Article

Accessing data when the store changes

Guarantee that a context won't see store changes until you tell it to look.

Overview

Query generations give your UI a stable view of data in the database, regardless of changes happening to the store underneath. Whenever you read from a context, you see the same generation, or snapshot, of data until you choose to advance it to a later generation.

Use query generations when you want to isolate your view context from any changes made in the store by background threads in your app, app extensions, CloudKit, or other sources.

Ensure the correct type and mode for the persistent store

To use query generations, the persistent store must be an `NSSQLiteStoreType` in write-ahead logging (WAL) journal mode. Core Data creates SQLite stores with WAL mode enabled by default.

Query generations leverage WAL mode to let you query against the historical state of the database. Core Data appends transactions to a `.sqlite-wal` file, or journal, in the same directory as the main store file. When your context reads from the journal, it starts at the transaction associated with a specific generation, instead of at the most recent transaction.

To confirm whether a custom store has WAL mode enabled, turn on SQL logging. Choose Product > Scheme > Edit Scheme, then choose the Run action, and add the following line under Arguments Passed on Launch:

```
-com.apple.CoreData.SQLDebug 1
```

Run your app, and look for the following output in the console:

```
CoreData: sql: pragma journal_mode=wal
```

If you try to use query generations with a store that's not an `NSSQLiteStoreType` in WAL journal mode, your contexts gracefully revert to unpinned behavior.

Pin the context to a store generation

By default, contexts are unpinned, and read from the store at the generation of the most recent transaction. Pinned contexts read from the store at the generation of a specific transaction.

To pin a context, call `setQueryGenerationFrom(_:)` and pass an opaque `NSQueryGenerationToken`. The context updates to the specified generation lazily on the next read (fetching or faulting) operation.

Use the `current` generation token to pin the context to the generation corresponding to the most recent store transaction. For example, pass the `current` generation token when setting up your stack to pin the view context to the first generation that it fetches.

```
// Pin the context to the generation that corresponds with the most recent
// store transaction.
do {
    try persistentContainer.viewContext.setQueryGenerationFrom(.current)
} catch {
    // Handle the error appropriately.
    print("Failed to pin the context:", error.localizedDescription)
}
```

Alternatively, use the `queryGenerationToken` from another pinned context to align both contexts to the same generation.

To unpin a context, call `setQueryGenerationFrom(_:)`, passing `nil`.

```
// Unpin the context.
do {
    try persistentContainer.viewContext.setQueryGenerationFrom(nil)
} catch {
    // Handle the error appropriately.
    print("Failed to unpin the context:", error.localizedDescription)
}
```

Nested contexts inherit their parent's generation. They're implicitly unpinned, but they see data as viewed through the generation of their parent with the addition of their parent's pending changes.

A generation doesn't include stores added to the store coordinator after the generation's creation. Additionally, if you remove a store from the coordinator, don't try to load data from the deleted store into a context.

Update the view context to the current store generation

Advance a context to the generation of the most recent transaction, and pin it there, by calling `setQueryGenerationFrom(_:)` and passing the `current` token. The context updates to the specified generation lazily on the next read (fetching or faulting) operation.

```
// Advance the context to the generation of the most recent store transaction.
do {
    try persistentContainer.viewContext.setQueryGenerationFrom(.current)
} catch {
    // Handle the error appropriately.
    print("Failed to set the query generation:", error.localizedDescription)
}
```

Alternatively, update a context's generation by calling any of the following.

- `setQueryGenerationFrom(_:)`
- `save()`
- `mergeChanges(fromContextDidSave:)`
- `mergeChanges(fromRemoteContextSave:into:)`
- `reset()`

Update contexts to the `current` generation as soon as a specific generation is no longer needed. Query generations hold a file lock open to maintain the integrity of the journal for the duration of a query generation. Once no contexts refer to a query generation, it expires, and the system can reclaim the journal disk space.

Refresh objects

Refresh any managed objects registered to the context after you change the context's query generation or unpin the context. Managed objects don't automatically refresh, as this behavior may not be desirable and is difficult to revert.

Call `refreshAllObjects()` on the context to refresh its existing managed objects.

```
// Refresh existing managed objects.
persistentContainer.viewContext.refreshAllObjects()
```

You can also refresh your objects by fetching them again. Call `fetch(_:)` on the context to retrieve a fresh set of managed objects matching your request criteria.

```
// Alternatively, refresh objects by fetching them again.
let request = NSFetchRequest<ShoppingItem>(entityName: "ShoppingItem")
request.fetchBatchSize = 10

// Execute the fetch.
let results = await persistentContainer.viewContext.perform {
    do {
        return try self.persistentContainer.viewContext.fetch(request)
    } catch {
        // Handle the error appropriately. It's useful to use
        // `fatalError(_:file:line:)` during development.
        fatalError("Failed to refresh the objects: \(error.localizedDescription)")
    }
}
```

The fetch reads the journal from the context's query generation if pinned, or from the most recent transaction if unpinned.

See Also

Change processing

Consuming relevant store changes

Filter store transactions for changes relevant to the current view.

Persistent history

Use persistent history tracking to determine what changes have occurred in the store since the enabling of persistent history tracking.