Extracted from:

Core Data, 2nd Edition

Data Storage and Management for iOS, OS X, and iCloud

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Core Data
Second Edition

Data Storage and Management for iOS, OS X, and iCloud

Marcus S. Zarra

Edited by Colleen Toporek

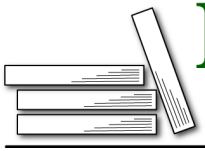
Core Data, 2nd Edition

Data Storage and Management for iOS, OS X, and iCloud

Marcus S. Zarra

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Colleen Toporek (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-08-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2013

4.2 Optimizing Your Data Model

When we design our data model, we need to consider several factors. Where we put our binary data can be extremely important because its size and storage location plays a key role in the performance of our application. Likewise, relationships must be carefully balanced and used appropriately. Also, entity inheritance, a powerful feature of Core Data, must be used with a delicate hand because the underlying structure may be surprising.

Although it is easy to think of Core Data as a database API, we must remember that it is not and that structuring the data with data normalization may not yield the most efficient results. In many cases, denormalizing the data can yield greater performance gains.

Where to Put Binary Data

One of the easiest ways to kill performance in a Core Data application is to stick large amounts of binary data into frequently accessed tables. For example, if we were to put the pictures of our recipes into the recipe table, we would start seeing performance degradation after only a couple hundred recipes had been added. Every time we accessed a `Recipe` entity, we would have to load its image data, even if we were not going to display the image. Since our application displays all the recipes in a list, this means every image would reside in memory immediately upon launch and remain there until the application quit. Imagine this situation with a few thousand recipes!

But where do we draw the line? What is considered a small enough piece of binary data to fit into a table, and what should not be put into the repository at all?

If you are developing an application that is targeting iOS 6.0 or greater (or Mac OS X 10.8 or greater), the answer is simple: turn on external binary storage in the model and let Core Data solve the problem for you (see Figure 13, *Turn on the external record flag*, on page 6). This feature instructs Core Data to determine how to store binary data. With this flag on, Core Data decides whether the image is small enough to store inside of the SQLite file or whether it is too big and therefore should be stored on disk separately. In either case, the decision is an implementation detail from the perspective of our application. We access the binary data just like any other attribute on the entity.

If the application is still targeting an older version of the operating system (iOS or Mac OS X), then the application is responsible for dealing with binary data in a performant way.

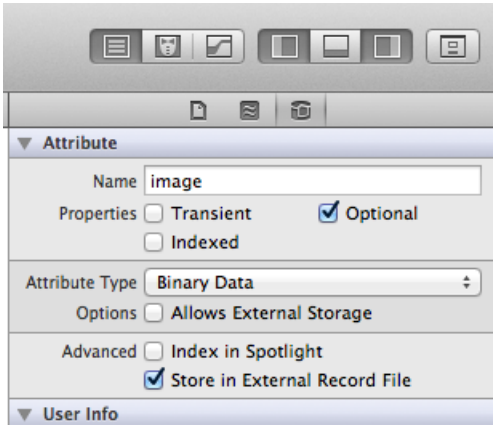


Figure 13—Turn on the external record flag.

Small Binary Data

Anything smaller than 100 kilobytes is considered to be small binary data. Icons or small avatars are a couple examples of data of this size. When working with something this small, it is most efficient to store it directly as a property value in its corresponding table. The performance impact of binary data this size is negligible. The transformable attribute type is ideal for this use.

Medium Binary Data

Medium binary data is anything larger than 100 kilobytes and smaller than 1 megabyte in size. Average-sized images and small audio clips are a few examples of data in this size range. Data of this size can also be stored directly in the repository. However, the data should be stored in its own table on the other end of a relationship with the primary tables. This allows the binary data to remain a fault until it is actually needed. In the previous recipe example, even though the `Recipe` entity would be loaded into memory for display, the image would be loaded only when it is needed by the UI.

SQLite has shown itself to be quite efficient at disk access. There are cases where loading data from the SQLite store can actually be faster than direct disk access. This is one of the reasons why medium binary data can be stored directly in the repository.

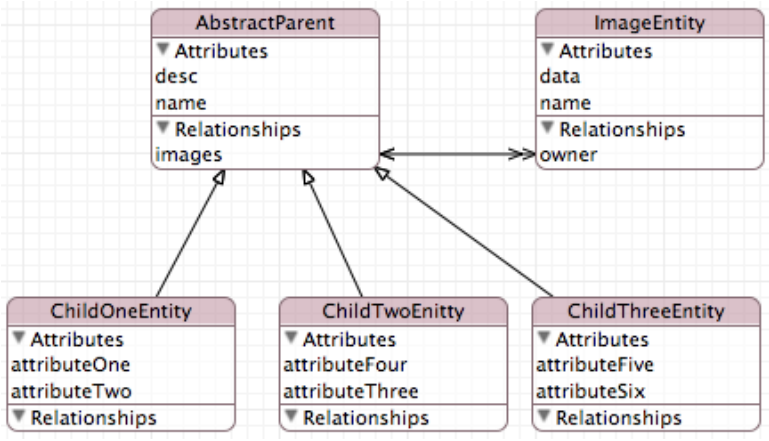
Large Binary Data

Large binary data is anything greater than 1 megabyte in size. Large images, audio files, and video files are just some examples of data of this size. Any

binary data of this size should be stored on disk as opposed to in the repository. When working with data of this size, it is best to store its path information directly in the primary entity (such as the Recipe entity) and store the binary data in a known location on disk (such as in the Application Support subdirectory for your application).

Entity Inheritance

Entity inheritance is a very powerful feature within Core Data. It allows you to build an object-like inheritance tree in your data model. However, this feature comes at a rather large cost. For example, let's look at an example model that makes moderate use of entity inheritance, as shown here:



The object model itself looks quite reasonable. We are sharing name, desc, and a one-to-many relationship to the ImageEntity. However, the underlying table structure actually looks like this:

	name	desc	attributeOne	attributeTwo	attributeThree	attributeFour	attributeFive	attributeSix
Entity One								
Entity Two								
Entity Three								

The reason for this is how Core Data handles the object model to relational table mapping. Instead of creating one table for each child object, Core Data creates one large table that includes all the properties for the parent entity as well as its children. The end result is an extremely wide and tall table in the database with a high percentage of empty values.

Although the entity inheritance feature of Core Data is extremely useful, we should be aware of what is going on underneath the object model to avoid a performance penalty. We should not treat entity inheritance as an equal to object inheritance. There is certainly some overlap, but they are not equal, and treating them as such will have a negative impact on the performance of the repository.

Denormalizing Data to Improve Performance

Although the most powerful persistent store available for Core Data is a database, we must always be conscious of the fact that Core Data is not just a database. Core Data is an object hierarchy that can be persisted to a database format. The difference is subtle but important. Core Data is first a collection of objects that we use to display data in a user interface of some form and allow the user to access that data. Therefore, although database normalization might be the first place to look for performance improvements, we should not take it too far. There are six levels of database normalization,[1] but a Core Data repository should rarely, if ever, be taken beyond the second level. There are several cases where we can gain a greater performance benefit by denormalizing the data.

Search-Only Properties

Searching within properties can be quite expensive. For properties that have a large amount of text or, worse, Unicode text, a single search field can cause a huge performance hit. One way to improve this situation is to create a derived attribute based on the text in an entity. For example, searching in our description property of the `Recipe` entity can potentially be very expensive if the user has verbose descriptions and/or uses Unicode characters in the description.

To improve the performance of searches in this field, we could create a second property on the `Recipe` entity that strips the Unicode characters from the description and also removes common words such as *a*, *the*, *and*, and so on. If we then perform the search on this derived property, we can drastically improve search performance.

The downside to using search-only properties is that we need to maintain them. Every time the description field is edited, we need to update the derived property as well.

Expensive Calculations

In a normalized database, calculated values are not stored. It is considered cheaper to recalculate the value as needed than to store it in the database. However, from a user experience point of view, the opposite can frequently be true. In cases where the calculation takes a human-noticeable amount of time, it may very well be better for the user if we were to store that calculation in the entity and recalculate it only when one of its dependent values has

1. See http://en.wikipedia.org/wiki/Database_normalization for details.

changed. For example, if we store the first and last names of a user in our Core Data repository, it might make sense to store the full name as well.

Intelligent Relationships

Relationships in a Core Data model are like salt in a cooking recipe. Too much and you ruin the recipe; too little and something is missing. Fortunately, there are some simple rules we can follow when it comes to relationships in a Core Data repository.

Follow the Object Model

Core Data is first and foremost an object model. The entities in our model should represent the data as accurately as possible. Just because a value might be duplicated across several objects (or rows from the database point of view) does not mean it should be extruded into its own table. Many times it is more efficient for us to store that string several times over in the entity itself than to traverse a relationship to get it.

Traversing a relationship is generally more expensive than accessing an attribute on the entity. Therefore, if the value being stored is simple, it's better to leave it in the entity it is associated with.

Separate Commonly Used from Rarely Used Data

If the object design calls for a one-to-many relationship or a many-to-many relationship, we should definitely create a relationship for it. This is usually the case where the data is more than a single property or contains binary data or would be difficult to properly model inside the parent object. For example, if we have a user entity, it is more efficient to store the user's address in its own object as opposed to having several attributes in the user object for address, city, state, postal code, and so on.

A balance needs to be carefully maintained between what is stored on the other end of a relationship and what is stored in the primary entity. Crossing key paths is more expensive than accessing attributes, but creating objects that are very wide also slows down data access.