Hibernate Shards

Horizontal Partitioning With Hibernate

Version: 3.0.0.Beta2

Table of Contents

Preface

You can't always put all your relational data in a single relational database. Sometimes you simply have too much data. Sometimes you have a distributed deployment architecture (network latency between California and India might be too high to have a single database). There might even be non-technical reasons (a potential customer simply won't do the deal unless her company's data lives in its own db instance). Whatever your reasons, talking to multiple relational databases inevitably complicates the development of your application. Hibernate Shards is a framework that is designed to encapsulate and minimize this complexity by adding support for horizontal partitioning [http://en.wikipedia.org/w/index.php?title=Partition_%28database%29&oldid=99996308] on top of Hibernate Core. Simply put, we aim to provide a unified view of multiple databases via Hibernate.

So what's a shard? Good question. "Shard" is just another word for "segment" or "partition," but it's the term of choice within Google. Hibernate Shards was originally the 20 percent project [http://www.google.com/support/jobs/bin/static.py?page=about.html] of a small team of Google engineers, so the project nomenclature revolved around shards from the beginning. We're open sourcing what we have so far because we want the Hibernate community to be able to benefit from our efforts as soon as possible, but also with the hope and expectation that this community will be able to help us reach a GA release much faster than if we kept this under wraps. We fully expect to find glitches in both our design and implementation, and we appreciate your patience as we work through them.

Chapter 1. Architecture

1.1. Overview

Hibernate Shards is an extension to Hibernate Core that is designed to encapsulate and minimize the complexity of working with sharded (horizontally partitioned) data. Hibernate Shards can be conceptually divided into two areas, both of which you will need to understand in order to be successful. The two areas are:

- Generalized sharding logic
- Application specific sharding logic

We'll discuss each of these areas in turn.

1.2. Generalized Sharding Logic

The primary goal of Hibernate Shards is to enable application developers to query and transact against sharded datasets using the standard Hibernate Core API. This allows existing applications that use Hibernate but do not yet need sharding to adopt our solution without major refactoring if and when they do reach this stage. This also allows application developers who are familiar with Hibernate, need sharding, and are starting from scratch to become productive in a short amount of time because there will be no need to ramp-up on a new toolset. With this goal in mind, it should come as no surprise that Hibernate Shards primarily consists of shard-aware implementations of many of the Hibernate Core interfaces you already know and love.

Most Hibernate-related application code primarily interacts with four interfaces provided by Hibernate Core:

- `org.hibernate.Session`
- `org.hibernate.SessionFactory`
- `org.hibernate.Criteria`
- `org.hibernate.Query`

Hibernate Shards provides shard-aware extensions of these four interfaces so that your code does not need to know that it is interacting with a sharded dataset (unless of course you have specific reasons for exposing this fact). The shard-aware extensions are:

- `org.hibernate.shards.session.ShardedSession`
- `org.hibernate.shards.ShardedSessionFactory`
- `org.hibernate.shards.criteria.ShardedCriteria`
- `org.hibernate.shards.query.ShardedQuery`

The implementations we provide for these four shard-aware interfaces serve as a sharding engine that knows how to apply your application-specific sharding logic across your various data stores. We don't expect application developers to need to write much code that knowingly interacts with these interfaces, so if you do find yourself declaring or passing around Sharded instances take a step back and see if you can make do with the parent interface instead.

1.3. Application Specific Sharding Logic

Every application that uses Hibernate Shards will have its own rules for how data gets distributed across its shards. Rather than attempt to anticipate all these rules (an effort practically guaranteed to fail) we have instead provided a set of interfaces behind which you can encode your application's data distribution logic. These interfaces are:

- `org.hibernate.shards.strategy.selection.ShardSelectionStrategy`
- `org.hibernate.shards.strategy.resolution.ShardResolutionStrategy`
- `org.hibernate.shards.strategy.access.ShardAccessStrategy`

The implementations you provide for these three interfaces plus the id generation implementation you choose (more on this in the Sharding Strategy chapter) comprise the *Sharding Strategy* for your application. The sharding engine described in the previous section knows how to use the Sharding Strategy you provide.

In order to help you get up and running quickly, Hibernate Shards comes with a couple simple implementations of these interfaces. We expect that they will aid you in your prototyping or in the early stages of actual application development, but we also expect that, sooner or later, most applications will provide their own implementations.

For more information on Sharding Strategies please consult the chapter of the same name.

1.4. System Requirements

Hibernate Shards has the same system requirements as Hibernate Core, with the additional restriction that we require Java 1.5 or higher.

Chapter 2. Configuration

2.1. Overview

When using Hibernate Shards you will find yourself making typical Hibernate Core API calls most of them time. However, in order to get your shard-aware datasource properly configured you'll need to understand a few concepts that are specific to Hibernate Shards. We'll introduce these new concepts as part of a concrete example. Let's take a look at the object model, database schema, and mapping we'll be using in our examples throughout the documentation.

Our example application will receive weather reports from cities all over the world and store this information in a relational database.

2.1.1. Weather Report Database Schema

```
CREATE TABLE WEATHER_REPORT (
    REPORT_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    CONTINENT ENUM('AFRICA', 'ANTARCTICA', 'ASIA', 'AUSTRALIA', 'EUROPE', 'NORTH AMERICA', 'SOUTH AMER
    LATITUDE FLOAT,
    LONGITUDE FLOAT,
    TEMPERATURE INT,
    REPORT_TIME TIMESTAMP
);
```

2.1.2. Weather Report Object Model

```
public class WeatherReport {
    private Integer reportId;
    private String continent;
    private BigDecimal latitude;
    private BigDecimal longitude;
    private int temperature;
    private Date reportTime;

    ... // getters and setters
}
```

2.1.3. Contents of weather.hbm.xml

```
<hibernate-mapping package="org.hibernate.shards.example.model">
    <class name="WeatherReport" table="WEATHER_REPORT">
        <id name="reportId" column="REPORT_ID">
            <generator class="native"/>
        </id>
        <property name="continent" column="CONTINENT"/>
        <property name="latitude" column="LATITUDE"/>
        <property name="longitude" column="LONGITUDE"/>
        <property name="temperature" column="TEMPERATURE"/>
        <property name="reportTime" type="timestamp" column="REPORT_TIME"/>
    </class>
</hibernate-mapping>
```

2.2. Obtaining a ShardedSessionFactory

Before we show you how to obtain a `ShardedSessionFactory` let's look at some code that allows you to obtain a standard `SessionFactory`.

```
1    public SessionFactory createSessionFactory() {
2        Configuration config = new Configuration();
3        config.configure("weather.hibernate.cfg.xml");
4        config.addResource("weather.hbm.xml");
5        return config.buildSessionFactory();
6    }
```

This is pretty straightforward. We're instantiating a new `Configuration` object (line 2), telling that `Configuration` to read its properties from a resource named "weather.hibernate.cfg.xml" (line 3), and then providing "weather.hbm.xml" as a source of OR mapping data (line 4). We are then asking the `Configuration` to build a `SessionFactory`, which we return (line 5).

Let's also take a look at the configuration file we're loading in:

```
1    <!-- Contents of weather.hibernate.cfg.xml -->
2    <hibernate-configuration>
3      <session-factory name="HibernateSessionFactory">
4        <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
5        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
6        <property name="connection.url">jdbc:mysql://localhost:3306/mydb</property>
7        <property name="connection.username">my_user</property>
8        <property name="connection.password">my_password</property>
9      </session-factory>
10   </hibernate-configuration>
```

As you can see, there's nothing particularly interesting going on in the configuration file or the mapping file.

You'll be pleased to know that the process of configuring your application to use Hibernate Shards is not radically different. The main difference is that we're providing connectivity information for multiple datasources, and we're also describing our desired sharding behavior via a `ShardStrategyFactory`. Let's look at some sample configuration code for our weather report application, which we're going to run with 3 shards.

```
1     public SessionFactory createSessionFactory() {
2         Configuration prototypeConfig = new Configuration().configure("shard0.hibernate.cfg.xml");
3         prototypeConfig.addResource("weather.hbm.xml");
4         List<ShardConfiguration> shardConfigs = new ArrayList<ShardConfiguration>();
5         shardConfigs.add(buildShardConfig("shard0.hibernate.cfg.xml"));
6         shardConfigs.add(buildShardConfig("shard1.hibernate.cfg.xml"));
7         shardConfigs.add(buildShardConfig("shard2.hibernate.cfg.xml"));
8         ShardStrategyFactory shardStrategyFactory = buildShardStrategyFactory();
9         ShardedConfiguration shardedConfig = new ShardedConfiguration(
10            prototypeConfig,
11            shardConfigs,
12            shardStrategyFactory);
13        return shardedConfig.buildShardedSessionFactory();
14    }
15
16    ShardStrategyFactory buildShardStrategyFactory() {
17        ShardStrategyFactory shardStrategyFactory = new ShardStrategyFactory() {
18            public ShardStrategy newShardStrategy(List<ShardId> shardIds) {
19                RoundRobinShardLoadBalancer loadBalancer = new RoundRobinShardLoadBalancer(shardIds
20                ShardSelectionStrategy pss = new RoundRobinShardSelectionStrategy(loadBalancer);
21                ShardResolutionStrategy prs = new AllShardsShardResolutionStrategy(shardIds);
22                ShardAccessStrategy pas = new SequentialShardAccessStrategy();
23                return new ShardStrategyImpl(pss, prs, pas);
```

```
24            }
25        };
26        return shardStrategyFactory;
27    }
28
29    ShardConfiguration buildShardConfig(String configFile) {
30        Configuration config = new Configuration().configure(configFile);
31        return new ConfigurationToShardConfigurationAdapter(config);
32    }
```

So what's going on here? First, you'll notice that we're actually allocating four `Configurations`. The first `Configuration` we allocate (line 2) is the prototype `Configuration`. The `ShardedSessionFactory` we eventually construct (line 13) will contain references to 3 standard `SessionFactory` objects. Each of these 3 standard `SessionFactory` objects will have been constructed from the prototype configuration. The only attributes that will differ across these standard `SessionFactory` objects are:

- connection.url
- connection.user
- connection.password
- connection.datasource
- cache.region_prefix

The three `ShardConfiguration` objects we're loading (lines 5 - 7) will be consulted for the shard-specific database url, database user, database password, datasource identifier, cache region prefix, and that's all. (For a discussion of what these properties are and how they are used, please consult the Hibernate Core documentation.) This means that if you change the connection pool parameters in shard1.hibernate.cfg.xml, those parameters will be ignored. If you add another mapping file to the `Configuration` loaded with the properties defined in shard2.hibernate.cfg.xml, that mapping will be ignored. With the exception of the properties listed above, the configuration of our shard-aware `SessionFactory` comes entirely from the prototype `Configuration`. This may seem a bit strict, but the sharding code needs to assume that all shards are identically configured.

If you're looking at this code and thinking it seems a bit silly to provide fully-formed configuration documents that, save a couple special properties, are ignored, rest assured we've looked at this code and thought the same thing. That's why the `ShardedConfiguration` constructor takes a `List<ShardConfiguration>` as opposed to a `List<Configuration>`. `ShardConfiguration` is an interface so you can make the shard-specific configuration data available any way you'd like. In our example we're using an implementation of this interface that wraps a standard `Configuration` (line 31) just to avoid introducing any unfamiliar configuration mechanisms.

Once we've built our `Configuration` objects we need to put together a `ShardStrategyFactory` (line 8). A `ShardStrategyFactory` is an object that knows how to create the 3 types of strategies that programmers can use to control the sharding behavior of the system. For more information on these strategies please see the chapters titled Sharding Strategies.

Once we've instantiated our `ShardStrategyFactory` we can construct a `ShardedConfiguration` (line 9), and once we've constructed our `ShardedConfiguration` we can ask it to create a `ShardedSessionFactory` (line 13). It's important to note that `ShardedSessionFactory` extends `SessionFactory`. This means we can return a standard `SessionFactory` (line 1). Our application's Hibernate code doesn't need to know that it's interacting with sharded data.

Now let's take a look at the configuration and mapping files that we loaded in. You'll definitely recognize them, but there are a few key additions and modifications related to sharding.

```
1    <!-- Contents of shard0.hibernate.cfg.xml -->
2    <hibernate-configuration>
3      <session-factory name="HibernateSessionFactory0"> <!-- note the different name -->
4        <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
5        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
6        <property name="connection.url">jdbc:mysql://dbhost0:3306/mydb</property>
7        <property name="connection.username">my_user</property>
8        <property name="connection.password">my_password</property>
9        <property name="hibernate.connection.shard_id">0</property> <!-- new -->
10      <property name="hibernate.shard.enable_cross_shard_relationship_checks">true</property> <!--
11    </session-factory>
12  </hibernate-configuration>
```

```
1    <!-- Contents of shard1.hibernate.cfg.xml -->
2    <hibernate-configuration>
3      <session-factory name="HibernateSessionFactory1"> <!-- note the different name -->
4        <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
5        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
6        <property name="connection.url">jdbc:mysql://dbhost1:3306/mydb</property>
7        <property name="connection.username">my_user</property>
8        <property name="connection.password">my_password</property>
9        <property name="hibernate.connection.shard_id">1</property> <!-- new -->
10      <property name="hibernate.shard.enable_cross_shard_relationship_checks">true</property> <!--
11    </session-factory>
12  </hibernate-configuration>
```

We'll skip the contents of shard2.hibernate.cfg.xml because the pattern should by now be obvious. We're giving each session factory a unique name via the name attribute of the session-factory element, and we're associating each session factory with a different database server. We're also giving each session factory a shard id. This is required. If you try to configure a `ShardedSessionFactory` with a `Configuration` object that does not have a shard id you'll get an error. At the moment we require that the shard id of one of your session factories be 0. Beyond that, the internal representation of a shard id is a `java.lang.Integer` so all values within that range are legal. Finally, each shard that is mapped into a `ShardedSessionFactory` must have a unique shard id. If you have a duplicate shard id you'll get an error.

The other noteworthy addition is the rather verbose but hopefully descriptive "hibernate.shard.enable_cross_shard_relationship_checks." You can read more about this in the chapter on limitations.

Now let's still see how the mapping file has changed.

```
<hibernate-mapping package="org.hibernate.shards.example.model">
    <class name="WeatherReport" table="WEATHER_REPORT">
        <id name="reportId" column="REPORT_ID" type="long">
            <generator class="org.hibernate.shards.id.ShardedTableHiLoGenerator"/>
        </id>
        <property name="continent" column="CONTINENT"/>
        <property name="latitude" column="LATITUDE"/>
        <property name="longitude" column="LONGITUDE"/>
        <property name="temperature" column="TEMPERATURE"/>
        <property name="reportTime" type="timestamp" column="REPORT_TIME"/>
    </class>
</hibernate-mapping>
```

The only meaningful change in the mapping file from the non-sharded version is in our selection of a shard-aware id generator. We'll cover id generation in more detail in the chapter on Shard Strategies.

2.3. Using Hibernate Annotations With Shards

In the above example we're using Hibernate mapping files (hbm.xml) to specify our mappings, but it's just as easy to use Hibernate Annotations. We can annotate our `WeatherReport` class as follows:

```
@Entity
@Table(name="WEATHER_REPORT")
public class WeatherReport {

    @Id @GeneratedValue(generator="WeatherReportIdGenerator")
    @GenericGenerator(name="WeatherReportIdGenerator", strategy="org.hibernate.shards.id.ShardedUUIDG
    @Column(name="REPORT_ID")
    private Integer reportId;

    @Column(name="CONTINENT")
    private String continent;

    @Column(name="LATITUDE")
    private BigDecimal latitude;

    @Column(name="LONGITUDE")
    private BigDecimal longitude;

    @Column(name="TEMPERATURE")
    private int temperature;

    @Column(name="REPORT_TIME")
    private Date reportTime;

    ... // getters and setters
}
```

This is a pretty standard use of Hibernate Annotations. The only thing here that's particularly noteworthy is the use of the `GenericGenerator` annotation, which is part of Hibernate Annotations but not JPA. We need this to specify our shard-aware id generator.

The changes we now need to make to the `createSessionFactory()` method we implemented above are actually quite small:

```
1     public SessionFactory createSessionFactory() {
2         AnnotationConfiguration prototypeConfig = new AnnotationConfiguration().configure("shard0.h
3         prototypeConfig.addAnnotatedClass(WeatherReport.class);
4         List<ShardConfiguration> shardConfigs = new ArrayList<ShardConfiguration>();
5         shardConfigs.add(buildShardConfig("shard0.hibernate.cfg.xml"));
6         shardConfigs.add(buildShardConfig("shard1.hibernate.cfg.xml"));
7         shardConfigs.add(buildShardConfig("shard2.hibernate.cfg.xml"));
8         ShardStrategyFactory shardStrategyFactory = buildShardStrategyFactory();
9         ShardedConfiguration shardedConfig = new ShardedConfiguration(
10            prototypeConfig,
11            shardConfigs,
12            shardStrategyFactory);
13        return shardedConfig.buildShardedSessionFactory();
14    }
```

The only changes between this method and the non-annotated versions are on lines 2 and 3. On line 2 we're declaring and instantiating an `AnnotationConfiguration` instead of a `Configuration`, and on line 3 we're adding an annotated class to the configuration instead of an xml mapping file. That's it!

Please note that while Hibernate Shards works with Hibernate Annotations, Hibernate Shards does not ship with Hibernate Annotations. You'll need to download Hibernate Annotations and its dependencies separately.

2.4. Configuration Limitations

Many of you will quickly realize that the configuration mechanism we've provided won't work if you're configuring your `SessionFactory` via JPA. It's true. We expect this deficiency to be addressed shortly.

Chapter 3. Shard Strategy

3.1. Overview

Hibernate Shards gives you enormous flexibility in configuring how your data is distributed across your shards and how your data is queried across your shards. The entry point for this configuration is the `org.hibernate.shards.strategy.ShardStrategy` interface:

```
public interface ShardStrategy {
    ShardSelectionStrategy getShardSelectionStrategy();
    ShardResolutionStrategy getShardResolutionStrategy();
    ShardAccessStrategy getShardAccessStrategy();
}
```

As you can see, a `ShardStrategy` is comprised of three sub-strategies. We'll discuss each of these in turn.

3.2. ShardAccessStrategy

We'll start with the most simple of the strategies: `ShardAccessStrategy`. Hibernate Shards uses the `ShardAccessStrategy` to determine how to apply database operations across multiple shards. The `ShardAccessStrategy` is consulted whenever you execute a query against your shards. We've already provided two implementations of this interface that we expect will suffice for the majority of applications.

3.2.1. SequentialShardAccessStrategy

`SequentialShardAccessStrategy` behaves in the exact way that is implied by its name: queries are executed against your shards in sequence. Depending on the types of queries you execute you may want to avoid this implementation because it will execute queries against your shards in the same order every time. If you execute a lot of row-limited, unordered queries this *could* result in poor utilization across your shards (shards that appear early in your list will get hammered and shards that appear late will sit idly by, twiddling their shard-thumbs). If this is a concern you should consider using the `LoadBalancedSequentialShardAccessStrategy` instead. This implementation receives a rotated view of your shards on each invocation, thereby distributing query load evenly.

3.2.2. ParallelShardAccessStrategy

`ParallelShardAccessStrategy` also behaves in the exact way that is implied by its name: queries are executed against your shards in parallel. When you use this implementation you need to provide a `java.util.concurrent.ThreadPoolExecutor` that is suitable for the performance and throughput needs of your application. Here's a simple example:

```
    ThreadFactory factory = new ThreadFactory() {
        public Thread newThread(Runnable r) {
            Thread t = Executors.defaultThreadFactory().newThread(r);
            t.setDaemon(true);
            return t;
        }
    };

    ThreadPoolExecutor exec =
        new ThreadPoolExecutor(
```

```
            10,
            50,
            60,
            TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(),
            factory);

    return new ParallelShardAccessStrategy(exec);
```

Please note that these are just sample values - proper thread pool configuration is beyond the scope of this document.

3.3. ShardSelectionStrategy

Hibernate Shards uses the `ShardSelectionStrategy` to determine the shard on which a new object should be created. It's entirely up to you to decide what you want your implementation of this interface to look like, but we've provided a round-robin implementation to get you started (`RoundRobinShardSelectionStrategy`). We expect many applications will want to implement attribute-based sharding, so for our example application that stores weather reports let's shard reports by the continents on which the reports originate:

```
public class WeatherReportShardSelectionStrategy implements ShardSelectionStrategy {
    public ShardId selectShardIdForNewObject(Object obj) {
        if(obj instanceof WeatherReport) {
            return ((WeatherReport)obj).getContinent().getShardId();
        }
        throw new IllegalArgumentException();
    }
}
```

It's important to note that if a multi-level object graph is being saved via Hibernate's cascading functionality, the `ShardSelectionStrategy` will only be consulted when saving the top-level object. All child objects will automatically be saved to the same shard as the parent. You may find your `ShardSelectionStrategy` easier to implement if you prevent developers from creating new objects at more than one level in your object hierarchy. You can accomplish this by making your `ShardSelectionStrategy` implementation aware of the top-level objects in your model and having it throw an exception if it encounters an object that is not in this set. If you do not wish to impose this restriction that's fine, just remember that if you're doing attribute-based shard selection, the attributes you use to make your decision need to be available on every object that gets passed to session.save().

3.4. ShardResolutionStrategy

Hibernate Shards uses the `ShardResolutionStrategy` to determine the set of shards on which an object with a given id might reside. Let's go back to our weather report application and suppose, for example, that each continent has a range of ids associated with it. Whenever we assign an id to a WeatherReport we pick one that falls within the legal range for the continent to which the WeatherReport belongs. Our `ShardResolutionStrategy` can use this information to identify which shard a WeatherReport resides on simply by looking at the id:

```
public class WeatherReportShardResolutionStrategy extends AllShardsShardResolutionStrategy {
    public WeatherReportShardResolutionStrategy(List<ShardId> shardIds) {
        super(shardIds);
    }

    public List<ShardId> selectShardIdsFromShardResolutionStrategyData(
            ShardResolutionStrategyData srsd) {
```

```
        if(srsd.getEntityName().equals(WeatherReport.class.getName())) {
            return Continent.getContinentByReportId(srsd.getId()).getShardId();
        }
        return super.selectShardIdsFromShardResolutionStrategyData(srsd);
    }
}
```

It's worth pointing out that while we have not (yet) implemented a cache that maps entity name/id to shard, the `ShardResolutionStrategy` would be an excellent place to plug in such a cache.

Shard Resolution is tightly tied to ID Generation. If you select an ID Generator for your class that encodes the shard id in the id of the object, your `ShardResolutionStrategy` will never even be called. If you plan to only use ID Generators that encode the shard id in the ids of your object you should use `AllShardsShardResolutionStrategy` as your `ShardResolutionStrategy`.

3.5. ID Generation

Hibernate Sharding supports any ID generation strategy; the only requirement is that object IDs have to be unique across all the shards. There are a few simple ID generation strategies which support this requirement:

- *Native ID generation* - use Hibernate's `native` ID generation strategy, and configure your databases so that the IDs never collide. For example, if you are using `identity` ID generation, you have 5 databases across which you will evenly distribute the data, and you don't expect you will ever have more than 1 million records, you could configure database 0 to return IDs starting at 0, configure database 1 to return IDs starting at 200000, configure database 2 to return IDs starting at 400000, and so on. As long as your assumptions about the data are correct, the IDs of your objects would never collide.
- *Application-level UUID generation* - by definition you don't have to worry about ID collisions, but you do need to be willing to deal with potentially unwieldy primary keys for your objects.

  Hibernate Shards provides an implementation of a simple, shard-aware UUID generator - `ShardedUUIDGenerator`.
- *Distributed hilo generation* - the idea is to have a hilo table on only one shard, which ensures that the identifiers generated by the hi/lo algorithm are unique across all shards. Two main drawbacks of this approach are that the access to the hilo table can become the bottleneck in ID generation, and that storing the hilo table on a single database creates a single point of failure of the system.

  Hibernate Shards provides an implementation of a distributed hilo generation algorithm - `ShardedTableHiLoGenerator`. This implementation is based on `org.hibernate.id.TableHiLoGenerator`, so for information on the expected structure of the database table on which the implementation depends please see the documentation for this class.

ID generation is also tightly tied with the shard resolution. The objective of shard resolution is to find the shard an object lives on, given the object's ID. There are two ways to accomplish this objective:

- Use the `ShardResolutionStrategy`, described above
- Encode the shard ID into the object ID during the ID generation, and retrieve the shard ID during shard resolution

The main advantage of encoding the shard ID into the object ID is that it enables Hibernate Shards to resolve the shard from the object's ID much faster without database lookups, cache lookups, etc. Hibernate Shards does not require any specific algorithm for encoding/decoding of the shard ID - all you have to do is use an ID gen-

erator that implements the `ShardEncodingIdentifierGenerator` interface. Of the two ID generators included with Hibernate Shards, the `ShardedUUIDGenerator` implements this interface.

Chapter 4. Resharding

When an application's dataset grows beyond the capacity of the databases originally allocated to the application it becomes necessary to add more databases, and it is often desirable to redistribute the data across the shards (either to achieve proper load balancing or to satisfy application invariants) - this is called resharding. Resharding is a complicated problem, and it has the potential to cause major complications in the management of your production application if it is not considered during the design. In order to ease some of the pain associated with resharding, Hibernate Shards provides support for virtual shards.

4.1. Virtual Shards

In the general case, each object lives on a shard. Resharding consists of two tasks: moving the object to another shard, and changing object-shard mappings. The object-shard mapping is captured either by the shard ID encoded into the object ID or by the internal logic of the shard resolution strategy which the object uses. In the former case, resharding would require changing all the object IDs and FKs. In the latter case, resharding could require anything from changing the runtime configuration of a given `ShardResolutionStrategy` to changing the algorithm of the `ShardResolutionStrategy`. Unfortunately, the problem of changing object-shard mappings becomes even worse once we consider the fact that Hibernate Shards does not support cross-shard relationships. This limitation prevents us from moving a subset of an object graph from one shard to another.

The task of changing object-shard mappings can be simplified by adding a level of indirection - each object lives on a virtual shard, and each virtual shard is mapped to one physical shard. During design, developers must decide on the maximum number of physical shards the application will ever require. This maximum is then used as the number of virtual shards, and these virtual shards are then mapped to the physical shards currently required by the application. Since Hibernate Shards' `ShardSelectionStrategy`, `ShardResolutionStrategy`, and `ShardEncodingIdentifierGenerator` all operate on virtual shards, the objects will correctly be distributed across virtual shards. During resharding, object-shard mappings can now simply be changed by changing virtual shard to physical shard mappings.

If you're worried about correctly estimating the maximum number of physical shards your application will ever require, aim high. Virtual shards are cheap. Down the road you'll be much better off with extra virtual than if you have to add virtual shards.

In order to enable virtual sharding you need to create your `ShardedConfiguration` with a Map from virtual shard ids to physical shard ids. Here's an example where we have 4 virtual shards mapped to 2 physical shards.

```
Map<Integer, Integer> virtualShardMap = new HashMap<Integer, Integer>();
virtualShardMap.put(0, 0);
virtualShardMap.put(1, 0);
virtualShardMap.put(2, 1);
virtualShardMap.put(3, 1);
ShardedConfiguration shardedConfig =
    new ShardedConfiguration(
        prototypeConfiguration,
        configurations,
        strategyFactory,
        virtualShardMap);
return shardedConfig.buildShardedSessionFactory();
```

In order to change the virtual shard to physical shard mapping later on it is only necessary to change the `virtualShardToShardMap` passed to this constructor.

We mentioned that the second task during resharding is moving data from one physical shard to another. Hibernate Shards does not try to provide automatic support for this as this is usually very application-specific, and

complexity varies based on the potential need for hot-resharding, deployment architecture of the application, etc.

Chapter 5. Querying

5.1. Overview

Executing queries across shards can be hard. In this chapter we'll discuss what works, what doesn't, and what you can do to stay out of trouble.

5.2. Criteria

As we discuss in the chapter on Limitations, we do not yet have a complete implementation of the Hibernate Core API. This limitation applies to `ShardedCriteriaImpl`, which is a shard-aware implementation of the `Criteria` interface. In this chapter we won't go into the details of specific things that haven't been implemented. Rather, we're going to discuss the types of `Criteria` queries that are problematic in a sharded environment.

Simply put, queries that do sorting are trouble. Why? Because we can't return a properly sorted list without the ability to compare any value in the list to any other value in the list, and the entire list isn't available until the results of the individual queries have been collected in the application tier. The sorting needs to take place inside Hibernate Shards, and in order for this to happen we require that all objects returned by a `Criteria` query with an order-by clause implement the `Comparable` interface. If the type of the objects you return do not implement this interface you'll receive an exception.

Distinct clauses are trouble as well. So much trouble, in fact, that at the moment we don't even support them. Sorry about that.

On the other hand, while distinct and order-by are trouble, aggregation works just fine. Consider the following example:

```
// fetch the average of all temperatures recorded since last thursday
Criteria crit = session.createCriteria(WeatherReport.class);
crit.add(Restrictions.gt("timestamp", lastThursday));
crit.setProjection(Projections.avg("temperature"));
return crit.list();
```

In a single-shard environment this query can be easily answered, but in a multi-shard environment it's a little bit trickier. Why? Because just getting the average from each shard isn't enough to calculate the average across all shards. In order to calculate this piece of information we need not just the average but the number of records from each shard. This is exactly what we do, and the performance hit (doing an extra count as part of each query) is probably negligible. Now, if we wanted the median we'd be in trouble (just adding the count to the query would not provide enough information to perform the calculation), but at the moment `Criteria` doesn't expose a median function so we'll deal with that if and when it becomes and issue.

5.3. HQL

Our support for HQL is, at this point, not nearly as good as the support we have for `Criteria` queries. We have not yet implemented any extensions to the query parser, so we don't support distinct, order-by, or aggregations. This means you can only use HQL for very simple queries. You're probably better off staying clear of HQL in this release if you can help it.

5.4. Use of Shard Strategy When Querying

The only component of your shard strategy that is consulted when executing a query (`Criteria` or HQL) is the `ShardAccessStrategy`. `ShardSelectionStrategy` is ignored because executing a query doesn't create any new records in the database. `ShardResolutionStrategy` is ignored because we currently assume that you always want your query executed on all shards. If this isn't the case, the best thing to do is just downcast your `Session` to a `ShardedSession` and dig out the shard-specific `Sessions` you need. Clunky, but it works. We'll come up with a better solution for this in later releases.

Chapter 6. Limitations

6.1. Incomplete Implementation of Hibernate API

In order to speed-up the initial release of Hibernate Shards, some parts of the Hibernate API that we rarely use were left unimplemented. Of course things that we rarely used are probably critical for some applications, so if we've left you out in the cold we apologize. We're committed to getting the rest of the API implemented quickly. For details on which methods were not implemented, please see the Javadoc for `ShardedSessionImpl`, `ShardedCriteriaImpl`, and `ShardedQueryImpl`.

6.2. Cross-Shard Object Graphs

Hibernate Shards does not currently support cross-shard object graphs.

In other words, it is illegal to create an association between objects A and B when A and B live on different shards. The workaround is to define a property on A which uniquely identifies an object of type B, and to use that property to load object B (remember what life was like before Hibernate? Yeah, just like that.)

For example:

```
                --need domain for examples--
```

In some applications your model may be constructed in such a way that it is difficult to make this kind of mistake, but in some applications it may be easier. The scary thing here is that if you make this mistake, Hibernate will consider the "bad" object in the list to be a new object and, assuming you have cascades enabled for this relationship, it will create a new version of this object on a different shard. This is trouble. In order to help prevent this sort of thing from happening we have an interceptor called `CrossShardRelationshipDetectingInterceptor` that checks for cross-shard relationships on every object that is created or saved.

Unfortunately there is a cost associated with using the `CrossShardRelationshipDetectingInterceptor`. In order to determine the shard on which an associated object resides we need to fetch the object from the database, so if you have lazy-loaded associations the interceptor will resolve those associations as part of its checks. This is potentially quite expensive, and may not be suitable for a production system. With this in mind, we've made it easy to configure whether or not this check is performed via the "hibernate.shard.enable_cross_shard_relationship_checks" property we referenced in the chapter on configuration. If this property is set to "true" a `CrossShardRelationshipDetectingInterceptor` will be registered with every `ShardedSession` that is established. Don't worry, you can still register your own interceptor as well. Our expectation is that most applications will have this check enabled in their dev and qa environments and disabled in their staging, load and performance, and production environments.

6.3. Distributed Transactions

Hibernate Shards does not provide support for distributed transactions within a non-managed environment. If your application requires distributed transactions you need to plug in a transaction management implementation that supports distributed transactions.

6.4. Stateful Interceptors

We've done our best to make sure that, by and large, Hibernate Core code runs just fine when using Hibernate Shards. There are, unfortunately, exceptions, and one of those exceptions is when your application needs to use an `org.hibernate.Interceptor` that maintains state.

Stateful interceptors need special handling because, under the hood, we're instantiating one `org.hibernate.SessionImpl` per shard. If we want an `Interceptor` associated with the `Session`, we need to pass in whatever `Interceptor` was provided when the `ShardedSession` was created. If that `Interceptor` is stateful, the `Interceptor` state for one `Session` will be visible in all `Sessions`. When you consider the sorts of things that are typically done in stateful `Interceptors` (auditing for example), you can see how this can pose a problem.

Our solution is to require users to provide a `StatefulInterceptorFactory` when they establish their `Session` objects (which are really `ShardedSessions`). If the provided `Interceptor` implements this interface, Hibernate Shards will ensure that a fresh instance of the type of `Interceptor` returned by `StatefulInterceptorFactory.newInstance()` will be passed to each `Session` that is established under the hood. Here's an example:

```
public class MyStatefulInterceptorFactory extends BaseStatefulInterceptorFactory {
    public Interceptor newInstance() {
        return new MyInterceptor();
    }
}
```

Many `Interceptor` implementations require a reference to the `Session` with which they're associated. In the case of a stateful `Interceptor`, you want your `Interceptor` to have a reference to the real (shard-specific) `Session`, not the shard-aware `Session`. In order to facilitate this, you have the choice of having the type of `Interceptor` that is constructed by the `StatefulInterceptorFactory` implement the `RequiresSession` interface. If the `Interceptor` constructed by the `StatefulInterceptorFactory` implements this interface, Hibernate Shards will provide the `Interceptor` with a reference to the real (shard-specific) `Session` once the factory constructs it. This way your `Interceptor` can safely and accurately interact with a specific shard. Here's an example:

```
public class MyStatefulInterceptor implements Interceptor, RequiresSession {
    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }

    ... // Interceptor interface impl
}
```

Due to the basic nature of the problem we don't expect this to change anytime soon.

6.5. Objects With Ids That Are Base Types

With Hibernate your model objects can use whatever they want as their ids so long as the id can be represented by a `Serializable` (or autoboxed into a `Serializable`). With Hibernate Shards you are slightly more constrained because we don't support base types.

So this is no good:

```
public class WeatherReport {
```

```
    private int weatherReportId;  // trouble

    public int getWeatherReportId() {
        return weatherReportId;
    }

    public void setWeatherReportId(int id) {
        weatherReportId = id;
    }
}
```

But this is just lovely:

```
public class WeatherReport {
    private Integer weatherReportId;  // goodness

    public Integer getWeatherReportId() {
        return weatherReportId;
    }

    public void setWeatherReportId(Integer id) {
        weatherReportId = id;
    }
}
```

Do we have a good reason for this limitation? Not really. It's the result of an implementation choice that has leaked out and made everyone's lives a tiny bit worse. If you simply must use Hibernate Shards and you simply must model your ids with base types, don't call `Session.saveOrUpdate`. We aim to address this leak soon and let you get back to modeling whatever way you like (although for the record, we prefer object ids because they make it easy to determine whether or not an object has had an id assigned).

6.6. Replicated Data

Even though this is a framework for horizontal partitioning, there is almost always read-only (or at least slow changing) data that lives on every shard. If you're just reading these entities we don't have a problem, but if you want to associate these entities with sharded entities we run into trouble. Suppose you have a Country table on every shard with the exact same data, and suppose WeatherReport has a Country member. How do we guarantee that the Country you associate with that WeatherReport is associated with the same shard as the WeatherReport? If we get it wrong we'll end up with a cross-shard relationship, and that's bad.

We have a number of ideas about how to make this easy to deal with but we have not yet implemented any of them. In the short term, we think your best bet is to either not create object relationships between sharded entities and replicated entities. In other words, just model the relationship like you would if you weren't using an OR Mapping tool. We know this is clunky and annoying. We'll take care of it soon.