# Hibernate Shards

## Horizontal Partitioning With Hibernate

Version: 3.0.0.Beta1

# Table des matières

# Préface

Traducteur(s) : Vincent Ricard

Vous ne pouvez pas toujours mettre vos données relationnelles dans une seule base de données relationnelle. Parfois vous avez simplement trop de données. Parfois vous avez une architecture de déploiement répartie (la latence réseau entre la Californie et l'Inde peut être trop grande pour avoir une seule base de données). Il peut même y avoir des raisons non techniques (un client potentiel ne traitera simplement pas à moins que les données de sa compagnie soient dans sa propre instance de base). Quelques soient vos raisons, parler de plusieurs bases de données relationnelles complique inévitablement le développement de votre application. Hibernate Shards est un framework qui est conçu pour encapsuler et minimiser cette complexité en ajoutant la prise en charge du partitionnement horizontal [http://en.wikipedia.org/w/index.php?title=Partition_%28database%29&oldid=99996308] au dessus d'Hibernate Core. Nous avons simplement pour but de founir une vue unifiée de plusieurs bases de données via Hibernate.

Qu'est donc un "shard" (NdT : fragment) ? Bonne question. "Shard" est juste un autre mot pour "segment" ou "partition", mais c'est le terme choisi par Google. Hibernate Shards était à l'origine le projet de 20 pourcents [http://www.google.com/support/jobs/bin/static.py?page=about.html] d'une petite équipe d'ingénieurs Google, donc la nomenclature du projet tournait autour des fragments depuis le début. Nous ouvrons les sources que nous avons jusqu'ici parce que nous voulons que la communauté Hibernate puisse bénéficier de nos efforts dès que possible, mais aussi avec l'espoir et le désir que cette communauté puisse nous aider à atteindre une version GA plus rapidement que si nous gardions pour nous. Nous nous attendons tout à fait à trouver des défauts dans notre conception et notre implémentation, et nous apprécions votre patience pendant que nous travaillons à les corriger.

# Chapitre 1. Architecture

## 1.1. Vue d'ensemble

Hibernate Shards est une extension d'Hibernate Core, conçu pour encapsuler et minimiser la complexité de travailler avec des données fragmentées (horizontalement partitionnées). Hibernate Shards peut être conceptuellement divisé en deux domaines que vous aurez besoin de comprendre pour réussir. Les deux domaines sont :

- la logique de fragmentation généralisée ;
- la logique de fragmentation spécifique à l'application.

Nous discuterons de chacun de ces domaines à leur tour.

## 1.2. La logique de fragmentation généralisée

Le but premier d'Hibernate Shards est de permettre aux développeurs d'application d'interroger et de négocier des ensembles de données fragmentés en utilisant l'API standard d'Hibernate Core. Ceci permet aux applications existantes qui utilisent Hibernate, mais pas encore la fragmentation, d'adopter notre solution sans modification majeure s'ils atteignent cette étape. Ceci permet aussi aux développeurs d'application qui connaissent Hibernate, qui ont besoin de fragmentation et qui partent de zéro, de devenir productifs en un minimum de temps parce qu'il n'y a pas besoin de découvrir un nouvel outil. Avec cet objectif en tête, il n'est pas surprenant qu'Hibernate Shards soit composé principalement d'implémentations prenant en compte la fragmentation de beaucoup d'interfaces d'Hibernate Core que vous connaissez et aimez.

La plupart du code applicatif en rapport avec Hibernate interagit avec quatre interfaces fournies par Hibernate Core :

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

Hibernate Shards fournit des extensions prenant en compte la fragmentation à ces quatre interfaces, ainsi votre code n'a pas besoin de savoir qu'il interagit avec un ensemble de données fragmenté (à moins, bien sûr, que vous ayez des raisons spécifiques pour révéler ce fait). Les extensions sont :

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

Les implémentations que nous fournissons pour ces quatre interfaces servent de moteur à fragmentation, lequel connaît la logique de fragmentation spécifique à l'application à travers vos différents stockages de données.

Nous n'espérons pas que les développeurs d'application aient besoin d'écrire trop de code qui interagit sciemment avec ces interfaces, donc si vous vous retrouvez en train de déclarer ou de passer des instances "Sharded", revenez un pas en arrière et regardez si vous ne pouvez pas plutôt le faire avec l'interface parente.

## 1.3. La logique de fragmentation spécifique à l'application

Chaque application qui utilise Hibernate Shards aura ses propres règles pour répartir ses données à travers les fragments. Plutôt que de tenter d'anticiper toutes ces règles (effort pratiquement voué à l'échec), nous avons fourni un ensemble d'interfaces derrière lesquelles vous pouvez coder la logique de distribution des données de votre application. Ces interfaces sont :

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

Les implémentations que vous fournissez pour ces trois interfaces, plus l'implémentation de la génération des identifiants que vous choisissez (plus d'informations à ce sujet dans le chapitre "Stratégie de fragmentation") constituent la *stratégie de fragmentation* de votre application.

Pour vous aider à mettre en place cela rapidement, Hibernate Shards arrive avec deux implémentations simples de ces interfaces. Nous attendons qu'elles vous aident dans votre prototypage ou dans les premières étapes du développement applicatif réel, mais nous espérons aussi que, tôt ou tard, la plupart des applications fourniront leurs propres implémentations.

Pour plus d'informations sur les stratégies de fragmentation, veuillez consulter le chapitre du même nom.

## 1.4. Pré-requis système

Hibernate Shards a les mêmes pré-requis qu'Hibernate Core, avec la restriction supplémentaire qui exige Java 1.5 ou supérieur.

# Chapitre 2. Configuration

## 2.1. Vue d'ensemble

Lors de l'utilisation d'Hibernate Shards, vous vous retrouvez la plupart du temps en train de faire des appels typiques à l'API d'Hibernate Core. Cependant, pour avoir votre source de données fragmentées proprement configurée, vous aurez besoin de comprendre quelques concepts spécifiques à Hibernate Shards. Nous présenterons ces nouveaux concepts dans le cadre d'un exemple concret. Examinons le modèle objet, le schéma de base de données, et le mapping que nous utiliserons dans nos exemples à travers la documentation.

Notre application d'exemple recevra des rapports météorologiques de villes du monde entier et stockera cette information dans une base de données relationnelles.

### 2.1.1. Schéma de base de données d'un rapport météorologique

```
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. Modèle objet d'un rapport météorologique

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

    ... // getters et setters
}
```

### 2.1.3. Contenu de 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. Obtenir une ShardedSessionFactory

Avant que nous vous montrions comment obtenir une `ShardedSessionFactory`, examinons le code qui vous permet d'avoir une `SessionFactory` standard.

```
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    }
```

C'est assez simple. Nous instancions un nouvel objet `Configuration` object (ligne 2), indiquons à `Configuration` de lire ses propriétés à partir d'une ressource nommée "weather.hibernate.cfg.xml" (ligne 3), et ensuite fournissons "weather.hbm.xml" comme une source de données de mapping OR (ligne 4). Nous demandons alors à `Configuration` de construire une `SessionFactory`, que nous retournons (ligne 5).

Regardons aussi le fichier de configuration que nous chargeons :

```
1    <!-- Contenu de 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>
```

Comme vous pouvez le voir, il n'y a rien de particulièrement intéressant dans le fichier de configuration ou le fichier de mapping.

Vous serez content d'apprendre que le processus de configuration de votre application pout utiliser Hibernate Shards n'est pas radicalement différent. La principale différence est que nous fournissons l'information de connexité pour plusieurs sources de données, et nous décrivons aussi le comportement de fragmentation désiré via une `ShardStrategyFactory`. Examinons une exemple de code de configuration pour notre application de rapports météorologiques, que nous allons exécuter avec 3 fragments.

```
1     public SessionFactory createSessionFactory() {
2         Configuration prototypeConfig = new Configuration().configure("shard0.hibernate.cfg.xml");
3         prototypeConfig.addResource("weather.hbm.xml");
4         List<Configuration> shardConfigs = new ArrayList<Configuration>();
5         shardConfigs.add(new Configuration().configure("shard0.hibernate.cfg.xml"));
6         shardConfigs.add(new Configuration().configure("shard1.hibernate.cfg.xml"));
7         shardConfigs.add(new Configuration().configure("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 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    }
```

Que se passe-t-il ici ? D'abord, vous noterez que nous allouons réellement quatre `Configurations`. La première `Configuration` que nous allouons (ligne 2) est la `Configuration` prototype. La `ShardedSessionFactory` que nous contruisons éventuellement (ligne 13) contiendra des références aux 3 objets `SessionFactory` standards. Chacun de ces 3 objets `SessionFactory` standards aura été contruit à partir de la configuration prototype. Les seuls attributs qui différeront de ces objets `SessionFactory` standards sont :

- connection.url
- connection.user
- connection.password

Les trois objets `Configuration` que nous chargeons (lignes 5 à 7) seront consultés pour l'url, l'utilisateur et le mot de passe spécifiques aux bases de données des fragments, et c'est tout. Ce qui veut dire que si vous changez les paramètres du pool de connexions dans shard1.hibernate.cfg.xml, ils seront ignorés. Si vous ajoutez un autre fichier de mapping à la `Configuration` chargée avec les propriétés définies dans shard2.hibernate.cfg.xml, il sera ignoré. A l'exception des propriétés listées plus haut, la configuration de notre `SessionFactory` vient entièrement de la `Configuration` prototype. Ceci peut sembler un peu strict, mais le code de fragmentation a besoin de supposer que tous les fragments sont configurés de la même manière.

Si vous examinez ce code et pensez qu'il semble un peu trop stupide pour fournir des documents de configuration pleinement formés qui, pour économiser deux propriétés spéciales, sont ignorés, soyez rassurés, nous avons regardé ce code et pensé la même chose. Nous prévoyons de faire évoluer le mécanisme de configuration. Nous avons choisi ce mécanisme-ci parce qu'il autorisait la plus grande réutilisation de code de configuration qui était déjà disponible dans Hibernate Core.

Une fois que nous avons construit nos objets `Configuration`, nous avons besoin d'assembler une `ShardStrategyFactory` (ligne 8). Une `ShardStrategyFactory` est un objet qui sait comment créer les 3 types de stratégie que les programmeurs peuvent utiliser pour contrôler le comportement de fragmentation du système. Pour plus d'informations à propos de ces stratégies, veuillez regarder les chapitres intitulés "Stratégies de fragmentation".

Maintenant que nous avons instancié notre `ShardStrategyFactory`, nous pouvons construire une `ShardedConfiguration` (ligne 9), et une fois que nous avons notre `ShardedConfiguration` nous pouvons lui demander de créer une `ShardedSessionFactory` (ligne 13). Il est important de noter que `ShardedSessionFactory` étend `SessionFactory`. Ceci signifie que nous pouvons retourner une `SessionFactory` standard (ligne 1). Le code Hibernate de notre application n'a pas besoin de savoir qu'il interagit avec des données fragmentées.

Examinons maintenant les fichiers de configuration et de mapping que nous avons chargé. Vous les reconnaîtrez, mais il y a quelques ajouts et modifications clef en rapport avec la fragmentation.

```
1    <!-- Contenu de shard0.hibernate.cfg.xml -->
2    <hibernate-configuration>
3      <session-factory name="HibernateSessionFactory0"> <!-- notez le nom différent -->
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        <property name="hibernate.connection.shard_id">0</property> <!-- nouveau -->
10       <property name="hibernate.shard.enable_cross_shard_relationship_checks">true</property> <!-
11    </session-factory>
12  </hibernate-configuration>
```

```
1    <!-- Contenu de shard1.hibernate.cfg.xml -->
2    <hibernate-configuration>
3      <session-factory name="HibernateSessionFactory1"> <!-- notez le nom différent -->
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        <property name="hibernate.connection.shard_id">1</property> <!-- nouveau -->
10       <property name="hibernate.shard.enable_cross_shard_relationship_checks">true</property> <!-
11    </session-factory>
12  </hibernate-configuration>
```

Nous passerons outre le contenu de shard2.hibernate.cfg.xml puisqu'il devrait être évident. Nous donnons à chaque session factory un nom unique via l'attribut "name" de l'élément "session-factory", et nous leur donnons aussi un identifiant de fragment. Ceci est obligatoire. Si vous essayez de configurer une `ShardedSessionFactory` avec un objet `Configuration` qui n'a pas d'identifiant de fragment, vous obtiendrez une erreur. Actuellement nous obligeons à ce que l'idenfiant de fragment d'une des session factory soit 0. Au-delà de ça, la représentation interne d'un identifiant de fragment est un, donc toutes les valeurs dans cette plage sont légales. Finalement, chaque fragment qui est mappé dans une `ShardedSessionFactory` doit avoir un identifiant unique. Si vous avez un fragment dupliqué, vous aurez une erreur.

L'autre ajout notable est la propriété, plutôt verbeuse mais heureusement descriptive, "hibernate.shard.enable_cross_shard_relationship_checks.". Vous pouvez lire d'avantage à ce propos dans le chapitre sur les limitations.

Maintenant regardons de nouveau comment le fichier de mapping a changé.

```
<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>
```

Le seul changement signficatif dans le fichier de mapping par rapport à la version sans fragmentation est dans notre sélection d'un générateur d'idenfitiant pour données fragmentées. Nous couvrirons cette génération d'identifiants plus en détail dans le chapitre sur les stratégies de fragmentation.

## 2.3. Limitations de la configuration

Beaucoup d'entre vous réaliserons rapidement que le mécanisme de configuration que nous avons fourni ne fonctionnera pas si vous configurez votre `SessionFactory` via JPA ou Hibernate Annotations. C'est vrai. Nous espérons que ces insuffisances seront corrigées sous peu.

# Chapitre 3. Shard Strategy

## 3.1. Vue d'ensemble

Hibernate Shards vous donne une énorme flexibilité pour configurer la manière dont vos données sont réparties à travers vos fragments et la façon d'interroger vos données à travers vos fragments. Le point d'entrée pour cette configuration est l'interface `org.hibernate.shards.strategy.ShardStrategy` :

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

Comme vous pouvez le voir, une `ShardStrategy` est composée de trois sous-stratégies. Nous parlerons d'elles chacune leur tour.

## 3.2. ShardAccessStrategy

Nous commencerons avec la plus simple des stratégies : `ShardAccessStrategy`. Hibernate Shards utilise la `ShardAccessStrategy` pour déterminer comment appliquer les opérations de base de données à travers plusieurs fragments. La `ShardAccessStrategy` est consultée lorsque vous exécutez une requête sur vos fragments. Nous avons déjà fourni deux implémentations de cette interface que nous pensons suffisantes pour la majorité des applications.

### 3.2.1. SequentialShardAccessStrategy

`SequentialShardAccessStrategy` se comporte exactement comme l'indique son nom : les requêtes sont exécutées en séquence sur vos fragments. Selon le type de requêtes que vous exécutez, vous pouvez vouloir éviter cette implémentation parce qu'elle exécutera les requêtes à travers les fragments dans le même ordre à chaque fois. Si vous exécutez beaucoup de requêtes limitées en nombre de lignes retournées et non triées, ceci *pourrait* donner lieu à une pauvre utilisation de vos fragments (les fragments apparaissant en tête de liste seront harcelés, et ceux en fin de liste resteront là à ne rien faire, à se croiser les doigts). Si ceci vous concerne, vous devriez plutôt envisager d'utiliser la `LoadBalancedSequentialShardAccessStrategy`. Cette implémentation reçoit une vue alternée de vos fragments à chaque invocation, et ainsi distribue également la charge de requêtes.

### 3.2.2. ParallelShardAccessStrategy

`ParallelShardAccessStrategy` se comporte exactement comme l'indique son nom : les requêtes sont exécutées sur les fragments en parallèle. Lorsque vous utilisez cette implémentation, vous avez besoin de fournir un `java.util.concurrent.ThreadPoolExecutor` qui soit approprié aux performances et aux besoins de votre application. Voici un simple exemple :

```
    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);
```

Veuillez noter que ce sont juste des valeurs d'exemple - une configuration propre d'un pool de threads va au-delà de la portée de ce document.

## 3.3. ShardSelectionStrategy

Hibernate Shards utilise la `ShardSelectionStrategy` pour déterminer le fragment sur lequel un nouvel objet devrait être créé. Il vous revient entièrement de décider à quoi doit ressembler l'implémentation de cette interface, mais nous avons fourni une implémentation round-robin pour commencer (`RoundRobinShardSelectionStrategy`). Nous espérons que de nombreuses applications voudrons implémenter une fragmentation basée sur les attributs, ainsi pour notre application d'exemple qui stocke les rapports météo fragmentons les rapports par continent dont les rapports sont originaires :

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

Il est important de noter que si un graphe d'objets multi-niveau est sauvegardé via la fonctionnalité de cascade d'Hibernate, la `ShardSelectionStrategy` sera seulement consultée lors de la sauvegarde de l'objet de plus haut niveau. Tous les objets enfants seront automatiquement sauvegardés sur le même fragment que le parent. Vous pouvez trouver votre `ShardSelectionStrategy` plus facile à implémenter si vous empêcher les développeur de créer de nouveaux objets à plus d'un niveau dans votre hiérarchie d'objets. Vous vous pouvez accomplir cela en informant votre `ShardSelectionStrategy` des objets de plus haut niveau de votre modèle, et ainsi lever une exception si elle rencontre un objet qui ne fait pas partie de cet ensemble. Si vous ne souhaitez pas imposer cette restriction, souvenez-vous juste que si vous effectuez une sélection des fragments basée sur les attributs, les attributs que vous utilisez pour prendre votre décision ont besoin d'être disponibles sur chaque objet qui est passé à session.save().

## 3.4. ShardResolutionStrategy

Hibernate Shards utilise la `ShardResolutionStrategy` pour déterminer l'ensemble des fragments sur lesquels un objet avec un identifiant donné peut résider. Revenons à notre application de rapports météorologiques et supposons, par exemple, que chaque continent a un éventail d'identifiants qui lui sont associés. N'importe quand nous assignons un identifiant à un WeatherReport, nous en prenons un qui tombe dans l'intervalle légal pour le continent auquel le WeatherReport appartient. Notre `ShardResolutionStrategy` peut utiliser cette information pour identifier sur quel fragment un WeatherReport réside simplement en regardant l'identifiant :

```
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);
    }
}
```

Il est intéressant de montrer que nous n'avons pas (encore) implémenté de cache qui mette en correspondance le nom de l'entité et l'identifiant du fragment, la `ShardResolutionStrategy` serait un excellent endroit pour brancher un tel cache.

Shard Resolution est étroitement lié à la génération d'identifiants. Si vous sélectionnez un générateur d'identifiants pour votre classe qui code l'identifiant du fragment dans l'identifiant de l'objet, votre `ShardResolutionStrategy` ne sera plus jamais appelée. Si vous avez l'intention d'utiliser seulement des générateurs d'identifiant qui codent l'identifiant du fragment dans les identifiants de vos objets, vous devriez utiliser `AllShardsShardResolutionStrategy` en tant que `ShardResolutionStrategy`.

## 3.5. Génération d'identifiants

Hibernate Sharding prend en charge n'importe quelle stratégie de génération d'identifiant ; le seul pré-requis est que les identifiants d'objet soient uniques à travers tous les fragments. Il y a quelques simples stratégies de génération d'identifiant qui prennent en charge ce pré-requis :

- *Génération native d'identifiants* - utilisez la stratégie de génération d'identfiants `native` d'Hibernate, et configurez vos bases de données de manière à ce que les identifiants n'entrent jamais en collision. Par exemple, si vous utilisez la génération d'identifiants `identity`, vous avez 5 bases de données à travers lesquelles vous répartirez les données de manière égale, et vous ne vous attendez pas à n'avoir jamais plus d'1 million d'enregistrements, vous pourriez configurer la base de données 0 pour retourner des identifiants commençant à 0, la base de données 1 pour retourner des identifiants commençant à 200000, la base de données 2 pour retourner des identifiants commençant à 400000, etc. Tant que vos suppositions concernant les données sont correctes, les identifiants de vos objets n'entreront jamais en collision.

- *Génération d'UUID au niveau applicatif* - par définition vous ne devez pas vous préoccuper des collisions d'identifiants, mais vous avez besoin d'être disposé à traiter les clefs primaires peu maniables de nos objets.

  Hibernate Shards fournit une implémentation d'un générateur d'UUID simple et prenant en compte les fragments - `ShardedUUIDGenerator`.

- *Génération hilo répartie* - l'idée est d'avoir une table hilo sur un seul fragment, lequel assure que les identifiants générés par l'algorithme hi/lo sont uniques à travers tous les fragments. Les deux principaux inconvénients de cette approche sont que les accès à la table hilo peuvent devenir le goulot d'étranglement dans la génération d'identifiants, et que stocker la table hilo sur une seule base de données crée un seul point de panne du système.

  Hibernate Shards fournit une implémentation de l'algorithme de génération hilo répartie - `ShardedTableHiLoGenerator`. Cette implémentation est basée sur `org.hibernate.id.TableHiLoGenerator`, donc pour des informations sur la structure attendue de la table de la base de données table de laquelle l'implémentation dépend, veuillez lire la documentation de cette classe.

La génération d'identifiants est aussi étroitement liée à la résolution de fragment. L'objectif de la résolution de fragment est de trouve le fragment sur lequel vit un objet, pour un identifiant d'objet donné. Il y a deux manières d'accomplir cela :

- Utiliser la `ShardResolutionStrategy`, décrite au-dessus
- Coder l'identifiant du fragment dans l'identifiant de l'objet durant la génération de l'identifiant, et récupérer l'identifiant du fragment pendant la réslution du fragment

Le principal avantage de coder l'identifiant du fragment dans l'identifiant de l'objet est que cela permet à Hibernate Shards de résoudre le fragment à partir de l'identifiant de l'objet beaucoup plus rapidement, sans recherche en base de données, sans recherche dans un cache, etc. Hibernate Shards ne requiert aucun algorithme spécifique pour coder/décoder l'identifiant d'un fragment - tout ce que vous avezà faire est d'utiliser un générateur d'identifiants qui implémente l'interface `ShardEncodingIdentifierGenerator`. Des deux générateurs d'identifiants inclus dans Hibernate Shards, le `ShardedUUIDGenerator` implémente cette interface.

# Chapitre 4. Refragmentation

Quand un ensemble de données d'une application grossit au-delà de la capacité des bases de données allouée à l'application, il devient nécessaire d'ajouter plus de bases de données, et il est souvent désirable de redistribuer les données à travers les fragments (soit pour réussir une répartition des charges propre, soit pour satisfaire les invariants de l'application) : ceci s'appelle la refragmentation. La refragmentation est problème compliqué, et elle peut être la source de complication majeures dans la gestion de votre application de production si elle n'est pas prise en compte durant la conception. Pour atténuer le supplice associé à la refragmentation, Hibernate Shards fournit la prise en charge de fragments virtuels.

## 4.1. Fragments virtuels

Dans le cas général, chaque objet vit sur un fragment. Refragmenter consiste en deux tâches : déplacer l'objet vers un autre fragment, et changer les mappings objet-fragment. Le mapping objet-fragment est capturé soit par l'identifiant du fragment codé dans l'identifiant de l'objet, soit par la logique interne de la stratégie de résolution du fragment que l'objet utilise. Dans le premier cas, refragmenter demanderait de changer tous les identifiants des objets et les clefs étrangères. Dans le second cas, refragmenter pourrait demander n'importe quoi allant du changement de configuration d'exécution d'une `ShardResolutionStrategy` donnée au changement d'algorithme de la `ShardResolutionStrategy`. Malheureusement, le problème de changer des mappings objet-fragment devient même pire une fois que l'on prend en compte le fait que Hibernate Shards ne prend pas en charge les relations à travers plusieurs fragments. Cette limitation nous empêche de déplacer un sous-ensemble d'un graphe d'objets d'un fragment vers un autre.

La tâche de changer de mapping objet-fragment peut être simplifiée en ajoutant un niveau d'indirection - chaque objet vit sur un fragment virtuel, et chaque fragment virtuel est mappé vers un fragment physique. Durant la conception, les développeurs doivent décider du nombre maximum de fragments physiques dont l'application aura besoin. Ce maximum est alors utilisé comme le nombre de fragments virtuels, et ces fragments virtuels sont alors mappés vers des fragments physiques actuellement requis par l'application. Puisque tous les `ShardSelectionStrategy`, `ShardResolutionStrategy`, et `ShardEncodingIdentifierGenerator` d'Hibernate Shards opèrent sur des fragments virtuels, les objets seront répartis correctement à travers les fragments virtuels. Durant la refragmentation, les mappings objet-fragment peuvent maintenant être simplement modifié en changeant les mappings de fragments virtuels vers des fragments physiques.

Si vous vous inquiétez à propos de l'estimation correcte du nombre maximum de fragments physiques dont votre application a besoin, visez haut. Les fragments virtuels ne coûtent pas grand chose. En fin de compte, vous serez bien mieux avec des fragments virtuels en trop plutôt que de devoir en rajouter.

Pour activer la fragmentation virtuelle, vous avez besoin de créer votre `ShardedConfiguration` avec une Map des identifiants des fragments virtuels vers les identifiants des fragments physiques. Voici un exemple où nous avons 4 fragments virtuels mappés vers 2 fragments physiques.

```
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();
```

Pour ensuite transformer le fragment virtuel en mapping de fragments physiques, il faut seulement changer la `virtualShardToShardMap` passée au constructeur.

Nous avons mentionné que la deuxième tâche durant la repartition est de déplacer les données d'un fragment physique vers un autre. Hibernate Shards n'essaie pas de fournir de prise en charge automatique pour cela puisque c'est en général très spécifique à l'application, et la complexité varie selon le besoin potentiel de refragmentation à chaud, de l'architecture de déploiement de l'application, etc.

# Chapitre 5. Requêtes

## 5.1. Vue d'ensemble

Exécuter des requêtes à travers des fragments peut être difficile. Dans ce chapitre nous parlerons de ce qui fonctionne, ce qui ne fonctionne pas, et ce que vous pouvez faire pour éviter les ennuis.

## 5.2. Criteria

Comme nous en avons parlé dans le chapitre sur les limitations, nous n'avons pas encore d'implémentation complète de l'API Hibernate Core. Cette limitation s'applique à `ShardedCriteriaImpl`, qui est une implémentation de l'interface `Criteria` prenant en compte les fragments. Dans ce chapitre nous n'entrerons pas dans les détails des choses spécifiques qui n'ont pas été implémentées. Nous allons plutôt discuter des types de requêtes `Criteria` qui sont problématiques dans un environnement fragmenté.

Dis simplement, les requêtes qui effectuent des tris posent problème. Pourquoi ? Parce que nous ne pouvons pas retourner une liste proprement triée sans la capacité de comparer une valeur de la liste à toute autre valeur de la liste, et la liste entière n'est pas disponible jusqu'à ce que tous les résultats des requêtes individuelles aient été collectés dans la partie applicative. Le tri a besoin de s'effectuer à l'intérieur d'Hibernate Shards, et pour que cela arrive, nous demandons à tous les objets retournés par une requête `Criteria` avec une clause "order-by" d'implémenter l'interface `Comparable`. Si le type des objets que vous retournez n'implémente pas cette interface, vous aurez une exception.

Les clauses "distinct" posent aussi problème. Tellement de problèmes, en fait, que pour le moment nous les prenons même pas en charge. Désolé pour ça.

D'un autre côté, alors que "distinct" et "order-by" posent problèmes, les agrégats fonctionnent bien. Considérez l'exemple suivante :

```
        // récupère la moyenne de toutes les températures enregistrées depuis jeudi dernier
        Criteria crit = session.createCriteria(WeatherReport.class);
        crit.add(Restrictions.gt("timestamp", lastThursday));
        crit.setProjection(Projections.avg("temperature"));
        return crit.list();
```

Dans un environnement avec un seul fragment, cette requête peut obtenir une réponse facilement, mais dans un environnement avec plusieurs fragments c'est un peu plus embêtant. Pourquoi ? Parce qu'obtenir la moyenne de chaque fragment n'est pas suffisant pour calculer la moyenne à travers tous les fragments. Pour calculer ce morceau d'informations, nous n'avons pas seulement besoin de la moyenne mais du nombre d'enregistrements de chaque fragment. C'est exactement ce que nous faisons, et l'impact sur les performances (faire un "count" supplémentaire dans chaque requête) est probablement négligeable. Maintenant, si nous voulions la médiane, nous aurions des problèmes (ajouter le "count" à la requête ne fournirait pas assez d'informations pour réaliser le calcul), mais pour le moment `Criteria` n'expose pas de fonction médiane, donc nous traiterons ça lorsque cela arrivera et sera un problème.

## 5.3. HQL

Notre prise en charge de HQL n'est pas, pour le moment, aussi bon que la prise en charge des requêtes `Criteria`. Nous n'avons pas encore implémenté d'extensions à l'analyseur lexico-syntaxique de requêtes, donc

nous prenons pas en charge "distinct", "order-by", ou les agrégats. Cela signifie que vous pouvez seulement utiliser HQL pour des requêtes très simples. Vous feriez probablement mieux d'éviter le HQL de cette version si vous le pouvez.

## 5.4. Use of Shard Strategy When Querying

Le seul composant de votre stratégie de fragmentation qui est consulté lors de l'exécution d'une requête (`Criteria` ou HQL) est la `ShardAccessStrategy`. `ShardSelectionStrategy` est ignorée parce qu'exécuter une requête ne crée pas de nouvel enregistrement dans la base de données. `ShardResolutionStrategy` est ignorée parce qu'actuellement nous partons du principe que vous voulez toujours que votre requête soit exécutée sur tous les fragments. Si ce n'est pas le cas, la meilleure chose à faire de transtyper votre `Session` en une `ShardedSession` et d'en extraire la `Session` spécifique au fragment dont vous avez besoin. Maladroit, mais ça fonctionne. Nous proposerons une meilleure solution dans une prochaine version.

# Chapitre 6. Limitations

## 6.1. Implémentation incomplète de l'API Hibernate

Pour accélérer la sortie initiale d'Hibernate Shards, certaines parties de l'API Hibernate que nous utilisons rarement n'ont pas été implémentées. Bien sûr, des choses que nous utilisons rarement sont probablement critiques pour d'autres applications, donc si nous vous avons délaissé, nous nous en excusons. Nous prévoyons d'implémenter le reste de l'API rapidement. Pour savoir quelles méthodes ne sont pas implémentées, veuillez voir la javadoc de `ShardedSessionImpl`, `ShardedCriteriaImpl`, et `ShardedQueryImpl`.

## 6.2. Graphes d'objets inter-fragments

Hibernate Shards ne prend pas en charge actuellement les graphes d'objets inter-fragments.

En d'autres mots, il est illégal de créer une association entre des objets A et B quand A et B vivent sur des fragments différents. Pour contourner cela, il faut définir une propriété sur A qui identifie de manière unique un objet de type B, et l'utiliser pour charger un objet B (Vous souvenez-vous comment la vie était avant Hibernate ? Oui, juste comme ça).

Par exemple :

```
                --besoin d'un domaine pour les exemples--
```

Dans certaines applications, votre modèle peut être construit de telle manière qu'il est difficile de faire ce genre d'erreur, mais dans d'autres ça peut être plus facile. La chose effrayante ici est que si vous faîtes cette erreur, Hibernate prendra en compte le "mauvais" objet dans la liste pour en faire un nouvel objet et, supposant que vous avez activé les opérations en cascade pour cette relation, créera une nouvelle version de cet objet sur un fragment différent. C'est le problème. Pour aider à éviter ce genre de chose, nous avons un intercepteur appelé `CrossShardRelationshipDetectingInterceptor` qui vérifie les relations inter-fragments pour tous les objets qui sont créés ou sauvegargés.

Malheureusement il y a un coût associé à l'utilisation de `CrossShardRelationshipDetectingInterceptor`. Pour déterminer le fragment sur lequel un objet associé réside, nous avons besoin de récupérer l'objet en base de données, donc si vous avez des associations chargées à la demande l'intercepteur résoudra ces associations comme partie de ses vérifications. C'est potentiellement assez coûteux, et peut ne pas être approprié pour un système de production. Avec ça en tête, nous avons simplifié l'activation ou non de cette vérification via la propriété "hibernate.shard.enable_cross_shard_relationship_checks" que nous avons référencé dans le chapitre sur la configuration. Si cette propriété est positionnée à "true", un `CrossShardRelationshipDetectingInterceptor` sera inscrit à chaque `ShardedSession` créée. Ne vous inquiétez pas, vous pouvez toujours inscrire votre propre intercepteur. Notre attente est que la plupart des applications auront activé cette vérification dans leurs environnements de développement et d'assurance qualité, et désactivé dans leurs environnements de tests et de production.

## 6.3. Transactions réparties

Hibernate Shards ne fournit pas de prise en charge pour les transactions réparties dans un environnement non géré. Si votre application requiert des transactions réparties, vous avez besoin de brancher une implémentation

de gestion de transactions qui prend en charge les transactions réparties.

## 6.4. Intercepteurs à état

Nous avons fait de notre mieux pour que, dans l'ensemble, le code d'Hibernate Core s'exécute bien lors de l'utilisation d'Hibernate Shards. Il y a, malheureusement, des exceptions, et une d'entre elles est quand votre application a besoin d'utliser un `org.hibernate.Interceptor` qui maintient son état.

Les intercepteurs à état (NdT : stateful) ont besoin d'un traitement particulier parce que, sous le capot, nous instancions une `org.hibernate.SessionImpl` par fragment. Si nous voulons un `Interceptor` associé à la `Session`, nous avons besoin de passer par l'`Interceptor`, quelqu'il soit, qui était fourni quand la `ShardedSession` a été créée. Si cet `Interceptor` est à état, l'état de l'`Interceptor` pour une `Session` sera visible dans toutes les `Sessions`. Si vous réfléchissez aux choses qui sont typiquement faites dans des `Interceptors` à état (audit par exemple), vous pouvez voir comment cela peut poser problème.

Notre solution est d'obliger les utilisateurs à fournir une `StatefulInterceptorFactory` quand ils créent leurs objets `Session` (lesquels sont réllement des `ShardedSessions`). Si l'`Interceptor` fourni implémente cette interface, Hibernate Shards assurera qu'une nouvelle instance du type de `Interceptor` retournée par `StatefulInterceptorFactory.newInstance()` sera passée à chaque `Session` qui est créée sous le capot. Voici un exemple :

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

Beaucoup d'implémentations d'`Interceptor` requièrent une référence à la `Session` à laquelle elles sont associées. Dans le cas d'un `Interceptor` à état, vous voulez que votre `Interceptor` ait une référence à la `Session` réelle (spécifique au fragment). Pour faciliter cela, vous avez le choix d'avoir le type d'`Interceptor` qui est construit par la `StatefulInterceptorFactory` [...] Si l'`Interceptor` contruit par la `StatefulInterceptorFactory` implémente cette interface, Hibernate Shards fournira 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;
    }

    ... // Implémentation de l'interface Interceptor
}
```

Vu la nature fondamentale du problème, nous ne nous attendons pas à changer cela de suite.

## 6.5. Des objets avec des identifiants qui sont des types de

## base

Avec Hibernate, peu importe ce que vos objets du modèle utilisent commen identifiant tant qu'il peut être représenté par un `Serializable` (ou encapsulé automatiquement dans un `Serializable`). Avec Hibernate Shards vous êtres légèrement plus contraints parce que nous ne prenons pas en charge les types de base.

Ainsi, ceci n'est pas bon :

```
public class WeatherReport {
    private int weatherReportId;  // problème

    public int getWeatherReportId() {
        return weatherReportId;
    }

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

Mais ceci est adorable :

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

    public Integer getWeatherReportId() {
        return weatherReportId;
    }

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

Avons-nous une bonne raison à cette limitation ? Pas réellement. C'est le résultat d'un choix d'implémentation qui a filtré et qui a un peu aggravé la vie de chacun. Si vous devez simplement utiliser Hibernate Shards et modéliser vos identifiants avec des types de base, n'appelez pas `Session.saveOrUpdate`. Nous avons pour objectif de trouver une solution à ce problème bientôt et de vous laisser modéliser comme bon vous semble (malgré cela, nous préférerions des identifiants objet parce qu'ils permettent de déterminer plus facilement si un objet s'est vu assigner un identifiant ou pas).

# 6.6. Données répliquées

Même si c'est un framework pour le partionnement horizontal, il y a pratiquement toujours des données en lecture seules (ou du moins changeant rarement) qui vivent dans chaque fragment. Si vous lisez juste ces entités nous n'avons pas de problème, mais si vous voulez associer ces entités avec des entités fragmentées nous allons avoir des problèmes. Supposez que vous ayez une table Country sur chaque fragment avec exactement les mêmes données, et supposez que WeatherReport a un membre Country. Comment garantissons-nous que le Country que vous associez à ce WeatherReport est associé au même fragment que celui du WeatherReport ? Si nous nous trompons, nous finirons avec une relation entre plusieurs fragments, et ce n'est pas bien.

Nous avons des idées pour rendre cela facile à traiter, mais nous ne les avons pas encore implémentées. Pour faire court, nous pensons qu'il est plus sûr pour vous de ne pas créer relations objet entre des entités fragmentées et des entités répliquées. En d'autres mots, modélisez seulement la relation comme vous feriez si vous n'utilisiez pas d'ORM. Nous savons que c'est maladroit et annuyant. Nous nous en occuperons bientôt.