Getting Started With Ehcache Search API For Clustered Caches

Ehcache Search is a powerful search API for querying clustered caches in a Terracotta cluster. Designed to be easy to integrate with existing projects, the Ehcache Search API can be implemented with configuration or programmatically. The following is an example snipped from an Ehcache configuration file:

```
<cache name="myCache" maxElementsInMemory="0" eternal="true"
  overflowToDisk="false">
  <searchable>
    <searchAttribute name="age" />
    <searchAttribute name="first_name" expression="value.getFirstName()" />
    <searchAttribute name="last_name" expression="value.getLastName()" />
    <searchAttribute name="zip_code" expression="value.getZipCode()" />
  </searchable>
  <terracotta />
</cache>
```

Let's assume there is a Person class that serves as the value for elements in myCache. Note that each `expression` attribute in `searchAttribute` is set to use an accessor method on the cache element's value, with the exception of "age", which is given bean-style. The Person class must have accessor methods to match the configured expressions. In addition, let's assume that there is code that populates the cache. Here, then, is an example of search code based on this assumptions:

```
        // After CacheManager and Cache created, create a query for myCache:
        Query query = myCache.createQuery();

        // Create the Attribute objects.
        Attribute<String> last_name = myCache.getSearchAttribute("last_name");
        Attribute<Integer> zip_code = myCache.getSearchAttribute("zip_code");
        Attribute<Integer> age = myCache.getSearchAttribute("age");

        // Specify the type of content for the result set. Executing the query without specifying desired
results returns no results even if
        // Executing the query without specifying desired results returns no results even if there are hits.
        query.includeKeys(); // Return the keys for values that are hits.

        // Define the search criteria. This one uses Criteria.and() to set criteria to find
        // adults with the last name "Marley" whose address has the zip code "94102".
        query.addCriteria(last_name.eq("Marley").and(zip_code.eq(94102)));

        // Execute the query, putting the result set (keys to element that meet the search criteria) in Results
object.
        Results results = query.execute();

        // Find the number of results -- the number of hits.
        int size = results.size();

        // Discard the results when done to free up cache resources.

        results.discard();

        // Using an aggregator in a query to get an average age of adults:
        Query averageAgeOfAdultsQuery = myCache.createQuery();
        averageAgeOfAdultsQuery.addCriteria(age.ge(18));
        averageAgeOfAdultsQuery.includeAggregator(age.average());
        Results averageAgeOfAdults = averageAgeOfAdultsQuery.execute();
        List aggregateResults = averageAgeOfAdults.getAggregatorResults();
        double averageAge = (Double) aggregateResults.get(0);
```

The following shows how to programmatically create the cache configuration, with search attributes.

```
        Configuration cacheManagerConfig = new Configuration();
        cacheManagerConfig.addDefaultCache(new CacheConfiguration()); // A default cache must be present.
        CacheConfiguration cacheConfig = new CacheConfiguration("myCache", 0).eternal(true);
        Searchable searchable = new Searchable();
        cacheConfig.addSearchable(searchable);

        // Create attributes to use in queries.
        searchable.addSearchAttribute(new SearchAttribute().name("age"));

        // Use an expression for accessing values.
        searchable.addSearchAttribute(new SearchAttribute().name("first_name").expression("value.
getFirstName()"));
        searchable.addSearchAttribute(new SearchAttribute().name("last_name").expression("value.
getLastName()"));
        searchable.addSearchAttribute(new SearchAttribute().name("zip_code").expression("value.getZipCode()"));

        cacheManager = new CacheManager(cacheManagerConfig);
        cacheManager.addCache(new Cache(cacheConfig));

        Ehcache myCache = cacheManager.getEhcache("myCache");

        // Now create the attributes and queries, then execute.
```

To learn more about the Ehcache Search API, see the net.sf.ehcache.search* packages in this Javadoc.

Stored Search Indexes

Searches occur on indexes held by the Terracotta server. By default, index files are stored in `/index` under the server's data directory. However, you can specify a different path using the <index> element:

```
...
<server>
  <data>%(user.home)/terracotta/server-data</data>
  <index>%(user.home)/terracotta/index</index>
  <logs>%(user.home)/terracotta/server-logs</logs>
  <statistics>%(user.home)/terracotta/server-statistics</statistics>
...
</server>
...
```

To enhance performance, it is recommended that you store server data and search indexes on different disks.

Troubleshooting Ehcache Search

This section contains information on avoiding potential issues with using the Ehcache Search API.

Large Result Sets

If your query returns a result set containing a very large amount of data, performance degradation or errors (such as OOME) could occur due to network and resource limitations. You can manage the size of result sets by following these best practices:

- Limit the size of the results set with `Query.maxResults(int number_of_hits)`.
- Page the results set using Results.range(int start_index, int number_of_hits).
- Use a built-in Aggregator function to return a summary statistic (see the net.sf.ehcache.search.aggregator package in this Javadoc.

Case Sensitivity of Attribute Criteria

`Attribute.like()` is case sensitive.

Inconsistent Data

Ehcache Search guarantees that all local changes made before a query is executed are available to search results. However, to maintain a high level of performance, cluster locks are not checked. This "eventual" synchronization of clustered caches and search indexes means that remote changes may not be available to queries until the changes are applied cluster wide (or, for transactional caches, `commit()` is called). Thus it is possible to get results that include removed elements, inconsistent data from the same query executed at different times, and calculated values that are no longer accurate (such those returned by aggregators).

You can take certain precautions to prevent these types of problems. For example, if your search uses aggregators, add all aggregators to the same query to get consistent data. If your code attempts to get values using keys returned by a query, use null guards.

Failures

Failure to return is handled by a CacheException. If an exception occurs when the value extractor (either expression-based or custom) executes, the affected attribute value is omitted from the search index.