Extracted from:

# Core Data in Objective-C, Third Edition

## Data Storage and Management for iOS and OS X



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Third Edition

# Core Data in Objective-C

## Data Storage and Management for iOS and OS X

Marcus S. Zarra

*Edited by Jacquelyn Carter*

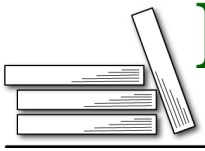
# Core Data in Objective-C, Third Edition

## Data Storage and Management for iOS and OS X

Marcus S. Zarra

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



Printed in the United States of America.
ISBN-13: 978-1-68050-123-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2016

Over the years that Core Data has been in production, there have been a few complaints about the framework that struck home and were accurate. Easily the most well-known complaint was regarding the ability to change a value in a large number of objects without requiring those objects to all be loaded into memory and then persisted back out to disk. The second most well-known complaint was about deleting a large number of objects. Again, the desire is to delete a large number of objects without having to load them into memory and then write back out to the persistent store again.

Both of these complaints only apply to the `NSSQLite` store. Since atomic stores such as the binary store require all of the data to be in memory, there's no issue with doing bulk changes or bulk deletes. But with the SQLite store, either of these changes can be incredibly CPU, disk, and memory intensive.

With the introduction of iOS 8.0 and OS X Yosemite, the first complaint was addressed. With the introduction of iOS 9.0 and OS X El Capitan, the second complaint was addressed.

## Running with Scissors

Both of these APIs work by making changes directly on disk. When we use either of these APIs, Core Data will construct the appropriate SQL calls and then pass them to SQLite. Nothing gets loaded into memory and therefore the API is executed very quickly—just slightly slower than SQLite itself.

If we can just make changes and/or deletes on disk and avoid having to load them all into memory, why don't we just do that all the time?

This API comes at a fairly significant cost. The changes that we make on disk aren't passed to the `NSManagedObjectContext` instances in our application.

This means that we can very easily make a change to the data on disk and then our `NSManagedObjectContext` will try to make a different change and cause issues. When the first API was introduced, Apple likened these APIs to running with scissors. You can do it, but the risk is greater.

First, data validation is performed in memory. When we make a change directly on disk we're bypassing the validation steps that Core Data normally performs. This means we can break a relationship and have dangling references, we can inject data that's not valid, and so forth. Worse, our application won't notice the issue until it attempts to load the data later and then the user is left in a bad state.

Second, when the changes are made on disk, the version number of the object is updated (as it should be). However, since nothing in memory knows of this

change, the version number in memory won't match. If Core Data were to attempt to do a save of an object in this state, a merge conflict would result with a potentially negative outcome.

And of course there's the obvious issue: our user interface won't know about the change and therefore the older data will still be displayed.

We can address these issues, but doing so requires more code on our part. Let's start by looking at a bulk update.

## Doing Bulk Updates

Doing a bulk update isn't a common event in most application life cycles. Selecting a large number of emails, or a large number of news items, and marking them as read is a common example of doing a bulk update. Although these situations do occur, they are unusual and shouldn't be considered a core function of the application. Bulk updates are generally used to get us out of a coding or design "corner."

In our recipes application, we're going to use the bulk update API to change the values of some of our recipes on the first launch after a migration. When we migrate the application to the fourth version, we'll add a Boolean to indicate whether it's a favorite recipe; the default for all recipes is `NO`. Once the migration is complete, we then want to go through all of the recipes and change that default to `YES` for some of them.

To start with, we want to detect if this change has already been made. There are several ways to accomplish this, and we've used other methods in the past. In this demonstration, we're going to use the metadata that's contained with the persistent store to determine whether the change has already been processed. This change to the initialization of our Core Data stack determines whether we need to do any post-migration processing.

**Batch/PPRecipes/PPRDataController.m**

```
dispatch_queue_t queue = NULL;
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{
  NSFileManager *fileManager = [NSFileManager defaultManager];
  NSURL *storeURL = nil;
  storeURL = [[fileManager URLsForDirectory:NSDocumentDirectory
                                  inDomains:NSUserDomainMask] lastObject];
  storeURL = [storeURL URLByAppendingPathComponent:@"PPRecipes.sqlite"];

  NSMutableDictionary *options = [[NSMutableDictionary alloc] init];
  [options setValue:[NSNumber numberWithBool:YES]
            forKey:NSMigratePersistentStoresAutomaticallyOption];
```

```
[options setValue:[NSNumber numberWithBool:YES]
           forKey:NSInferMappingModelAutomaticallyOption];
NSError *error = nil;
NSPersistentStore *store = nil;
store = [psc addPersistentStoreWithType:NSSQLiteStoreType
                          configuration:nil
                                    URL:storeURL
                                options:options
                                  error:&error];
if (!store) {
  ALog(@"Error adding persistent store to coordinator %@\n%@",
       [error localizedDescription], [error userInfo]);
}

NSDictionary *metadata = [store metadata];
if (!metadata[FAVORITE_METADATA_KEY]) {
  [self bulkUpdateFavorites];
}
```

Every persistent store contains metadata. The metadata resolves to a NSDictionary that we can query. We can also update this metadata as needed.

In this part of the code, we're looking for a key named FAVORITE_METADATA_KEY. If that key exists, then we know that this particular bit of post-processing has already been done. If the key is missing, we need to perform the task.

**Batch/PPRecipes/PPRDataController.m**

```
- (void)bulkUpdateFavorites
{
  NSManagedObjectContext *moc = [self writerContext];

  [moc performBlock:^{
    NSBatchUpdateRequest *request = nil;
    NSMutableDictionary *propertyChanges = nil;
    NSPredicate *pred = nil;
    NSBatchUpdateResult *result = nil;
    NSError *error = nil;

    request = [[NSBatchUpdateRequest alloc] initWithEntityName:@"Recipe"];

    NSDate *aMonthAgo = [self dateFrom1MonthAgo];
    pred = [NSPredicate predicateWithFormat:@"lastUsed >= %@", aMonthAgo];
    [request setPredicate:pred];

    propertyChanges = [NSMutableDictionary new];
    propertyChanges[@"favorite"] = @(YES);
    [request setPropertiesToUpdate:propertyChanges];
    [request setResultType:NSUpdatedObjectIDsResultType];

    result = [moc executeRequest:request error:&error];
    if (!result) {
      ALog(@"Failed to execute batch update: %@\n%@",
           [error localizedDescription], [error userInfo]);
```

```
}
//Notify the contexts of the changes
[self mergeExternalChanges:[result result] ofType:NSUpdatedObjectsKey];
```

The -bulkUpdateFavorites method is where we're using the bulk update API. Once we are positive that we are executing on the proper queue for our main NSManagedObjectContext, we start off by creating a new NSBatchUpdateRequest. The NSBatchUpdateRequest is a subclass of NSPersistentStoreRequest, which is a class that was introduced in OS X 10.7 and iOS 5. An NSBatchUpdateRequest contains all of the properties that Core Data needs to execute our update directly on disk. First, we initialize the request with the name of the entity we want to access. We then pass it the predicate to filter the entities that will be updated.

In this example, we're going to find all the recipe entities that have been used in the last month and mark those as favorites. We construct a date object that represents one month ago and then pass that to the predicate and then pass the predicate into the NSBatchUpdateRequest.

In addition to the predicate, we need to tell Core Data what properties need to be changed. We do this with a NSDictionary, where the key is the property to change and the value is the new value to apply to the entity. As you can see, we don't have a lot of control over the changes here. There's no logic that we can apply. These are simple, brute-force data changes at the database/persistent store level.

Once we pass the dictionary to NSBatchUpdateRequest via -propertiesToUpdate, we can define what kind of result we want back. We have three options:

- NSStatusOnlyResultType, which won't return anything. If we aren't going to do anything with the response, there's no reason to ask for one.
- NSUpdatedObjectIDsResultType, which will give us the NSManagedObjectIDs for each changed entity. If we're going to notify the application of the changes, then we'll want these to do the notification.
- NSUpdatedObjectsCountResultType, which will give us a simple count of the number of entities altered.

In this example, we'll walk through updating the user interface of the changes, so we'll ask for the NSManagedObjectID instances back.

Once we have the NSBatchUpdateRequest fully constructed, we can then hand it off to any NSManagedObjectContext we want for processing. Here I'm using the writer context because it's closest to the NSPersistentStoreCoordinator. But since

this API doesn't notify the NSManagedObjectContext of the change, it really doesn't matter which context we use.

The call to -executeRequest: error: returns a simple id, and it's up to us to know what the call is giving back to us. Since we set the -resultType to be NSUpdatedObjectIDsResultType, we know that we're going to be getting an NSArray back.

If we get back a nil from this API call, we know that there was an error and we can respond to that error. As always in the code in this book, we're going to treat the error as a fatal condition and crash. How to respond to those errors is a business decision determined by your application's design and requirements.

The call to -executeRequest: error: is a blocking call. This means that the call can take a significant amount of time—still far less than loading all of the objects into memory, performing the change, and saving them back out to disk, but it will take some time. This is another argument for using the API against the private writer context instead of the context that the user interface is associated with.

## Notifying the Application of Changes

Once we've made changes on disk, we need to notify the contexts of these changes.

In this particular example, doing so isn't strictly necessary. Most likely the user interface hasn't touched any of these objects yet. If the objects haven't been loaded into memory, there's no risk of a conflict. However, it's best that we don't assume they haven't been loaded yet. Users can be very clever.

There are two basic ways to notify our NSManagedObjectContext instances of the changes. We can reset each object individually in each NSManagedObjectContext that it might be associated with, or we can use the new API that was added in iOS 9.0. Let's look at the harder way first.

### Manual Object Refreshing

If the situation calls for it, we can instruct each instance of NSManagedObject to refresh individually. This might make sense if we have a view that's observing one specific instance of an entity or we have a user interface that's watching a small subset of objects.

Batch/PPRecipes/PPRDataController.m

```
- (void)manuallyRefreshObjects:(NSArray*)oIDArray;
{
```

```
  NSManagedObjectContext *moc = [self managedObjectContext];
  [moc performBlockAndWait:^{
    [oIDArray enumerateObjectsUsingBlock:^(NSManagedObjectID *objectID,
                                           NSUInteger index, BOOL *stop) {
      NSManagedObject *object = [moc objectRegisteredForID:objectID];
      if (!object || [object isFault]) return;
      [moc refreshObject:object mergeChanges:YES];
    }];
  }];
}
```

The -manuallyRefreshObjects: method accepts an NSArray of NSManagedObjectID instances and walks through that array. Because this method is going to be working with the NSManagedObjectContext that's on the main queue, we want a guarantee that our code will also be executed on the main queue. Therefore, we start by executing the code in a -performBlock: to ensure we're on the correct queue.

Inside the block we iterate over the array of NSManagedObjectID instances and retrieve a reference to the associated NSManagedObject. Note that we're using -objectRegisteredForID: in this method. -objectRegisteredForID: will only return a non-nil result if the object is registered with the referenced NSManagedObjectContext. If it isn't referenced, we certainly don't want to load it, so this method is a perfect fit. From that call we need to see if we got an object back and that it isn't a fault. If it is a fault, we don't need to refresh it because the values aren't in memory.

Once we confirm the NSManagedObject is registered and isn't a fault, we call -refreshObject: mergeChanges:, which will force the object to reload from the persistent store.

That's a fair amount of work for *each individual NSManagedObject* in the array. Fortunately, there's an easier way.

## Remote Notifications

As part of the update for iOS 9.0 and OS X 10.11, the Core Data team gave us a new class method to handle this situation. We can now call +[NSManagedObjectContext mergeChangesFromRemoteContextSave: intoContexts:] to handle changes that occurred outside our Core Data stack.

The method accepts a dictionary of arrays of NSManagedObjectID instances (either true NSManagedObjectID objects or NSURL representations of them) and also accepts an array of NSManagedObjectContext instances.

Like traditional NSManagedObjectContextDidSaveNotification calls, we can pass in objects that are in three possible states: inserted, updated, and deleted.

Batch/PPRecipes/PPRDataController.m

```
- (void)mergeExternalChanges:(NSArray*)oIDArray ofType:(NSString*)type
{
  NSDictionary *save = @{type : oIDArray};

  NSArray *contexts = @[[self managedObjectContext], [self writerContext]];

  [NSManagedObjectContext mergeChangesFromRemoteContextSave:save
                                                intoContexts:contexts];
}
```

Note in this method that we aren't concerned with what queue things are being executed on. The API call handles that for us. Also note that we're able to update all of the contexts that exist in our application at once.

This method will basically do the same thing that we did in our manual method. But this method will handle it for every context we give it, and it's faster than our manual method. This is the recommended way to consume remote notifications. With this method, our original bulk update call can easily notify the rest of the application of the changes.