

SQLite instead of Core Data: A Lightweight Alternative

Chris Cieslak

Chicago Cocoaheads, March 9, 2010

“SQLite doesn’t
compete with Oracle.
It competes with
fopen().”

-D. Richard Hipp,
Creator of SQLite

SQLite

- A large subset of SQL commands are supported
- No foreign keys (at least on the versions on Mac/iPhone)
- No client-server structure; it hits a local file directly
- C API that can be wrapped in Obj-C easily
- Extensible via C functions

Core Data is great, but...

- It's not a relational database
- It has to load objects into memory to do anything to them
- You have to get the data into Core Data in the first place
- Sometimes it's just too much for the data at hand

Core Data: Not a RDBMS

- Core Data manages the connections between objects, but leaves most data constraints to the programmer
- Primary keys and indexes are abstracted away by Core Data- any unique key validation must be done in code using [managedObject objectID] or UUIDs
- On the other hand, Core Data is much more robust at connections than SQLite

Core Data and Memory

- SQLite: “DROP TABLE `my_table`: one operation, no rows loaded into memory
- Core Data: [context deleteObject:myObject]: each object in memory deleted; changes written when [context save:] called
- If doing changes to a large number of objects, have to be mindful of memory management on Core Data

Importing into Core Data

- if using Core Data on iPhone, most likely you will be creating the base store on the Mac
- Not difficult, but more work
- But what if you already have an existing SQL database?
- `sqlite3 mysqlite.db < mydata.sql...` and done

Object Creation

- For smaller data sets, you might not want to create many `NSManagedObjects`
- Easier to pass around `NSDictionary`s or `NSString`s

Speed & Optimization

- Core Data engineers are definitely smarter than I am
- You don't have to worry about backend store optimization in Core Data- a lot of heavy lifting is done in memory as opposed to disk
- However, if you are going to use SQLite, there are ways to optimize performance

Basic Optimizations

- SQLite is pretty good at optimizing your query strings
- If doing multiple tests in a query, move the query which returns the fewest rows to the left
- You can use “.explain” in the sqlite3 command line tool to see how SQLite will handle a particular query in the VM

```
sqlite> EXPLAIN select * from `bus_stops`,`bus_patterns`
where `bus_stops`.`stopID` = `bus_patterns`.`stopID`;
```

addr	opcode	p1	p2	p3	p4	p5	comment
0	Trace	0	0	0		00	
1	Goto	0	20	0		00	
2	OpenRead	1	9	0	5	00	
3	OpenRead	0	4	0	2	00	
4	Rewind	1	17	0		00	
5	Column	1	1	1		00	
6	MustBeInt	1	16	0		00	
7	NotExists	0	16	1		00	
8	Rowid	0	2	0		00	
9	Column	0	1	3		00	
10	Column	1	0	4		00	
11	Column	1	1	5		00	
12	Column	1	2	6		00	
13	Column	1	3	7		00	
14	Column	1	4	8		00	
15	ResultRow	2	7	0		00	
16	Next	1	5	0		01	
17	Close	1	0	0		00	
18	Close	0	0	0		00	
19	Halt	0	0	0		00	
20	Transaction	0	0	0		00	
21	VerifyCookie	0	69	0		00	
22	TableLock	0	9	0	bus_patterns	00	
23	TableLock	0	4	0	bus_stops	00	
24	Goto	0	2	0		00	

Profiling

- Use simple profiling techniques to see how your query runs on the hardware

```
NSDate *queryStartTime = [NSDate date];  
  
// Code that hits the DB goes here...  
  
NSTimeInterval elapsed =  
[queryStartTime timeIntervalSinceNow];  
NSLog(@"DB operation took %f seconds",  
-elapsed);
```

CREATE INDEX

- CREATE INDEX allows you to specify an index column in an SQLite table
- You really only want to do this on integer data
- Doesn't work if you're using LIKE matching
- Can improve some searches while delaying others
- Remember, your primary key or rowid is already an index

Wrapping SQLite in Obj-C

- There are a number of Objective-C wrappers that “Cocoa-ify” the native SQLite C API
- These are not object persistence frameworks, just methods to execute queries and enumerate through result rows
- Can help with threading issues

“Threads are evil.
Avoid them.”

-SQLite FAQ

FMDB

```
FMDatabase *db = [[FMDatabase alloc]
initWithPath:@"~/Users/cjc/my.db"];
[db open];
BOOL success = [db executeUpdate:@"CREATE TABLE
test (a text, b integer, c double);"];
FMResultSet *result = [db executeQuery:@"select
* from table1 where a = ?", @"test"];
while ([rs next]) {
    NSLog(@"Column b:%@", [rs
stringForColumn:@"b"]);
}
[rs close];
[db close];
[db release];
```

FMDB

- Simple and lightweight
- Does not handle threading issues; if 2 different methods attempt to query or update the database simultaneously, the query simply fails
- Must leave queries open while iterating
- Must explicitly close result sets and databases

EGODatabase

```
EGODatabase *db = [[EGODatabase alloc]
initWithPath:@"~/Users/cjc/my.db"];
BOOL success = [db executeUpdate:@"CREATE TABLE
test (a text, b integer, c double);"];
EGODatabaseResult *result = [db executeQuery:
@"select * from table1 where a = ?" parameters:
[NSArray arrayWithObject:@"test"]];
for (EGODatabaseRow* row in result) {
    NSLog(@"Column b:%@", [row
stringForColumn:@"b"]);
}
[db release];
```

EGODatabase

- Uses NSLock for thread-safety; will delay calls to the SQLite database until the current call is finished
- Uses fast enumeration
- Populates its EGODatabaseRow class with SQLite data so it can close the query quickly

Creating SQLite Functions

- You can create or redefine your own function or aggregate

```
int err = sqlite3_create_function(handle,  
"multByTwo", 1, SQLITE_UTF8, NULL, &multFunc, NULL,  
NULL);
```

```
static void multFunc(sqlite3_context *context, int  
argc, sqlite3_value **argv) {  
    double result = sqlite_value_double(argv[0]);  
    sqlite3_result_double(context, result * 2);  
}
```

Creating Functions

- Instead of returning a large amount of data and iterating through it, have SQLite optimize the order in which the subqueries are made

BNR Persistence

- Unlike FMDB/EGODB, persists objects
- Uses key-value store instead of SQLite
- Closer to Core Data (BNRStore = NSManagedObjectContext, BNRStoredObject = NSManagedObject)
- Can be 10-20 times faster than Core Data

```
[db close];
```