1 Understanding object/relational persistence

In this chapter

- Persistence with SQL databases in Java applications
- The object/relational paradigm mismatch
- Introducing ORM, JPA, and Hibernate

This book is about Hibernate; our focus is on using Hibernate as a provider of the Java Persistence API. We cover basic and advanced features and describe some ways to develop new applications using Java Persistence. Often, these recommendations aren't specific to Hibernate. Sometimes they're our own ideas about the *best* ways to do things when working with persistent data, explained in the context of Hibernate.

The approach to managing persistent data has been a key design decision in every software project we've worked on. Given that persistent data isn't a new or unusual requirement for Java applications, you'd expect to be able to make a simple choice among similar, well-established persistence solutions. Think of web application frameworks (JavaServer Faces versus Struts versus GWT), GUI component

frameworks (Swing versus SWT), or template engines (JSP versus Thymeleaf). Each of the competing solutions has various advantages and disadvantages, but they all share the same scope and overall approach. Unfortunately, this isn't yet the case with persistence technologies, where we see some wildly differing solutions to the same problem.

Persistence has always been a hot topic of debate in the Java community. Is persistence a problem that is already solved by SQL and extensions such as stored procedures, or is it a more pervasive problem that must be addressed by special Java component models, such as EJBs? Should we hand-code even the most primitive CRUD (create, read, update, delete) operations in SQL and JDBC, or should this work be automated? How do we achieve portability if every database management system has its own SQL dialect? Should we abandon SQL completely and adopt a different database technology, such as object database systems or NoSQL systems? The debate may never end, but a solution called *object/relational mapping* (ORM) now has wide acceptance, thanks in large part to the innovations of Hibernate, an open source ORM service implementation.

Before we can get started with Hibernate, you need to understand the core problems of object persistence and ORM. This chapter explains why you need tools like Hibernate and specifications such as the *Java Persistence API* (JPA).

First we define persistent data management in the context of object-oriented applications and discuss the relationship of SQL, JDBC, and Java, the underlying technologies and standards that Hibernate builds on. We then discuss the so-called *object/relational paradigm mismatch* and the generic problems we encounter in object-oriented software development with SQL databases. These problems make it clear that we need tools and patterns to minimize the time we have to spend on the persistence-related code in our applications.

The best way to learn Hibernate isn't necessarily linear. We understand that you may want to try Hibernate right away. If this is how you'd like to proceed, skip to the next chapter and set up a project with the "Hello World" example. We recommend that you return here at some point as you go through this book; that way, you'll be prepared and have all the background concepts you need for the rest of the material.

1.1 What is persistence?

Almost all applications require persistent data. Persistence is one of the fundamental concepts in application development. If an information system didn't preserve data when it was powered off, the system would be of little practical use. *Object persistence* means individual objects can outlive the application process; they can be saved to a data store and be re-created at a later point in time. When we talk about persistence in Java, we're normally talking about mapping and storing object instances in a database using SQL. We start by taking a brief look at the technology and how it's used in Java. Armed with this information, we then continue our discussion of persistence and how it's implemented in object-oriented applications.

1.1.1 Relational databases

You, like most other software engineers, have probably worked with SQL and relational databases; many of us handle such systems every day. Relational database management systems have SQL-based application programming interfaces; hence, we call today's relational database products SQL *database management systems* (DBMS) or, when we're talking about particular systems, SQL *databases*.

Relational technology is a known quantity, and this alone is sufficient reason for many organizations to choose it. But to say only this is to pay less respect than is due. Relational databases are entrenched because they're an incredibly flexible and robust approach to data management. Due to the well-researched theoretical foundation of the relational data model, relational databases can guarantee and protect the integrity of the stored data, among other desirable characteristics. You may be familiar with E.F. Codd's four-decades-old introduction of the relational model, *A Relational Model of Data for Large Shared Data Banks* (Codd, 1970). A more recent compendium worth reading, with a focus on SQL, is C. J. Date's *SQL and Relational Theory* (Date, 2009).

Relational DBMSs aren't specific to Java, nor is an SQL database specific to a particular application. This important principle is known as *data independence*. In other words, and we can't stress this important fact enough, *data lives longer than any application does*. Relational technology provides a way of sharing data among different applications, or among different parts of the same overall system (the data entry application and the reporting application, for example). Relational technology is a common denominator of many disparate systems and technology platforms. Hence, the relational data model is often the foundation for the common enterprise-wide representation of business entities.

Before we go into more detail about the practical aspects of SQL databases, we have to mention an important issue: although marketed as relational, a database system providing only an SQL data language interface isn't really relational and in many ways isn't even close to the original concept. Naturally, this has led to confusion. SQL practitioners blame the relational data model for shortcomings in the SQL language, and relational data management experts blame the SQL standard for being a weak implementation of the relational model and ideals. Application engineers are stuck somewhere in the middle, with the burden of delivering something that works. We highlight some important and significant aspects of this issue throughout this book, but generally we focus on the practical aspects. If you're interested in more background material, we highly recommend *Practical Issues in Database Management: A Reference for the Thinking Practitioner* by Fabian Pascal (Pascal, 2000) and *An Introduction to Database Systems* by Chris Date (Date, 2003) for the theory, concepts, and ideals of (relational) database systems. The latter book is an excellent reference (it's big) for all questions you may possibly have about databases and data management.

1.1.2 Understanding SQL

To use Hibernate effectively, you must start with a solid understanding of the relational model and SQL. You need to understand the relational model and topics such as normalization to guarantee the integrity of your data, and you'll need to use your knowledge of SQL to tune the performance of your Hibernate application. Hibernate automates many repetitive coding tasks, but your knowledge of persistence technology must extend beyond Hibernate itself if you want to take advantage of the full power of modern SQL databases. To dig deeper, consult the bibliography at the end of this book.

You've probably used SQL for many years and are familiar with the basic operations and statements written in this language. Still, we know from our own experience that SQL is sometimes hard to remember, and some terms vary in usage.

Let's review some of the SQL terms used in this book. You use SQL as a *data definition language* (DDL) when *creating*, *altering*, and *dropping* artifacts such as tables and constraints in the catalog of the DBMS. When this *schema* is ready, you use SQL as a *data manipulation language* (DML) to perform operations on data, including *insertions*, *updates*, and *deletions*. You retrieve data by executing queries with *restrictions*, *projections*, and *Cartesian products*. For efficient reporting, you use SQL to *join*, *aggregate*, and *group* data as necessary. You can even nest SQL statements inside each other—a technique that uses *subselects*. When your business requirements change, you'll have to modify the database schema again with DDL statements after data has been stored; this is known as *schema evolution*.

If you're an SQL veteran and you want to know more about optimization and how SQL is executed, get a copy of the excellent book *SQL Tuning*, by Dan Tow (Tow, 2003). For a look at the practical side of SQL through the lens of how not to use SQL, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (Karwin, 2010) is a good resource.

Although the SQL database is one part of ORM, the other part, of course, consists of the data in your Java application that needs to be persisted to and loaded from the database.

1.1.3 Using SQL in Java

When you work with an SQL database in a Java application, you issue SQL statements to the database via the Java Database Connectivity (JDBC) API. Whether the SQL was written by hand and embedded in the Java code or generated on the fly by Java code, you use the JDBC API to bind arguments when preparing query parameters, executing the query, scrolling through the query result, retrieving values from the result set, and so on. These are low-level data access tasks; as application engineers, we're more interested in the business problem that requires this data access. What we'd really like to write is code that saves and retrieves instances of our classes, relieving us of this low-level drudgery.

Because these data access tasks are often so tedious, we have to ask, are the relational data model and (especially) SQL the right choices for persistence in object-oriented applications? We answer this question unequivocally: yes! There are many reasons why SQL databases dominate the computing industry—relational database management systems are the only proven generic data management technology, and they're almost always a *requirement* in Java projects.

Note that we aren't claiming that relational technology is *always* the best solution. There are many data management requirements that warrant a completely different approach. For example, internet-scale distributed systems (web search engines, content distribution networks, peer-to-peer sharing, instant messaging) have to deal with exceptional transaction volumes. Many of these systems don't require that after a data update completes, all processes see the same updated data (strong transactional consistency). Users might be happy with weak consistency; after an update, there might be a window of inconsistency before all processes see the updated data. Some scientific applications work with enormous but very specialized datasets. Such systems and their unique challenges typically require equally unique and often custom-made persistence solutions. Generic data management tools such as ACID-compliant transactional SQL databases, JDBC, and Hibernate would play only a minor role.

Relational systems at internet scale

To understand why relational systems, and the data-integrity guarantees associated with them, are difficult to scale, we recommend that you first familiarize yourself with the *CAP theorem*. According to this rule, a distributed system can't be *consistent*, *available*, and *tolerant against partition failures* all at the same time.

A system may guarantee that all nodes will see the same data at the same time and that data read and write requests are always answered. But when a part of the system fails due to a host, network, or data center problem, you must either give up strong consistency (linearizability) or 100% availability. In practice, this means you need a strategy that detects partition failures and restores either consistency or availability to a certain degree (for example, by making some part of the system temporarily unavailable for data synchronization to occur in the background). Often it depends on the data, the user, or the operation whether strong consistency is necessary.

For relational DBMSs designed to scale easily, have a look at VoltDB (www.voltdb.com) and NuoDB (www.nuodb.com). Another interesting read is how Google scales its most important database, for the advertising business, and why it's relational/SQL, in "F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business" (Shute, 2012).

In this book, we'll think of the problems of data storage and sharing in the context of an object-oriented application that uses a *domain model*. Instead of directly working with the rows and columns of a `java.sql.ResultSet`, the business logic of an application interacts with the application-specific object-oriented domain model. If the SQL

database schema of an online auction system has `ITEM` and `BID` tables, for example, the Java application defines `Item` and `Bid` classes. Instead of reading and writing the value of a particular row and column with the `ResultSet` API, the application loads and stores instances of `Item` and `Bid` classes.

At runtime, the application therefore operates with instances of these classes. Each instance of a `Bid` has a reference to an auction `Item`, and each `Item` may have a collection of references to `Bid` instances. The business logic isn't executed in the database (as an SQL stored procedure); it's implemented in Java and executed in the application tier. This allows business logic to use sophisticated object-oriented concepts such as inheritance and polymorphism. For example, we could use well-known design patterns such as *Strategy*, *Mediator*, and *Composite* (see *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma, 1995]), all of which depend on polymorphic method calls.

Now a caveat: not all Java applications are designed this way, nor should they be. Simple applications may be much better off without a domain model. Use the JDBC `ResultSet` if that's all you need. Call existing stored procedures, and read their SQL result sets, too. Many applications need to execute procedures that modify large sets of data, close to the data. You might implement some reporting functionality with plain SQL queries and render the result directly onscreen. SQL and the JDBC API are perfectly serviceable for dealing with tabular data representations, and the JDBC `RowSet` makes CRUD operations even easier. Working with such a representation of persistent data is straightforward and well understood.

But in the case of applications with nontrivial business logic, the domain model approach helps to improve code reuse and maintainability significantly. In practice, *both* strategies are common and needed.

For several decades, developers have spoken of a *paradigm mismatch*. This mismatch explains why every enterprise project expends so much effort on persistence-related concerns. The *paradigms* referred to are object modeling and relational modeling, or, more practically, object-oriented programming and SQL.

With this realization, you can begin to see the problems—some well understood and some less well understood—that an application that combines both data representations must solve: an object-oriented domain model and a persistent relational model. Let's take a closer look at this so-called paradigm mismatch.

1.2 The paradigm mismatch

The object/relational paradigm mismatch can be broken into several parts, which we examine one at a time. Let's start our exploration with a simple example that is problem free. As we build on it, you'll see the mismatch begin to appear.

Suppose you have to design and implement an online e-commerce application. In this application, you need a class to represent information about a user of the system, and you need another class to represent information about the user's billing details, as shown in figure 1.1.

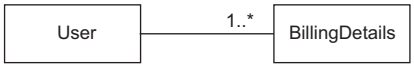

Figure 1.1 A simple UML diagram of the `User` and `BillingDetails` entities

In this diagram, you can see that a `User` has many `BillingDetails`. You can navigate the relationship between the classes in both directions; this means you can iterate through collections or call methods to get to the "other" side of the relationship. The classes representing these entities may be extremely simple:

```
public class User {

    String username;
    String address;
    Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
}

public class BillingDetails {

    String account;
    String bankname;
    User user;

    // Accessor  methods (getter/setter), business methods, etc.
}
```

Note that you're only interested in the state of the entities with regard to persistence, so we've omitted the implementation of property accessors and business methods, such as `getUsername()` or `billAuction()`.

It's easy to come up with an SQL schema design for this case:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS varchar(255) not null
);

create table BILLINGDETAILS (
    ACCOUNT varchar(15) not null primary key,
    BANKNAME varchar(255) not null,
    USERNAME varchar(15) not null,
    foreign key (USERNAME) references USERS
);
```

The foreign key–constrained column `USERNAME` in `BILLINGDETAILS` represents the relationship between the two entities. For this simple domain model, the object/relational mismatch is barely in evidence; it's straightforward to write JDBC code to insert, update, and delete information about users and billing details.

Now let's see what happens when you consider something a little more realistic. The paradigm mismatch will be visible when you add more entities and entity relationships to your application.

1.2.1 The problem of granularity

The most glaringly obvious problem with the current implementation is that you've designed an address as a simple `String` value. In most systems, it's necessary to store street, city, state, country, and ZIP code information separately. Of course, you could add these properties directly to the `User` class, but because it's highly likely that other classes in the system will also carry address information, it makes more sense to create an `Address` class. Figure 1.2 shows the updated model.

Figure 1.2 The `User` has an `Address`.

Should you also add an `ADDRESS` table? Not necessarily; it's common to keep address information in the `USERS` table, in individual columns. This design is likely to perform better, because a table join isn't needed if you want to retrieve the user and address in a single query. The nicest solution may be to create a new SQL data type to represent addresses, and to add a single column of that new type in the `USERS` table instead of several new columns.

You have the choice of adding either several columns or a single column (of a new SQL data type). This is clearly a problem of *granularity*. Broadly speaking, granularity refers to the relative size of the types you're working with.

Let's return to the example. Adding a new data type to the database catalog, to store `Address` Java instances in a single column, sounds like the best approach:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS address not null
);
```

A new `Address` type (class) in Java and a new `ADDRESS` SQL data type should guarantee interoperability. But you'll find various problems if you check the support for user-defined data types (UDTs) in today's SQL database management systems.

UDT support is one of a number of so-called *object-relational extensions* to traditional SQL. This term alone is confusing, because it means the database management system has (or is supposed to support) a sophisticated data type system—something you take for granted if somebody sells you a system that can handle data in a relational fashion. Unfortunately, UDT support is a somewhat obscure feature of most SQL DBMSs and certainly isn't portable between different products. Furthermore, the SQL standard supports user-defined data types, but poorly.

This limitation isn't the fault of the relational data model. You can consider the failure to standardize such an important piece of functionality as fallout from the object-relational database wars between vendors in the mid-1990s. Today, most engineers accept that SQL products have limited type systems—no questions asked. Even with a sophisticated UDT system in your SQL DBMS, you would still likely duplicate the type declarations, writing the new type in Java and again in SQL. Attempts to find a

better solution for the Java space, such as SQLJ, unfortunately, have not had much success. DBMS products rarely support deploying and executing Java classes directly on the database, and if support is available, it's typically limited to very basic functionality and complex in everyday usage.

For these and whatever other reasons, use of UDTs or Java types in an SQL database isn't common practice in the industry at this time, and it's unlikely that you'll encounter a legacy schema that makes extensive use of UDTs. You therefore can't and won't store instances of your new `Address` class in a single new column that has the same data type as the Java layer.

The pragmatic solution for this problem has several columns of built-in vendor-defined SQL types (such as Boolean, numeric, and string data types). You usually define the `USERS` table as follows:

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    ADDRESS_STREET varchar(255) not null,
    ADDRESS_ZIPCODE varchar(5) not null,
    ADDRESS_CITY varchar(255) not null
);
```

Classes in the Java domain model come in a range of different levels of granularity: from coarse-grained entity classes like `User`, to finer-grained classes like `Address`, down to simple `SwissZipCode` extending `AbstractNumericZipCode` (or whatever your desired level of abstraction is). In contrast, just two levels of type granularity are visible in the SQL database: relation types created by you, like `USERS` and `BILLINGDETAILS`, and built-in data types such as `VARCHAR`, `BIGINT`, or `TIMESTAMP`.

Many simple persistence mechanisms fail to recognize this mismatch and so end up forcing the less flexible representation of SQL products on the object-oriented model, effectively flattening it.

It turns out that the granularity problem isn't especially difficult to solve. We probably wouldn't even discuss it, were it not for the fact that it's visible in so many existing systems. We describe the solution to this problem in section 4.1.

A much more difficult and interesting problem arises when we consider domain models that rely on *inheritance*, a feature of object-oriented design you may use to bill the users of your e-commerce application in new and interesting ways.

1.2.2 The problem of subtypes

In Java, you implement type inheritance using superclasses and subclasses. To illustrate why this can present a mismatch problem, let's add to your e-commerce application so that you now can accept not only bank account billing, but also credit and debit cards. The most natural way to reflect this change in the model is to use inheritance for the `BillingDetails` superclass, along with several concrete subclasses: `CreditCard`, `BankAccount`, and so on. Each of these subclasses defines slightly different data (and completely different functionality that acts on that data). The UML class diagram in figure 1.3 illustrates this model.

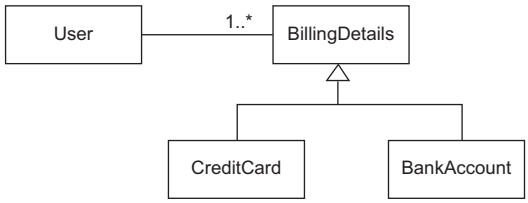

Figure 1.3 Using inheritance for different billing strategies

What changes must you make to support this updated Java class structure? Can you create a table `CREDITCARD` that *extends* `BILLINGDETAILS`? SQL database products don't generally implement table inheritance (or even data type inheritance), and if they do implement it, they don't follow a standard syntax and might expose us to data integrity problems (limited integrity rules for updatable views).

We aren't finished with inheritance. As soon as we introduce inheritance into the model, we have the possibility of *polymorphism*.

The `User` class has an association to the `BillingDetails` superclass. This is a *polymorphic association*. At runtime, a `User` instance may reference an instance of any of the subclasses of `BillingDetails`. Similarly, you want to be able to write *polymorphic queries* that refer to the `BillingDetails` class, and have the query return instances of its subclasses.

SQL databases also lack an obvious way (or at least a standardized way) to represent a polymorphic association. A foreign key constraint refers to exactly one target table; it isn't straightforward to define a foreign key that refers to multiple tables. You'd have to write a procedural constraint to enforce this kind of integrity rule.

The result of this mismatch of subtypes is that the inheritance structure in a model must be persisted in an SQL database that doesn't offer an inheritance mechanism. In chapter 6, we discuss how ORM solutions such as Hibernate solve the problem of persisting a class hierarchy to an SQL database table or tables, and how polymorphic behavior can be implemented. Fortunately, this problem is now well understood in the community, and most solutions support approximately the same functionality.

The next aspect of the object/relational mismatch problem is the issue of *object identity*. You probably noticed that the example defined `USERNAME` as the primary key of the `USERS` table. Was that a good choice? How do you handle identical objects in Java?

1.2.3 The problem of identity

Although the problem of identity may not be obvious at first, you'll encounter it often in your growing and expanding e-commerce system, such as when you need to check whether two instances are identical. There are three ways to tackle this problem: two in the Java world and one in your SQL database. As expected, they work together only with some help.

Java defines two different notions of *sameness*:

- Instance identity (roughly equivalent to memory location, checked with `a == b`)
- Instance equality, as determined by the implementation of the `equals()` method (also called *equality by value*)

On the other hand, the identity of a database row is expressed as a comparison of primary key values. As you'll see in section 10.1.2, neither `equals()` nor `==` is always equivalent to a comparison of primary key values. It's common for several non-identical instances in Java to simultaneously represent the same row of the database—for example, in concurrently running application threads. Furthermore, some subtle difficulties are involved in implementing `equals()` correctly for a persistent class and understanding when this might be necessary.

Let's use an example to discuss another problem related to database identity. In the table definition for `USERS`, `USERNAME` is the primary key. Unfortunately, this decision makes it difficult to change a user's name; you need to update not only the row in `USERS`, but also the foreign key values in (many) rows of `BILLINGDETAILS`. To solve this problem, later in this book we recommend that you use *surrogate keys* whenever you can't find a good natural key. We also discuss what makes a good primary key. A surrogate key column is a primary key column with no meaning to the application user—in other words, a key that isn't presented to the application user. Its only purpose is identifying data inside the application.

For example, you may change your table definitions to look like this:

```
create table USERS (
    ID bigint not null primary key,
    USERNAME varchar(15) not null unique,
    ...
);

create table BILLINGDETAILS (
    ID bigint not null primary key,
    ACCOUNT varchar(15) not null,
    BANKNAME varchar(255) not null,
    USER_ID bigint not null,
    foreign key (USER_ID) references USERS
);
```

The `ID` columns contain system-generated values. These columns were introduced purely for the benefit of the data model, so how (if at all) should they be represented in the Java domain model? We discuss this question in section 4.2, and we find a solution with ORM.

In the context of persistence, identity is closely related to how the system handles caching and transactions. Different persistence solutions have chosen different strategies, and this has been an area of confusion. We cover all these interesting topics—and show how they're related—in section 10.1.

So far, the skeleton e-commerce application you've designed has exposed the paradigm mismatch problems with mapping granularity, subtypes, and identity. You're almost ready to move on to other parts of the application, but first we need to discuss the important concept of *associations*: how the relationships between entities are mapped and handled. Is the foreign key constraint in the database all you need?

1.2.4 Problems relating to associations

In your domain model, associations represent the relationships between entities. The `User`, `Address`, and `BillingDetails` classes are all associated; but unlike `Address`, `BillingDetails` stands on its own. `BillingDetails` instances are stored in their own table. Association mapping and the management of entity associations are central concepts in any object persistence solution.

Object-oriented languages represent associations using *object references*; but in the relational world, a *foreign key–constrained column* represents an association, with copies of key values. The constraint is a rule that guarantees integrity of the association. There are substantial differences between the two mechanisms.

Object references are inherently directional; the association is from one instance to the other. They're pointers. If an association between instances should be navigable in both directions, you must define the association *twice*, once in each of the associated classes. You've already seen this in the domain model classes:

```
public class User {
    Set billingDetails;
}

public class BillingDetails {
    User user;
}
```

Navigation in a particular direction has no meaning for a relational data model because you can create arbitrary data associations with *join* and *projection* operators. The challenge is to map a completely open data model, which is independent of the application that works with the data, to an application-dependent navigational model—a constrained view of the associations needed by this particular application.

Java associations can have *many-to-many* multiplicity. For example, the classes could look like this:

```
public class User {
    Set billingDetails;
}

public class BillingDetails {
    Set users;
}
```

But the foreign key declaration on the `BILLINGDETAILS` table is a *many-to-one* association: each bank account is linked to a particular user. Each user may have multiple linked bank accounts.

If you wish to represent a *many-to-many* association in an SQL database, you must introduce a new table, usually called a *link table*. In most cases, this table doesn't appear anywhere in the domain model. For this example, if you consider the relationship between the user and the billing information to be *many-to-many*, you define the link table as follows:

```
create table USER_BILLINGDETAILS (
    USER_ID bigint,
    BILLINGDETAILS_ID bigint,
    primary key (USER_ID, BILLINGDETAILS_ID),
    foreign key (USER_ID) references USERS,
    foreign key (BILLINGDETAILS_ID) references BILLINGDETAILS
);
```

You no longer need the `USER_ID` foreign key column and constraint on the `BILLINGDETAILS` table; this additional table now manages the links between the two entities. We discuss association and collection mappings in detail in chapter 7.

So far, the issues we've considered are mainly *structural*: you can see them by considering a purely static view of the system. Perhaps the most difficult problem in object persistence is a *dynamic* problem: how data is accessed at runtime.

1.2.5 The problem of data navigation

There is a fundamental difference in how you access data in Java and in a relational database. In Java, when you access a user's billing information, you call `someUser.getBillingDetails().iterator().next()` or something similar. This is the most natural way to access object-oriented data, and it's often described as *walking the object network*. You navigate from one instance to another, even iterating collections, following prepared pointers between classes. Unfortunately, this isn't an efficient way to retrieve data from an SQL database.

The single most important thing you can do to improve the performance of data access code is to *minimize the number of requests to the database*. The most obvious way to do this is to minimize the number of SQL queries. (Of course, other, more sophisticated, ways—such as extensive caching—follow as a second step.)

Therefore, efficient access to relational data with SQL usually requires joins between the tables of interest. The number of tables included in the join when retrieving data determines the depth of the object network you can navigate in memory. For example, if you need to retrieve a `User` and aren't interested in the user's billing information, you can write this simple query:

```
select * from USERS u where u.ID = 123
```

On the other hand, if you need to retrieve a `User` and then subsequently visit each of the associated `BillingDetails` instances (let's say, to list all the user's bank accounts), you write a different query:

```
select * from USERS u
    left outer join BILLINGDETAILS bd
        on bd.USER_ID = u.ID
where u.ID = 123
```

As you can see, to use joins efficiently you need to know what portion of the object network you plan to access when you retrieve the initial instance *before* you start navigating the object network! Careful, though: if you retrieve too much data (probably

more than you might need), you're wasting memory in the application tier. You may also overwhelm the SQL database with huge *Cartesian product* result sets. Imagine retrieving not only users and bank accounts in one query, but also all orders paid from each bank account, the products in each order, and so on.

Any object persistence solution worth its salt provides functionality for fetching the data of associated instances only when the association is first accessed in Java code. This is known as *lazy loading*: retrieving data on demand only. This piecemeal style of data access is fundamentally inefficient in the context of an SQL database, because it requires executing one statement for each node or collection of the object network that is accessed. This is the dreaded *n+1 selects* problem.

This mismatch in the way you access data in Java and in a relational database is perhaps the single most common source of performance problems in Java information systems. Yet although we've been blessed with innumerable books and articles advising us to use `StringBuffer` for string concatenation, avoiding the *Cartesian product* and *n+1 selects* problems is still a mystery for many Java programmers. (Admit it: you just thought `StringBuilder` would be much better than `StringBuffer`.)

Hibernate provides sophisticated features for efficiently and transparently fetching networks of objects from the database to the application accessing them. We discuss these features in chapter 12.

We now have quite a list of object/relational mismatch problems, and it can be costly (in time and effort) to find solutions, as you may know from experience. It will take us most of this book to provide a complete answer to these questions and to demonstrate ORM as a viable solution. Let's get started with an overview of ORM, the Java Persistence standard, and the Hibernate project.

1.3 ORM and JPA

In a nutshell, object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in an SQL database, using metadata that describes the mapping between the classes of the application and the schema of the SQL database. In essence, ORM works by transforming (reversibly) data from one representation to another. Before we move on, you need to understand what Hibernate *can't* do for you.

A supposed advantage of ORM is that it shields developers from messy SQL. This view holds that object-oriented developers can't be expected to understand SQL or relational databases well and that they find SQL somehow offensive. On the contrary, we believe that Java developers must have a sufficient level of familiarity with—and appreciation of—relational modeling and SQL in order to work with Hibernate. ORM is an advanced technique used by developers who have already done it the hard way. To use Hibernate effectively, you must be able to view and interpret the SQL statements it issues and understand their performance implications.

Let's look at some of the benefits of Hibernate:

- *Productivity*—Hibernate eliminates much of the grunt work (more than you'd expect) and lets you concentrate on the business problem. No matter which application-development strategy you prefer—top-down, starting with a domain model, or bottom-up, starting with an existing database schema—Hibernate, used together with the appropriate tools, will significantly *reduce development time.*
- *Maintainability*—Automated ORM with Hibernate reduces lines of code (LOC), making the system *more understandable* and *easier to refactor.* Hibernate provides a buffer between the domain model and the SQL schema, insulating each model from minor changes to the other.
- *Performance*—Although hand-coded persistence might be faster in the same sense that assembly code can be faster than Java code, automated solutions like Hibernate allow the use of many optimizations *at all times.* One example of this is efficient and easily tunable caching in the application tier. This means developers can spend more energy hand-optimizing the few remaining real bottlenecks instead of prematurely optimizing everything.
- *Vendor independence*—Hibernate can help mitigate some of the risks associated with vendor lock-in. Even if you plan never to change your DBMS product, ORM tools that support a number of different DBMSs enable *a certain level of portability.* In addition, DBMS independence helps in development scenarios where *engineers use a lightweight local database* but deploy for testing and production on a different system.

The Hibernate approach to persistence was well received by Java developers, and the standard Java Persistence API was designed along similar lines.

JPA became a key part of the simplifications introduced in recent EJB and Java EE specifications. We should be clear up front that neither Java Persistence nor Hibernate are limited to the Java EE environment; they're general-purpose solutions to the persistence problem that any type of Java (or Groovy, or Scala) application can use.

The JPA specification defines the following:

- A facility for specifying mapping metadata—how persistent classes and their properties relate to the database schema. JPA relies heavily on Java annotations in domain model classes, but you can also write mappings in XML files.
- APIs for performing basic CRUD operations on instances of persistent classes, most prominently `javax.persistence.EntityManager` to store and load data.
- A language and APIs for specifying queries that refer to classes and properties of classes. This language is the Java Persistence Query Language (JPQL) and looks similar to SQL. The standardized API allows for programmatic creation of *criteria queries* without string manipulation.
- How the persistence engine interacts with transactional instances to perform dirty checking, association fetching, and other optimization functions. The latest JPA specification covers some basic caching strategies.

Hibernate implements JPA and supports all the standardized mappings, queries, and programming interfaces.

1.4 Summary

- With *object persistence*, individual objects can outlive their application process, be saved to a data store, and be re-created later. The object/relational mismatch comes into play when the data store is an SQL-based relational database management system. For instance, a network of objects can't be saved to a database table; it must be disassembled and persisted to columns of portable SQL data types. A good solution for this problem is object/relational mapping (ORM).
- ORM isn't a silver bullet for all persistence tasks; its job is to relieve the developer of 95% of object persistence work, such as writing complex SQL statements with many table joins and copying values from JDBC result sets to objects or graphs of objects.
- A full-featured ORM middleware solution may provide database portability, certain optimization techniques like caching, and other viable functions that aren't easy to hand-code in a limited time with SQL and JDBC.
- Better solutions than ORM might exist someday. We (and many others) may have to rethink everything we know about data management systems and their languages, persistence API standards, and application integration. But the evolution of today's systems into true relational database systems with seamless object-oriented integration remains pure speculation. We can't wait, and there is no sign that any of these issues will improve soon (a multibillion-dollar industry isn't very agile). ORM is the best solution currently available, and it's a timesaver for developers facing the object/relational mismatch every day.