Image Search Based on Elasticsearch

By Pang Cong 04/04/2014

PURPOSE

To build a near-real-time Image feature indexing and search system based on elasticsearch.

WHAT HAVE BEEN DONE[[1]](#footnote-2)

I have modified the ES to be a quick runnable version for image search. For example:

1. utilize the “document store” characteristics of Elasticsearch as a distributed -type NoSQL-database to store binary data;
2. index: create index which is stored in memory;
3. search: for ranking purposes, the calculation of the document score is modified be based on the distance of image feature, rather than the default Lucene distance function.

**Acronyms and Synonym**

|  |  |
| --- | --- |
| Abbreviation | Meaning |
| Database | ES index |
| Database 1,2,3 | ES sharding index 1,2,3 |

**Performance of Search**

|  |  |  |
| --- | --- | --- |
| Scale of Data | Hardware configuration | latency |
| 200,000 | 1 computer (Intel® Core™ i7-4770 CPU, Ubuntu) | 70ms |
| 200,000 | 3 computers (Intel® Core™ i7-4770 CPU, Ubuntu) | 50ms |
| 3,000,000 | 5 computers (Intel® Core™ i7-4770 CPU, Ubuntu) | 280ms |

TO DO LIST

The following is the list of things can be done to improve the current modified version of image search of ES:

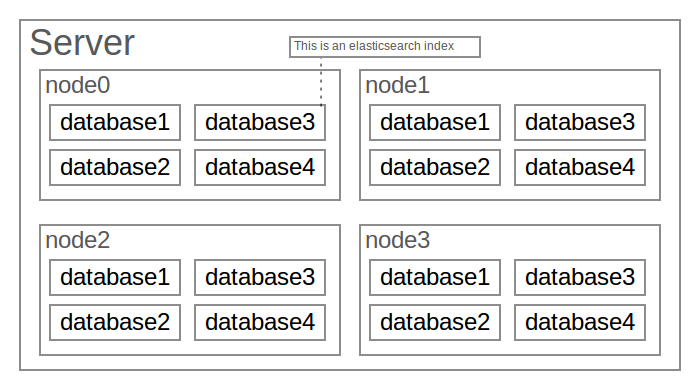
1. hashmap should be structured(nested), table name should be added.
2. reverse of hashmap must be synchronized.
3. query should be standard. we need to build our own query structure.
4. when a image feature is deleted from the elasticsearch, the delete function from hashmap should be implemented

**Elasticsearch is a lot of things. It uses Lucene for full text search. It is schema free, document oriented and a NoSQL key-value store. Everything is dynamic with out of the box scaling and failover. High availability and near realtime search results are easily attained.**

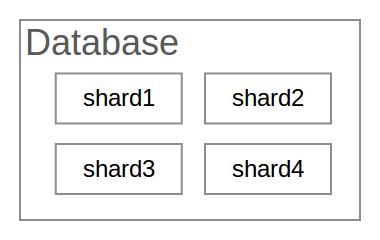
**This is all accessible through beautiful APIs available in multiple formats.**

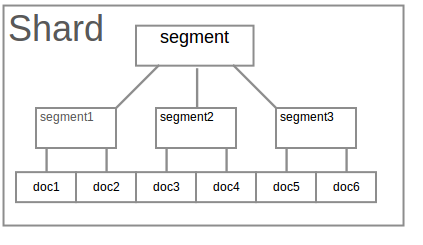
RELATED KEY ES FEATURES

**A node** is an elasticsearch instance.A server can hold multiple nodes. But usually only one node run on one server in the real distributed system.



**An Elasticsearch index**[[2]](#footnote-3) is made up of one or more shards, which can have zero or more replicas. These are all individual Lucene indexes. That is, an Elasticsearch index is made up of many Lucene indexes, which in turn is made up of index segments. When you search an Elasticsearch index, the search is executed on all the shards - and in turn, all the segments - and merged. The same is true when you search multiple Elasticsearch indexes. Actually, searching two Elasticsearch indexes with one shard each is pretty much the same as searching one index with two shards. In both cases, two underlying Lucene indexes are searched.





**Shard** is the basic scaling unit for Elasticsearch. As documents are added to the index, it is routed to a shard. By default, this is done in a round-robin fashion, based on the hash of the document’s id. In the second part of this series, we will look more into how shards are moved around. It is important to know, however, that the number of shards is specified at index creation time, and cannot be changed later on. An early [presentation on Elasticsearch by Shay](http://vimeo.com/26710663) has excellent coverage of why a shard is actually a complete Lucene index, and its various benefits and tradeoffs compared to other methods.

Which Elasticsearch indexes, and what shards (and replicas) search requests are sent to, can be customized in many ways. By combining index patterns, index aliases, and document and search routing, lots of different partitioning and data flow strategies can be implemented. We will not go into them here, but we can recommend Zachary Tong’s article on [customizing document routing](http://www.elasticsearch.org/blog/customizing-your-document-routing/) and Shay Banon’s presentation on [big data, search and analytics](http://vimeo.com/44716955). Just to give you some ideas, here are some examples:

* Lots of data is time based, e.g. logs, tweets, etc. By creating an index per day (or week, month, …), we can efficiently limit searches to certain time ranges - and expunge old data. Remember, we cannot efficiently delete from an existing index, but deleting an entire index is cheap.
* When searches must be limited to a certain user (e.g. “search your messages”), it can be useful to route all the documents for that user to the same shard, to reduce the number of indexes that must be searched.

**Index Summary**

* Indexes are built first in-memory, then occasionally flushed in *segments* to disk.
* Index segments are immutable. Deleted documents are *marked* as such. All changes are added to the auxiliary index(segment) in batches
* An index is made up of multiple segments. A search is done on every segment, with the results merged.
* Segments are occasionally merged. Deleted documents is actually removed during merge process
* Field and filter caches are per segment.

DETAIL OF THE MODIFICATION TO ES

We have done the following things to modified the ES for image search purposes:

1. utilize the “document store” characteristics of Elasticsearch as a distributed -type NoSQL-database to store binary data;
2. index: create index which is stored in memory;
3. search: for ranking purposes, the calculation of the document score is modified be based on the distance of image feature, rather than the default Lucene distance function.

**Store Binary Data**

The binary type is a base64 representation of binary data that can be stored in the index. The field is not stored by default and not indexed at all.

We input image feature data as binary type so this part of data will go directly to the document which will finally be stored on the disk.

**Store index in memory**

we build a hashmap to store all the inserted documents (image visual features). Data is inserted in the hashmap at two places, which are when inserting a new document and the other is when elasticsearch is restarted.

**Scenario 1: Inserting a New Document**

curl -XPOST 'http://127.0.0.1:9200/database1/table2/1 -d '{"message":"pangcong1"}'

In the insert command above, “database1” is the database name, “table2” is the table name in the database, “1” is the image id, “{"message":"pangcong1"}” is a term identified by image id. “pangcong1” is the image feature.

When elasticseach received this command, it will finally go to the file src/main/java/org/apache/lucene/index/IndexWriter.java

We add a hashmap here to keep the feature data in memory after luncene store it in a document. Below is the code

if(features != null)

{

String hashKey = null;

String hashValue = null;

for(IndexableField field : doc) {

final String fieldName = field.name();

if(fieldName.equals("\_uid"))

{

hashKey = field.stringValue();

}

else if(fieldName.equals("message"))

{

hashValue = field.stringValue();

}

}

if(hashKey != null && hashValue != null)

{

byte[] feature = Base64.decodeBase64(hashValue);

int length = feature.length/4;

float[] value = new float[length];

for(int i = 0; i < length; i++)

{

int j = i\*4;

int asInt = (feature[j+0] & 0xFF)

| ((feature[j+1] & 0xFF) << 8)

| ((feature[j+2] & 0xFF) << 16)

| ((feature[j+3] & 0xFF) << 24);

value[i] = Float.intBitsToFloat(asInt);

}

features.put(hashKey,value);

}

}

**Scenario 2: when elasticsearch is restarted**

When a collapsed elasticsearch, we should load all the stored feature to the hashmap. We add a function in

src/main/java/org/elasticsearch/index/engine/internal/InternalEngine.java to do this part of work.

public void fillHashMap()

{

IndexSearcher indexSearcher = null;

if (searcherManager == null) {

throw new EngineClosedException(shardId);

}

try {

indexSearcher = searcherManager.acquire();

} catch (Throwable ex) {

logger.error("failed to acquire searcher, source {}", ex);

throw new EngineException(shardId, ex.getMessage());

}

if (indexWriter == null) {

throw new EngineClosedException(shardId, failedEngine);

}

// load features now

IndexReader reader = indexSearcher.getIndexReader();

org.apache.lucene.util.Bits liveDocs = MultiFields.getLiveDocs(reader);

int maxDoc = indexSearcher.getIndexReader().maxDoc();

try{

for (int idoc=0; idoc<maxDoc; idoc++) {

if (liveDocs != null && !liveDocs.get(idoc))

continue;

org.elasticsearch.index.fieldvisitor.FieldsVisitor visitor = new AllFieldsVisitor();

reader.document(idoc,visitor);

String key = visitor.uid().toString();

String[] values = null;

try

{

values = visitor.source().toUtf8().split("\"");

}

catch (Throwable e)

{

continue;

}

if(values.length< 5)

{

continue;

}

byte[] feature = Base64.decodeBase64(values[3]);

int length = feature.length/4;

float[] value = new float[length];

for(int i = 0; i < length; i++)

{

int j = i\*4;

int asInt = (feature[j+0] & 0xFF)

| ((feature[j+1] & 0xFF) << 8)

| ((feature[j+2] & 0xFF) << 16)

| ((feature[j+3] & 0xFF) << 24);

value[i] = Float.intBitsToFloat(asInt);

}

features.put(key,value);

}

}

catch (Throwable ex)

{

logger.error("failed to read document, source {}", ex);

throw new EngineClosedException(shardId, failedEngine);

}

}

**Search**

The default scoring is the DefaultSimilarity algorithm in core Lucene. Apparently, it does not apply to our image feature scoring measured by vector distances. Our modification is based on the scoring system of elasticsearch and lucene.

In elasticsearch, based on the query string, there are two different query structure, XFilteredQuery and XConstantScoreQuery. XFilteredQuery(elasticsearch) extends from Query(lucene). XConstantScoreQuery(elasticsearch) extends from ConstantScoreQuery(lucene) which extends from Query(lucene)

For example, it our query string is

curl -XPOST 'http://127.0.0.1:9200/database1/table2/\_search' -d '{"query":{"term":{"message":"pangcong1"}}}'

It will choose XFilteredQuery.

Or if the query string is

curl -XPOST 'http://127.0.0.1:9200/database1/table2/\_search'

It will choose XConstantScoreQuery.

ConstantScoreQuery

We need to use our own scoring method with a query string like this

curl -XPOST 'http://127.0.0.1:9200/database1/table2/\_search' -d '{"query":{"term":{"message":"pangcong1"}}}'

XFilteredQuery is too complicated which contains too many nonsense work from lucene. So we choose XConstantScoreQuery.

Our first step is to force elasticsearch using XConstantScoreQuery. The part modified is in DefaultSearchContext.java. In the function “preProcess”.

if (searchFilter != null) {

// if (Queries.isConstantMatchAllQuery(query())) {

Query q = new org.apache.lucene.search.ConstantScoreQuery(searchFilter,query());

q.setBoost(query().getBoost());

parsedQuery(new ParsedQuery(q, parsedQuery()));

// } else {

// parsedQuery(new ParsedQuery(new XFilteredQuery(query(), searchFilter), parsedQuery()));

// }

}

**ConstantScoreQuery.java and TopScoreDocCollector.java**

ConstantScoreQuery.java is modified to adapt to the query parameters. In ConstantScoreQuery.java

A parameter(public final FilteredQuery delegate) is added in the class so it can store the query term.

Then in src/main/java/org/apache/lucene/search/IndexSearcher.java, following code are added

((TopScoreDocCollector)collector).setTermValue(((TermQuery) ((ConstantScoreQuery) (((ConstantScoreQuery) weight.getQuery()).getQuery())).delegate.getQuery()).getTerm().bytes());

((TopScoreDocCollector)collector).setContext(readerContext);

((TopScoreDocCollector) collector).features = features;

//features = null;

if(features != null)

{

collector.collect(0);

return;

}

In src/main/java/org/apache/lucene/search/TopScoreDocCollector.java, we add our new scoring method. The function collect() in this file is the final step in lucene, luncene collects the finded file id and score in a heap. (The heap is then reconstructed and returned to elasticsearch in the next function)

public void collect(int doc).

{

……

else

{

byte[] target = Base64.decodeBase64(termValue.bytes);

int length = target.length/4;

float[] targetFeature = new float[length];

for(int i = 0; i < length; i++)

{

int j = i\*4;

int asInt = (target[j+0] & 0xFF)

| ((target[j+1] & 0xFF) << 8)

| ((target[j+2] & 0xFF) << 16)

| ((target[j+3] & 0xFF) << 24);

targetFeature[i] = Float.intBitsToFloat(asInt);

}

for (java.util.Map.Entry<String, float[]> entry : features.entrySet()) {

String key = entry.getKey();

float[] value = entry.getValue();

float distance = 0;

if(value.length >= targetFeature.length)

{

for(int i = 0; i < length; i++)

{

float dis = targetFeature[i] - value[i];

distance += dis\*dis;

}

}

else

{

continue;

}

// scoring by distance

if(distance < 10 )

{

score = 10 - distance;

}

// This collector cannot handle these scores:

assert score != Float.NEGATIVE\_INFINITY;

assert !Float.isNaN(score);

totalHits++;

if (score <= pqTop.score) {

// Since docs are returned in-order (i.e., increasing doc Id), a document

// with equal score to pqTop.score cannot compete since HitQueue favors

// documents with lower doc Ids. Therefore reject those docs too.

continue;

}

pqTop.doc = -1;

pqTop.score = score;

pqTop.key = key;

pqTop = pq.updateTop();

}

// get docId from key;

final ScoreDoc[] scoreDocs = new ScoreDoc[pq.size()];

for (int i = pq.size() - 1; i >= 0; i--) // put docs in array

{

scoreDocs[i] = pq.pop();

org.apache.lucene.index.Term term = new org.apache.lucene.index.Term("\_uid",scoreDocs[i].key);

scoreDocs[i].doc = org.elasticsearch.common.lucene.uid.Versions.loadRealDocId(context.reader(), term);

}

pq.clear();

for(int i = scoreDocs.length-1; i>=0;i--)

{

pq.add(scoreDocs[i]);

}

}

…...

}

### Introduction to Lucene

Elasticsearch uses Lucene for full text search.

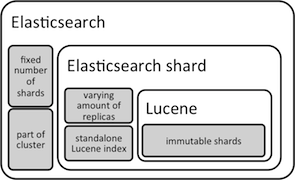
Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

Elasticsearch shards documents based on a hash of a document's unique identifier. Each shard is a standalone Lucene-index. Multiple shards make up a search-index in elasticsearch. An index can have multiple types. One could compare databases to search indices; shards, pages and partitions to shards and lucene-indices and tables to types.

The number of shards is specified during index creation and then fixed. The number of replicas of each shard can be changed at will.

There is no optimal number of shards: it is dependent on the requirement of your data. A default of 5 shards and 1 replica works for most cases but probably won't give you the best results for your particular data. Having more *shards* enhances the *indexing* performance and allows to *distribute* a big index across machines. Having more *replicas* enhances the *search* performance and improves the cluster *availability*.

When it has been determined that the defaults don't suit your needs, it is important to conduct a proper investigation into what those needs are and how they can be accomplished. One should be careful when deviating from these defaults.



Graphical layout of the structure of an Elasticsearch node in a cluster.

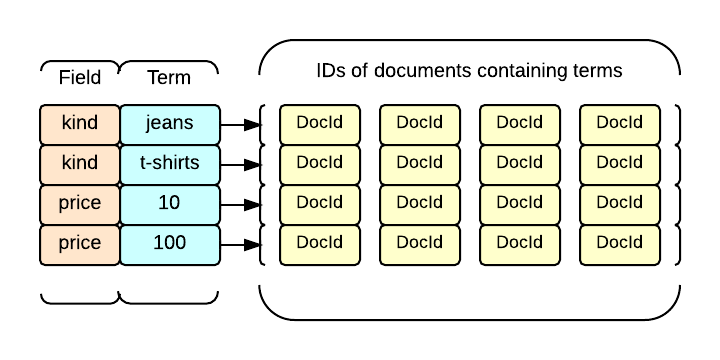
**Segments**

Lucene indexes may be composed of multiple sub-indexes, or *segments*. Each segment is a fully independent index, which could be searched separately. Indexes evolve by:

1. Creating new segments for newly added documents.
2. Merging existing segments.

Searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments.

#### **Terms**

Designing good schemata to optimize your search results is completely different than designing schemata in other systems. In SQL based stores you design a scheme based on relationships and data types. However, when optimizing for text search, you have to go one step further. Here the lowest denominator is a single term. All input gets reduced to terms, whether it is boolean, numeric, date related or just text. It doesn't matter.

1. Detailed commit in Github: https://github.com/visenze/elasticsearch [↑](#footnote-ref-2)
2. we may refer Elasticsearch index as database sometimes. [↑](#footnote-ref-3)