Indexing and Querying with {brandname} 10.0

Table of Contents

Chapter 1. Indexing and Querying

1.1. Overview

{brandname} supports indexing and searching of Java Pojo(s) or objects encoded via Protocol Buffers stored in the grid using powerful search APIs which complement its main Map-like API.

Querying is possible both in library and client/server mode (for Java, C#, Node.js and other clients), and {brandname} can index data using Apache Lucene, offering an efficient full-text capable search engine in order to cover a wide range of data retrieval use cases.

Indexing configuration relies on a schema definition, and for that {brandname} can use annotated Java classes when in library mode, and protobuf schemas for remote clients written in other languages. By standardizing on protobuf, {brandname} allows full interoperability between Java and non-Java clients.

Apart from indexed queries, {brandname} can run queries over non-indexed data (indexless queries) and over partially indexed data (hybrid queries).

In terms of Search APIs, {brandname} has its own query language called *Ickle*, which is string-based and adds support for full-text querying. The Query DSL can be used for both embedded and remote java clients when full-text is not required; for Java embedded clients {brandname} offers the Hibernate Search Query API which supports running Lucene queries in the grid, apart from advanced search capabilities like Faceted and Spatial search.

Finally, {brandname} has support for Continuous Queries, which works in a reverse manner to the other APIs: instead of creating, executing a query and obtain results, it allows a client to register queries that will be evaluated continuously as data in the cluster changes, generating notifications whenever the changed data matches the queries.

1.2. Embedded Querying

Embedded querying is available when {brandname} is used as a library. No protobuf mapping is required, and both indexing and searching are done on top of Java objects. When in library mode, it is possible to run Lucene queries directly and use all the available Query APIs and it also allows flexible indexing configurations to keep latency to a minimal.

1.2.1. Quick example

We're going to store *Book* instances in an {brandname} cache called "books". *Book* instances will be indexed, so we enable indexing for the cache, letting {brandname} configure the indexing automatically:

{brandname} configuration:

infinispan.xml

```
<infinispan>
    <cache-container>
        <transport cluster="infinispan-cluster"/>
        <distributed-cache name="books">
            <indexing index="LOCAL" auto-config="true"/>
        </distributed-cache>
    </cache-container>
</infinispan>
```

Obtaining the cache:

```
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan.xml");
Cache<String, Book> cache = manager.getCache("books");
```

Each *Book* will be defined as in the following example; we have to choose which properties are indexed, and for each property we can optionally choose advanced indexing options using the annotations defined in the Hibernate Search project.

Book.java

```
import org.hibernate.search.annotations.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

//Values you want to index need to be annotated with @Indexed, then you pick which
fields and how they are to be indexed:
@Indexed
public class Book {
   @Field String title;
   @Field String description;
   @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
   @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```
public class Author {
   @Field String name;
   @Field String surname;
   // hashCode() and equals() omitted
}
```

Now assuming we stored several *Book* instances in our {brandname} *Cache* , we can search them for any matching field as in the following example.

Using a Lucene Query:

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager(cache);

// create any standard Lucene query, via Lucene's QueryParser or any other means:
org.apache.lucene.search.Query fullTextQuery = //any Apache Lucene Query

// convert the Lucene query to a CacheQuery:
CacheQuery cacheQuery = searchManager.getQuery( fullTextQuery );

// get the results:
List<Object> found = cacheQuery.list();
```

A Lucene Query is often created by parsing a query in text format such as "title:infinispan AND authors.name:sanne", or by using the query builder provided by Hibernate Search.

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager( cache );

// you could make the queries via Lucene APIs, or use some helpers:
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// the queryBuilder has a nice fluent API which guides you through all options.
// this has some knowledge about your object, for example which Analyzers
// need to be applied, but the output is a fairly standard Lucene Query.
org.apache.lucene.search.Query luceneQuery = queryBuilder.phrase()
                  .onField("description")
                  .andField("title")
                  .sentence("a book on highly scalable query engines")
                  .createQuery();

// the query API itself accepts any Lucene Query, and on top of that
// you can restrict the result to selected class types:
CacheQuery query = searchManager.getQuery(luceneQuery, Book.class);

// and there are your results!
List objectList = query.list();

for (Object book : objectList) {
      System.out.println(book);
}
```

Apart from *list()* you have the option for streaming results, or use pagination.

For searches that do not require Lucene or full-text capabilities and are mostly about aggregation

and exact matches, we can use the {brandname} Query DSL API:

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.Search;

// get the query factory:
QueryFactory queryFactory = Search.getQueryFactory(cache);

Query q = queryFactory.from(Book.class)
            .having("author.surname").eq("King")
            .build();

List<Book> list = q.list();
```

Finally, we can use an Ickle query directly, allowing for Lucene syntax in one or more predicates:

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

// get the query factory:
QueryFactory queryFactory = Search.getQueryFactory(cache);


Query q = queryFactory.create("from Book b where b.author.name = 'Stephen' and " +
                "b.description : (+'dark' -'tower')");

List<Book> list = q.list();
```

1.2.2. Indexing

Indexing in {brandname} happens on a per-cache basis and by default a cache is not indexed. Enabling indexing is not mandatory but queries using an index will have a vastly superior performance. On the other hand, enabling indexing can impact negatively the write throughput of a cluster, so make sure to check the query performance guide for some strategies to minimize this impact depending on the cache type and use case.

Configuration

General format

To enable indexing via XML, you need to add the `<indexing>` element plus the `index` (index mode) to your cache configuration, and optionally pass additional properties.

```
<infinispan>
   <cache-container default-cache="default">
      <replicated-cache name="default">
         <indexing index="ALL">
            <property name="property.name">some value</property>
         </indexing>
      </replicated-cache>
   </cache-container>
</infinispan>
```

Programmatic:

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().index(Index.ALL)
      .addProperty("property name", "propery value")
```

Index names

Each property inside the `index` element is prefixed with the index name, for the index named `org.infinispan.sample.Car` the `directory_provider` is `local-heap`:

```
  ...
  <indexing index="ALL">
      <property name="org.infinispan.sample.Car.directory_provider">local-
heap</property>
  </indexing>
...
</infinispan>
```

```
cacheCfg.indexing()
   .index(Index.ALL)
      .addProperty("org.infinispan.sample.Car.directory_provider", "local-heap")
```

{brandname} creates an index for each entity existent in a cache, and it allows to configure those indexes independently. For a class annotated with `@Indexed`, the index name is the fully qualified class name, unless overridden with the `name` argument in the annotation.

In the snippet below, the default storage for all entities is `infinispan`, but `Boat` instances will be stored on `local-heap` in an index named `boatIndex`. `Airplane` entities will also be stored in `local-heap`. Any other entity's index will be configured with the property prefixed by `default`.

```
package org.infinispan.sample;

@Indexed(name = "boatIndex")
public class Boat {

}

@Indexed
public class Airplane {

}
```

```
    ...
    <indexing index="ALL">
        <property name="default.directory_provider">infinispan</property>
        <property name="boatIndex.directory_provider">local-heap</property>
        <property name="org.infinispan.sample.Airplane.directory_provider">
            ram
        </property>
    </indexing>
    ...
</infinispan>
```

Specifying indexed Entities

{brandname} can automatically recognize and manage indexes for different entity types in a cache. Future versions of {brandname} will remove this capability so it's recommended to declare upfront which types are going to be indexed (list them by their fully qualified class name). This can be done via xml:

```
<infinispan>
   <cache-container default-cache="default">
      <replicated-cache name="default">
         <indexing index="ALL">
            <indexed-entities>
                <indexed-entity>com.acme.query.test.Car</indexed-entity>
                <indexed-entity>com.acme.query.test.Truck</indexed-entity>
            </indexed-entities>
         </indexing>
      </replicated-cache>
   </cache-container>
</infinispan>
```

or programmatically:

```
cacheCfg.indexing()
      .index(Index.ALL)
       .addIndexedEntity(Car.class)
       .addIndexedEntity(Truck.class)
```

In server mode, the class names listed under the 'indexed-entities' element must use the 'extended' class name format which is composed of a JBoss Modules module identifier, a slot name, and the fully qualified class name, these three components being separated by the ':' character, (eg. "com.acme.my-module-with-entity-classes:my-slot:com.acme.query.test.Car"). The entity classes must be located in the referenced module, which can be either a user supplied module deployed in the 'modules' folder of your server or a plain jar deployed in the 'deployments' folder. The module in question will become an automatic dependency of your Cache, so its eventual redeployment will cause the cache to be restarted.

Only for server, if you fail to follow the requirement of using 'extended' class names and use a plain class name its resolution will fail due to missing class because the wrong ClassLoader is being used (the {brandname}'s internal class path is being used).

Index mode

An {brandname} node typically receives data from two sources: local and remote. Local translates to clients manipulating data using the map API in the same JVM; remote data comes from other {brandname} nodes during replication or rebalancing.

The index mode configuration defines, from a node in the cluster point of view, which data gets indexed.

Possible values:

- ALL: all data is indexed, local and remote.
- LOCAL: only local data is indexed.
- PRIMARY_OWNER: Only entries containing keys that the node is primary owner will be indexed, regardless of local or remote origin.
- NONE: no data is indexed. Equivalent to not configure indexing at all.

Index Managers

Index managers are central components in {brandname} Querying responsible for the indexing configuration, distribution and internal lifecycle of several query components such as Lucene's *IndexReader* and *IndexWriter*. Each Index Manager is associated with a *Directory Provider*, which defines the physical storage of the index.

Regarding index distribution, {brandname} can be configured with shared or non-shared indexes.

Shared indexes

A shared index is a single, distributed, cluster-wide index for a certain cache. The main advantage is that the index is visible from every node and can be queried as if the index were local, there is no need to broadcast queries to all members and aggregate the results. The downside is that Lucene does not allow more than a single process writing to the index at the same time, and the coordination of lock acquisitions needs to be done by a proper shared index capable index manager. In any case, having a single write lock cluster-wise can lead to some degree of contention under heavy writing.

{brandname} supports shared indexes leveraging the {brandname} Directory Provider, which stores indexes in a separate set of caches. Two index managers are available to use shared indexes: InfinispanIndexManager and AffinityIndexManager.

Effect of the index mode

Shared indexes should not use the `ALL` index mode since it'd lead to redundant indexing: since there is a single index cluster wide, the entry would get indexed when inserted via Cache API, and another time when {brandname} replicates it to another node. The `ALL` mode is usually associates with non-shared indexes in order to create full index replicas on each node.

InfinispanIndexManager

This index manager uses the {brandname} Directory Provider, and is suitable for creating shared indexes. Index mode should be set to `LOCAL` in this configuration.

Configuration:

```
<distributed-cache name="default" >
    <indexing index="LOCAL">
        <property name="default.indexmanager">
            org.infinispan.query.indexmanager.InfinispanIndexManager
        </property>
        <!-- optional: tailor each index cache -->
        <property name="default.locking_cachename">
LuceneIndexesLocking_custom</property>
        <property name="default.data_cachename">LuceneIndexesData_custom</property>
        <property name="default.metadata_cachename">
LuceneIndexesMetadata_custom</property>
    </indexing>
</distributed-cache>

<!-- Optional -->
<replicated-cache name="LuceneIndexesLocking_custom">
    <indexing index="NONE" />
    <-- extra configuration -->
</replicated-cache>

<!-- Optional -->
<replicated-cache name="LuceneIndexesMetadata_custom">
    <indexing index="NONE" />
    <-- extra configuration -->
</replicated-cache>

<!-- Optional -->
<distributed-cache name="LuceneIndexesData_custom">
    <-- extra configuration -->
    <indexing index="NONE" />
</distributed-cache>
```

Indexes are stored in a set of clustered caches, called by default *LuceneIndexesData*, *LuceneIndexesMetadata* and *LuceneIndexesLocking*.

The *LuceneIndexesLocking* cache is used to store Lucene locks, and it is a very small cache: it will contain one entry per entity (index).

The *LuceneIndexesMetadata* cache is used to store info about the logical files that are part of the index, such as names, chunks and sizes and it is also small in size.

The *LuceneIndexesData* cache is where most of the index is located: it is much bigger then the other two but should be smaller than the data in the cache itself, thanks to Lucene's efficient storing techniques.

It's not necessary to redefine the configuration of those 3 cases, {brandname} will pick sensible defaults. Reasons re-define them would be performance tuning for a specific scenario, or for example to make them persistent by configuring a cache store.

In order to avoid index corruption when two or more nodes of the cluster try to write to the index

at the same time, the *InfinispanIndexManager* internally elects a master in the cluster (which is the JGroups coordinator) and forwards all indexing works to this master.

AffinityIndexManager

The AffinityIndexManager is an **experimental** index manager used for shared indexes that also stores indexes using the {brandname} Directory Provider. Unlike the InfinispanIndexManager, it does not have a single node (master) that handles all the indexing cluster wide, but rather splits the index using multiple shards, each shard being responsible for indexing data associated with one or more {brandname} segments. For an in-depth description of the inner workings, please see the design doc.

The PRIMARY_OWNER index mode is required, together with a special kind of `KeyPartitioner`.

XML Configuration:

```
<distributed-cache name="default"
                   key-partitioner=
"org.infinispan.distribution.ch.impl.AffinityPartitioner">
    <indexing index="PRIMARY_OWNER">
        <property name="default.indexmanager">
            org.infinispan.query.affinity.AffinityIndexManager
        </property>
        <!-- optional: control the number of shards, the default is 4 -->
        <property name="default.sharding_strategy.nbr_of_shards">10</property>
    </indexing>
</distributed-cache>
```

Programmatic:

```
import org.infinispan.distribution.ch.impl.AffinityPartitioner;
import org.infinispan.query.affinity.AffinityIndexManager;

ConfigurationBuilder cacheCfg = ...
cacheCfg.clustering().hash().keyPartitioner(new AffinityPartitioner());
cacheCfg.indexing()
      .index(Index.PRIMARY_OWNER)
      .addProperty("default.indexmanager", AffinityIndexManager.class.getName())
      .addProperty("default.sharding_strategy.nbr_of_shards", "10")
```

The `AffinityIndexManager` by default will have as many shards as {brandname} segments, but this value is configurable as seen in the example above.

The number of shards affects directly the query performance and writing throughput: generally speaking, a high number of shards offers better write throughput but has an adverse effect on query performance.

Non-shared indexes

Non-shared indexes are independent indexes at each node. This setup is particularly advantageous for replicated caches where each node has all the cluster data and thus can hold all the indexes as well, offering optimal query performance with zero network latency when querying. Another advantage is, since the index is local to each node, there is less contention during writes due to the fact that each node is subjected to its own index lock, not a cluster wide one.

Since each node might hold a partial index, it may be necessary to link#query_clustered_query_api[broadcast] queries in order to get correct search results, which can add latency. If the cache is REPL, though, the broadcast is not necessary: each node can hold a full local copy of the index and queries runs at optimal speed taking advantage of a local index.

{brandname} has two index managers suitable for non-shared indexes: `directory-based` and `near-real-time`. Storage wise, non-shared indexes can be located in ram, filesystem, or {brandname} local caches.

Effect of the index mode

The `directory-based` and `near-real-time` index managers can be associated with different index modes, resulting in different index distributions.

REPL caches combined with the `ALL` index mode will result in a full copy of the cluster-wide index on each node. This mode allows queries to become effectively local without network latency. This is the recommended mode to index any REPL cache, and that's the choice picked by the auto-config when the a REPL cache is detected. The `ALL` mode should not be used with DIST caches.

REPL or DIST caches combined with `LOCAL` index mode will cause each node to index only data inserted from the same JVM, causing an uneven distribution of the index. In order to obtain correct query results, it's necessary to use broadcast queries.

REPL or DIST caches combined with `PRIMARY_OWNER` will also need broadcast queries. Differently from the `LOCAL` mode, each node's index will contain indexed entries which key is primarily owned by the node according to the consistent hash, leading to a more evenly distributed indexes among the nodes.

directory-based index manager

This is the default Index Manager used when no index manager is configured. The `directory-based` index manager is used to manage indexes backed by a local lucene directory. It supports *ram*, *filesystem* and non-clustered *infinispan* storage.

Filesystem storage

This is the default storage, and used when index manager configuration is omitted. The index is stored in the filesystem using a MMapDirectory. It is the recommended storage for local indexes. Although indexes are persistent on disk, they get memory mapped by Lucene and thus offer decent query performance.

Configuration:

```
<replicated-cache name="myCache">
   <indexing index="ALL">
      <!-- Optional: define base folder for indexes -->
      <property name="default.indexBase">${java.io.tmpdir}/baseDir</property>
   </indexing>
</replicated-cache>
```

{brandname} will create a different folder under `default.indexBase` for each entity (index) present in the cache.

Ram storage

Index is stored in memory using a Lucene RAMDirectory. Not recommended for large indexes or highly concurrent situations. Indexes stored in Ram are not persistent, so after a cluster shutdown a re-index is needed. Configuration:

```
<replicated-cache name="myCache">
   <indexing index="ALL">
      <property name="default.directory_provider">local-heap</property>
   </indexing>
</replicated-cache>
```

{brandname} storage

{brandname} storage makes use of the {brandname} Lucene directory that saves the indexes to a set of caches; those caches can be configured like any other {brandname} cache, for example by adding a cache store to have indexes persisted elsewhere apart from memory. In order to use {brandname} storage with a non-shared index, it's necessary to use LOCAL caches for the indexes:

```
<replicated-cache name="default">
    <indexing index="ALL">
        <property name="default.locking_cachename">
LuceneIndexesLocking_custom</property>
        <property name="default.data_cachename">LuceneIndexesData_custom</property>
        <property name="default.metadata_cachename">
LuceneIndexesMetadata_custom</property>
    </indexing>
</replicated-cache>

<local-cache name="LuceneIndexesLocking_custom">
    <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesMetadata_custom">
    <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesData_custom">
    <indexing index="NONE" />
</local-cache>
```

near-real-time index manager

Similar to the `directory-based` index manager but takes advantage of the Near-Real-Time features of Lucene. It has better write performance than the `directory-based` because it flushes the index to the underlying store less often. The drawback is that unflushed index changes can be lost in case of a non-clean shutdown. Can be used in conjunction with `local-heap`, `filesystem` and local infinispan storage. Configuration for each different storage type is the same as the directory-based index manager.

Example with ram:

```
<replicated-cache name="default">
    <indexing index="ALL">
        <property name="default.indexmanager">near-real-time</property>
        <property name="default.directory_provider">local-heap</property>
    </indexing>
</replicated-cache>
```

Example with filesystem:

```
<replicated-cache name="default">
    <indexing index="ALL">
        <property name="default.indexmanager">near-real-time</property>
    </indexing>
</replicated-cache>
```

External indexes

Apart from having shared and non-shared indexes managed by {brandname} itself, it is possible to offload indexing to a third party search engine: currently {brandname} supports Elasticsearch as a external index storage.

Elasticsearch IndexManager (experimental)

This index manager forwards all indexes to an external Elasticsearch server. This is an experimental integration and some features may not be available, for example `indexNullAs` for `@IndexedEmbedded` annotations is not currently supported.

Configuration:

```
<indexing index="LOCAL">
    <property name="default.indexmanager">elasticsearch</property>
    <property name="default.elasticsearch.host">
link:http://elasticHost:9200</property>
    <!-- other elasticsearch configurations -->
</indexing>
```

The index mode should be set to `LOCAL`, since {brandname} considers Elasticsearch as a single shared index. More information about Elasticsearch integration, including the full description of the configuration properties can be found at the Hibernate Search manual.

Automatic configuration

The attribute auto-config provides a simple way of configuring indexing based on the cache type. For replicated and local caches, the indexing is configured to be persisted on disk and not shared with any other processes. Also, it is configured so that minimum delay exists between the moment an object is indexed and the moment it is available for searches (near real time).

```
<local-cache name="default">
   <indexing index="LOCAL" auto-config="true">
   </indexing>
</local-cache>
```

it is possible to redefine any property added via auto-config, and also add new properties, allowing for advanced tuning.

The auto config adds the following properties for replicated and local caches:

Property name	value	description
default.directory_provider	filesystem	Filesystem based index. More details at Hibernate Search documentation

Property name	value	description
default.exclusive_index_use	true	indexing operation in exclusive mode, allowing Hibernate Search to optimize writes
default.indexmanager	near-real-time	make use of Lucene near real time feature, meaning indexed objects are promptly available to searches
default.reader.strategy	shared	Reuse index reader across several queries, thus avoiding reopening it

For distributed caches, the auto-config configure indexes in {brandname} itself, internally handled as a master-slave mechanism where indexing operations are sent to a single node which is responsible to write to the index.

The auto config properties for distributed caches are:

Property name	value	description
default.directory_provider	infinispan	Indexes stored in {brandname}. More details at Hibernate Search documentation
default.exclusive_index_use	true	indexing operation in exclusive mode, allowing Hibernate Search to optimize writes
default.indexmanager	org.infinispan.query.indexmanager.InfinispanIndexManager	Delegates index writing to a single node in the {brandname} cluster
default.reader.strategy	shared	Reuse index reader across several queries, avoiding reopening it

Re-indexing

Occasionally you might need to rebuild the Lucene index by reconstructing it from the data stored in the Cache. You need to rebuild the index if you change the definition of what is indexed on your types, or if you change for example some *Analyzer* parameter, as Analyzers affect how the index is written. Also, you might need to rebuild the index if you had it destroyed by some system administration mistake. To rebuild the index just get a reference to the MassIndexer and start it; beware it might take some time as it needs to reprocess all data in the grid!

```
// Blocking execution
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();

// Non blocking execution
CompletableFuture<Void> future = searchManager.getMassIndexer().startAsyc();
```

This is also available as a `start` JMX operation on the MassIndexer MBean registered under the name `org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=MassIndexer`.

Indexless

TODO

Hybrid

TODO

Mapping Entities

{brandname} relies on the rich API of Hibernate Search in order to define fine grained configuration for indexing at entity level. This configuration includes which fields are annotated, which analyzers should be used, how to map nested objects and so on. Detailed documentation is available at the Hibernate Search manual.

@DocumentId

Unlike Hibernate Search, using *@DocumentId* to mark a field as identifier does not apply to {brandname} values; in {brandname} the identifier for all *@Indexed* objects is the key used to store the value. You can still customize how the key is indexed using a combination of *@Transformable* , custom types and custom *FieldBridge* implementations.

@Transformable keys

The key for each value needs to be indexed as well, and the key instance must be transformed in a *String*. {brandname} includes some default transformation routines to encode common primitives, but to use a custom key you must provide an implementation of *org.infinispan.query.Transformer* .

Registering a key Transformer via annotations

You can annotate your key class with *org.infinispan.query.Transformable* and your custom transformer implementation will be picked up automatically:

```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
   ...
}

public class CustomTransformer implements Transformer {
   @Override
   public Object fromString(String s) {
      ...
      return new CustomKey(...);
   }

   @Override
   public String toString(Object customType) {
      CustomKey ck = (CustomKey) customType;
      return ...
   }
}
```

Registering a key Transformer via the cache indexing configuration

You can use the *key-transformers* xml element in both embedded and server config:

```
<replicated-cache name="test">
    <indexing index="ALL" auto-config="true">
        <key-transformers>
            <key-transformer key="com.mycompany.CustomKey" transformer=
"com.mycompany.CustomTransformer"/>
        </key-transformers>
    </indexing>
</replicated-cache>
```

or alternatively, you can achieve the same effect by using the Java configuration API (embedded mode):

```
   ConfigurationBuilder builder = ...
   builder.indexing().autoConfig(true)
         .addKeyTransformer(CustomKey.class, CustomTransformer.class);
```

Registering a Transformer programmatically at runtime

Using this technique, you don't have to annotate your custom key type and you also do not add the transformer to the, cache indexing configuration, instead, you can add it to the *SearchManagerImplementor* dynamically at runtime by invoking *org.infinispan.query.spi.SearchManagerImplementor.registerKeyTransformer(Class<?>, Class<? extends Transformer>)*:

```
org.infinispan.query.spi.SearchManagerImplementor manager = Search.getSearchManager
(cache).unwrap(SearchManagerImplementor.class);
manager.registerKeyTransformer(keyClass, keyTransformerClass);
```

This approach is deprecated since 10.0 because it can lead to situations when a newly started node receives cache entries via initial state transfer and is not able to index them because the needed key transformers are not yet registered (and can only be registered after the Cache has been fully started). This undesirable situation is avoided if you register your key transformers using the other available approaches (configuration and annotation).

Programmatic mapping

Instead of using annotations to map an entity to the index, it's also possible to configure it programmatically.

In the following example we map an object *Author* which is to be stored in the grid and made searchable on two properties but without annotating the class.

```
import org.apache.lucene.search.Query;
import org.hibernate.search.cfg.Environment;
import org.hibernate.search.cfg.SearchMapping;
import org.hibernate.search.query.dsl.QueryBuilder;
import org.infinispan.Cache;
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.cache.Index;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.query.CacheQuery;
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;

import java.io.IOException;
import java.lang.annotation.ElementType;
import java.util.Properties;

SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed()
       .property("name", ElementType.METHOD).field()
       .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
        .indexing().index(Index.LOCAL)
        .withProperties(properties)
        .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q = qb.keyword().onField("name").matching("Manik").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
assert cq.getResultSize() == 1;
```

1.2.3. Querying APIs

You can query {brandname} using:

- Lucene or Hibernate Search Queries. {brandname} exposes the Hibernate Search DSL, which

produces Lucene queries. You can run Lucene queries on single nodes or broadcast queries to multiple nodes in an {brandname} cluster.

- Ickle queries, a custom string-based query language with full-text extensions.

Hibernate Search

Apart from supporting Hibernate Search annotations to configure indexing, it's also possible to query the cache using other Hibernate Search APIs

Running Lucene queries

To run a Lucene query directly, simply create and wrap it in a *CacheQuery*:

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.Query;


SearchManager searchManager = Search.getSearchManager(cache);
Query query = searchManager.buildQueryBuilderForClass(Book.class).get()
            .keyword().wildcard().onField("description").matching("*test*")
.createQuery();
CacheQuery<Book> cacheQuery = searchManager.getQuery(query);
```

Using the Hibernate Search DSL

The Hibernate Search DSL can be used to create the Lucene Query, example:

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.search.Query;

Cache<String, Book> cache = ...

SearchManager searchManager = Search.getSearchManager(cache);

Query luceneQuery = searchManager
                        .buildQueryBuilderForClass(Book.class).get()
                        .range().onField("year").from(2005).to(2010)
                        .createQuery();

List<Object> results = searchManager.getQuery(luceneQuery).list();
```

For a detailed description of the query capabilities of this DSL, see the relevant section of the Hibernate Search manual.

Faceted Search

{brandname} support Faceted Searches by using the Hibernate Search `FacetManager`:

```
// Cache is indexed
Cache<Integer, Book> cache = ...

// Obtain the Search Manager
SearchManager searchManager = Search.getSearchManager(cache);

// Create the query builder
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// Build any Lucene Query. Here it's using the DSL to do a Lucene term query on a book
name
Query luceneQuery = queryBuilder.keyword().wildcard().onField("name").matching(
"bitcoin").createQuery();

// Wrap into a cache Query
CacheQuery<Book> query = searchManager.getQuery(luceneQuery);

// Define the Facet characteristics
FacetingRequest request = queryBuilder.facet()
                .name("year_facet")
                .onField("year")
                .discrete()
                .orderedBy(FacetSortOrder.COUNT_ASC)
                .createFacetingRequest();

// Associated the FacetRequest with the query
FacetManager facetManager = query.getFacetManager().enableFaceting(request);

// Obtain the facets
List<Facet> facetList = facetManager.getFacets("year_facet");
```

A Faceted search like above will return the number books that match 'bitcoin' released on a yearly basis, for example:

```
AbstractFacet{facetingName='year_facet', fieldName='year', value='2008', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2009', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2010', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2011', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2012', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2016', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2015', count=2}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2013', count=3}
```

For more info about Faceted Search, see Hibernate Search Faceting

Spatial Queries

{brandname} also supports Spatial Queries, allowing to combining full-text with restrictions based on distances, geometries or geographic coordinates.

Example, we start by using the `@Spatial` annotation in our entity that will be searched, together with `@Latitude` and `@Longitude`:

```
@Indexed
@Spatial
public class Restaurant {

      @Latitude
      private Double latitude;

      @Longitude
      private Double longitude;

      @Field(store = Store.YES)
      String name;

      // Getters, Setters and other members omitted

   }
```

to run spatial queries, the Hibernate Search DSL can be used:

```
// Cache is configured as indexed
Cache<String, Restaurant> cache = ...

// Obtain the SearchManager
Searchmanager searchManager = Search.getSearchManager(cache);

// Build the Lucene Spatial Query
Query query = Search.getSearchManager(cache).buildQueryBuilderForClass(Restaurant
.class).get()
          .spatial()
            .within( 2, Unit.KM )
              .ofLatitude( centerLatitude )
              .andLongitude( centerLongitude )
            .createQuery();

// Wrap in a cache Query
CacheQuery<Restaurant> cacheQuery = searchManager.getQuery(query);

List<Restaurant> nearBy = cacheQuery.list();
```

More info on Hibernate Search manual

IndexedQueryMode

It's possible to specify a query mode for indexed queries. IndexedQueryMode.BROADCAST allows to broadcast a query to each node of the cluster, retrieve the results and combine them before returning to the caller. It is suitable for use in conjunction with non-shared indexes, since each node's local index will have only a subset of the data indexed.

IndexedQueryMode.FETCH will execute the query in the caller. If all the indexes for the cluster wide data are available locally, performance will be optimal, otherwise this query mode may involve fetching indexes data from remote nodes.

The IndexedQueryMode is supported for Lucene Queries and Ickle String queries at the moment (no {brandname} Query DSL).

Example:

```
CacheQuery<Person> broadcastQuery = Search.getSearchManager(cache).getQuery(new
MatchAllDocsQuery(), IndexedQueryMode.BROADCAST);

List<Person> result = broadcastQuery.list();
```

{brandname} Query DSL

{brandname} provides its own query DSL, independent of Lucene and Hibernate Search. Decoupling the query API from the underlying query and indexing mechanism makes it possible to introduce new alternative engines in the future, besides Lucene, and still being able to use the same uniform query API. The current implementation of indexing and searching is still based on Hibernate Search and Lucene so all indexing related aspects presented in this chapter still apply.

The new API simplifies the writing of queries by not exposing the user to the low level details of constructing Lucene query objects and also has the advantage of being available to remote Hot Rod clients. But before delving into further details, let's examine first a simple example of writing a query for the *Book* entity from the previous example.

Query example using {brandname}'s query DSL

```
import org.infinispan.query.dsl.*;

// get the DSL query factory from the cache, to be used for constructing the Query
object:
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);

// create a query for all the books that have a title which contains "engine":
org.infinispan.query.dsl.Query query = qf.from(Book.class)
      .having("title").like("%engine%")
      .build();

// get the results:
List<Book> list = query.list();
```

The API is located in the *org.infinispan.query.dsl* package. A query is created with the help of the *QueryFactory* instance which is obtained from the per-cache *SearchManager*. Each *QueryFactory* instance is bound to the same *Cache* instance as the *SearchManager*, but it is otherwise a stateless and thread-safe object that can be used for creating multiple queries in parallel.

Query creation starts with the invocation of the `from(Class entityType)` method which returns a *QueryBuilder* object that is further responsible for creating queries targeted to the specified entity class from the given cache.

A query will always target a single entity type and is evaluated over the contents of a single cache. Running a query over multiple caches or creating queries that target several entity types (joins) is not supported.

The *QueryBuilder* accumulates search criteria and configuration specified through the invocation of its DSL methods and is ultimately used to build a *Query* object by the invocation of the `QueryBuilder.build()` method that completes the construction. Being a stateful object, it cannot be used for constructing multiple queries at the same time (except for nested queries) but can be reused afterwards.

This *QueryBuilder* is different from the one from Hibernate Search but has a somewhat similar purpose, hence the same name. We are considering renaming it in near future to prevent ambiguity.

Executing the query and fetching the results is as simple as invoking the `list()` method of the *Query* object. Once executed the *Query* object is not reusable. If you need to re-execute it in order to obtain fresh results then a new instance must be obtained by calling `QueryBuilder.build()`.

Filtering operators

Constructing a query is a hierarchical process of composing multiple criteria and is best explained following this hierarchy.

The simplest possible form of a query criteria is a restriction on the values of an entity attribute according to a filtering operator that accepts zero or more arguments. The entity attribute is specified by invoking the `having(String attributePath)` method of the query builder which returns an intermediate context object (*FilterConditionEndContext*) that exposes all the available operators. Each of the methods defined by *FilterConditionEndContext* is an operator that accepts an argument, except for `between` which has two arguments and `isNull` which has no arguments. The arguments are statically evaluated at the time the query is constructed, so if you're looking for a feature similar to SQL's correlated sub-queries, that is not currently available.

```
// a single query criterion
QueryBuilder qb = ...
qb.having("title").eq("Hibernate Search in Action");
```

Table 1. FilterConditionEndContext exposes the following filtering operators:

Filter	Arguments	Description
in	Collection values	Checks that the left operand is equal to one of the elements from the Collection of values given as argument.
in	Object… values	Checks that the left operand is equal to one of the (fixed) list of values given as argument.
contains	Object value	Checks that the left argument (which is expected to be an array or a Collection) contains the given element.
containsAll	Collection values	Checks that the left argument (which is expected to be an array or a Collection) contains all the elements of the given collection, in any order.
containsAll	Object… values	Checks that the left argument (which is expected to be an array or a Collection) contains all of the the given elements, in any order.
containsAny	Collection values	Checks that the left argument (which is expected to be an array or a Collection) contains any of the elements of the given collection.
containsAny	Object… values	Checks that the left argument (which is expected to be an array or a Collection) contains any of the the given elements.
isNull		Checks that the left argument is null.
like	String pattern	Checks that the left argument (which is expected to be a String) matches a wildcard pattern that follows the JPA rules.
eq	Object value	Checks that the left argument is equal to the given value.
equal	Object value	Alias for eq.
gt	Object value	Checks that the left argument is greater than the given value.
gte	Object value	Checks that the left argument is greater than or equal to the given value.

Filter	Arguments	Description
lt	Object value	Checks that the left argument is less than the given value.
lte	Object value	Checks that the left argument is less than or equal to the given value.
between	Object from, Object to	Checks that the left argument is between the given range limits.

It's important to note that query construction requires a multi-step chaining of method invocation that must be done in the proper sequence, must be properly completed exactly *once* and must not be done twice, or it will result in an error. The following examples are invalid, and depending on each case they lead to criteria being ignored (in benign cases) or an exception being thrown (in more serious ones).

```
// Incomplete construction. This query does not have any filter on "title" attribute
yet,
// although the author may have intended to add one.
QueryBuilder qb1 = ...
qb1.having("title");
Query q1 = qb1.build(); // consequently, this query matches all Book instances
regardless of title!

// Duplicated completion. This results in an exception at run-time.
// Maybe the author intended to connect two conditions with a boolean operator,
// but this does NOT actually happen here.
QueryBuilder qb2 = ...
qb2.having("title").like("%Data Grid%");
qb2.having("description").like("%clustering%");   // will throw
java.lang.IllegalStateException: Sentence already started. Cannot use 'having(..)'
again.
Query q2 = qb2.build();
```

Filtering based on attributes of embedded entities

The `having` method also accepts dot separated attribute paths for referring to *embedded entity* attributes, so the following is a valid query:

```
// match all books that have an author named "Manik"
Query query = queryFactory.from(Book.class)
      .having("author.name").eq("Manik")
      .build();
```

Each part of the attribute path must refer to an existing indexed attribute in the corresponding entity or embedded entity class respectively. It's possible to have multiple levels of embedding.

Boolean conditions

Combining multiple attribute conditions with logical conjunction (`and`) and disjunction (`or`) operators in order to create more complex conditions is demonstrated in the following example. The well known operator precedence rule for boolean operators applies here, so the order of DSL method invocations during construction is irrelevant. Here `and` operator still has higher priority than `or` even though `or` was invoked first.

```
// match all books that have "Data Grid" in their title
// or have an author named "Manik" and their description contains "clustering"
Query query = queryFactory.from(Book.class)
  .having("title").like("%Data Grid%")
  .or().having("author.name").eq("Manik")
  .and().having("description").like("%clustering%")
  .build();
```

Boolean negation is achieved with the `not` operator, which has highest precedence among logical operators and applies only to the next simple attribute condition.

```
// match all books that do not have "Data Grid" in their title and are authored by
"Manik"
Query query = queryFactory.from(Book.class)
  .not().having("title").like("%Data Grid%")
  .and().having("author.name").eq("Manik")
  .build();
```

Nested conditions

Changing the precedence of logical operators is achieved with nested filter conditions. Logical operators can be used to connect two simple attribute conditions as presented before, but can also connect a simple attribute condition with the subsequent complex condition created with the same query factory.

```
// match all books that have an author named "Manik" and their title contains
// "Data Grid" or their description contains "clustering"
Query query = queryFactory.from(Book.class)
  .having("author.name").eq("Manik")
  .and(queryFactory.having("title").like("%Data Grid%")
          .or().having("description").like("%clustering%"))
  .build();
```

Projections

In some use cases returning the whole domain object is overkill if only a small subset of the attributes are actually used by the application, especially if the domain entity has embedded entities. The query language allows you to specify a subset of attributes (or attribute paths) to return - the projection. If projections are used then the `Query.list()` will not return the whole

domain entity but will return a *List* of *Object[]*, each slot in the array corresponding to a projected attribute.

TODO document what needs to be configured for an attribute to be available for projection.

```
// match all books that have "Data Grid" in their title or description
// and return only their title and publication year
Query query = queryFactory.from(Book.class)
  .select("title", "publicationYear")
  .having("title").like("%Data Grid%")
  .or().having("description").like("%Data Grid%"))
  .build();
```

Sorting

Ordering the results based on one or more attributes or attribute paths is done with the `QueryBuilder.orderBy( )` method which accepts an attribute path and a sorting direction. If multiple sorting criteria are specified, then the order of invocation of `orderBy` method will dictate their precedence. But you have to think of the multiple sorting criteria as acting together on the tuple of specified attributes rather than in a sequence of individual sorting operations on each attribute.

TODO document what needs to be configured for an attribute to be available for sorting.

```
// match all books that have "Data Grid" in their title or description
// and return them sorted by the publication year and title
Query query = queryFactory.from(Book.class)
  .orderBy("publicationYear", SortOrder.DESC)
  .orderBy("title", SortOrder.ASC)
  .having("title").like("%Data Grid%")
  .or().having("description").like("%Data Grid%"))
  .build();
```

Pagination

You can limit the number of returned results by setting the *maxResults* property of *QueryBuilder*. This can be used in conjunction with setting the *startOffset* in order to achieve pagination of the result set.

```
// match all books that have "clustering" in their title
// sorted by publication year and title
// and return 3'rd page of 10 results
Query query = queryFactory.from(Book.class)
  .orderBy("publicationYear", SortOrder.DESC)
  .orderBy("title", SortOrder.ASC)
  .startOffset(20)
  .maxResults(10)
  .having("title").like("%clustering%")
  .build();
```

Even if the results being fetched are limited to *maxResults* you can still find the total number of matching results by calling `Query.getResultSize()`.

TODO Does pagination make sense if no stable sort criteria is defined? Luckily when running on Lucene and no sort criteria is specified we still have the order of relevance, but this has to be defined for other search engines.

Grouping and Aggregation

{brandname} has the ability to group query results according to a set of grouping fields and construct aggregations of the results from each group by applying an aggregation function to the set of values that fall into each group. Grouping and aggregation can only be applied to projection queries. The supported aggregations are: avg, sum, count, max, min. The set of grouping fields is specified with the *groupBy(field)* method, which can be invoked multiple times. The order used for defining grouping fields is not relevant. All fields selected in the projection must either be grouping fields or else they must be aggregated using one of the grouping functions described below. A projection field can be aggregated and used for grouping at the same time. A query that selects only grouping fields but no aggregation fields is legal.

Example: Grouping Books by author and counting them.

```
Query query = queryFactory.from(Book.class)
    .select(Expression.property("author"), Expression.count("title"))
    .having("title").like("%engine%")
    .groupBy("author")
    .build();
```

A projection query in which all selected fields have an aggregation function applied and no fields are used for grouping is allowed. In this case the aggregations will be computed globally as if there was a single global group.

Aggregations

The following aggregation functions may be applied to a field: avg, sum, count, max, min

- avg() - Computes the average of a set of numbers. Accepted values are primitive numbers and

instances of *java.lang.Number*. The result is represented as *java.lang.Double*. If there are no non-null values the result is *null* instead.

- count() - Counts the number of non-null rows and returns a *java.lang.Long*. If there are no non-null values the result is *0* instead.
- max() - Returns the greatest value found. Accepted values must be instances of *java.lang.Comparable*. If there are no non-null values the result is *null* instead.
- min() - Returns the smallest value found. Accepted values must be instances of *java.lang.Comparable*. If there are no non-null values the result is *null* instead.
- sum() - Computes the sum of a set of Numbers. If there are no non-null values the result is *null* instead. The following table indicates the return type based on the specified field.

Table 2. Table sum return type

Field Type	Return Type
Integral (other than BigInteger)	Long
Float or Double	Double
BigInteger	BigInteger
BigDecimal	BigDecimal

Evaluation of queries with grouping and aggregation

Aggregation queries can include filtering conditions, like usual queries. Filtering can be performed in two stages: before and after the grouping operation. All filter conditions defined before invoking the *groupBy* method will be applied before the grouping operation is performed, directly to the cache entries (not to the final projection). These filter conditions may reference any fields of the queried entity type, and are meant to restrict the data set that is going to be the input for the grouping stage. All filter conditions defined after invoking the *groupBy* method will be applied to the projection that results from the projection and grouping operation. These filter conditions can either reference any of the *groupBy* fields or aggregated fields. Referencing aggregated fields that are not specified in the select clause is allowed; however, referencing non-aggregated and non-grouping fields is forbidden. Filtering in this phase will reduce the amount of groups based on their properties. Sorting may also be specified similar to usual queries. The ordering operation is performed after the grouping operation and can reference any of the *groupBy* fields or aggregated fields.

Using Named Query Parameters

Instead of building a new Query object for every execution it is possible to include named parameters in the query which can be substituted with actual values before execution. This allows a query to be defined once and be efficiently executed many times. Parameters can only be used on the right-hand side of an operator and are defined when the query is created by supplying an object produced by the *org.infinispan.query.dsl.Expression.param(String paramName)* method to the operator instead of the usual constant value. Once the parameters have been defined they can be set by invoking either *Query.setParameter(parameterName, value)* or *Query.setParameters(parameterMap)* as shown in the examples below.

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;
[...]

QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query query = queryFactory.from(Book.class)
    .select("title")
    .having("author").eq(Expression.param("authorName"))
    .and()
    .having("publicationYear").eq(Expression.param("publicationYear"))
    .build();

// Set actual parameter values
query.setParameter("authorName", "Doe");
query.setParameter("publicationYear", 2010);

// Execute the query
List<Book> found = query.list();
```

Alternatively, multiple parameters may be set at once by supplying a map of actual parameter values:

Setting multiple named parameters at once

```
import java.util.Map;
import java.util.HashMap;

[...]

Map<String, Object> parameterMap = new HashMap<>();
parameterMap.put("authorName", "Doe");
parameterMap.put("publicationYear", 2010);

query.setParameters(parameterMap);
```

A significant portion of the query parsing, validation and execution planning effort is performed during the first execution of a query with parameters. This effort is not repeated during subsequent executions leading to better performance compared to a similar query using constant values instead of query parameters.

More Query DSL samples

Probably the best way to explore using the Query DSL API is to have a look at our tests suite. QueryDslConditionsTest is a fine example.

Ickle

Create relational and full-text queries in both Library and Remote Client-Server mode with the Ickle query language.

Ickle is string-based and has the following characteristics:

- Query Java classes and supports Protocol Buffers.
- Queries can target a single entity type.
- Queries can filter on properties of embedded objects, including collections.
- Supports projections, aggregations, sorting, named parameters.
- Supports indexed and non-indexed execution.
- Supports complex boolean expressions.
- Supports full-text queries.
- Does not support computations in expressions, such as `user.age > sqrt(user.shoeSize+3)`.
- Does not support joins.
- Does not support subqueries.
- Is supported across various {{brandname}} APIs. Whenever a Query is produced by the QueryBuilder is accepted, including continuous queries or in event filters for listeners.

To use the API, first obtain a QueryFactory to the cache and then call the .create() method, passing in the string to use in the query. For instance:

```
QueryFactory qf = Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");
```

When using Ickle all fields used with full-text operators must be both `Indexed` and `Analysed`.

Ickle Query Language Parser Syntax

The parser syntax for the Ickle query language has some notable rules:

- Whitespace is not significant.
- Wildcards are not supported in field names.
- A field name or path must always be specified, as there is no default field.
- `&&` and `||` are accepted instead of `AND` or `OR` in both full-text and JPA predicates.
- `!` may be used instead of `NOT`.
- A missing boolean operator is interpreted as `OR`.
- String terms must be enclosed with either single or double quotes.
- Fuzziness and boosting are not accepted in arbitrary order; fuzziness always comes first.
- `!=` is accepted instead of `<>`.

- Boosting cannot be applied to >,>=,<,⇐ operators. Ranges may be used to achieve the same result.

Fuzzy Queries

To execute a fuzzy query add ~ along with an integer, representing the distance from the term used, after the term. For instance

```
Query fuzzyQuery = qf.create("from sample_bank_account.Transaction where description :
'cofee'~2");
```

Range Queries

To execute a range query define the given boundaries within a pair of braces, as seen in the following example:

Query rangeQuery = qf.create("from sample_bank_account.Transaction where amount : [20 to 50]");

Phrase Queries

A group of words may be searched by surrounding them in quotation marks, as seen in the following example:

Query q = qf.create("from sample_bank_account.Transaction where description : 'bus fare'");

Proximity Queries

To execute a proximity query, finding two terms within a specific distance, add a ~ along with the distance after the phrase. For instance, the following example will find the words canceling and fee provided they are not more than 3 words apart:

```
Query proximityQuery = qf.create("from sample_bank_account.Transaction where
description : 'canceling fee'~3 ");
```

Wildcard Queries

Both single-character and multi-character wildcard searches may be performed:

- A single-character wildcard search may be used with the ? character.
- A multi-character wildcard search may be used with the * character.

To search for text or test the following single-character wildcard search would be used:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where
description : 'te?t'");
```

To search for test, tests, or tester the following multi-character wildcard search would be useD:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where
description : 'test*'");
```

Regular Expression Queries

Regular expression queries may be performed by specifing a pattern between /. Ickle uses Lucene's regular expression syntax, so to search for the words moat or boat the following could be used:

```
Query regExpQuery = qf.create("from sample_library.Book  where title : /[mb]oat/");
```

Boosting Queries

Terms may be boosted by adding a ^ after the term to increase their relevance in a given query, the higher the boost factor the more relevant the term will be. For instance to search for titles containing beer and wine with a higher relevance on beer, by a factor of 3, the following could be used:

```
Query boostedQuery = qf.create("from sample_library.Book where title : beer^3 OR wine
");
```

Continuous Query

Continuous Queries allow an application to register a listener which will receive the entries that currently match a query filter, and will be continuously notified of any changes to the queried data set that result from further cache operations. This includes incoming matches, for values that have joined the set, updated matches, for matching values that were modified and continue to match, and outgoing matches, for values that have left the set. By using a Continuous Query the application receives a steady stream of events instead of having to repeatedly execute the same query to discover changes, resulting in a more efficient use of resources. For instance, all of the following use cases could utilize Continuous Queries:

- Return all persons with an age between 18 and 25 (assuming the Person entity has an *age* property and is updated by the user application).
- Return all transactions higher than $2000.
- Return all times where the lap speed of F1 racers were less than 1:45.00s (assuming the cache contains Lap entries and that laps are entered live during the race).

Continuous Query Execution

A continuous query uses a listener that is notified when:

- An entry starts matching the specified query, represented by a *Join* event.
- A matching entry is updated and continues to match the query, represented by an *Update* event.
- An entry stops matching the query, represented by a *Leave* event.

When a client registers a continuous query listener it immediately begins to receive the results currently matching the query, received as *Join* events as described above. In addition, it will receive subsequent notifications when other entries begin matching the query, as *Join* events, or stop matching the query, as *Leave* events, as a consequence of any cache operations that would normally generate creation, modification, removal, or expiration events. Updated cache entries will generate *Update* events if the entry matches the query filter before and after the operation. To summarize, the logic used to determine if the listener receives a *Join*, *Update* or *Leave* event is:

1. If the query on both the old and new values evaluate false, then the event is suppressed.
2. If the query on the old value evaluates false and on the new value evaluates true, then a *Join* event is sent.
3. If the query on both the old and new values evaluate true, then an *Update* event is sent.
4. If the query on the old value evaluates true and on the new value evaluates false, then a *Leave* event is sent.
5. If the query on the old value evaluates true and the entry is removed or expired, then a *Leave* event is sent.

Continuous Queries can use the full power of the Query DSL except: grouping, aggregation, and sorting operations.

Running Continuous Queries

To create a continuous query you'll start by creating a Query object first. This is described in the Query DSL section. Then you'll need to obtain the ContinuousQuery (*org.infinispan.query.api.continuous.ContinuousQuery*) object of your cache and register the query and a continuous query listener (*org.infinispan.query.api.continuous.ContinuousQueryListener*) with it. A ContinuousQuery object associated to a cache can be obtained by calling the static method *org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)* if running in remote mode or *org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)* when running in embedded mode. Once the listener has been created it may be registered by using the addContinuousQueryListener method of ContinuousQuery:

```
continuousQuery.addContinuousQueryListener(query, listener);
```

The following example demonstrates a simple continuous query use case in embedded mode:

Registering a Continuous Query

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
```

```
[...]

// We have a cache of Persons
Cache<Integer, Person> cache = ...

// We begin by creating a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);

// Define our query. In this case we will be looking for any Person instances under 21
years of age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.from(Person.class)
    .having("age").lt(21)
    .build();

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener
<Integer, Person>() {
    @Override
    public void resultJoining(Integer key, Person value) {
        matches.put(key, value);
    }

    @Override
    public void resultUpdated(Integer key, Person value) {
        // we do not process this event
    }

    @Override
    public void resultLeaving(Integer key) {
        matches.remove(key);
    }
};

// Add the listener and the query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);
```

As Person instances having an age less than 21 are added to the cache they will be received by the listener and will be placed into the *matches* map, and when these entries are removed from the cache or their age is modified to be greater or equal than 21 they will be removed from *matches*.

Removing Continuous Queries

To stop the query from further execution just remove the listener:

```
continuousQuery.removeContinuousQueryListener(listener);
```

Notes on performance of Continuous Queries

Continuous queries are designed to provide a constant stream of updates to the application, potentially resulting in a very large number of events being generated for particularly broad queries. A new temporary memory allocation is made for each event. This behavior may result in memory pressure, potentially leading to *OutOfMemoryErrors* (especially in remote mode) if queries are not carefully designed. To prevent such issues it is strongly recommended to ensure that each query captures the minimal information needed both in terms of number of matched entries and size of each match (projections can be used to capture the interesting properties), and that each *ContinuousQueryListener* is designed to quickly process all received events without blocking and to avoid performing actions that will lead to the generation of new matching events from the cache it listens to.

1.3. Remote Querying

Apart from supporting indexing and searching of Java entities to embedded clients, {brandname} introduced support for remote, language neutral, querying.

This leap required two major changes:

- Since non-JVM clients cannot benefit from directly using Apache Lucene's Java API, {brandname} defines its own new query language, based on an internal DSL that is easily implementable in all languages for which we currently have an implementation of the Hot Rod client.
- In order to enable indexing, the entities put in the cache by clients can no longer be opaque binary blobs understood solely by the client. Their structure has to be known to both server and client, so a common way of encoding structured data had to be adopted. Furthermore, allowing multi-language clients to access the data requires a language and platform-neutral encoding. Google's Protocol Buffers was elected as an encoding format for both over-the-wire and storage due to its efficiency, robustness, good multi-language support and support for schema evolution.

1.3.1. Storing Protobuf encoded entities

Remote clients that want to be able to index and query their stored entities must do so using the Protobuf encoding format. This is *key* for the search capability to work. But it's also possible to store Protobuf entities just for gaining the benefit of platform independence and not enable indexing if you do not need it.

Protobuf is all about structured data, so first thing you do to use it is define the structure of your data. This is accomplished by declaring protocol buffer message types in .proto files, like in the following example. Protobuf is a broad subject, we will not detail it here, so please consult the Protobuf Developer Guide for an in-depth explanation. It suffices to say for now that our example

defines an entity (message type in protobuf speak) named *Book*, placed in a package named *book_sample*. Our entity declares several fields of primitive types and a repeatable field (an array basically) named *authors*. The *Author* message instances are embedded in the *Book* message instance.

library.proto

```
package book_sample;

message Book {
  required string title = 1;
  required string description = 2;
  required int32 publicationYear = 3; // no native Date type available in Protobuf

  repeated Author authors = 4;
}

message Author {
  required string name = 1;
  required string surname = 2;
}
```

There are a few important notes we need to make about Protobuf messages:

- nesting of messages is possible, but the resulting structure is strictly a tree, never a graph
- there is no concept of type inheritance
- collections are not supported but arrays can be easily emulated using repeated fields

Using Protobuf with the Java Hot Rod client is a two step process. First, the client must be configured to use a dedicated marshaller, *ProtoStreamMarshaller*. This marshaller uses the *ProtoStream* library to assist you in encoding your objects. The second step is instructing *ProtoStream* library on how to marshall your message types. The following example highlights this process.

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.SerializationContext;
...

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("10.1.2.3").port(11234)
    .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());

SerializationContext serCtx = ProtoStreamMarshaller.getSerializationContext
(remoteCacheManager);

FileDescriptorSource fds = new FileDescriptorSource();
fds.addProtoFiles("/library.proto");
serCtx.registerProtoFiles(fds);
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());

// Book and Author classes omitted for brevity
```

The interesting part in this sample is obtaining the *SerializationContext* associated to the *RemoteCacheManager* and then instructing ProtoStream about the protobuf types we want to marshall. The *SerializationContext* is provided by the library for this purpose. The `SerializationContext.registerProtoFiles` method receives the name of one or more classpath resources that is expected to be a protobuf definition containing our type declarations.

A *RemoteCacheManager* has no *SerializationContext* associated with it unless it was configured to use a *ProtoStreamMarshaller*.

The next relevant part is the registration of per entity marshallers for our domain model types. They must be provided by the user for each type or marshalling will fail. Writing marshallers is a simple process. The *BookMarshaller* example should get you started. The most important thing you need to consider is they need to be stateless and threadsafe as a single instance of them is being used.

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;
...

public class BookMarshaller implements MessageMarshaller<Book> {

   @Override
   public String getTypeName() {
      return "book_sample.Book";
   }

   @Override
   public Class<? extends Book> getJavaClass() {
      return Book.class;
   }

   @Override
   public void writeTo(ProtoStreamWriter writer, Book book) throws IOException {
      writer.writeString("title", book.getTitle());
      writer.writeString("description", book.getDescription());
      writer.writeInt("publicationYear", book.getPublicationYear());
      writer.writeCollection("authors", book.getAuthors(), Author.class);
   }

   @Override
   public Book readFrom(ProtoStreamReader reader) throws IOException {
      String title = reader.readString("title");
      String description = reader.readString("description");
      int publicationYear = reader.readInt("publicationYear");
      Set<Author> authors = reader.readCollection("authors", new HashSet<>(), Author
.class);
      return new Book(title, description, publicationYear, authors);
   }
}
```

Once you've followed these steps to setup your client you can start reading and writing Java objects to the remote cache and the actual data stored in the cache will be protobuf encoded provided that marshallers were registered with the remote client for all involved types (*Book* and *Author* in our example). Keeping your objects stored in protobuf format has the benefit of being able to consume them with compatible clients written in different languages.

TODO Add reference to sample in C++ client user guide

1.3.2. Using annotations

TODO

1.3.3. Indexing of Protobuf encoded entries

After configuring the client as described in the previous section you can start configuring indexing for your caches on the server side. Activating indexing and the various indexing specific configurations is identical to embedded mode and is detailed in the Querying {brandname} chapter.

There is however an extra configuration step involved. While in embedded mode the indexing metadata is obtained via Java reflection by analyzing the presence of various Hibernate Search annotations on the entry's class, this is obviously not possible if the entry is protobuf encoded. The server needs to obtain the relevant metadata from the same descriptor (.proto file) as the client. The descriptors are stored in a dedicated cache on the server named *'___protobuf_metadata'*. Both keys and values in this cache are plain strings. Registering a new schema is therefore as simple as performing a *put* operation on this cache using the schema's name as key and the schema file itself as the value. Alternatively you can use the CLI (via the cache-container=*:register-proto-schemas() operation), the Management Console or the *ProtobufMetadataManager* MBean via JMX. Be aware that, when security is enabled, access to the schema cache via the remote protocols requires that the user belongs to the '___schema_manager' role.

Once indexing is enabled for a cache all fields of Protobuf encoded entries will be fully indexed unless you use the *@Indexed* and *@Field* protobuf schema pseudo-annotations in order to control precisely what fields need to get indexed. The default behaviour can be very inefficient when dealing with types having many or very larger fields so we encourage you to always specify what fields should be indexed instead of relying on the default indexing behaviour. The indexing behaviour for protobuf message types that are not annotated can also be modified per each schema file by setting the protobuf schema option *'indexed_by_default'* to *false* (its default value is considered *true*) at the beginning of your schema file.

```
option indexed_by_default = false;  // This disables indexing of types that are not
annotated for indexing
```

1.3.4. A remote query example

You've managed to configure both client and server to talk protobuf and you've enabled indexing. Let's put some data in the cache and try to search for it then!

```
import org.infinispan.client.hotrod.*;
import org.infinispan.query.dsl.*;
...

RemoteCacheManager remoteCacheManager = ...;
RemoteCache<Integer, Book> remoteCache = remoteCacheManager.getCache();

Book book1 = new Book();
book1.setTitle("Hibernate in Action");
remoteCache.put(1, book1);

Book book2 = new Book();
book2.setTile("Hibernate Search in Action");
remoteCache.put(2, book2);

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(Book.class)
            .having("title").like("%Hibernate Search%")
            .build();

List<Book> list = query.list(); // Voila! We have our book back from the cache!
```

The key part of creating a query is obtaining the *QueryFactory* for the remote cache using the *org.infinispan.client.hotrod.Search.getQueryFactory()* method. Once you have this creating the query is similar to embedded mode which is covered in this section.

1.3.5. Analysis

Analysis is a process that converts input data into one or more terms that you can index and query.

Default Analyzers

{brandname} provides a set of default analyzers as follows:

Definition	Description
standard	Splits text fields into tokens, treating whitespace and punctuation as delimiters.
simple	Tokenizes input streams by delimiting at non-letters and then converting all letters to lowercase characters. Whitespace and non-letters are discarded.
whitespace	Splits text streams on whitespace and returns sequences of non-whitespace characters as tokens.
keyword	Treats entire text fields as single tokens.
stemmer	Stems English words using the Snowball Porter filter.

Definition	Description
`ngram`	Generates n-gram tokens that are 3 grams in size by default.
`filename`	Splits text fields into larger size tokens than the `standard` analyzer, treating whitespace as a delimiter and converts all letters to lowercase characters.

These analyzer definitions are based on Apache Lucene and are provided "as-is". For more information about tokenizers, filters, and CharFilters, see the appropriate Lucene documentation.

Using Analyzer Definitions

To use analyzer definitions, reference them by name in the *.proto* schema file.

1. Include the `Analyze.YES` attribute to indicate that the property is analyzed.
2. Specify the analyzer definition with the `@Analyzer` annotation.

The following example shows referenced analyzer definitions:

```
/* @Indexed */
message TestEntity {

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = "keyword")) */
    optional string id = 1;

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = "simple")) */
    optional string name = 2;
}
```

Creating Custom Analyzer Definitions

If you require custom analyzer definitions, do the following:

1. Create an implementation of the `ProgrammaticSearchMappingProvider` interface packaged in a `JAR` file.
2. Provide a file named `org.infinispan.query.spi.ProgrammaticSearchMappingProvider` in the `META-INF/services/` directory of your `JAR`. This file should contain the fully qualified class name of your implementation.
3. Copy the `JAR` to the `standalone/deployments` directory of your {brandname} installation.

Your deployment must be available to the {brandname} server during startup. You cannot add the deployment if the server is already running.

The following is an example implementation of the `ProgrammaticSearchMappingProvider`

interface:

```
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.core.StopFilterFactory;
import org.apache.lucene.analysis.standard.StandardFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;
import org.hibernate.search.cfg.SearchMapping;
import org.infinispan.Cache;
import org.infinispan.query.spi.ProgrammaticSearchMappingProvider;

public final class MyAnalyzerProvider implements ProgrammaticSearchMappingProvider
{

   @Override
   public void defineMappings(Cache cache, SearchMapping searchMapping) {
      searchMapping
            .analyzerDef("standard-with-stop", StandardTokenizerFactory.class)
               .filter(StandardFilterFactory.class)
               .filter(LowerCaseFilterFactory.class)
               .filter(StopFilterFactory.class);
   }
}
```

4. Specify the JAR in the cache container configuration, for example:

```
<cache-container name="mycache" default-cache="default">
   <modules>
     <module name="deployment.analyzers.jar"/>
   </modules>
...
```

1.4. Statistics

Query *Statistics* can be obtained from the *SearchManager*, as demonstrated in the following code snippet.

```
SearchManager searchManager = Search.getSearchManager(cache);
org.hibernate.search.stat.Statistics statistics = searchManager.getStatistics();
```

This data is also available via JMX through the Hibernate Search StatisticsInfoMBean registered under the name `org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=Statistics`. Please note this MBean is always registered by {brandname} but the statistics are collected only if statistics collection is enabled at cache level.

Hibernate Search has its own configuration properties `hibernate.search.jmx_enabled` and `hibernate.search.generate_statistics` for JMX statistics as explained here. Using them with {brandname} Query is forbidden as it will only lead to duplicated MBeans and unpredictable results.

1.5. Performance Tuning

1.5.1. Batch writing in SYNC mode

By default, the Index Managers work in sync mode, meaning when data is written to {brandname}, it will perform the indexing operations synchronously. This synchronicity guarantees indexes are always consistent with the data (and thus visible in searches), but can slowdown write operations since it will also perform a commit to the index. Committing is an extremely expensive operation in Lucene, and for that reason, multiple writes from different nodes can be automatically batched into a single commit to reduce the impact.

So, when doing data loads to {brandname} with index enabled, try to use multiple threads to take advantage of this batching.

If using multiple threads does not result in the required performance, an alternative is to load data with indexing temporarily disabled and run a re-indexing operation afterwards. This can be done writing data with the `SKIP_INDEXING` flag:

```
cache.getAdvancedCache().withFlags(Flag.SKIP_INDEXING).put("key","value");
```

1.5.2. Writing using async mode

If it's acceptable a small delay between data writes and when that data is visible in queries, an index manager can be configured to work in **async mode**. The async mode offers much better writing performance, since in this mode commits happen at a configurable interval.

Configuration:

```
<distributed-cache name="default">
    <indexing index="LOCAL">
        <property name="default.indexmanager">
              org.infinispan.query.indexmanager.InfinispanIndexManager
        </property>
        <!-- Index data in async mode -->
        <property name="default.worker.execution">async</property>
        <!-- Optional: configure the commit interval, default is 1000ms -->
        <property name="default.index_flush_interval">500</property>
    </indexing>
</distributed-cache>
```

1.5.3. Index reader async strategy

Lucene internally works with snapshots of the index: once an *IndexReader* is opened, it will only see the index changes up to the point it was opened; further index changes will not be visible until the *IndexReader* is refreshed. The Index Managers used in {brandname} by default will check the freshness of the index readers before every query and refresh them if necessary.

It is possible to tune this strategy to relax this freshness checking to a pre-configured interval by using the `reader.strategy` configuration set as `async`:

```
<distributed-cache name="default"
                   key-partitioner=
"org.infinispan.distribution.ch.impl.AffinityPartitioner">
    <indexing index="PRIMARY_OWNER">
        <property name="default.indexmanager">
              org.infinispan.query.affinity.AffinityIndexManager
        </property>
        <property name="default.reader.strategy">async</property>
        <!-- refresh reader every 1s, default is 5s -->
        <property name="default.reader.async_refresh_period_ms">1000</property>
    </indexing>
</distributed-cache>
```

The async reader strategy is particularly useful for Index Managers that rely on shards, such as the AffinityIndexManager.

1.5.4. Lucene Options

It is possible to apply tuning options in Lucene directly. For more details, see the Hibernate Search manual.