



Hibernate Search

Apache Lucene Integration

Version: 3.0.1.GA

Table of Contents

Preface	iv
1. Getting started	1
1.1. System Requirements	1
1.2. Maven	1
1.3. Configuration	2
1.4. Indexing	3
1.5. Searching	4
1.6. Analyzer	5
1.7. What's next	6
2. Architecture	7
2.1. Overview	7
2.2. Back end	8
2.2.1. Lucene	8
2.2.2. JMS	8
2.3. Work execution	9
2.3.1. Synchronous	9
2.3.2. Asynchronous	10
2.4. Reader strategy	10
2.4.1. Shared	10
2.4.2. Not-shared	10
2.4.3. Custom	10
3. Configuration	11
3.1. Directory configuration	11
3.2. Index sharding	12
3.3. Worker configuration	13
3.4. JMS Master/Slave configuration	14
3.4.1. Slave nodes	14
3.4.2. Master node	15
3.5. Reader strategy configuration	16
3.6. Enabling Hibernate Search and automatic indexing	16
3.6.1. Enabling Hibernate Search	16
3.6.1.1. Hibernate Core 3.2.6 and beyond	17
3.6.2. Automatic indexing	17
3.7. Tuning Lucene indexing performance	17
4. Mapping entities to the index structure	20
4.1. Mapping an entity	20
4.1.1. Basic mapping	20
4.1.2. Mapping properties multiple times	21
4.1.3. Embedded and associated objects	22
4.1.4. Boost factor	24
4.1.5. Analyzer	25
4.2. Property/Field Bridge	26
4.2.1. Built-in bridges	26
4.2.2. Custom Bridge	26
4.2.2.1. StringBridge	27
4.2.2.2. FieldBridge	28
4.2.2.3. @ClassBridge	29
5. Querying	31

5.1. Building queries	31
5.1.1. Building a Lucene query	31
5.1.2. Building a Hibernate Search query	32
5.1.2.1. Generality	32
5.1.2.2. Pagination	32
5.1.2.3. Sorting	32
5.1.2.4. Fetching strategy	32
5.1.2.5. Projection	33
5.2. Retrieving the results	34
5.2.1. Performance considerations	34
5.2.2. Result size	34
5.2.3. ResultTransformer	35
5.3. Filters	35
5.4. Optimizing the query process	37
5.5. Native Lucene Queries	38
6. Manual indexing	39
6.1. Indexing	39
6.2. Purging	39
7. Index Optimization	41
7.1. Automatic optimization	41
7.2. Manual optimization	41
7.3. Adjusting optimization	42
8. Accessing Lucene natively	43
8.1. SearchFactory	43
8.2. Accessing a Lucene Directory	43
8.3. Using an IndexReader	43

Preface

Full text search engines like Apache Lucene™ are very powerful technologies to add efficient free text search capabilities to applications. However, they suffer several mismatches when dealing with object domain models. Amongst other things indexes have to be kept up to date and mismatches between index structure and domain model as well as query mismatches have to be avoided.

Hibernate Search indexes your domain model with the help of a few annotations, takes care of database/index synchronization and brings back regular managed objects from free text queries. To achieve this Hibernate Search is combining the power of Hibernate [<http://www.hibernate.org>] and Apache Lucene [<http://lucene.apache.org>].

Chapter 1. Getting started

Welcome to Hibernate Search! The following chapter will guide you through the initial steps required to integrate Hibernate Search into an existing Hibernate enabled application. In case you are a Hibernate new timer we recommend you start here [<http://hibernate.org/152.html>].

1.1. System Requirements

Table 1.1. System requirements

Java Runtime	A JDK or JRE version 5 or greater. You can download a Java Runtime for Windows/Linux/Solaris here [http://java.sun.com/javase/downloads/].
Hibernate Search	<code>hibernate-search.jar</code> and all the dependencies from the <code>lib</code> directory of the Hibernate Search distribution, especially lucene :)
Hibernate Core	This instructions have been tested against Hibernate 3.2.x. Next to the main <code>hibernate3.jar</code> you will need all required libraries from the <code>lib</code> directory of the distribution. Refer to <code>README.txt</code> in the <code>lib</code> directory of the distibution to determine the minimum runtime requirements.
Hibernate Annotations	Even though Hibernate Search can be used without Hibernate Annotations the following instructions will use them for ease of use. The tutorial is tested against version 3.3.x of Hibernate Annotations.

You can download all dependencies from the Hibernate download site [<http://www.hibernate.org/6.html>]. You can also verify the dependency versions against the Hibernate Compatibility Matrix [<http://www.hibernate.org/6.html#A3>].

1.2. Maven

Instead of managing all dependencies yourself maven users have the possibility to use the JBoss maven repository [<http://repository.jboss.com/maven2>]. Just add the JBoss repository url to the *repositories* section of your `pom.xml` or `settings.xml`:

```
<repository>
  <id>repository.jboss.org</id>
  <name>JBoss Maven Repository</name>
  <url>http://repository.jboss.com/maven2</url>
  <layout>default</layout>
</repository>
```

Then add the following dependencies to your `pom.xml`:

```

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search</artifactId>
  <version>3.0.0.ga</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>3.3.0.ga</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>3.3.1.ga</version>
</dependency>

```

Not all three dependencies are required. *hibernate-search* alone contains everything needed to use Hibernate Search. *hibernate-annotations* is only needed if you use non Hibernate Search annotations like we do in the examples of this tutorial. Last but not least, *hibernate-entitymanager* is only required if you use Hibernate Search in conjunction with JPA.

1.3. Configuration

Once you have downloaded and added all required dependencies to your application you have to add a few properties to your hibernate configuration file. If you are using Hibernate directly this can be done in `hibernate.properties` or `hibernate.cfg.xml`. If you are using Hibernate via JPA you can also add the properties to `persistence.xml`. The good news is that for standard use most properties offer a sensible default.

Apache Lucene has a notion of `Directory` to store the index files. Hibernate Search handles the initialization and configuration of a Lucene `Directory` instance via a `DirectoryProvider`. In this tutorial we will use a subclass of `DirectoryProvider` called `FSDirectoryProvider`. This will give us the ability to physically inspect the Lucene indexes created by Hibernate Search (eg via Luke [<http://www.getopt.org/luke/>]). Once you have a working configuration you can start experimenting with other directory providers (see Section 3.1, “Directory configuration”).

Lets assume that your application contains the Hibernate managed class `example.Book` and you now want to add free text search capabilities to your application in order to search body and summary of the books contained in your database.

```

package exmaple.Book
...
@Entity
public class Book {

    @Id
    private Integer id;
    private String body;
    private String summary;
    @ManyToMany private Set<Author> authors = new HashSet<Author>();
    @ManyToOne private Author mainAuthor;
    private Date publicationDate;

    public Book() {
    }

    // standard getters/setters follow here
    ...
}

```

First you have to tell Hibernate Search which `DirectoryProvider` to use. This can be achieved by setting the `hibernate.search.default.directory_provider` property. You also have to specify the default root directory for all indexes via `hibernate.search.default.indexBase`.

```
...
# the default directory provider
hibernate.search.default.directory_provider = org.hibernate.search.store.FSDirectoryProvider

# the default base directory for the indecies
hibernate.search.default.indexBase = /var/lucene/indexes
...
```

Next you have to add three annotations to the `Book` class. The first annotation `@Indexed` marks `Book` as indexable. By design Hibernate Search needs to store an untokenized id in the index to ensure index unicity for a given entity. `@DocumentId` marks the property to use for this purpose. Most if not all the time, the property is the database primary key. Last but not least you have to index the fields you want to make searchable. In our example these fields are `body` and `summary`. Both properties get annotated with `@Field`. The property `index=Index.TOKENIZED` will ensure that the text will be tokenized using the default Lucene analyzer whereas `store=Store.NO` ensures that the actual data will not be stored in the index. Usually, tokenizing means chunking a sentence into individual words (and potentially excluding common words like `a`, `the` etc).

These settings are sufficient for an initial test. For more details on entity mapping refer to Section 4.1, “Mapping an entity”. In case you want to store and retrieve the indexed data in order to avoid database roundtrips, refer to projections in Section 5.1.2.5, “Projection”

```
package exmaple.Book
...
@Entity
@Indexed
public class Book {

    @Id
    @DocumentId
    private Integer id;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    private String body;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    private String summary;
    @ManyToMany private Set<Author> authors = new HashSet<Author>();
    @ManyToOne private Author mainAuthor;
    private Date publicationDate;

    public Book() {
    }

    // standard getters/setters follow here
    ...
}
```

1.4. Indexing

Hibernate Search will index every entity persisted, updated or removed through Hibernate core transparently for the application. However, the data already present in your database needs to be indexed once to populate the Lucene index. Once you have added the above properties and annotations it is time to trigger an initial batch index of your books. You can achieve this by adding one of the following code examples to your code (see also

Chapter 6, *Manual indexing*):

Example using Hibernate Session:

```
FullTextSession fullTextSession = Search.createFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
List books = session.createQuery("from Book as book").list();
for (Book book : books) {
    fullTextSession.index(book);
}
tx.commit(); //index are written at commit time
```

Example using JPA:

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager = Search.createFullTextEntityManager(em);
List books = em.createQuery("select book from Book as book").getResultList();
for (Book book : books) {
    fullTextEntityManager.index(book);
}
```

After executing the above code, you should be able to see a Lucene index under `var/lucene/indexes/example.Book`. Go ahead and inspect this index. It will help you to understand how Hibernate Search works.

1.5. Searching

Now it is time to execute a first search. The following code will prepare a query against the fields `summary` and `body`, execute it and return a list of `Books`:

Example using Hibernate Session:

```
FullTextSession fullTextSession = Search.createFullTextSession(session);

Transaction tx = fullTextSession.beginTransaction();

MultiFieldQueryParser parser = new MultiFieldQueryParser( new String[]{"summary", "body"},
    new StandardAnalyzer());
Query query = parser.parse( "Java rocks!" );
org.hibernate.Query hibQuery = fullTextSession.createFullTextQuery( query, Book.class );
List result = hibQuery.list();

tx.commit();
session.close();
```

Example using JPA:

```
EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =
    org.hibernate.hibernate.search.jpa.Search.createFullTextEntityManager(em);
MultiFieldQueryParser parser = new MultiFieldQueryParser( new String[]{"summary", "body"},
    new StandardAnalyzer());
Query query = parser.parse( "Java rocks!" );
org.hibernate.Query hibQuery = fullTextEntityManager.createFullTextQuery( query, Book.class );
List result = hibQuery.list();
```


1.6. Analyzer

Assume that one of your indexed book entities contains the text "Java rocks" and you want to get hits for all of the following queries: "rock", "rocks", "rocked" and "rocking". In Lucene this can be achieved by choosing an analyzer class which applies word stemming during the indexing process. Hibernate Search offers several ways to configure the analyzer to use (see Section 4.1.5, “Analyzer”):

- Setting the `hibernate.search.analyzer` property in the configuration file. The specified class will then be the default analyzer.
- Setting the `Analyzer` annotation at the entity level.
- Setting the `Analyzer` annotation at the field level.

The following example uses the entity level annotation to apply a English language analyzer which would help you to achieve your goal. The class `EnglishAnalyzer` is a custom class using the Snowball English Stemmer from the Lucene Sandbox [<http://lucene.apache.org/java/docs/lucene-sandbox/>].

```
package example.Book
...
@Entity
@Indexed
@Analyzer(impl = example.EnglishAnalyzer.class)
public class Book {

    @Id
    @DocumentId
    private Integer id;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    private String body;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    private String summary;
    @ManyToMany private Set<Author> authors = new HashSet<Author>();
    @ManyToOne private Author mainAuthor;
    private Date publicationDate;

    public Book() {
    }

    // standard getters/setters follow here
    ...
}

public class EnglishAnalyzer extends Analyzer {
    /**
     * {@inheritDoc}
     */
    @Override
    public TokenStream tokenStream(String fieldName, Reader reader) {
        TokenStream result = new StandardTokenizer(reader);
        result = new StandardFilter(result);
        result = new LowerCaseFilter(result);
        result = new SnowballFilter(result, name);
        return result;
    }
}
```

1.7. What's next

The above paragraphs hopefully helped you getting started with Hibernate Search. You should by now have a file system based index and be able to search and retrieve a list of managed objects via Hibernate Search. The next step is to get more familiar with the overall architecture ((Chapter 2, *Architecture*)) and explore the basic features in more detail.

Two topics which where only briefly touched in this tutorial were analyzer configuration (Section 4.1.5, “Analyzer”) and field bridges (Section 4.2, “Property/Field Bridge”), both important features required for more fine-grained indexing.

More advanced topics cover clustering (Section 3.4, “JMS Master/Slave configuration”) and large indexes handling (Section 3.2, “Index sharding”).

Chapter 2. Architecture

2.1. Overview

Hibernate Search consists of an indexing and an index search engine. Both are backed by Apache Lucene.

When an entity is inserted, updated or removed in/from the database, Hibernate Search keeps track of this event (through the Hibernate event system) and schedules an index update. All the index updates are handled for you without you having to use the Apache Lucene APIs (see Section 3.6, “Enabling Hibernate Search and automatic indexing”).

To interact with Apache Lucene indexes, Hibernate Search has the notion of `DirectoryProviders`. A directory provider will manage a given Lucene `Directory` type. You can configure directory providers to adjust the directory target (see Section 3.1, “Directory configuration”).

Hibernate Search can also use the Lucene index to search an entity and return a list of managed entities saving you the tedious object to Lucene document mapping. The same persistence context is shared between Hibernate and Hibernate Search; as a matter of fact, the Search Session is built on top of the Hibernate Session. The application code use the unified `org.hibernate.Query` or `javax.persistence.Query` APIs exactly the way a HQL, JPA-QL or native queries would do.

To be more efficient, Hibernate Search batches the write interactions with the Lucene index. There is currently two types of batching depending on the expected scope.

Outside a transaction, the index update operation is executed right after the actual database operation. This scope is really a no scoping setup and no batching is performed.

It is however recommended, for both your database and Hibernate Search, to execute your operation in a transaction be it JDBC or JTA. When in a transaction, the index update operation is scheduled for the transaction commit and discarded in case of transaction rollback. The batching scope is the transaction. There are two immediate benefits:

- Performance: Lucene indexing works better when operation are executed in batch.
- ACIDity: The work executed has the same scoping as the one executed by the database transaction and is executed if and only if the transaction is committed.

Note

Disclaimer, the work is not ACID in the strict sense of it, but ACID behavior is rarely useful for full text search indexes since they can be rebuilt from the source at any time.

You can think of those two scopes (no scope vs transactional) as the equivalent of the (infamous) autocommit vs transactional behavior. From a performance perspective, the *in transaction* mode is recommended. The scoping choice is made transparently: Hibernate Search detects the presence of a transaction and adjust the scoping.

Note

Hibernate Search works perfectly fine in the Hibernate / EntityManager long conversation pattern aka. atomic conversation.

Note

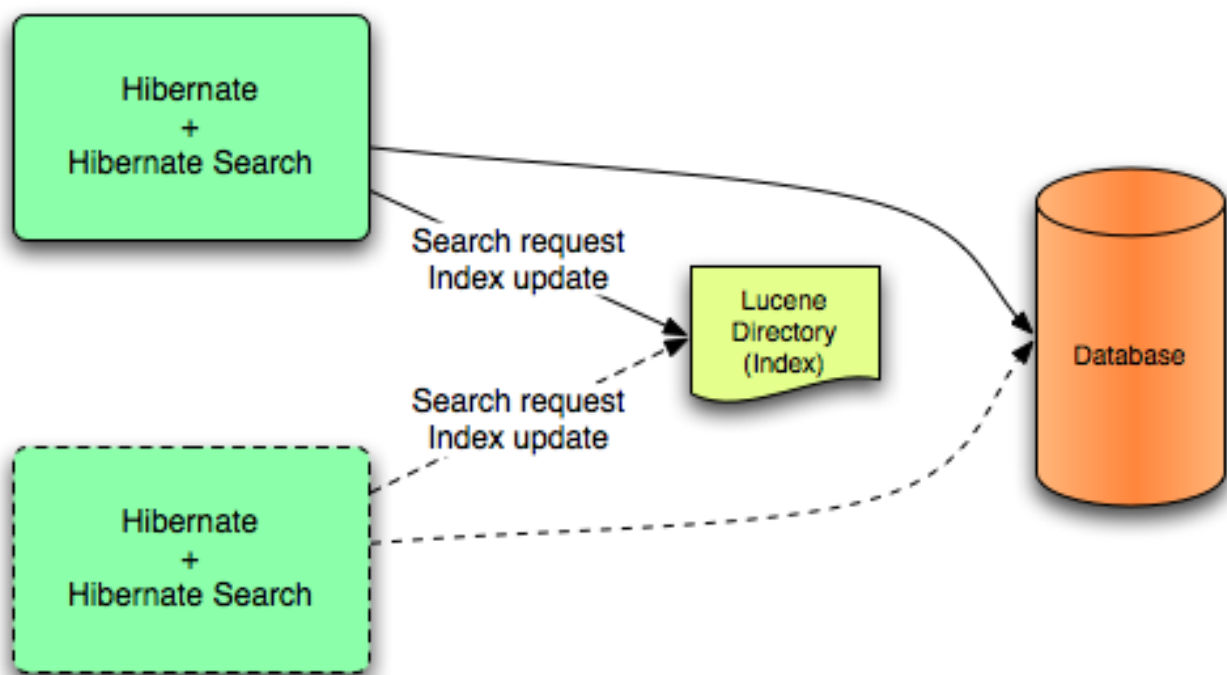
Depending on user demand, additional scoping will be considered, the pluggability mechanism being already in place.

2.2. Back end

Hibernate Search offers the ability to let the scoped work being processed by different back ends. Two back ends are provided out of the box for two different scenarios.

2.2.1. Lucene

In this mode, all index update operations applied on a given node (JVM) will be executed to the Lucene directories (through the directory providers) by the same node. This mode is typically used in non clustered environment or in clustered environments where the directory store is shared.

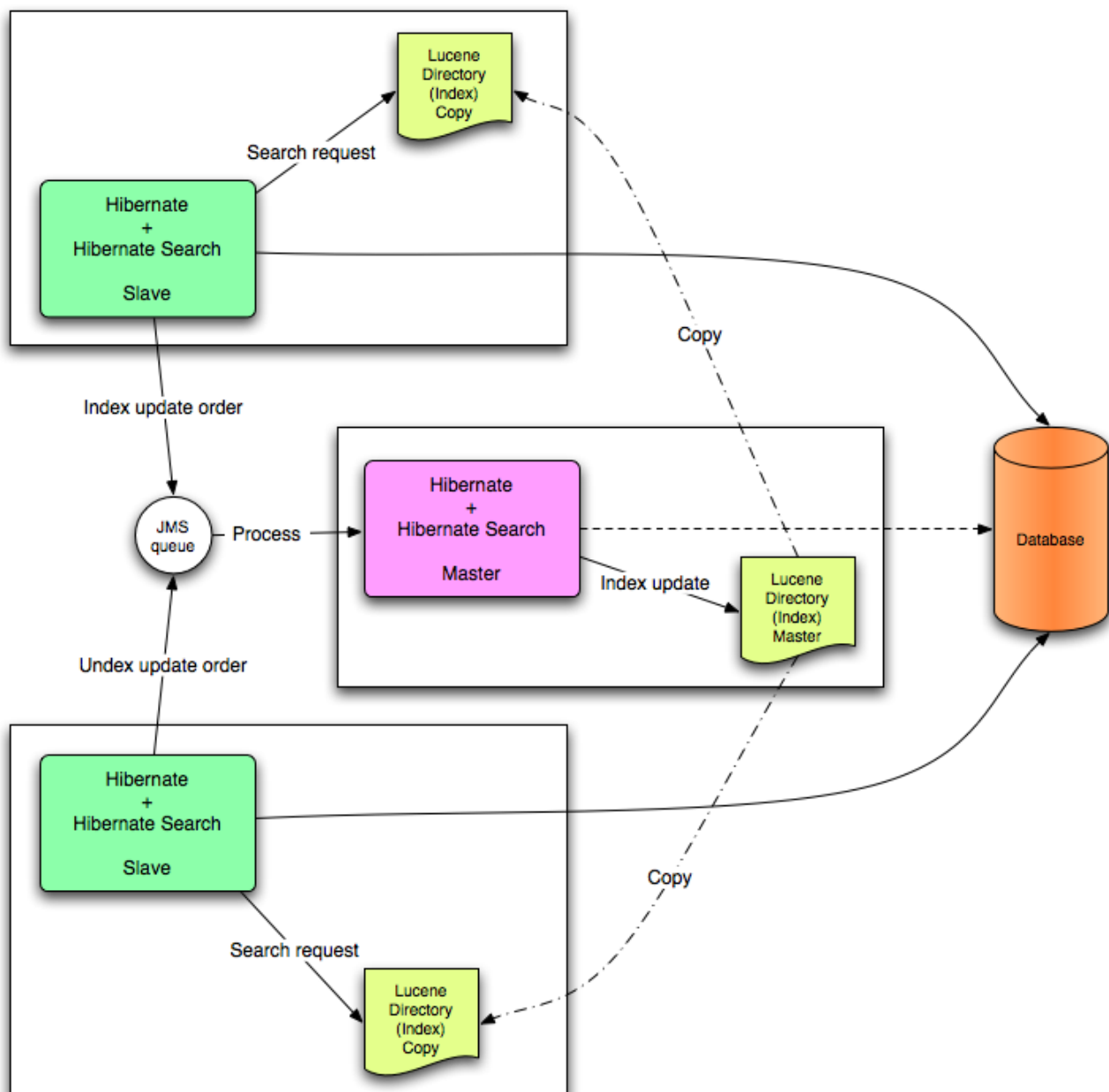


This mode targets non clustered applications, or clustered applications where the Directory is taking care of the locking strategy.

The main advantage is simplicity and immediate visibility of the changes in Lucene queries (a requirement is some applications).

2.2.2. JMS

All index update operations applied on a given node are sent to a JMS queue. A unique reader will then process the queue and update the master Lucene index. The master index is then replicated on a regular basis to the slave copies. This is known as the master / slaves pattern. The master is the sole responsible for updating the Lucene index. The slaves can accept read as well as write operations. However, they only process the read operation on their local index copy and delegate the update operations to the master.



This mode targets clustered environments where throughput is critical, and index update delays are affordable. Reliability is ensured by the JMS provider and by having the slaves working on a local copy of the index.

Note

Hibernate Search is an extensible architecture. While not yet part of the public API, plugging a third party back end is possible. Feel free to drop ideas to hibernate-dev@lists.jboss.org.

2.3. Work execution

The indexing work (done by the back end) can be executed synchronously with the transaction commit (or update operation if out of transaction), or asynchronously.

2.3.1. Synchronous

This is the safe mode where the back end work is executed in concert with the transaction commit. Under highly concurrent environment, this can lead to throughput limitations (due to the Apache Lucene lock mechanism).

ism) and it can increase the system response time if the backend is significantly slower than the transactional process and if a lot of IO operations are involved.

2.3.2. Asynchronous

This mode delegates the work done by the back end to a different thread. That way, throughput and response time are (to a certain extent) decorrelated from the back end performance. The drawback is that a small delay appears between the transaction commit and the index update and a small overhead is introduced to deal with thread management.

It is recommended to use synchronous execution first and evaluate asynchronous execution if performance problems occur and after having set up a proper benchmark (ie not a lonely cowboy hitting the system in a completely unrealistic way).

2.4. Reader strategy

When executing a query, Hibernate Search interacts with the Apache Lucene indexes through a reader strategy. Choosing a reader strategy will depend on the profile of the application (frequent updates, read mostly, asynchronous index update etc). See also Section 3.5, “Reader strategy configuration”

2.4.1. Shared

With this strategy, Hibernate Search will share the same `IndexReader`, for a given Lucene index, across multiple queries and threads provided that the `IndexReader` is still up-to-date. If the `IndexReader` is not up-to-date, a new one is opened and provided. Generally speaking, this strategy provides much better performances than the `not-shared` strategy. It is especially true if the number of updates is much lower than the reads. This strategy is the default.

2.4.2. Not-shared

Every time a query is executed, a Lucene `IndexReader` is opened. This strategy is not the most efficient since opening and warming up an `IndexReader` can be a relatively expensive operation.

2.4.3. Custom

You can write your own reader strategy that suits your application needs by implementing `org.hibernate.search.reader.ReaderProvider`. The implementation must be thread safe.

Note

Some additional strategies are planned in future versions of Hibernate Search

Chapter 3. Configuration

3.1. Directory configuration

Apache Lucene has a notion of `Directory` to store the index files. The `Directory` implementation can be customized, but Lucene comes bundled with a file system (`FSDirectoryProvider`) and a in memory (`RAMDirectoryProvider`) implementation. Hibernate Search has the notion of `DirectoryProvider` that handles the configuration and the initialization of the Lucene `Directory`.

Table 3.1. List of built-in Directory Providers

Class	Description	Properties
<code>org.hibernate.search.store.FSDirectoryProvider</code>	File system based directory. The directory used will be <code><indexBase>/< @Indexed.name ></code>	<code>indexBase</code> : Base directory <code>indexName</code> : override <code>@Index.name</code> (useful for sharded indexes)
<code>org.hibernate.search.store.FSMasterDirectoryProvider</code>	<p>File system based directory. Like <code>FSDirectoryProvider</code>. It also copies the index to a source directory (aka copy directory) on a regular basis.</p> <p>The recommended value for the refresh period is (at least) 50% higher than the time to copy the information (default 3600 seconds - 60 minutes).</p> <p>Note that the copy is based on an incremental copy mechanism reducing the average copy time.</p> <p><code>DirectoryProvider</code> typically used on the master node in a JMS back end cluster.</p> <p><code>DirectoryProvider</code> typically used on slave nodes using a JMS back end.</p>	<code>indexBase</code> : Base directory <code>indexName</code> : override <code>@Index.name</code> (useful for sharded indexes) <code>sourceBase</code> : Source (copy) base directory. <code>source</code> : Source directory suffix (default to <code>@Indexed.name</code>). The actual source directory name being <code><sourceBase>/<source></code> <code>refresh</code> : refresh period in second (the copy will take place every refresh seconds).
<code>org.hibernate.search.store.FSSlaveDirectoryProvider</code>	<p>File system based directory. Like <code>FSDirectoryProvider</code>, but retrieves a master version (source) on a regular basis. To avoid locking and inconsistent search results, 2 local copies are kept.</p> <p>The recommended value for the refresh period is (at least) 50% higher than the time to copy the inform-</p>	<code>indexBase</code> : Base directory <code>indexName</code> : override <code>@Index.name</code> (useful for sharded indexes) <code>sourceBase</code> : Source (copy) base directory. <code>source</code> : Source directory suffix (default to <code>@Indexed.name</code>). The

Class	Description	Properties
	<p>ation (default 3600 seconds - 60 minutes).</p> <p>Note that the copy is based on an incremental copy mechanism reducing the average copy time.</p> <p>DirectoryProvider typically used on slave nodes using a JMS back end.</p>	<p>actual source directory name being <code><sourceBase>/<source></code></p> <p>refresh: refresh period in second (the copy will take place every refresh seconds).</p>
<code>org.hibernate.search.store.RAMDirectoryProvider</code>	Memory based directory, the directory will be uniquely identified (in the same deployment unit) by the <code>@Indexed.name</code> element	none

If the built-in directory providers does not fit your needs, you can write your own directory provider by implementing the `org.hibernate.store.DirectoryProvider` interface

Each indexed entity is associated to a Lucene index (an index can be shared by several entities but this is not usually the case). You can configure the index through properties prefixed by `hibernate.search.indexname`. Default properties inherited to all indexes can be defined using the prefix `hibernate.search.default`.

To define the directory provider of a given index, you use the `hibernate.search.indexname.directory_provider`

```
hibernate.search.default.directory_provider org.hibernate.search.store.FSDirectoryProvider
hibernate.search.default.indexBase=/usr/lucene/indexes

hibernate.search.Rules.directory_provider org.hibernate.search.store.RAMDirectoryProvider
```

applied on

```
@Indexed(name="Status")
public class Status { ... }

@Indexed(name="Rules")
public class Rule { ... }
```

will create a file system directory in `/usr/lucene/indexes/Status` where the Status entities will be indexed, and use an in memory directory named `Rules` where Rule entities will be indexed.

You can easily define common rules like the directory provider and base directory, and override those default later on on a per index basis.

Writing your own `DirectoryProvider`, you can utilize this configuration mechanism as well.

3.2. Index sharding

In some extreme cases involving huge indexes (in size), it is necessary to split (shard) the indexing data of a given entity type into several Lucene indexes. This solution is not recommended until you reach significant index sizes and index update time are slowing down. The main drawback of index sharding is that searches will

end up being slower since more files have to be opened for a single search. In other words don't do it until you have problems :)

Despite this strong warning, Hibernate Search allows you to index a given entity type into several sub indexes. Data is sharded into the different sub indexes thanks to an `IndexShardingStrategy`. By default, no sharding strategy is enabled, unless the number of shards is configured. To configure the number of shards use the following property

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards 5
```

This will use 5 different shards.

The default sharding strategy, when shards are set up, splits the data according to the hash value of the id string representation (generated by the Field Bridge). This ensures a fairly balanced sharding. You can replace the strategy by implementing `IndexShardingStrategy` and by setting the following property

```
hibernate.search.<indexName>.sharding_strategy my.shardingstrategy.Implementation
```

Each shard has an independent directory provider configuration as described in Section 3.1, “Directory configuration”. The `DirectoryProvider` default name for the previous example are `<indexName>.0` to `<indexName>.4`. In other words, each shard has the name of its owning index followed by `.` (dot) and its index number.

```
hibernate.search.default.indexBase /usr/lucene/indexes

hibernate.search.Animal.sharding_strategy.nbr_of_shards 5
hibernate.search.Animal.directory_provider org.hibernate.search.store.FSDirectoryProvider
hibernate.search.Animal.0.indexName Animal00
hibernate.search.Animal.3.indexBase /usr/lucene/sharded
hibernate.search.Animal.3.indexName Animal03
```

This configuration uses the default id string hashing strategy and shards the `Animal` index into 5 subindexes. All subindexes are `FSDirectoryProvider` instances and the directory where each subindex is stored is as follows:

- for subindex 0: `/usr/lucene/indexes/Animal00` (shared `indexBase` but overridden `indexName`)
- for subindex 1: `/usr/lucene/indexes/Animal.1` (shared `indexBase`, default `indexName`)
- for subindex 2: `/usr/lucene/indexes/Animal.2` (shared `indexBase`, default `indexName`)
- for subindex 3: `/usr/lucene/sharded/Animal03` (overridden `indexBase`, overridden `indexName`)
- for subindex 4: `/usr/lucene/indexes/Animal.4` (shared `indexBase`, default `indexName`)

3.3. Worker configuration

It is possible to refine how Hibernate Search interacts with Lucene through the worker configuration. The work can be executed to the Lucene directory or sent to a JMS queue for later processing. When processed to the Lucene directory, the work can be processed synchronously or asynchronously to the transaction commit.

You can define the worker configuration using the following properties

Table 3.2. worker configuration

Property	Description
<code>hibernate.worker.backend</code>	Out of the box support for the Apache Lucene back end and the JMS back end. Default to <code>lucene</code> . Supports also <code>jms</code> .
<code>hibernate.worker.execution</code>	Supports synchronous and asynchronous execution. Default to <code>sync</code> . Supports also <code>async</code> .
<code>hibernate.worker.thread_pool.size</code>	Defines the number of threads in the pool. useful only for asynchronous execution. Default to 1.
<code>hibernate.worker.buffer_queue.max</code>	Defines the maximal number of work queue if the thread poll is starved. Useful only for asynchronous execution. Default to infinite. If the limit is reached, the work is done by the main thread.
<code>hibernate.worker.jndi.*</code>	Defines the JNDI properties to initiate the InitialContext (if needed). JNDI is only used by the JMS back end.
<code>hibernate.worker.jms.connection_factory</code>	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS connection factory from (<code>java:/ConnectionFactory</code> by default in JBoss AS)
<code>hibernate.worker.jms.queue</code>	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS queue from. The queue will be used to post work messages.
<code>hibernate.worker.batch_size</code>	Defines the maximum number of elements indexed before flushing the transaction-bound queue. Default to 0 (ie no limit). See Chapter 6, <i>Manual indexing</i> for more information.

3.4. JMS Master/Slave configuration

This section describes in greater detail how to configure the Master / Slaves Hibernate Search architecture.

3.4.1. Slave nodes

Every index update operation is sent to a JMS queue. Index quering operations are executed on a local index copy.

```
### slave configuration

## DirectoryProvider
# (remote) master location
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local copy location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = org.hibernate.search.store.FSSlaveDirectoryProvider
```

```
## Backend configuration
hibernate.search.worker.backend = jms
hibernate.search.worker.jms.connection_factory = java:/ConnectionFactory
hibernate.search.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your JMS provider for more information)

## Optional asynchronous execution strategy
# org.hibernate.worker.execution = async
# org.hibernate.worker.thread_pool.size = 2
# org.hibernate.worker.buffer_queue.max = 50
```

A file system local copy is recommended for faster search results.

The refresh period should be higher than the expected time copy.

3.4.2. Master node

Every index update operation is taken from a JMS queue and executed. The master index(es) is(are) copied on a regular basis.

```
### master configuration

## DirectoryProvider
# (remote) master location where information is copied to
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local master location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = org.hibernate.search.store.FSMasterDirectoryProvider

## Backend configuration
#Backend is the default lucene one
```

The refresh period should be higher than the expected time copy.

In addition to the Hibernate Search framework configuration, a Message Driven Bean should be written and set up to process index works queue through JMS.

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination", propertyValue="queue/hibernatesearch"),
    @ActivationConfigProperty(propertyName="DLQMaxResent", propertyValue="1")
} )
public class MDBSearchController extends AbstractJMSHibernateSearchController implements MessageListener {
    @PersistenceContext EntityManager em;

    //method retrieving the appropriate session
    protected Session getSession() {
        return (Session) em.getDelegate();
    }

    //potentially close the session opened in #getSession(), not needed here
    protected void cleanSessionIfNeeded(Session session)
    {
    }
}
```

This example inherits the abstract JMS controller class available and implements a JavaEE 5 MDB. This implementation

mentation is given as an example and, while most likely more complex, can be adjusted to make use of non Java EE Message Driven Beans. For more information about the `getSession()` and `cleanSessionIfNeeded()`, please check `AbstractJMSHibernateSearchController`'s javadoc.

Note

Hibernate Search test suite makes use of JBoss Embedded to test the JMS integration. It allows the unit test to run both the MDB container and JBoss Messaging (JMS provider) in a standalone way (marketed by some as "lightweight").

3.5. Reader strategy configuration

The different reader strategies are described in Reader strategy. The default reader strategy is `shared`. This can be adjusted:

```
hibernate.search.reader.strategy = not-shared
```

Adding this property switch to the `non shared` strategy.

Or if you have a custom reader strategy:

```
hibernate.search.reader.strategy = my.corp.myapp.CustomReaderProvider
```

where `my.corp.myapp.CustomReaderProvider` is the custom strategy implementation

3.6. Enabling Hibernate Search and automatic indexing

3.6.1. Enabling Hibernate Search

Hibernate Search is enabled out of the box when using Hibernate Annotations or Hibernate EntityManager. If, for some reason you need to disable it, set `hibernate.search.autoregister_listeners` to `false`. Note that there is no performance runtime when the listeners are enabled while no entity is indexable.

To enable Hibernate Search in Hibernate Core, add the `FullTextIndexEventListener` for the three Hibernate events that occur after changes are executed to the database. Once again, such a configuration is not useful with Hibernate Annotations or Hibernate EntityManager.

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="post-update"/>
      <listener class="org.hibernate.search.event.FullTextIndexEventListener"/>
    </event>
    <event type="post-insert"/>
      <listener class="org.hibernate.search.event.FullTextIndexEventListener"/>
    </event>
    <event type="post-delete"/>
      <listener class="org.hibernate.search.event.FullTextIndexEventListener"/>
    </event>
  </session-factory>
</hibernate-configuration>
```

Be sure to add the appropriate jar files in your classpath. Check `lib/README.TXT` for the list of third party lib-

aries. A typical installation on top of Hibernate Annotations will add:

- `hibernate-search.jar`: the core engine
- `lucene-core-*.jar`: Lucene core engine

3.6.1.1. Hibernate Core 3.2.6 and beyond

If you use Hibernate Core 3.2.6 and beyond, make sure to add three additional event listeners that cope with collection events

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="post-collection-recreate"/>
      <listener class="org.hibernate.search.event.FullTextIndexCollectionEventListener"/>
    </event>
    <event type="post-collection-remove"/>
      <listener class="org.hibernate.search.event.FullTextIndexCollectionEventListener"/>
    </event>
    <event type="post-collection-update"/>
      <listener class="org.hibernate.search.event.FullTextIndexCollectionEventListener"/>
    </event>
  </session-factory>
</hibernate-configuration>
```

Those additional event listeners have been introduced in Hibernate 3.2.6. note the `FullTextIndexCollectionEventListener` usage. You need to explicitly reference those event listeners unless you use [Hibernate Annotations 3.3.1](#) and above.

3.6.2. Automatic indexing

By default, every time an object is inserted, updated or deleted through Hibernate, Hibernate Search updates the according Lucene index. It is sometimes desirable to disable that features if either your index is read-only or if index updates are done in a batch way (see [Chapter 6, Manual indexing](#)).

To disable event based indexing, set

```
hibernate.search.indexing_strategy manual
```

Note

In most case, the JMS backend provides the best of both world, a lightweight event based system keeps track of all changes in the system, and the heavyweight indexing process is done by a separate process or machine.

3.7. Tuning Lucene indexing performance

Hibernate Search allows you to tune the Lucene indexing performance by specifying a set of parameters which are passed through to underlying Lucene `IndexWriter` such as `mergeFactor`, `maxMergeDocs` and `maxBufferedDocs`. You can specify these parameters either as default values applying for all indexes or on a per index basis.

There are two sets of parameters allowing for different performance settings depending on the use case. During indexing operations triggered by database modifications, the following ones are used:

- `hibernate.search.[default|<indexname>].transaction.merge_factor`
- `hibernate.search.[default|<indexname>].transaction.max_merge_docs`
- `hibernate.search.[default|<indexname>].transaction.max_buffered_docs`

When indexing occurs via `FullTextSession.index()` (see Chapter 6, *Manual indexing*), the following properties are used:

- `hibernate.search.[default|<indexname>].batch.merge_factor`
- `hibernate.search.[default|<indexname>].batch.max_merge_docs`
- `hibernate.search.[default|<indexname>].batch.max_buffered_docs`

Unless the corresponding `.batch` property is explicitly set, the value will default to the `.transaction` property.

For more information about Lucene indexing performances, please refer to the Lucene documentation.

Table 3.3. List of indexing performance properties

Property	Description	Default Value
<code>hibernate.search.[default <indexname>].transaction.merge_factor</code>	<p>Controls segment merge frequency and size.</p> <p>Determines how often segment indices are merged when insertion occurs. With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indices that are interactively maintained. The value must no be lower than 2.</p> <p>Used by Hibernate Search during index update operations as part of database modifications.</p>	10
<code>hibernate.search.[default <indexname>].transaction.max_merge_docs</code>	<p>Defines the largest number of documents allowed in a segment.</p> <p>Used by Hibernate Search during index update operations as part of database modifications.</p>	Unlimited (Integer.MAX_VALUE)

Property	Description	Default Value
<code>hibernate.search.[default <indexname>].transaction.max_buffered_docs</code>	<p>Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.</p> <p>Used by Hibernate Search during index update operations as part of database modifications.</p>	10
<code>hibernate.search.[default <indexname>].batch.merge_factor</code>	<p>Controls segment merge frequency and size.</p> <p>Determines how often segment indices are merged when insertion occurs. With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indices that are interactively maintained. The value must no be lower than 2.</p> <p>Used during indexing via <code>FullTextSession.index()</code></p>	10
<code>hibernate.search.[default <indexname>].batch.max_merge_docs</code>	<p>Defines the largest number of documents allowed in a segment.</p> <p>Used during indexing via <code>FullTextSession.index()</code></p>	Unlimited (Integer.MAX_VALUE)
<code>hibernate.search.[default <indexname>].batch.max_buffered_docs</code>	<p>Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.</p> <p>Used during indexing via <code>FullTextSession.index()</code></p>	10

Chapter 4. Mapping entities to the index structure

All the metadata information needed to index entities is described through some Java annotations. There is no need for xml mapping files nor a list of indexed entities. The list is discovered at startup time scanning the Hibernate mapped entities.

4.1. Mapping an entity

4.1.1. Basic mapping

First, we must declare a persistent class as indexable. This is done by annotating the class with `@Indexed` (all entities not annotated with `@Indexed` will be ignored by the indexing process):

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

The `index` attribute tells Hibernate what the Lucene directory name is (usually a directory on your file system). If you wish to define a base directory for all Lucene indexes, you can use the `hibernate.search.default.indexBase` property in your configuration file. Each entity instance will be represented by a Lucene `Document` inside the given index (aka `Directory`).

For each property (or attribute) of your entity, you have the ability to describe how it will be indexed. The default (ie no annotation) means that the property is completely ignored by the indexing process. `@Field` does declare a property as indexed. When indexing an element to a Lucene document you can specify how it is indexed:

- `name` : describe under which name, the property should be stored in the Lucene Document. The default value is the property name (following the JavaBeans convention)
- `store` : describe whether or not the property is stored in the Lucene index. You can store the value `Store.YES` (consuming more space in the index but allowing projection, see Section 5.1.2.5, “Projection” for more information), store it in a compressed way `Store.COMPRESS` (this does consume more CPU), or avoid any storage `Store.NO` (this is the default value). When a property is stored, you can retrieve it from the Lucene Document (note that this is not related to whether the element is indexed or not).
- `index`: describe how the element is indexed (ie the process used to index the property and the type of information store). The different values are `Index.NO` (no indexing, ie cannot be found by a query), `Index.TOKENIZED` (use an analyzer to process the property), `Index.UN_TOKENISED` (no analyzer pre processing), `Index.NO_NORM` (do not store the normalization data). The default value is `TOKENIZED`.

These attributes are part of the `@Field` annotation.

Whether or not you want to store the data depends on how you wish to use the index query result. For a regular Hibernate Search usage, storing is not necessary. However you might want to store some fields to subsequently project them (see Section 5.1.2.5, “Projection” for more information).

Whether or not you want to tokenize a property depends on whether you wish to search the element as is, or by the words it contains. It make sense to tokenize a text field, but it does not to do it for a date field (or an id

field). Note that fields used for sorting must not be tokenized.

Finally, the `id` property of an entity is a special property used by Hibernate Search to ensure index unicity of a given entity. By design, an `id` has to be stored and must not be tokenized. To mark a property as index `id`, use the `@DocumentId` annotation.

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", index=Index.TOKENIZED, store=Store.YES)
    public String getSummary() { return summary; }

    @Lob
    @Field(index=Index.TOKENIZED)
    public String getText() { return text; }
}
```

These annotations define an index with three fields: `id`, `Abstract` and `text`. Note that by default the field name is decapitalized, following the JavaBean specification.

Note

You *must* specify `@DocumentId` on the identifier property of your entity class.

4.1.2. Mapping properties multiple times

It is sometimes needed to map a property multiple times per index, with slightly different indexing strategies. Especially, sorting a query by field requires the field to be `UN_TOKENIZED`. If one want to search by words in this property and still sort it, one need to index it twice, once tokenized, once untokenized. `@Fields` allows to achieve this goal.

```
@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
        @Field(index = Index.TOKENIZED),
        @Field(name = "summary_forSort", index = Index.UN_TOKENIZED, store = Store.YES)
    } )
    public String getSummary() {
        return summary;
    }

    ...
}
```

The field `summary` is indexed twice, once as `summary` in a tokenized way, and once as `summary_forSort` in an untokenized way. `@Field` supports 2 attributes useful when `@Fields` is used:

- analyzer: defines a `@Analyzer` annotation per field rather than per property
- bridge: defines a `@FieldBridge` annotation per field rather than per property

See below for more information about analyzers and field bridges.

4.1.3. Embedded and associated objects

Associated objects as well as embedded objects can be indexed as part of the root entity index. It is necessary if you expect to search a given entity based on properties of the associated object(s). In the following example, the use case is to return the places whose city is Atlanta (In the Lucene query parser language, it would translate into `address.city:Atlanta`).

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field( index = Index.TOKENIZED )
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}
```

In this example, the place fields will be indexed in the `Place` index. The `Place` index documents will also contain the fields `address.id`, `address.street`, and `address.city` which you will be able to query. This is enabled by the `@IndexedEmbedded` annotation.

Be careful. Because the data is denormalized in the Lucene index when using the `@IndexedEmbedded` technique, Hibernate Search needs to be aware of any change in the `Place` object and any change in the `Address` object to keep the index up to date. To make sure the `Place` Lucene document is updated when it's `Address` changes, you need to mark the other side of the bidirectional relationship with `@ContainedIn`.

`@ContainedIn` is only useful on associations pointing to entities as opposed to embedded (collection of) objects.

Let's make our example a bit more complex:

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;
```

```

    @Field( index = Index.TOKENIZED )
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}

@Embeddable
public class Owner {
    @Field(index = Index.TOKENIZED)
    private String name;
    ...
}

```

Any `@*ToOne` and `@Embedded` attribute can be annotated with `@IndexedEmbedded`. The attributes of the associated class will then be added to the main entity index. In the previous example, the index will contain the following fields

- `id`
- `name`
- `address.street`
- `address.city`
- `address.ownedBy_name`

The default prefix is `propertyName.`, following the traditional object navigation convention. You can override it using the `prefix` attribute as it is shown on the `ownedBy` property.

`depth` is necessary when the object graph contains a cyclic dependency of classes (not instances). For example, if `Owner` points to `Place`. Hibernate Search will stop including Indexed embedded attributes after reaching the expected depth (or the object graph boundaries are reached). A class having a self reference is an example of cyclic dependency. In our example, because `depth` is set to 1, any `@IndexedEmbedded` attribute in `Owner` (if any) will be ignored.

Such a feature (`@IndexedEmbedded`) is very useful to express queries referring to associated objects, such as:

- Return places where name contains JBoss and where address city is Atlanta. In Lucene query this would be

```
+name:jboss +address.city:atlanta
```

- Return places where name contains JBoss and where owner's name contain Joe. In Lucene query this would be

```
+name:jboss +address.orderBy_name:joe
```

In a way it mimics the relational join operation in a more efficient way (at the cost of data duplication). Remember that, out of the box, Lucene indexes have no notion of association, the join operation is simply non-existent. It might help to keep the relational model normalized while benefiting from the full text index speed and feature richness.

Note

An associated object can itself be (but don't have to) `@Indexed`

When `@IndexedEmbedded` points to an entity, the association has to be directional and the other side has to be annotated `@ContainedIn` (as see in the previous example). If not, Hibernate Search has no way to update the root index when the associated entity is updated (in our example, a `Place` index document has to be updated when the associated `Address` instance is updated).

Sometimes, the object type annotated by `@IndexedEmbedded` is not the object type targeted by Hibernate and Hibernate Search especially when interface are used in lieu of their implementation. You can override the object type targeted by Hibernate Search using the `targetElement` parameter.

```
@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field(index= Index.TOKENIZED)
    private String street;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_", targetElement = Owner.class)
    @Target(Owner.class)
    private Person ownedBy;

    ...
}

@Embeddable
public class Owner implements Person { ... }
```

4.1.4. Boost factor

Lucene has the notion of *boost factor*. It's a way to give more weight to a field or to an indexed element over another during the indexation process. You can use `@Boost` at the field or the class level.

```
@Entity
```

```

@Indexed(index="indexes/essays")
@Boost(2)
public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", index=Index.TOKENIZED, store=Store.YES)
    @Boost(2.5f)
    public String getSummary() { return summary; }

    @Lob
    @Field(index=Index.TOKENIZED)
    public String getText() { return text; }
}

```

In our example, Essay's probability to reach the top of the search list will be multiplied by 2 and the summary field will be 2.5 more important than the test field. Note that this explanation is actually wrong, but it is simple and close enough to the reality. Please check the Lucene documentation or the excellent *Lucene In Action* from Otis Gospodnetic and Erik Hatcher.

4.1.5. Analyzer

The default analyzer class used to index the elements is configurable through the `hibernate.search.analyzer` property. If none is defined, `org.apache.lucene.analysis.standard.StandardAnalyzer` is used as the default.

You can also define the analyzer class per entity, per property and even per `@Field` (useful when multiple fields are indexed from a single property).

```

@Entity
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field(index = Index.TOKENIZED)
    private String name;

    @Field(index = Index.TOKENIZED)
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(index = Index.TOKENIZED, analyzer = @Analyzer(impl = FieldAnalyzer.class))
    private String body;

    ...
}

```

In this example, `EntityAnalyzer` is used index all tokenized properties (eg. `name`), except for `summary` and `body` which are indexed with `PropertyAnalyzer` and `FieldAnalyzer` respectively.

Caution

Mixing different analyzers in the same entity is most of the time a bad practice. It makes query building more complex and results less predictable (for the novice), especially if you are using a `QueryParser`

(which uses the same analyzer for the whole query). As a thumb rule, the same analyzer should be used for both the indexing and the query for a given field.

4.2. Property/Field Bridge

In Lucene all index fields have to be represented as Strings. For this reason all entity properties annotated with `@Field` have to be indexed in a String form. For most of your properties, Hibernate Search does the translation job for you thanks to a built-in set of bridges. In some cases, though you need a more fine grain control over the translation process.

4.2.1. Built-in bridges

Hibernate Search comes bundled with a set of built-in bridges between a Java property type and its full text representation.

`null`

null elements are not indexed. Lucene does not support null elements and this does not make much sense either.

`java.lang.String`

String are indexed as is

`short`, `Short`, `integer`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`, `BigInteger`, `BigDecimal`

Numbers are converted in their String representation. Note that numbers cannot be compared by Lucene (ie used in ranged queries) out of the box: they have to be padded ¹

`java.util.Date`

Dates are stored as `yyyyMMddHHmmssSSS` in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). You shouldn't really bother with the internal format. What is important is that when using a `DateRange Query`, you should know that the dates have to be expressed in GMT time.

Usually, storing the date up to the millisecond is not necessary. `@DateBridge` defines the appropriate resolution you are willing to store in the index (`@DateBridge(resolution=Resolution.DAY)`). The date pattern will then be truncated accordingly.

```
@Entity
@Indexed
public class Meeting {
    @Field(index=Index.UN_TOKENIZED)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    ...
}
```

Warning

A Date whose resolution is lower than `MILLISECOND` cannot be a `@DocumentId`

4.2.2. Custom Bridge

¹Using a Range query is debatable and has drawbacks, an alternative approach is to use a Filter query which will filter the result query to the appropriate range.

Hibernate Search will support a padding mechanism

It can happen that the built-in bridges of Hibernate Search do not cover some of your property types, or that the String representation used is not what you expect. The following paragraphs several solutions for this problem.

4.2.2.1. StringBridge

The simplest custom solution is to give Hibernate Search™ an implementation of your expected *object to String* bridge. To do so you need to implements the `org.hibernate.search.bridge.StringBridge` interface

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < PADDING ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

Then any property or field can use this bridge thanks to the `@FieldBridge` annotation

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

Parameters can be passed to the Bridge implementation making it more flexible. The Bridge implementation implements a `ParameterizedBridge` interface, and the parameters are passed through the `@FieldBridge` annotation.

```
public class PaddedIntegerBridge implements StringBridge, ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}

//property
@FieldBridge(impl = PaddedIntegerBridge.class,
    params = @Parameter(name="padding", value="10")
```

```

    )
    private Integer length;

```

The `ParameterizedBridge` interface can be implemented by `StringBridge`, `TwoWayStringBridge`, `FieldBridge` implementations (see below).

If you expect to use your bridge implementation on for an id property (ie annotated with `@DocumentId`), you need to use a slightly extended version of `StringBridge` named `TwoWayStringBridge`. Hibernate Search needs to read the string representation of the identifier and generate the object out of it. There is not difference in the way the `@FieldBridge` annotation is used.

```

public class PaddedIntegerBridge implements TwoWayStringBridge, ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}

//id property
@DocumentId
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
private Integer id;

```

It is critically important for the two-way process to be idempotent (ie `object = stringToObject(objectToString(object))`).

4.2.2.2. FieldBridge

Some usecase requires more than a simple object to string translation when mapping a property to a Lucene index. To give you most of the flexibility you can also implement a bridge as a `FieldBridge`. This interface give you a property value and let you map it the way you want in your Lucene Document. This interface is very similar in its concept to the Hibernate™ `UserType`.

You can for example store a given property in two different document fields

```

/**
 * Store the date in 3 different field year, month, day
 * to ease Range Query per year, month or day
 * (eg get all the elements of december for the last 5 years)
 *
 * @author Emmanuel Bernard
 */

```



```

public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name, Object value, Document document, Field.Store
        store, Field.Index index, Float boost) {

        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance( GMT );
        cal.setTime( date );
        int year = cal.get( Calendar.YEAR );
        int month = cal.get( Calendar.MONTH ) + 1;
        int day = cal.get( Calendar.DAY_OF_MONTH );
        //set year
        Field field = new Field( name + ".year", String.valueOf(year), store, index );
        if ( boost != null ) field.setBoost( boost );
        document.add( field );
        //set month and pad it if needed
        field = new Field( name + ".month", month < 10 ? "0" : "" + String.valueOf(month), store, index );
        if ( boost != null ) field.setBoost( boost );
        document.add( field );
        //set day and pad it if needed
        field = new Field( name + ".day", day < 10 ? "0" : "" + String.valueOf(day), store, index );
        if ( boost != null ) field.setBoost( boost );
        document.add( field );
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Integer length;

```

4.2.2.3. @ClassBridge

It is sometimes useful to combine more than one property of a given entity and index this combination in a specific way into the Lucene index. The `@ClassBridge` and `@ClassBridges` annotations can be defined at the class level (as opposed to the property level). In this case the custom field bridge implementation receives the entity instance as the value parameter instead of a particular property.

```

@Entity
@Indexed
@ClassBridge(name="branchnetwork",
    index=Index.TOKENIZED,
    store=Store.YES,
    impl = CatFieldsClassBridge.class,
    params = @Parameter( name="sepChar", value=" " ) )
public class Department {
    private int id;
    private String network;
    private String branchHead;
    private String branch;
    private Integer maxEmployees;
    ...
}

public class CatFieldsClassBridge implements FieldBridge, ParameterizedBridge {

    private String sepChar;

    public void setParameterValues(Map parameters) {
        this.sepChar = (String) parameters.get( "sepChar" );
    }

    public void set(String name,
        Object value, //the department instance (entity) in this case
        Document document, //the Lucene document
        Field.Store store, Field.Index index, Float boost) {

```

```
// In this particular class the name of the new field was passed
// from the name field of the ClassBridge Annotation. This is not
// a requirement. It just works that way in this instance. The
// actual name could be supplied by hard coding it below.
Department dep = (Department) value;
String fieldValue1 = dep.getBranch();
if ( fieldValue1 == null ) {
    fieldValue1 = "";
}
String fieldValue2 = dep.getNetwork();
if ( fieldValue2 == null ) {
    fieldValue2 = "";
}
String fieldValue = fieldValue1 + sepChar + fieldValue2;
Field field = new Field( name, fieldValue, store, index );
if ( boost != null ) field.setBoost( boost );
document.add( field );
}
```

In this example, the particular `CatFieldsClassBridge` is applied to the `department` instance, the field bridge then concatenate both branch and network and index the concatenation.

Chapter 5. Querying

The second most important capability of Hibernate Search is the ability to execute a Lucene query and retrieve entities managed by an Hibernate session, providing the power of Lucene without living the Hibernate paradigm, and giving another dimension to the Hibernate classic search mechanisms (HQL, Criteria query, native SQL query).

To access the Hibernate Search™ querying facilities, you have to use an `org.hibernate.Session`. A `Search Session` wraps a regular `org.hibernate.Session` to provide query and indexing capabilities.

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession = Search.createFullTextSession(session);
```

The search facility is built on native Lucene queries.

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse( "summary:Festina Or brand:Seiko" );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );

List result = fullTextQuery.list(); //return a list of managed objects
```

The Hibernate query built on top of the Lucene query is a regular `org.hibernate.Query`, you are in the same paradigm as the other Hibernate query facilities (HQL, Native or Criteria). The regular `list()`, `uniqueResult()`, `iterate()` and `scroll()` can be used.

For people using Java Persistence (aka EJB 3.0 Persistence) APIs of Hibernate, the same extensions exist:

```
EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =
    org.hibernate.hibernate.search.jpa.Search.createFullTextEntityManager(em);

...
org.apache.lucene.queryParser.QueryParser parser = new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse( "summary:Festina Or brand:Seiko" );
javax.persistence.Query fullTextQuery = fullTextEntityManager.createFullTextQuery( luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of managed objects
```

The following examples show the Hibernate APIs but the same example can be easily rewritten with the Java Persistence API by just adjusting the way the `FullTextQuery` is retrieved.

5.1. Building queries

Hibernate Search queries are built on top of Lucene queries. It gives you a total freedom on the kind of Lucene queries you are willing to execute. However, once built, Hibernate Search abstract the query processing from your application using `org.hibernate.Query` as your primary query manipulation API.

5.1.1. Building a Lucene query

This subject is generally speaking out of the scope of this documentation. Please refer to the Lucene documentation or Lucene In Action.

5.1.2. Building a Hibernate Search query

5.1.2.1. Generality

Once the Lucene query is built, it needs to be wrapped into an Hibernate Query.

```
FullTextSession fullTextSession = Search.createFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
```

If not specified otherwise, the query will be executed against all indexed entities, potentially returning all types of indexed classes. It is advised, from a performance point of view, to restrict the returned types:

```
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Customer.class
//or
fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

The first example returns only matching customers, the second returns matching actors and items.

5.1.2.2. Pagination

It is recommended to restrict the number of returned objects per query. It is a very common use case as well, the user usually navigate from one page to an other. The way to define pagination is exactly the way you would define pagination in a plain HQL or Criteria query.

```
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Customer.class
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```

Note

It is still possible to get the total number of matching elements regardless of the pagination. See `getResultSize()` below

5.1.2.3. Sorting

Apache Lucene provides a very flexible and powerful way to sort results. While the default sorting (by relevance) is appropriate most of the time, it can be interesting to sort by one or several properties.

Inject the Lucene Sort object to apply a Lucene sorting strategy to an Hibernate Search.

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query, Book.class );
org.apache.lucene.search.Sort sort = new Sort(new SortField("title"));
query.setSort(sort);
List results = query.list();
```

One can notice the `FullTextQuery` interface which is a sub interface of `org.hibernate.Query`.

Fields used for sorting must not be tokenized.

5.1.2.4. Fetching strategy

When you restrict the return types to one class, Hibernate Search loads the objects using a single query. It also respects the static fetching strategy defined in your domain model.

It is often useful, however, to refine the fetching strategy for a specific use case.

```
Criteria criteria = s.createCriteria( Book.class ).setFetchMode( "authors", FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

In this example, the query will return all Books matching the luceneQuery. The authors collection will be loaded from the same query using an SQL outer join.

When defining a criteria query, it is not needed to restrict the entity types returned while creating the Hibernate Search query from the full text session: the type is guessed from the criteria query itself. Only fetch mode can be adjusted, refrain from applying any other restriction.

One cannot use `setCriteriaQuery` if more than one entity type is expected to be returned.

5.1.2.5. Projection

For some use cases, returning the domain object (graph) is overkill. Only a small subset of the properties is necessary. Hibernate Search allows you to return a subset of properties:

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "id", "summary", "body", "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
Integer id = firstResult[0];
String summary = firstResult[1];
String body = firstResult[2];
String authorName = firstResult[3];
```

Hibernate Search extracts the properties from the Lucene index and convert them back to their object representation, returning a list of `Object[]`. Projections avoid a potential database round trip (useful if the query response time is critical), but has some constraints:

- the properties projected must be stored in the index (`@Field(store=Store.YES)`), which increase the index size
- the properties projected must use a `FieldBridge` implementing `org.hibernate.search.bridge.TwoWayFieldBridge` or `org.hibernate.search.bridge.TwoWayStringBridge`, the latter being the simpler version. All Hibernate Search built-in types are two-way.

Projection is useful for another kind of usecases. Lucene provides some metadata informations to the user about the results. By using some special placeholders, the projection mechanism can retrieve them:

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( FullTextQuery.SCORE, FullTextQuery.BOOST, FullTextQuery.THIS, "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
float boost = firstResult[1];
Book book = firstResult[2];
String authorName = firstResult[3];
```

You can mix and match regular fields and special placeholders. Here is the list of available placeholders:

- `FullTextQuery.THIS`: returns the initialized and managed entity (as a non projected query would have done)
- `FullTextQuery.DOCUMENT`: returns the Lucene Document related to the object projected
- `FullTextQuery.SCORE`: returns the document score in the query. The score is guaranteed to be between 0

and 1 but the highest score is not necessarily equals to 1. Scores are handy to compare one result against an other for a given query but are useless when comparing the result of different queries.

- `FullTextQuery.BOOST`: the boost value of the Lucene Document
- `FullTextQuery.ID`: the id property value of the projected object
- `FullTextQuery.DOCUMENT_ID`: the Lucene document id. Careful, Lucene document id can change over-time between two different `IndexReader` opening (this feature is experimental)

5.2. Retrieving the results

Once the Hibernate Search query is built, executing it is in no way different than executing a HQL or Criteria query. The same paradigm and object semantic apply. All the common operations are available: `list()`, `uniqueResult()`, `iterate()`, `scroll()`.

5.2.1. Performance considerations

If you expect a reasonable number of results (for example using pagination) and expect to work on all of them, `list()` or `uniqueResult()` are recommended. `list()` work best if the entity `batch-size` is set up properly. Note that Hibernate Search has to process all Lucene Hits elements (within the pagination) when using `list()`, `uniqueResult()` and `iterate()`.

If you wish to minimize Lucene document loading, `scroll()` is more appropriate. Don't forget to close the `ScrollableResults` object when you're done, since it keeps Lucene resources. If you expect to use `scroll` but wish to load objects in batch, you can use `query.setFetchSize()`: When an object is accessed, and if not already loaded, Hibernate Search will load the next `fetchSize` objects in one pass.

Pagination is a preferred method over scrolling though.

5.2.2. Result size

It is sometime useful to know the total number of matching documents:

- for the Google-like feature 1-10 of about 888,000,000
- to implement a fast pagination navigation
- to implement a multi step search engine (adding approximation if the restricted query return no or not enough results)

But it would be costly to retrieve all the matching documents.

Hibernate Search allows you to retrieve the total number of matching documents regardless of the pagination parameters. Even more interesting, you can retrieve the number of matching elements without triggering a single object load.

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( luceneQuery, Book.class );
assert 3245 == query.getResultSize(); //return the number of matching books without loading a single o

org.hibernate.search.FullTextQuery query = s.createFullTextQuery( luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
```

```
assert 3245 == query.getResultSize(); //return the total number of matching books regardless of pagination
```

Note

Like Google, the number of results is approximative if the index is not fully up-to-date with the database (asynchronous cluster for example).

5.2.3. ResultTransformer

Especially when using projection, the data structure returned by a query (an object array in this case), is not always matching the application needs. It is possible to apply a `ResultTransformer` operation post query to match the targeted data structure:

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer(
    new StaticAliasToBeanResultTransformer( BookView.class, "title", "author" )
);
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}
```

Examples of `ResultTransformer` implementations can be found in the Hibernate Core codebase.

5.3. Filters

Apache Lucene has a powerful feature that allows to filters results from a query according to a custom filtering process. This is a very powerful way to apply some data restrictions after a query, especially since filters can be cached and reused. Some interesting usecases are:

- security
- temporal data (eg. view only last month's data)
- population filter (eg. search limited to a given category)
- and many more

Hibernate Search pushes the concept further by introducing the notion of parameterizable named filters which are transparently cached. For people familiar with the notion of Hibernate Core filters, the API is very similar.

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter( "login", "andre" );
fullTextQuery.list(); //returns only best drivers where andre has credentials
```

In this example we enabled 2 filters on top of this query. You can enable (or disable) as many filters as you want.

Declaring filters is done through the `@FullTextFilterDef` annotation. This annotation can be on any `@Indexed` entity regardless of the filter operation.

```
@Entity
@Indexed
```

```
@FullTextFilterDefs( {
    @FullTextFilterDef(name = "bestDriver", impl = BestDriversFilter.class, cache=false), //actual Fi
    @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class) //Filter factory with p
})
public class Driver { ... }
```

Each named filter points to an actual filter implementation.

```
public class BestDriversFilter extends org.apache.lucene.search.Filter {

    public BitSet bits(IndexReader reader) throws IOException {
        BitSet bitSet = new BitSet( reader.maxDoc() );
        TermDocs termDocs = reader.termDocs( new Term("score", "5") );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );
        }
        return bitSet;
    }
}
```

`BestDriversFilter` is an example of a simple Lucene filter that will filter all results to only return drivers whose score is 5. The filters must have a no-arg constructor when referenced in a `FulltextFilterDef.impl`.

The `cache` flag, defaulted to `true`, tells Hibernate Search to search the filter in its internal cache and reuses it if found.

Note that, usually, filter using the `IndexReader` are wrapped in a Lucene `CachingWrapperFilter` to benefit from some caching speed improvement. If your Filter creation requires additional steps or if the filter you are willing to use does not have a no-arg constructor, you can use the factory pattern:

```
@Entity
@Indexed
@FullTextFilterDef(name = "bestDriver", impl = BestDriversFilterFactory.class) //Filter factory
public class Driver { ... }

public class BestDriversFilterFactory {

    @Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

Hibernate Search will look for a `@Factory` annotated method and use it to build the filter instance. The factory must have a no-arg constructor. For people familiar with JBoss Seam, this is similar to the component factory pattern, but the annotation is different!

Named filters comes in handy where the filters have parameters. For example a security filter needs to know which credentials you are willing to filter by:

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5 );
```

Each parameter name should have an associated setter on either the filter or filter factory of the targeted named filter definition.

```
public class SecurityFilterFactory {
    private Integer level;

    /**
```



```

    * injected parameter
    */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key
    public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter( level );
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery( new Term("level", level.toString() ) );
        return new CachingWrapperFilter( new QueryWrapperFilter(query) );
    }
}

```

Note the method annotated `@Key` and returning a `FilterKey` object. The returned object has a special contract: the key object must implement `equals` / `hashCode` so that 2 keys are equals if and only if the given Filter types are the same and the set of parameters are the same. In other words, 2 filter keys are equal if and only if the filters from which the keys are generated can be interchanged. The key object is used as a key in the cache mechanism.

`@Key` methods are needed only if:

- you enabled the filter caching system (enabled by default)
- your filter has parameters

In most cases, using the `StandardFilterKey` implementation will be good enough. It delegates the `equals` / `hashCode` implementation to each of the parameters `equals` and `hashCode` methods.

Why should filters be cached? There are two area where filter caching shines:

- the system does not update the targeted entity index often (in other words, the `IndexReader` is reused a lot)
- the `Filter BitSet` is expensive to compute (compared to the time spent to execute the query)

Cache is enabled by default and use the notion of `SoftReferences` to dispose memory when needed. To adjust the size of the hard reference cache, use `hibernate.search.filter.cache_strategy.size` (defaults to 128). Don't forget to use a `CachingWrapperFilter` when the filter is cacheable and the `Filter`'s `bits` methods makes use of `IndexReader`.

For advance use of filter caching, you can implement your own `FilterCachingStrategy`. The classname is defined by `hibernate.search.filter.cache_strategy`.

5.4. Optimizing the query process

Query performance depends on several criteria:

- the Lucene query itself: read the literature on this subject
- the number of object loaded: use pagination (always ;-) or index projection (if needed)

- the way Hibernate Search interacts with the Lucene readers: defines the appropriate Reader strategy.

5.5. Native Lucene Queries

If you wish to use some specific features of Lucene, you can always run Lucene specific queries. Check Chapter 8, *Accessing Lucene natively* for more informations.

Chapter 6. Manual indexing

6.1. Indexing

It is sometimes useful to index an object even if this object is not inserted nor updated to the database. This is especially true when you want to build your index for the first time. You can achieve that goal using the `FullTextSession`.

```
FullTextSession fullTextSession = Search.createFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.index(customer);
}
tx.commit(); //index are written at commit time
```

For maximum efficiency, Hibernate Search batches index operations and executes them at commit time (Note: you don't need to use `org.hibernate.Transaction` in a JTA environment).

If you expect to index a lot of data, you need to be careful about memory consumption: since all documents are kept in a queue until the transaction commit, you can potentially face an `OutOfMemoryException`.

To avoid that, you can set up the `hibernate.search.worker.batch_size` property to a sensitive value: all index operations are queued until `batch_size` is reached. Every time `batch_size` is reached (or if the transaction is committed), the queue is processed (freeing memory) and emptied. Be aware that the changes cannot be rolled back if the number of index elements goes beyond `batch_size`. Be also aware that the queue limits are also applied on regular transparent indexing (and not only when `session.index()` is used). That's why a sensitive `batch_size` value is expected.

Other parameters which also can affect indexing time and memory consumption are `hibernate.search.[default|<indexname>].batch.merge_factor`, `hibernate.search.[default|<indexname>].batch.max_merge_docs` and `hibernate.search.[default|<indexname>].batch.max_buffered_docs`. These parameters are Lucene specific and Hibernate Search is just passing these parameters through - see Section 3.7, "Tuning Lucene indexing performance" for more details.

Here is an especially efficient way to index a given class (useful for index (re)initialization):

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria( Email.class ).scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % batchSize == 0) s.clear(); //clear every batchSize since the queue is processed
}
transaction.commit();
```

It is critical that `batchSize` in the previous example matches the `batch_size` value described previously.

6.2. Purging

It is equally possible to remove an entity or all entities of a given type from a Lucene index without the need to physically remove them from the database. This operation is named purging and is done through the `FullTextSession`.

```
FullTextSession fullTextSession = Search.createFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.purge( Customer.class, customer.getId() );
}
tx.commit(); //index are written at commit time
```

Purging will remove the entity with the given id from the Lucene index but will not touch the database.

If you need to remove all entities of a given type, you can use the `purgeAll` method.

```
FullTextSession fullTextSession = Search.createFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index are written at commit time
```

It is recommended to optimize the index after such an operation.

Note

Methods `index`, `purge` and `purgeAll` are available on `FullTextEntityManager` as well

Chapter 7. Index Optimization

From time to time, the Lucene index needs to be optimized. The process is essentially a defragmentation: until the optimization occurs, deleted documents are just marked as such, no physical deletion is applied, the optimization can also adjust the number of files in the Lucene Directory.

The optimization speeds up searches but in no way speeds up indexation (update). During an optimization, searches can be performed (but will most likely be slowed down), and all index updates will be stopped. Prefer optimizing:

- on an idle system or when the searches are less frequent
- after a lot of index modifications (doing so before will not speed up the indexation process)

7.1. Automatic optimization

Hibernate Search can optimize automatically an index after:

- a certain amount of operations have been applied (insertion, deletion)
- or a certain amount of transactions have been applied

The configuration can be global or defined at the index level:

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100

hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

An optimization will be triggered to the `Animal` index as soon as either:

- the number of addition and deletion reaches 1000
- the number of transactions reaches 50 (`hibernate.search.Animal.optimizer.transaction_limit.max` having priority over `hibernate.search.default.optimizer.transaction_limit.max`)

If none of these parameters are defined, not optimization is processed automatically.

7.2. Manual optimization

You can programmatically optimize (defragment) a Lucene index from Hibernate Search through the `SearchFactory`

```
searchFactory.optimize(Order.class);

searchFactory.optimize();
```

The first example reindex the Lucene index holding `Orders`, the second, optimize all indexes.

The `SearchFactory` can be accessed from a `FullTextSession`:

```
FullTextSession fullTextSession = Search.createFullTextSession(regularSession);  
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

Note that `searchFactory.optimize()` has no effect on a JMS backend. You must apply the optimize operation on the Master node.

7.3. Adjusting optimization

Apache Lucene has a few parameters to influence how optimization is performed. Hibernate Search expose those parameters.

Further index optimisation parameters include `hibernate.search.[default|<indexname>].merge_factor`, `hibernate.search.[default|<indexname>].max_merge_docs` and `hibernate.search.[default|<indexname>].max_buffered_docs` - see Section 3.7, “Tuning Lucene indexing performance” for more details.

Chapter 8. Accessing Lucene natively

8.1. SearchFactory

The `SearchFactory` object keeps track of the underlying Lucene resources for Hibernate Search, it's also a convenient way to access Lucene natively. The `SearchFactory` can be accessed from a `FullTextSession`:

```
FullTextSession fullTextSession = Search.createFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

8.2. Accessing a Lucene Directory

You can always access the Lucene directories through plain Lucene, the `Directory` structure is in no way different with or without Hibernate Search. However there are some more convenient ways to access a given `Directory`. The `SearchFactory` keeps track of the `DirectoryProviders` per indexed class. One directory provider can be shared amongst several indexed classes if the classes share the same underlying index directory. While usually not the case, a given entity can have several `DirectoryProviders` if the index is sharded (see Section 3.2, “Index sharding”).

```
DirectoryProvider[] provider = searchFactory.getDirectoryProviders(Order.class);
org.apache.lucene.store.Directory directory = provider[0].getDirectory();
```

In this example, `directory` points to the lucene index storing `Orders` information. Note that the obtained Lucene directory must not be closed (this is Hibernate Search responsibility).

8.3. Using an IndexReader

Queries in Lucene are executed on an `IndexReader`. Hibernate Search caches such index readers to maximize performances. Your code can access such cached / shared resources. You will just have to follow some "good citizen" rules.

```
DirectoryProvider orderProvider = searchFactory.getDirectoryProviders(Order.class)[0];
DirectoryProvider clientProvider = searchFactory.getDirectoryProviders(Client.class)[0];

ReaderProvider readerProvider = searchFactory.getReaderProvider();
IndexReader reader = readerProvider.openReader(orderProvider, clientProvider);

try {
    //do read-only operations on the reader
}
finally {
    readerProvider.closeReader(reader);
}
```

The `ReaderProvider` (described in Reader strategy), will open an `IndexReader` on top of the index(es) referenced by the directory providers. This `IndexReader` being shared amongst several clients, you must adhere to the following rules:

- Never call `indexReader.close()`, but always call `readerProvider.closeReader(reader)`; (a finally block is the best area).

- This `indexReader` must not be used for modification operations (especially delete), if you want to use an read/write index reader, open one from the Lucene Directory object.

Aside from those rules, you can use the `IndexReader` freely, especially to do native queries. Using the shared `IndexReaders` will make most queries more efficient.