

Red Hat

Red Hat Data Grid 8.3

Data Grid キャッシュのクエリー

Data Grid キャッシュ内のデータをクエリーする

Red Hat Data Grid 8.3 Data Grid キャッシュのクエリー

Data Grid キャッシュ内のデータをクエリーする

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

Data Grid を使用すると、埋め込みキャッシュとリモートキャッシュを使用してクエリーを実行し、データセットで値を効率的に検索できます。

目次

RED HAT DATA GRID	3
DATA GRID のドキュメント	4
DATA GRID のダウンロード	5
多様性を受け入れるオープンソースの強化	6
第1章 DATA GRID キャッシュのインデックス作成	7
1.1. キャッシュをインデックス化するための DATA GRID の設定	7
1.2. アノテーションのインデックス作成	13
1.3. インデックスの再構築	14
1.4. インデックスが作成されていないクエリー	14
第2章 ICKLE クエリーの作成	15
2.1. ICKLE クエリー	15
2.2. ICKLE クエリー言語構文	17
2.3. フルテキストクエリー	21
第3章 リモートキャッシュのクエリー	23
3.1. HOT ROD クライアントからのキャッシュの作成	23
3.2. DATA GRID コンソールと CLI からのキャッシュのクエリー	26
3.3. リモートキャッシュでアナライザーを使用する	28
第4章 組み込みキャッシュのクエリー	31
4.1. 組み込みキャッシュのクエリー	31
4.2. エンティティーマッピングアノテーション	32
4.3. プログラムでエンティティーをマッピングする	34
第5章 繼続的なクエリーの作成	36
5.1. 繼続的なクエリー	36
5.2. 繼続的なクエリーの作成	37
第6章 DATA GRID クエリーの監視およびチューニング	39
6.1. クエリー統計の取得	39
6.2. クエリーパフォーマンスのチューニング	39

RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

エラスティックスケーリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

データの相互運用性

さまざまなエンドポイントからグリッド内のデータを保存、取得、およびクエリーします。

DATA GRID のドキュメント

Data Grid のドキュメントは、Red Hat カスタマーポータルで入手できます。

- [Data Grid 8.3 ドキュメント](#)
- [Data Grid 8.3 コンポーネントの詳細](#)
- [Data Grid 8.3 でサポートされる設定](#)
- [Data Grid 8 機能のサポート](#)
- [Data Grid で非推奨の機能](#)

DATA GRID のダウンロード

Red Hat カスタマー ポータルで [Data Grid Software Downloads](#) にアクセスします。



注記

Data Grid ソフトウェアにアクセスしてダウンロードするには、Red Hat アカウントが必要です。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#) をご覧ください。

第1章 DATA GRID キャッシュのインデックス作成

Data Grid は、キャッシュに値のインデックスを作成して、クエリーパフォーマンスを向上できます。これにより、インデックスのないクエリーよりも高速な結果が得られます。インデックス作成により、クエリーで全文検索機能を使用することもできます。



注記

Data Grid は、[Apache Lucene](#) テクノロジーを使用して、キャッシュ内の値にインデックスを付けます。

1.1. キャッシュをインデックス化するための DATA GRID の設定

キャッシュ設定でのインデックス作成を有効にし、インデックスの作成時にどのエンティティ Data Grid を含めるかを指定します。

クエリーを使用する場合は、常に Data Grid がキャッシュをインデックス化するように設定してください。インデックス作成により、クエリーのパフォーマンスが大幅に改善されるため、データへの洞察を得ることができます。

手順

1. キャッシュ設定でのインデックス作成を有効にします。

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

ヒント

設定に **indexing** 要素を追加すると、**enabled=true** 属性を含めなくてもインデックスを作成できます。

この要素を追加するリモートキャッシュでは、エンコーディングを ProtoStream として暗黙的に設定します。

2. **indexed-entity** 要素でインデックスを作成するエンティティを指定します。

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

Protobuf メッセージ

- スキーマで宣言されたメッセージを **indexed-entity** 要素の値として指定します。以下に例を示します。

```
<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>
```

この設定は、**book_sample** パッケージ名で、スキーマの **Book** メッセージをインデックス化します。

```
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 オブジェクト

- **@Indexed** アノテーションを含む各クラスの完全修飾名 (FQN) を指定します。

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(Book.class);
```

関連情報

- [org.infinispan.configuration.cache.IndexingConfigurationBuilder](#)

1.1.1. インデックス設定

Data Grid 設定は、インデックスの保存および構築方法を制御します。

1.1.1.1. インデックスストレージ

Data Grid がインデックスを保存する方法を設定できます。

- ホストファイルシステム上。これはデフォルトであり、再起動間でインデックスを保持します。
- JVM ヒープメモリー。これはインデックスが再起動後も存続しないことを意味します。インデックスは、小さなデータセットの場合にのみ、JVM ヒープメモリーに格納する必要があります。

ファイルシステム

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

JVM ヒープメモリー

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

1.1.1.2. インデックスリーダー

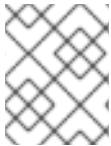
インデックスリーダーは、クエリーを実行するためにインデックスへのアクセスを提供する内部コンポーネントです。インデックスのコンテンツが変更されると、Data Grid はリーダーを更新し、検索結果が最新の状態になるようにする必要があります。インデックスリーダーの更新間隔を設定できます。デフォルトでは、インデックスが最終更新以降に変更された場合、各クエリーの前に Data Grid はインデックスを読み取ります。

```
<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>
```

1.1.1.3. インデックスライター

インデックスライターは、パフォーマンスを改善するために時間の経過とともにマージできる1つ以上のセグメント(サブインデックス)で設定されるインデックスを構築する内部コンポーネントです。インデックスリーダーの操作では、すべてのセグメントを考慮する必要があるため、通常、セグメントが少ないということは、クエリー中のオーバーヘッドが少ないと意味します。

Data Grid は Apache Lucene を内部的に使用し、メモリーとストレージという2つの層でエントリーにインデックスを付けます。新規エントリーは、最初にメモリーインデックスに移動してから、フラッシュが実行されると、設定されたインデックスストレージに移動します。定期的なコミット操作は、フラッシュしたデータからセグメントを作成し、すべてのインデックス変更を永続化します。



注記

index-writer 設定は任意です。デフォルトはほとんどの場合に機能するはずであり、カスタム設定はパフォーマンスを調整するためにのみ使用する必要があります。

```
<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>
```

表1.1 インデックスライター設定属性

属性	説明
commit-interval	メモリーにバッファーリングされたインデックスの変更がインデックスストレージにフラッシュされ、コミットが実行される時間(ミリ秒単位)。操作にはコストがかかるため、小さな値は避けてください。デフォルトは1000ミリ秒(1秒)です。
max-buffered-entries	インデックスストレージにフラッシュされる前に、インメモリーにバッファーリングできるエントリーの最大数。値が大きくなると、インデックスが高速になりますが、より多くのメモリーが使用されます。 ram-buffer-size 属性と組み合わせて使用すると、フラッシュは、最初に発生するイベントに対して発生します。

属性	説明
ram-buffer-size	追加されたエントリーと削除をインデックスストレージにフラッシュする前にバッファーリングするために使用できるメモリーの最大量。値が大きくなると、インデックスが高速になりますが、より多くのメモリーが使用されます。インデックス作成のパフォーマンスを向上させるには、 max-buffered-entries の代わりに、この属性を設定する必要があります。 max-buffered-entries 属性と組み合わせて使用すると、最初に発生したイベントに対してフラッシュが発生します。
thread-pool-size	インデックスへの書き込み操作を実行するスレッドの数。
queue-count	それぞれのインデックス化されたタイプに使用する内部キューの数。各キューは、インデックスに適用される変更のバッチを保持し、キューは並行して処理されます。キューの数を増やすと、インデックスのスループットが増えますが、ボトルネックがCPUである場合に限ります。最適な結果を得るには、 thread-pool-size 値よりも大きい queue-count の値を設定しないでください。
queue-size	各キューが保持できる要素の最大数。 queue-size の値を増やすと、インデックス操作中に使用されるメモリー量が増えます。小さすぎる値を設定すると、インデックス作成操作がブロックされる可能性があります。
low-level-trace	インデックス化操作の低レベルのトレース情報を有効にします。この属性を有効にすると、パフォーマンスが大幅に低下します。この低レベルのトレースは、トラブルシューティングの最後のリソースとしてのみ使用する必要があります。

Data Grid がインデックスセグメントをマージする方法を設定するには、**index-merge** サブ要素を使用します。

表1.2 インデックスのマージ設定属性

属性	説明
----	----

属性	説明
max-entries	インデックスセグメントがマージする前に持つことができるエントリーの最大数。この数を超えるエントリーを持つセグメントはマージされません。値を小さくすると、頻繁に変更されるインデックスでのパフォーマンスが向上します。値を大きくすると、インデックスが頻繁に変更されない場合に検索パフォーマンスが向上します。
factor	一度にマージされるセグメントの数。値が小さいほど、マージが頻繁に発生し、より多くのリソースが使用されますが、セグメントの総数は平均して少くなり、検索パフォーマンスが向上します。より大きな値(10より大きい値)は、大量の書き込みシナリオに最適です。
min-size	バックグラウンドマージのセグメントの最小ターゲットサイズ(MB単位)。このサイズよりも小さなセグメントは積極的にマージされます。値が大きすぎると、頻度は低くなりますが、マージ操作のコストが高くなる可能性があります。
max-size	バックグラウンドマージのセグメントの最大サイズ(MB単位)。このサイズよりも大きなセグメントは、バックグラウンドでマージされることはありません。これを低い値に設定すると、メモリー要件が軽減され、最適な検索速度を犠牲にして、一部のマージ操作が回避されます。インデックスを強制的にマージする場合、この属性は無視され、代わりに max-forced-size が適用されます。
max-forced-size	強制マージのセグメントの最大サイズ(MB単位)で、 max-size 属性をオーバーライドします。これを max-size 以下の同じ値に設定します。ただし、値を低く設定しすぎると、ドキュメントが削除されるため、検索パフォーマンスが低下します。
calibrate-by-deletes	セグメントのエントリーをカウントする際に、インデックスで削除されたエントリーの数が考慮されるかどうか。 false を設定すると、 max-entries によってマージが頻繁に発生しますが、削除されたドキュメントが多いセグメントをより積極的にマージして、クエリーのパフォーマンスを向上させます。

関連情報

- [Data Grid 設定スキーマ参照](#)

1.2. アノテーションのインデックス作成

キャッシュでインデックス作成を有効にする場合は、インデックスを作成するように Data Grid を設定します。また、実際にインデックス化できるように、キャッシュ内のエンティティーの構造化表現で Data Grid を提供する必要があります。

Data Grid インデックスには、エンティティーとフィールドを制御する 2 つのアノテーションがあります。

@Indexed

Data Grid がインデックスをインデックス化するエンティティーまたは Protobuf メッセージタイプを示します。

@Field

Data Grid がインデックスをインデックス化し、かつ以下の属性があるフィールドを示します。

属性	説明	値
index	Data Grid にインデックスのフィールドを含めるかどうかを制御します。	index.YES または Index.NO
store	Data Grid がフィールドをインデックスに格納できるようにして、フィールドをプロジェクトに使用できるようにします。	Store.YES または Store.NO . Store.YES を使用し、ソートに使用する必要があるフィールドには sortable = true を設定します。
analyze	全文検索にフィールドを含めます。	analyze.NO または アナライザの定義を指定します。

リモートキャッシュ

次の 2 つの方法で、リモートキャッシュのインデックスアノテーションを Data Grid に提供できます。

- Java クラスに `@ProtoDoc("@Indexed")` および `@ProtoDoc("@Field(...)")` に直接アノテーションを付けます。
次に、Protobuf スキーマ `.proto` ファイルを生成してから、それらを Data Grid Server にアップロードします。
- Protobuf スキーマに `@Indexed` および `@Field` で直接アノテーションを付けます。
次に、Protobuf スキーマを Data Grid Server にアップロードします。
たとえば、次のスキーマは `@Field` アノテーションを使用します。

```
/*
 * @Field(analyze = Analyze.YES, store = Store.YES, sortable = true)
 */
required string street = 1;
```

`@Field` アノテーションに `store = Store.YES` および `sortable = true` を含めることにより、`street` フィールドを使用して、警告メッセージや予期しない結果に遭遇することなく、クエリーをソートできます。

組み込みキャッシュ

埋め込みキャッシュの場合、Data Grid がエントリーを格納する方法に従って、Java クラスにインデックスアノテーションを追加します。

@**FullTextField** などの他の Hibernate Search アノテーションとともに、@**Indexed** および @**Field** アノテーションを使用します。

1.3. インデックスの再構築

インデックスを再構築すると、キャッシュに保存されているデータから再構築されます。インデックス付きタイプやアナライザーの定義を変更する際にインデックスを再構築する必要があります。同様に、削除後にインデックスを再構築できます。



重要

インデックスの再構築プロセスは、グリッド内のすべてのデータに対して行われるため、完了するまでに長い時間がかかる場合があります。再構築操作の進行中は、クエリーが返す結果も少なくなる可能性があります。

手順

以下のいずれかの方法でインデックスを再構築します。

- **reindexCache()** メソッドを呼び出して、Hot Rod Java クライアントからインデックスをプログラムで再構築します。

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

ヒント

リモートキャッシュの場合は、Data Grid コンソールからインデックスを再構築することもできます。

- **index.run()** メソッドを呼び出して、以下のように埋め込みキャッシュのインデックスを再構築します。

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

1.4. インデックスが作成されていないクエリー

Data Grid は、クエリーのパフォーマンスを最高にするために、キャッシュにインデックスを付けることをお勧めします。ただし、インデックスが作成されていないキャッシュをクエリーすることはできます。

- 埋め込みキャッシュの場合、Plain Old Java Object(POJO) に対してインデックスなしのクエリーを実行できます。
- リモートキャッシュの場合は、**application/x-protostream** メディアタイプで ProtoStream エンコーディングを使用して、インデックスなしのクエリーを実行する必要があります。

第2章 ICKLE クエリーの作成

Data Grid は、リレーションナルクエリーとフルテキストクエリーを作成可能にする Ickle クエリー言語を提供します。

2.1. ICKLE クエリー

API を使用するには、まず QueryFactory をキャッシュに取得してから、`.create()` メソッドを呼び出し、クエリーで使用する文字列を渡します。各 **QueryFactory** インスタンスは **Search** と同じ **Cache** インスタンスにバインドされますが、それ以外の場合は、複数のクエリーを並行して作成するために使用できるステートレスおよびスレッドセーフオブジェクトになります。

たとえば、以下のようになります。

```
// 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();
```



注記

クエリーは常に単一のエンティティータイプをターゲットにし、単一のキャッシュの内容に対して評価されます。複数のキャッシュでクエリーを実行したり、複数のエンティティータイプ(結合)を対象とするクエリーを作成したりすることは、サポートされていません。

クエリーの実行と結果のフェッチは、**Query** オブジェクトの `execute()` メソッドを呼び出すのと同じくらい簡単です。実行後に、同じインスタンスで `execute()` を呼び出すと、クエリーを再実行します。

2.1.1. ページネーション

Query.maxResults(int maxResults) を使用して、返される結果の数を制限することができます。これを **Query.startOffset(long startOffset)** と組み合わせて使用すると、結果セットのページネーションを実現できます。

```
// 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)
```

2.1.2. ヒット数

QueryResult オブジェクトには、ページネーションパラメーターに関係なく、クエリーの結果の合計数を返すための `.hitCount()` メソッドがあります。ヒット数は、パフォーマンス上の理由から、インデックス付きクエリーでのみ使用できます。

2.1.3. 反復

Query オブジェクトには、結果を遅延して取得するための `.iterator()` メソッドがあります。使用後に閉じる必要がある **CloseableIterator** のインスタンスを返します。



注記

リモートクエリーの反復サポートは現在制限されています。反復する前に、最初にすべてのエントリーをクライアントにフェッチするためです。

2.1.4. 名前付きクエリーパラメーター

実行ごとに新しい Query オブジェクトを作成する代わりに、実行前に実際の値に置き換えることができる名前付きパラメーターをクエリーに含めることができます。これにより、クエリーを1度定義し、複数回効率的に実行できます。パラメーターは、Operator の右側でのみ使用でき、通常の定数値ではなく、`org.infinispan.query.dsl.Expression.param(String paramName)` メソッドによって生成されたオブジェクトを Operator に提供することで、クエリーの作成時に定義されます。パラメーターが定義されたら、以下の例に示すように `Query.setParameter(parameterName, value)` または `Query.setParameters(parameterMap)` のいずれかを呼び出すことで設定できます。

```
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();
```

または、実際のパラメーター値のマップを指定して、複数のパラメーターを一度に設定することもできます。

複数の名前付きパラメーターを一度に設定する

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

query.setParameters(parameterMap);
```



注記

クエリーの解析、検証、および実行計画の作業の大部分は、パラメーターでのクエリーの最初の実行時に実行されます。この作業は後続の実行時には繰り返し行われないため、クエリーパラメーターではなく定数値を使用した同様のクエリーの場合よりもパフォーマンスが向上します。

2.1.5. クエリーの実行

Query API は、キャッシュで Ickle クエリーを実行する 2 つの方法を提供します。

- `Query.execute()` は SELECT ステートメントを実行し、結果を返します。
- `Query.executeUpdateStatement()` は DELETE ステートメントを実行し、データを変更します。



注記

常に `executeStatement()` を呼び出してデータを変更し、`execute()` を呼び出してクエリーの結果を取得する必要があります。

関連情報

- [org.infinispan.query.dsl.Query.execute\(\)](#)
- [org.infinispan.query.dsl.Query.executeUpdateStatement\(\)](#)

2.2. ICKLE クエリー言語構文

Ickle クエリー言語は、フルテキスト用のいくつかの拡張機能を含む JPQL クエリー言語のサブセットです。

パーサー構文には、以下のような重要なルールがあります。

- 空白は重要ではありません。
- フィールド名ではワイルドカードはサポートされません。
- デフォルトのフィールドがないため、フィールド名またはパスは必ず指定する必要があります。
- `&&` および `||` は、フルテキストと JPA 述語の両方で、`AND` または `OR` の代わりに使用できます。
- `!` は `NOT` の代わりに使用できます。
- 足りないブール値 Operator は `OR` として解釈されます。
- 文字列の用語は、一重引用符または二重引用符で囲む必要があります。
- ファジー性とブースティングは任意の順序で受け入れられず、常にファジー性が最初になります。
- `<>` の代わりに `!=` が許可されます。
- `>、>=、<、<=` 演算子にはブースティングを適用できません。同じ結果を達成するために範囲を使用することができます。

2.2.1. Operator のフィルタリング

Ickle はインデックス化されたフィールドとインデックス化されていないフィールドの両方に使用できる多くの Operator のフィルタリングをサポートします。

Operator	説明	例
----------	----	---

Operator	説明	例
in	左のオペランドが引数として指定された値のコレクションからの要素のいずれかと等しいことを確認します。	FROM Book WHERE isbn IN ('ZZ', 'X1234')
like	(文字列として想定される) 左側の引数が、JPA ルールに準拠するワイルドカードパターンと一致することを確認します。	FROM Book WHERE title LIKE '%Java%'
=	左側の引数が指定の値と完全に一致することを確認します。	FROM Book WHERE name = 'Programming Java'
!=	左側の引数が指定の値とは異なることを確認します。	FROM Book WHERE language != 'English'
>	左側の引数が指定の値よりも大きいことを確認します。	FROM Book WHERE price > 20
>=	左側の引数が指定の値以上であることを確認します。	FROM Book WHERE price >= 20
<	左側の引数が指定の値未満であることを確認します。	FROM Book WHERE year < 2020
<=	左側の引数が指定の値以下であることを確認します。	FROM Book WHERE price <= 50
between	左側の引数が指定された範囲の制限の間にあることを確認します。	FROM Book WHERE price BETWEEN 50 AND 100

2.2.2. ブール値の条件

以下の例では、複数の属性条件を論理結合 (**and**) および非結合 (**or**) 演算子と組み合わせて、より複雑な条件を作成する方法を示しています。ブール値演算子のよく知られる演算子の優先順位ルールがここで適用されるため、Operator の順序は関連性がありません。ここで、**or** が最初に呼び出された場合でも、**and** Operator の優先順位は **or** よりも高くなります。

```
# 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%'
```

ブール値の否定は論理演算子の中で優先され、次の単純な属性条件にのみ適用されます。

```
# 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'
```

2.2.3. ネストされた条件

論理演算子の優先順位の変更は、括弧を使用して行います。

```
# 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%' )
```

2.2.4. SELECT ステートメントによるプロジェクト

一部のユースケースでは、属性のごく一部のみがアプリケーションによって実際に使用されている場合、特にドメインエンティティーにエンティティーが埋め込まれている場合、ドメインオブジェクト全体を返すのはやり過ぎです。クエリー言語を使用すると、プロジェクトを返す属性（または属性パス）のサブセットを指定できます。展開が使用される場合、**QueryResult.list()** はドメインエンティティー全体を返しませんが、**Object[]** の **List**（プロジェクト化された属性に対応する配列）を返します。

```
# 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%'
```

ソート

1つ以上の属性または属性パスに基づいて結果の順序は **ORDER BY** 句で行われます。複数の並べ替え基準が指定されている場合は、順序によって優先順位が決まります。

```
# 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
```

2.2.5. グループ化およびアグリゲーション

Data Grid には、グループ化フィールドのセットに従ってクエリー結果をグループ化し、各グループに分類される値のセットに集計関数を適用することにより、各グループからの結果の集計を構築する機能があります。グループ化と集計は、プロジェクトクエリー（SELECT 句に1つ以上のフィールドがあるクエリー）にのみ適用できます。

サポートされる集約は **avg**、**sum**、**count**、**max** および **min** です。

グループ化フィールドセットは **GROUP BY** 句で指定し、グループ化フィールドの定義に使用される順番は関係ありません。プロジェクトで選択されたすべてのフィールドは、グループ化フィールドであるか、以下で説明するグループ化関数の1つを使用して集約される必要があります。Projection フィールドは集約され、同時にグループ化に使用できます。グループ化フィールドのみを選択し、集計フィールドは選択しないクエリーは有効です。例：ブックマークは作成者別にグループ化し、それらをカウントします。

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



注記

選択したすべてのフィールドに集計関数が適用され、グループ化にフィールドが使用されないプロジェクトクエリーが許可されます。この場合、集計は、単一のグローバルグループが存在するかのようにグローバルに計算されます。

集約

以下の集約関数をフィールドに適用できます。

表2.1 インデックスのマージ属性

集約関数	説明
avg()	一連の数字の平均を計算します。許可される値は、 java.lang.Number のプリミティブ番号およびインスタンスです。結果は java.lang.Double で表されます。null 以外の値がない場合、結果は代わりに null になります。
count()	null 以外の行の数をカウントし、 java.lang.Long を返します。null 以外の値がない場合、結果は代わりに 0 になります。
max()	見つかった最も大きな値を返します。許可される値は java.lang.Comparable のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに null になります。
min()	見つかった最小値を返します。許可される値は java.lang.Comparable のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに null になります。
sum()	数字のセットの合計を計算します。null 以外の値がない場合、結果は代わりに null になります。以下の表は、指定のフィールドに基づいて返されるタイプを示しています。

表2.2 テーブル合計戻り値のタイプ

フィールドタイプ	戻り値のタイプ
Integral (BigInteger 以外)	Long
Float または Double	double
BigInteger	BigInteger
BigDecimal	BigDecimal

グループ化および集計を使用したクエリーの評価

集計クエリーには、通常のクエリーのようにフィルター条件を含めることができます。フィルタリングは、グループ化操作の前後の2つのステージで実行できます。グループ化操作の実行前に **groupBy()** メソッドを起動する前に定義されたフィルター条件はすべて、キャッシュエントリーに直接(最終的な展開ではなく)キャッシュエントリーに適用されます。これらのフィルター条件は、照会されたエンティティータイプの任意のフィールドを参照でき、グループ化ステージの入力となるデータセットを制限することを目的としています。**groupBy()** メソッドの呼び出し後に定義されたフィルター条件はすべて、展開およびグループ化操作の結果が展開されます。このフィルター条件は、**groupBy()** フィールドまたは集約されたフィールドのいずれかを参照できます。select句で指定されていない集約フィールドを参照することは許可されています。ただし、非集計フィールドと非グループ化フィールドを参照することは禁止されています。このフェーズでフィルタリングすると、プロパティーに基づいてグループの数が減ります。通常のクエリーと同様に、並べ替えも指定できます。順序付け操作は、グループ化操作後に実行され、**groupBy()** フィールドまたは集約されたフィールドのいずれかを参照できます。

2.2.6. DELETE ステートメント

以下の構文を使用して、Data Grid キャッシュからエントリーを削除できます。

DELETE FROM <entityName> [WHERE condition]

- <**entityName**> で単一のエンティティのみを参照します。DELETE クエリーは結合を使用できません。
- WHERE 条件は任意です。

DELETE クエリーでは、次のいずれも使用できません。

- SELECT ステートメントによるプロジェクション
- グループ化およびアグリゲーション
- ORDER BY 句

ヒント

Query.executeStatement() メソッドを呼び出して、DELETE ステートメントを実行します。

関連情報

- [org.infinispan.query.dsl.Query.executeStatement\(\)](#)

2.3. フルテキストクエリー

Ickle クエリー言語を使用して、フルテキスト検索を実行できます。

2.3.1. Fuzzy クエリー

ファジークエリー add ~ を整数とともに実行するには、用語の後に使用される用語からの距離を表します。For instance

FROM sample_bank_account.Transaction WHERE description : 'cofee'~2

2.3.2. 範囲のクエリー

以下の例に示すように、範囲クエリーを実行するには、中括弧のペア内で指定の境界を定義します。

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

2.3.3. フレーズクエリー

次の例に示すように、単語のグループは引用符で囲むことで検索できます。

```
FROM sample_bank_account.Transaction WHERE description : 'bus fare'
```

2.3.4. 近接クエリー

特定の距離内で 2 つの用語を検索して近接クエリーを実行するには、フレーズの後に距離とともに ~ を追加します。たとえば、以下の例では、キャンセルと fee という単語が 3 個以上ありません。

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

2.3.5. ワイルドカードクエリー

"text" または "test" を検索するには、单一文字のワイルドカード検索 ? を使用します。

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

"test"、"tests"、"tester" を検索するには、マルチ文字のワイルドカード検索 * を使用します。

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

2.3.6. 正規表現のクエリー

正規表現クエリーは、/ の間のパターンを指定することで実行できます。Ickle は Lucene の正規表現構文を使用しているため、単語 **moat** または **boat** を検索するには、以下を使用できます。

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

2.3.7. クエリーのブースト

用語は、指定のクエリーにおける耐障害性を高めるために ^ を追加し、条件を強化できます。たとえば、ビールとビールとの関連性が 3 倍高いビールとワインを含むタイトルを検索するには、次のように使用できます。

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

第3章 リモートキャッシュのクエリー

Data Grid Server のリモートキャッシュにインデックスを付けてクエリーを実行できます。

3.1. HOT ROD クライアントからのキャッシュの作成

Data Grid を使用すると、HotRod エンドポイントを介して Java クライアントからリモートキャッシュをプログラムでクエリーできます。この手順では、**Book** インスタンスを保存するリモートキャッシュをインデックス化する方法を説明します。

前提条件

- ProtoStream プロセッサーを **pom.xml** に追加します。

Data Grid は、**@ProtoField** および **@ProtoDoc** アノテーションにこのプロセッサーを提供するため、Protobuf スキーマを生成してクエリーを実行できます。

```
<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>
```

手順

- 次の例のように、インデックスアノテーションをクラスに追加します。

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. 新しいクラスに **SerializationContextInitializer** インターフェイスを実装してから、@ **AutoProtoSchemaBuilder** アノテーションを追加します。
 - a. **includeClasses** パラメーターで **@ProtoField** と **@ProtoDoc** アノテーションを含んだクラスを参照。
 - b. 生成する Protobuf スキーマの名前と、**schemaFileName** および **schemaFilePath** パラメーターを使用したファイルシステムパスを定義します。
 - c. **schemaPackageName** パラメーターで Protobuf スキーマのパッケージ名を指定します。

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. プロジェクトをコンパイルします。
このプロシージャのコード例は、proto /book.proto スキーマとアノテーションされた Book クラスの RemoteQueryInitializerImpl.java 実装を生成します。

次のステップ

エンティティーにインデックスを付けるように Data Grid を設定するリモートキャッシュを作成します。たとえば、以下のリモートキャッシュは、前のステップで生成した book.proto スキーマの Book エンティティーをインデックス化します。

```

<replicated-cache name="books">
  <indexing>
    <indexed-entities>

```

```

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

```

以下の **RemoteQuery** クラスは以下を行います。

- **RemoteQueryInitializerImpl** シリアル化コンテキストを Hot Rod Java クライアントに登録します。
- Protobuf スキーマ **book.proto** を Data Grid Server に登録します。
- 2つの **Book** インスタンスをリモートキャッシングに追加します。
- タイトルのキーワードで本を照合する全文クエリーを実行します。

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!
}
}

```

関連情報

- シリアル化コンテキストの作成と Protobuf スキーマの登録の詳細は、[データのマーシャリングとエンコード](#)。
- [@ProtoField](#)、[@ProtoDoc](#)、および [@AutoProtoSchemaBuilder](#) アノテーションの詳細については、[ProtoStream アノテーション](#) を参照してください。

3.2. DATA GRID コンソールと CLI からのキャッシュのクエリー

Data Grid コンソールと Data Grid コマンドラインインターフェイス (CLI) を使用すると、インデックス付きおよびインデックスなしのリモートキャッシングをクエリーできます。また、任意の HTTP クライアントを使用して、RESTAPI を介してキャッシングにインデックスを付けてクエリーを実行することもできます。

この手順では、**Person** インスタンスを保存するリモートキャッシングをインデックス化してクエリーする方法を説明します。

前提条件

- 稼働中の Data Grid Server インスタンスが1つ以上ある。
- 作成権限を持つ Data Grid クレデンシャルを持っている。

手順

- 以下の例のように、インデックスアノテーションを Protobuf スキーマに追加します。

```

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;

}

```

Data Grid CLI で、以下のように **--upload=** 引数を指定して **schema** コマンドを使用します。

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

- ProtoStream エンコーディングを使用する **people** という名前のキャッシングを作成し、Data Grid が Protobuf スキーマで宣言されたエンティティーに設定します。
以下のキャッシングは、直前の手順で **Person** エンティティーをインデックス化します。

```

<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>

```

CLI で、以下のように **--file=** 引数を指定して **create cache** コマンドを使用します。

```
create cache --file=people.xml people
```

- キャッシングにエントリーを追加します。

リモートキャッシングをクエリーするには、一部のデータが含まれている必要があります。以下の手順例では、以下の JSON 値を使用するエントリーを作成します。

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
}
```

CLI で、以下のように **put** コマンドを使用して、**--file=** 引数を指定して各エントリーを追加します。

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

ヒント

Data Grid Console から、カスタムタイプを使用して JSON 形式で値を追加する際に、**Value content type** フィールドの **Custom Type** を選択する必要があります。

- リモートキャッシュをクエリーします。

CLI で、リモートキャッシュのコンテキストで **query** コマンドを使用します。

```
query "from org.infinispan.example.Person p WHERE p.name='Person' ORDER BY p.age ASC"
```

クエリーは昇順で **Person** と一致する名前を持つすべてのエントリーを返します。

関連情報

- [Data Grid REST API](#)

3.3. リモートキャッシュでアナライザーを使用する

アナライザーは、入力データを、索引付けおよび照会できる用語に変換します。アナライザーの定義は Java クラスの **@Field** アノテーションを使用するか、Protobuf スキーマに直接指定します。

手順

- Analyze.YES** 属性を追加して、プロパティーが分析されていることを示します。
- @Analyzer** アノテーションで アナライザ定義を指定します。

Protobuf スキーマ

```
/* @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 クラス

```
@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. デフォルトのアナライザー定義

Data Grid は、デフォルトのアナライザー定義のセットを提供します。

定義	説明
standard	テキストフィールドをトークンに分割し、空白と句読点を区切り文字として扱います。
simple	非文字で区切り、すべての文字を小文字に変換することにより、入力ストリームをトークン化します。空白と非文字は破棄されます。
whitespace	テキストストリームを空白で分割し、空白以外の文字のシーケンスをトークンとして返します。
キーワード	テキストフィールド全体を單一トークンとして扱います。
stemmer	SnowballPorter フィルターを使用して英語の単語を語幹にします。
ngram	デフォルトでサイズ 3 つのグラムである n-gram トークンを生成します。
filename	テキストフィールドを standard アナライザーよりも大きなサイズトークンに分割し、空白文字を区切り文字として扱い、すべての文字を小文字に変換します。

これらのアナライザー定義は Apache Lucene をベースとし、as-is で提供されます。tokenizers、filters、および CharFilters に関する詳細は、適切な Lucene のドキュメントを参照してください。

3.3.2. カスタムアナライザー定義の作成

カスタムアナライザー定義を作成し、それらを Data Grid Server インストールに追加します。

前提条件

- Data Grid Server が実行している場合は停止します。
Data Grid Server は、起動時にクラスのみを読み込みます。

手順

1. **ProgrammaticSearchMappingProvider** API を実装します。
2. 次のファイルの完全修飾クラス (FQN) を使用して、実装を JAR にパッケージ化します。
META-INF/services/org.infinispan.query.spi.ProgrammaticSearchMappingProvider
3. JAR ファイルを Data Grid Server インストールの **server/lib** ディレクトリーにコピーします。
4. Data Grid Server を起動します。

ProgrammaticSearchMappingProvider の例

```
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);
    }
}
```

第4章 組み込みキャッシュのクエリー

データグリッドをライブラリーとしてカスタムアプリケーションに追加する場合は、埋め込みクエリーを使用します。

埋め込みクエリーでは、Protobuf マッピングは必要ありません。インデックス作成とクエリーは、どちらも Java オブジェクト上で実行されます。

4.1. 組み込みキャッシュのクエリー

本セクションでは、インデックス化された **Book** インスタンスを保存する books という名前のサンプルキャッシュを使用して埋め込みキャッシュをクエリーする方法を説明します。

この例では、各 **Book** インスタンスはインデックス化されるプロパティを定義し、以下のように Hibernate Search アノテーションを使用して詳細なインデックスオプションを指定します。

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;
}

```

手順

- books キャッシュをインデックス化するように Data Grid を設定し、**org.infinispan.sample.Book** をインデックスのエンティティーとして指定します。

```

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

```

- キャッシュを取得します。

```

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");

```

- 以下の例のように、Data Grid キャッシュに保存されている **Book** インスタンスでフィールドのクエリーを実行します。

```

// 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. エンティティーマッピングアノテーション

Java クラスにアノテーションを追加して、エンティティーをインデックスにマップします。

Hibernate Search API

Data Grid は [Hibernate Search API](#) を使用して、エンティティーレベルでインデックス作成の詳細な設定を定義します。この設定には、アノテーションが付けられたフィールド、使用するアナライザー、ネストされたオブジェクトのマッピング方法などが含まれます。

以下のセクションでは、Data Grid で使用するエンティティーマッピングアノテーションに適用される情報を提供します。

これらのアノテーションの詳細については、[the Hibernate Search manual](#) を参照してください。

@DocumentId

Hibernate Search とは異なり、**@DocumentId** を使用してフィールドを識別子としてマーク付けすると、Data Grid は値を保存するために使用されるキーになります。すべての **@Indexed** オブジェクトの識別子は、値を保存するために使用されるキーになります。**@Transformable**、カスタム型、およびカスタム **FieldBridge** 実装の組み合わせを使用して、キーのインデックス化方法をカスタマイズできます。

@Transformable keys

各値のキーはインデックス化する必要があり、キーインスタンスを **String** で変換する必要があります。Data Grid には、共通のプリミティブをエンコードするためのデフォルトの変換ルーチンが含まれていますが、カスタムキーを使用するには **org.infinispan.query.Transformer** の実装を提供する必要があります。

アノテーションを使用したキートransformerの登録

キークラスに **org.infinispan.query.Transformer** のアノテーションを付け、カスタムトランスマッパー実装が自動的に選択されます。

```
@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 ...
    }
}
```

キャッシュインデックス設定を介したキートransformerの登録

埋め込みおよびサーバー設定の両方で、**key-transformers** xml 要素を使用します。

```
<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>

```

または、Java 設定 API(組み込みモード)を使用します。

```

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

```

4.3. プログラムでエンティティーをマッピングする

プログラムを用いて、Java クラスにアノテーションを付ける代わりに、エンティティーをインデックスにマップできます。

次の例では、グリッドに格納され、2つのプロパティで検索可能にするオブジェクト **Author** をマップします。

```

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;
```

第5章 継続的なクエリーの作成

アプリケーションはリスナーを登録して、クエリーフィルターに一致するキャッシュエントリーに関する継続的な更新を受け取ることができます。

5.1. 継続的なクエリー

継続的なクエリーは、クエリーでフィルターされる Data Grid キャッシュのデータに関するリアルタイムの通知をアプリケーションに提供します。エントリーがクエリー Data Grid と一致する場合は、更新されたデータを任意のリスナーに送信します。これは、クエリーを実行するアプリケーションではなく、イベントのストリームを提供します。

継続的なクエリーは、セットに参加した値についてアプリケーションに通知します。一致する値に対して一致し、変更して一致し続けた値については、そのセットを残す値について通知できます。

たとえば、継続的なクエリーはアプリケーションにすべてについて通知できます。

- 18~25 歳の人で、**Person** エンティティーに **age** プロパティーがあり、ユーザーアプリケーションによるエンティティー更新を想定する場合。
- 2000 ドルを超える金額のトランザクション。
- キャッシュにラップエントリーが含まれ、レース中にラップが入力されたと仮定して、F1 レーサーのラップ速度が 1:45.00 秒未満であった時間。



注記

継続的なクエリーは、グループ化、集約、およびソート操作以外のすべてのクエリー機能を使用できます。

継続的なクエリーの仕組み

継続的なクエリーは、以下のイベントでクライアントリスナーに通知します。

Join

キャッシュエントリーがクエリーと一致します。

更新

クエリーに一致するキャッシュエントリーが更新され、引き続きクエリーに一致します。

Leave

キャッシュエントリーがクエリーと一致しなくなりました。

クライアントが継続的なクエリーリスナーを登録すると、クエリーに一致するエントリーの **Join** イベントをすぐに受信します。クライアントリスナーは、キャッシュ操作がクエリーに一致するエントリーを変更するたびに、後続のイベントを受け取ります。

Data Grid は、以下のように **Join**、**Update**、または **Leave** イベントをクライアントリスナーに送信するタイミングを決定します。

- 古い値と新しい値のクエリーが一致しない場合、Data Grid はイベントを送信しません。
- 古い値のクエリーが一致しそう、新しい値を指定すると、Data Grid は **Join** イベントを送信します。

- 古い値と新しい値の両方のクエリーが一致する場合、Data Grid は **Update** イベントを送信します。
- 古い値のクエリーが一致しても、新しい値がない場合、Data Grid は **Leave** イベントを送信します。
- 古い値のクエリーが一致し、エントリーが削除されるか、期限切れになると、Data Grid は **Leave** イベントを送信します。

5.1.1. 継続クエリーと Data Grid のパフォーマンス

継続的なクエリーは、アプリケーションに更新の定数ストリームを提供しており、大量のイベントを生成できます。Data Grid は、生成する各イベントにメモリーを一時的に割り当てます。これにより、メモリー不足が発生し、特にリモートキャッシングに対して **OutOfMemoryError** 例外が発生する可能性があります。このため、パフォーマンスへの影響を回避するために、継続的なクエリーを慎重に設計する必要があります。

Data Grid は、継続的なクエリーのスコープを必要な情報の最小量に制限することを強く推奨します。これを実行するには、プロジェクトおよび述語を使用できます。たとえば、以下のステートメントでは、エントリー全体ではなく基準に一致するフィールドのサブセットのみに関する結果を表示します。

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

また、各 **ContinuousQueryListener** は、ブロッキングスレッドを使用せずに受信したすべてのイベントを迅速に処理できるようにすることが重要です。これを実行するには、イベントを必要に生成するキャッシング操作を回避する必要があります。

5.2. 継続的なクエリーの作成

リモートおよび組み込みキャッシングの継続的なクエリーを作成できます。

手順

1. **Query** オブジェクトを作成します。
2. 適切なメソッドを呼び出して、キャッシングの **ContinuousQuery** オブジェクトを取得します。
 - リモートキャッシング: `org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)`
 - 埋め込みキャッシング: `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)`
3. 以下のようにクエリーと **ContinuousQueryListener** オブジェクトを登録します。

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

4. 継続的なクエリーが必要なくなった場合は、以下のようにリスナーを削除します。

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

継続的なクエリーの例

以下のコード例は、組み込みキャッシングを使用した単純な継続的なクエリーを示しています。

この例では、年齢が 21 歳未満の **Person** インスタンスがキャッシュに追加されると、リスナーは通知を受け取ります。これらの **Person** インスタンスも matches マップに追加されます。エントリーがキャッシュから削除されるか、その年齢が 21 以上になると、matches マップから削除されます。

継続的クエリーの登録

```

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);

```

第6章 DATA GRID クエリーの監視およびチューニング

Data Grid はクエリーの統計を公開し、クエリーのパフォーマンスを改善するために調整できる属性を提供します。

6.1. クエリー統計の取得

インデックスのタイプやクエリーの完了の平均時間などの情報など、インデックスおよびクエリーのパフォーマンスに関する情報を収集する統計を収集します。

手順

次のいずれかを行います。

- 埋め込みキャッシュの `getSearchStatistics()` メソッドまたは `getClusteredSearchStatistics()` メソッドを呼び出します。
- GET** リクエストを使用して、REST API からリモートキャッシュの統計を取得します。

組み込みキャッシュ

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

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

リモートキャッシュ

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

6.2. クエリーパフォーマンスのチューニング

次のガイドラインを使用して、インデックス作成操作とクエリーのパフォーマンスを向上させます。

インデックスの使用状況の統計値の確認

部分的にインデックス付けされたキャッシュに対するクエリーは、より遅い結果を返します。たとえば、スキーマの一部のフィールドにアノテーションが付けられていない場合、生成されるインデックスにはこれらのフィールドが含まれません。

クエリーの各タイプの実行にかかる時間を確認して、クエリーのパフォーマンスのチューニングを開始します。クエリーが遅いと思われる場合は、クエリーがキャッシュのインデックスを使用していること、およびすべてのエンティティーとフィールドマッピングにインデックスが付けられていることを確認する必要があります。

インデックスのコミット間隔の調整

インデックス処理は、Data Grid クラスターの書き込みスループットを低下させる可能性があります。`commit-interval` 属性は、メモリーにバッファーリングされたインデックスの変更がインデックスストレージにフラッシュされ、コミットが実行される間隔をミリ秒単位で定義します。

この操作にはコストがかかるため、小さすぎる間隔の設定は避けてください。デフォルトは 1000 ミリ秒 (1 秒) です。

クエリーの更新間隔の調整

refresh-interval 属性は、インデックスリーダーの更新の間隔 (ミリ秒単位) を定義します。

デフォルト値は **0** で、キャッシュに書き込まれるとすぐにクエリーのデータを返します。

値が **0** より大きいと、クエリー結果が古くなりますが、特に書き込みが多いシナリオでは、スループットが大幅に向上します。書き込まれた直後にクエリーで返されるデータが必要ない場合は、更新間隔を調整してクエリーのパフォーマンスを向上させる必要があります。