10

Fetching and Saving Data

This chapter examines the methods and techniques you use to move data between the server and your application. It's divided into two major sections:

- *Fetching*. The data-fetching demonstration given in Chapter 8 was far from the whole story. The Database Kit lets you adjust the fetching machinery, as explained in the section ªDetails of Fetching.º
- *Saving and validating data.* One of the thornier problems confronting a database application developer is to ensure that the data that's drawn into the application and modified there will ªfitº when it's sent back to the server. The checks through which a client of a server, such as your application, attempts to ensure that its data will fit is called ªvalidationº or ªverificationº (this book adopts the former term). Validation is a highly permutable activity, depending greatly on the nature of the server, the resources of the adaptor, and the type of data that your application uses. Thus, a comprehensive and universally-applicable ªvalidation suiteº is impossible to create. Nonetheless, the Database Kit provides a few validation techniques that any application should be able to use. These are described in the section ªSaving and Validating Data.º

Details of Fetching

There are a number of ways that you can tinker with the fetching machinery. You can:

- Perform the fetch in a background thread.
- Limit the number of records that are retrieved during a single fetch.
- Set the order of the records that are retrieved.
- Concatenate the results of consecutive fetches.
- Fetch a specific record based on its primary key value.

These manipulations are examined in the following sections. But first, the DBRecordList's ªretrieve status,º upon which some of these tinkerings depends, is described.

Retrieve Status

A DBRecordList maintains a ªretrieve status,º a constant that tells you where the DBRecordList is with regard to a ªfetching session.º There are five retrieve states, returned through the **currentRetrieveStatus** method as one of these DBRecordRetrieveStatus constants:

- **DB_NotReady**. This is the status of a freshly made DBRecordList; it indicates that the object doesn't contain any properties. If you try to fetch while the DBRecordList isn't ready, the fetch will fail.
- **DB_Ready**. This status indicates that the DBRecordList is ready for a fetch. More

specifically, it means that the object has had its properties set, but hasn't fetched any records, whether because it's been recently created or has received an **empty** or **clear** message.

- **DB_FetchInProgress**. You can only catch a DBRecordList in the act of fetching if the object is set to fetch in the background; this status indicates that a background fetch has commenced (whether its actually in progress or has completed can't be determined).
- **DB_FetchLimitReached**. This is used to indicate that a previous fetch operation was completed because the record limit was reached.
- **DB_FetchCompleted**. This status means that the previous fetch operation is finished, and that the entire range of records, given the latest qualification, have been retrieved from the server.

The normal sequence of states for a DBRecordList depends on how the object's fetching mechanism is set up. For a default fetch, the states proceed as shown in the following ªcausalº table:

Invoking this method...	**...induces this status**
init, **clear**, or **empty**	DB_NotReady
setProperties:ofSource:	DB_Ready
fetchUsingQualifier:	DB_FetchInProgress (background fetches only) DB_FetchLimitReached (record-limited fetches only) DB_FetchCompleted (foreground fetches only)

Fetching in the Background

By default, fetches are performed in the foreground: The fetch method doesn't return until it's finished fetching. You can also set a DBRecordList to fetch in the background, thus allowing the fetch method to return immediately while the fetch continues asynchronously in its own process thread.

To set the ªfetch strategyº of a DBRecordList, you pass one of the DBRecordListRetrieveMode constants to the **setRetrieveMode:** method:

- **DB_SynchronousStrategy**. This is the default foreground strategy.
- **DB_BackgroundStrategy**. Records are fetched in the background, as explained above, allowing your application to proceed. However, if you try to move the DBRecordList's cursor to a record that hasn't yet been fetched, the cursor-positioning method will block until the specified record is retrieved. (Note that sending a **setLast** message to a background-fetching DBRecordList will block until all the records have been fetched and the cursor can be correctly set to the absolute last record in the record table.)
- **DB_BackgroundNoBlockingStrategy**. This is the same as DB_BackgroundStrategy, except that it doesn't cause the cursor-moving methods to block; if you try to point the cursor to an as-yet-unretrieved record, the method will fail and immediately return.

The **retrieveMode** method returns a DBRecordList's current fetch strategy as one of these constants.

Warning: You should only set a DBRecordList to fetch in the background if you're using the Application Kit and you have its main event loop running.

Checking for the Completion of a Background Fetch

After you've started a background fetch, you'll probably want to know when the fetch is complete. To be so informed, you need to supply a ªbinderº delegate that implements the **binderDidFetch:**

method. The binder delegate, and the messages it receives, is described in greater detail later in this chapter. The following example provides a quick tutorial on how to prepare your application to catch the **binderDidFetch:** message. First, you create a delegate class:

```
/* Declare the interface for a class that implements the
   binderDidFetch: method. */
@interface MyBinderDelegate : Object
{}
- binderDidFetch:aBinder;
@end;

/*    Define the class. */
@implementation MyBinderDelegate
- binderDidFetch:aBinder
{
   /* Do what you need to do here. The return value is ignored, but
      it isn't declared void, so, to be polite, we return self. */
      return self;
}
```

Then you set the DBRecordList's binder delegate to an instance of your class:

```
MyBinderDelegate *aDelegate= [[MyBinderDelegate alloc] init];
DBRecordList *aRecordList = [[DBRecordList alloc] init];

[aRecordList setBinderDelegate:aDelegate];
```

When the fetch is complete, the binder delegate is sent the **binderDidFetch:** message. (The message is sent regardless of the DBRecordList's retrieval strategy; however, it's of particular importance when the object is fetching in the background.)

Background Fetches and Retrieval Status

After a background fetch has begun, the DBRecordList's retrieval status is set to DB_FetchInProgress. It's important to note that the status doesn't change when the fetch is completeÐit remains DB_FetchInProgress until the DBRecordList is cleared, emptied, or has its properties reset. Because of this, you can't use the status to gauge if a background fetch has finished.

Stopping a Background Fetch

If you get tired of waiting for a background fetch to complete, you can tell it to stop by sending the **cancelFetch** message to the DBRecordList. This doesn't wipe out the fetch entirelyÐrecords that have already been fetched and placed in the DBRecordList aren't thrown awayÐit simply stops the fetch process. As with a naturally-completed background fetch, cancelling a fetch doesn't cause the DBRecordList's status to changeÐit remains DB_FetchInProgress.

Cancelling a background fetch thwarts the **binderDidFetch:** delegate message.

Record Limit

When a DBRecordList is commanded to fetch, it normally fetches the entire range of records that pass the qualification. You can set the maximum number of records that you want a fetch to retrieve through the **setRecordLimit:** method. The **currentRecordLimit** method returns the currently set record limit; a return of DB_NoIndex indicates that there is no limit (this is the default). You can use the DB_NoIndex value as the argument to **setRecordLimit:** to erase the current limit.

Important: For the record limit to have an affect, the DBRecordList's fetching strategy must be DB_SynchronousStrategy: You can't set the record limit of a DBRecordList that fetches in the background.

There are two ways to tell if the record limit was reached during a fetch (these are examined in the following sections):

- The DBRecordList's status is set to DB_FetchLimitReached.
- A **recordStream:willFailForReason:** message is sent to the DBRecordList's delegate, with DB_RecordLimitReached as the second argument.

Fetch Limit Reached Status

The DB_FetchLimitReached status is essentially the same as DB_FetchCompleted. You don't have to do anything to ªresetº the object or otherwise clean up after the fetch. The status is provided merely to inform you that the fetch didn't (necessarily) retrieve all the records that pass the qualifier.

More important, this status doesn't affect a subsequent fetch: The next fetch *doesn't* continue where the previous one left off, it begins fetching at the ªtopº of the server's list of records. In other words, you can't use the record limit feature to perform incremental fetches.

If the record limit is greater than the number of candidate records in the server, the DBRecordList's status, after a fetch, is set to DB_FetchCompleted. (If the two values are equal, the status is DB_FetchLimitReached, even though all the records were fetched.)

The Delegate Message

When, during a fetch, a DBRecordList's record limit is reached, the object's delegate is sent a **recordStream:willFailForReason:** message, with the DBRecordList object as the first argument and the value DB_RecordLimitReached (a DBFailureCode constant) as the second. The delegate's response to this boolean message is important:

- A response of YES tells the DBRecordList to obey the record limit; the fetch is halted, the object's status set to DB_FetchLimitReached, and the fetch method returns immediately.
- A response of NO tells the DBRecordList to ignore the limit and continue fetching. If no other errors are encountered, the object's status will be set to DB_FetchCompleted when the fetch method returns.

An implementation of **recordStream:willFailForReason:** can assume that the DBRecordList contains (at least) the number of records specified by the limit (if the DBRecordList didn't empty before fetching, then it may contain more). The values held in these records may be examined to determine whether the method should return YES or NO.

Sorting Records

DBRecordList provides methods that let you declare the basis upon which fetched records are sorted into the record table. This is done by associating a ªretrieve orderº constant with a particular property in the DBRecordList object. There are three retrieve order constants, declared as DBRetrieveOrder data types; their names describe their meanings:

DB_NoOrder
DB_AscendingOrder
DB_DescendingOrder

The association between a retrieve order and a property is established through the **addRetrieveOrder:for:** method:

```
/* We'll assume that the objects used below exist; "employees" is a
   DBRecordList; "name" is a property object rooted at "employees"
   source entity. */
[employees addRetrieveOrder:DB_AscendingOrder for:name];
```

When the DBRecordList fetches, in this example, the records are sorted into alphabetical order based on the values for the name property. Similarly, the following causes the records to be stored in reverse alphabetical order:

```
[employees addRetrieveOrder:DB_DescendingOrder for:name]
```

The property argument (the argument to the **for:** keyword) can be a natural property or DBExpression object. Furthermore, the object needn't be present in the DBRecordList's property list. The only requirement is that it be rooted at the DBRecordList's source entity.

Secondary Sorting

The example shown above raises a questionÐwhat happens if two records have the same value for the name property? You can provide a tie-breaker simply by specifying the retrieve order for yet another property. The order in which retrieve-ordered properties are applied during a fetch is the order in which the **addRetrieveOrder:for:** messages that declared them were sent.

For example, consider a slightly different Employee entity that has the properties firstName and lastName. To sort employee records such that they're sorted alphabetically by last name and then, in the case of identical last names, by first names, you would send these messages in this order:

```
/* The firstName and lastName object are assumed to exist. */
[employees addRetrieveOrder:DB_AscendingOrder for:lastName];
[employees addRetrieveOrder:DB_AscendingOrder for:firstName];
```

You can add any number of properties to the retrieve order equation.

Concatenating Fetches

By default, a DBRecordList empties itself before it fetches dataÐthis means that any previously fetched data that's stored in the object is thrown away and replaced with the results of the new fetch. You can tell a DBRecordList to retain ªoldº data by passing NO as the second argument to the **fetchUsingQualifier:empty:** method. This causes the results of the new fetch to be appended to the records that are already in the DBRecordList (the new records aren't sorted among the old records).

Fetching without emptying is useful if you're performing successive fetches with a different qualification at each fetch. For example, let's say you've set up a DBQualifier with which you've fetched employee records for employees whose last names begin with ªAº. Now you want to retrieve the ªBº employees but you don't want to throw away the ªAº set. To do this, you modify the DBQualifier to match last names that start with ªBº and apply the object in a **fetchUsingQualifier:empty:** message, passing NO as the second argument. By repeating this across the alphabet, you can perform an incremental fetch.

You can't use the fetch-no-empty feature to fetch dissimilar sets of data (in other words, records with different sets of properties) into the same DBRecordList. This is because the **setProperties:ofSource:** method automatically empties the receiving DBRecordList.

Saving and Validating Data

Writing modified data back to the server is simple: You invoke DBRecordList's **saveModifications** method. The method steps through the record table and identifies those records that contain modified data. As it checks for modified data, the **saveModifications** method also performs an ªintegrityº test that warns you if the data that you're saving will overwrite someone else's changes.

These two routinesÐfinding modified records and ensuring the integrity of the data these records containÐconstitute the Database Kit's validation process; the two parts of this process are described in detail in the following sections.

You should note that the Kit does nothing, at the time data is saved, to check data value ªcorrectnessº or data format consistency. In other words, it doesn't look at each value to make sure that (for example) it falls within certain bounds, or that the value's data type is in the proper format. However, these subjects aren't completely unaddressed:

- Checking the value for an attribute is highly dependent on contextÐonly your application can determine whether a value is appropriate. The methods described in the next section show you how to identify modified values so you can perform value checks yourself.
- As described in Chapter 8, the data type of a field in a records is immutable. The value in a particular field is converted to the data type declared by the attribute for which the field holds data; it maintains that type regardless of value modifications. Thus, data format is, for most applications, excused from the validation arena.

Finding Modified Data

There are three boolean DBRecordList methods that help you find modified data:

- **isModified** tells you if the object's record table has been modified such that it might not correspond to data on the server. Specifically, the method returns YES if records have been added or deleted from the table, or if the value of a field within a record has been set (even if the new value is the same as the old). The order of the records in the table isn't considered; moving and swapping records in the table won't, of themselves, mark the table as having been modified.
- **isModifiedAt:** takes a record index argument and returns YES if the record is new or if the value of any of the record's fields has been set (again, even if the new value is the same as the old).
- **isModifiedForProperty:at:** looks at the specific record field designated by the property object and record index arguments and returns YES if its value has been set (or if the record is new).

You can use these methods to find the fields that have been modified and so, for example, check their values before sending them back to the server.

The Modified Record List

When you tell a DBRecordList to save its modifications, the object compiles a list of records (its ªmodified record listº) that are involved in the operation. This list consists of new or modified records and deleted records. Only these records are sent back to (or deleted from) the server. There are two reasons why records are selected for modification (as opposed to blithely sending the entire record table):

- It's more efficient to send back only those records that have been modified or deleted.
- The data integrity test (which is described in the next section) won't stumble over data that you're

not interested in.

Although you can easily determine the new or data-modified records that will be included (by using the methods described in the previous section), there's no way to ask for the object's deleted records.

The Integrity Test

The Database Kit performs a ªdata integrityº test to make sure that the data in the records that you're saving won't overwrite someone else's changes. The manner in which the test is conducted depends on the adaptor that you're using; the tests for the Oracle and Sybase adaptors are examined in the following sections. You should be aware that the integrity test is only run on data-modified or deleted records. New records (records that have been added to the DBRecordList) pass the test by default.

The Oracle Test

The Oracle integrity test examines each modified record and determines if any field in that record has changed on the server since the record was fetched. If a field has been changed, the record won't be used to update the server's data.

For example, let's say you've fetched some employee records into a record table that, after the fetch, looks like this:

Name	Location	Salary
Runyon	New York	10000
Smith	Atlanta	5000
Jones	Boston	7000

The Database Kit makes a copy of these just-fetched records for later use. Changes that you make to the records (in your record table) won't affect the Kit's copy.

In your record table, you change Runyon's location to San Diego, and Smith's location to Des Moines and his salary to $2,000:

Name	Location	Salary
Runyon	San Diego	10000
Smith	Des Moines	2000
Jones	Boston	7000

In the meantime, someone else fetches the same data and changes Smith's location to Omaha (but leaves all other fields unchanged):

Name	Location	Salary
Runyon	New York	10000
Smith	Omaha	5000
Jones	Boston	7000

Let's assume that the other user sends **saveModifications** before you do. After the save (in which only the Smith record is sent to the server since it was the only one that was modified) the data on the server looks like the table shown immediately above.

Now you send **saveModifications**. The Database Kit looks at each record and determines that only Runyon's and Smith's need to be saved. Next, the Kit re-fetches these records from the server and compares them to its copy of the just-fetched records. If there are any differences, it will refuse to save the record. Your Runyon record is accepted. However, since Smith's location was changed by the other user, your Smith record is rejected. After the save, the server's data looks like this (the

server data is shown in bold):

Name	Location	Salary
Runyon	San Diego	10000
Smith	Omaha	5000
Jones	Boston	7000

You should note the finer points of the Oracle integrity test mechanism:

- Data is accepted or rejected on a record-by-record basis. Even though your Smith record didn't pass the integrity test, Runyon did. Thus, Smith was rejected but Runyon was accepted and written to the server. Furthermore, the *entire* Smith record was rejected; even though your salary modification was valid (in that the other user didn't set the field to a new value), your new salary value was rejected along with the rest of the record.
- After the save, your record table will still hold modified data for Smith: According to your record table Smith is in Des Moines making $2,000. Even though the Database Kit knows that your record table is out of sych with the server, it won't alter your table to hold the server's data.

The Sybase Test

The Sybase integrity test proceeds much like that of the Oracle test: A just-fetched copy of your records is used to determine if the records have changed since the time that they were fetched. Unlike Oracle, the Sybase mechanism doesn't automatically reject a record that shows a discrepancy between the just-fetched copy and the current (server) state of the record. By default, all modifications are accepted, even if they overwrite someone else's changes.

As an example of how this works, let's re-visit the demonstration used above. Here we have the employee table before anybody fetches from it:

Name	Location	Salary
Runyon	New York	10000
Smith	Atlanta	5000
Jones	Boston	7000

You fetch the data into a DBRecordList, and modify the Runyon and Smith records as described above:

Name	Location	Salary
Runyon	San Diego	10000
Smith	Des Moines	2000
Jones	Boston	7000

The other user also fetches and modifies the Smith record:

Name	Location	Salary
Runyon	New York	10000
Smith	Omaha	5000
Jones	Boston	7000

The other user sends **saveModifications** before you do; the server now holds data as shown in the table immediately above. When you send **saveModifications**, the Smith record doesn't pass the integrity test, but it doesn't matterÐyour record is still saved, clobbering that of the other user. Thus, after you save the server's Smith record looks like yours:

Name	Location	Salary
Smith	Des Moines	2000

When the integrity test finds a questionable record, it sends the DBRecordList's delegate a

recordStream:willFailForReason: message. The DBRecordList's delegate can implement this method to thwart the impending overwrite. This is explained in the next section.

There's another way in which the save-mechanism based on the Sybase adaptor differs from Oracle's: Whereas Oracle accepts or rejects whole records, Sybase will accept changes to individual fields. Turning, once again, to the example, consider the case in which you and the other user modify different fields in the same record. Let's say you change Smith's location to Des Moines, but you don't change his salary:

Name	Location	Salary
Smith	Des Moines	5000

And the other user's changes his salary to $4,000, but leaves him in Atlanta:

Name	Location	Salary
Smith	Atlanta	4000

The other user saves, writing the new salary value to the server. Then you save; since location was the only attribute you changed, only the value of that field is written to the server. The server's Smith record, after both changes, looks like this:

Name	Location	Salary
Smith	Des Moines	4000

Even though your modification doesn't overwrite the other user's, your Smith record, as it's being processed by the save-mechanism, will fail the integrity test. Thus, you have a chance, through the **recordStream:willFailForReason:** delegate method, to reject your own non-destructive change.

Controlling a Save

The **saveModifications** method sends the DBRecordList's delegate a **recordStream:willFailForReason:** message if there's a problem saving the data. The message's first argument is the DBRecordList object; the second is a DBFailureCode constant that describes the failing situation.

The first two failure codes indicate an error in the set up of the DBRecordList:

- **DB_RecordStreamNotReady**. This indicates that the DBRecordList isn't ready to save, probably because it hasn't had its property list set.
- **DB_NoRecordKey**. This is similar to the above, although it's a bit more precise: It means that the DBRecordList doesn't have a primary key attribute. Given the automatic primary key-searching mechanism described in Chapter 8, this failure code should rarely be seen. If you do see it, you should suspect the model that you're using (it probably doesn't have a primary key designation).

For both of these errors, the value that's returned by the **recordStream:willFailForReason:** method is ignored and the **saveModifications** method returns immediately, returning the value DB_NoIndex.

The other error codes that can be passed as the second argument in a **recordStream:willFailForReason:** message are listed below. These codes indicate a ªrecord error.º In other words, a particular record either didn't pass the integrity test or was otherwise unable to be saved:

- **DB_RecordKeyNotUnique**. This means that the fetch that was performed by the integrity test found more than one record (on the server) with primary key values that match the primary key value of the current record (where ªcurrent recordº means the record in the DBRecordList's list of modified records that's currently being tested or that the DBRecordList is attempting to write to the server).

- **DB_TransactionError**. This error indicates that the current record failed the integrity test because the analogous record on the server couldn't be found, possibly because some other process deleted it.

- **DB_RecordHasChanged**. This means that the current record failed the integrity test because of changed data. In other words, the just-fetch copy of the record didn't match the server's current version.

- **DB_AdaptorError**. This indicates that the adaptor encountered an error while writing the current record to (or deleting it from) the server.

Because these codes point to errors in particular records, you may see more than one of them within the course of a single **saveModifications** invocation.

For each of the four record errors (but not for the DBRecordList errors listed previously) the boolean value returned by **recordStream:willFailForReason:** is important. It answers the question ªshould this save be aborted?º:

- If **recordStream:willFailForReason:** returns YES, the entire transaction is aborted. No more records are looked at, and any previous records that were successfully saved are ªunsavedº (or *rolled back*). The **saveModifications** method returns immediately, returning the value DB_NoIndex.

- If it returns NO, the save process isn't aborted and previously saved records aren't rolled back. What happens to the current record (the record that caused the error) depends on the adaptor. The Oracle adaptor skips the recordÐit makes no attempt to send it to or delete it from the server. The Sybase adaptor goes ahead and processes the record, possibly clobbering someone else's changes, as described in the previous section. If the save isn't aborted by a subsequently rejected record, the **saveModifications** method will, in this case, return 1. This indicates that the record table holds modified but unsaved data.

As a review, the following table describes the meanings of the values that can be returned by **saveModifications**:

Return Value	Meaning
DB_NoIndex	The save was aborted (or never got to the first record).
1	The DBRecordList may contain some unsaved records.
0	The save was completely successful.

Implementing the Delegate Method

When **recordStream:willFailForReason:** is invoked, the DBRecordList is guaranteed to be ªpointing toº the record that caused the failure. This is true even if the record is being deleted and so doesn't otherwise appear in the DBRecordList. The following methods, which query the pointed to record, can be used within a delegate's implementation of the **recordStream:willFailForReason:** method:

- getValue:forProperty:
- getRecordKeyValue:
- currentPosition

Warning: These ªrecord cursorº methods, which were proscribed in Chapter 8, should only be used in the implementation of the **recordStream:willFailForReason:** method. They should otherwise be avoided.

The **currentPosition** method is particularly useful, for you can use the value it returns (the record index of the failed record) to determine which properties were modified. Specifically, you can use the current record index value as the last argument to the **isModifiedForProperty:at:**

method.

For example, let's say you've set up a record table that holds employee records, as in the previous section. Of the properties that you've set (name, location, and salary), you want to make sure that the values for location maintain absolute integrityÐyou don't care so much about names and money, but if a record is rejected because of a bad location value, then you want the entire save to be aborted. Here's what the implementation of **recordStream:willFailForReason:** would look like:

```
- (BOOL) recordStream:streamOrList
          willFailforReason:(DBFailureCode)aCode
{
   switch (aCode) {
      case DB_RecordKeyNotUnique:
      case DB_TransactionError:
      case DB_RecordHasChanged:
      case DB_AdaptorError:
          /* We'll assume that locProp, which represents the
             "locations" property, is declared globally. */
          if ([streamOrList isModifiedForProperty:locProp
                        at:[streamOrList currentPosition]])
              return YES;
          else
              return NO;
      default:
          return YES;
   }
}
```

Warning: You shouldn't try to fix the error or in any other way modify the record in the implementation of this method. By the time the method is invoked, it's too late for the failed record.

Warning: If you're using the Sybase adaptor provided by the Database Kit, you also must not fetch during the **recordStream:willFailForReason:** method.

Keep in mind that **recordStream:willFailForReason:** is also invoked when a fetch has reached a declared record limit. This was described in the section on fetching, earlier in this chapter. As a convenience, the complete set of failure codes that can appear as the second argument to **recordStream:willFailForReason:** are shown below:

Error Code	Meaning	Return Value
DB_RecordStreamNotReady	Bad DBRecordList	Ignored
DB_NoRecordKey	Bad DBRecordList	Ignored
DB_RecordKeyNotUnique	Bad record	Abort? (YES or NO)
DB_TransactionError	Bad record	Abort?
DB_RecordHasChanged	Bad record	Abort?
DB_AdaptorError	Bad record	Abort?
DB_RecordLimitReached	Record limit met during fetch	Continue fetch?