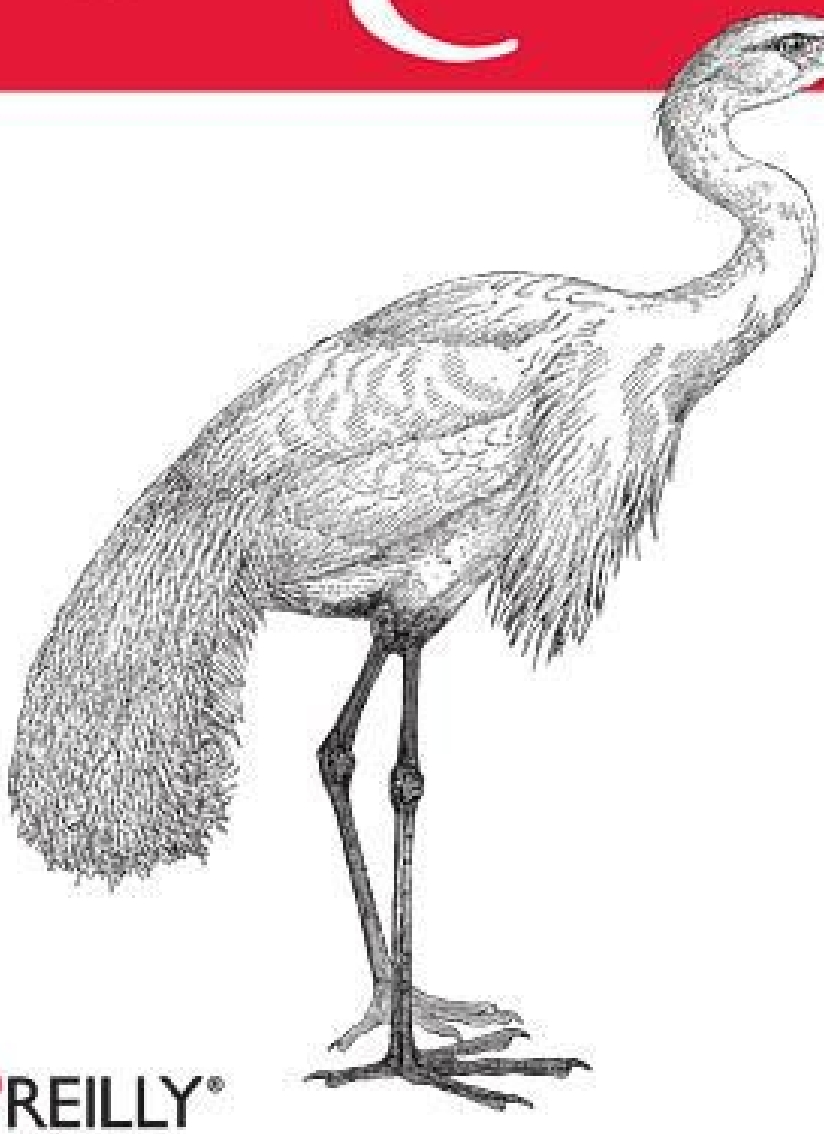
Using

SQLite



O'REILLY®

Jay A. Kreibich

Table of Contents

CHAPTER 8

Additional Features and APIs

This chapter touches on a number of different areas, mostly having to do with features and interfaces that go beyond the basic database engine. The first section covers the SQLite time and date features, which are provided as a small set of scalar functions. We'll also briefly look at some of the standard extensions that ship with SQLite, such as the ICU internationalization extension, the FTS3 text search module, and the R*Tree module. We'll also be looking at some of the alternate interfaces available for SQLite in different scripting languages and other environments. Finally, we'll wrap things up with a quick discussion on some of the things to watch out for when doing development on mobile or embedded systems.

Date and Time Features

Most relational database products have several native datatypes that deal with recording dates, times, timestamps, and durations of all sorts. SQLite does not. Rather than having specific datatypes, SQLite provides a small set of time and date conversion functions. These functions can be used to convert time, date, or duration information to or from one of the more generic datatypes, such as a number or a text value.

This approach fits well with the simple and flexible design goals of SQLite. Dates and times can get extremely complicated. Odd time zones and changing daylight saving time (DST) rules can complicate time values, while dates outside of the last few hundred years are subject to calendaring systems that have been changed and modified throughout history. Creating a native type would require picking a specific calendaring system and a specific set of conversion rules that may or may not be suitable for the task at hand. This is one of the reasons a typical database has so many different time and date datatypes.

Chapter 8. Additional Features and APIs. Using SQLite, ISBN: 9781449394592

Using external conversion functions is much more flexible. The developer can choose a format and datatype that best fits the needs of the application. Using the simpler underlying datatypes is also a much better fit for SQLite's dynamic typing system. A more generic approach also keeps the internal code simpler and smaller, which is a plus for most SQLite environments.

Application Requirements

When creating date and time data values, there are a few basic questions that need to be answered. Many of these seem obvious enough, but skipping over them too quickly can lead to big problems down the road.

First, you need to figure out what you're trying to store. It might be a time of day, such as a standalone hour, minute, second value without any associated date. You may need a date record, that refers to a specific day, month, year, but has no associated time. Many applications require timestamps, which include both a date and time to mark a specific point in time. Some applications need to record a specific day of the year, but not for a specific year (for example, a holiday). Many applications also use durations (time deltas). Even if the application doesn't store durations or offsets, they are often computed for display purposes, such as the amount of time between "now" and a specific event.

It is also worth considering the range and precision required by your application. As already discussed, dates in the far past (or far future) are poorly represented by some calendaring systems. A database that holds reservations for a conference room may require minute precision, while a database that holds network packet dumps may require precision of a microsecond or better.

Representations

As with many other datatypes, the class of data an application needs to store, along with the required range and precision, often drives the decision on what representation to use. The two most common representations used by SQLite are some type of formatted text-based value or a floating-point value.

Julian Day

The simplest and most compact representation is the Julian Day. This is a single floating-point value used to count the number of days since noon, Greenwich time, on 24 November 4714 BCE. SQLite uses the proleptic Gregorian calendar for this representation. The Julian value for midnight, 1 January 2010 is 2455197.5. When stored as a 64-bit floating-point value, modern age dates have a precision a tad better than one millisecond.

Many developers have never encountered the Julian Day calendar, but conceptually it is not much different than the more familiar POSIX `time()` value—it just uses a different value (days, rather than seconds) and a different starting point.

Julian Day values have a relatively compact storage format and are fairly easy to work with. Durations and differences are simple and efficient to calculate, and use the same data representation as points in time. Values are automatically normalized and can be utilize simple mathematic operations. Julian values are also able to express a very broad range of dates, making them useful for historic records. The main disadvantage is that they require conversion before being displayed.

Text values

The other popular representation is a formatted text value. These are typically used to hold a date value, a time value, or a combination of both. Although SQLite recognizes a number of formats, most commonly dates are given in the format `YYYY-MM-DD`, while times are formatted `HH:MM:SS`, using an hour value of 00 to 23. If a full timestamp is required, these values can be combined. For example, `YYYY-MM-DD HH:MM:SS`. Although this style of date may not be the most natural representation, these formats are based off the ISO 8601 international standard for representing dates and times. They also have the advantage of sorting chronologically using a simple string sort.

The main advantage of using a text representation is that they are very easy to read. The stored values do not require any kind of translation and can be easily browsed and understood in their native format. You can also pick and choose what parts of the data value are required, storing only a date or only a time of day, making it a bit more clear about the intention of the value. Or, at least, that would be true if it wasn't for the time zone issue. As we'll see, times and dates are rarely stored relative to the local time zone, so even text values usually require conversion before being displayed.

The major disadvantage of text values is that any operation (other than display) requires a significant amount of data conversion. Time and date conversions require some complex math, and can make a noticeable impact in the performance of some applications. For example, moving a date one week into the future requires a conversion of the original date into some generalized format, offsetting the value, and converting it back into an appropriate text value. Calculating durations also requires a significant amount of conversion. The conversion cost may not be significant for a simple update or insert, but it can make a very noticeable difference if found in a search conditional.

Text values also require careful normalization of all input values into a standardized format. Many operations, such as sorts and simple comparisons, require that values use the *exact* same format. Alternate formats can result in equivalent time values being represented by nonequivalent text values. This can lead to inconsistent results from any procedures that operate directly on the text representation. The problem is not just limited to single columns. If a time or date value is used as a key or join column, these operations will only work properly if all of the time and date values use the same format.

For all these concerns, there is still no denying that text values are the easiest to display and debug. While there is significant value in this, make sure you consider the full range of pros and cons of text values (or any other format) before you make a choice.

Time zones

You may have noticed that none of these formats support a time zone field. SQLite assumes all time and date information is stored in *UTC*, or Coordinated Universal Time. UTC is essentially Greenwich Mean Time, although there are some minor technical differences.

There are some significant advantages to using UTC time. First and foremost, UTC is unaffected by location. This may seem like a minor thing, but if your database is sitting on a mobile device, it is going to move. Occasionally, it is going to move across time zones. Any displayed time value better shift with the device.

If your database is accessible over the Internet, chances are good it will be accessed from more than one time zone. In short, you can't ignore the time zone issue, and sooner or later you're going to have to translate between time zones. Having a universal base format makes this process much easier.

Similarly, UTC is not affected by Daylight Saving Time . There are no shifts, jumps, or repeats of UTC values. DST rules are extremely complex, and can easily differ by location, time of year, or even the year itself, as switch-over times are shifted and moved. DST essentially adds a second, calendar-sensitive time zone to any location, compounding the problems of location and local time conversions. All of these issues can create a considerable number of headaches.

In the end, there are very few justifiable reasons to use anything except UTC. As the name implies, it provides a universal time system that best represents unique moments in time without any context or translation. It might seem silly to convert values to UTC as you input them, and convert them back to local time to display them, but it has the advantage of working correctly, even if the local time zone changes or the DST state changes. Thankfully, SQLite makes all of these conversions simple.

Time and Date Functions

Nearly all of the time and date functionality within SQLite comes from five SQL functions. One of these functions is essentially a universal translator, designed to convert nearly any time or date format into any other format. The other four functions act as convenience wrappers that provide a fixed, predefined output format.

In addition to the conversion functions, SQLite also provides a three literal expressions. When an expression is evaluated, these literals will be translated into an appropriate time value that represents "now."

Conversion Function

The main utility to manipulate time and date values is the `strftime()` SQL function:

```
strftime( format, time, modifier, modifier... )
```

The `strftime()` SQL function is modeled after the POSIX `strftime()` C function. It uses `printf()` style formatting markers to specify an output string. The first parameter is the format string, which defines the format of the returned text value. The second parameter is a source time value that represents the base input time. This is followed by zero or more modifiers that can be used to shift or translate the input value before it is formatted. Typically all of these parameters are text expressions or text literals, although the time value may be numeric.

In addition to any literal characters, the format string can contain any of the following markers:

- `%d` — day of the month (*DD*), 01-31
- `%f` — seconds with fractional part (*SS.sss*), 00-59 plus decimal portion
- `%H` — hour (*HH*), 00-23
- `%j` — day of the year (*NNN*), 001-366
- `%J` — Julian day number (*DDDDDDD.dddddd*)
- `%m` — month (*MM*), 01-12
- `%M` — minute (*MM*), 00-59
- `%s` — seconds since 1970-01-01 (POSIX time value)
- `%S` — seconds (*SS*), 00-59
- `%w` — day of the week (*N*), 0-6, starting with Sunday as 0
- `%W` — week of the year (*WW*), 00-53
- `%Y` — year (*YYYY*)
- `%%` — a literal `%`

For example, the time format *HH*:*MM*:*SS.sss* (including fractional seconds) can be represented by the format string `'%H:%M:%f'`.

SQLite understands a number of input values. If the format of the time string is not recognized and cannot be decoded, `strftime()` will return NULL. All of the following input formats will be recognized:

- *YYYY-MM-DD*
- *YYYY-MM-DD HH:MM*
- *YYYY-MM-DD HH:MM:SS*
- *YYYY-MM-DD HH:MM:SS.sss*
- *YYYY-MM-DD*T*HH:MM*
- *YYYY-MM-DD*T*HH:MM:SS*

- *YYYY-MM-DDTHH:MM:SS.sss*
- *HH:MM*
- *HH:MM:SS*
- *HH:MM:SS.sss*
- now
- *DDDDDDD*
- *DDDDDDD.ddddddd*

In the case of the second, third, and fourth formats, there is a single literal space character between the date portion and the time portion. The fifth, six, and seventh formats have a literal T between the date and time portions. This format is specified by a number of ISO standards, including the standard format for XML timestamps. The last two formats are assumed to be a Julian Day or (with a modifier) a POSIX time value. These last two don't require a specific number of digits, and can be passed in as numeric values.

Internally, strftime() will always compute a full timestamp that contains both a date and time value. Any fields that are not specified by the input time string will assume default values. The default hour, minute, and second values are zero, or midnight, while the default date is 1 January 2000.

In addition to doing translations between formats and representations, the strftime() function can also be used to manipulate and modify time values before they are formatted and returned. Zero or more modifiers can be provided:

- [+-]*NNN* day[s]
- [+-]*NNN* hour[s]
- [+-]*NNN* minute[s]
- [+-]*NNN* second[s]
- [+-]*NNN.nnn* second[s]
- [+-]*NNN* month[s]
- [+-]*NNN* year[s]
- start of month
- start of year
- start of day
- weekday *N*
- unixepoch
- localtime
- utc

The first seven modifiers simply add or subtract the specified amount of time. This is done by translating the time and date into a segregated representation and then adding or subtracting the specified value. This can lead to invalid dates, however. For example, applying the modifier `'+1 month'` to the date `'2010-01-31'` would result in the date `'2010-02-31'`, which does not exist. To avoid this problem, after each modifier is applied, the date and time values are normalized back to legitimate dates. For example, the hypothetical date `'2010-02-31'` would become `'2010-03-03'`, since the unnormalized date was three days past the end of February.

The fact that the normalization is done after each modifier is applied means the order of the modifiers can be very important. Careful consideration should be given to how modifiers are applied, or you may encounter some unexpected results. For example, applying the modifier `'+1 month'` followed by `'-1 month'` to the date `'2010-01-31'`, will result in the date `'2010-02-03'`, which is three days off from the original value. This is because the first modifier gets normalized to `'2010-03-03'`, which is then moved back to `'2010-02-03'`. If the modifiers are applied in the opposite order, `'-1 month'` will convert our starting date to `'2009-12-31'`, and the `'+1 month'` modifier will then convert the date back to the original starting date of `'2010-01-31'`. In this instance we end up back at the original date, but that might not always be the case.

The three `start of...` modifiers shift the current date back in time to the specified point, while the `weekday` modifier will shift the date forward zero to six days, in order to find a date that falls on the specified day of the week. Acceptable `weekday` values are 0-6, with Sunday being 0.

The `unixepoch` modifier can only be used as an initial modifier to a numeric time value. In that case, the value is assumed to represent a POSIX time, rather than a Julian Day, and is translated appropriately. Although the `unixepoch` modifier must appear as the first modifier, additional modifiers can still be applied.

The last two modifiers are used to translate between UTC and local time representations. The modifier name describes the translation destination, so `localtime` assumes a UTC input and produces a local output. Conversely, the `utc` modifier assumes a local time input and produces a UTC output. SQLite is dependent on the local operating system (and its time zone and DST configuration) for these translations. As a result, these modifiers are subject to any errors and bugs that may be present in the time and date libraries of the host operating system.

Convenience functions

In an effort to help standardized text formats, avoid errors, and provide a more convenient way to covert dates and times into their most common representations, SQLite has a number convenience functions. Conceptually, these are wrapper functions around `strftime()` that output the date or time in a fixed format. All four of these functions take the same parameter set, which is essentially the same as the parameters used by `strftime()`, minus the initial format string.

`date( timestring, modifier, modifier... )`
: Translates the time string, applies any modifiers, and outputs the date in the format `YYYY-MM-DD`. Equivalent to the format string `'%Y-%m-%d'`.

`time( timestring, modifier, modifier... )`
: Translates the time string, applies any modifiers, and outputs the date in the format `HH:MM:SS`. Equivalent to the format string `'%H:%M:%S'`.

`datetime( timestring, modifier, modifier... )`
: Translates the time string, applies any modifiers, and outputs the date in the format `YYYY-MM-DD HH:MM:SS`. Equivalent to the format string `'%Y-%m-%d %H:%M:%S'`.

`julianday( timestring, modifier, modifier... )`
: Translates the time string, applies any modifiers, and outputs the Julian Day. Equivalent to the format string `'%J'`. This function differs slightly from the `strftime()` function, as `strftime()` will return a Julian Day as a text representation of a floating-point number, while this function will return an actual floating-point number.

All four of these functions recognize the same time string and modifier values that `strftime()` uses.

Time literals

SQLite recognizes three literal expressions. When an expression that contains one of these literals is evaluated, the literal will take on the appropriate text representation of the current date or time in UTC.

`CURRENT_TIME`
: Provides the current time in UTC. The format will be `HH:MM:SS` with an hour value between 00 and 23, inclusive. This is the same as the SQL expression `time( 'now' )`.

`CURRENT_DATE`
: Provides the current date in UTC. The format will be `YYYY-MM-DD`. This is the same as the SQL expression `date( 'now' )`.

`CURRENT_TIMESTAMP`
: Provides the current date and time in UTC. The format will be `YYYY-MM-DD HH:MM:SS`. There is a single space character between the date and time segments. This is the same as the SQL expression `datetime( 'now' )`. Note that the name of the SQL function is `datetime()`, while the literal is `_TIMESTAMP`.

Because these literals return the appropriate value in UTC, an expression such as `SELECT CURRENT_TIMESTAMP;` may not return the expected result. To get date and time in the local representation, you need to use an expression such as:

```
SELECT datetime( CURRENT_TIMESTAMP, 'localtime' );
```

In this case, the literal `CURRENT_TIMESTAMP` could also be replaced with `'now'`.

Examples

In some ways, the simplicity of the date and time functions can mask their power. The following examples demonstrate how to accomplish basic tasks.

Here is an example of how to take a local timestamp and store it as a UTC Julian value:

```
julianday( input_value, 'utc' )
```

This type of expression might appear in an `INSERT` statement. To insert the current time, this could be simplified to the `'now'` value, which is always given in UTC:

```
julianday( 'now' )
```

If you want to display a Julian value out of the database, you'll want to convert the UTC Julian value to the local time zone and format it. This can be done like this:

```
datetime( jul_date, 'localtime' )
```

It might also be appropriate to put an expression like this into a view.

If you wanted to present the date in a format more comfortable for readers from the United States, you might do something like this:

```
strftime( '%m/%d/%Y', '2010-01-31', 'localtime' );
```

This will display the date as `01/31/2010`. The second parameter could also be a Julian value, or any other recognized date format.

To get the current POSIX time value (which is always in UTC):

```
strftime( '%s', 'now' )
```

Or to display the local date and time, given a POSIX time value:

```
datetime( time_value, 'unixepoch', 'localtime' )
```

Don't forget that the input value is usually a simple text value. The value can be built up by concatenating together individual values, if required. For example, the following will calculate a Julian value from individual year, month, and day values that are bound to the statement parameters. Just be sure to bind them as text values with the proper number of leading zeros:

```
julianday( :year || '-' || :month || '-' || :day )
```

As you can see, once you understand how to combine different input formats with the correct modifiers, moving back and forth between time representations is fairly easy. This makes it much simpler to store date and time values using native representations that are otherwise unintelligible to most people.

ICU Internationalization Extension

SQLite provides full support for Unicode text values. Unicode provides a way to encode many different character representations, allowing a string of bytes to represent written characters, glyphs, and accents from a multitude of languages and writing systems.

What Unicode does *not* provide is any information or understanding of the sorting rules, capitalization rules, or equivalence rules and customs of a given language or location.

This is a problem for pattern matching, sorting, or anything that depends on comparing text values. For example, most text-sorting systems will ignore case differences between words. Some languages will also ignore certain accent marks, but often those rules depend on the specific accent mark and character. Occasionally, the rules and conventions used within a language change from location to location. By default, the only character system SQLite understands is 7-bit ASCII. Any character encoding of 128 or above will be treated as a binary value with no awareness of capitalization or equivalence conventions. While this is often sufficient for English, it is usually insufficient for other languages.

For more complete internationalization support, you'll need to build SQLite with the ICU extension enabled. The *International Components for Unicode* project is an open-source library that implements a vast number of language-related functions. These functions are customized for different locales. The SQLite ICU extension allows SQLite to utilize different aspects of the ICU library, allowing locale-aware sorts and comparisons, as well as locale-aware versions of `upper()` and `lower()`.

To use the ICU extension, you must first download and build the ICU library. The library source code, along with build instructions, can be downloaded from the project website at *http://www.icu-project.org/*. You must then build SQLite with the ICU extension enabled, and link it against the ICU library. To enable the ICU extension in an amalgamation build, define the `SQLITE_ENABLE_ICU` compiler directive.

You'll want to take a look at the original README document. It explains how to utilize the extension to create locale-specific collations and operators. You can find a copy of the README file in the full source distribution (in the `ext/icu` directory) or online at *http://www.sqlite.org/src/artifact?ci=trunk&filename=ext/icu/README.txt*.

The main disadvantage of the ICU library is size. In addition to the library itself, the locale information for all of the languages and locations adds up to a considerable bulk. This extra data may not be significant for a desktop system, but it may prove impractical on a handheld or embedded device.

Although the ICU extension can provide location-aware sorting and comparison capabilities, you still need to pick a specific locale to define those sorting and comparison rules. This is simple enough if you're only working with one language in one location, but it can be quite complex when languages are mixed. If you must deal with cross-locale sorts or other complex internationalization issues, it may be easier to pull that logic up into your application's code.

Full-Text Search Module

SQLite includes a Full-Text Search (FTS) engine. The current version is known as FTS3. The FTS3 engine is designed to catalog and index large bodies of text. Once indexed, the FTS3 engine can quickly search for documents based off various types of keyword searches. Although the FTS3 source is now maintained by the SQLite team, parts of the engine were originally contributed by members of Google's engineering team.

The FTS3 engine is a virtual table module. Virtual tables are similar to views, in that they wrap a data source to make it look and act like a normal table. Views get their data from a `SELECT` statement, while virtual tables depend on user-defined C functions. All of the functions required to implement a virtual table are wrapped up in an extension known as a module. For more information on how SQLite modules and virtual tables work, see Chapter 10.

Full-Text Search is an important and evolving technology, and is one of the areas that is targeted for improvements and enhancements as this book goes to press. Although this section gives a brief overview of the core FTS3 features, if you find yourself considering the FTS3 module, I would encourage you to review the full documentation on the SQLite website.

The FTS3 engine is included in all standard distributions of the SQLite source (including the amalgamation), but is turned off by default. To enable basic FTS functionality, define the `SQLITE_ENABLE_FTS3` compiler directive when building the SQLite library. To enable the more advanced matching syntax, also define `SQLITE_ENABLE_FTS3_PARENTHESIS`.

Creating and Populating FTS Tables

Once SQLite has been compiled with the FTS3 engine enabled, you can create a document table with an SQL statement similar to this:

```
CREATE VIRTUAL TABLE table_name USING FTS3 ( col1,... );
```

In addition to providing a table name, you can define zero or more column names. The name of the column is the only information that will actually be used. Any type information or column constraints will be ignored. If no column names are given, FTS will automatically create one column with the name `content`.

FTS tables are often used to hold whole documents, in which case they only need one column. Other times, they are used to hold different categories of related information, and require multiple columns. For example, if you wanted to store email messages in an FTS table, it might make sense to create separate columns for the `"SUBJECT:"` line, `"FROM:"` line, `"TO:"` line, and message body. This would allow you to limit searches to a specific column (and the data it contains). The column specification for an FTS table is largely determined by how you want to search for the data. FTS also provides an optimized way to look for a search term across all of the indexed columns.

You can use the standard `INSERT`, `UPDATE`, and `DELETE` statements to manipulate data within an FTS table. Like traditional tables, FTS tables have a `ROWID` column that contains a unique integer for each entry in the table. This column can also be referred to using the alias `DOCID`. Unlike traditional tables, the `ROWID` of an FTS table is stable through a vacuum (`VACUUM` in Appendix C), so it can be reliably referenced through a foreign key. Additionally, FTS tables have an internal column with the same name as the table name. This column is used for special operations. You cannot insert or update data in this column.

Any virtual table, including FTS tables, can be deleted with the standard `DROP TABLE` command.

Searching FTS Tables

FTS tables are designed so that any `SELECT` statement will work correctly. You can even search for specific text values or patterns directly with the `==` or `LIKE` operators. These will work, although they'll be somewhat slow, since standard operators will require a full table scan.

The real power of the FTS system comes from a custom `MATCH` operator. This operator is able to take advantage of the indexes built around the individual text values, allowing very fast searches over large bodies of text. Generally, searches are done using a query similar to:

```
SELECT * FROM fts_table WHERE fts_column MATCH search_term;
```

The search term used by the `MATCH` operator has very similar semantics to those used in a web search engine. Search terms are broken down and matched against words and terms found in the text values of the FTS table. Generally, the FTS `MATCH` operator is case-insensitive and will only match against whole words. For example, the search term `'data'` will match `'research data'`, but not `'database'`. The order of the search terms does not matter. The terms `'cat dog'` and `'dog cat'` will match the same set of rows.

By default, `MATCH` will only match records that contain every word or term found in the search term. If the extended syntax is enabled, more complex logic statements can also be used to define more complex search patterns.

Normally, the terms will only be matched against the specified column. However, every FTS table has a special hidden column that has the same name as the table itself. If this column is specified in the match expression, then all of the columns will be searched. This makes it easy to construct "find all" type searches.

More Details

The FTS module is fairly advanced, and offers a large number of search options and index optimizations. If you plan on using the FTS engine in your application, I strongly suggest you spend some time reading the online documentation (*http://www.sqlite.org/fts3.html*). The official documentation is quite extensive and covers the more advanced search features in some detail, complete with examples.

In addition to explaining the different search patterns, the online documentation also covers index optimization and maintenance. While this level of detail isn't always required, it can be beneficial for applications that heavily depend on FTS. The documents also explain how to provide a custom tokenizer to adapt the FTS engine to specific applications.

For those that want to dig even deeper, later sections of the document explain some of the internal workings of the FTS index. Information is provided on the shadow tables used by the FTS engine, as well as the process used to generate the token index. This level of knowledge isn't required to use the FTS system, but it is extremely useful if you are looking to modify the code, or if you're just curious about what is going on under the hood.

R*Trees and Spatial Indexing Module

The R*Tree module is a standard extension to SQLite that provides an index structure that is optimized for multi-dimensional ranged data. The R*Tree name refers to the internal algorithm used to organize and query the stored data. For example, in a two-dimensional R*Tree, the rows might contain rectangles, in the form of a minimum and maximum longitude value, along with a minimum and maximum latitude. Queries can be made to quickly find all of the rows that contain or overlap a specific geological location or area. Adding more dimensions, such as altitude, allows more complex and specific searches.

R*Trees are not limited to just spacial information, but can be used with any type of numeric range data that includes pairs of minimum and maximum values. For example, an R*Tree table might be used to index the start and stop times of events. The index could then quickly return all of the events that were active at a specific point or range of time.

The R*Tree implementation included with SQLite can index up to five dimensions of data (five sets of min/max pairs). Tables consist of an integer primary key column, followed by one to five pairs of floating-point columns. This will result in a table with an odd number of 3 to 11 columns. Data values must always be given in pairs. If you wish to store a point, simply use the same value for both the minimum and maximum component.

Generally, R*Trees act as detail tables for more traditional tables. A traditional table can store whatever data is required to define the object in question, including a key reference to the R*Tree data. The R*Tree table is used to hold just the dimensional data.

Multi-dimensional R*Trees, especially those used to store bounding rectangles or bounding volumes, are often approximations of the records they are indexing. In these cases, R*Trees are not always able to provide an exact result set, but are used to efficiently provide a first approximation. Essentially, the R*Tree is used as an initial filter to quickly and efficiently screen out all but a small percentage of the total rows. A more specific (and often more expensive) filter expression can be applied to get the final result set. In most cases, the query optimizer understands how to best utilize the R*Tree, so that it is applied before any other conditional expressions.

R*Trees are quite powerful, but they serve a very specific need. Because of this, we won't be spending the time to cover all the details. If an R*Tree index sounds like something your application can take advantage of, I encourage you to check out the online documentation (*http://www.sqlite.org/rtree.html*). This will provide a full description on how to create, populate, and utilize an R*Tree index.

Scripting Languages and Other Interfaces

Like most database products, SQLite has bindings that allow the functionality of the C APIs to be accessed from a number of different scripting languages and other environments. In many cases, these extensions try to follow a standardized database interface developed by the language community. In other cases, the driver or scripting extension attempts to faithfully represent the native SQLite API.

With the exception of the Tcl extension, all of these packages were developed by the SQLite user community. As such, they are *not* part of the core SQLite project, nor are they supported by the SQLite development team. If you're having issues installing or configuring one of these drivers, asking for support on the main SQLite user's mailing list may produce limited results. You may have more luck on a project mailing list or, failing that, a more general language-specific support forum.

As is common with this type of software, support and long-term maintenance can be somewhat hit-or-miss. At the time of publishing, most of the drivers listed here have a good history of keeping current and in sync with new releases of the SQLite library. However, before you build your whole project around a specific wrapper or extension, make sure the project is still active.

Perl

The preferred Perl module is *DBD::SQLite*, and is available on CPAN (*http://www.cpan.org*). This package provides a standardized, DBI-compliant interface to SQLite, as well as a number of custom functions that provide support for SQLite specific features.

The DBI provides a standard interface for SQL command processing. The custom functions provide some additional coverage of the SQLite API, and provide the ability to define SQL functions, aggregates, and collations using Perl. While the custom functions do not provide full coverage of the SQLite API, most of the more common operations are included.

PHP

As the PHP language has evolved, so have the SQLite access methods. PHP5 includes several different SQLite extensions that provide both vendor-specific interfaces, as well as drivers for the standardized PDO (PHP Data Objects) interface.

There are two vendor-specific extensions. The *sqlite* extension has been included and enabled by default since PHP 5.0, and provides support for the SQLite v2 library. The *sqlite3* extension has been included and enabled by default since PHP 5.3.0 and, as you might guess, provides an interface for the current SQLite 3 library. The *sqlite3* library provides a pretty basic class interface to the SQL command APIs. It also supports the creation of SQL functions and aggregates using PHP.

PHP 5.1 introduced the PDO interfaces. The PDO extension is the latest solution to the problem of providing unified database access mechanisms. PDO acts as a replacement for the PEAR-DB and MDB2 interfaces found in other versions of PHP. The *PDO_SQLITE* extension provides a PDO driver for the current SQLite v3 library. In addition to supporting the standard PDO access methods, this driver also provides custom methods to create SQL functions and aggregates using PHP.

Given that there is very little functional difference between the SQLite 3 vendor-specific library and the PDO SQLite 3 library, I suggest that new development utilize the PDO driver.

Python

There are two popular Python interfaces available. Each wrapper addresses a different set of needs and requirements. At the time of this writing, both modules were under active development.

The *PySQLite* module (*http://code.google.com/p/pysqlite/*) offers a standardized Python DB-API 2.0 compliant interface to the SQLite engine. PySQLite allows applications to develop against a relatively database-independent interface. This is very useful for systems that need to support more than one database. Using a standardized interface also allows rapid prototyping with SQLite, while leaving a migration path to larger, more complex database systems. As of Python 2.5, PySQLite has become part of the Python Standard Library.

The *APSW* module (*Another Python SQLite Wrapper*; *http://code.google.com/p/apsw/*) has a very different design goal. The APSW provides a very minimal abstraction layer

that is designed to mimic the native SQLite C API as much as possible. APSW makes no attempt to provide compatibility with any other database product, but provides very broad coverage of the SQLite library, including many of the low-level features. This allows very fine-grain control, including the ability to create user-defined SQL functions, aggregates, and collations in Python. APSW can even be used to write a virtual table implementation in Python.

Both modules have their strong points. Which module is right for your application depends on your needs. If your database needs are fairly straightforward and you want a standardized interface that allows future migration, then PySQLite is a better fit. If you don't care about other database engines, but need very detailed control over SQLite, then APSW is worth a look.

Java

There are a number of interfaces available to the Java language. Some of these are wrappers around the native C API, while others conform to the standardized Java Database Compatibility (JDBC) API.

One of the older wrappers is *Java SQLite* (*http://www.ch-werner.de/javasqlite/*), which provides support for both SQLite 2 and SQLite 3. The core of this library uses Java Native Interface (JNI) to produce an interface based off the native C interface. The library also contains a JDBC interface. It is a good choice if you need direct access to the SQLite API.

A more modern JDBC-only driver is the *SQLiteJDBC* package (*http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC*). This is a rather nice distribution, as the JAR file contains both the Java classes, as well as native SQLite libraries for Windows, Mac OS X, and Intel-based Linux. This makes cross-platform distribution quite easy. The driver is also heavily utilized by Xerial, so it tends to be well maintained.

Tcl

SQLite has a strong history with the Tcl language. In fact, what we now know as SQLite started life as a Tcl extension. Much of the testing and development tools for SQLite are written in Tcl. In addition to the native C API, the Tcl extension is the only API supported by the core SQLite team.

To enable the Tcl bindings, download the TEA (Tcl Extension Architecture) distribution of the SQLite source from the SQLite website (*http://www.sqlite.org/download.html*). This version of the code is essentially the amalgamation distribution with the Tcl bindings appended to the end. This will build into a Tcl extension that can then be imported into any Tcl environment. Documentation on the Tcl interface can be found at *http://www.sqlite.org/tclsqlite.html*.

ODBC

The ODBC (Open Database Connectivity) specification provides a standardized database API for a wide variety of database products. Like many of the language-specific extensions, ODBC drivers act as a bridge between an ODBC library and a specific database API. Using ODBC allows developers to write to a single API, which can then use any number of connectors to speak to a wide variety of database products. Many generic database tools utilize ODBC to support a broad range of database systems.

The best known connector for SQLite is *SQLiteODBC* (*http://www.ch-werner.de/sqliteodbc/*). SQLiteODBC is tested with a number of ODBC libraries, ensuring it should work with most tools and applications that utilize ODBC support.

.NET

There are a number of independent SQLite projects that use .NET technologies. Some of these are simple C# wrappers that do little more than provide an object context for the SQLite API. Other projects attempt to integrate SQLite into larger frameworks, such as ADO (ActiveX Data Objects).

One of the more established open source projects is the *System.Data.SQLite* (*http://sqlite.phxsoftware.com/*) package. This package provides broad ADO support, as well as LINQ support.

There are also commercial ADO and LINQ drivers available. See the SQLite wiki for more information.

C++

Although the SQLite C API can be accessed directly by C++ applications, some people prefer a more object-oriented interface. If you would prefer to use an existing library, there are a number of wrappers available. You can check the SQLite website or do some web searches if you're interested.

Be warned that few of these libraries are well maintained. You might be better off just writing and maintaining your own wrapper classes. The SQLite API has a rather object-influenced design, with most functions performing some type of manipulation or action on a specific SQLite data structure. As a result, most C++ wrappers are somewhat thin, providing little more than syntactical translation. Maintaining a private wrapper is normally not a significant burden.

Just remember that the core SQLite library is C, not C++, and cannot be compiled with most C++ compilers. Even if you choose to wrap the SQLite API in a C++ class-based interface, you'll still need to compile the core SQLite library with a C compiler.

Other Languages

In addition to those languages listed there, there are wrappers, libraries, and extensions for a great number of other languages and environments. The wiki section of the SQLite website has an extensive list of third-party drivers at *http://www.sqlite.org/cvstrac/wiki?p=SqliteWrappers*. Many of the listed drivers are no longer actively maintained, so be sure to research the project websites before investing in a particular driver. Those that are known to be abandoned are marked as such, but it is difficult to keep this kind of information up to date.

Mobile and Embedded Development

As the power and capacity of smartphones, mobile devices, and other embedded systems continue to increase, these devices are able to deal with larger and more complex data. Many mobile devices are centered around organizing, searching, and displaying large quantities of structured data. This might be something as simple as an address book, or something much more complex, like mapping and route applications.

In many cases, application requirements for data storage and management fit very well with the relational model provided by SQLite. SQLite is a fairly small and very resource-aware product, making it run well in restricted environments. The database-in-a-file model also makes it easy to copy or back up datastores easily and quickly. Given all these factors, it should come as no surprise that almost every major smartphone SDK supports SQLite out of the box, or allows it to be easily compiled for their platform.

Memory

Most mobile devices have limited memory resources. Applications must be conscious of their memory usage, and often need to limit the resources that may be consumed. In most cases, the majority of SQLite memory usage comes from the page cache. By picking a sensible page size and cache size, the majority of memory use can be controlled. Remember that each open or attached database normally has its own, independent cache. The page size can be adjusted at database creation with the `PRAGMA page_size` command, while the cache size can be adjusted at any time with `PRAGMA cache_size`. See `page_size` and `cache_size` in Appendix F for more details.

Be aware that larger cache sizes are not always significantly better. Because some types of flash storage systems have no effective seek time, it is sometimes possible to utilize a relatively small page cache while still maintaining acceptable performance. The faster response time of the storage system reduces the cost of pulling pages into memory, lessening the impact of the cache. Just how fast the storage system can respond has a great deal to do with types of flash chips and how they are configured in the device, but depending on your system, you may find it acceptable to use a relatively small cache. This should help keep your memory footprint under control.

If you're working in an extremely constrained environment, you can preallocate buffers and make them available to SQLite through the `sqlite3_config()` interface. Different buffers can be assigned for the SQLite heap, scratch buffers, and page cache. If buffers are configured before the SQLite library is initialized, all memory allocations will come from these buffers. This allows a host application precise control over the memory resources SQLite may use.

Most other issues are fairly self-evident. For example, the use of in-memory databases is generally discouraged on memory-bound devices, unless the database is very small and the performance gains are significant. Similarly, be aware of queries that can generate large intermediate result sets, such as `ORDER BY` clauses. In some cases it may make more sense to pull some of the processing or business logic out of the database and into your application, where you have better control over resource utilization.

Storage

Nearly all mobile devices use some type of solid-state storage media. The storage may be on-board, or it may be an expansion card, such as an SD (Secure Digital) card, or even an external thumb drive. Although these storage systems provide the same basic functionality as their "real computer" counterparts, these storage devices often have noticeably different operating characteristics from traditional mass-store devices.

If possible, try to match the SQLite page size to the native block size of the storage system. Matching the block sizes will allow the system to write database pages more quickly and more efficiently. You want the database page size to be the same size as one or more whole filesystem blocks. Pages that use partial blocks will be much slower. You don't want to make the page too large, however, or you'll be limiting your cache performance. Finding the right balance can take some experimentation.

Normally, SQLite depends heavily on filesystem locking to provide proper concurrency support. Unfortunately, this functionality can be limited on mobile and embedded platforms. To avoid problems, it is best to forego multiple database connections to the same database file, even from the same application. If multiple connections are required, make sure the operating system is providing proper locking, or use an alternate locking system. Also consider configuring database connections to acquire and hold any locks (use the `PRAGMA locking_mode` command; see `locking_mode` in Appendix F). While this makes access exclusive to one connection, it increases performance while still providing protection.

It may be tempting to turn off SQLite's synchronization and journaling mechanism, but you should consider any possible consequences of disabling these procedures. While there are often significant performance gains to be found in disabling synchronizations and journal files, there is also the significant danger of unrecoverable data corruption.

For starters, mobile devices run off batteries. As we all know, batteries have a tendency to go dead at very annoying times. Even if the operating system provides low-power warnings and automatic sleep modes, on many models it is all too easy to instantly dislodge the battery if the device is dropped or mishandled. Additionally, many devices utilize removable storage, which has a tendency to be removed and disappear at inconvenient times. In all cases, the main defense against a corrupt database is the storage synchronization and journaling procedure.

Storage failures and database corruption can be particularly devastating in a mobile or embedded environment. Because mobile platforms tend to be more closed to the user, it is often difficult for the end-user to back up and recover data from individual applications, even if they are disciplined enough to regularly back up their data. Finally, data entry is often slow and cumbersome on mobile devices, making the prospect of manual recovery a long and undesirable prospect. Mobile applications should be as forgiving and error tolerant as possible. In many cases, losing a customer's data will result in losing a customer.

Other Resources

Beyond special memory handlers and storage considerations, most other concerns boil down to being aware of the limitations of your platform and keeping resource use under control. If you're preparing a custom SQLite build for your application, you should take some time to run through all the available compiler directives and see if there are any defaults you want to alter or any features you might want to disable (see Appendix A). Disabling unused features can reduce the code and memory footprints even further. Disabling some features also provides minor performance increases.

It is also a good idea to read through the available `PRAGMA` statements and see if there are any further configuration options to customize SQLite behavior for your specific environment. `PRAGMA` commands can also be used to dynamically adjust resource use. For example, it might be possible to temporarily boost the cache size for an I/O intensive operation if it is done at a time when you know more memory is available. The cache size could then be reduced, allowing the memory to be used elsewhere in the application.

iPhone Support

When the iPhone and iPod touch were first released, Apple heavily advocated the use of SQLite. The SQLite library was provided as a system framework and was well documented, complete with code examples, in the SDK.

With the release of version 3.0, Apple has made their Core Data system available on the iPhone OS. Core Data has been available on the Macintosh platform for a number of years, and provides a high-level data abstraction framework that offers integrated

design tools and runtime support to address complex data management needs. Unlike SQLite, the Core Data model is not strictly relational in nature.

Now that the higher level library is available on their mobile platform, Apple is encouraging people to migrate to Core Data. Most of the SQLite documentation and code examples have been removed from the SDK, and the system-level framework is no longer available. However, because Core Data utilizes SQLite in its storage layer, there is still a standard SQLite system library available for use. It is also relatively easy to compile application-specific versions of the SQLite library. This is required if you want to take advantage of some of the more recent features, as the system version of SQLite is often several versions behind.

Core Data has some significant advantages. Apple provides development tools that allow a developer to quickly lay out their data requirements and relationships. This can reduce development time and save on code. The Core Data package is also well integrated into current Mac OS X systems, allowing data to move back and forth between the platforms quite easily.

For all the advantages that Core Data provides, there are still situations where it makes sense to use SQLite directly. The most obvious consideration is if your development needs extend beyond Apple platforms. Unlike Core Data, the SQLite library is available on nearly any platform, allowing data files to be moved and accessed almost anywhere on any platform. Core Data also uses a different storage and retrieval model than SQLite. If your application is particularly well suited to the Relational Model, there may be advantages to having direct SQL query access to the data storage layer. Using the SQLite library directly also eliminates a number of abstraction layers from the application design. While this may lead to more detailed code, it is also likely to result in better performance, especially with larger datasets.

Like many engineering choices, there are benefits and concerns with both approaches. Assuming the platform limitations aren't a concern, Core Data can provide a very rapid solution for moderately simple systems. On the other hand, SQLite allows better cross-platform compatibility (in both code and data), and allows direct manipulation of complex data models. If you're not dependent on the latest version of SQLite, you may even be able to reduce the size of your application by using the existing SQLite system library. Which set of factors has higher value to you is likely to be dependent on your platform requirements, and the complexity of your data model.

Other Environments

A number of smartphone environments require application development to be done in Java or some similar language. These systems often provide no C compilers and limit the ability to deploy anything but byte-code. While most of these platforms provide custom wrappers to system SQLite libraries, these wrappers are often somewhat limited. Typically, the system libraries are several versions behind, and the Java wrappers are often limited to the essential core function calls.

While this may limit the ability to customize the SQLite build and use advanced features of the SQLite product, these libraries still provide full access to the SQL layer and all the functionality that comes with it, including constraints, triggers, and transactions. To get around some limitations (like the lack of user-defined functions), it may sometimes be necessary to pull some of the business logic up into the application. This is best done by designing an access layer into the application that centralizes all of the database functions. Centralizing allows the application code to consistently enforce any database design constraints, even when the database is unable to fully do so. It is also a good idea to include some type of verification function that can scan a database, identifying (and hopefully correcting) any problems.

Additional Extensions

In addition to those interfaces and modules covered here, there are numerous other extensions and third-party packages available for SQLite. Some are simple but useful extensions, such as a complete set of mathematical functions. The best way to find these are through web searches, or by asking on the SQLite User's mailing list. You can also start by having a look at *http://sqlite.org/contrib/* for a list of some of the older code contributions.

In addition to database extensions, there are also several SQLite-specific tools and database managers available. In addition to the command-line shell, several GUI interfaces exist. One of the more popular is SQLite Manager, a Firefox extension available at *http://code.google.com/p/sqlite-manager/*.

A small number of commercial extensions for SQLite also exist. As already discussed, some of the database drivers require a commercial license. Hwaci, Inc., (the company responsible for developing SQLite) offers two commercial extensions. The *SQLite Encryption Extension* (SEE) encrypts database pages as they are written to disk, effectively encrypting any database. The *Compressed and Encrypted Read-Only Database* (CEROD) extension goes further by compressing database pages. The compression reduces the size of the database file, but also makes the database read-only. This extension can be useful for distributing licensed data archives or reference materials. For more information on these extensions, see *http://www.sqlite.org/support.html*.