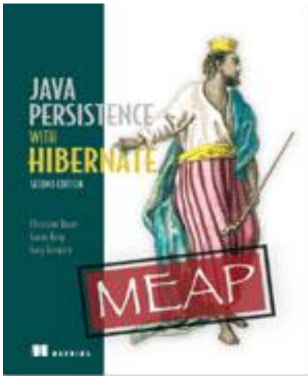
What is persistence?

By Christian Bauer, Gavin King, and Gary Gregory, Java Persistence with Hibernate



When we talk about persistence in Java, we're normally talking about storing data in a database using SQL. In this article, we'll take a brief look at the technology and how we use it with Java.

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. When we talk about persistence in Java, we're normally talking about storing data in a database using SQL. We'll take a brief look at the technology and how we use it with Java.

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. Most of you will be familiar with E.F. Codd's four-decades-old introduction of the relational model.

Relational DBMS's aren't specific to Java, nor is an actual 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.

Data Independence

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 to deliver something that works. 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 and *An Introduction to Database Systems* by Chris Date 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.
NoSQL

Understanding SQL

To use Hibernate effectively, a solid understanding of the relational model and SQL is a prerequisite. 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.

Let's look at some SQL terms. 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 manipulate and retrieve data. The manipulation operations include *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 will have to modify the database schema again with DDL statements after data has been stored, this is known as *schema evolution*.

If you are 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. 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* 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.

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, execute the query, scroll through the query result, and retrieve 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 are not claiming that relational technology is *always* the best solution. There are indeed many data management requirements that would 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 data sets. 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 only play 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 you first familiarize yourself with the *CAP theorem*. According to this rule, a distributed system cannot be *Consistent*, *Available*, and tolerant against *Partition failures* all at the same time. Your 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. However, when a part of your system fails, caused by 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 on whether strong consistency is even necessary. For relational DBMS designed to scale easily, have a look at VoltDB and NuoDB.

When developing with Hibernate, we are thinking 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 make use of sophisticated object-oriented concepts such as inheritance and polymorphism. For example, we could use well-known design patterns such as *Strategy*, *Mediator*, and *Composite*, 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 on screen. 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.

However, 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.