Sharding isn't for everyone, but it's one way that relational systems can meet the demands of big data. For some shops, sharding means being able to keep a trusted RDBMS in place without sacrificing data scalability or system performance. In this installment of the Java development 2.0 series, find out when sharding works, and when it doesn't, and then get your hands busy sharding a simple application capable of handling terabytes of data.

When relational databases attempt to store terabytes of data in single tables, overall performance typically degrades. Indexing all that data is obviously expensive for reads, but also for writes. While NoSQL datastores are particularly suited to storing big data (think Google's Bigtable), NoSQL is a patently non-relational approach. For the developer who prefers the ACID-ity and solid structure of a relational database, or the project that requires it, sharding could be an exciting alternative.

Sharding, an offshoot of database partitioning, isn't a native database technique — it happens at the level of the application. Among various sharding implementations, Hibernate Shards is possibly the most popular in the world of Java™ technology. This nifty project lets you work more or less seamlessly with sharded datasets (I will explain the "more or less" part shortly) using POJOs that are mapped to a logical database. When you use Hibernate Shards, you don't have to specifically map your POJOs to shards — you map them as you would any normal relational database in the Hibernate way. Hibernate Shards manages the low-level sharding stuff for you.

So far in this series, I've used a simple domain based on the analogy of races and runners to demonstrate various data storage technologies. This month, I'll use this familiar example to introduce a practical sharding strategy, then implement it in Hibernate Shards. Note that the brunt of the work related to sharding isn't necessarily related to Hibernate; in fact, coding for Hibernate Shards is the easy part. The real work is figuring out how and what you'll shard.
About this series
The Java development landscape has changed radically since Java technology first emerged. Thanks to mature open source frameworks and reliable for-rent deployment infrastructures, it's now possible to assemble, test, run, and maintain Java applications quickly and inexpensively. In this series, Andrew Glover explores the spectrum of technologies and tools that make this new Java development paradigm possible.

**Sharding at a glance**

*Database partitioning* is an inherently relational process of dividing a table's rows by some logical piece of data into smaller groups. If you were partitioning a gigantic table named `foo` based on timestamps, for instance, all the data for August 2010 would go in Partition A, while anything since then would be in Partition B, and so on. Partitioning has the effect of making reads and writes faster because they target smaller datasets in individual partitions.

Partitioning isn't always available (MySQL didn't support it until version 5.1), and the cost of doing it with a commercial system can be prohibitive. What's more, most partitioning implementations store data on the same physical machine, so you're still bound to the limits of your hardware. Partitioning also doesn't resolve the reliability, or lack thereof, of your hardware. Thus, various smart people started looking for new ways to scale.

*Sharding* is essentially partitioning at the database level: rather than divide a table's rows by pieces of data, the database itself is split up (usually across different machines) by some logical data element. That is, rather than splitting up a *table* into smaller chunks, sharding splits up an *entire database* into smaller chunks.

The canonical example for sharding is based on dividing a large database storing worldwide customer data by region: Shard A for customers in the United States, Shard B for Asia, Shard C for Europe, and so on. The shards themselves would live on different machines and each shard would hold all related data, such as customer preferences or order history.

The benefit of sharding (like partitioning) is that it compacts big data: individual tables are smaller in each shard, which allows for faster reads and writes, which increases performance. Sharding also conceivably improves reliability, because even if one shard unexpectedly fails, others are still able to serve data. And because sharding is done at the application layer, you can do it for databases that don't support regular partitioning. The monetary cost is also potentially lower.

**Sharding and strategy**

Like most technologies, sharding does entail some trade-offs. Because sharding isn't a native database technique — that is, you must implement it in your application — you'll need to map out your sharding strategy before you begin. Both primary keys and cross-shard queries play a major role when sharding, mainly by defining what you can't do.

**Primary keys**

Sharding leverages multiple databases, all of which function autonomously, without awareness of their peers. As a result, if you rely on database sequences (such as for automatic primary key generation), it's likely that an identical primary key will show up across a set of databases. It's
possible to coordinate sequences across a distributed database but doing so increases system complexity. The safest way to prohibit duplicate primary keys is to have your application (which will be managing a sharded system anyway) generate keys.

**Cross-shard queries**
Most sharding implementations (including Hibernate Shards) don't permit cross-shard querying, which means you have to go to extra lengths if you want to leverage two sets of data from different shards. (Interestingly, Amazon's SimpleDB also prohibits cross-domain queries.) For instance, if you're storing United States customers in Shard 1, you also need to store all of their related data there. If you try to store that data in Shard 2, things will get complicated, and system performance will probably suffer. This situation is also related to the point made earlier — if you somehow end up needing to do cross-shard joins, you had better be managing keys in a way that eliminates the possibility of duplicates!

Clearly, you'll need to fully consider a sharding strategy before you set up your database. And once you've chosen a particular direction, you're more or less tied to it — it's hard to move data around after it's been sharded.

---

**Avoid premature sharding**
Sharding is best employed late in the game. Like premature optimization, sharding based on expected data growth could be a recipe for disaster. Successful sharding implementations are based on measurably understanding an application's data growth over time, and then extrapolating to the future. Once you've sharded your data it can be extraordinarily hard to move around.

---

**A strategy example**
Because sharding binds you to a linear data model (that is, you can't easily join data in different shards), you should start with a clear picture of how your data will be logically organized per shard. This is usually easiest by focusing on the primary node of a domain. In the case of an e-commerce system, the primary node could be either an order or a customer. Thus, if you choose "customer" as the basis for your sharding strategy, then all data related to customers will be moved into the respective shards, though you'll still have to choose to which shard to move that data.

For customers, you could shard based on location (Europe, Asia, Africa, etc.), or you could shard based on something else. It's up to you. Your shard strategy should, however, incorporate some means of distributing data evenly among all of your shards. The whole idea of sharding is to break up big data sets into smaller ones; thus, if a particular e-commerce domain had a large set of European customers and relatively few in the United States, it probably wouldn't make sense to shard based on customer location.

**Off to the races — with sharding!**
Getting back to the familiar example of my racing application, I can shard by race or by runner. In this case, I'm going to shard by race, because I see the domain being organized by runners who belong to races. So the race is the root of my domain. I'm also going to shard based on race distance, because my racing application holds myriad races of different lengths, along with myriad runners.
Note that in making these decisions, I have already accepted a trade-off: what if a runner participates in more than one race, each of them living in different shards? Hibernate Shards (like most sharding implementations) doesn't support cross-shard joins. I'm going to have to live with this slight inconvenience and allow runners to live in multiple shards — that is, I will recreate each runner in the shards where his or her various races live.

To keep things simple, I'm going to create two shards: one for races less than 10 miles and another for anything greater than 10 miles.

**Implementing Hibernate Shards**

Hibernate Shards is made to work almost seamlessly with existing Hibernate projects. The only catch is that Hibernate Shards needs some specific information and behavior from you. Namely, it needs a shard-access strategy, a shard-selection strategy, and a shard-resolution strategy. These are interfaces you must implement, though in some cases you can use default ones. We'll look at each interface separately in the following sections.

**ShardAccessStrategy**

When a query is executed, Hibernate Shards needs a mechanism for determining which shard to hit first, second, and so on. Hibernate Shards doesn't necessarily figure out what a query is looking for (that's for the Hibernate Core and underlying database to do), but it does recognize that a query might need to execute against multiple shards before an answer is obtained. So, Hibernate Shards provides two logical implementations out of the box: one executes a query in a sequential mechanism (one at a time) against shards until an answer is returned, or until all of the shards have been queried. The other implementation is a parallel-access strategy, which uses a threading model to hit all of the shards at once.

I'm going to keep things simple and utilize the sequential strategy, aptly named `SequentialShardAccessStrategy`. We'll configure it shortly.

**ShardSelectionStrategy**

When a new object is created (that is, when a new Race or Runner is created via Hibernate), Hibernate Shards needs to know what shard the corresponding data should be written to. Accordingly, you must implement this interface and code the sharding logic. If you want a default implementation, there's one dubbed `RoundRobinShardSelectionStrategy`, which uses a round-robin strategy for putting data into shards.

For the racing application, I need to provide behavior that shards by race distance. Accordingly, I'll need to implement the `ShardSelectionStrategy` interface and provide some simple logic that shards based on a Race object's distance in the `selectShardIdForNewObject` method. (I'll show the Race object shortly.)

At runtime, when a call is made to some `save`-like method on my domain objects, this interface's behavior is leveraged deep down in Hibernate's core.
Listing 1. A simple shard-selection strategy

```java
import org.hibernate.shards.ShardId;
import org.hibernate.shards.strategy.selection.ShardSelectionStrategy;

public class RacerShardSelectionStrategy implements ShardSelectionStrategy {
    public ShardId selectShardIdForNewObject(Object obj) {
        if (obj instanceof Race) {
            Race rce = (Race) obj;
            return this.determineShardId(rce.getDistance());
        } else if (obj instanceof Runner) {
            Runner runnr = (Runner) obj;
            if (runnr.getRaces().isEmpty()) {
                throw new IllegalArgumentException("runners must have at least one race");
            } else {
                double dist = 0.0;
                for (Race rce : runnr.getRaces()) {
                    dist = rce.getDistance();
                    break;
                }
                return this.determineShardId(dist);
            }
        } else {
            throw new IllegalArgumentException("a non-shardable object is being created");
        }
    }

    private ShardId determineShardId(double distance) {
        if (distance > 10.0) {
            return new ShardId(1);
        } else {
            return new ShardId(0);
        }
    }
}
```

As you can see in Listing 1, if the object being persisted is a `Race`, then its distance is determined and, accordingly, a shard is picked. In this case, there are two shards: 0 and 1, where Shard 1 holds races with a distance greater than 10 miles and Shard 0 holds all others.

If a `Runner` or some other object is being persisted, things get a bit more involved. I've coded a logical rule that has three stipulations:

- A `Runner` can't exist without a corresponding `Race`.
- If a `Runner` has been created with multiple `Races`, the `Runner` will be persisted in the shard for the first `Race` found. (This rule has negative implications for the future, by the way.)
- If some other domain object is being saved, for now, an exception will be thrown.

With that, you can wipe the sweat from your brow, because most of the hard work is done. The logic I've captured might not be flexible enough as the racing application grows, but it'll work for the purpose of this demonstration!

**ShardResolutionStrategy**

When searching for an object by its key, Hibernate Shards needs a way of determining which shard to hit first. You'll use the `ShardResolutionStrategy` interface to guide it.
As I mentioned earlier, sharding forces you to be keenly aware of primary keys, as you'll manage them yourself. Luckily, Hibernate is already good at providing key or UUID generation. Consequently, out of the box, Hibernate Shards provides an ID generator dubbed ShardedUUIDGenerator, which has the smarts to embed shard ID information in the UUID itself.

If you end up using ShardedUUIDGenerator for key generation (as I will for this article), then you can also use the Hibernate Shards out-of-the-box ShardResolutionStrategy implementation dubbed AllShardsShardResolutionStrategy, which can determine what shard to search based on a particular object's ID.

Having configured the three interfaces required for Hibernate Shards to work properly, we're ready for the next step in sharding the example application. It's time to launch Hibernate's SessionFactory.

Configuring Hibernate Shards

One of Hibernate's core interface objects is its SessionFactory. All the Hibernate magic happens via this little object as it configures a Hibernate application, for instance by loading mapping files and configurations. If you use annotations or Hibernate's venerable .hbm files, you still need a SessionFactory to allow Hibernate to know which objects are persistable, and where to persist them.

Thus, with Hibernate Shards, you must leverage an augmented SessionFactory type that is capable of configuring multiple databases. It's appropriately named ShardedSessionFactory and it is, of course, of type SessionFactory. When creating a ShardedSessionFactory, you must provide the previously configured three shard implementation types (ShardAccessStrategy, ShardSelectionStrategy, and ShardResolutionStrategy). You'll also have to provide any mapping files required for your POJOs. (It's slightly different if you use an annotations-based Hibernate POJO configuration.) Lastly, a ShardedSessionFactory instance needs to have multiple Hibernate configuration files corresponding to each shard you wish to leverage.

Creating a Hibernate Configuration

I've created a ShardedSessionFactoryBuilder type that has one primary method, createSessionFactory, which creates an appropriately configured SessionFactory. Later I'll wire everything together with Spring (who doesn't leverage an IOC container, these days?). For now, Listing 2 shows the primary function of the ShardedSessionFactoryBuilder: to create a Hibernate Configuration:

```java
private Configuration getPrototypeConfig(String hibernateFile, List<String> resourceFiles) {
    Configuration config = new Configuration().configure(hibernateFile);
    for (String res : resourceFiles) {
        configs.addResource(res);
    }
    return config;
}
```

Listing 2. Creating a Hibernate Configuration
As you can see in **Listing 2**, a simple configuration is created from a Hibernate configuration file. This file holds information like what type of database is being used, username, password, etc., as well as any necessary resource files, such as .hbm files for POJOs. In a sharded situation where you're using multiple database configurations, Hibernate Shards makes it simple to use just one hibernate.cfg.xml file (you'll need one for each shard you intend to use, however, as you can see in **Listing 4**).

Next, in **Listing 3**, I collect all the shard configurations into a `List`:

**Listing 3. A List of shard configurations**

```java
List<ShardConfiguration> shardConfigs = new ArrayList<ShardConfiguration>();
for (String hibconfig : this.hibernateConfigurations) {
    shardConfigs.add(buildShardConfig(hibconfig));
}
```

**The Spring configuration**

In **Listing 3**, the reference to `hibernateConfigurations` points to a `List` of `String`s, each containing the name of a Hibernate configuration file. This `List` will be autowired by Spring. **Listing 4** is a snippet from my Spring configuration file showing this piece:

**Listing 4. Part of a Spring configuration file**

```xml
<bean id="shardedSessionFactoryBuilder" class="org.disco.racer.shardsupport.ShardedSessionFactoryBuilder">
    <property name="resourceConfigurations">
        <list>
            <value>racer.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateConfigurations">
        <list>
            <value>shard0.hibernate.cfg.xml</value>
            <value>shard1.hibernate.cfg.xml</value>
        </list>
    </property>
</bean>
```

As you can see, in **Listing 4**, the `shardedSessionFactoryBuilder` is being wired with one POJO mapping file and two shard configuration files. A snippet of the POJO file is shown in **Listing 5**:

**Listing 5. Race POJO mapping**

```xml
<class name="org.disco.racer.domain.Race" table="race" dynamic-update="true" dynamic-insert="true">
    <id name="id" column="RACE_ID" unsaved-value="-1">
        <generator class="org.hibernate.shards.id.ShardedUUIDGenerator"/>
    </id>

    <set name="participants" cascade="save-update" inverse="false" table="race_participants" lazy="false">
        <key column="race_id"/>
        <many-to-many column="runner_id" class="org.disco.racer.domain.Runner"/>
    </set>

    <set name="results" inverse="true" table="race_results" lazy="false">
        <key column="race_id"/>
    </set>
</class>
```
Note how the only unique aspect of the POJO mapping in Listing 5 is the generator class for the ID — it's the ShardedUUIDGenerator, which (as you'll recall) embeds shard ID information in the UUID itself. That's the only specific aspect to sharding in my POJO mapping.

**Shard configuration files**
Next, in Listing 6, I've configured one shard — in this case, Shard 0. Shard 1's file would be identical except for the shard ID and connection information.

**Listing 6. A Hibernate Shards configuration file**
```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory name="HibernateSessionFactory0">
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:file:.../db01/db01</property>
    <property name="connection.username">SA</property>
    <property name="hibernate.connection.shard_id">0</property>
  </session-factory>
</hibernate-configuration>
```

As its name suggests, the enable_cross_shard_relationship_checks property checks for cross-shard relationships. According to the Hibernate Shards documentation, this property is quite expensive and should be turned off in a production environment.

Finally, the ShardedSessionFactoryBuilder puts everything together by creating a ShardStrategyFactory, then adding three types (including the RacerShardSelectionStrategy from Listing 1), like in Listing 7:

**Listing 7. Creating a ShardStrategyFactory**
```java
private ShardStrategyFactory buildShardStrategyFactory() {
  ShardStrategyFactory shardStrategyFactory = new ShardStrategyFactory() {
    public ShardStrategy newShardStrategy(List<ShardId> shardIds) {
      ShardSelectionStrategy pss = new RacerShardSelectionStrategy();
      ShardResolutionStrategy prs = new AllShardsShardResolutionStrategy(shardIds);
      return new ShardStrategyImpl(pss, prs, pas);
    }
  };
  return shardStrategyFactory;
}
```
At last, I execute that nifty method dubbed `createSessionFactory`, which in this case creates a `ShardedSessionFactory`, shown in Listing 8:

**Listing 8. Creating a ShardedSessionFactory**

```java
public SessionFactory createSessionFactory() {
    Configuration prototypeConfig = this.getPrototypeConfig
    (this.hibernateConfigurations.get(0), this.resourceConfigurations);

    List<ShardConfiguration> shardConfigs = new ArrayList<ShardConfiguration>();
    for (String hibconfig : this.hibernateConfigurations) {
        shardConfigs.add(buildShardConfig(hibconfig));
    }

    ShardStrategyFactory shardStrategyFactory = buildShardStrategyFactory();
    ShardedConfiguration shardedConfig = new ShardedConfiguration(
        prototypeConfig, shardConfigs, shardStrategyFactory);
    return shardedConfig.buildShardedSessionFactory();
}
```

**Wiring domain objects with Spring**

Take a deep breath, now, because we're just about finished. Thus far, I've created a builder class that properly configures a `ShardedSessionFactory`, which really is just an implementation of Hibernate's ubiquitous `SessionFactory` type. The `ShardedSessionFactory` does all the magic of sharding. It leverages the shard-selection strategy I laid out in **Listing 1** and writes and reads data from the two shards I've configured. (**Listing 6** shows the configuration for Shard 0, and Shard 1 is almost identical.)

All I have to do now is wire up my domain objects, which in this case, because they'll rely on Hibernate, require a `SessionFactory` type to work. I'll just use my `ShardedSessionFactoryBuilder` to provide a `SessionFactory` type, like in Listing 9:

**Listing 9. Wiring a POJO in Spring**

```xml
<bean id="mySessionFactory"
    factory-bean="shardedSessionFactoryBuilder"
    factory-method="createSessionFactory">  
</bean>

<bean id="race_dao" class="org.disco.racer.domain.RaceDAOImpl">
    <property name="sessionFactory">
        <ref bean="mySessionFactory"/>
    </property>
</bean>
```

As you can see in **Listing 9**, I've first created a factory-like bean in Spring; that is, my `RaceDAOImpl` type has a property named `sessionFactory`, which is of type `SessionFactory`. Consequently, the `mySessionFactory` reference creates an instance of `SessionFactory` by invoking the `createSessionFactory` method on the `ShardedSessionFactoryBuilder`, which was defined in **Listing 4**.

When I ask Spring (which I'm basically using as a giant factory for returning preconfigured objects) for an instance of my `Race` object, everything will be set. While not shown, the `RaceDAOImpl` type is...
an object that leverages Hibernate templates for data storage and retrieval. My `Race` type holds an instance of `RaceDAOImpl`, to which it defers any datastore-related activities. Pretty cozy, huh?

Note that my DAOs aren't tied to Hibernate Shards in code, but by configuration. The configuration (in Listing 5) ties them to a sharding-specific UUID generation scheme, which means I can reuse domain objects from an existing Hibernate implementation when I need to shard.

**Sharding: A test drive with easyb**

Next, I need to verify that my sharding implementation works. I have two databases and I'm sharding by distance, so when I create a marathon (which is greater than 10 miles), say, that `Race` instance should be found in Shard 1. A smaller race, like a 5K (which is 3.1 miles), should be found in Shard 0. After I create a `Race`, I can check an individual database for its record.

In Listing 10, I've created a marathon and then proceeded to verify that the record is indeed in Shard 1 and not in Shard 0. To make things extra interesting (and easy) I've used easyb, a Groovy-based behavior-driven development framework that facilitates natural-language verification. easyb easily works with Java code too. Even without knowing Groovy or easyb, you should be able to follow the code in Listing 10 and see that everything works as planned. (Note that I helped create easyb and have written elsewhere on developerWorks about it.)

**Listing 10. A snippet of an easyb story that verifies shard correctness**

```java
scenario "races greater than 10.0 miles should be in shard 1 or db02", {
    given "a newly created race that is over 10.0 miles", {
        new Race("Leesburg Marathon", new Date(), 26.2,
            "Race the beautiful streets of Leesburg!").create()
    }
    then "everything should work fine w/respect to Hibernate", {
        rce = Race.findByName("Leesburg Marathon")
        rce.distance.shouldBe 26.2
    }
    and "the race should be stored in shard 1 or db02", {
        sql = Sql.newInstance(db02url, name, psswrd, driver)
        sql.eachRow("select race_id, distance, name from race where name=?",
            ["Leesburg Marathon"]) { row ->
            row.distance.shouldBe 26.2
        }
        sql.close()
    }
    and "the race should NOT be stored in shard 0 or db01", {
        sql = Sql.newInstance(db01url, name, psswrd, driver)
        sql.eachRow("select race_id, distance, name from race where name=?",
            ["Leesburg Marathon"]) { row ->
            fail "shard 0 contains a marathon!"
        }
        sql.close()
    }
}
```

Of course, my job's not done — I still need to create a shorter race and verify that it lands in Shard 0 and not in Shard 1. You can see that verification exercise in the code download that comes with this article!
The pros and cons of sharding

Sharding can speed up application reads and writes, especially if your application holds a tremendous amount of data — think terabytes — or if you are in a domain with unbounded growth, like Google or Facebook.

Before you shard, make sure your application’s size and growth merit it. Sharding's costs (or cons) include the burden of coding application-specific logic for how data will be stored and retrieved. Once you shard, you are also more or less locked into your sharding model, because resharding isn't easy to do.

In the right situation, sharding could be key to unlocking scale and speed in a traditional RDBMS. Sharding is a particularly cost-effective decision for organizations tied to a relational infrastructure that cannot continue to upgrade hardware to meet the need for massively scalable data storage.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code for this article</td>
<td>j-javadev2-11.zip</td>
<td>15KB</td>
</tr>
</tbody>
</table>
Resources

Learn

- **Java development 2.0**: This developerWorks series explores technologies and tools that are redefining the Java development landscape, including NoSQL (May 2010), SimpleDB (June 2010), and CouchDB (November 2009).
- "**Sharding with Max Ross: Hibernate Shards**" (JavaWorld, July 2008): Andy Glover talks with the founder of Hibernate Shards about the motivations behind the project and how to leverage shards effectively.
- "**Think twice before sharding**" (Andrew Glover, thediscoblog.com, June 2008): Sharding doesn’t come without risks, so find out more about them before you commit.
- "**Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes**" (Dare Obasanjo, 25HoursADay.com, January 2009): Uses Facebook as a real-world example to explore sharding strategies.
- "**Rethinking the traditional DAO pattern**" (Andrew Glover, thediscoblog.com, June 2008): Learn more about folding DAOs into normal domain objects, *a la* Grails and Rails.
- "**Drive development with easyb**" (Andrew Glover, developerWorks, November 2008): A tutorial introduction to easyb from the framework’s creator.
- Browse the [Java technology bookstore](https://www.ibm.com/developerworks/java/library/j-jacbook/) for books on these and other technical topics.

Get products and technologies

- **Hibernate Shards**: Designed to encapsulate and minimize data complexity by adding support for horizontal partitioning to the Hibernate Core.
- **easyb**: By using a specification based Domain Specific Language, easyb aims to enable executable, yet readable documentation.

Discuss

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. He is the founder of the easyb Behavior-Driven Development (BDD) framework and is the co-author of three books: Continuous Integration, Groovy in Action, and Java Testing Patterns. You can keep up with him at his blog and by following him on Twitter.