QuickLite v1.5

Purpose of this document

QuickLite 1.5 incorporates many new features and improvements. The purpose of this document is to introduce the developer to these additions and changes, as well as explain the reasoning behind some of the new implementations.

Datatype support

SQLite is inherently type-less. Any type of data can be stored in the database, strings, sounds, numbers, BLOBs... anything. This brings the following problem: what is stored in the database and how can we interpret it? In v1.0, we just can't.

New features such as QuickEdit and the new caching method requires datatype support because it's essential to instantiate the proper object for the data stored in the database. These are the new datatypes supported in QuickLite 1.5:

Datatype	Cocoa class	Purpose
QL_String	NSString	String of characters
QL_Number	NSNumber	Integer and floating point
QL_DateTime	NSCalendarDate	Date and time with timezone support
QL_Boolean	NSNumber	Boolean values
QL_Container	NSData	Any type of data, a BLOB

QuickLite will store the data transparently, based on its declared datatype. In the case of BLOBs, for example, the NSData object will be encoded using Base64 and stored in the database. Likewise, a BLOB will be decoded back properly and converted to NSData prior to be returned.

Better data caching

Typically, the developer must choose between maximum performance and lower memory footprint. To get the best performance, data should be stored in RAM. This can lead to a large data set being pushed to RAM. In most systems today, it may not be an issue, but developers may still want to keep the memory footprint leaner. Starting in v1.5, three new caching methods are supported:

Cache Method	What it does	Performance	Memory Footprint
CacheAllData	Caches all selected data	Best	Large
CacheDataOnDemand	Caches data as it is requested	Better	May grow large
DoNotCacheData	Fetches data from disk each time	Good	Best

It's important to understand that even though CacheDataOnDemand starts with an empty cache, it can grow quickly if the app requests lots of data for most of its columns. If memory is definitely a concern, DoNotCache is the best method because it'll fetch data from disk each time. Obviously, performance will not be the best but since SQLite fetches data very quickly, the negative on performance should not be too bad. Your mileage may vary depending on the application being written.

QuickEdit: edit and delete data the OOP way

Data is retrieved from the database performing a query and obtaining a QuickLiteCursor. To access the data, a QuickLiteRow must be obtained first through an index. Once we have the row object, the data can be accessed via its accessors.

In v1.0, the concept of Namespaces were introduced. Namespaces has brought serious limitations in v1.5 while implementing the new caching scheme as well as QuickEdit. For these reasons alone, the concept of Namespace has been removed. Starting in v1.5, the API expects columns to be declared in the following form: table.column (i.e. address.ZIP). The only exceptions are those methods which require a 'table' parameter. A few examples follow:

```
- (BOOL)createTable:(NSString*)table withColumns:(NSArray*)columns andDatatypes:(NSArray*)datatypes;
- (BOOL)addColumn:(NSString*)columnName withDataType:(NSString*)dataType toTable:(NSString*)table;
- (BOOL)insertValues:(NSArray*)values forColumns:(NSArray*)columns inTable:(NSString*)table;
```

In the above examples, columns are declared without prefixing the table name to the column name.

In short, QuickEdit allows cursors to be edited directly without dealing with SQL statements at all. Typically, cursors are static, that is, its data cannot be modified and new data cannot be appended in any way. QuickEdit takes cursors to a new level, allowing data to be modified, added or deleted:

```
- (BOOL)setValue:(id)value forColumn:(NSString*)tableAndColumn;
- (BOOL)insertRowWithValues:(NSArray*)values andColumns:(NSArray*)columns;
- (BOOL)removeAllRowUIDsAtIndex:(unsigned long)rowIndex;
```

When any combination of these actions take place, the cursor registers with the QuickLiteDatabase object. This registration allows the database object to communicate to the cursor when the database is about to be closed. The cursor then proceeds to commit the changes to the database. Of course, the developer is in full control, so uncommitted changes made to a cursor can safely be discarded via [QuickLiteCursor revert] or [Database revert]. Data can be committed to disk at once via [QuickLiteDatabase save] or on a cursor-per-cursor basis via [QuickLiteCursor save]. After either a save or revert has taken place, the cursors de-register automatically from the database object.

When new data is added to a cursor, it is appended at the end of its data set. If you need to know the range of rows that comprise these newly added rows, you can use the following method found in QuickLiteCursor:

```
- (NSRange)rangeOfInsertedRows;
```

If no new rows have been added, this method returns a range of {NSNotFound, 0}.

Changes can be applied to an entire column without having to traverse the cursor. In QuickLiteCursor, the following method does that:

```
- (BOOL)setValue:(id)value forColumn:(NSString*)tableAndColumn;
```

When setting data either on a row-per-row basis or to an entire column, the value datatype must match the column datatype. For instance, setting a string to a column declared as anything other than a QLString will produce an error and the change will not be honored. Another point to be aware of: QuickEdit requires that all columns specified when adding/modifying/deleting data must exist in the cursor. New columns cannot be added on-the-fly and only columns included as a result of the SELECT statement are allowed.

Special care must be taken when deleting rows. Row data can contain one or more rowUIDs collected from a SELECT query. For example, the following columns could be specified:

```
person.ROWID | person.first | order.ROWID | order.total
1673 | Joe | 984 | 231.45
```

You can delete a row via 'removeAllRowUIDsAtIndex', in QuickLiteCursor:

```
- (BOOL)removeAllRowUIDsAtIndex:(unsigned long)rowIndex;
```

QuickEdit will annotate all references to ROWID, no matter what table. In the case specified above, it will mark two records for deletion: ROWID 1673 in table 'person' and ROWID 984 in table 'order'. These marked deletions can be reclaimed via [QuickLiteCursor revert], but will go into effect once [QuickLiteDatabase save] or [QuickLiteCursor save] is called.

If this is not what you want to do, you can obtain the modified rowUIDs with:

```
- (NSDictionary*)modifiedRowUIDs;
```

The XML representation of the dictionary would be something like this:

```
<dict>
	<key>people</key>
	<data>
		<array>
			<data>3</data>
			<data>12</data>
			<data>214</data>
		</array>
	</data>
	<key>order</key>
	<data>
		<array>
			<data>121</data>
		</array>
	</data>
</dict>
```

This way you can delete the rowUIDs you desire at your own discretion with the following methods found in QuickLiteDatabase:

```
- (BOOL)deleteRowWithUID:(NSString*)rowUID inTable:(NSString*)table;
- (BOOL)deleteRowsWithUIDObjects:(NSArray*)rowUIDs inTable:(NSString*)table;
```

If you want to remove all entries in a specific table you can use the following method:

```
- (BOOL)deleteAllRowsInTable:(NSString*)table;
```

QuickEdit is completely optional. Since it is an extension of QuickLite, it can be ignored completely if you're not interested.

Notify data changes via notifications

Notifications are essential to keep QuickLiteCursors in sync with the database. Every QuickEdit operation is kept in memory until [QuickLiteDatabase save] or [QuickLiteCursor save] is called. When this happens, QuickLite issues an NSDistributedNotification, QLP_QuickLiteDatabaseDidChangeNotification, with a userInfo dictionary containing the following information:

```
QLP_ProcessID: NSNumber containing the process ID that generated the notification.
QLDatabasePathIdentifier: NSString containing the path of the database involved.
QLTimestampIdentifier: NSCalendarDate containing the timestamp the when SQL statement was executed.
```

This notification is also broadcasted to other currently logged-in users on Mac OS X 10.3 or later.

When you receive a QLP_QuickLiteDatabaseDidChangeNotification, you may want to update the current cursors by calling [QuickLiteCursor refresh]. This action executes the very same SQL statement that created the cursor in the first place, and while the static data is refreshed, it *does not* clear the other cached data (edits, inserts and/or deletes).

Debugging Tools

Two options allows the developer to obtain the SQL statements invoked and to trace the QuickLite methods called. These two options are found in [QuickLiteDatabase open: cacheMethod: exposeSQLOnNotify: debugMode:]

Setting 'exposeSQLOnNotify' sends a distributed notification, QLDatabaseHasSentSQLStatementNotification, each time a SQL statement is executed. The dictionary sent along the notification contains the following info:

QLP_ProcessID: NSNumber containing the process ID that generated the notification.
QLDatabasePathIdentifier: NSString containing the path of the database involved.
QLTimestampIdentifier: NSCalendarDate containing the timestamp the when SQL statement was executed.
QLSQLStatementIdentifier: NSString containing the SQL statement.

Be aware that turning this option on slows down QuickLite by ~50%, so make sure that it's turned off in your deployed version. For more info, refer to the QuickLiteListener example, found in the Examples folder.

If you want to trace which QuickLite methods are being called, turn on the 'debugMode' option. An example of the log output follows:

```
2004-08-21 14:19:07.695 SuperTableView[4610] [QuickLiteDatabase] beginTransaction
2004-08-21 14:19:07.698 SuperTableView[4610] [QuickLiteDatabase] dropTable:
2004-08-21 14:19:07.699 SuperTableView[4610] [QuickLiteDatabase] datatypesForTable:
2004-08-21 14:19:07.700 SuperTableView[4610] [QuickLiteCursor] valueForColumn:andRow:
2004-08-21 14:19:07.700 SuperTableView[4610] [QuickLiteDatabase] dropTable:
2004-08-21 14:19:07.700 SuperTableView[4610] [QuickLiteDatabase] createTable:withColumns:andDatatypes:
2004-08-21 14:19:07.702 SuperTableView[4610] [QuickLiteDatabase] datatypesForTable:
2004-08-21 14:19:07.702 SuperTableView[4610] [QuickLiteCursor] valueForColumn:andRow:
2004-08-21 14:19:07.703 SuperTableView[4610] [QuickLiteDatabase] datatypesForTable:
2004-08-21 14:19:07.703 SuperTableView[4610] [QuickLiteCursor] valueForColumn:andRow:
2004-08-21 14:19:07.722 SuperTableView[4610] [QuickLiteDatabase] insertValues:forColumns:inTable:
2004-08-21 14:19:07.722 SuperTableView[4610] [QuickLiteDatabase] commitTransaction
2004-08-21 14:19:07.766 SuperTableView[4610] [QuickLiteDatabase] compact
2004-08-21 14:19:07.792 SuperTableView[4610] [QuickLiteDatabase] performQuery:
```

These options are useful to know what goes on 'behind the scenes', and should be turned off if you want to obtain the optimal performance.

Summary of features

- Updated with SQLite 3.0.7
- Data integrity check provided by SQLite
- Datatype support includes QLString, QLNumber, QLBoolean, QLDateTime, and QLContainer
- Better data caching: CacheAllData, CacheOnDemand, or DoNotCacheData
- QuickEdit: add, edit, and remove rows without a single SQL statement, the OOP-way.
- Save and revert supported, database-wide and on a cursor-by-cursor basis
- Distributed notifications, Fast User Switching-aware, sent when a commit action takes place
- Debugging facilities to observe all SQL statements executed, as well as tracing QuickLite methods.
- To know about the latest changes, please read the Release Notes at: http://www.webbotech.com

- End -