vmware® vFabric GemFire

LUCENE-GEMFIRE
INDEX SYSTEM INTEGRATION

vFabric Field Center of Excellence
VMware, Inc.

3401 Hillview Ave
Palo Alto, CA 94304 USA
Tel: 1-877-486-9273
Fax: 650-427-5001

ABSTRACT

GemFire is a scalable compute and caching solution used to perform analytics, event driven compute, distributed data storage and various other solutions. This paper presents the Lucene indexing platform as an alternative indexing solution for GemFire.

The project integrates GemFire and Lucene Index platform, using Spring-Data, to provide a flexible parallel fast search engine. By combining the two independent products we can leverage each product to its fullest capability. The end result provides users with an elastic search technology with in memory data speeds of a distributed cache platform with high availability.

If you are interested in learning more about how to take advantage of all of what GemFire offers please contact your VMware representative and we can schedule a one on one architecture session.

INTRODUCTION

The motivation of the project was to provide an alternative search capability for GemFire while providing users a natural method to define searchable domain object attributes. Performance was also a key driver to ensure constant search performance irrespective of scale. The solution provides a baseline approach for developer to build upon.

Goals

- Advanced search capability using established search technology.
- Deterministic search performance irrespective of scale.
- Simple API to allow developers to define searchable attributes on domain classes.
- Abstract complexity to clean simple interfaces.
- Fault tolerant search process.

Various business use cases can take advantage of this integration to improve search use cases either by retrieval time, criteria flexibility and high availability search indexes.

Example Business Cases

- Fast online trading symbol or company lookups to narrow search results to smaller set of companies, similar to Google Finance, for user to select desired symbol.
- Provide the ability to search through large document stores to retrieve content that is relevant to the user. For example provide searchable legal document stores to retrieve similar case notes using document abstracts as the searchable material.

- Provide location based business search services to find local businesses relative to own location. For example find local dry clearers for traveling sales person.

ARCHITECTURE

The architecture builds Lucene search capability within GemFire cache data nodes. GemFire is a distributed caching platform and by embedding the Lucene search engine within each cache node JVM, the result is a distributed powerful searchable store with all the functional benefits of GemFire.

Each GemFire cluster node manages its own independent set of Lucene region indexes within the same JVM process. This provides search isolation and elasticity across the GemFire cluster.

The figure 1 presents a high level view of a single GemFire cache node. Each cache node has one or many data regions holding key/value objects. Each region will have its own Lucene index repository. The index can be set to be highly available, using a configuration described later in this paper.

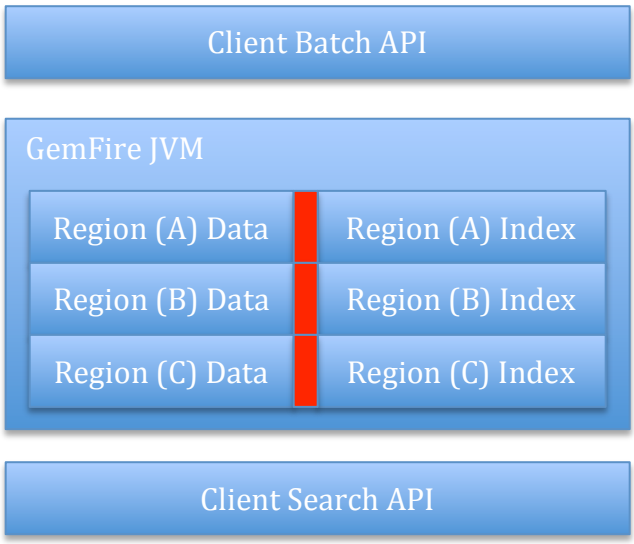

Figure 1 - Architecture

Features

- All user data is stored in GemFire.
- Each region has its own index repository.
- Spring Index repositories contain Lucene search documents.
- Index repositories and data reside within same JVM process.
- Index repositories can be configured to be highly available.
- User domain objects are decorated with @Searchable annotation.

- Client interfaces provided for searching and batch operations.

REGION INDEX MECHANISM

At the GemFire data region level a Lucene index is applied using a Spring configuration. The configuration creates a dedicated index repository for the specified region. Along with this setting a cache listener, *LuceneIndexWriter*, is configured to capture all region events to manage the index repository.

For example on a region update the *CacheListener* (see GemFire Developer's Guide for details on cache event processing) receives an event and updates the index repository synchronously, this process is suitable if data updates are infrequent. Figure 2 represents this structure where the red intersection is the *LuceneIndexWriter* cache listener. The index itself is not held in a region but rather in an instance of a Spring-Data specialized repository.

Figure 2 - Data region

For high volume data loads a GemFire function is used to batch updates into manageable Lucene index transactions to reduce update latency. This is the preferred method when the GemFire cluster is being primed or for intra-day batch update processing. This technique can be applied for high frequency updates via an asynchronous process too.

An important point to mention is index operations are performed synchronously and occur after the cache insert. This can be changed by switching to a *CacheWriter* where the index update will occur before the cache update, not implemented. This implementation strategy is left to the user to decide upon at design time.

CONFIGURATION

This section outlines the configuration details required for Lucene region indexing.

To get started a *LuceneFactory* singleton object is required to manage all the local cache JVM region indexes. Figure 3 provides the configuration detail.

```
<bean name="luceneFactory" class="com.vmware.demo.sgf.lucene.LuceneIndexFactory">
        <property name="indexDirectory" value="${index.dir}"/>
</bean>
```

Figure 3 – LuceneFactory configuration

Figure 4 presents the configuration to setup a GemFire partition region with zero redundancy, using Spring-GemFire, to use the Lucene Index system.

```
<gfe:partitioned-region id="Instrument" copies="0">
     <gfe:cache-listener>
       <bean class="com.vmware.demo.sgf.lucene.cache.LuceneIndexWriter">
              <property name="repo" ref="luceneFactory"></property>
       </bean>
     </gfe:cache-listener>
</gfe:partitioned-region>
```

Figure 4 - Region Index configuration

This configuration is provides a simple non-HA set up useful to test the system functionality.

INDEX HIGH AVAILABILITY

Each region index can be set to be highly available. In the event of a failure, an index is rebuilt within the redundant cache node, providing a region has been set to have a redundancy. For partitioned regions a GemFire *PartitionListener* implementation is used to determine if a node has become primary and would then build the index based on its own local data set.

```
<gfe:partition-listener>
    <bean class="com.vmware.demo.sgf.lucene.cache.PartitionRegionLuceneIndexBuilder"/>
</gfe:partition-listener>
```

Figure 5 - High Availability

Figure 5 presents the high availability configuration required for partitioned regions in red, replicated regions do not require this configuration as they are already redundant.

COMPLETE CONFIGURATION

The resulting configuration for a partitioned region will setup the index writer and ensure the region has a copy of itself elsewhere in the cluster. The redundancy recovery-delay is set to zero to ensure redundancy is recovered immediately in the event of a node failure.

```
<gfe:partitioned-region id="Instrument" copies="1" recovery-delay="0">
    <gfe:cache-listener>
        <bean class="com.vmware.demo.sgf.lucene.cache.LuceneIndexWriter">
          <property name="repo" ref="luceneFactory"></property>
        </bean>
    </gfe:cache-listener>

    <gfe:partition-listener>
        <bean class="com.vmware.demo.sgf.lucene.cache.PartitionRegionLuceneIndexBuilder"/>
    </gfe:partition-listener>
</gfe:partitioned-region>
```

Figure 6 - Final region configuration

For replicated regions the partition listener is not required as the region already is redundant, i.e. all region data exists in all nodes where the region is defined.

CLIENT API'S

This section discusses the various client API's used to insert searchable objects into the cache and perform searches against the cluster.

SEARCHABLE OBJECT

GemFire is a Key/Value store, similar to a distributed HashMap. Data is inserted in to a region as a key/value entry. Using the value object we place searchable annotations on attributes we want to perform searches upon. This makes sense as value objects generally contain more data rather than the key. From the example below, figure 7, we can see how annotations are applied to the value class attributes.

```
public class Instrument implements DataSerializable {

        @Searchable
        String symbol;

        @Searchable
        String description;

        . . .
}
```

Figure 2 - Searchable annotations on value object.

The key is used to perform direct value retrieval on a search hit. This is achieved by adding the key, in serialized form, to the Lucene index document. By extracting out the searchable attributes and linking them to the target key using a Lucene index document provides the core integration functionality.

SEARCHING

Searching for data is performed using the *ClientClusterSearch* API, see figure 8 below. In summary the API provides two search methods, a streaming search method useful for large search results and a search count method.

```
/**
* Perform a cluster search for the given field and searchText items on a region.
* @param field        - Field to perform search upon.
* @param searchText   - Array list of search text items.
* @param regionName   - Actual region path.
* @param maxItems     - Maximum number of items to return.
* @return             - List of found elements.
*/
List search(String field, String[] searchText, String regionName)

/**
* Perform a cluster search for the given field and searchText items on a region.
* @param field        - Field to perform search upon.
* @param searchText   - Array list of search text items.
* @param regionName   - Actual region path.
* @param maxItems     - Maximum number of items to return.
* @return             - List of found elements.
*/
List search(String field, String[] searchText, String regionName, int maxItems)

/**
* Stream search results as they are found.
* @param field        - Field to perform search upon.
* @param searchText   - Array list of search text items.
* @param regionName   - Actual region path.
*/
void streamSearch(String field, String[] searchText, String regionName,
                  final StreamResultCallback callback )

/**
* Get a count of the number of found elements.
* @param field        - Field to perform search upon.
* @param searchText   - Array list of search text items.
* @param regionName   - Actual region path.
* @return             - Count of found elements.
*/
long searchCount(String field, String[] searchText, String regionName)
```

Figure 8 - ClientClusterSearch API

The methods allow for multiple search strings to be passed for a given value field attribute, similar to an OR condition. The search strings are not restricted to plain text but take full advantage of the Lucene *QueryParser* syntax.

BATCH PROCESSING

The *ClientBatchOperation* API provides users the ability to batch load, update and delete items from the cache and index via a simple interface. These methods are useful to reduce overall operation latency for large datasets. Priming the cache on startup or intra-day batch operations are ideal for these methods.

```
/**
* Load a set of data in to GemFire using a batch process.
* @param map             - Key/Value pairs to load.
* @param regionName      - Target region path.
* @return                - True is operation was successful false otherwise.
*/
boolean batchLoad(final Map<? extends Object,? extends Object> map, final String regionName)

/**
* Update a set of data in to GemFire using a batch process.
* @param map             - Key/Value pairs to load.
* @param regionName      - Target region path.
* @return                - True is operation was successful false otherwise.
*/
boolean batchUpdate(final Map<? extends Object,? extends Object> map, final String regionName)

/**
* Delete a set of data in to GemFire using a batch process.
* @param map             - Key/Value pairs to load.
* @param regionName      - Target region path.
* @return                - True is operation was successful false otherwise.
*/
boolean batchDelete(final Map<? extends Object,? extends Object> map, final String regionName)
```

Figure 3 - ClientBatchOperation API

TECHNOLOGIES

Below describes the various technologies used to implement the solution.

GemFire

GemFire is a distributed data management platform providing dynamic scalability, high performance, and database-like persistence. It blends advanced techniques like replication, partitioning, data-aware routing, and continuous querying.

Lucene

Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. The current implementation is built from the v4.0 platform.

Spring

Two Spring projects were used to integrate GemFire and Lucene together.

Spring-Data

Spring Data makes it easier to build Spring-powered applications that use new data access technologies such as non-relational databases, map-reduce frameworks etc.

Spring-GemFire

Project to simplify the development of building highly scalable applications using GemFire as a distributed data management platform.

Appendix

GemFire Concepts

Since GemFire Enterprise Data Fabric has some unique terms it might be helpful if I discuss some of them here before we get started.

Region A logical grouping within the data fabric for a single data set. You can define any number of regions within your fabric. Each region has its own configurable settings governing such things as the data storage model, local data storage and management, partitioning, data event distribution, and data persistence.
A Region is the core of GemFire, and is the main API that is used when developing with GemFire. A Region implements the `ConcurrentMap interface,` which is also an extension of `Map`. This means that if your developers can use a hash map, your developers can use GemFire.

Partitioning A logical division of a region that is distributed over members of the data fabric. For high availability, configure redundant copies so that each data bucket is stored in more than one member, with one member holding the primary copy.

Bucket A unit of storage for distribution and replication, which is distributed in accordance to the region's attribute settings.

Server This is what provides all of the form and function that makes up GemFire.

Client / Client Cache The client enables any Java, C++, or C# application to access, interact with, and register interest in data managed by a Data Management Node. Clients can also be embedded within a Java application server's process to provide HTTP session replication or Hibernate L2 object caching with eviction expiry and overflow to disk.

Serialization The process in which an object state can be saved, transmitted, received and restored. The process of saving or transmitting state is called serializing. The process of receiving and restoring state is called deserialization.

Locator A GemFire locator is a registry process where servers can advertise their location so clients and other servers can discover the active servers.

Cache.xml This allows system architects to declaratively create the cache through xml. While the file can be any name it commonly referred to as a "cache.xml" file.

Cache Is a set of Regions. This is also referred to as a data fabric.

Redundancy Highly available partitioned regions provide reliability by maintaining redundant data. When you configure a partitioned region for redundancy, each entry in the region is stored in at least two members' caches. If one of its members fails, operations continue on the partitioned region with no interruption of service. Recovering redundancy can be configured to take place immediately, or delayed for a configurable amount of time until a replacement system is started.

Without redundancy, failure of any of the members hosting the partitioned region results in partial loss of data. Typically redundancy is not used when applications can directly read from another data source, or when write performance out weighs read performance. The addition of a redundant copy ensures high availability of your data in the partitioned region.

Failover When a server hosting a subscription queue fails, the queueing responsibilities pass to another server. How this happens depends on whether the new server is a secondary server. In any case, all failover activities are carried out automatically by the GemFire system.

Online Material

GemFire Documentation: https://www.vmware.com/support/pubs/vfabric-gemfire.html

GemFire Forums: http://communities.vmware.com/community/vmtn/appplatform/vfabric_gemfire

Lucene documentation: http://lucene.apache.org/core

Spring-GemFire documentation: http://www.springsource.org/spring-gemfire

Spring-Data documentation: http://www.springsource.org/spring-data