Core Data Best Practices

Session 214

Ben Trumbull
Core Data Engineering Manager

Today's Roadmap

- Concurrency
- Nested Contexts
- Performance
- Schema Design
- Search Optimization

Topics

- Using Core Data with multiple threads
- Sharing unsaved changes between contexts
- Debugging performance with Instruments
- Tuning your model
- Improving your predicates

Concurrency

Challenges

- Thread safety
- Transactionality
- Performance

NSManagedObjectContext Concurrency

NSManagedObjectContext Concurrency

```
[moc performSelectorOnMainThread:
                @selector(mergeChangesFromContextDidSaveNotification:)
                withObject:note
                waitUntilDone:NO];
```

NSManagedObjectContext Concurrency

- Block support

```
NSManagedObjectContext -performBlock:
NSManagedObjectContext -performBlockAndWait:
```

NSManagedObjectContext Concurrency

- Block support

```
NSManagedObjectContext -performBlock:
NSManagedObjectContext -performBlockAndWait:
```

```
[[NSManagedObjectContext alloc] initWithConcurrencyType:NSMainThreadConcurrencyType];
[moc performBlock:^{
   [moc mergeChangesFromContextDidSaveNotification:aNotification];
}];
```

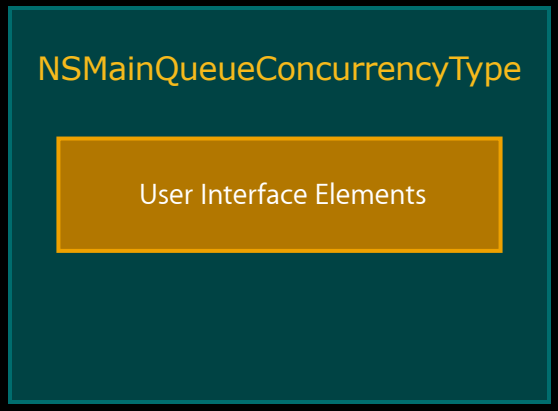
Core Data Concurrency Options

```
NSManagedObjectContext –initWithConcurrencyType:
```

Core Data Concurrency Options

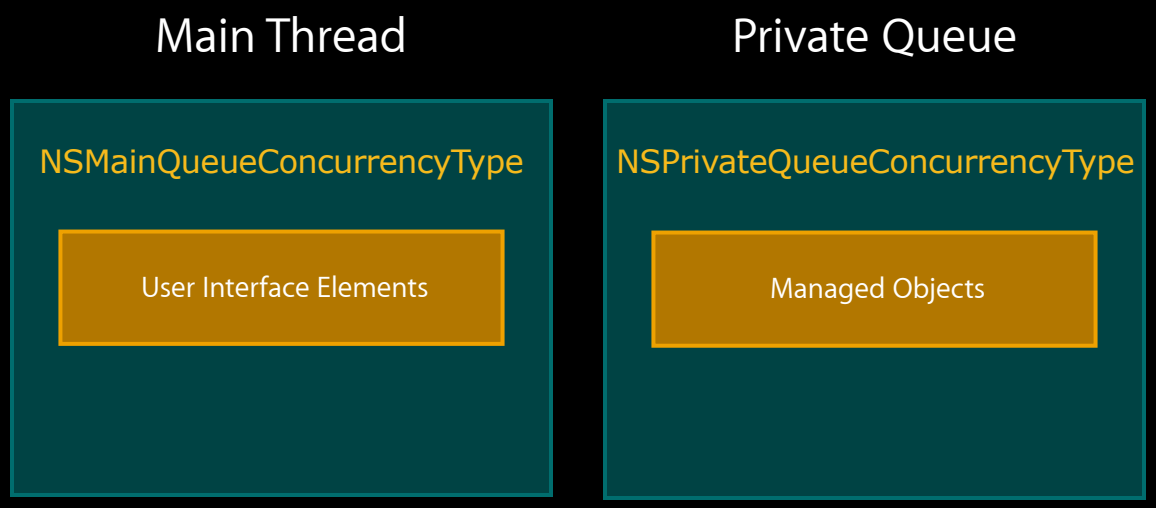
```
NSManagedObjectContext –initWithConcurrencyType:
```

Main Thread



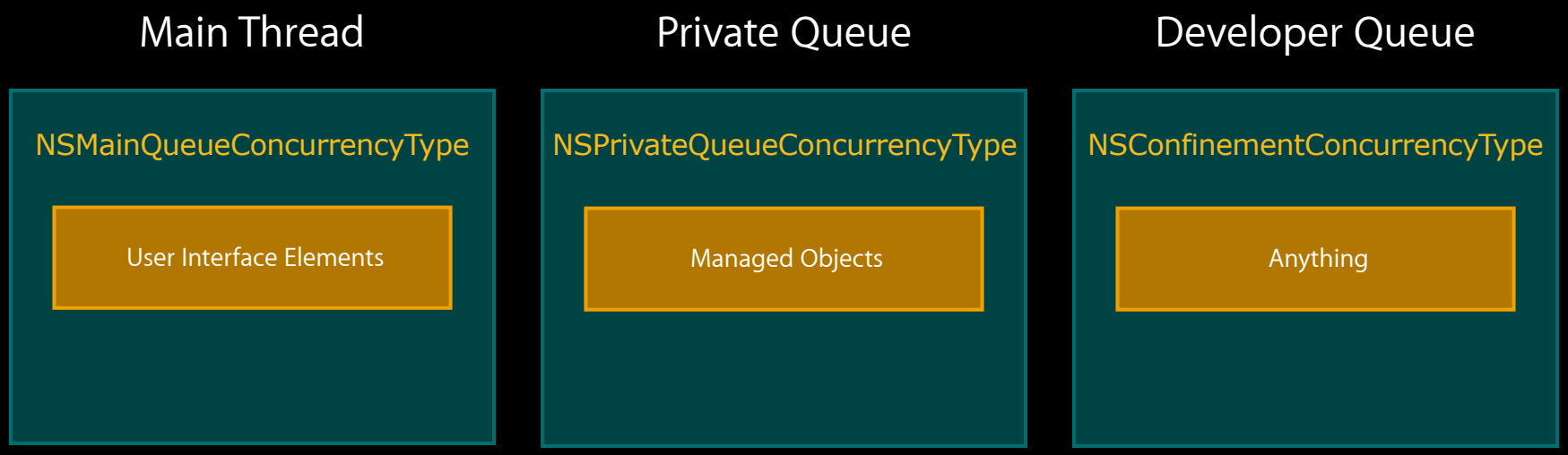
Core Data Concurrency Options

```
NSManagedObjectContext –initWithConcurrencyType:
```



Core Data Concurrency Options

```
NSManagedObjectContext –initWithConcurrencyType:
```



NSConfinementConcurrencyType

- Separate contexts for each thread
- MOCs only used on thread or queue that created them
- Default, legacy option

Confinement with Thread or Queue

- Thread or serialized queue as single control flow
- Serialized dispatch queues
- NSOperationQueue with maximum concurrency one

Thread Confinement

- Safe and efficient for transactions
- Easy to understand
- But harder to manage

Thread Confinement Issues

- Tracking which context goes with which thread
- Potentially keeping extra threads around
- Main thread behaviors inferred
- User events are runloop driven

NSPrivateQueueConcurrencyType

- MOC maintains its own serialized queue
- Can only be used on its own private queue
- Use `-performBlock:` and `-performBlockAndWait:` from other threads
- Within block use APIs normally

Queue Is Private

- Do not use `dispatch_get_current_queue`
- Callback to your own queue with `dispatch_sync`
- Capture references in your blocks

Private Queue Advantages

- MOC responsible for routing blocks to correct queue
- Other threads just call `-performBlock:`
- Can be created from any other thread
- Idle queues more efficient than extra threads

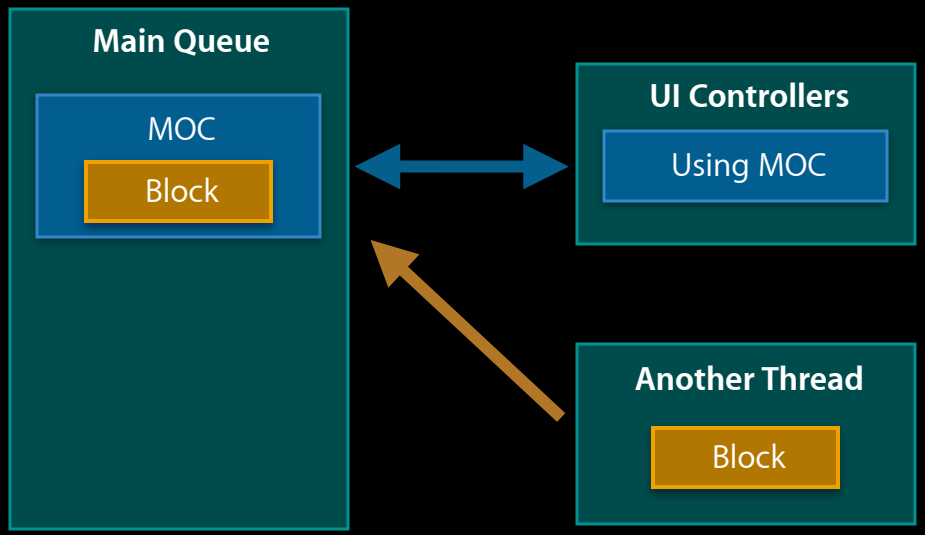
NSMainQueueConcurrencyType

- Similar to private queue
- Queue is always the main queue
- Non-main threads must use `-performBlock:`
- User events driven by main runloop

NSMainQueueConcurrencyType

- UI and controllers on main thread can use
- Great for receiving results from background
- Always uses main thread behaviors

Using Main Queue MOCs



What's a User Event?

- Automatic as application main event loop
- Provides
  - Change coalescing
  - Delete propagation
  - Undo
  - NSNotifications
- Time in between calls to `-processPendingChanges`

For All Concurrency Types

- Managed objects owned by their context
- ObjectIDs are safe, immutable value objects
- Retain, release are always thread safe on Core Data objects

Good Times to Pass Updates Around

- NSManagedObjectContextObjectsDidChangeNotification
- NSManagedObjectContextDidSaveNotification

Refreshing Other MOCs After Save

`mergeChangesFromContextDidSaveNotification:`

- You are only responsible for thread safety of receiver
- Core Data handles issues with notification parameter

Useful NSManagedObject Methods

```
changedValuesForCurrentEvent
changedValues
committedValuesForKeys
```

Block-Based APIs

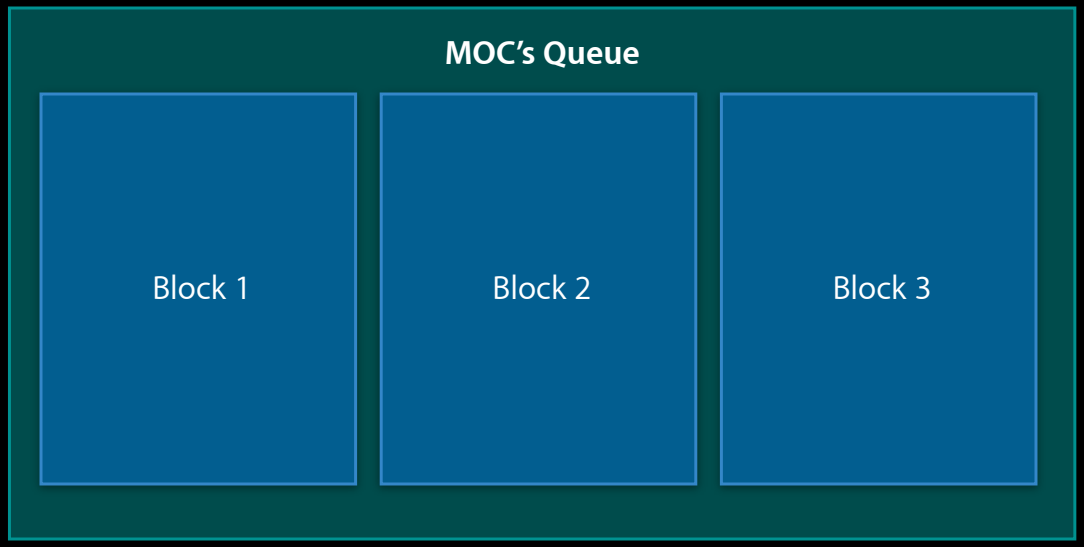
Challenges

- Passing work to other threads
- Demarcating cohesive changes
- Integrating with platform concurrency APIs

-performBlock:

- Asynchronous
- A “user event”
- Convenient autorelease pool
- No support for reentrancy
- Illegal to throw an exception out of your block

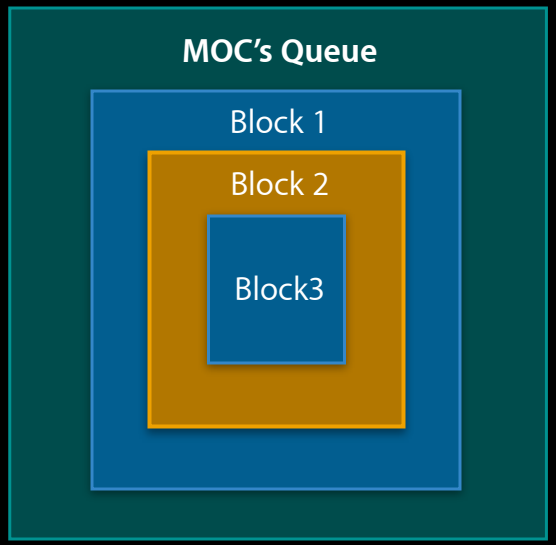
Recursive performBlock



-performBlockAndWait:

- Synchronous
- Not an event
- No autorelease pool
- Supports reentrancy
- Illegal to throw an exception out of your block

Recursive performBlockAndWait



NSManagedObjectContext Block APIs

- Fast
- Lightweight
- Serialized
- Changes scoped by block

Working with Data Between Blocks

- ObjectIDs often useful
  - Rematerialize into MO with `objectWithID:`
  - `objectWithID` will reuse cached data
- Also, okay to retain MOs but not look or use outside block
- Use __block variables
- Remember NSError are autoreleased

Fetching from a Private Queue MOC

```
__block NSArray* oids = nil;

[context performAndWait:^(NSManagedObjectContext* moc) {
  NSError *error = nil;
  NSArray* results = [moc executeFetchRequest:fr error:&error];
  if (results != nil) {
    oids = [results valueForKey:@”objectID”];
  } else {
     // handle error
  }
}];

NSLog(@"retrieved %d items", (int)[oids count]);
```

Coordinating Using Blocks

```
__block dispatch_queue_t yourQueue;

[context perform:^(NSManagedObjectContext* moc) {
     // work
     dispatch_sync(yourQueue, ^(){
        // callback work
     });
}];
```

Coordinating Using Semaphores

```
__block dispatch_semaphore_t waiter = dispatch_semaphore_create(0);

[context perform:^(NSManagedObjectContext* moc) {
     // work
     dispatch_semaphore_signal(waiter);
}];

dispatch_semaphore_wait(waiter, yourtimeout);
```

Interfacing with libdispatch

- Create a dispatch group
- Call `dispatch_group_enter`
- Worker block call `dispatch_group_leave`
- Use `dispatch_group_wait` and `dispatch_group_notify` normally

Coordinating Using Groups

```
__block dispatch_group_t group = dispatch_group_create();

dispatch_group_enter(group);

[context perform:^(NSManagedObjectContext* moc) {
     // work
     dispatch_group_leave(group);
}];

dispatch_group_wait(group, yourtimeout);
```

Nested Contexts

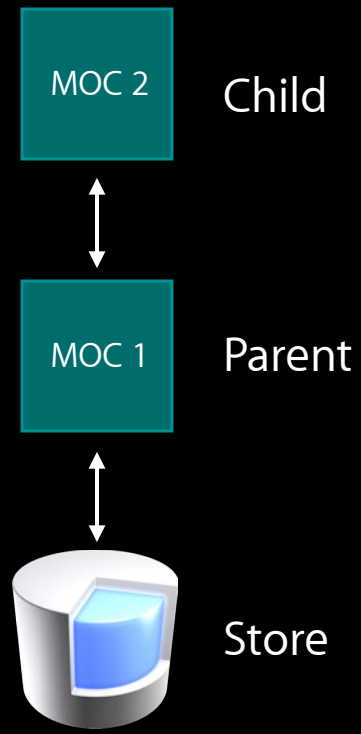
Challenges

- Sharing unsaved changes
- Asynchronous saving

Nested Contexts

- Parent context acts like a persistent store for the child
- Children see state as it is in the parent
- Children inherit unsaved changes from parent
- Children marshal their saves in memory to the parent

Nested Contexts



Why Use Nested Contexts?

- Sharing unsaved changes between MOCs
- Asynchronous saves
- Inheriting changes in a detail inspector

Sharing Unsaved Changes

- Push to parent context with save
- Pull in peer contexts
  - Fetching
  - Merging
  - Refreshing

Asynchronous Save

- Save child
- Asynchronously ask parent to save
- Changes not written to disk until root parent saves

Asynchronous Save

```
NSManagedObjectContext *child, *parent;
parent = [[NSManagedObjectContext alloc]
             initWithConcurrencyType:NSPrivateQueueConcurrencyType];
[child setParentContext:parent];
// ...
[child save:&error];
[parent performBlock:^{
   [parent save:&parentError];
}];
```

Inheriting Changes in Detail Inspector

- Create a child context
- Save pushes changes into parent
- Fetch incorporates unsaved changes in parent
- Toss child context to cancel detail changes

Things to Remember

- Saving only pushes changes up one level
- Fetching pulls data through all levels
- `-objectWithID:` pulls fewest levels necessary
- Parent contexts must adopt a queue type

Child MOCs Depend on Their Parents

- Parent context must not block upon children
- Children MOCs can `performAndWait` on parent
- Parents cannot `performAndWait` on children
- Requests flow up the tree of MOCs
- Results flow down from the parent

Performance

Session 214

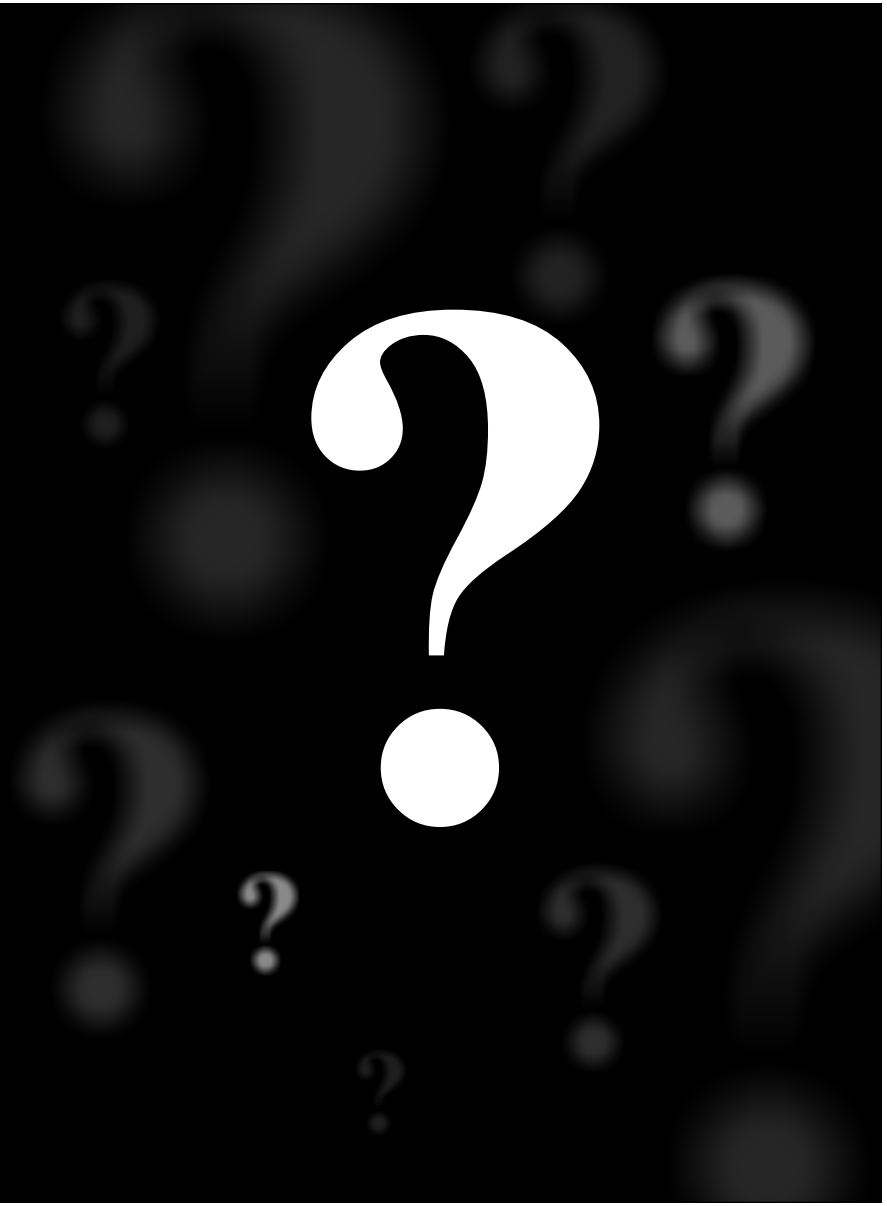
Melissa Turner
Sr Core Data Engineer

Recognizing problems

- Environment
- Application should do
- Application does do

Recognizing problems

- Environment
- Application should do
- Application does do



Environment

- Test on your minimal configuration
- Design for environment
  - Network
- Sufficient vs optimal

Should

User driven workflow

Should

User driven workflow

File
Access

Should

User driven workflow

File
Access

Network
Access

Should

User driven workflow

File
Access

Network
Access

Background
processing

Should

Automatic processing

Should

Automatic processing

Updates

Should

Automatic processing

Updates

Notifications

Should

Automatic processing

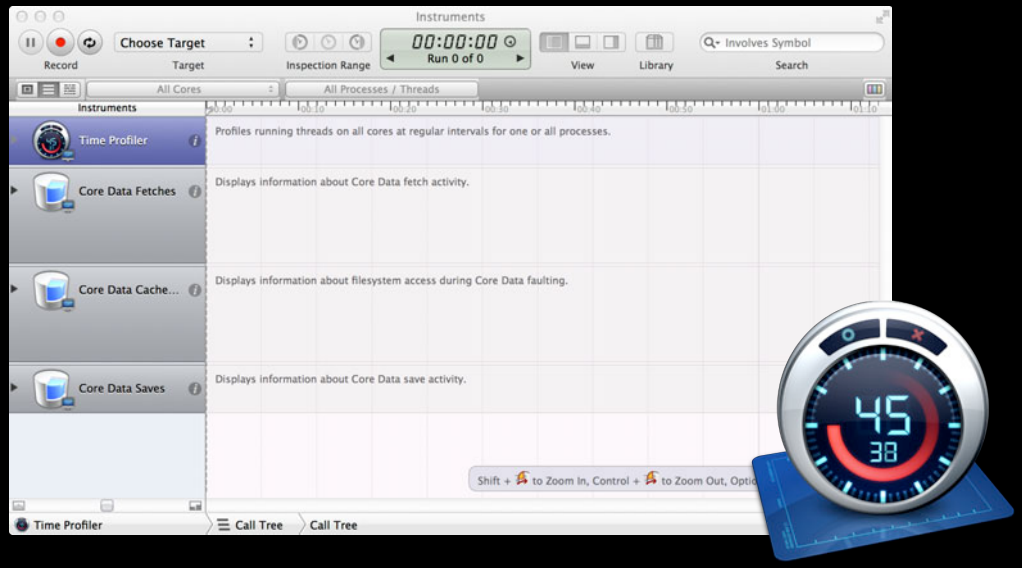
Updates

Notifications

Background Processing

Does

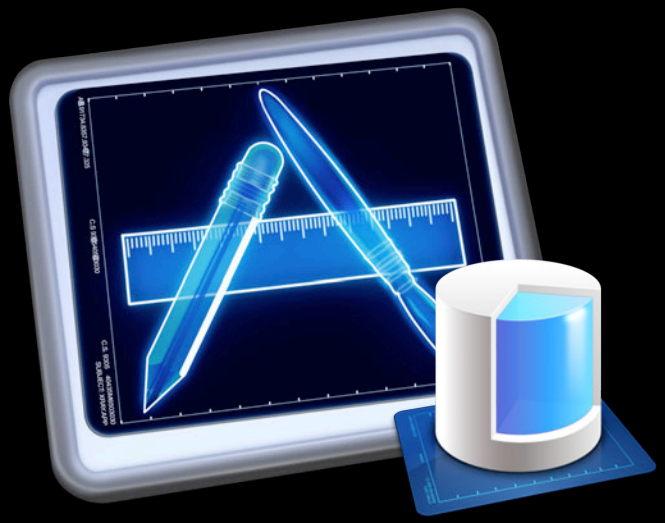
Measure, measure, measure



Time Profiler

Does

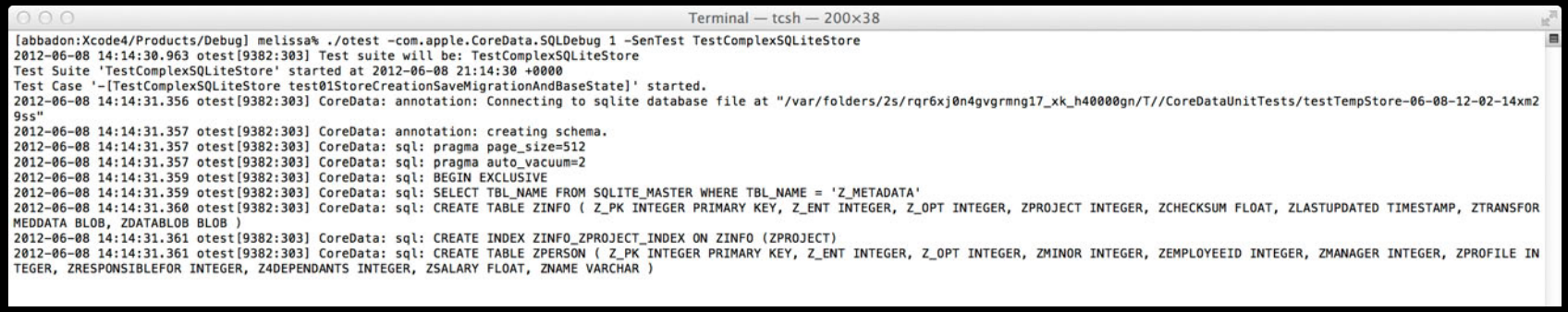
Measure, measure, measure



Core Data Template

Does

Measure, measure, measure



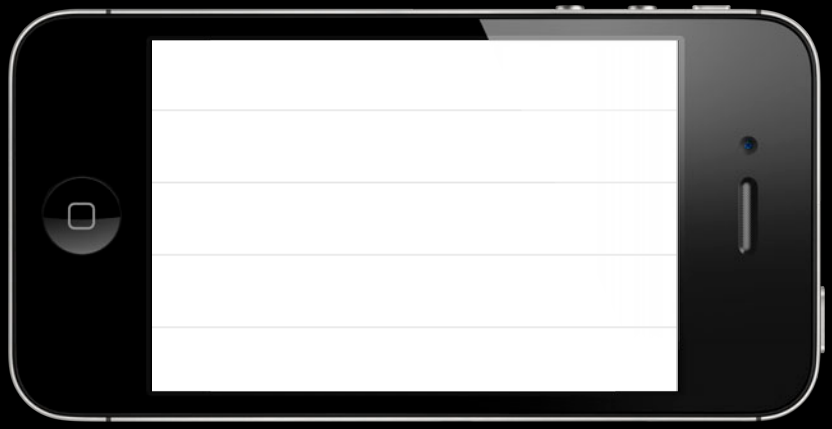
-com.apple.CoreData.SQLDebug 1 (or 3)

Measure twice.
Cut once.

The Target—Table Views

- Easy to visualize
  - Too much data
  - Too little data
  - Badly formed data
- Lessons are generally applicable
- Disclaimer

In the Beginning…



Schema Version 1

Schema Version 1

Rome vacation

Schema Version 1

Colosseum Architecture

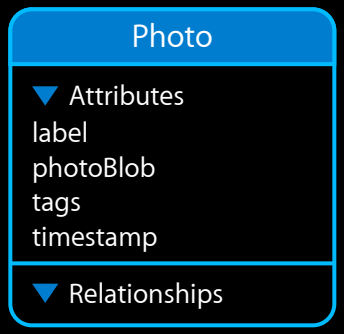
Schema Version 1

Rome vacation

Colosseum Architecture

Schema Version 1

Schema Version 1



Demo

Not fast, ergo furious

NSFetchRequest

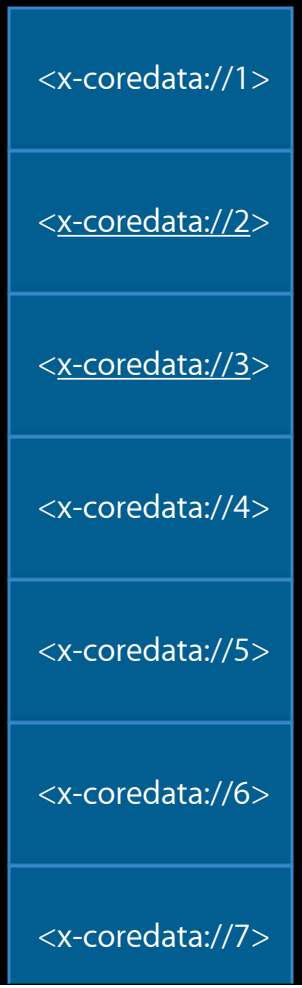
Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates



NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

NSFetchRequest

Ground zero for data optimization

- Batching
- Fetch limits and offsets
- Predicates
- Grouping
- Aggregates

Schema Design and Optimization

There is no one true schema

Designing Your Schema

Build a solid foundation

- Application concept will drive UI
- UI will drive schema
- No one true schema

Normalization

Eliminate data duplication

- Reduce possibility of skew
- Minimize storage space
- Minimize memory usage
- Faster searching

Normalization

Minimize storage space

Label	Timestamp	Data	Tags
Rome	2012/02/01	<abcde000 ...>	Family, Vacation
Hawaii	2012/02/01	<012409e0 ...>	Family, Work
Beijing	2012/02/12	<deadbeef ...>	Work

Normalization

Minimize storage space

Label	Timestamp	Data	Tags
Rome	2012/02/01	<abcde000 ...>	Family, Vacation
Hawaii	2012/02/01	<012409e0 ...>	Family, Work
Beijing	2012/02/12	<deadbeef ...>	Work

Normalization

Minimize storage space

Label	Timestamp	Data	Tags
Rome	2012/02/01	<abcde000 ...>	Family, Vacation
Hawaii	2012/02/01	<012409e0 ...>	Family, Work
Beijing	2012/02/12	<deadbeef ...>	Work

Normalization

Minimize storage space

Label	Timestamp	Data	Tags
Rome	2012/02/01	<abcde000 ...>	Family, Vacation
Hawaii	2012/02/01	<012409e0 ...>	Family, Work
Beijing	2012/02/12	<deadbeef ...>	Work

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Label
Family
Vacation
Work

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Label
Family
Vacation
Work

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Normalization

Minimize storage space

Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

Normalization

Minimize storage space

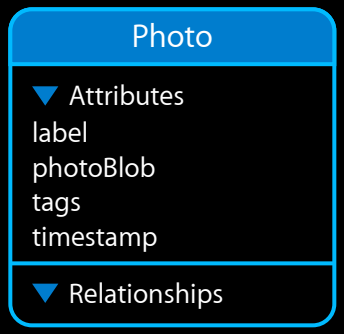
Label	Timestamp	Data
Rome	2012/02/01	<abcde000 ...>
Hawaii	2012/02/01	<012409e0 ...>
Beijing	2012/02/12	<deadbeef ...>

External Data References

- Move data out of store into co-located file
- Best stored on dedicated objects
- Refresh object after initial save

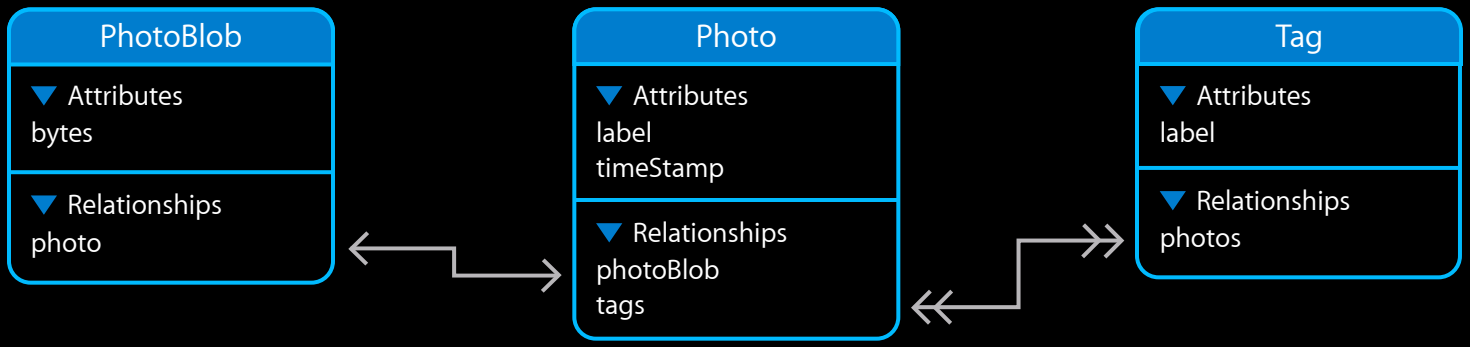
Schema Version 2

Breakdown



Schema Version 2

Breakdown



Demo

Getting closer

Denormalization

Reduce number of joins

- Minimize repeated transforms
- Minimize relationship fault firing
  - Store relationship meta-information on source
    - Existence
    - Count
    - Aggregate values

Normalization

Minimize relationship fault firing

Label	Timestamp	Label
Rome	2012/02/01	Family
Hawaii	2012/02/01	Vacation
Beijing	2012/02/12	Work

Normalization

Minimize relationship fault firing

Label	Timestamp	TagCount	Label
Rome	2012/02/01	2	Family
Hawaii	2012/02/01	2	Vacation
Beijing	2012/02/12	1	Work

Denormalization

Minimize repeated transforms

Label	Timestamp	TagCount	Data
Rome	2012/02/01	2	<abcde000 ...>
Hawaii	2012/02/01	2	<012409e0 ...>
Beijing	2012/02/12	1	<deadbeef ...>

Denormalization

Minimize repeated transforms

Label	Timestamp	TagCount	Thumbnail	Data
Rome	2012/02/01	2	<12480f0f ...>	<abcde000 ...>
Hawaii	2012/02/01	2	<cafebabe ...>	<012409e0 ...>
Beijing	2012/02/12	1	<66666600 ...>	<deadbeef ...>

Canonicalization

Minimize relationship fault firing

Label	Timestamp	TagCount	Thumbnail	Data
Rome	2012/02/01	2	<12480f0f ...>	<abcde000 ...>
Hawaii	2012/02/01	2	<cafebabe ...>	<012409e0 ...>
Beijing	2012/02/12	1	<66666600 ...>	<deadbeef ...>

Canonicalization

Minimize relationship fault firing

Label	Timestamp	TagCount	Thumbnail	Data
Rome	2012/02/01	2	<12480f0f ...>	<abcde000 ...>
Hawaii	2012/02/01	2	<cafebabe ...>	<012409e0 ...>
Beijing	2012/02/12	1	<66666600 ...>	<deadbeef ...>

Search Optimization

Before You Start the Fetch

- Initial view empty
- Search as you type vs on completion

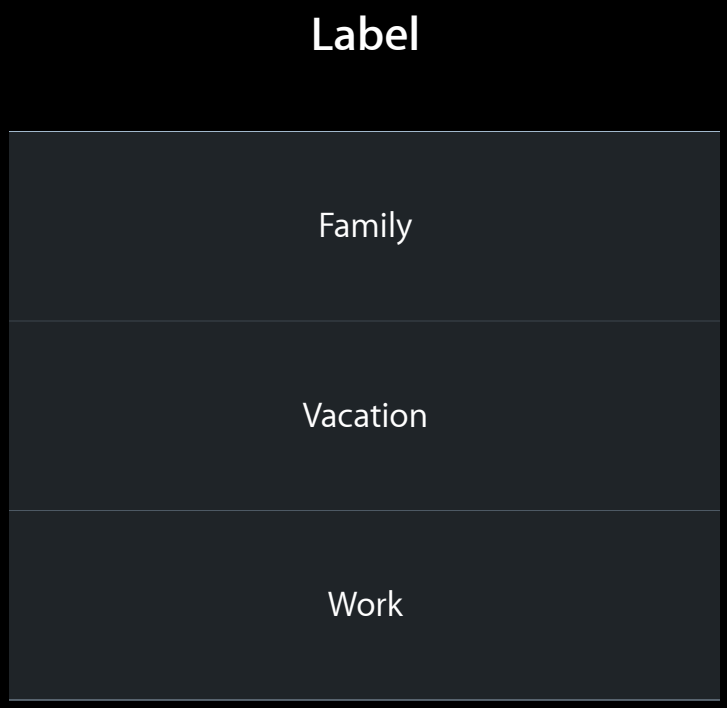
Searching

Strategies for making searches faster

- String canonicalization
- Fetch tokens
- Canonical objects

Canonicalization

Minimize CPU cycles



Canonicalization

Minimize CPU cycles

Label	SearchString
Family	family
Vacation	vacation
Work	work

Playing with Strings

Regex is not your user's friend

- Case and diacritic insensitivity
- beginswith vs contains
- Avoidance of wildcards

Predicate Optimization

```
label LIKE[cd] “Red*”
```

Predicate Optimization

```
searchString BEGINSWITH[n] “red”
```

Predicate Optimization

```
label MATCHES[cd] ".*Red.*"
```

Predicate Optimization

```
searchString CONTAINS “red”
```

Predicate Optimization

```
label MATCHES[cd] “.*Red.blue.*”
```

Predicate Optimization

```
searchString MATCHES[n] “.*red.blue.*”
```

Optimizing Your Predicate

Order matters

- No query optimizer in SQLite
- Eliminate largest groups first
- Group size vs comparison speed
- Put heaviest operations last

Predicate Optimization

```
searchString CONTAINS “red” OR timestamp BETWEEN (X, Y)
```

Predicate Optimization

```
timestamp BETWEEN (X, Y) OR searchString CONTAINS “red”
```

Predicate Optimization

```
ANY tag.searchString == “rome” AND label == “rome”
```

Predicate Optimization

```
ANY tag IN FETCH(%@, %@, NO) AND label == “red”
```

Query Optimization

Label	Timestamp	TagCount	Thumbnail
Rome	2012/02/01	2	<12480f0f ...>
Hawaii	2012/02/01	2	<cafebabe ...>
Beijing	2012/02/12	1	<66666600 ...>
Rome	2011/01/01	0	<87654320 ...>
San Francisco	2011/12/12	2	<42424240 ...>

Query Optimization

Label	Timestamp	TagCount	Thumbnail
Rome	2012/02/01	2	<12480f0f ...>
Hawaii	2012/02/01	2	<cafebabe ...>
Beijing	2012/02/12	1	<66666600 ...>
Rome	2011/01/01	0	<87654320 ...>
San Francisco	2011/12/12	2	<42424240 ...>

Query Optimization

Label	Timestamp	TagCount	Thumbnail
Rome	2012/02/01	2	<12480f0f ...>
Hawaii	2012/02/01	2	<cafebabe ...>
Beijing	2012/02/12	1	<66666600 ...>
Rome	2011/01/01	0	<87654320 ...>
San Francisco	2011/12/12	2	<42424240 ...>

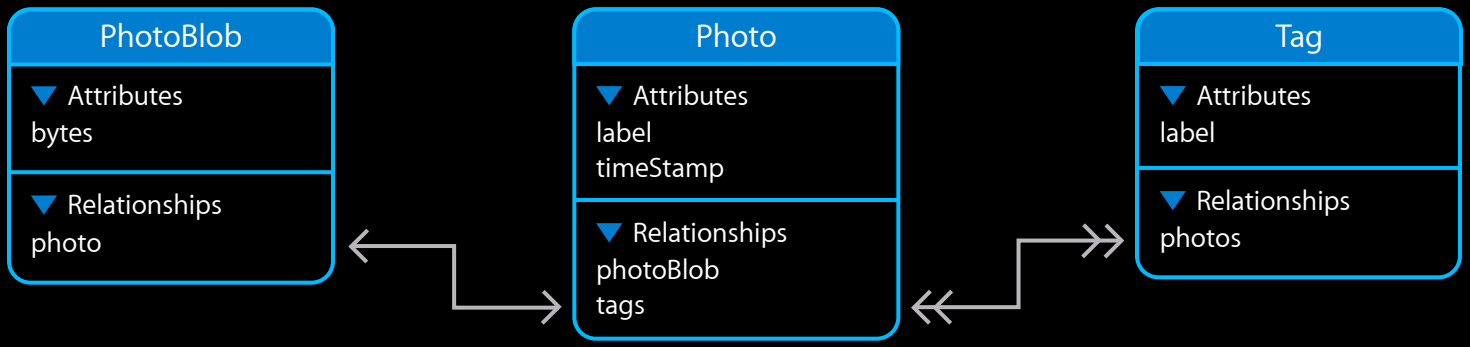
Query Optimization

Label	Timestamp	TagCount	Thumbnail
Rome	2012/02/01	2	<12480f0f ...>
Hawaii	2012/02/01	2	<cafebabe ...>
Beijing	2012/02/12	1	<66666600 ...>

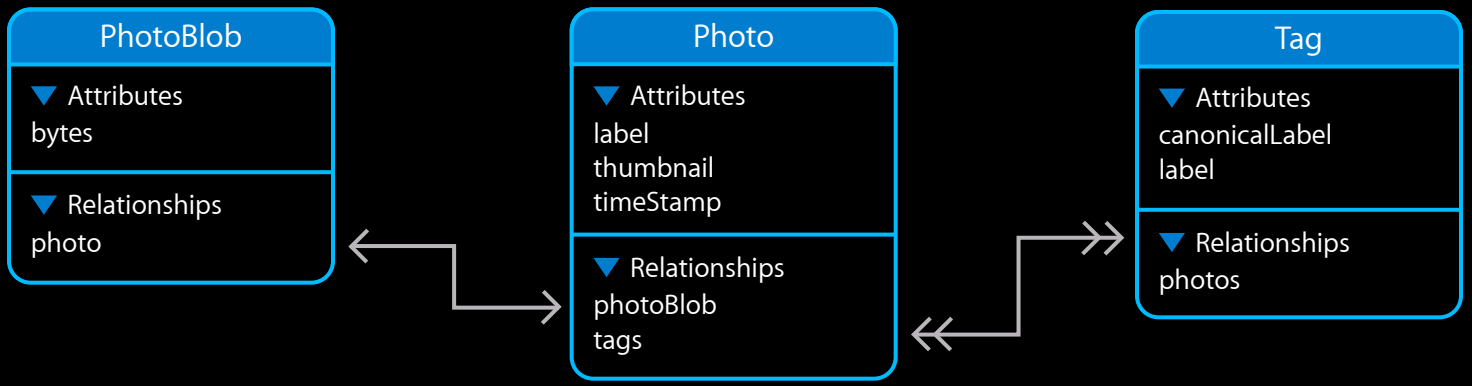
Query Optimization

Label	Timestamp	TagCount	Thumbnail
Rome	2012/02/01	2	<12480f0f ...>
Hawaii	2012/02/01	2	<cafebabe ...>
Beijing	2012/02/12	1	<66666600 ...>

Schema Version 3



Schema Version 3



Demo

Much better

Cleaning Up

Get rid of no-longer-interesting data

- Autorelease pools
- `-[NSManagedObjectContext refreshObject: mo mergeChanges: [mo hasChanges]]`
- `-[NSManagedObjectContext reset]`

Today's Roadmap

- Concurrency
- Nested Contexts
- Performance
- Schema Design
- Search Optimization

http://bugreport.apple.com

- We don't know unless you tell us
- Bugs fixed faster with
  - Steps to reproduce
  - Sample project
- Also use for
  - Feature requests
  - Enhancement requests
  - Performance issues
  - Documentation requests

More Information

Michael Jurewitz
Technology Evangelist
jury@apple.com

Cocoa Feedback
cocoa-feedback@apple.com

Core Data Documentation
Programming Guides, Examples, Tutorials
http://developer.apple.com/

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Using iCloud with Core Data	Mission Wednesday 4:30PM

Labs

Core Data Lab	Essentials Lab A Wednesday 2:00PM
Core Data Lab	Developer Tools Lab A Thursday 9:00AM
Core Data Lab	Essentials Lab B Friday 9:00AM


WWDC2012

The last 3 slides after the logo are intentionally left blank for all presentations.

The last 3 slides after the logo are intentionally left blank for all presentations.

The last 3 slides after the logo are intentionally left blank for all presentations.