Querying Infinispan caches

Table of Contents

Infinispan lets you perform queries with embedded and remote caches to efficiently and quickly look up values in your data set.

Chapter 1. Indexing Infinispan caches

Infinispan can create indexes of values in your caches to improve query performance, providing faster results than non-indexed queries. Indexing also lets you use full-text search capabilities in your queries.

Infinispan uses Apache Lucene technology to index values in caches.

1.1. Configuring Infinispan to index caches

Enable indexing in your cache configuration and specify which entities Infinispan should include when creating indexes.

You should always configure Infinispan to index caches when using queries. Indexing provides a significant performance boost to your queries, allowing you to get faster insights into your data.

Procedure

1. Enable indexing in your cache configuration.

```
<distributed-cache>
  <indexing>
    <!-- Indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

Adding an `indexing` element to your configuration enables indexing without the need to include the `enabled=true` attribute.

For remote caches adding this element also implicitly configures encoding as ProtoStream.

2. Specify the entities to index with the `indexed-entity` element.

```
<distributed-cache>
  <indexing>
    <indexed-entities>
      <indexed-entity>...</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

Protobuf messages

- Specify the message declared in the schema as the value of the `indexed-entity` element, for example:

```
<distributed-cache>
  <indexing>
    <indexed-entities>
      <indexed-entity>org.infinispan.sample.Car</indexed-entity>
      <indexed-entity>org.infinispan.sample.Truck</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

This configuration indexes the `Book` message in a schema with the `book_sample` package name.

```
package book_sample;

/* @Indexed */
message Book {

    /* @Field(store = Store.YES, analyze = Analyze.YES) */
    optional string title = 1;

    /* @Field(store = Store.YES, analyze = Analyze.YES) */
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in
Protobuf

    repeated Author authors = 4;
}

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

Java objects

- Specify the fully qualified name (FQN) of each class that includes the `@Indexed` annotation.

XML

```
<distributed-cache>
  <indexing>
    <indexed-entities>
      <indexed-entity>book_sample.Book</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

`ConfigurationBuilder`

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

ConfigurationBuilder config=new ConfigurationBuilder();
config.indexing().enable().storage(FILESYSTEM).path("/some/folder").addIndexedEntity(B
ook.class);
```

Additional resources

- `org.infinispan.configuration.cache.IndexingConfigurationBuilder`

1.1.1. Index configuration

Infinispan configuration controls how indexes are stored and constructed.

Index storage

You can configure how Infinispan stores indexes:

- On the host file system, which is the default and persists indexes between restarts.
- In JVM heap memory, which means that indexes do not survive restarts.
  You should store indexes in JVM heap memory only for small datasets.

File system

```
<distributed-cache>
  <indexing storage="filesystem" path="${java.io.tmpdir}/baseDir">
    <!-- Indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

JVM heap memory

```
<distributed-cache>
  <indexing storage="local-heap">
    <!-- Additional indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

Index reader

The index reader is an internal component that provides access to the indexes to perform queries. As the index content changes, Infinispan needs to refresh the reader so that search results are up to date. You can configure the refresh interval for the index reader. By default Infinispan reads the index before each query if the index changed since the last refresh.

```
<distributed-cache>
  <indexing storage="filesystem" path="${java.io.tmpdir}/baseDir">
    <!-- Sets an interval of one second for the index reader. -->
    <index-reader refresh-interval="1000"/>
    <!-- Additional indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

Index writer

The index writer is an internal component that constructs an index composed of one or more segments (sub-indexes) that can be merged over time to improve performance. Fewer segments usually means less overhead during a query because index reader operations need to take into account all segments.

Infinispan uses Apache Lucene internally and indexes entries in two tiers: memory and storage. New entries go to the memory index first and then, when a flush happens, to the configured index storage. Periodic commit operations occur that create segments from the previously flushed data and make all the index changes permanent.

The `index-writer` configuration is optional. The defaults should work for most cases and custom configurations should only be used to tune performance.

```
<distributed-cache>
  <indexing storage="filesystem" path="${java.io.tmpdir}/baseDir">
    <index-writer commit-interval="2000"
                  low-level-trace="false"
                  max-buffered-entries="32"
                  queue-count="1"
                  queue-size="10000"
                  ram-buffer-size="400"
                  thread-pool-size="2">
      <index-merge calibrate-by-deletes="true"
                   factor="3"
                   max-entries="2000"
                   min-size="10"
                   max-size="20"/>
    </index-writer>
    <!-- Additional indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

Table 1. Index writer configuration attributes

Attribute	Description
`commit-interval`	Amount of time, in milliseconds, that index changes that are buffered in memory are flushed to the index storage and a commit is performed. Because operation is costly, small values should be avoided. The default is 1000 ms (1 second).
`max-buffered-entries`	Maximum number of entries that can be buffered in-memory before they are flushed to the index storage. Large values result in faster indexing but use more memory. When used in combination with the `ram-buffer-size` attribute, a flush occurs for whichever event happens first.
`ram-buffer-size`	Maximum amount of memory that can be used for buffering added entries and deletions before they are flushed to the index storage. Large values result in faster indexing but use more memory. For faster indexing performance you should set this attribute instead of `max-buffered-entries`. When used in combination with the `max-buffered-entries` attribute, a flush occurs for whichever event happens first.
`thread-pool-size`	Number of threads that execute write operations to the index.
`queue-count`	Number of internal queues to use for each indexed type. Each queue holds a batch of modifications that is applied to the index and queues are processed in parallel. Increasing the number of queues will lead to an increase of indexing throughput, but only if the bottleneck is CPU. For optimum results, do not set a value for `queue-count` that is larger than the value for `thread-pool-size`.
`queue-size`	Maximum number of elements each queue can hold. Increasing the `queue-size` value increases the amount of memory that is used during indexing operations. Setting a value that is too small can block indexing operations.
`low-level-trace`	Enables low-level trace information for indexing operations. Enabling this attribute substantially degrades performance. You should use this low-level tracing only as a last resource for troubleshooting.

To configure how Infinispan merges index segments, you use the `index-merge` sub-element.

Table 2. Index merge configuration attributes

Attribute	Description
`max-entries`	Maximum number of entries that an index segment can have before merging. Segments with more than this number of entries are not merged. Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.
`factor`	Number of segments that are merged at once. With smaller values, merging happens more often, which uses more resources, but the total number of segments will be lower on average, increasing search performance. Larger values (greater than 10) are best for heavy writing scenarios.
`min-size`	Minimum target size of segments, in MB, for background merges. Segments smaller than this size are merged more aggressively. Setting a value that is too large might result in expensive merge operations, even though they are less frequent.
`max-size`	Maximum size of segments, in MB, for background merges. Segments larger than this size are never merged in the background. Settings this to a lower value helps reduce memory requirements and avoids some merging operations at the cost of optimal search speed. This attribute is ignored when forcefully merging an index and `max-forced-size` applies instead.
`max-forced-size`	Maximum size of segments, in MB, for forced merges and overrides the `max-size` attribute. Set this to the same value as `max-size` or lower. However setting the value too low degrades search performance because documents are deleted.

Attribute	Description
`calibrate-by-deletes`	Whether the number of deleted entries in an index should be taken into account when counting the entries in the segment. Setting `false` will lead to more frequent merges caused by `max-entries`, but will more aggressively merge segments with many deleted documents, improving query performance.

Reference

For more information about indexing elements and attributes, refer to the Infinispan Configuration Schema.

1.2. Indexing annotations

When you enable indexing in caches, you configure Infinispan to create indexes. You also need to provide Infinispan with a structured representation of the entities in your caches so it can actually index them.

There are two annotations that control the entities and fields that Infinispan indexes:

`@Indexed`
: Indicates entities, or Protobuf message types, that Infinispan indexes.

`@Field`
: Indicates fields that Infinispan indexes and has the following attributes:

Attribute	Description	Values
`index`	Controls if Infinispan includes fields in indexes.	`Index.YES` or `Index.NO`
`store`	Allows Infinispan to store fields in indexes so you can use them for projections.	`Store.YES` or `Store.NO`
`analyze`	Includes fields in full-text searches.	`Analyze.NO` or specifies an analyzer definition

Remote caches

You can provide Infinispan with indexing annotations for remote caches in two ways:

- Annotate your Java classes directly with `@ProtoDoc("@Indexed")` and `@ProtoDoc("@Field(…)")`. You then generate Protobuf schema, `.proto` files, before uploading them to Infinispan Server.
- Annotate Protobuf schema directly with `@Indexed` and `@Field(…)`. You then upload your Protobuf schema to Infinispan Server.

Embedded caches

For embedded caches, you add indexing annotations to your Java classes according to how Infinispan stores your entries.

- Use the `@Indexed` and `@Field` annotations, along with other Hibernate Search annotations such as `@FullTextField`, for Plain Old Java Objects (POJOs).
- Use the `@ProtoDoc("@Indexed")` and `@ProtoDoc("@Field(…)")` annotations for objects encoded as Protobuf.

1.3. Rebuilding indexes

Rebuilding an index reconstructs it from the data stored in the cache. You should rebuild indexes when you change things like the definitions of indexed types or analyzers. Likewise, you can rebuild indexes after you delete them for whatever reason.

Rebuilding indexes can take a long time to complete because the process takes place for all data in the grid. While the rebuild operation is in progress, queries might also return fewer results.

Procedure

Rebuild indexes in one of the following ways:

- Call the `reindexCache()` method to programmatically rebuild an index from a Hot Rod Java client:

```
remoteCacheManager.administration().reindexCache("MyCache");
```

For remote caches you can also rebuild indexes from Infinispan Console.

- Call the `index.run()` method to rebuild indexes for embedded caches as follows:

```
Indexer indexer = Search.getIndexer(cache);
CompletionStage<Void> future = index.run();
```

1.4. Non-indexed queries

Infinispan recommends indexing caches for the best performance for queries. However you can query caches that are non-indexed.

- For embedded caches, you can perform non-indexed queries on Plain Old Java Objects (POJOs).
- For remote caches, you must use ProtoStream encoding with the `application/x-protostream` media type to perform non-indexed queries.

Chapter 2. Creating Ickle queries

Infinispan provides an Ickle query language that lets you create relational and full-text queries.

2.1. Ickle queries

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. Each `QueryFactory` instance is bound to the same `Cache` instance as the `Search`, but it is otherwise a stateless and thread-safe object that can be used for creating multiple queries in parallel.

For instance:

```
// Remote Query, using protobuf
QueryFactory qf = org.infinispan.client.hotrod.Search.getQueryFactory(remoteCache);
Query<Transaction> q = qf.create("from sample_bank_account.Transaction where amount >
20");

// Embedded Query using Java Objects
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);
Query<Transaction> q = qf.create("from org.infinispan.sample.Book where price > 20");

// Execute the query
QueryResult<Book> queryResult = q.execute();
```

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.

Executing the query and fetching the results is as simple as invoking the `execute()` method of the `Query` object. Once executed, calling `execute()` on the same instance will re-execute the query.

Pagination

You can limit the number of returned results by using the `Query.maxResults(int maxResults)`. This can be used in conjunction with `Query.startOffset(long startOffset)` to achieve pagination of the result set.

```
// sorted by year and match all books that have "clustering" in their title
// and return the third page of 10 results
Query<Book> query = queryFactory.create("FROM org.infinispan.sample.Book WHERE title
like '%clustering%' ORDER BY year").startOffset(20).maxResults(10)
```

Number of hits

The `QueryResult` object has the `.hitCount()` method to return the total number of results of the

query, regardless of any pagination parameter. The hit count is only available for indexed queries for performance reasons.

Iteration

The `Query` object has the `.iterator()` method to obtain the results lazily. It returns an instance of `CloseableIterator` that must be closed after usage.

The iteration support for Remote Queries is currently limited, as it will first fetch all entries to the client before iterating.

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.

```
QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query<Book> query = queryFactory.create("SELECT title FROM org.infinispan.sample.Book
WHERE author = :authorName AND publicationYear = :publicationYear").build();

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

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

Alternatively, you can supply a map of actual parameter values to set multiple parameters at once:

Setting multiple named parameters at once

```
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.

2.2. Ickle query language syntax

The Ickle query language is subset of the JPQL query language, with some extensions for full-text.

The parser syntax 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.

Filtering operators

Ickle support many filtering operators that can be used for both indexed and non-indexed fields.

Operator	Description	Example
in	Checks that the left operand is equal to one of the elements from the Collection of values given as argument.	FROM Book WHERE isbn IN ('ZZ', 'X1234')
like	Checks that the left argument (which is expected to be a String) matches a wildcard pattern that follows the JPA rules.	FROM Book WHERE title LIKE '%Java%'
=	Checks that the left argument is an exact match of the given value	FROM Book WHERE name = 'Programming Java'
!=	Checks that the left argument is different from the given value	FROM Book WHERE language != 'English'

Operator	Description	Example
>	Checks that the left argument is greater than the given value.	FROM Book WHERE price > 20
>=	Checks that the left argument is greater than or equal to the given value.	FROM Book WHERE price >= 20
<	Checks that the left argument is less than the given value.	FROM Book WHERE year < 2012
⇐	Checks that the left argument is less than or equal to the given value.	FROM Book WHERE price ⇐ 50
between	Checks that the left argument is between the given range limits.	FROM Book WHERE price BETWEEN 50 AND 100

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 the operators 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"

FROM org.infinispan.sample.Book WHERE title LIKE '%Data Grid%' OR author.name = '
Manik' AND description like '%clustering%'
```

Boolean negation 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"
FROM org.infinispan.sample.Book WHERE title != 'Data Grid' AND author.name = 'Manik'
```

Nested conditions

Changing the precedence of logical operators is achieved with parenthesis:

```
# match all books that have an author named "Manik" and their title contains
# "Data Grid" or their description contains "clustering"
FROM org.infinispan.sample.Book WHERE author.name = 'Manik' AND ( title like '%Data
Grid%' OR description like '% clustering%')
```

Selecting attributes

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 `QueryResult.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.

```
# match all books that have "Data Grid" in their title or description
# and return only their title and publication year
SELECT title, publicationYear FROM org.infinispan.sample.Book WHERE title like '%Data
Grid%' OR description like '%Data Grid%'
```

Sorting

Ordering the results based on one or more attributes or attribute paths is done with the `ORDER BY` clause. If multiple sorting criteria are specified, then the order will dictate their precedence.

```
# match all books that have "Data Grid" in their title or description
# and return them sorted by the publication year and title
FROM org.infinispan.sample.Book WHERE title like '%Data Grid%' ORDER BY
publicationYear DESC, title ASC
```

Grouping and Aggregation

Infinispan 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 (queries with one or more field in the SELECT clause).

The supported aggregations are: `avg`, `sum`, `count`, `max`, and `min`.

The set of grouping fields is specified with the `GROUP BY` clause and 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.

```
SELECT author, COUNT(title) FROM org.infinispan.sample.Book WHERE title LIKE '
%engine%' GROUP BY author
```

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

You can apply the following aggregation functions to a field:

Table 3. Index merge attributes

Aggregation function	Description
`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 4. 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 can 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 can 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.

2.3. Full-text queries

You can perform full-text searches with the Ickle query language.

Fuzzy queries

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

```
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:

```
FROM sample_bank_account.Transaction WHERE amount : [20 to 50]
```

Phrase queries

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

```
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:

```
FROM sample_bank_account.Transaction WHERE description : 'canceling fee'~3
```

Wildcard queries

To search for "text" or "test", use the `?` single-character wildcard search:

```
FROM sample_bank_account.Transaction where description : 'te?t'
```

To search for "test", "tests", or "tester", use the * multi-character wildcard search:

```
FROM sample_bank_account.Transaction where description : 'test*'
```

Regular expression queries

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

```
FROM sample_library.Book  where title : /[mb]oat/
```

Boosting queries

Terms can 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:

```
FROM sample_library.Book WHERE title : beer^3 OR wine
```

Chapter 3. Querying remote caches

You can index and query remote caches on Infinispan Server.

3.1. Querying caches from Hot Rod Java clients

Infinispan lets you programmatically query remote caches from Java clients through the Hot Rod endpoint. This procedure explains how to index query a remote cache that stores `Book` instances.

Prerequisites

- Add the ProtoStream processor to your `pom.xml`.

Infinispan provides this processor for the `@ProtoField` and `@ProtoDoc` annotations so you can generate Protobuf schemas and perform queries.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.infinispan.protostream</groupId>
    <artifactId>protostream-processor</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Procedure

1. Add indexing annotations to your class, as in the following example:

Book.java

```
import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoDoc("@Indexed")
public class Book {

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 1)
   final String title;

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 2)
   final String description;

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 3, defaultValue = "0")
   final int publicationYear;


   @ProtoFactory
   Book(String title, String description, int publicationYear) {
      this.title = title;
      this.description = description;
      this.publicationYear = publicationYear;
   }
   // public Getter methods omitted for brevity
}
```

2. Implement the `SerializationContextInitializer` interface in a new class and then add the `@AutoProtoSchemaBuilder` annotation.
   a. Reference the class that includes the `@ProtoField` and `@ProtoDoc` annotations with the `includeClasses` parameter.
   b. Define a name for the Protobuf schema that you generate and filesystem path with the `schemaFileName` and `schemaFilePath` parameters.
   c. Specify the package name for the Protobuf schema with the `schemaPackageName` parameter.

RemoteQueryInitializer.java

```
import org.infinispan.protostream.SerializationContextInitializer;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(
      includeClasses = {
            Book.class
      },
      schemaFileName = "book.proto",
      schemaFilePath = "proto/",
      schemaPackageName = "book_sample")
public interface RemoteQueryInitializer extends SerializationContextInitializer
{
}
```

3. Compile your project.

   The code examples in this procedure generate a `proto/book.proto` schema and an `RemoteQueryInitializerImpl.java` implementation of the annotated `Book` class.

Next steps

Create a remote cache that configures Infinispan to index your entities. For example, the following remote cache indexes the `Book` entity in the `book.proto` schema that you generated in the previous step:

```
<replicated-cache name="books">
  <indexing>
    <indexed-entities>
      <indexed-entity>book_sample.Book</indexed-entity>
    </indexed-entities>
  </indexing>
</replicated-cache>
```

The following `RemoteQuery` class does the following:

- Registers the `RemoteQueryInitializerImpl` serialization context with a Hot Rod Java client.
- Registers the Protobuf schema, `book.proto`, with Infinispan Server.
- Adds two `Book` instances to the remote cache.
- Performs a full-text query that matches books by keywords in the title.

RemoteQuery.java

```
package org.infinispan;

import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.nio.file.Paths;
import java.util.List;

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.Search;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

public class RemoteQuery {

   public static void main(String[] args) throws Exception {
      ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
      // RemoteQueryInitializerImpl is generated
      clientBuilder.addServer().host("127.0.0.1").port(11222)
            .security().authentication().username("user").password("user")
            .addContextInitializers(new RemoteQueryInitializerImpl());

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

      // Grab the generated protobuf schema and registers in the server.
      Path proto = Paths.get(RemoteQuery.class.getClassLoader()
            .getResource("proto/book.proto").toURI());
      String protoBufCacheName = ProtobufMetadataManagerConstants
.PROTOBUF_METADATA_CACHE_NAME;
      remoteCacheManager.getCache(protoBufCacheName).put("book.proto", Files
.readString(proto));

      // Obtain the 'books' remote cache
      RemoteCache<Object, Object> remoteCache = remoteCacheManager.getCache("books");

      // Add some Books
      Book book1 = new Book("Infinispan in Action", "Learn Infinispan with using it",
2015);
      Book book2 = new Book("Cloud-Native Applications with Java and Quarkus", "Build
robust and reliable cloud applications", 2019);

      remoteCache.put(1, book1);
      remoteCache.put(2, book2);

      // Execute a full-text query
      QueryFactory queryFactory = Search.getQueryFactory(remoteCache);
      Query<Book> query = queryFactory.create("FROM book_sample.Book WHERE
title:'java'");

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

```
}
```

Additional resources

- Marshalling and Encoding Data for more information about creating serialization contexts and registering Protobuf schema.
- ProtoStream annotations for more information about the `@ProtoField`, `@ProtoDoc`, and `@AutoProtoSchemaBuilder` annotations.

3.2. Querying caches from Infinispan Console and CLI

Infinispan Console and the Infinispan Command Line Interface (CLI) let you query indexed and non-indexed remote caches. You can also use any HTTP client to index and query caches via the REST API.

This procedure explains how to index and query a remote cache that stores `Person` instances.

Prerequisites

- Have at least one running Infinispan Server instance.
- Have Infinispan credentials with create permissions.

Procedure

1. Add indexing annotations to your Protobuf schema, as in the following example:

```
package org.infinispan.example;

/* @Indexed */
message Person {
    /* @Field(index=Index.YES, store = Store.NO, analyze = Analyze.NO) */
    optional int32 id = 1;

    /* @Field(index=Index.YES, store = Store.YES, analyze = Analyze.NO) */
    required string name = 2;

    /* @Field(index=Index.YES, store = Store.YES, analyze = Analyze.NO) */
    required string surname = 3;

    /* @Field(index=Index.YES, store = Store.YES, analyze = Analyze.NO) */
    optional int32 age = 6;

}
```

   From the Infinispan CLI, use the `schema` command with the `--upload=` argument as follows:

```
[//containers/default]> schema --upload=person.proto person.proto
```

2. Create a cache named **people** that uses ProtoStream encoding and configures Infinispan to

index entities declared in your Protobuf schema.

The following cache indexes the `Person` entity from the previous step:

```
<distributed-cache name="people">
  <encoding media-type="application/x-protostream"/>
  <indexing>
    <indexed-entities>
      <indexed-entity>org.infinispan.example.Person</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

From the CLI, use the `create cache` command with the `--file=` argument as follows:

```
[//containers/default]> create cache --file=people.xml people
```

3. Add entries to the cache.

   To query a remote cache, it needs to contain some data. For this example procedure, create entries that use the following JSON values:

   PersonOne

   ```
   {
     "_type":"org.infinispan.example.Person",
     "id":1,
     "name":"Person",
     "surname":"One",
     "age":44
   }
   ```

   PersonTwo

   ```
   {
     "_type":"org.infinispan.example.Person",
     "id":2,
     "name":"Person",
     "surname":"Two",
     "age":27
   }
   ```

PersonThree

```
{
  "_type":"org.infinispan.example.Person",
  "id":3,
  "name":"Person",
  "surname":"Three",
  "age":35
}
```

From the CLI, use the `put` command with the `--file=` argument to add each entry, as follows:

```
[//containers/default/caches/people]> put --encoding=application/json
--file=personone.json personone
```

From Infinispan Console, you must select **Custom Type** for the **Value content type** field when you add values in JSON format with custom types .

4. Query your remote cache.

   From the CLI, use the `query` command from the context of the remote cache.

   ```
   [//containers/default/caches/people]> query "from org.infinispan.example.Person p
   WHERE p.name='Person' ORDER BY p.age ASC"
   ```

   The query returns all entries with a name that matches `Person` by age in ascending order.

Additional resources

- Infinispan REST API

3.3. Using analyzers with remote caches

Analyzers convert input data into terms that you can index and query. You specify analyzer definitions with the `@Field` annotation in your Java classes or directly in Protobuf schema.

Procedure

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

Protobuf schema

```
/* @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;
}
```

Java classes

```
@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = \"keyword\"))")
@ProtoField(1)
final String id;

@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = \"simple\"))")
@ProtoField(2)
final String description;
```

3.3.1. Default analyzer definitions

Infinispan provides a set of default analyzer definitions.

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.
`ngram`	Generates n-gram tokens that are 3 grams in size by default.

Definition	Description
`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.

3.3.2. Creating custom analyzer definitions

Create custom analyzer definitions and add them to your Infinispan Server installations.

Prerequisites

- Stop Infinispan Server if it is running.

  Infinispan Server loads classes at startup only.

Procedure

1. Implement the `ProgrammaticSearchMappingProvider` API.
2. Package your implementation in a JAR with the fully qualified class (FQN) in the following file:

   ```
   META-INF/services/org.infinispan.query.spi.ProgrammaticSearchMappingProvider
   ```

3. Copy your JAR file to the `server/lib` directory of your Infinispan Server installation.
4. Start Infinispan Server.

`ProgrammaticSearchMappingProvider` *example*

```
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);
   }
}
```

Chapter 4. Querying embedded caches

Use embedded queries when you add Infinispan as a library to custom applications.

Protobuf mapping is not required with embedded queries. Indexing and querying are both done on top of Java objects.

4.1. Querying embedded caches

This section explains how to query an embedded cache using an example cache named "books" that stores indexed `Book` instances.

In this example, each `Book` instance defines which properties are indexed and specifies some advanced indexing options with Hibernate Search annotations as follows:

Book.java

```
package org.infinispan.sample;

import java.time.LocalDate;
import java.util.HashSet;
import java.util.Set;

import org.hibernate.search.mapper.pojo.mapping.definition.annotation.*;

// Annotate values with @Indexed to add them to indexes
// Annotate each fields according to how you want to index it
@Indexed
public class Book {
   @FullTextField
   String title;

   @FullTextField
   String description;

   @KeywordField
   String isbn;

   @GenericField
   LocalDate publicationDate;

   @IndexedEmbedded
   Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```
package org.infinispan.sample;

import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;

public class Author {
   @FullTextField
   String name;

   @FullTextField
   String surname;
}
```

Procedure

1. Configure Infinispan to index the "books" cache and specify `org.infinispan.sample.Book` as the entity to index.

```
<distributed-cache name="books">
  <indexing path="${user.home}/index">
    <indexed-entities>
      <indexed-entity>org.infinispan.sample.Book</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

2. Obtain 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");
```

3. Perform queries for fields in the `Book` instances that are stored in the Infinispan cache, as in the following example:

```
// Get the query factory from the cache
QueryFactory queryFactory = org.infinispan.query.Search.getQueryFactory(cache);

// Create an Ickle query that performs a full-text search using the ':' operator on
the 'title' and 'authors.name' fields
// You can perform full-text search only on indexed caches
Query<Book> fullTextQuery = queryFactory.create("FROM org.infinispan.sample.Book b
WHERE b.title:'infinispan' AND b.authors.name:'sanne'");

// Use the '=' operator to query fields in caches that are indexed or not
// Non full-text operators apply only to fields that are not analyzed
Query<Book> exactMatchQuery=queryFactory.create("FROM org.infinispan.sample.Book b
WHERE b.isbn = '12345678' AND b.authors.name : 'sanne'");

// You can use full-text and non-full text operators in the same query
Query<Book> query=queryFactory.create("FROM org.infinispan.sample.Book b where
b.authors.name : 'Stephen' and b.description : (+'dark' -'tower')");

// Get the results
List<Book> found=query.execute().list();
```

4.2. Entity mapping annotations

Add annotations to your Java classes to map your entities to indexes.

Hibernate Search API

Infinispan uses the Hibernate Search API 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.

The following sections provide information that applies to entity mapping annotations for use with Infinispan.

For complete detail about these annotations, you should refer to the Hibernate Search manual.

@DocumentId

Unlike Hibernate Search, using `@DocumentId` to mark a field as identifier does not apply to Infinispan values; in Infinispan 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`. Infinispan 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

Use the `key-transformers` xml element in both embedded and server config:

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

Alternatively, use the Java configuration API (embedded mode):

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

4.3. Programmatically mapping entities

You can programmatically map entities to the index as an alternative to annotating Java classes.

In the following example we map an object `Author` which is to be stored in the grid and made

searchable on two properties:

```
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.NONE)
        .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;
```

Chapter 5. Creating continuous queries

Applications can register listeners to receive continual updates about cache entries that match query filters.

5.1. Continuous queries

Continuous queries provide applications with real-time notifications about data in Infinispan caches that are filtered by queries. When entries match the query Infinispan sends the updated data to any listeners, which provides a stream of events instead of applications having to execute the query.

Continuous queries can notify applications about 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.

For example, continuous queries can notify applications about all:

- Persons with an age between 18 and 25, assuming the `Person` entity has an `age` property and is updated by the user application.
- Transactions for dollar amounts larger than $2000.
- Times where the lap speed of F1 racers were less than 1:45.00 seconds, assuming the cache contains Lap entries and that laps are entered during the race.

Continuous queries can use all query capabilities except for grouping, aggregation, and sorting operations.

How continuous queries work

Continuous queries notify client listeners with the following events:

`Join`
: A cache entry matches the query.

`Update`
: A cache entry that matches the query is updated and still matches the query.

`Leave`
: A cache entry no longer matches the query.

When a client registers a continuous query listener it immediately receives `Join` events for any entries that match the query. Client listeners receive subsequent events each time a cache operation modifies entries that match the query.

Infinispan determines when to send `Join`, `Update`, or `Leave` events to client listeners as follows:

- If the query on both the old and new value does not match, Infinispan does not sent an event.
- If the query on the old value does not match but the new value does, Infinispan sends a `Join`

event.

- If the query on both the old and new values match, Infinispan sends an `Update` event.
- If the query on the old value matches but the new value does not, Infinispan sends a `Leave` event.
- If the query on the old value matches and the entry is then deleted or it expires, Infinispan sends a `Leave` event.

Continuous query performance impact

Continuous queries are designed to provide a constant stream of updates to applications, which can generate a significant number of events. Infinispan temporarily allocates memory for each event it generates, which can result in memory pressure and potentially lead to `OutOfMemory` exceptions, especially for remote caches. For this reason, you should carefully design your continuous queries to avoid any performance impact.

Infinispan strongly recommends that you limit the scope of your continuous queries to the smallest amount of information that you need. To achieve this, you can use projections and predicates. For example, `SELECT field1, field2 FROM Entity WHERE x AND y` provides results about only a subset of fields that match the criteria rather than the entire entry.

It is also important to ensure that each `ContinuousQueryListener` you create can quickly process all received events without blocking threads. To achieve this, you should avoid any cache operations that generate events unnecessarily.

5.2. Creating continuous queries

You can create continuous queries for remote and embedded caches.

Procedure

1. Create a `Query` object.
2. Obtain the `ContinuousQuery` object of your cache by calling the appropriate method:
   - Remote caches: `org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)`
   - Embedded caches: `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)`
3. Register the query and a `ContinuousQueryListener` object as follows:

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

4. When you no longer need the continuous query, remove the listener as follows:

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

Continuous query example

The following code example demonstrates a simple continuous query with an embedded cache.

In this example, the listener receives notifications when any `Person` instances under the age of 21 are added to the cache. Those `Person` instances are also added to the "matches" map. When the entries are removed from the cache or their age becomes greater than or equal to 21, they are removed from "matches" map.

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 Person objects.
Cache<Integer, Person> cache = ...

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

// Define a query.
// In this example, we search for Person instances under 21 years of age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.create("FROM Person p WHERE p.age < 21");

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);
```

Chapter 6. Monitoring and tuning Infinispan queries

Infinispan exposes statistics for queries and provides attributes that you can adjust to improve query performance.

6.1. Getting query statistics

Collect statistics to gather information about performance of your indexes and queries, including information such as the types of indexes and average time for queries to complete.

Procedure

Do one of the following:

- Invoke the `getSearchStatistics()` or `getClusteredSearchStatistics()` methods for embedded caches.
- Use `GET` requests to obtain statistics for remote caches from the REST API.

Embedded caches

```
// Statistics for the local cluster member
SearchStatistics statistics = Search.getSearchStatistics(cache);

// Consolidated statistics for the whole cluster
CompletionStage<SearchStatisticsSnapshot> statistics = Search
.getClusteredSearchStatistics(cache)
```

Remote caches

```
GET /v2/caches/{cacheName}/search/stats
```

6.2. Tuning query performance

Use the following guidelines to help you improve the performance of indexing operations and queries.

Checking index usage statistics

Queries against partially indexed caches return slower results. For instance, if some fields in a schema are not annotated then the resulting index does not include those fields.

Start tuning query performance by checking the time it takes for each type of query to run. If your queries seem to be slow, you should make sure that queries are using the indexes for caches and that all entities and field mappings are indexed.

Adjusting the commit interval for indexes

Indexing can degrade write throughput for Infinispan clusters. The `commit-interval` attribute defines the interval, in milliseconds, between which index changes that are buffered in memory are flushed to the index storage and a commit is performed.

This operation is costly so you should avoid configuring an interval that is too small. The default is 1000 ms (1 second).

Adjusting the refresh interval for queries

The `refresh-interval` attribute defines the interval, in milliseconds, between which the index reader is refreshed.

The default value is `0`, which returns data in queries as soon as it is written to a cache.

A value greater than `0` results in some stale query results but substantially increases throughput, especially in write-heavy scenarios. If you do not need data returned in queries as soon as it is written, you should adjust the refresh interval to improve query performance.