Volume 1, Issue 1

Fall, 2014

Golden Bits Tech Newsletter

Golden Bits Software ®

iOS Core Data vs. SQLite Comparing Storage Technologies

This article has been published in Dr. Dobbs see:
http://www.drdobbs.com/mobile/ios-data-storage-core-data-vs-sqlite/240168843

Highlights:

- Sample code can be downloaded from the Golden Bits Web site. Use this link:

http://www.docs-goldenbits.com/newsletters/coredata-sqlite/DataStoreTester.zip

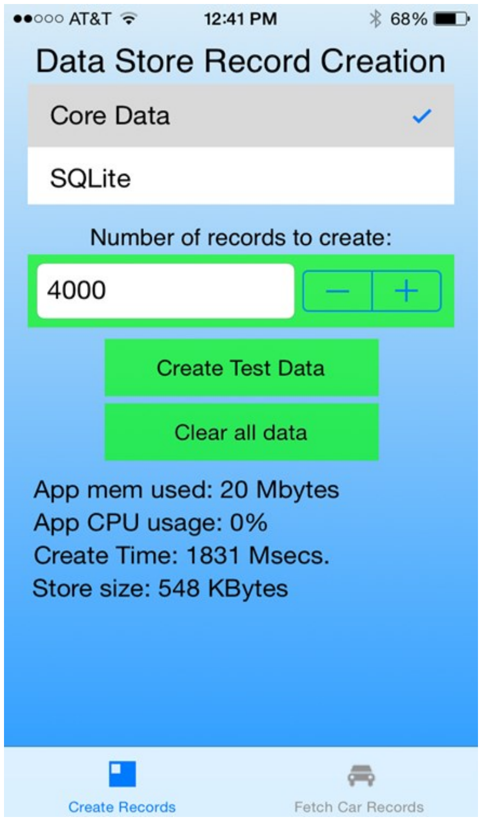
Golden Bits Software, Inc.
3525 Del Mar Heights Road, Suite 158
San Diego, CA 92130
858-259-3870 phone
858-259-7655 fax

Inside:

Introduction

When developing an iOS application, chances are you will need to store some type of data. For iOS the two main storage technologies available to developers are Core Data and SQLite. Both technologies have the advantages and disadvantages depending on the amount and type of data you need to store and manage. This newsletter and accompanying sample application provides an overview of Core Data and SQLite, compares the two technologies, and provides a sample code. The sample application is an iPhone application that switches between Core Data and SQLite, thus enabling a direct comparison between these two technologies. Some simple metrics are displayed, memory usage, application CPU usage, and data store size.

A screen shot of the application is show to the right.



Data Store Record Creation

Core Data ✓

SQLite

Number of records to create: 4000

Create Test Data

Clear all data

App mem used: 20 Mbytes
App CPU usage: 0%
Create Time: 1831 Msecs.
Store size: 548 KBytes

Screen shot of the sample iPhone app

Golden Bits Software ® is a software engineering firm providing consulting services in a wide range of diverse technologies.

Contact information:

Email: deang@goldenbits.com
Web: goldenbits.com

Core Data vs. SQLite

Swift

Apple released Swift at WWDC this past June 2nd. Swift is a new programming language that can be used to develop Mac and iOS applications.

Importantly Swift provides interoperability with existing frameworks — including Core Data. This means it is possible to use Core Data and SQLite in you next Swift project.

Sample Data Set

The best way to compare Core Data and SQLite is to use the same data set when testing. While this is not a perfect approach, since a single data set cannot possibly illustrate all of the capabilities of Core Data and SQLite, for practical purposes it is a reasonable approach. Our sample data set consist of a list of cars, car manufacturers, car type, and car details. For a particular car, we want to store the car type, details, and the manufacturer information. The tables below (Table 1 & 2) illustrates the data set.

Core Data Primer

A quick primer on Core Data follows; however, to really understand Core Data you should read Apple's *"Core Data Programming Guide"* on the developer.apple.com web site.

Core Data's focus is on objects rather than a traditional table-driven relational database approach. When storing data, you are actually storing an object's contents, where an object is represented by an Objective-C class that inherits from the NSManagedObject class. A typical application will have several objects used together, forming an object graph. For our sample data set, a car object contains car type, details, and manufacturer objects as members. Your application modifies the objects directly, when saving the objects the NSManagedObjectContext save method is called. Conversely, your applica-

You can download the sample application from this link:

http://www.docs-goldenbits.com/newsletters/coredata-sqlite/DataStoreTester.zip

Car	Type	Details	Manufacturer
Pinto, $16,000	Compact	AC, Automatic transmission	Ford, Dearborn MI, 100,000 employees
Tahoe, $45,000	SUV	Leather, Power Windows	GM, Dearborn, MI, 100,00 employees
Ferrari, $15,000	Sports Car	V12	Ferrari Italy.

Table 1

If you don't like cars, substitute surfboards. For example:

Bike	Type	Details	Manufacturer
Slinger, $850	High Performance	6', single fin	Infinity, Dana Point, CA 50 employees

Table 2

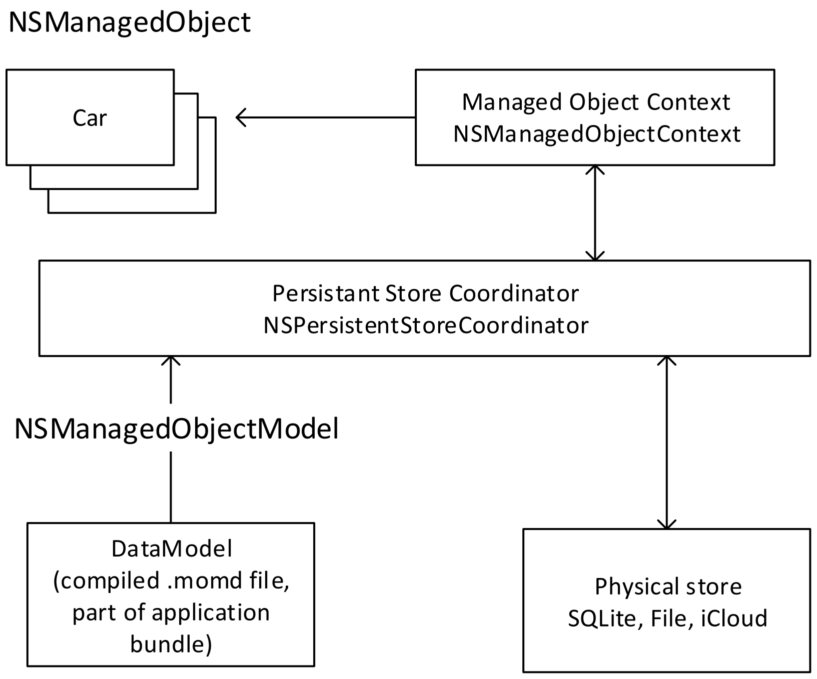


Figure 1

tion will fetch stored objects using the NSManagedObjectContext object. Core Data handles all of the details of saving the changes. The above figure (Figure 1) shows the main components of Core Data .

The Data Model is where you define your data objects and their relations. This is done using the Data Model Editor which is part of the XCode IDE. The data model file is stored locally (on your development system) as an XML file; however when you application is built, this file is compiled into a binary file (with a .momd extension) that is bundled with your iOS application. Each object is referred to as an "Entity" where an Entity contains one or more attributes. Don't get confused with the terminology here, an Entity is an object and an attribute is a member of your object. XCode will generate the source code (.m and .h files) for the classes defined in the Data Model Editor. To do so, in XCode select the data model and then the menu selection: "Editor/ Create NSManagedObject Subclass".

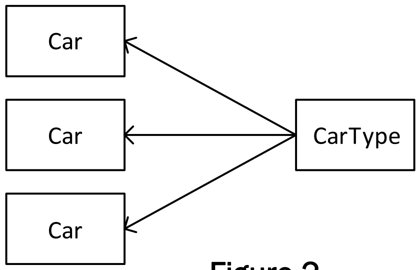


Figure 2

An important design consideration for your objects is the relationships between them. By relationship I am referring to when one object contains a reference to another. Is this a reference to many (one-to-many)? Or a reference each way too many (many-to-many)? For example, in the sample application the Car object references a CarType object where the CarType object is also referenced by other Car object instances. This is a one-to-many relationship between the CarType and Car object, one CarType can reference many Car objects. This is similar in concept to a SQL foreign key and is important to understand because any change to the CarType record will affect all of the Car objects. Figure 2 illustrates this relationship.

The Persistent Store Coordinator (NSPersistentStoreCoordinator) handles the details of the actual physical storage, whether the storage is

Core Data vs. SQLite

a SQLite database, binary file, or iCloud. Your application doesn't know or care, it just works with the application objects directly. The nice thing about using Core Data is the ability to use these different storage types seamlessly. The Persistent Store Coordinator can handle different store instances. For example, you might want to store frequently changing price data (such as gas prices) using iCloud and static data locally (such as gas station location). How you model, store, and manage your data should be driven by your application's requirements. The Persistent Store Coordinator uses the compiled data model file to determine the structure and organization of the objects being stored.

The Managed Object Context (NSManagedObjectContext) works with the Persistent Store Coordinator to fetch, save, and track the collection of objects. These are powerful features, the application does not have to track if a one object in a collection has been modified or the details of storing the objects. The Managed Object Context also acts as scratch pad of sorts for your object collection. If your application makes changes to the objects and later needs to discard these changes, no problem; the application can use the Undo Manager (NSUndoManager) or simply resets the Managed Object Context (using the `reset` method) and discard references to any of the objects.

There can be more than one Managed Object Context instance for a single store. For example, an application may use different contexts for different fetch results. As a result, an object instance can exist in both contexts simultaneously potentially causing data inconsistencies. Each managed object is assigned a unique id when the object is saved (a temporary id is assigned if the object has never been saved), an application can use this id to insure data consistency when using multiple contexts. However now the application is forced to track object changes, which doesn't make sense, that's the job of Cord Data. In short, use multiple contexts if there is a compelling reason

The Figure 3 below illustrates the concept of how the Managed Object Context manages the core data objects (NSManagedObject).

One of the downsides of the Managed Object Context is all of the objects are operated on together; it is not possible to save just one NSManagedObject instance, all are saved at once. If your application must work with a lot of NSManagedObjects at once, be careful about the amount of memory used. In the sample application, when creating 250,000 Car records, 260 Mbyte is used. To put this memory usage

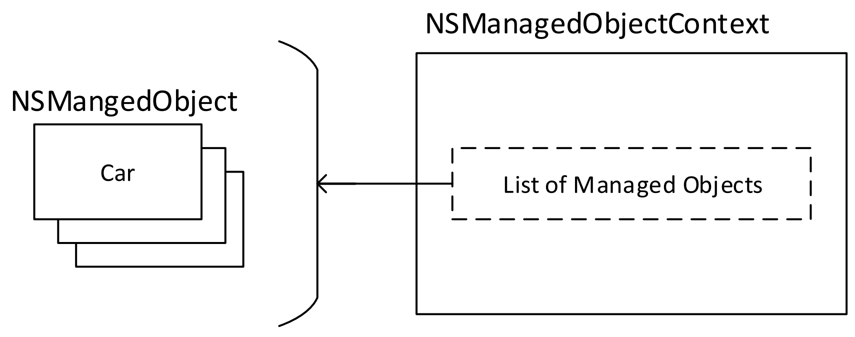


Figure 3

in context, the iPhone 5s has 1 GigaByte of memory, the iPhone 4s has 512 MByte of memory. On an iPhone 5s, if more than 350,000 records are created, the test application receives a memory warning from the phone iOS operating system directly. (`-(void) didReceiveMemoryWarning` is called). Yes 260 MByte is a lot, but remember that each of the 250,000 objects are in memory, fully realized, meaning all of the Car member variables are also in memory. If you restart the sample application and fetch these 250,000 records you just created, you will notice the amount of memory used is 105 Mbyte. Much less than the 260 MByte used to create the records – what happen? The answer is due to one of Core Data's powerful feathers – object faulting.

Object faulting is the ability of the Managed Object Context to load an object in memory when it is accessed, if the object is never accessed the memory is saved. This happens behind the scenes, your application runs along happily. In the attached sample application, when accessing the CarType from the Car object, if the CarType isn't in memory Core Data will automatically load it in memory with the correct data for you. The NSManagedObject method `isFault` enables your application to determine if the object is truly in memory or maybe loaded into memory when accessed.

Creating a managed object is a little different, instead of the usual `alloc init` pattern to create a new object instance, an application should call the `NSEntityDescription insertNewObjectForEntityForName` method using the Managed Object Context as an input argument. The following code snippet shows how the sample code Car object is created:

```
Car *car = [NSEntityDescription
    insertNewObjectForEntityForName:@"Car"
    inManagedObjectCotext:_managedObjectContext];
```

`NSEntityDescription insertNewObjectForEntitryForName` is actually a convenience method, underneath the hood an NSManagedObject is created for the entity @Car and inserted into the NSManagedObjectContext. The important point here is to understand that the Managed Object Context knows about all of the objects.

Core Data provides a lot of tools and capabilities to handle almost any type of data. The downside is that has a lengthy learning curve.

In the accompanying sample application, all of the Core Data code is located in the source files GB_CarsCoreData.m and .h.

SQLite

SQLite an open source light weight, robust, well supported, self-contained, cross platform, relational database that is very popular. Chances are that you have used SQLite at some point in the past; it is a very good option for storing application data. The SQLite web site is sqlite.org and is worth taking a moment and browsing the web site. Unlike most open source projects, the SQLite documentation and support is excellent. SQLite version 3.7.13 is supported on iOS 7 and 8. SQLite is also supported on Android and Windows phones, which is a huge benefit. If you are developing a mobile application targeting multiple platforms, you would be crazy not to use SQLite.

SQLite stores data in tables, where a table contains one or more columns, each column contains data for a specific data type. One of the tasks an RDMS is good at is handling a normalized data set. Instead of each table containing all of the possible columns of data, resulting in duplicated data for each table row, the tables are typically organized to reference another table (via a key), that contains the duplicated data. In the sample application, the manufacture information is kept in a separate mfg_info table which is referenced by the car table.

You application interacts with SQLite via the SQL language and the SQLite C api (see sqlite.h). This means to insert data into the database an "insert" SQL statement must be created along with the data to insert. The following snippet shows the insert statement used by the sample application.

```
insert into car (model, msrp, year,
mfg_id, cartype_id, cardetails_id)
values ("Pinto", 17000, 1970, 1, 2,
3)
```

Conversely to fetch data a "select" SQL statement is needed. For example, the following snippet shows how a select statement is used to fetch all of the car records:

```
  select * from car
```

This is the simple case where all of the car table information is fetched; however, things get complicated quickly when you want to get all of the information for a specific car. In the sample application, this means multiple join statements as show in the next code snippet.

As you can see, even a moderately involved data set requires a solid understanding of the SQL syntax and behavior. Another drawback to SQLite is the tedious nature of insuring that the columns and data are in the correct order. For example, when iterating through the SELECT results if you make a mistake for the column number in the sqlite3_column_*() call, the wrong data will be fetched. This can easily happen when additional columns are added or removed later.

```
select car.model, car.year,
car.msrp, cardetails.detailgroup,
cardetails.info_1,
cardetails.info_2, manufactur-
er.hq_location, manufacturer.name,
manufacturer.num_employees,
cartype.type, cartype.type_desc from
car inner join cartype on
car.cartype_id = cartype.id inner
join cardetails on car.cardetails_id
= cardetails.id inner join manufac-
turer on car.mfg_id = manufactur-
er.id where car.id = %@
```

SQLite implements a C API, thus you will have to convert between NSStrings * and char * strings when your application interacts with the database. There are Objective C wrappers for SQLite, the most popular being the FMDB which can be downloaded from this URL: https://github.com/ccgus/fmdb. For most applications, using the SQLite API directly is not overly cumbersome. The attached sample application uses the SQLite API directly.

Differences between Core Data and SQLite

Core Data and SQLIte are fundamentally different, so it is difficult to compare the two technologies. However, from a developer's viewpoint can gauge how these differences manifest themselves in an application. What is the cost and benefit of each approach? The accompanying sample iOS iPhone application is designed to test both technologies. From the main screen, select either Core Data or SQLite, create the desired number of records, and select the Car tab to fetch the records. The memory usage number is for the entire application and is updated once per second. The storage size is the size of the actual persistent file.

The following tables shows the differences between Core Data and SQLite.

Core Data vs. SQLite

Memory Usage

Core Data uses more memory, from 40% to 100% more than SQLite. This makes sense considering how Core Data is designed. Specifically how the NSManagedObjectContext tracks all of the objects, where each object has a memory footprint of some size depending if the object's contents have been faulted in (realized in memory versus being a fault). See Table 3 below.

NOTE: 5s == iPhone 5s, 4s == iPhone 4s.

Record Type	50,000 Recs	100,000 Recs	200,000 Recs
Core Data	32 MB (5s) 24 MB (4s)	53 MB (5s) 38 MB (4s)	91 MB (5s) 67 MB (4s)
SQLite	21 MB (5s) 16 MB (4s)	29 MB (5s) 22 MB (4s)	46 MB (5s) 36 MB (4s)

Table 3

A side note about memory testing. When creating records the entire Car record is created in memory and the higher memory usage reflects this. However, most user scenarios do not involve creating a large number of records. A better measurement of the memory usage is on application startup and fetching all of the Car records, which is exactly what the above table reflects. To get this measurement in testing, the application was restarted (terminated versus put in the background) and the records were re-fetched.

Storage Size

Core Data uses more storage space, a lot more storage space, approximately 4x more than SQLite as Table 4 below shows.

Record Type	50,000 Recs	100,000 Recs	200,000 Recs
Core Data	6532 KB (5s) 6412 KB (4s)	13296 KB (5s) 13180 KB (4s)	27004 KB (5s) 2,912 KB (4s)
SQLite	1676 KB (5s) 1676 KB (4s)	3364 KB (5s) 3364 KB (4s)	6824 KB (5s) 6824 KB (4s)

Table 4

Speed

Both Core Data and SQLite are fast when fetching records. For the iPhone 5s, the differences are slight; however, for the 4s, Core Data is nearly 2x faster. This table shows just one operation, fetching all of the Car records. See Table 5 (below) for details.

Record Type	50,000 Recs	100,000 Recs	200,000 Recs
Core Data	107 msec (5s) 397 msec (4s)	230 msec (5s) 850 msec (4s)	475 msec (5s) 1644 msec (4s))
SQLite	140 msec (5s) 730 msec (4s)	280 msec (5s) 1447 msec (4s)	580 msec (5s) 3077 msec (5s)

Table 5

Core Data vs. SQLite

Summary

No one technology, framework, or diet pill will make your life instantly better. When designing an application, a lot of design criteria must be considered.

Hopefully this article will provide you with some guidance when selecting a storage approach for your next iOS application. I encourage you to download the sample application and take it for a spin.

Some final key points:

SQLite:
SQLite is, as advertised, light weight.
SQLite uses less memory and storage space.
SQLite can be tedious and error prone to code.
SQLite is supported on Android and Windows Phone.

Core Data:
Learning curve, takes some study to understand.
Objects are easier to work with.
Underlying storage details are handled atomically (support for iCloud).
Undo and Redo features.

Golden Bits Project Experience

Embedded Devices/OS. Experience with embedded OS systems: Linux, TI DSP, and Threadx. Developed application and system level code including USB enhancements and embedded management system for a large blade enclosure. Ported embedded Linux driver code to Power PC 44GX and Broadcom 1255 processors.

Digital Media. Designed and developed media pipeline for encoding video using H264, Mpeg encoders.

Device Drivers. Windows, Linux, Mac. Developed a variety of Windows, Linux, and Mac rivers to handle network packet inspection, USB devices, SCSI Fibre channel adapters, ethernet adapters, and job scheduling to a device.

Set Top Box. Helped port a STB to new Broadcom 7405 chip. Developed a script language and compiler used to code the television UI (guide menus, channel select).

Security. Experienced with PKI, x509 certificates, IPsec (StrongSwan), and RSA encryption.

SCSI Port driver for Fibre Channel. Designed the operating system layer for a SCSI storage driver (XP, Win2K, NT, Linux) for a fibre channel HBA (host bus adapter – PCI/SBUS card).