Query your data in Data Grid caches
Query your data in Data Grid caches
Abstract

Data Grid lets you perform queries with embedded and remote caches to efficiently and quickly look up values in your data set.
Table of Contents

RED HAT DATA GRID .............................................................................. 4

DATA GRID DOCUMENTATION .............................................................. 5

DATA GRID DOWNLOADS ..................................................................... 6

MAKING OPEN SOURCE MORE INCLUSIVE .......................................... 7

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION ..................... 8

CHAPTER 1. INDEXING DATA GRID CACHES .......................................... 9
  1.1. CONFIGURING DATA GRID TO INDEX CACHES ......................... 9
    Protobuf messages ........................................................................ 9
    Java objects .............................................................................. 10
    1.1.1. Index configuration ............................................................. 11
      Index storage ........................................................................ 11
      Index path ........................................................................... 11
      Index startup mode ................................................................. 11
      Index reader ....................................................................... 12
      Index writer ...................................................................... 12
  1.2. DATA GRID NATIVE INDEXING Annotations ............................... 15
    Overview of the Data Grid indexing annotations .......................... 15
    Using Data Grid annotations ..................................................... 16
  1.3. REBUILDING INDEXES ................................................................ 16
  1.4. UPDATING INDEX SCHEMA ........................................................ 17
  1.5. NON-INDEXED QUERIES .............................................................. 18

CHAPTER 2. CREATING ICKLE QUERIES .............................................. 19
  2.1. ICKLE QUERIES ......................................................................... 19
    2.1.1. Pagination .................................................................... 19
    2.1.2. Number of hits ............................................................... 19
      2.1.2.1. Hit count accuracy .................................................... 20
    2.1.3. Iteration ....................................................................... 20
    2.1.4. Named query parameters .............................................. 20
    2.1.5. Query execution .............................................................. 21
  2.2. ICKLE QUERY LANGUAGE SYNTAX ......................................... 21
    2.2.1. Filtering operators ........................................................... 22
    2.2.2. Boolean conditions ........................................................ 22
    2.2.3. Nested conditions .......................................................... 23
    2.2.4. Projections with SELECT statements ............................... 23
      2.2.4.1. Project cache entry version ...................................... 23
      2.2.4.2. Project cache entry value ........................................ 23
    Sorting ..................................................................................... 23
    2.2.5. Grouping and aggregation .............................................. 24
      Aggregations ...................................................................... 24
      Evaluation of queries with grouping and aggregation ............ 25
    2.2.6. DELETE statements ........................................................ 25
  2.3. FULL-TEXT QUERIES ................................................................. 26
    2.3.1. Fuzzy queries ................................................................ 26
    2.3.2. Range queries ................................................................. 26
    2.3.3. Phrase queries ............................................................... 26
    2.3.4. Proximity queries ........................................................... 26
    2.3.5. Wildcard queries ............................................................ 26
2.3.6. Regular expression queries 27
2.3.7. Boosting queries 27

CHAPTER 3. QUERYING REMOTE CACHES ................................................. 28
  3.1. QUERYING CACHES FROM HOT ROD JAVA CLIENTS 28
  3.2. QUERYING PROTOSTREAM COMMON TYPES 31
  3.3. QUERYING CACHES FROM DATA GRID CONSOLE AND CLI 32
  3.4. USING ANALYZERS WITH REMOTE CACHES 34
    3.4.1. Default analyzer definitions 35
    3.4.2. Creating custom analyzer definitions 35

CHAPTER 4. QUERYING EMBEDDED CACHES ........................................ 37
  4.1. QUERYING EMBEDDED CACHES 37
  4.2. ENTITY MAPPING ANNOTATIONS 38
    @DocumentId 39
    @Transformable keys 39

CHAPTER 5. CREATING CONTINUOUS QUERIES ................................. 41
  5.1. CONTINUOUS QUERIES 41
    How continuous queries work 41
    5.1.1. Continuous queries and Data Grid performance 42
  5.2. CREATING CONTINUOUS QUERIES 42
    Continuous query example 42

CHAPTER 6. MONITORING AND TUNING DATA GRID QUERIES ............. 44
  6.1. GETTING QUERY STATISTICS 44
  6.2. TUNING QUERY PERFORMANCE 44
RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

**Schemaless data structure**
Flexibility to store different objects as key-value pairs.

**Grid-based data storage**
Designed to distribute and replicate data across clusters.

**Elastic scaling**
Dynamically adjust the number of nodes to meet demand without service disruption.

**Data interoperability**
Store, retrieve, and query data in the grid from different endpoints.
DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.4 Documentation
- Data Grid 8.4 Component Details
- Supported Configurations for Data Grid 8.4
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality
DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE
You must have a Red Hat account to access and download Data Grid software.
Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you’d like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.

NOTE

You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the Multi-page HTML format.
2. Click the Feedback button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the Add Feedback dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click Submit.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click Submit and start watching the issue or add more comments.

Thank you for the valuable feedback.
CHAPTER 1. INDEXING DATA GRID CACHES

Data Grid 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.

NOTE

Data Grid uses Apache Lucene technology to index values in caches.

1.1. CONFIGURING DATA GRID TO INDEX CACHES

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

You should always configure Data Grid 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>
```

TIP

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>org.infinispan.sample.Car</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-entities>
  </indexing>
</distributed-cache>
```
This configuration indexes the **Book** message in a schema with the **book_sample** package name.

```java
package book_sample;

/*** @Indexed */
message Book {

  /** @Text(projectable = true) */
  optional string title = 1;

  /** @Text(projectable = true) */
  optional string description = 2;

  // no native Date type available in Protobuf
  optional int32 publicationYear = 3;

  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**

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

**ConfigurationBuilder**

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

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

**Additional resources**
1.1.1. Index configuration

Data Grid configuration controls how indexes are stored and constructed.

Index storage
You can configure how Data Grid 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

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

JVM heap memory

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

Index path
Specifies a filesystem path for the index when storage is 'filesystem'. The value can be a relative or absolute path. Relative paths are created relative to the configured global persistent location, or to the current working directory when global state is disabled.

By default, the cache name is used as a relative path for index path.

**IMPORTANT**

When setting a custom value, ensure that there are no conflicts between caches using the same indexed entities.

Index startup mode
When Data Grid starts caches it can perform operations to ensure the index is consistent with data in the cache. By default no indexing operation takes place when a cache starts but you can configure Data Grid to:

- Clear the index when the cache starts.
  - Data Grid performs the clear (purge) operation synchronously. The cache becomes available only when the purge completes.
- Reindex the cache when it starts.
  - Data Grid performs the reindex operation asynchronously. The reindex operation might
take a longer time to complete, depending on the size of the cache.

- Automatically clear or reindex the cache.
  - If data is volatile and the index is persistent then Data Grid clears the cache when it starts.
  - If data is persistent and the index is volatile then Data Grid reindexes the cache when it starts.

### Clear the index when the cache starts

```xml
<distributed-cache>
  <indexing storage="filesystem" startup-mode="purge">
    <!-- Additional indexing configuration goes here. -->
  </indexing>
</distributed-cache>
```

### Rebuild the index when the cache starts

```xml
<distributed-cache>
  <indexing storage="local-heap" startup-mode="reindex">
    <!-- 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, Data Grid 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 Data Grid reads the index before each query if the index changed since the last refresh.

```xml
<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.

Data Grid 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.

**NOTE**

The `index-writer` configuration is optional. The defaults should work for most cases and custom configurations should only be used to tune performance.
Table 1.1. Index writer configuration attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit-interval</td>
<td>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).</td>
</tr>
<tr>
<td>max-buffered-entries</td>
<td>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.</td>
</tr>
<tr>
<td>ram-buffer-size</td>
<td>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.</td>
</tr>
<tr>
<td>thread-pool-size</td>
<td>Number of threads that execute write operations to the index.</td>
</tr>
</tbody>
</table>
To configure how Data Grid merges index segments, you use the index-merge sub-element.

**Table 1.2. Index merge configuration attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>max-entries</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>factor</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>min-size</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
### Attribute | Description
--- | ---
**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.

**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.

### 1.2. DATA GRID NATIVE INDEXING ANNOTATIONS

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

#### Overview of the Data Grid indexing annotations

**@Indexed**

Indicates entities, or Protobuf message types, that Data Grid indexes.

To indicate the fields that Data Grid indexes use the indexing annotations. You can use these annotations the same way for both embedded and remote queries.

**@Basic**

Supports any type of field. Use the **@Basic** annotation for numbers and short strings that don’t require any transformation or processing.

**@Decimal**

Use this annotation for fields that represent decimal values.

**@Keyword**

Use this annotation for fields that are strings and intended for exact matching. Keyword fields are not analyzed or tokenized during indexing.

**@Text**

---

**Additional resources**

- [Data Grid configuration schema reference](#)
Use this annotation for fields that contain textual data and are intended for full-text search capabilities. You can use the analyzer to process the text and to generate individual tokens.

@Embedded
Use this annotation to mark a field as an embedded object within the parent entity. The **NESTED** structure preserves the original object relationship structure while the **FLATTENED** structure makes the leaf fields multivalued of the parent entity. The default structure used by **@Embedded** is **NESTED**.

Each of the annotations supports a set of attributes that you can use to further describe how the entity is indexed.

**Table 1.3. Data Grid annotations and supported attributes**

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Supported attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Basic</td>
<td>Searchable, sortable, projectable, aggregable, indexNullAs</td>
</tr>
<tr>
<td>@Decimal</td>
<td>Searchable, sortable, projectable, aggregable, indexNullAs, decimalScale</td>
</tr>
<tr>
<td>@Keyword</td>
<td>Searchable, sortable, projectable, aggregable, indexNullAs, normalizer, norms</td>
</tr>
<tr>
<td>@Text</td>
<td>Searchable, projectable, norms, analyzer, searchAnalyzer, termVector</td>
</tr>
</tbody>
</table>

**Using Data Grid annotations**
You can provide Data Grid with indexing annotations in two ways:

- Annotate your Java classes or fields directly using the Data Grid annotations. You then generate or update your Protobuf schema, `.proto` files, before uploading them to Data Grid Server.

- Annotate Protobuf schema directly with **@Indexed** and **@Basic, @Keyword** or **@Text**. You then upload your Protobuf schema to Data Grid Server.

For example, the following schema uses the **@Text** annotation:

```java
/**
 * @Text(projectable = true)
 */
required string street = 1;
```

**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.
IMPORTANT

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

  **TIP**

  For remote caches you can also rebuild indexes from Data Grid Console.

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

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

  - Check the status of reindexing operation with the `reindexing` attribute of the index statistics.

1.4. UPDATING INDEX SCHEMA

The update index schema operation lets you add schema changes with a minimal downtime. Instead of removing previously indexed data and recreating the index schema, Data Grid adds new fields to the existing schema. Updating index schema is much faster than rebuilding the index but you can update schema only when your changes do not affect fields that were already indexed.

**IMPORTANT**

You can update index schema only when your changes does not affect previously indexed fields. When you change index field definitions or when you delete fields, you must rebuild the index.

Procedure

- Update index schema for a given cache:
  
  - Call the `updateIndexSchema()` method to programmatically update the index schema from a Hot Rod Java client:
    
    ```java
    remoteCacheManager.administration().updateIndexSchema("MyCache");
    ```

  **TIP**

  For remote caches, you can update index schema using the REST API.
Additional resources

- Rebuilding indexes

1.5. NON-INDEXED QUERIES

Data Grid 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

Data Grid 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:

```java
// 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();
```

**NOTE**

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.

2.1.1. Pagination

You can limit the number of returned results by using the Query.

```java
// 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)
```

**NOTE**

If you don’t explicitly set the maxResults for a query instance, Data Grid limits the number of results returned by the query to 100. You can change the default limit by setting the query.default-max-results cache property.

2.1.2. Number of hits
The `QueryResult` object includes the `hitCount()` method, which returns a hit count value that represents the total number of results from a query, regardless of any pagination parameter. The hit count is only available for indexed queries for performance reasons.

### 2.1.2.1. Hit count accuracy

To optimize performance, the default accuracy of the hit count is set to 10000. You can limit the required accuracy of hit counts by setting `query.hit-count-accuracy` cache property. Alternatively, you can set the limit on each query instance.

When the hit count exceeds the specified limit, Data Grid does not return any value for the hit count. When using the HotRod client or embedded query API, Data Grid returns `null`, and when using the REST query API, Data Grid responds with `-1L`. While setting the hit count accuracy to `Integer.MAX` would return accurate results for any query, it would negatively impact query performance.

For optimal performance, set the property value slightly above the expected hit count. If you do not require precise hit counts, set it to a low value.

### 2.1.3. 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.

**NOTE**

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

### 2.1.4. 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.

```java
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
NOTE
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.

2.1.5. Query execution
The Query API provides two methods for executing Ickle queries on a cache:

- `Query.execute()` runs a SELECT statement and returns a result.
- `Query.executeStatement()` runs a DELETE statement and modifies data.

NOTE
You should always invoke `executeStatement()` to modify data and invoke `execute()` to get the result of a query.

Additional resources
- `org.infinispan.query.dsl.Query.execute()`
- `org.infinispan.query.dsl.Query.executeStatement()`

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 $>\geq<\leq$ operators. Ranges may be used to achieve the same result.

### 2.2.1. Filtering operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in</code></td>
<td>Checks that the left operand is equal to one of the elements from the Collection of values given as argument.</td>
<td>FROM Book WHERE isbn IN (‘ZZ’, ’X1234’)</td>
</tr>
<tr>
<td><code>like</code></td>
<td>Checks that the left argument (which is expected to be a String) matches a wildcard pattern that follows the JPA rules.</td>
<td>FROM Book WHERE title LIKE '%Java%'</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Checks that the left argument is an exact match of the given value.</td>
<td>FROM Book WHERE name = ’Programming Java’</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Checks that the left argument is different from the given value.</td>
<td>FROM Book WHERE language != ’English’</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Checks that the left argument is greater than the given value.</td>
<td>FROM Book WHERE price &gt; 20</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Checks that the left argument is greater than or equal to the given value.</td>
<td>FROM Book WHERE price &gt;= 20</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Checks that the left argument is less than the given value.</td>
<td>FROM Book WHERE year &lt; 2020</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Checks that the left argument is less than or equal to the given value.</td>
<td>FROM Book WHERE price &lt;= 50</td>
</tr>
<tr>
<td><code>between</code></td>
<td>Checks that the left argument is between the given range limits.</td>
<td>FROM Book WHERE price BETWEEN 50 AND 100</td>
</tr>
</tbody>
</table>

### 2.2.2. 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.

```sql
# match all books that have "Data Grid" in their title
# or have an author named "Manik" and their description contains "clustering"
```
Boolean negation has highest precedence among logical operators and applies only to the next simple attribute condition.

2.2.3. Nested conditions

Changing the precedence of logical operators is achieved with parenthesis:

```sql
# 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'
```

2.2.4. Projections with SELECT statements

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.

```sql
# 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 '% clustering%'
```

2.2.4.1. Project cache entry version

It is possible to project the cache entry version, using the `version` projection function.

```sql
# return the title, publication year and the cache entry version
SELECT b.title, b.publicationYear, version(b) FROM org.infinispan.sample.Book b WHERE b.title like '%Data Grid%'
```

2.2.4.2. Project cache entry value

It is possible to project the cache entry value together with other projections. It can be used for instance to project the cache entry value together with the cache entry version in the same `Object[]` returned hit.

```sql
# return the cache entry value and the cache entry version
SELECT b, version(b) FROM org.infinispan.sample.Book b WHERE b.title 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.
2.2.5. Grouping and aggregation

Data Grid 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.

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

**NOTE**

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>
<thead>
<tr>
<th>Aggregation function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg()</td>
<td>Computes the average of a set of numbers. Accepted values are primitive numbers and instances of <strong>java.lang.Number</strong>. The result is represented as <strong>java.lang.Double</strong>. If there are no non-null values the result is <strong>null</strong> instead.</td>
</tr>
<tr>
<td>count()</td>
<td>Counts the number of non-null rows and returns a <strong>java.lang.Long</strong>. If there are no non-null values the result is <strong>0</strong> instead.</td>
</tr>
<tr>
<td>max()</td>
<td>Returns the greatest value found. Accepted values must be instances of <strong>java.lang.Comparable</strong>. If there are no non-null values the result is <strong>null</strong> instead.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Field Type</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral (other than BigInteger)</td>
<td>Long</td>
</tr>
<tr>
<td>Float or Double</td>
<td>Double</td>
</tr>
<tr>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>BigDecimal</td>
</tr>
</tbody>
</table>

Table 2.2. Table sum return type

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.

2.2.6. DELETE statements

You can delete entities from Data Grid caches with the following syntax:

```
DELETE FROM <entityName> [WHERE condition]
```

- Reference only single entities with `<entityName>`. DELETE queries cannot use joins.
- WHERE conditions are optional.

DELETE queries cannot use any of the following:

- Projections with SELECT statements
Grouping and aggregation

ORDER BY clauses

TIP

Invoke the `Query.executeStatement()` method to execute DELETE statements.

Additional resources

- `org.infinispan.query.dsl.Query.executeStatement()`

2.3. FULL-TEXT QUERIES

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

2.3.1. 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
```

2.3.2. 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]
```

2.3.3. 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'
```

2.3.4. 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
```

2.3.5. 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*' 
```

2.3.6. 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/ 
```

2.3.7. 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 Data Grid Server.

3.1. QUERYING CACHES FROM HOT ROD JAVA CLIENTS

Data Grid 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.

Data Grid provides this processor for the @ProtoField annotations so you can generate Protobuf schemas and perform queries.

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
    </dependency>
    <dependency>
      <groupId>org.infinispan.protostream</groupId>
      <artifactId>protostream-processor</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.infinispan.api.annotations.indexing</groupId>
    <artifactId>org.infinispan.protostream</artifactId>
    <version>${version.infinispan}</version>
  </dependency>
</dependencies>
```

Procedure

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

```java
import org.infinispan.api.annotations.indexing.Basic;
import org.infinispan.api.annotations.indexing.Indexed;
import org.infinispan.api.annotations.indexing.Text;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@Indexed
public class Book {

    @Text
    @ProtoField(number = 1)
    final String title;
```
Implement the `SerializationContextInitializer` interface in a new class and then add the `@AutoProtoSchemaBuilder` annotation.

a. Reference the class that includes the `@ProtoField` 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**

```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 {
}
```

2. 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 Data Grid 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:

```xml
<replicated-cache name="books">
```
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 Data Grid Server.
- Adds two `Book` instances to the remote cache.
- Performs a full-text query that matches books by keywords in the title.

### RemoteQuery.java

```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");
    }
}
```
Additional resources

- Encoding Infinispan caches and marshalling data
- Encoding Data Grid caches and marshalling data: ProtoStream annotations

### 3.2. QUERYING PROTOSTREAM COMMON TYPES

Perform Ickle queries on caches that store data as ProtoStream common types such as `BigInteger` and `BigDecimal`.

**Procedure**

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

```java
@Indexed
public class CalculusIndexed {
    @Basic
    @ProtoField(value = 1)
    public BigInteger getPurchases() {
        return purchases;
    }

    @Decimal // the scale is 2 by default
    @ProtoField(value = 2)
    public BigDecimal getProspect() {
        return prospect;
    }
}
```

2. Set the `dependsOn` attribute to `CommonTypes.class` to indicate that the generated Protobuf schema can reference and use `CommonTypes` types such as `BigInteger` and `BigDecimal`:

```java
@AutoProtoSchemaBuilder(includeClasses = CalculusIndexed.class, dependsOn = CommonTypes.class,
    schemaFilePath = "protostream", schemaFileName = "calculus-indexed.proto",
```
Perform queries:

```java
Query<Product> query = queryFactory.create("from lab.indexed.CalculusIndexed c where c.purchases > 9");
QueryResult<Product> result = query.execute();
// play with the result

query = queryFactory.create("from lab.indexed.CalculusIndexed c where c.prospect = 2.2");
result = query.execute();
// play with the result
```

Additional resources

- ProtoStream Types Marshalling

### 3.3. QUERYING CACHES FROM DATA GRID CONSOLE AND CLI

Data Grid Console and the Data Grid 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 Data Grid Server instance.
- Have Data Grid credentials with create permissions.

**Procedure**

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

```protobuf
class Person {
  // @Indexed
  optional int32 id = 1;

  // @Keyword(projectable = true)
  required string name = 2;

  // @Keyword(projectable = true)
  required string surname = 3;

  // @Basic(projectable = true, sortable = true)
  optional int32 age = 6;
}
```

```java
public interface CalculusIndexedSchema extends GeneratedSchema {

  // ... (other methods and implementations)
}
```
From the Data Grid CLI, use the `schema` command with the `--upload=` argument as follows:

```
schema --upload=person.proto person.proto
```

2. Create a cache named `people` that uses ProtoStream encoding and configures Data Grid to index entities declared in your Protobuf schema.

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

```xml
<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:

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

```
put --encoding=application/json --file=personone.json personone
```

**TIP**

From Data Grid 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.

```
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**

- Data Grid REST API

**3.4. USING ANALYZERS WITH REMOTE CACHES**

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

**Procedure**

1. Annotate the property with the `@Text` annotation to indicate that its value is analyzed.

2. Use the `analyzer` attribute to specify the desired analyzer that you want to use for indexing and searching.

**Protobuf schema**

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

    /* @Keyword(projectable = true) */
    optional string id = 1;

    /* @Text(projectable = true, analyzer = "simple") */
    optional string name = 2;
}
```

**Java classes**
Data Grid provides a set of default analyzer definitions.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>Splits text fields into tokens, treating whitespace and punctuation as delimiters.</td>
</tr>
<tr>
<td>simple</td>
<td>Tokenizes input streams by delimiting at non-letters and then converting all letters to lowercase characters. Whitespace and non-letters are discarded.</td>
</tr>
<tr>
<td>whitespace</td>
<td>Splits text streams on whitespace and returns sequences of non-whitespace characters as tokens.</td>
</tr>
<tr>
<td>keyword</td>
<td>Treats entire text fields as single tokens.</td>
</tr>
<tr>
<td>stemmer</td>
<td>Stems English words using the Snowball Porter filter.</td>
</tr>
<tr>
<td>ngram</td>
<td>Generates n-gram tokens that are 3 grams in size by default.</td>
</tr>
<tr>
<td>filename</td>
<td>Splits text fields into larger size tokens than the standard analyzer, treating whitespace as a delimiter and converts all letters to lowercase characters.</td>
</tr>
<tr>
<td>lowercase</td>
<td>Converts all the letters of the text to lowercase characters, the text is not tokenized (normalizer).</td>
</tr>
</tbody>
</table>

These analyzer definitions are based on Apache Lucene. For more information about tokenizers, filters, and CharFilters, see the Apache Lucene documentation.

Additional resources
- Apache Lucene Documentation

3.4.2. Creating custom analyzer definitions

Create custom analyzer definitions and add them to your Data Grid Server installations.
Prerequisites

- Stop Data Grid Server if it is running.
  Data Grid 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 Data Grid Server installation.


`ProgrammaticSearchMappingProvider` example

```java
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 Data Grid 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

```java
package org.infinispan.sample;

import java.time.LocalDate;
import java.util.HashSet;
import java.util.Set;
import org.infinispan.api.annotations.indexing.*;

// Annotate values with @Indexed to add them to indexes
// Annotate each field according to how you want to index it
@Indexed
public class Book {
    @Keyword
    String title;
    @Text
    String description;
    @Keyword
    String isbn;
    @Basic
    LocalDate publicationDate;
    @Embedded
    Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```java
package org.infinispan.sample;

import org.infinispan.api.annotations.indexing.Text;

public class Author {
    @Text
    String name;
```
Procedure

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

```xml
<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.

```java
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 Data Grid cache, as in the following example:

```java
// 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 MAPPINGANNOTATIONS

Add annotations to your Java classes to map your entities to indexes.
Hibernate Search API

Data Grid 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 Data Grid.

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 Data Grid values; in Data Grid 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. Data Grid 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:

```java
@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:

```xml
<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):

```java
ConfigurationBuilder builder = ...
builder.indexing().enable()
   .addKeyTransformer(CustomKey.class, CustomTransformer.class);
```
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 Data Grid caches that are filtered by queries. When entries match the query Data Grid 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.

**NOTE**

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.

Data Grid 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, Data Grid does not send an event.
- If the query on the old value does not match but the new value does, Data Grid sends a Join event.
- If the query on both the old and new values match, Data Grid sends an Update event.
If the query on the old value matches but the new value does not, Data Grid sends a **Leave** event.

If the query on the old value matches and the entry is then deleted or it expires, Data Grid sends a **Leave** event.

### 5.1.1. Continuous queries and Data Grid performance

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

Data Grid 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, the following statement provides results about only a subset of fields that match the criteria rather than the entire entry:

```sql
SELECT field1, field2 FROM Entity WHERE x AND y
```

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:
   - Embedded caches: `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)`

3. Register the query and a **ContinuousQueryListener** object as follows:

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

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

   ```java
   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

```java
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 DATA GRID QUERIES

Data Grid 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

```java
// 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

```bash
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 Data Grid 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.