the command-line options --with-sqlite=DIR, --with-sqlite3=DIR, and --with-jdk=DIR. To specify the place where the resulting library should be installed (libsqlite_jni.so file), use the --prefix=DIR option. On POSIX systems, the default location is /usr/local/lib (i.e., the prefix defaults to /usr/local). To specify where the sqlite.jar is to be installed, use the --with-jardir=DIR option. On POSIX systems, the default is /usr/local/share/java. This file contains the high-level part and the JDBC driver. At runtime, it is necessary to tell the JVM both places with the -classpath and -Djava.library.path=.. command-line options.

For Windows, the makefiles javasqlite.mak and javasqlite3.mak are provided in the distribution. They contain some build instructions and use the J2SE 1.4.2 from Sun and Microsoft Visual C++ 6.0. A DLL with the native JNI part (including SQLite 3.2.1) and the JAR file with the Java part can be downloaded from the website.

Connecting

You connect to a database using SQLiteDatabase::open(), which takes the name of the database and the file mode, as shown in Listing 8-26. The file mode is an artifact from SQLite 2 API and is no longer used. You can provide any value to satisfy the function. The code in Listing 8-26 is taken from the example SQLiteJNIExample.java, which illustrates all main facets of database operations: connecting, querying, using functions, and so forth.

Listing 8-26. *The JavaSQLite Test Program*

```
import SQLite.*;

public class SQLiteJNIExample
{
    public static void main(String args[])
    {
        SQLite.Database db = new SQLite.Database();

        try
        {
            db.open("foods.db", 700);

            // Trace SQL statements
            db.trace(new SQLTrace());

            // Query example
            query(db);

            // Function example
            user_defined_function(db);

            // Aggregate example
            user_defined_aggregate(db);

            db.close();
        }
        catch (java.lang.Exception e)
        {
            System.err.println("error: " + e);
        }
    }

    ...
}
```

Query Processing

You perform queries using `SQLiteDatabase::compile()`. This function can process a string containing multiple SQL statements. It returns a `VM` (virtual machine) object that holds all of the statements.

The `VM` object parses each individual SQL statement on each call to `compile()`, returning `true` if a complete SQL statement was compiled, and `false` otherwise. You can therefore iterate through all SQL statements in a loop, breaking when the `VM` has processed the last statement.

When the `VM` has compiled a statement, you can execute it using `VM::step()`. This function takes a single object, which implements a `SQLite.Callback` interface. The example uses a class called `Row` for this purpose, which is shown in Listing 8-27.

Listing 8-27. *The Row Class*

```
class Row implements SQLite.Callback
{
    private String row[];

    public void columns(String col[]) {}
    public void types(String types[]) {}

    public boolean newrow(String data[])
    {
        // Copy to internal array
        row = data;
        return false;
    }

    public String print()
    {
        return "Row:    [" + StringUtil.join(row, ", ") + "]";
    }
}
```

The `SQLite.Callback` interface has three methods: `columns()`, `types()`, and `newrow()`. They process the column names, column types, and row data, respectively. Each call to `VM::step()` updates all of the column, type, and row information.

The use of `VM` is illustrated in the `query()` function of the example, which is listed in Listing 8-28.

Listing 8-28. *The query() Function*

```
public static void query(SQLite.Database db)
    throws SQLite.Exception
{
    System.out.println("\nQuery Processing:\n");

    Row row = new Row();
    db.set_authorizer(new AuthorizeFilter());

    Vm vm = db.compile( "select * from foods LIMIT 5;" +
                        "delete from foods where id = 5;" +
                        "insert into foods (type_id, name) values (5, 'Java');" +
                        "select * from foods LIMIT 5" );
    do
    {
        while (vm.step(row))
        {
            System.err.println(row.print());
        }
    }
    while (vm.compile());
}
```

The SQLite.Database::exec() performs self-contained queries and has the following form:

```
void exec(String sql, Callback cb, String[] params)
```

The params array corresponds to %q or %Q parameters in the SQL statement. An example is shown in Listing 8-29.

Listing 8-29. *The exec_query() Function*

```
public static void exec_query(SQLite.Database db)
    throws SQLite.Exception
{
    System.out.println("\nExec Query:\n");

    String sql = "insert into foods (type_id, name) values (5, '%q')";
    ResultSet result = new ResultSet();

    String params[] = {"Java"};
    db.exec(sql, result, params);

    System.out.println("Result: last_insert_id(): " + db.last_insert_rowid());
    System.out.println("Result:        changes(): " + db.changes());
}
```

Note that this is not the same thing as parameter binding. Rather, this is sprintf style substitution using sqlite3_vmprintf() in the SQLite C API.

User-Defined Functions and Aggregates

The SQLite.Function interface is used to implement both user-defined functions and user-defined aggregates. The hello_newman() function in Java is illustrated in Listing 8-30.

Listing 8-30. *hello_newman() in Java*

```
class HelloNewman implements SQLite.Function
{
    public void function(FunctionContext fc, String args[])
    {
        if (args.length > 0)
        {
            fc.set_result("Hello " + StringUtil.join(args, ", "));
        }
        else
        {
            fc.set_result("Hello Jerry");
        }
    }
```

```
    public void step(FunctionContext fc, String args[]){}
    public void last_step(FunctionContext fc)
    {
        fc.set_result(0);
    }

}
```

Notice that the `step()` and `last_step()` functions, while specific to aggregates, must also be implemented even though they do nothing in user-defined functions. This is because the `SQLite.Function` interface defines methods for both functions and aggregates. This class is registered using `SQLite.Database.create_function()`. The function's return type must also be registered using `SQLite.Database.function_type()`. Listing 8-31 illustrates using the Java implementation of `hello_newman()`.

Listing 8-31. *The hello_newman() Test Code*

```
// Register function
db.create_function("hello_newman", -1, new HelloNewman());

// Set return type
db.function_type("hello_newman", Constants.SQLITE_TEXT);

// Test
PrintResult r = new PrintResult();
db.exec("select hello_newman()", r);
db.exec("select hello_newman('Elaine', 'Jerry')", r);
db.exec("select hello_newman('Elaine', 'Jerry', 'George')", r);
```

JDBC

The Java extension also includes support for JDBC. To use the driver, specify `SQLite.JDBCDriver` as the JDBC driver's class name. Also, make sure that you have `sqlite.jar` in your class path and the native library in your Java library path. The JDBC URLs to connect to a SQLite database have the format `jdbc:sqlite:/path`, where `path` has to be specified as the path name to the SQLite database, for example

```
jdbc:sqlite://dirA/dirB/dbfile
jdbc:sqlite:/DRIVE:/dirA/dirB/dbfile
jdbc:sqlite:///COMPUTERNAME/shareA/dirB/dbfile
```

Currently, the supported data types on SQLite tables are `java.lang.String`, `short`, `int`, `float`, and `double`. Some support exists for `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp` although it is mostly untested. The data type mapping depends mostly on the availability of the SQLite pragmas `show_datatypes` and `table_info`. Enough basic database metadata methods are implemented such that it is possible to access SQLite databases with JDK 1.3 or higher and the iSQL-Viewer tool.

Listing 8-32 is a simple example (located in `SQLiteJDBCExample.java`) of using the JDBC driver to query the `foods` table.

Listing 8-32. *The SQLite JDBC Test Program*

```
import java.sql.*;
import SQLite.JDBCDriver;

public class SQLiteJDBCExample {

public static void main ( String [ ] args )
{
    try
    {
        Class.forName("SQLite.JDBCDriver");
        Connection c = DriverManager.getConnection( "jdbc:sqlite://tmp/foods.db",
                                                    ""  // username (NA),
                                                    ""  // password (NA));

        Statement s = c.createStatement();
        ResultSet rs = s.executeQuery ("SELECT * FROM foods LIMIT 10");
        int cols = (rs.getMetaData()).getColumnCount();

        while (rs.next())
        {
            String fields[] = new String[cols];

            for(int i=0; i<cols; i++)
            {
                fields[i] = rs.getString(i+1);
            }

            System.out.println("[" + join(fields, ", ") + "]");
        }
    }
    catch( Exception x )
    {
        x.printStackTrace();
    }
}

static String join( String[] array, String delim )
{
    StringBuffer sb = join(array, delim, new StringBuffer());
    return sb.toString();
}

static StringBuffer join( String[] array, String delim, StringBuffer sb )
{
```

```
    for ( int i=0; i<array.length; i++ )
    {
        if (i!=0) sb.append(delim);
        sb.append("'" + array[i] + "'");
    }

    return sb;
}

}
```

This example produces the following output:

```
['1', '1', 'Bagels']
['2', '1', 'Bagels, raisin']
['3', '1', 'Bavarian Cream Pie']
['4', '1', 'Bear Claws']
['6', '1', 'Bread (with nuts)']
['7', '1', 'Butterfingers']
['8', '1', 'Carrot Cake']
['9', '1', 'Chips Ahoy Cookies']
['10', '1', 'Chocolate Bobka']
['11', '1', 'Chocolate Eclairs']
```

Tcl

SQLite's author wrote and maintains the Tcl extension. All of SQLite's testing code (more than 30,000 lines) is implemented in Tcl, and thus uses the SQLite Tcl extension. It is safe to say that this extension is both stable and well tested itself. The Tcl extension is included as a part of the SQLite source distribution. Complete documentation for this extension can be found on the SQLite website: `www.sqlite.org/tclsqlite.html`.

Installation

The SQLite GNU Autoconf script will automatically search for Tcl and build the Tcl extension if it finds it. So simply having Tcl installed before you build SQLite should provide you with the SQLite Tcl extension. Nevertheless, precompiled bindings for the Tcl extension for both Linux and Windows are available on the SQLite website.

Connecting

The SQLite extension is located in the `sqlite3` package, which must be loaded using the `package require` directive. To connect to a database, use the `sqlite3` command to create a database handle. This command takes two arguments: the first is the name of the database handle to be created and the second is the path to the database file. The following example illustrates connecting to a database:

```
#!/usr/bin/env tclsh

package require sqlite3

puts "\nConnecting."
sqlite3 db ./foods.db
```

The usual database connection rules apply: passing `:memory:` will create an in-memory database; passing in the name of a new file will create a new database; and so forth. The database handle returned corresponds to a connection to the specified database; however, it is not yet open—it does not open the connection until you try to use it. The database handle is the sole object through which you work with the database.

To disconnect, use the `close` method of the database handle. This will automatically roll back any pending transactions.

Query Processing

The extension executes queries using the `eval` method, which can process one or more queries at a time. `eval` can be used in several ways. The first way is to iterate through all records in a script following the SQL code. The script will be executed once for each row returned in the result set. The fields for each row are set as local variables within the script. For example:

```
puts "\nSelecting 5 records."
db eval {SELECT * FROM foods LIMIT 5} {
    puts "$id $name"
}
```

There is another form in which you can assign the field values to an array. To do this, specify the array name after the SQL and before the script. For example:

```
puts "\nSelecting 5 records."
db eval {SELECT * FROM foods LIMIT 5} values {
    puts "$values(id) $values(name)"
}
```

Both of the above code snippets produce the following output:

```
Selecting 5 records.
1 Bagels
2 Bagels, raisin
3 Bavarian Cream Pie
4 Bear Claws
5 Black and White cookies
```

`eval` will return the result set if no script is provided. The result set is returned as one long list of values, and you must determine the record boundaries. For example, consider the following statement:

```
set x [db eval {SELECT * FROM foods LIMIT 2}]
```

This will return a list in variable $x that is six elements long:

```
[1, 1, 'Bagels', 2, 1, 'Bagels, raisin']
```

This corresponds to two records, each of which has three fields (id, type_id, and name).

For non-SELECT statements, eval returns information regarding modified records, as illustrated in Listing 8-33.

Listing 8-33. *Examining Changes in Tcl*

```
db eval BEGIN

puts "\nUpdating all rows."
db eval { UPDATE foods set type_id=0 }
puts "Changes             : [db changes]"

puts "\nDeleting all rows."
# Delete all rows
db eval { DELETE FROM foods }

puts "\nInserting a row."
# Insert a row
db eval { INSERT INTO foods (type_id, name) VALUES (9, 'Junior Mints') }

puts "Changes             : [db changes]"
puts "last_insert_rowid() : [db last_insert_rowid]"

puts "\nRolling back transaction."
db eval ROLLBACK
```

The code in Listing 8-33 produces the following output:

```
Updating all rows.
Changes             : 415

Deleting all rows.

Inserting a row.
Changes             : 1
last_insert_rowid() : 1

Rolling back transaction.
Total records       : 415
```

Transaction scope can be automatically handled within Tcl code using the transaction method. If all code inside the transaction method's script runs without error, the transaction method will commit; otherwise, it will roll back. For example, if you wanted to perform the code from Listing 8-33 in a single transaction, you would have to check the status of each command

after it is executed, and if it failed, then you would roll back the transaction and abort any further commands. The more commands you have to run in the transaction, the messier the code will get. However, all of this can be done automatically with the `transaction` method, as illustrated in Listing 8-34.

Listing 8-34. *Transaction Scope in Tcl*

```
db transaction {
puts "\nUpdating all rows."
db eval { UPDATE foods set type_id=0 }
puts "Changes             : [db changes]"

puts "\nDeleting all rows."
# Delete all rows
db eval { DELETE FROM foods }

puts "\nInserting a row."
# Insert a row
db eval { INSERT INTO foods (type_id, name) VALUES (9, 'Junior Mints') }

puts "Changes             : [db changes]"
puts "last_insert_rowid() : [db last_insert_rowid]"
}
```

Now, if any of the commands fail, `transaction` will roll back all commands without having to check any return codes. Furthermore, `transaction` also works with existing transactions. That is, if a transaction is already started, it will work within that transaction, and not attempt a commit or rollback. If an error occurs, it just aborts the script, returning the appropriate error code.

User-Defined Functions

User-defined functions are created using the `function` method, which takes the name of the function and a Tcl method that implements the function. Listing 8-35 illustrates an implementation of `hello_newman()` in Tcl.

Listing 8-35. *hello_newman() in Tcl*

```
proc hello_newman {args} {
    set l [llength $args]
    if {$l == 0} {
        return "Hello Jerry"
```

```
    } else {
        return "Hello [join $args {, } ]"
    }
}

db function hello_newman hello_newman
puts [db onecolumn {select hello_newman()}]
puts [db onecolumn {select hello_newman('Elaine')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry', 'George')}]
```

This code produces the following output:

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

PHP

Starting with PHP version 5, SQLite became part of the PHP standard library. If you have PHP 5, you have SQLite installed on your system as well. As of PHP 5.1, there are three interfaces in PHP that allow you to work with SQLite.

The first extension—the SQLite extension introduced in version 5—is specific to SQLite version 2 and reflects its API. This extension provides two interfaces: a procedural interface and an object-oriented (OO) interface. For the most part, both interfaces are logically equivalent. The OO interface seems to help in writing simpler, cleaner code, but it doesn't encapsulate all of the functions of the procedural interface.

Since this extension is written explicitly around SQLite version 2 exclusively, there are some slight differences in the way this extension works compared to the other extensions covered in this book.

PHP 5.1 introduced a new database abstraction layer called PHP Data Objects, or PDO for short. This API uses drivers to support a standard database interface. PDO has drivers for both SQLite versions 2 and 3; therefore, SQLite version 3 is accessible via PHP. The PDO interface is an OO interface that is very similar to the OO interface in the SQLite extension. Since PDO is an abstraction layer, it is meant to accommodate many different databases and is therefore somewhat generic. Despite this, it is still possible for PDO drivers to provide access to database specific features as well. As a result, the PDO drivers provide a complete OO interface that works equally well with both SQLite version 2 and version 3.

Because this book is about using SQLite version 3, this section covers the PHP PDO extension for SQLite version 3.

Installation

Installation of PHP, while straightforward, is beyond the scope of this book. Detailed instructions on building and installing PHP can be found in the documentation on the PHP website: `www.php.net`.

Connections

There are two important issues to consider before you open a database: location and permissions. By default, the file path in PHP is relative to the directory in which the script is run (unless you provide a full path, relative to the root filesystem). So if you specify just a database name, you are opening or creating a database within the public area of your website, and the security of that database file depends on how the web server is configured. Since SQLite databases are normal operating system files just like HTML documents or images, it may be possible for someone to fetch them just like a regular document. This could be a potential security problem, depending on the sensitivity of your data. As a general security precaution, it may be a good idea to keep database files outside of public folders, so that only PHP scripts can access them.

With regard to permissions, the relevant file and folder permissions must allow the web server process running PHP both read and write access to the database files. Both of these are administrative details that must be addressed on a case-by-case basis.

In PDO, connections are encapsulated in the `PDO` class. The constructor takes a single argument called a data source name (DSN). The DSN is a colon-delimited string composed of two parameters. The first parameter is the driver name, which is either `sqlite` (which corresponds to SQLite version 3) or `sqlite2` (which corresponds to SQLite version 2). The second parameter is the path to the database file. If the connection attempt fails for any reason, the constructor will throw a `PDOException`. The following example connects to a SQLite version 3 database (`foods.db`):

```
<?php
try {
   $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
   echo 'Connection failed: ' . $e->getMessage();
}?>
```

Queries

PDO is made up of two basic classes—`PDO` and `PDOStatement`. The first represents a connection, which internally contains a `sqlite3` structure, and the other a statement handle, which internally contains a `sqlite3_stmt` structure. The query methods in the PDO class closely follow the methods in the SQLite C API. There are three ways to execute queries:

- `exec()`: Executes queries that don't return any data. Returns the number of affected rows, or `FALSE` if there is an error. This mirrors `sqlite3_exec()`.
- `query()`: Executes a query and returns a `PDOStatement` object representing the result set, or `FALSE` if there is an error.
- `prepare()`: Compiles a query and returns a `PDOStatement` object or `FALSE` if there is an error. This offers better performance than `query()` for statements that need to be executed multiple times, as it can be reset, avoiding the need to recompile the query.

Additionally, transaction management in PDO can be performed through a method invocation using `beginTransaction()`, `commit()`, and `rollback()`. Listing 8-36 shows a basic example of using the PDO class to open a database and perform basic queries within a transaction.

Listing 8-36. *Basic Queries with PDO*

```
<?php

try {
    $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

$dbh->beginTransaction();
$sql = 'SELECT * FROM foods LIMIT 10';
foreach ($dbh->query($sql) as $row) {
    print $row['type_id'] . " ";
    print $row['name']    . "<br>";
}

$dressing = $dbh->quote("Newman's Own Vinegarette");
$dbh->exec("INSERT into FOODS values (NULL, 4, $dressing)");
echo $dbh->lastInsertId();
$dbh->rollback();
?>
```

PDO uses the `setAttribute()` and `getAttribute()` methods to set various database connection parameters. Currently, the only parameter that applies to SQLite is `PDO_ATTR_TIMEOUT`, which sets the busy timeout.

The `prepare()` method uses a SQLite statement to navigate through a result set. It also supports both bound and positional parameters. To bind a SQL parameter, there must be some associated variable in PHP to bind it to. PHP variables are bound to parameters using `PDOStatement::bindParam()`, which has the following declaration:

```
nool bindParam ( mixed parameter, mixed &variable,
                 int data_type, int length,
                 mixed driver_options );
```

The `parameter` argument specifies the SQL parameter. For numbered parameters, this is an integer value. For named parameters, it is a string. The `variable` argument is a reference to a PHP variable to bind. The `data_type` argument specifies the data type of the variable. Finally, the `length` argument specifies the length of the value if the variable is a string. The value specifies the maximum length of the string to return. Listing 8-37 is a complete example of using positional parameters.

Listing 8-37. *Positional Parameters with PDO*

```
<?php
$dbh     = new PDO("sqlite:foods.db");
$sql     = 'SELECT * from FOODS WHERE type_id=? And name=?';
$stmt    = $dbh->prepare($sql);
$type_id = 9;
$name    = 'JujyFruit';
$stmt->bindParam(1, $type_id, PDO_PARAM_INT);
$stmt->bindParam(1, $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

Named parameters work similarly. When you bind these parameters, you identify them by their names (rather than integers). Listing 8-38 is a modification of the previous example using named parameters.

Listing 8-38. *Named Parameters with PDO*

```
<?php
$dbh     = new PDO("sqlite:foods.db");
$sql     = 'SELECT * from FOODS WHERE type_id=:type And name=:name;';
$stmt    = $dbh->prepare($sql);
$type_id = 9;
$name    = 'JujyFruit';
$stmt->bindParam('type', $type_id, PDO_PARAM_INT);
$stmt->bindParam('name', $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

PDO also allows you to bind columns of result sets to PHP variables. This is done using `PDOStatement::bindColumn()`, which has the following declaration:

```
bool bindColumn (mixed column, mixed &param, int type)
```

The `column` argument refers to either the name of the column or its index in the `SELECT` clause. The `param` argument is a reference to a PHP variable, and the `type` argument specifies the type of the PHP variable. The following example binds two variables `$name` and `$type_id` to a result set:

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'SELECT * from FOODS LIMIT 10';
$stmt = $dbh->prepare($sql);
$stmt->execute();
$name;
$type_id;
$stmt->bindColumn('type_id', $type_id, PDO_PARAM_INT);
$stmt->bindColumn('name', $name, PDO_PARAM_STR);
```

```
while ($row = $stmt->fetch()) {
    print "$type_id $name <br>";
}
```

User-Defined Functions and Aggregates

User-defined functions are implemented using `sqliteCreateFunction()`. Aggregates are implemented using `sqliteCreateAggregate()`. `sqliteCreateFunction()` has the following form:

```
void PDO::createFunction ( string function_name,
                           callback callback,
                           int num_args );
```

The arguments are defined as follows:

- `function_name`: The name of the function as it is to appear in SQL
- `callback`: The PHP (callback) function to be invoked when the SQL function is called
- `num_args`: The number of arguments the function takes

The following example is a PHP implementation of `hello_newman()`:

```
<?php
function hello_newman() {
    return 'Hello Jerry';
}

$db = new PDO("sqlite:foods.db");
$db->createFunction('hello_newman', hello_newman, 0);
$row = $db->query('SELECT hello_newman()')->fetch();
print $row[0]
?>
```

You create aggregates in a similar fashion using the `sqliteCreateAggregate` function, declared as follows:

```
void PDO::createAggregate ( string function_name,
                            callback step_func,
                            callback finalize_func,
                            int num_args )
```

The following code is a simple implementation of the above `SUM` aggregate, called `phpsum`:

```
<?php
function phpsum_step(&$context, $value) {
    $context = $context + $value;
}

function phpsum_finalize(&$context) {
    return $context;
}
```

```
$db = new PDO("sqlite:foods.db");
$db->createAggregate('phpsum', phpsum_step, phpsum_finalize);
$row = $db->query('SELECT phpsum(id) from food_types')->fetch();
print $row[0]
?>
```

Summary

This has been a brief survey of several different language extensions and how they work with SQLite. While using SQLite with the C API is quite straightforward, using SQLite in language extensions is considerably easier. Many of the concepts are very similar. As you can see, there are many things in common even in cases where an extension conforms to a language-specific database API. All queries ultimately involve a connection object of some kind, which maps to an internal `sqlite3` structure, and a statement or cursor object, which internally maps to a `sqlite3_stmt` structure.

These extensions, developed by the open source community, make using SQLite convenient and easy, making it accessible to many more applications ranging from system administration to website development. There are plenty more extensions for these and more languages with which to use SQLite. As mentioned earlier, you need only to look on the SQLite Wiki to find them: `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`.