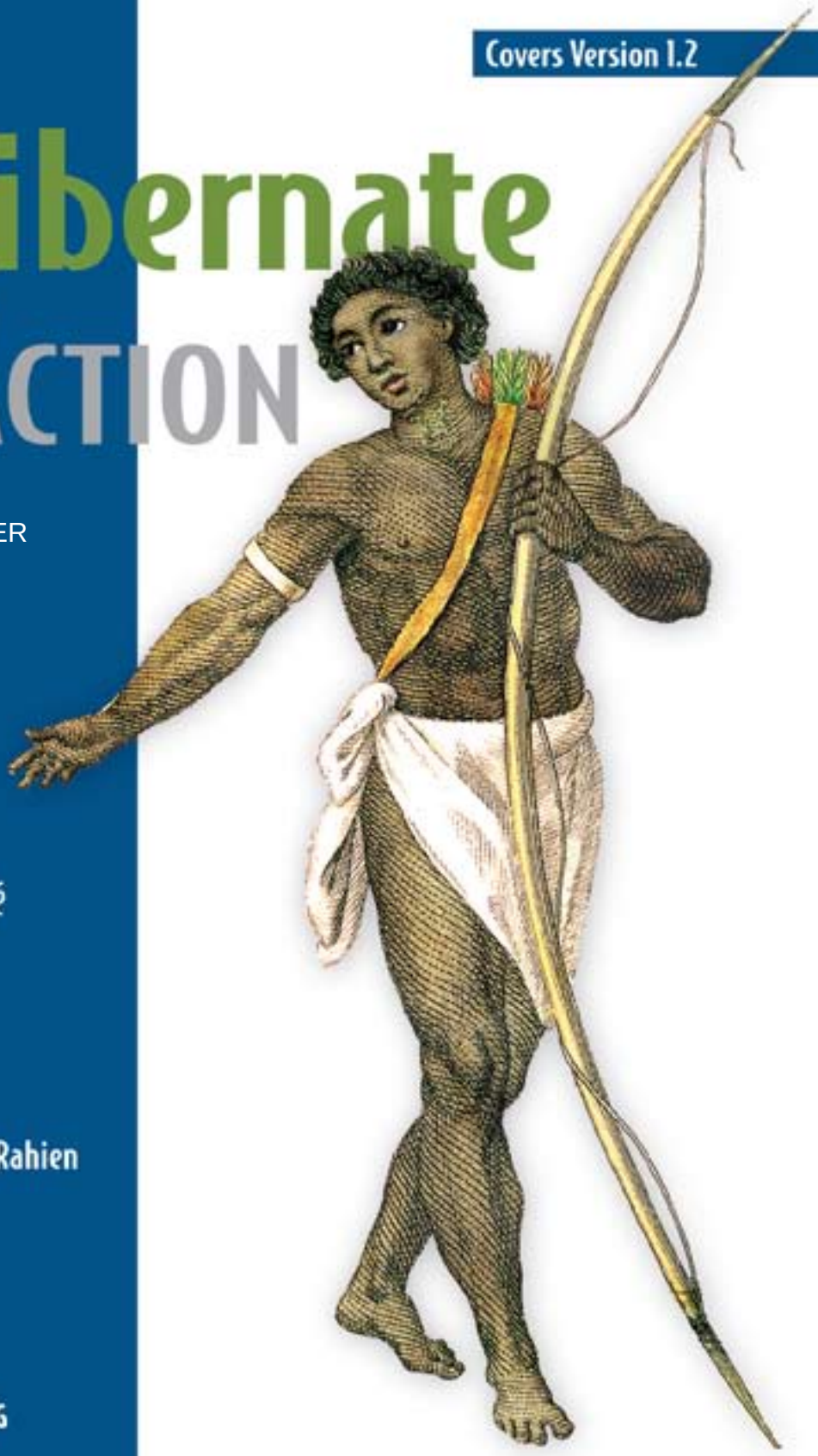
Covers Version 1.2

NHibernate IN ACTION

SAMPLE CHAPTER



Pierre Henri Kuaté
Tobin Harris
Christian Bauer
Gavin King

FOREWORD BY Ayende Rahien

MANNING

NHibernate in Action

Pierre Henri Kuaté
Tobin Harris
Christian Bauer
Gavin King

Chapter 5

brief contents

5 Transactions, concurrency, and caching

This chapter covers

- Database transactions and locking
- Long-running conversations
- The NHibernate first- and second-level caches
- The caching system in practice with CaveatEmptor

Now that you understand the basics of object/relational mapping with NHibernate, let's take a closer look at one of the core issues in database application design: transaction management. In this chapter, we examine how you use NHibernate to manage transactions, how concurrency is handled, and how caching is related to both aspects. Let's look at our example application.

Some application functionality requires that several different things be done together. For example, when an auction finishes, the CaveatEmptor application has to perform four tasks:

1. Mark the winning (highest amount) bid.
2. Charge the seller the cost of the auction.
3. Charge the successful bidder the price of the winning bid.
4. Notify the seller and the successful bidder.

What happens if you can't bill the auction costs because of a failure in the external credit card system? Your business requirements may state that either all listed actions must succeed or none must succeed. If so, you call these steps collectively a transaction or a unit of work. If only one step fails, the whole unit of work must fail. We say that the transaction is *atomic*: several operations are grouped together as a single indivisible unit.

Furthermore, transactions allow multiple users to work concurrently with the same data without compromising the integrity and correctness of the data; a particular transaction shouldn't be visible to and shouldn't influence other concurrently running transactions. Several different strategies are used to implement this behavior, which is called *isolation*. We'll explore them in this chapter.

Transactions are also said to exhibit *consistency* and *durability*. Consistency means that any transaction works with a consistent set of data and leaves the data in a consistent state when the transaction completes. Durability guarantees that once a transaction completes, all changes made during that transaction become persistent and aren't lost even if the system subsequently fails. Atomicity, consistency, isolation, and durability are together known as the ACID criteria.

We begin this chapter with a discussion of system-level *database transactions*, where the database guarantees ACID behavior. We look at the ADO.NET and Enterprise Services transactions APIs and see how NHibernate, working as a client of these APIs, is used to control database transactions.

In an online application, database transactions must have extremely short lifespans. A database transaction should span a single batch of database operations, interleaved with business logic. It should certainly not span interaction with the user. We'll augment your understanding of transactions with the notion of a long-running *user transaction* called a *conversation*, where database operations occur in several batches, alternating with user interaction. There are several ways to implement conversations in NHibernate applications, all of which are discussed in this chapter.

Finally, the subject of caching is much more closely related to transactions than it may appear at first sight. For example, caching lets you keep data close to where it's needed, but at the risk of getting stale over time. Therefore, caching strategies need to be balanced to also allow for consistent and durable transactions. In the second half of this chapter, armed with an understanding of transactions, we explore NHibernate's sophisticated cache architecture. You'll learn which data is a good candidate for caching and how to handle concurrency of the cache. You'll then enable caching in the CaveatEmptor application.

Let's begin with the basics and see how transactions work at the lowest level: the database.

5.1 Understanding database transactions

Databases implement the notion of a unit of work as a database transaction (sometimes called a *system transaction*). A database transaction groups data access operations. A transaction is guaranteed to end in one of two ways: it's either *committed* or *rolled back*. Hence, database transactions are always truly *atomic*. In figure 5.1, you can see this graphically.

If several database operations should be executed inside a transaction, you must mark the boundaries of the unit of work. You must start the transaction and, at some point, commit the changes. If an error occurs (either while executing operations or when committing the changes), you have to roll back the transaction to leave the data in a consistent state. This is known as *transaction demarcation*, and (depending on the API you use) it involves more or less manual intervention.

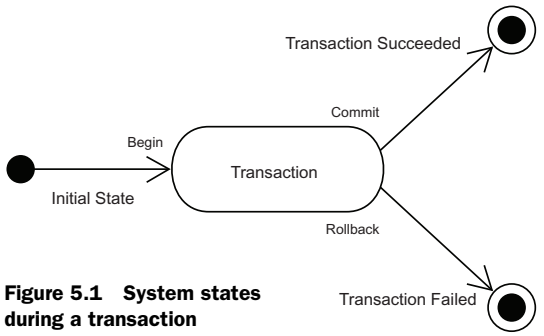


Figure 5.1 System states during a transaction

You may already have experience with two transaction-handling programming interfaces: the ADO.NET API and the COM+ automatic transaction processing service.

5.1.1 ADO.NET and Enterprise Services/COM+ transactions

Without Enterprise Services, the ADO.NET API is used to mark transaction boundaries. You begin a transaction by calling `BeginTransaction()` on an ADO.NET connection and end it by calling `Commit()`. You may, at any time, force an immediate rollback by calling `Rollback()`. Easy, huh?

In a system that stores data in multiple databases, a particular unit of work may involve access to more than one data store. In this case, you can't achieve atomicity using ADO.NET alone. You require a transaction manager with support for distributed transactions (two-phase commit). You communicate with the transaction manager using the COM+ automatic transaction-processing service.

With Enterprise Services, the automatic transaction-processing service is used not only for distributed transactions, but also for declarative transaction-processing features. Declarative transaction processing allows you to avoid explicit transaction demarcation calls in your application source code; rather, transaction demarcation is controlled by transaction attributes. The declarative transaction attribute specifies how an object participates in a transaction and is configured programmatically.

We aren't interested in the details of direct ADO.NET or Enterprise Services transaction demarcation. You'll be using these APIs mostly indirectly. Section 10.3 explains how to make NHibernate and Enterprise Services transactions work together.

NHibernate communicates with the database via an ADO.NET `IDbConnection`, and it provides its own abstraction layer, hiding the underlying transaction API. Using Enterprise Services doesn't require any change in the configuration of NHibernate.

Transaction management is exposed to the application developer via the NHibernate `ITransaction` interface. You aren't forced to use this API—NHibernate lets you control ADO.NET transactions directly. We don't discuss this option, because its use is discouraged; instead, we focus on the `ITransaction` API and its usage.

5.1.2 The NHibernate ITransaction API

The `ITransaction` interface provides methods for declaring the boundaries of a database transaction. Listing 5.1 shows an example of the basic usage of `ITransaction`.

Listing 5.1 Using the NHibernate `ITransaction` API

```
using( ISession session = sessions.OpenSession() )
using( session.BeginTransaction() ) {
   ConcludeAuction();
   session.Transaction.Commit();
}
```

The call to `session.BeginTransaction()` marks the beginning of a database transaction. This starts an ADO.NET transaction on the ADO.NET connection. With COM+, you don't need to create this transaction; the connection is automatically enlisted in the current distributed transaction, or you have to do it manually if you've disabled automatic transaction enlistment.

The call to `Transaction.Commit()` synchronizes the `ISession` state with the database. NHibernate then commits the underlying transaction if and only if `BeginTransaction()` started a new transaction (with COM+, you have to vote in favor of completing the distributed transaction).

If `ConcludeAuction()` threw an exception, the `using()` statement disposes the transaction (here, it means doing a rollback).

Do I need a transaction for read-only operations?

Due to the new connection release mode of NHibernate 1.2, a database connection is opened and closed for each transaction. As long as you're executing a single query, you can let NHibernate manage the transaction.

It's critically important to make sure the session is closed at the end in order to ensure that the ADO.NET connection is released and returned to the connection pool. (This step is the application's responsibility.)

NOTE After committing a transaction, the NHibernate session replaces it with a new transaction. This means you should keep a reference to the transaction you're committing if you think you'll need it afterward. This is necessary if you need to call `transaction.WasCommitted`.`session.Transaction.WasCommitted` always returns `false`.

Here is another version showing in detail where exceptions can be thrown and how to deal with them (this version is more complex than the one presented in chapter 2):

```
ISession session = sessions.OpenSession();
ITransaction tx = null;
```

```
try {
   tx = session.BeginTransaction();
   ConcludeAuction();
   tx.Commit();
} catch (Exception e) {
   if (tx != null) {
      try {
         tx.Rollback();
      } catch (HibernateException he) {
         //log here
      }
   }
   throw;
} finally {
   try {
      session.Close();
   } catch (HibernateException he) {
      throw;
   }
}
```

As you can see, even rolling back an `ITransaction` and closing the `ISession` can throw an exception. You shouldn't use this example as a template in your own application, because you should hide the exception handling with generic infrastructure code. You can, for example, wrap the thrown exception in your own `InfrastructureException`. We discuss this question of application design in more detail in section 8.1.

NOTE You must be aware of one important aspect: the `ISession` has to be immediately closed and discarded (not reused) when an exception occurs. NHibernate can't retry failed transactions. This is no problem in practice, because database exceptions are usually fatal (constraint violations, for example), and there is no well-defined state to continue after a failed transaction. An application in production shouldn't throw any database exceptions, either.

We've noted that the call to `Commit()` synchronizes the `ISession` state with the database. This is called *flushing*, a process you automatically trigger when you use the NHibernate `ITransaction` API.

5.1.3 Flushing the session

The NHibernate `ISession` implements transparent write-behind. This means changes to the domain model made in the scope of an `ISession` aren't immediately propagated to the database. Instead, NHibernate can coalesce many changes into a minimal number of database requests, helping minimize the impact of network latency.

For example, if a single property of an object is changed twice in the same `ITransaction`, NHibernate needs to execute only one SQL `UPDATE`.

NHibernate flushes occur only at the following times:

- When an `ITransaction` is committed
- Sometimes before a query is executed

- When the application calls
- `ISession.Flush()` explicitly

Flushing the `ISession` state to the database at the end of a database transaction is required in order to make the changes durable and is the common case. NHibernate doesn't flush before every query. But if changes are held in memory that would affect the results of the query, NHibernate will, by default, synchronize first.

You can control this behavior by explicitly setting the NHibernate `FlushMode` to the property `session.FlushMode`. The flush modes are as follow:

- *`FlushMode.Auto`*—The default. Enables the behavior just described.
- *`FlushMode.Commit`*—Specifies that the session won't be flushed before query execution (it will be flushed only at the end of the database transaction). Be aware that this setting may expose you to stale data: modifications you made to objects only in memory may conflict with the results of the query.
- *`FlushMode.Never`*—Lets you specify that only explicit calls to `Flush()` result in synchronization of session state with the database.

We don't recommend that you change this setting from the default. It's provided to allow performance optimization in rare cases. Likewise, most applications rarely need to call `Flush()` explicitly. This functionality is useful when you're working with triggers, mixing NHibernate with direct ADO.NET, or working with buggy ADO.NET drivers. You should be aware of the option but not necessarily look out for use cases.

We've discussed how NHibernate handles both transactions and the flushing of changes to the database. Another important responsibility of NHibernate is managing actual connections to the database. We discuss this next.

5.1.4 Understanding connection-release modes

As a valuable resource, the database connection should be held open for the shortest amount of time possible. NHibernate is smart enough to open it only when really necessary (opening the session doesn't automatically open the connection). Since NHibernate 1.2, it's also possible to define when the database connection should be closed.

Currently, two options are available. They're defined by the enumeration `NHibernate.ConnectionReleaseMode`:

- *`OnClose`*—This was the only mode available in NHibernate 1.0. In this case, the session releases the connection when it's closed.
- *`AfterTransaction`*—This is the default mode in NHibernate 1.2. The connection is released as soon as the transaction completes.

Note that you can use the `Disconnect()` method of the `ISession` interface to force the release of the connection (without closing the session) and the `Reconnect()` method to tell the session to obtain a new connection when needed.

Obviously, these modes are activated only for connections opened by NHibernate. If you open a connection and send it to NHibernate, you're also responsible for closing this connection.

To specify a mode, you must use the configuration parameter `hibernate.connection.release_mode`. Its default (and recommended) value is `auto`. It selects the best mode, which is currently `AfterTransaction`. The two other values are `on_close` and `after_transaction`.

Because NHibernate 1.2.0 has some problems dealing with APIs like `System.Transactions`, you must use `OnClose` mode if you discover that the session opens multiple connections in a single transaction.

Now that you understand the basic usage of database transactions with the NHibernate `ITransaction` interface, let's turn our attention more closely to the subject of concurrent data access.

It seems as though you shouldn't have to care about transaction isolation—the term implies that something either is or isn't isolated. This is misleading. *Complete* isolation of concurrent transactions is extremely expensive in terms of application scalability, so databases provide several degrees of isolation. For most applications, incomplete transaction isolation is acceptable. It's important to understand the degree of isolation you should choose for an application that uses NHibernate and how NHibernate integrates with the transaction capabilities of the database.

5.1.5 Understanding isolation levels

Databases (and other transactional systems) attempt to ensure *transaction isolation*, meaning that, from the point of view of each concurrent transaction, it appears no other transactions are in progress. Traditionally, this has been implemented using *locking*. A transaction may place a lock on a particular item of data, temporarily preventing access to that item by other transactions. Some modern databases such as Oracle and PostgreSQL implement transaction isolation using *multiversion concurrency control*, which is generally considered more scalable. We discuss isolation assuming a locking model (most of our observations are also applicable to multiversion concurrency).

This discussion is about database transactions and the isolation level provided by the database. NHibernate doesn't add additional semantics; it uses whatever is available with a given database. If you consider the many years of experience that database vendors have had with implementing concurrency control, you'll clearly see the advantage of this approach. Your part, as a NHibernate application developer, is to understand the capabilities of your database and how to change the database isolation behavior if required by your particular scenario (and by your data-integrity requirements).

ISOLATION ISSUES

First, let's look at several phenomena that break full transaction isolation. The ANSI SQL standard defines the standard transaction isolation levels in terms of which of these phenomena are permissible:

- *Lost update*—Two transactions both update a row, and then the second transaction aborts, causing both changes to be lost. This occurs in systems that don't implement any locking. The concurrent transactions aren't isolated.
- *Dirty read*—One transaction reads changes made by another transaction that hasn't yet been committed. This is dangerous, because those changes may later be rolled back.

- *Unrepeatable read*—A transaction reads a row twice and reads different state each time. For example, another transaction may have written to the row, and committed, between the two reads.
- *Second lost updates problem*—This is a special case of an unrepeatable read. Imagine that two concurrent transactions both read a row, one writes to it and commits, and then the second writes to it and commits. The changes made by the first writer are lost. This problem is also known as *last write wins*.
- *Phantom read*—A transaction executes a query twice, and the second result set includes rows that weren't visible in the first result set. (It need not be *exactly* the same query.) This situation is caused by another transaction inserting new rows between the execution of the two queries.

Now that you understand all the bad things that could occur, we define the various *transaction isolation levels* and see what problems they prevent.

ISOLATION LEVELS

The standard isolation levels are defined by the ANSI SQL standard. You'll use these levels to declare your desired transaction isolation later:

- *Read uncommitted*—Permits dirty reads but not lost updates. One transaction may not write to a row if another uncommitted transaction has already written to it. But any transaction may read any row. This isolation level may be implemented using exclusive write locks.
- *Read committed*—Permits unrepeatable reads but not dirty reads. This may be achieved using momentary shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row. But an uncommitted writing transaction blocks all other transactions from accessing the row.
- *Repeatable read*—Permits neither unrepeatable reads nor dirty reads. Phantom reads may occur. This may be achieved using shared read locks and exclusive write locks. Reading transactions block writing transactions (but not other reading transactions), and writing transactions block all other transactions.
- *Serializable*—Provides the strictest transaction isolation. It emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Serializability may not be implemented using only row-level locks; another mechanism must prevent a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row.

It's nice to know how all these technical terms are defined, but how does that help you choose an isolation level for your application?

5.1.6 Choosing an isolation level

Developers (ourselves included) are often unsure about what transaction isolation level to use in a production application. Too great a degree of isolation will harm performance of a highly concurrent application. Insufficient isolation may cause subtle bugs in your application that can't be reproduced and that you'll never find out about until the system is working under heavy load in the deployed environment.

Note that we refer to *caching* and *optimistic locking* (using versioning) in the following explanation, two concepts explained later in this chapter. You may want to skip this section and come back when it's time to make the decision about an isolation level in your application. Picking the right isolation level is, after all, highly dependent on your particular scenario. The following discussion contains recommendations; nothing is carved in stone.

NHibernate tries hard to be as transparent as possible regarding the transactional semantics of the database. Nevertheless, caching and optimistic locking affect these semantics. What is a sensible database-isolation level to choose in an NHibernate application?

First, eliminate the *read uncommitted* isolation level. It's extremely dangerous to use one transaction's uncommitted changes in a different transaction. The rollback or failure of one transaction would affect other concurrent transactions. Rollback of the first transaction could bring other transactions down with it or perhaps even cause them to leave the database in an inconsistent state. It's possible that changes made by a transaction that ends up being rolled back could be committed anyway, because they could be read and then propagated by another transaction that *is* successful!

Second, most applications don't need *serializable* isolation (phantom reads aren't usually a problem), and this isolation level tends to scale poorly. Few existing applications use serializable isolation in production; rather, they use pessimistic locks (see section 6.1.8), which effectively force a serialized execution of operations in certain situations.

This leaves you a choice between *read committed* and *repeatable read*. Let's first consider repeatable read. This isolation level eliminates the possibility that one transaction could overwrite changes made by another concurrent transaction (the second lost updates problem) if all data access is performed in a single atomic database transaction. This is an important issue, but using repeatable read isn't the only way to resolve it.

Let's assume you're using versioned data, something that NHibernate can do for you automatically. The combination of the (mandatory) NHibernate first-level session cache and versioning already gives you most of the features of repeatable-read isolation. In particular, versioning prevents the second lost update problem, and the first-level session cache ensures that the state of the persistent instances loaded by one transaction is isolated from changes made by other transactions. Thus read-committed isolation for all database transactions is acceptable if you use versioned data.

Repeatable read provides a bit more reproducibility for query result sets (only for the duration of the database transaction); but because phantom reads are still possible, there isn't much value in that. (It's also not common for web applications to query the same table twice in a single database transaction.)

You also have to consider the (optional) second-level NHibernate cache. It can provide the same transaction isolation as the underlying database transaction, but it may even weaken isolation. If you're heavily using a cache-concurrency strategy for the second-level cache that doesn't preserve repeatable-read semantics (for example,

the read-write and especially the nonstrict-read-write strategies, both discussed later in this chapter), the choice for a default isolation level is easy: you can't achieve repeatable read anyway, so there's no point slowing down the database. On the other hand, you may not be using second-level caching for critical classes, or you may be using a fully transactional cache that provides repeatable-read isolation. Should you use repeatable read in this case? You can if you like, but it's probably not worth the performance cost.

Setting the transaction isolation level allows you to choose a good default locking strategy for all your database transactions. How do you set the isolation level?

5.1.7 Setting an isolation level

Every ADO.NET connection to a database uses the database's default isolation level—usually read committed or repeatable read. This default can be changed in the database configuration. You may also set the transaction isolation for ADO.NET connections using an NHibernate configuration option:

```
<add
   key="hibernate.connection.isolation"
   value="ReadCommitted"
/>
```

NHibernate will then set this isolation level on every ADO.NET connection obtained from a connection pool before starting a transaction. Some of the sensible values for this option are as follow (you can also find them in `System.Data.IsolationLevel`):

- *`ReadUncommitted`*—Read-uncommitted isolation
- *`ReadCommitted`*—Read-committed isolation
- *`RepeatableRead`*—Repeatable-read isolation
- *`Serializable`*—Serializable isolation

Note that NHibernate never changes the isolation level of connections obtained from a datasource provided by COM+. You may change the default isolation using the `Isolation` property of `System.EnterpriseServices.TransactionAttribute`.

So far, we've introduced the issues that surround transaction isolation, the isolation levels available, and how to select the correct one for your application. As you can see, setting the isolation level is a global option that affects all connections and transactions. From time to time, it's useful to specify a more restrictive lock for a particular transaction. NHibernate allows you to explicitly specify the use of a *pessimistic* lock.

5.1.8 Using pessimistic locking

Locking is a mechanism that prevents concurrent access to a particular item of data. When one transaction holds a lock on an item, no concurrent transaction can read and/or modify this item. A lock may be just a momentary lock, held while the item is being read, or it may be held until the completion of the transaction. A *pessimistic lock* is a lock that is acquired when an item of data is read and that is held until transaction completion.

In read-committed mode (our preferred transaction isolation level), the database never acquires pessimistic locks unless explicitly requested by the application. Usually, pessimistic locks aren't the most scalable approach to concurrency. But in certain special circumstances, they may be used to prevent database-level deadlocks, which result in transaction failure. Some databases (Oracle, MySQL and PostgreSQL, for example, but not SQL Server) provide the SQL `SELECT...FOR UPDATE` syntax to allow the use of explicit pessimistic locks. You can check the NHibernate `Dialects` to find out if your database supports this feature. If your database isn't supported, NHibernate will always execute a normal `SELECT` without the `FOR UPDATE` clause.

The NHibernate `LockMode` class lets you request a pessimistic lock on a particular item. In addition, you can use the `LockMode` to force NHibernate to bypass the cache layer or to execute a simple version check. You'll see the benefit of these operations when we discuss versioning and caching.

Let's see how to use `LockMode`. Suppose you have a transaction that looks like this:

```
ITransaction tx = session.BeginTransaction();
Category cat = session.Get<Category>(catId);
cat.Name = "New Name";
tx.Commit();
```

It's possible to make this transaction use a pessimistic lock as follows:

```
ITransaction tx = session.BeginTransaction();
Category cat = session.Get<Category>(catId, LockMode.Upgrade);
cat.Name = "New Name";
tx.Commit();
```

With `LockMode.Upgrade`, NHibernate loads the `Category` using a `SELECT...FOR UPDATE`, thus locking the retrieved rows in the database until they're released when the transaction ends.

NHibernate defines several lock modes:

- *`LockMode.None`*—Don't go to the database unless the object isn't in either cache.
- *`LockMode.Read`*—Bypass both levels of the cache, and perform a version check to verify that the object in memory is the same version that currently exists in the database.
- *`LockMode.Upgrade`*—Bypass both levels of the cache, do a version check (if applicable), and obtain a database-level pessimistic upgrade lock, if that is supported.
- *`LockMode.UpgradeNoWait`*—The same as `UPGRADE`, but use a `SELECT...FOR UPDATE NOWAIT`, if that is supported. This disables waiting for concurrent lock releases, thus throwing a locking exception immediately if the lock can't be obtained.
- *`LockMode.Write`*—The lock is obtained automatically when NHibernate writes to a row in the current transaction (this is an internal mode; you can't specify it explicitly).

By default, `Load()` and `Get()` use `LockMode.None`. `LockMode.Read` is most useful with `ISession.Lock()` and a detached object. Here's an example:

```
Item item = ItemDAO.Load(1);
Bid bid = new Bid();
item.AddBid(bid);
//...
ITransaction tx = session.BeginTransaction();
session.Lock(item, LockMode.Read);
tx.Commit();
```

This code performs a version check on the detached `Item` instance to verify that the database row wasn't updated by another transaction since it was retrieved. If it was updated, a `StaleObjectStateException` is thrown.

Behind the scenes, NHibernate executes a `SELECT` to make sure there is a database row with the identifier (and version, if present) of the detached object. It doesn't check all the columns. This isn't a problem when the version is present because it's always updated when persisting the object using NHibernate. If, for some reason, you bypass NHibernate and use ADO.NET, don't forget to update the version.

By specifying an explicit `LockMode` other than `LockMode.None`, you force NHibernate to bypass both levels of the cache and go all the way to the database. We think that most of the time caching is more useful than pessimistic locking, so we don't use an explicit `LockMode` unless we really need it. Our advice is that if you have a professional DBA on your project, you should let the DBA decide which transactions require pessimistic locking once the application is up and running. This decision should depend on subtle details of the interactions between different transactions and can't be guessed up front.

Let's consider another aspect of concurrent data access. We think that most .NET developers are familiar with the notion of a database transaction, and that is what they usually mean by *transaction*. In this book, we consider this to be a *fine-grained* transaction, but we also consider a more coarse-grained notion. Coarse-grained transactions will correspond to what *the user of the application* considers a single unit of work. Why should this be any different than the fine-grained database transaction?

The database *isolates* the effects of concurrent database transactions. It should appear to the application that each transaction is the only transaction currently accessing the database (even when it isn't). Isolation is expensive. The database must allocate significant resources to each transaction for the duration of the transaction. In particular, as we've discussed, many databases lock rows that have been read or updated by a transaction, preventing access by any other transaction, until the first transaction completes. In highly concurrent systems with hundreds or thousands of updates per second, these locks can prevent scalability if they're held for longer than absolutely necessary. For this reason, you shouldn't hold the database transaction (or even the ADO.NET connection) open while waiting for user input. If a user takes a few minutes to enter data into a form while the database is locking resources, then other transactions may be blocked for that entire duration! All this, of course, also applies to

an NHibernate `ITransaction`, because it's an adaptor to the underlying database transaction mechanism.

If you want to handle long user "think time" while still taking advantage of the ACID attributes of transactions, simple database transactions aren't sufficient. You need a new concept: long-running user transactions also known as *conversations*.

5.2 Working with conversations

Business processes, which may be considered a single unit of work *from the point of view of the user*, necessarily span multiple user-client requests. This is especially true when a user makes a decision to update data on the basis of the current state of that data.

In an extreme example, suppose you collect data entered by the user on multiple screens, perhaps using wizard-style step-by-step navigation. You must read and write related items of data in several requests (hence several database transactions) until the user clicks Finish on the last screen. Throughout this process, the data must remain consistent and the user must be informed of any change to the data made by any concurrent transaction. We call this coarse-grained transaction concept a *conversation*: a broader notion of the unit of work.

We now restate this definition more precisely. Most .NET applications include several examples of the following type of functionality:

1. Data is retrieved and displayed on the screen, requiring the first database transaction as data is read.
2. The user has an opportunity to view and then modify the data in his own time. (Of course, no database transaction need here.)
3. The modifications are made persistent, which requires a second database transaction as data is written.

In more complicated applications, there may be several such interactions with the user before a particular business process is complete. This leads to the notion of a conversation (sometimes called a *long transaction*, *user transaction*, *application transaction,* or *business transaction*). We prefer the terms *conversation* and *user transaction* because they're less vague and emphasize the transaction aspect from the user's point of view.

During these long user-based transactions, you can't rely on the database to enforce isolation or atomicity of concurrent conversations. Isolation becomes something your application needs to deal with explicitly—and may even require getting the user's input.

5.2.1 An example scenario

Let's look at an example that uses a conversation. In the CaveatEmptor application, both the user who posted a comment and any system administrator can open an Edit Comment screen to delete or edit the text of a comment. Suppose two different administrators open the edit screen to view the same comment at the same time. Both edit the comment text and submit their changes. How can you handle this? There are three strategies:

- *Last commit wins*—Both updates are saved to the database, but the last one overwrites the first. No error message is shown to anyone, and the first update is silently lost forever.
- *First commit wins*—The first update is saved. When the second user attempts to save her changes, she receives an error message saying "your updates were lost because someone else updated the record while you were editing it." The user must start her edits again and hope she has more luck next time she clicks Save! This option is often called *optimistic locking*—the application optimistically assumes there won't be problems, but it checks and reports if there are.
- *Merge conflicting updates*—The first modification is persisted. When the second user saves, he's given the option of merging the records. This is also falls under the category of optimistic locking.

The first option, last commit wins, is problematic; the second user overwrites the changes of the first user without seeing the changes made by the first user or even knowing that they existed. In the example, this probably wouldn't matter, but it would be unacceptable in many scenarios. The second and third options are acceptable for most scenarios. In practice, there is no single best solution; you must investigate your own business requirements and select one of these three options.

When you're using NHibernate, the first option happens by default and requires no work on your part. You should assess which parts of your application, if any, can get away with this easy—but potentially dangerous—approach.

If you decide you need the optimistic-locking options, then you must add appropriate code to your application. NHibernate can help you implement this using *managed versioning for optimistic locking*, also known as *optimistic offline lock*.

5.2.2 Using managed versioning

Managed versioning relies on either a version number that is incremented or a timestamp that is updated to the current time, every time an object is modified. Note that it has no relation to SQL Server' TIMESTAMP column and that database-driven concurrency features aren't supported.

For NHibernate managed versioning, you must add a new property to your `Comment` class and map it as a version number using the `<version>` tag. First, let's look at the changes to the `Comment` class with the mapping attributes:

```
[Class(Table="COMMENTS")]
public class Comment {
   //...
   private int version;
   //...
   [Version(Column="VERSION")]
   public int Version {
      get { return version; }
      set { version = value; }
   }
}
```

You can also use a public scope for the setter and getter methods. When using XML, the `<version>` property mapping must come immediately after the identifier property mapping in the mapping file for the `Comment` class:

```
<class name="Comment" table="COMMENTS">
   <id ... >
   </id>
   <version name="Version" column="VERSION" />
   ...
</class>
```

The version number is just a counter value—it doesn't have any useful semantic value. Some people prefer to use a timestamp instead:

```
[Class(Table="COMMENTS")]
public class Comment {
   //...
   private DateTime lastUpdatedDatetime;
   //...
   [Timestamp(Column="LAST_UPDATED")]
   public DateTime LastUpdatedDatetime {
      get { return lastUpdatedDatetime; }
      set { lastUpdatedDatetime = value; }
   }
}
```

In theory, a timestamp is slightly less safe, because two concurrent transactions may both load and update the same item all in the same millisecond; in practice, this is unlikely to occur. But we recommend that new projects use a numeric version and not a timestamp.

You don't need to set the value of the version or timestamp property yourself; NHibernate will initialize the value when you first save a `Comment`, and increment or reset it whenever the object is modified.

NOTE Is the version of the parent updated if a child is modified? For example, if a single bid in the collection `bids` of an `Item` is modified, is the version number of the `Item` also increased by one or not? The answer to that and similar questions is simple: NHibernate increments the version number whenever an object is dirty. This includes all dirty properties, whether they're single-valued or collections. Think about the relationship between `Item` and `Bid`: if a `Bid` is modified, the version of the related `Item` isn't incremented. If you add or remove a `Bid` from the collection of bids, the version of the `Item` will be updated. (Of course, you would make `Bid` an immutable class, because it doesn't make sense to modify bids.)

Whenever NHibernate updates a comment, it uses the version column in the SQL `WHERE` clause:

```
update COMMENTS set COMMENT_TEXT='New comment text', VERSION=3
where COMMENT_ID=123 and VERSION=2
```

If another transaction had updated the same item since it was read by the current transaction, the `VERSION` column wouldn't contain the value 2, and the row wouldn't

be updated. NHibernate would check the row count returned by the ADO.NET driver—which in this case would be the number of rows updated, zero—and throw a `StaleObjectStateException`.

Using this exception, you can show the user of the second transaction an error message ("You have been working with stale data because another user modified it!") and let the *first commit win*. Alternatively, you can catch the exception, close the current session, and show the second user a new screen, allowing the user to manually *merge changes* between the two versions (using a new session).

It's possible to disable the increment of the version when a specific property is dirty. To do so, you must add `optimistic-lock="false"` to this property's mapping. This feature is available for properties, components, and collections (the owning entity's version isn't incremented).

It's also possible to implement optimistic locking without a version by writing

```
<class ... optimistic-lock="all">
```

In this case, NHibernate compares the states of all fields to find if any of them as changed. This feature works only for persistent objects; it can't work for detached objects because NHibernate doesn't have their state when they were loaded.

You may also write `<class ... optimistic-lock="dirty">` if you want NHibernate to allow concurrent modifications as long as they aren't done on the same columns. This allows, for example, an administrator to change the name of a customer while another administrator changes her phone number at the same time.

It's possible to avoid the execution of unnecessary updates (that will trigger *on update* events even when no changes have been made to detached instances) by mapping your entities using `<class ... select-before-update="true">`. NHibernate will select these entities and issue update commands only for dirty instances. But be aware of the performance implications of this feature.

As you can see, NHibernate makes it easy to use managed versioning to implement optimistic locking. Can you use optimistic locking and pessimistic locking together, or do you have to choose one? And why is it called *optimistic*?

5.2.3 Optimistic and pessimistic locking compared

An optimistic approach always assumes that everything will be OK and that conflicting data modifications are rare. Instead of being pessimistic and blocking concurrent data access immediately (and forcing execution to be serialized), optimistic concurrency control only blocks at the end of a unit of work and raises an error.

Both strategies have their places and uses, of course. Multi-user applications usually default to optimistic concurrency control and use pessimistic locks when appropriate. Note that the duration of a pessimistic lock in NHibernate is a single database transaction! This means you can't use an exclusive lock to block concurrent access longer than a single database transaction. We consider this a good thing, because the only solution would be an extremely expensive lock held in memory (or a so-called *lock table* in the database) for the duration of, for example, a conversation. This is almost always a performance bottleneck; every data access involves additional lock

checks to a synchronized lock manager. You may, if absolutely required in your particular application, implement a simple long pessimistic lock, using NHibernate to manage the lock table. Patterns for this can be found on the NHibernate website; but we definitely don't recommend this approach. You have to carefully examine the performance implications of this exceptional case.

Let's get back to conversations. You now know the basics of managed versioning and optimistic locking. In previous chapters (and earlier in this chapter), we talked about the NHibernate `ISession` not being the same as a transaction. An `ISession` has flexible scope, and you can use it in different ways with database and conversations. This means the *granularity* of an `ISession` is flexible; it can be any unit of work you want it to be.

5.2.4 Granularity of a session

To understand how you can use the NHibernate `ISession`, let's consider its relationship with transactions. Previously, we've discussed two related concepts:

- The scope of object identity (see section 4.1.4)
- The granularity of database and conversations

The NHibernate `ISession` instance defines the scope of object identity. The NHibernate `ITransaction` instance matches the scope of a database transaction.

What is the relationship between an `ISession` and a conversation? Let's start this discussion with the most common usage of the `ISession`. Usually, you open a new `ISession` for each client request (for example, a web browser request) and begin a new `ITransaction`. After executing the business logic, you commit the database transaction and close the `ISession`, before sending the response to the client (see figure 5.2).

The session (S1) and the database transaction (T1) have the same granularity. If you're not working with the concept of conversation, this simple approach is all you need in your application. We also like to call this approach *session-per-request*.

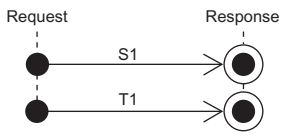


Figure 5.2 Using a one-to-one `ISession` and `ITransaction` per request/response cycle

If you need a long-running conversation, you may, thanks to detached objects (and NHibernate's support for optimistic locking, as discussed in the previous section), implement it using the same approach (see figure 5.3).

Suppose your conversation spans two client request/response cycles—for example, two HTTP requests in a web application. You can load the interesting objects in a first `ISession` and later reattach them to a new `ISession` after they've been

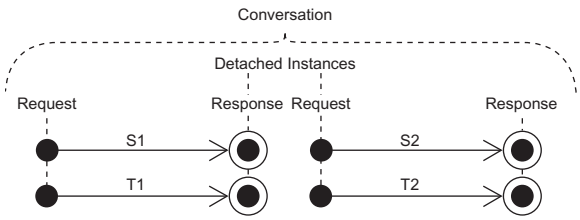


Figure 5.3 Implementing conversations with multiple `ISession`s, one for each request/response cycle

modified by the user. NHibernate will automatically perform a version check. The time between (S1, T1) and (S2, T2) can be "long": as long as your user needs to make his changes. This approach is also known as *session-per-request-with-detached-objects*.

Alternatively, you may prefer to use a single `ISession` that spans multiple requests to implement your conversation. In this case, you don't need to worry about reattaching detached objects, because the objects remain persistent in the context of the one long-running `ISession` (see figure 5.4). Of course, NHibernate is still responsible for performing optimistic locking.

A conversation keeps a reference to the session, although the session can be serialized, if required. The underlying ADO.NET connection must be closed, of course, and a new connection must be obtained on a subsequent request. This approach is known as *session-per-conversation* or *long session*.

Usually, your first choice should be to keep the NHibernate `ISession` open no longer than a single database transaction (session-per-request). Once the initial database transaction is complete, the longer the session remains open, the greater the chance that it holds stale data in its cache of persistent objects (the session is the mandatory first-level cache). Certainly, you should never reuse a single session for longer than it takes to complete a single conversation.

The question of conversations and the scope of the `ISession` is a matter of application design. We discuss implementation strategies with examples in section 10.2.

Finally, there is an important issue you may be concerned about. If you work with a legacy database schema, you probably can't add version or timestamp columns for NHibernate's optimistic locking.

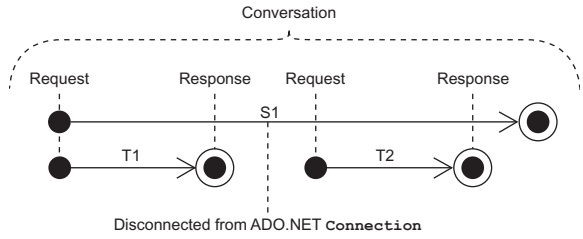


Figure 5.4 Implementing conversations with a long `ISession` using disconnection

5.2.5 Other ways to implement optimistic locking

If you don't have version or timestamp columns, NHibernate can still perform optimistic locking, but only for objects that are retrieved and modified in the same `ISession`. If you need optimistic locking for detached objects, you *must* use a version number or timestamp.

This alternative implementation of optimistic locking checks the current database state against the unmodified values of persistent properties at the time the object was retrieved (or the last time the session was flushed). You can enable this functionality by setting the `optimistic-lock` attribute on the class mapping:

```
<class name="Comment" table="COMMENT" optimistic-lock="all">
   <id ...../>
   ...
</class>
```

Now, NHibernate includes all properties in the `WHERE` clause:

```
update COMMENTS set COMMENT_TEXT='New text'
where COMMENT_ID=123
and   COMMENT_TEXT='Old Text'
and   RATING=5
and   ITEM_ID=3
and   FROM_USER_ID=45
```

Alternatively, NHibernate will include only the modified properties (only `COMMENT_TEXT`, in this example) if you set `optimistic-lock="dirty"`. (Note that this setting also requires you to set the class mapping to `dynamic-update="true"`.)

We don't recommend this approach; it's slower, more complex, and less reliable than version numbers and doesn't work if your conversation spans multiple sessions (which is the case if you're using detached objects).

Note that when you use `optimistic-lock="dirty"`, two concurrent conversations can update the same row as long as they change different columns; the result can be disastrous for the end user. If you have to use this feature, do it cautiously.

We now again switch perspective and consider a new aspect of NHibernate. We already mentioned the close relationship between transactions and caching in the introduction of this chapter. The fundamentals of transactions and locking, and also the session granularity concepts, are of central importance when we consider caching data in the application tier.

5.3 Caching theory and practice

A major justification for our claim that applications using an object/relational persistence layer are expected to outperform applications built using direct ADO.NET is the potential for caching. Although we'll argue passionately that most applications should be designed so that it's possible to achieve acceptable performance *without* the use of a cache, there is no doubt that for some kinds of applications—especially read-mostly applications or applications that keep significant metadata in the database—caching can have an enormous impact on performance.

We start our exploration of caching with some background information. This includes an explanation of the different caching and identity scopes and the impact of caching on transaction isolation. This information and these rules can be applied to caching in general; their validity isn't limited to NHibernate applications. This discussion gives you the background to understand why the NHibernate caching system is like it is. We then introduce the NHibernate caching system and show you how to enable, tune, and manage the first- and second-level NHibernate cache. We recommend that you carefully study the fundamentals laid out in this section before you start using the cache. Without the basics, you may quickly run into hard-to-debug concurrency problems and risk the integrity of your data.

A cache keeps a representation of current database state close to the application, either in memory or on the disk of the server machine. The cache is essentially merely a local copy of the data; it sits between your application and the database. The great

benefit is that your application can save time by not going to the database every time it needs data. This can be advantageous when it comes to reducing the strain on a busy database server and ensuring that data is served to the application quickly. The cache may be used to avoid a database hit whenever

- The application performs a lookup by identifier (primary key).
- The persistence layer resolves an association lazily.

It's also possible to cache the results of entire queries, so if the same query is issued repeatedly, the entire results are immediately available. As you'll see in chapter 8, this feature is used less often, but the performance gain of caching query results can be impressive in some situations.

Before we look at how NHibernate's cache works, let's walk through the different caching options and see how they're related to identity and concurrency.

5.3.1 Caching strategies and scopes

Caching is such a fundamental concept in object/relational persistence that you can't understand the performance, scalability, or transactional semantics of an ORM implementation without first knowing what caching strategies it uses. There are three main types of cache:

- *Transaction scope*—Attached to the current unit of work, which may be an actual database transaction or a conversation. It's valid and used as long as the unit of work runs. Every unit of work has its own cache.
- *Process scope*—Shared among many (possibly concurrent) units of work or transactions. Data in the process-scope cache is accessed by concurrently running transactions, obviously with implications on transaction isolation. A process-scope cache may store the persistent instances themselves in the cache, or it may store just their persistent state in some disassembled format.
- *Cluster scope*—Shared among multiple processes on the same machine or among multiple machines in a cluster. It requires some kind of *remote process communication* to maintain consistency. Caching information has to be replicated to all nodes in the cluster. For many (not all) applications, cluster-scope caching is of dubious value, because reading and updating the cache may be only marginally faster than going straight to the database. You must take many parameters into account, so a number of tests and tunings may be required before you make a decision.

Persistence layers may provide multiple levels of caching. For example, a *cache miss* (a cache lookup for an item that isn't contained in the cache) at transaction scope may be followed by a lookup at process scope. If that fails, going to the database for the data may be the last resort.

The type of cache used by a persistence layer affects the scope of object identity (the relationship between .NET object identity and database identity).

CACHING AND OBJECT IDENTITY

Consider a transaction-scope cache. It makes sense if this cache is also used as the identity scope of persistent objects. If, during a transaction, the application attempts to retrieve the same object twice, the transaction-scope cache ensures that both lookups return the same .NET instance. A transaction-scope cache is a good fit for persistence mechanisms that provide transaction-scoped object identity.

In the case of the process-scope cache, objects retrieved may be returned *by value*. Instead of storing and returning instances, the cache contains tuples of data. Each unit of work first retrieves a copy of the state from the cache (a tuple) and then uses that to construct its own persistent instance in memory. Unlike the transaction-scope cache discussed previously, the scope of the cache and the scope of the object identity are no longer the same.

A cluster-scope cache always requires remote communication because it's likely to operate over several machines. In the case of POCO-oriented persistence solutions like NHibernate, objects are always passed remotely by value. Therefore, the cluster-scope cache handles identity the same way as the process-scope cache; they each store copies of data and pass that data to the application so they can create their own instances from it. In NHibernate terms, they're both second-level caches, the main difference being that a cluster-scope cache can be distributed across several computers if needed.

Let's discuss which scenarios benefit from second-level caching and when to turn on the process- (or cluster-) scope second-level cache. Note that the first-level transaction scope cache is always on and is mandatory. The decisions to be made are whether to use the second-level cache, what type to use, and what data it should be used for.

CACHING AND TRANSACTION ISOLATION

A process- or cluster-scope cache makes data retrieved from the database in one unit of work visible to another unit of work. Essentially, the cache is allowing cached data to be shared among different units of work, multiple threads, or even multiple computers. This may have some nasty side effects on transaction isolation. We now discuss some critical considerations when you're choosing to use a process- or cluster-scope cache.

First, if more than one application is updating the database, then you shouldn't use process-scope caching, or you should use it only for data that changes rarely and may be safely refreshed by a cache expiry. This type of data occurs frequently in content-management applications but rarely in financial applications.

If you're designing your application to scale over several machines, you'll want to build it to support clustered operation. A process-scope cache doesn't maintain consistency between the different caches on different machines in the cluster. To achieve this, you should use a cluster-scope (distributed) cache instead of the process-scope cache.

Many .NET applications share access to their databases with other legacy applications. In this case, you shouldn't use any kind of cache beyond the mandatory transaction-scope cache. There is no way for a cache system to know when the legacy application updated the shared data. It's *possible* to implement application-level functionality to trigger an invalidation of the process- (or cluster-) scope cache when changes are made to the database, but we don't know of any standard or best way to

achieve this. It will never be a built-in feature of NHibernate. If you implement such a solution, you'll most likely be on your own, because it's extremely specific to the environment and products used.

After considering non-exclusive data access, you should establish what isolation level is required for the application data. Not every cache implementation respects all transaction isolation levels, and it's critical to find out what is required. Let's look at data that benefits most from a process- (or cluster-) scope cache.

A full ORM solution lets you configure second-level caching separately for each class. Good candidate classes for caching are classes that represent

- Data that rarely changes
- Noncritical data (for example, content-management data)
- Data that is local to the application and not shared

Bad candidates for second-level caching are

- Data that is updated often
- Financial data
- Data that is shared with a legacy application

But these aren't the only rules we usually apply. Many applications have a number of classes with the following properties:

- A small number of instances
- Each instance referenced by many instances of another class or classes
- Instances rarely (or never) updated

This kind of data is sometimes called *reference data*. Reference data is an excellent candidate for caching with a process or cluster scope, and any application that uses reference data heavily will benefit greatly if that data is cached. You allow the data to be refreshed when the cache-timeout period expires.

We've shaped a picture of a dual-layer caching system in the previous sections, with a transaction-scope first-level and an optional second-level process- or cluster-scope cache. This is close to the NHibernate caching system.

5.3.2 The NHibernate cache architecture

As we said earlier, NHibernate has a two-level cache architecture. The various elements of this system are shown in figure 5.5.

The first-level cache is the `ISession`. A session lifespan corresponds to either a database transaction or a conversation (as explained earlier in this chapter). We consider the cache associated with the `ISession` to be a transaction-scope cache. The first-level cache is mandatory and can't be turned off; it also guarantees object identity inside a transaction.

The second-level cache in NHibernate is pluggable and may be scoped to the process or cluster. This is a cache of state (returned by value), not of persistent instances. A cache-concurrency strategy defines the transaction isolation details for a particular

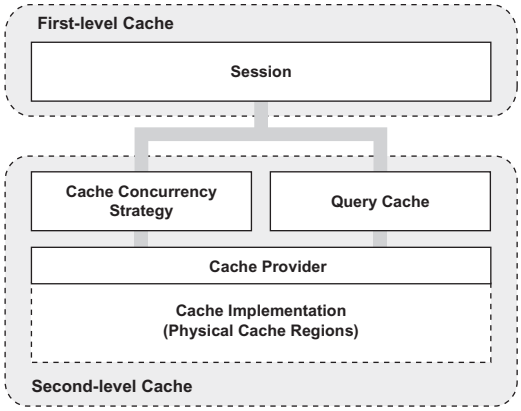


Figure 5.5 NHibernate's two-level cache architecture

item of data, whereas the cache provider represents the physical, actual cache implementation. Use of the second-level cache is optional and can be configured on a per-class and per-association basis.

NHibernate also implements a cache for query result set that integrates closely with the second-level cache. This is an optional feature. We discuss the query cache in chapter 8, because its usage is closely tied to the query being executed.

Let's start with using the first-level cache, also called the *session cache*.

USING THE FIRST-LEVEL CACHE

The session cache ensures that when the application requests the same persistent object twice in a particular session, it gets back the same (identical) .NET instance. This sometimes helps avoid unnecessary database traffic. More important, it ensures the following:

- The persistence layer isn't vulnerable to stack overflows in the case of circular references in a graph of objects.
- There can never be conflicting representations of the same database row at the end of a database transaction. At most a single object represents any database row. All changes made to that object may be safely written to the database (flushed).
- Changes made in a particular unit of work are always immediately visible to all other code executed inside that unit of work.

You don't have to do anything special to enable the session cache. It's always on and, for the reasons shown, can't be turned off.

Whenever you pass an object to `Save()`, `Update()`, or `SaveOrUpdate()`, and whenever you retrieve an object using `Load()`, `Find()`, `List()`, `Iterate()`, or `Filter()`, that object is added to the session cache. When `Flush()` is subsequently called, the state of that object is synchronized with the database.

If you don't want this synchronization to occur, or if you're processing a huge number of objects and need to manage memory efficiently, you can use the `Evict()`

method of the `ISession` to remove the object and its collections from the first-level cache. This can be useful in several scenarios.

MANAGING THE FIRST-LEVEL CACHE

Consider this frequently asked question: "I get an `OutOfMemoryException` when I try to load 100,000 objects and manipulate all of them. How can I do mass updates with NHibernate?"

It's our view that ORM isn't suitable for mass-update (or mass-delete) operations. If you have a use case like this, a different strategy is almost always better: call a stored procedure in the database, or use direct SQL `UPDATE` and `DELETE` statements for that particular use case. Don't transfer all the data to main memory for a simple operation if it can be performed more efficiently by the database. If your application is *mostly* mass-operation use cases, ORM isn't the right tool for the job!

If you insist on using NHibernate for mass operations, you can immediately `Evict()` each object after it has been processed (while iterating through a query result), and thus prevent memory exhaustion. To completely evict all objects from the session cache, call `Session.Clear()`. We aren't trying to convince you that evicting objects from the first-level cache is a bad thing in general, but that good use cases are rare. Sometimes, using projection and a report query, as discussed in section 8.4.5, may be a better solution.

Note that eviction, like save or delete operations, can be automatically applied to associated objects. NHibernate evicts associated instances from the `ISession` if the mapping attribute `cascade` is set to `all` or `all-delete-orphan` for a particular association.

When a first-level cache miss occurs, NHibernate tries again with the second-level cache if it's enabled for a particular class or association.

THE NHIBERNATE SECOND-LEVEL CACHE

The NHibernate second-level cache has process or cluster scope; all sessions share the same second-level cache. The second-level cache has the scope of an `ISessionFactory`.

Persistent instances are stored in the second-level cache in a *disassembled* form. Think of disassembly as a process a bit like serialization (but the algorithm is much, much faster than .NET serialization).

The internal implementation of this process/cluster scope cache isn't of much interest; more important is the correct usage of the *cache policies*—that is, caching strategies and physical cache providers.

Different kinds of data require different cache policies: the ratio of reads to writes varies, the size of the database tables varies, and some tables are shared with other external applications. So the second-level cache is configurable at the granularity of an individual class or collection role. This lets you, for example, enable the second-level cache for reference data classes and disable it for classes that represent financial records. The cache policy involves setting the following:

- Whether the second-level cache is enabled
- The NHibernate concurrency strategy
- The cache expiration policies (such as expiration or priority)

Not all classes benefit from caching, so it's extremely important to be able to disable the second-level cache. To repeat, the cache is usually useful only for read-mostly classes. If you have data that is updated more often than it's read, don't enable the second-level cache, even if all other conditions for caching are true! Furthermore, the second-level cache can be dangerous in systems that share the database with other writing applications. As we explained in earlier sections, you must exercise careful judgment here.

The NHibernate second-level cache is set up in two steps. First, you have to decide which *concurrency strategy* to use. After that, you configure cache expiration and cache attributes using the *cache provider*.

BUILT-IN CONCURRENCY STRATEGIES

A concurrency strategy is a mediator; it's responsible for storing items of data in the cache and retrieving them from the cache. This is an important role, because it also defines the transaction isolation semantics for that particular item. You have to decide, for each persistent class, which cache concurrency strategy to use, if you want to enable the second-level cache.

Three built-in concurrency strategies are available, representing decreasing levels of strictness in terms of transaction isolation:

- *Read-write*—Maintains *read-committed* isolation, using a timestamping mechanism. It's available only in nonclustered environments. Use this strategy for read-mostly data where it's critical to prevent stale data in concurrent transactions, in the rare case of an update.
- *Nonstrict-read-write*—Makes no guarantee of consistency between the cache and the database. If there is a possibility of concurrent access to the same entity, you should configure a sufficiently short expiry timeout. Otherwise, you may read stale data in the cache. Use this strategy if data rarely changes (many hours, days, or even a week) and a small likelihood of stale data isn't of critical concern. NHibernate invalidates the cached element if a modified object is flushed, but this is an asynchronous operation, without any cache locking or guarantee that the retrieved data is the latest version.
- *Read-only*—Suitable for data that never changes. Use it for reference data only.

Note that with decreasing strictness comes increasing performance. You have to carefully evaluate the performance of a clustered cache with full transaction isolation before using it in production. In many cases, you may be better off disabling the second-level cache for a particular class if stale data isn't an option. First, benchmark your application with the second-level cache disabled. Then, enable it for good candidate classes, one at a time, while continuously testing the performance of your system and evaluating concurrency strategies.

It's possible to define your own concurrency strategy by implementing `NHibernate.Cache.ICacheConcurrencyStrategy`, but this is a relatively difficult task and only appropriate for extremely rare cases of optimization.

Your next step after considering the concurrency strategies you'll use for your cache candidate classes is to pick a *cache provider*. The provider is a plug-in, the physical implementation of a cache system.

CHOOSING A CACHE PROVIDER

For now, NHibernate forces you to choose a single cache provider for the whole application. The following providers are released with NHibernate:

- *Hashtable*—Not intended for production use. It only caches in memory and can be set using its provider: `NHibernate.Cache.HashtableCacheProvider` (available in `NHibernate.dll`.).
- *SysCache*—Relies on `System.Web.Caching.Cache` for the underlying implementation, so you can refer to the documentation of the ASP.NET caching feature to understand how it works. Its NHibernate provider is the class `NHibernate.Caches.SysCache.SysCacheProvider` in library `NHibernate.Caches.SysCache.dll`. This provider should only be used with ASP.NET Web Applications.
- *Prevalence*—Makes it possible to use the underlying Bamboo.Prevalence implementation as a cache provider. Its NHibernate provider is `NHibernate.Caches.Prevalence.PrevalenceCacheProvider` in the library `NHibernate.Caches.Prevalence.dll`. You can also visit Bamboo.Prevalence's website: http://bbooprevalence.sourceforge.net/.

You'll learn about some distributed cache providers in the next section. And it's easy to write an adaptor for other products by implementing `NHibernate.Cache.ICacheProvider`.

Every cache provider supports NHibernate Query Cache and is compatible with every concurrency strategy (read-only, nonstrict-read-write, and read-write).

Setting up caching therefore involves two steps:

1. Look at the mapping files for your persistent classes, and decide which cache-concurrency strategy you'd like to use for each class and each association.
2. Enable your preferred cache provider in the NHibernate configuration, and customize the provider-specific settings.

Let's add caching to the CaveatEmptor `Category` and `Item` classes.

5.3.3 Caching in practice

Remember that you don't have to explicitly enable the first-level cache. Let's declare caching policies and set up cache providers for the second-level cache in the CaveatEmptor application.

The `Category` has a small number of instances and is rarely updated, and instances are shared among many users, so it's a great candidate for use of the second-level cache. Start by adding the mapping element required to tell NHibernate to cache `Category` instances:

```
[Class(Table="CATEGORY")]
public class Category {
```

```
    [Cache(-1, Usage=CacheUsage.ReadWrite)]
    [Id .... ]
    public long Id { ... }
}
```

Note that, like the attribute `[Discriminator]`, you can put `[Cache]` on any field/property; just be careful when mixing it with other attributes (here, you use the position -1 because it must come before the other attributes).

Here is the corresponding XML mapping:

```
<class
      name="Category"
      table="CATEGORY">
   <cache usage="read-write"/>
   <id .... >
</class>
```

The `usage="read-write"` attribute tells NHibernate to use a read-write concurrency strategy for the `Category` cache. NHibernate will now try the second-level cache whenever you navigate to a `Category` or when you load a `Category` by identifier.

You use read-write instead of nonstrict-read-write because `Category` is a highly concurrent class, shared among many concurrent transactions, and it's clear that a read-committed isolation level is good enough. Nonstrict-read-write would probably be an acceptable alternative, because a small probability of inconsistency between the cache and database is acceptable (the category hierarchy has little financial significance).

This mapping is enough to tell NHibernate to cache all simple `Category` property values but not the state of associated entities or collections. Collections require their own `<cache>` element. For the `Items` collection, you'll use a read-write concurrency strategy:

```
<class
      name="Category"
      table="CATEGORY">
      <cache usage="read-write"/>
      <id ....
      <set name="Items" lazy="true">
        <cache usage="read-write"/>
        <key ....
   </set>
</class>
```

This cache will be used when enumerating the collection `category.Items`, for example. Note that deleting an item that exists on a collection in the second-level cache will cause an exception; make sure that you remove the item from the collection in the cache before deleting it.

A collection cache holds only the identifiers of the associated item instances. If you require the instances themselves to be cached, you must enable caching of the `Item` class. A read-write strategy is especially appropriate here. Your users don't want to make decisions (placing a `Bid`) based on possibly stale data. Let's go a step further and consider the collection of `Bids`. A particular `Bid` in the `Bids` collection is immutable;

but you have to map the collection using `read-write`, because new bids may be made at any time (and it's critical that you be immediately aware of new bids):

```
<class
   name="Item"
   table="ITEM">
   <cache usage="read-write"/>
   <id ....
   <set name="Bids" lazy="true">
      <cache usage="read-write"/>
      <key ....
   </set>
</class>
```

To the immutable `Bid` class, you apply a read-only strategy:

```
<class
   name="Bid"
   table="BID">
   <cache usage="read-only"/>
   <id ....
</class>
```

Cached `Bid` data is valid indefinitely, because bids are never updated. No cache invalidation is required. (Instances may be evicted by the cache provider—for example, if the maximum number of objects in the cache is reached.)

`User` is an example of a class that could be cached with the nonstrict-read-write strategy, but we aren't certain that it makes sense to cache users.

Let's set the cache provider, expiration policies, and physical properties of the cache. You use *cache regions* to configure class and collection caching individually.

UNDERSTANDING CACHE REGIONS

NHibernate keeps different classes/collections in different cache *regions*. A region is a named cache: a handle by which you can reference classes and collections in the cache-provider configuration and set the expiration policies applicable to that region.

The name of the region is the class name, in the case of a class cache, or the class name together with the property name, in the case of a collection cache. `Category` instances are cached in a region named `NHibernate.Auction.Category`, and the `items` collection is cached in a region named `NHibernate.Auction.Category.Items`.

You can use the NHibernate configuration property `hibernate.cache.region_prefix` to specify a root region name for a particular `ISessionFactory`. For example, if the prefix was set to `Node1`, `Category` would be cached in a region named `Node1.NHibernate.Auction.Category`. This setting is useful if your application includes multiple `ISessionFactory` instances.

Now that you know about cache regions, let's configure the expiry policies for the `Category` cache. First you'll choose a cache provider.

SETTING UP A LOCAL CACHE PROVIDER

You need to set the property that selects a cache provider:

```
<add
   key="hibernate.cache.provider_class"
```

```
    value="NHibernate.Caches.SysCache.SysCacheProvider,
    NHibernate.Caches.SysCache"
/>
```

Here, you choose SysCache as your second-level cache.

Now, you need to specify the expiry policies for the cache regions. SysCache provides two parameters: an expiration value which is the number of seconds to wait before expiring each item (the default value is 300 seconds) and a priority value that is a numeric cost of expiring each item, where 1 is a low cost, 5 is the highest, and 3 is normal. Note that only values 1 through 5 are valid; they refer to the `System.Web.Caching.CacheItemPriority` enumeration.

SysCache has a configuration-file section handler to allow configuring different expirations and priorities for different regions. Here's how you can configure the `Category` class:

```
<?xml version="1.0" ?>
<configuration>
   <configSections>
      <section
         name="syscache"
         type="NHibernate.Caches.SysCache.SysCacheSectionHandler,
             NHibernate.Caches.SysCache" />
   </configSections>
   <syscache>
      <cache region="Category" expiration="36000" priority="5" />
   </syscache>
</configuration>
```

There are a small number of categories, and they're all shared among many concurrent transactions. You therefore define a high expiration value (10 hours) and give it a high priority so the categories stay in the cache as long as possible.

`Bid`s, on the other hand, are small and immutable, but there are many of them; you must configure SysCache to carefully manage the cache memory consumption. You use both a low expiration value and a low priority:

```
<cache region="Bid" expiration="300" priority="1" />
```

The result is that cached bids are removed from the cache after five minutes or if the cache is full (because they have the lowest priority).

Optimal cache-eviction policies are, as you can see, specific to the particular data and particular application. You must consider many external factors, including available memory on the application server machine, expected load on the database machine, network latency, existence of legacy applications, and so on. Some of these factors can't possibly be known at development time, so you'll often need to iteratively test the performance impact of different settings in the production environment or a simulation of it. This is especially true in a more complex scenario, with a replicated cache deployed to a cluster of server machines.

USING A DISTRIBUTED CACHE

SysCache and Prevalence are excellent cache providers if your application is deployed on a single machine. But enterprise applications supporting thousands of concurrent

users may require more computing power, and scaling your application may be critical to the success of your project. NHibernate applications are naturally scalable—that is, NHibernate behaves the same whether it's deployed to a single machine or to many machines. The only feature of NHibernate that must be configured specifically for clustered operation is the second-level cache. With a few changes to your cache configuration, you can use a clustered caching system.

It isn't necessarily wrong to use a purely local (non–cluster-aware) cache provider in a cluster. Some data—especially immutable data, or data that can be refreshed by cache timeout—doesn't require clustered invalidation and may safely be cached locally, even in a clustered environment. You may be able to have each node in the cluster use a local instance of SysCache, and carefully choose sufficiently short `expiration` values.

But if you require strict cache consistency in a clustered environment, you must use a more sophisticated cache provider. Some distributed cache providers are available for NHibernate. You can consider the following:

- *MemCache*—Released with NHibernate. It uses memcached, a distributed cache system available under Linux, so you can use it with Mono (or use the *VMWare Memcached appliance* under Windows). For more details, visit http://www.danga.com/memcached/. Its NHibernate provider is the class `NHibernate.Caches.MemCache.MemCacheProvider` in the library `NHibernate.Caches.MemCache.dll`.
- *NCache*—A commercial distributed cache provider. Its website is http://www.alachisoft.com/ncache/.
- *Microsoft Velocity*—A commercial distributed cache, currently in CTP2 (at technology preview stage).

We don't dig into the details of these distributed cache providers. Distributed caching is a complex topic; we recommend that you read some articles about this topic and test these providers.

Note that some distributed cache providers work only with some cache concurrency strategies. A nice trick can help you avoid checking your mapping files one by one. Instead of placing a `[Cache]` attribute in your entities or placing `<cache>` elements in your mapping files, you can centralize cache configuration in `hibernate.cfg.xml`:

```
<hibernate-configuration>
   <session-factory>
      <class-cache
            class="NHibernate.Auction.Model.Bid, NHibernate.Auction"
            usage="read-only"/>
      <collection-cache
            collection="NHibernate.Auction.Model.Item.Bids"
            usage="read-write"/>
   </session-factory>
</hibernate-configuration>
```

You enable read-only caching for `Bid` and read-write caching for the `Bids` collection in this example. But be aware of one important caveat: at the time of this writing,

NHibernate will run into a conflict if you also have `<cache>` elements in the mapping file for `Item`. You therefore can't use the global configuration to override the mapping file settings. We recommend that you use the centralized cache configuration from the start, especially if you aren't sure how your application may be deployed. It's also easier to tune cache settings with a centralized configuration.

There is an optional setting to consider. For cluster cache providers, it may be better to set the NHibernate configuration option `hibernate.cache.use_minimal_puts` to `true`. When this setting is enabled, NHibernate adds an item to the cache only after checking to ensure the item isn't already cached. This strategy performs better if cache writes (puts) are much more expensive than cache reads (gets). This is the case for a replicated cache in a cluster, but not for a local cache (the default is `false`, optimized for a local cache). Whether you're using a cluster or a local cache, you sometimes need to control it programmatically for testing or tuning purposes.

CONTROLLING THE SECOND-LEVEL CACHE

NHibernate has some useful methods that will help you test and tune your cache. You may wonder how to disable the second-level cache completely. NHibernate loads the cache provider and starts using the second-level cache only if you have any cache declarations in your mapping files or XML configuration file. If you comment them out, the cache is disabled. This is another good reason to prefer centralized cache configuration in `hibernate.cfg.xml`.

Just as the `ISession` provides methods for controlling the first-level cache programmatically, so does the `ISessionFactory` for the second-level cache.

You can call `Evict()` to remove an element from the cache, by specifying the class and the object identifer value:

```
SessionFactory.Evict( typeof(Category), 123 );
```

You can also evict all elements of a certain class or evict only a particular collection role:

```
SessionFactory.Evict( typeof(Category) );
```

You'll rarely need these control mechanisms; use them with care, because they don't respect any transaction isolation semantics of the usage strategy.

5.4 Summary

This chapter was dedicated to transactions (fine-grained and coarse-grained), concurrency and data caching.

You learned that for a single unit of work, either all operations should be completely successful or the whole unit of work should fail (and changes made to persistent state should be rolled back). This led us to the notion of a transaction and the ACID attributes. A transaction is atomic, leaves data in a consistent state, and is isolated from concurrently running transactions, and you have the guarantee that data changed by a transaction is durable.

You use two transaction concepts in NHibernate applications: short database transactions and long-running conversations. Usually, you use read committed isolation for database transactions, together with optimistic concurrency control (version and timestamp checking) for long conversations. NHibernate greatly simplifies the implementation of conversations because it manages version numbers and timestamps for you.

Finally, we discussed the fundamentals of caching, and you learned how to use caching effectively in NHibernate applications.

NHibernate provides a dual-layer caching system with a first-level object cache (the `ISession`) and a pluggable second-level data cache. The first-level cache is always active—it's used to resolve circular references in your object graph and to optimize performance in a single unit of work. The second-level cache, on the other hand, is optional and works best for read-mostly candidate classes. You can configure a nonvolatile second-level cache for reference (read-only) data or even a second-level cache with full transaction isolation for critical data. But you have to carefully examine whether the performance gain is worth the effort. The second-level cache can be customized fine-grained, for each persistent class and even for each collection and class association. Used correctly and thoroughly tested, caching in NHibernate gives you a level of performance that is almost unachievable in a hand-coded data access layer.

Now that we've covered most of the fundamental aspects key to NHibernate applications, we can delve into some of the more advanced capabilities of NHibernate. The next chapter will start by discussing some of the advanced NHibernate mapping concepts that will enable you to handle the most demanding persistence requirements.

.NET PROGRAMMING

NHibernate IN ACTION

Pierre Henri Kuaté • Tobin Harris • Christian Bauer • Gavin King

FOREWORD BY Ayende Rahien NHibernate Committer

Efficient and secure data access is key for software applications. Microsoft is promoting object/relational mapping to help achieve this, but is yet to offer a complete solution. In the meantime, with NHibernate you get an open source object/relational mapper that is based on the hugely popular Java Hibernate project. With a wealth of features and a straightforward setup, you can build best-practice enterprise .NET applications with less effort and less time.

NHibernate in Action is a carefully crafted guide that introduces NHibernate and ORM to .NET developers. You'll learn to map information between business objects and database tables and explore NHibernate's internal architecture. A complete example demonstrates entity and relationship mappings and CRUD operations, along with advanced techniques like caching, concurrency access, and isolation levels. The book shows you how to refactor to a layered architecture. It also discusses patterns for integrating services and for crossing distribution boundaries.

What's Inside

- Object/relational mapping for .NET
- NHibernate configuration, mapping, and query APIs
- Data-binding objects to .NET GUI controls
- Advanced session management and distributed transactions

About the Authors

Pierre Henri Kuaté is a developer on the NHibernate project team and author of the NHibernate.Mapping.Attributes library. Tobin Harris is an independent consultant and founder of the Open Source SqlBuddy project. Gavin King and Christian Bauer are the authors of Hibernate in Action and original founders of the Hibernate project.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/NHibernateinAction

MANNING US/CAN $49.99 [INCLUDING EBOOK]

"A much needed book for novices and experts"

—Mark Monster, Rubicon

"Finally, a great book covering NHibernate"

—Paul Wilson, McKesson

"A must-have for NHibernate developers"

—Ayende Rahien
NHibernate Committer

"Accelerates your learning curve for ORM in .NET."

—Doug Warren
Java Web Services

ISBN-13: 978-1932394924
ISBN-10: 1932394923