Architecture of Enterprise Applications 7
Full-text Searching

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Contents
  - Lucene
  - Solr
  - Elasticsearch

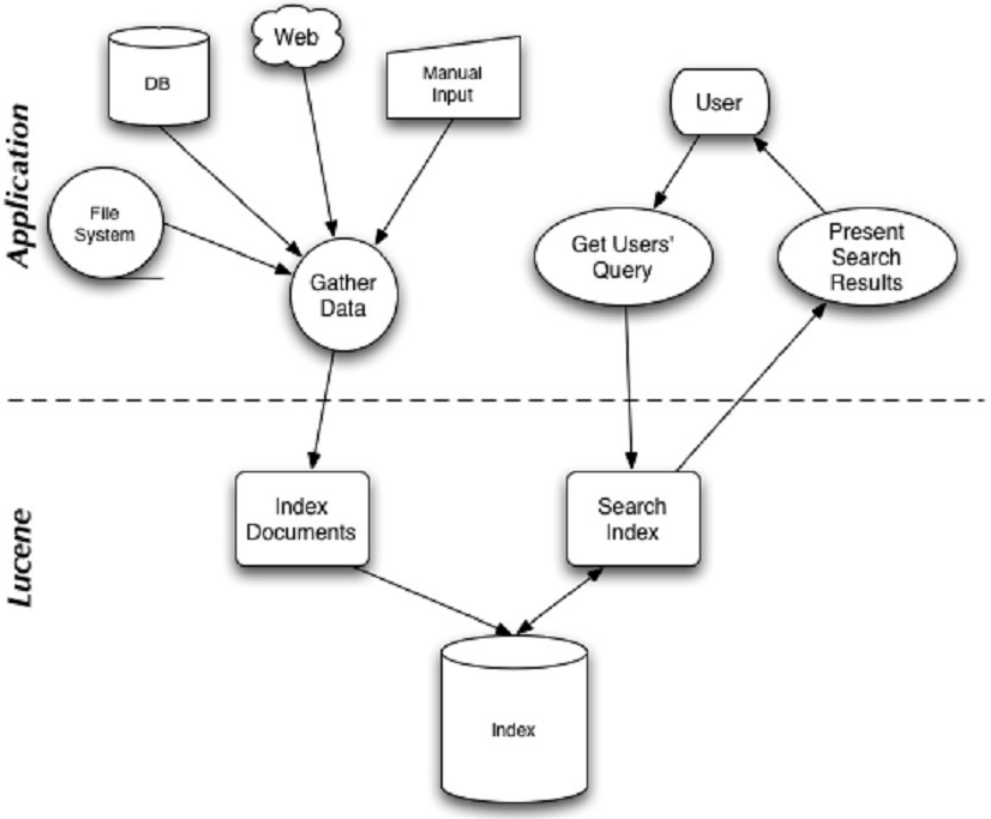
- Objectives
  - 能够根据业务需求，识别适合全文搜索引擎的场景，设计并实现基于搜索引擎的全文搜索方案

Lucene

- Lucene is a high performance, scalable Information Retrieval (IR) library.
  - It lets you add indexing and searching capabilities to your applications.
  - Lucene is a mature, free, open-source project implemented in Java.
  - it's a member of the popular Apache Jakarta family of projects, licensed under the liberal Apache Software License.

- Lucene provides a simple yet powerful core API
  - that requires minimal understanding of full-text indexing and searching.

Lucene


Application
DB
Web
Manual Input
File System
Gather Data
User
Get Users' Query
Present Search Results
Lucene
Index Documents
Search Index
Index

Indexing

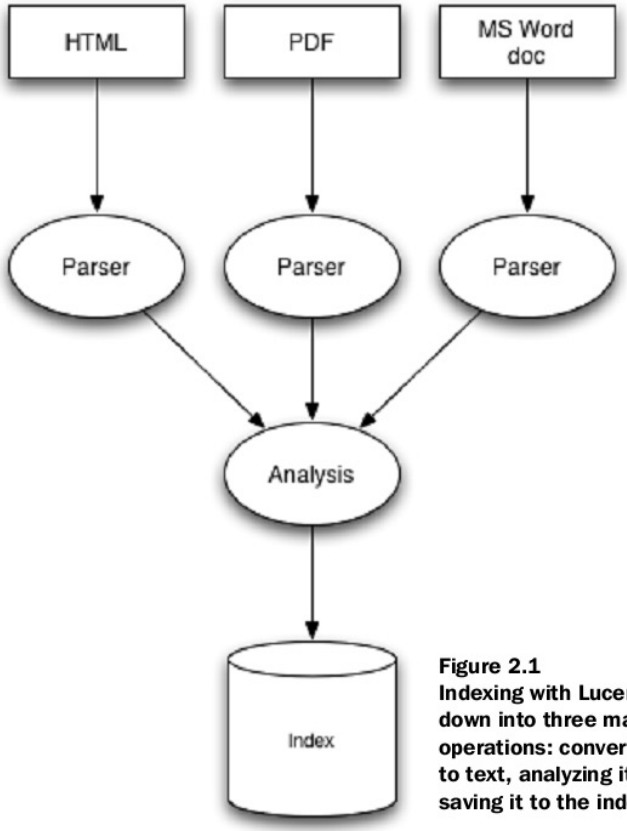


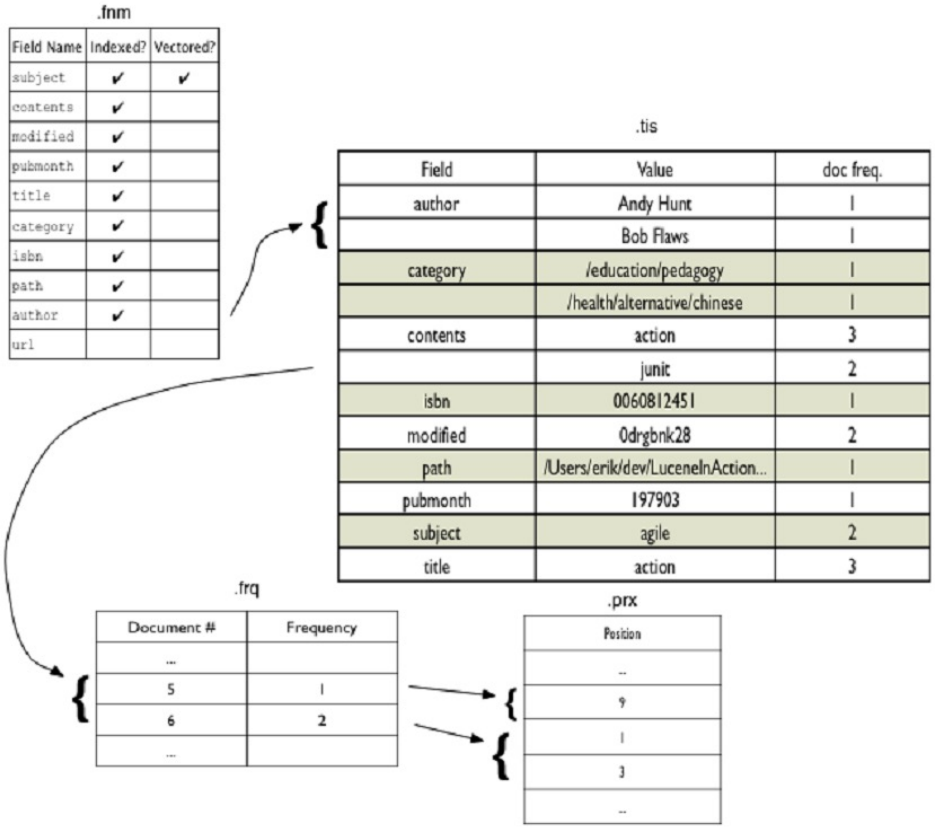
Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

- At the heart of all search engines is the concept of indexing:
  - processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching.

- Suppose you needed to search a large number of files, and you wanted to be able to find files that contained a certain word or a phrase
  - A naïve approach would be to sequentially scan each file for the given word or phrase.
  - This approach has a number of flaws, the most obvious of which is that it doesn't scale to larger file sets or cases where files are very large.

- This is where indexing comes in:
  - To search large amounts of text quickly, you must first index that text and convert it into a format that will let you search it rapidly, eliminating the slow sequential scanning process.
  - This conversion process is called indexing, and its output is called an index.
  - You can think of an index as a data structure that allows fast random access to words stored inside it.

Inverting index



Searching

- Searching is the process of looking up words in an index to find documents where they appear.

- The quality of a search is typically described using precision and recall metrics.
  - Recall measures how well the search system finds relevant documents, whereas precision measures how well the system filters out the irrelevant documents.

- A number of other factors
  - speed and the ability to quickly search large quantities of text.
  - Support for single and multi term queries, phrase queries, wildcards, result ranking, and sorting are also important, as is a friendly syntax for entering those queries.

A sample application

- Suppose you need to index and search files stored in a directory tree, not just in a single directory
- These example applications will familiarize you with Lucene's API, its ease of use, and its power.
- The code listings are complete, ready-to-use command-line programs.

Creating an Index


```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Indexer {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new Exception("Usage: java " + Indexer.class.getName()
        + " <index dir> <data dir>");
    }
    File indexDir = new File(args[0]);
    File dataDir = new File(args[1]);

    long start = new Date().getTime();
    int numIndexed = index(indexDir, dataDir);
    long end = new Date().getTime();

    System.out.println("Indexing " + numIndexed + " files took " + (end - start) + " milliseconds");
  }
```

Create Lucene index in this directory

Index files in this directory

Creating an Index


```
// open an index and start file directory traversal
 public static int index(File indexDir, File dataDir) throws IOException {
  if (!dataDir.exists() || !dataDir.isDirectory()) {
    throw new IOException(dataDir
      + " does not exist or is not a directory");
  }

  IndexWriter writer = new IndexWriter(indexDir,
      new StandardAnalyzer(), true);
  writer.setUseCompoundFile(false);

  indexDirectory(writer, dataDir);

  int numIndexed = writer.docCount();
  writer.optimize();
  writer.close();
  return numIndexed;
}
```

Create Lucene index

Close index

Creating an Index


```
// recursive method that calls itself when it finds a directory
private static void indexDirectory(IndexWriter writer, File dir)
  throws IOException {

  File[] files = dir.listFiles();

  for (int i = 0; i < files.length; i++) {
    File f = files[i];
    if (f.isDirectory()) {
      indexDirectory(writer, f);
    } else if (f.getName().endsWith(".txt")) {
        indexFile(writer, f);
    }
  }
}
```

recurse

Index .txt files only

Creating an Index


```
// method to actually index a file using Lucene
 private static void indexFile(IndexWriter writer, File f)
   throws IOException {

   if (f.isHidden() || !f.exists() || !f.canRead()) {
    return;
   }

   System.out.println("Indexing " + f.getCanonicalPath());

   Document doc = new Document();
   doc.add(Field.Text("contents", new FileReader(f)));

   doc.add(Field.Keyword("filename", f.getCanonicalPath()));
   writer.addDocument(doc);
 }
}
```

← Index file content

← Index file name

← Add document to Lucene index

Running Indexer


```
% java lia.meetlucene.Indexer build/index/lucene
Indexing /lucene/build/test/TestDoc/test.txt
Indexing /lucene/build/test/TestDoc/test2.txt
Indexing /lucene/BUILD.txt
Indexing /lucene/CHANGES.txt
Indexing /lucene/LICENSE.txt
Indexing /lucene/README.txt
Indexing /lucene/src/jsp/README.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/stemsUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/test1251.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/testKOI8.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/testUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/wordsUnicode.txt
Indexing /lucene/todo.txt
Indexing 13 files took 2205 milliseconds
```

Searching an index


```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Searcher {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new Exception("Usage: java " + Searcher.class.getName()
        + " <index dir> <query>");
    }
    File indexDir = new File(args[0]);
    String q = args[1];

    if (!indexDir.exists() || !indexDir.isDirectory()) {
      throw new Exception(indexDir +
        " does not exist or is not a directory.");
    }

    search(indexDir, q);
  }
```

Index directory created by Indexer

Query string

Searching an index


```
public static void search(File indexDir, String q)
    throws Exception {
    Directory fsDir = FSDirectory.getDirectory(indexDir, false);
    IndexSearcher is = new IndexSearcher(fsDir);

    Query query = QueryParser.parse(q, "contents",   new StandardAnalyzer());
    long start = new Date().getTime();
    Hits hits = is.search(query);
    long end = new Date().getTime();

    System.err.println("Found " + hits.length() +  " document(s) (in " + (end - start) +
                       " milliseconds) that matched query '" + q + "':");

    for (int i = 0; i < hits.length(); i++) {
      Document doc = hits.doc(i);
      System.out.println(doc.get("filename"));
    }
  }
}
```

Open Index

Parse query

Search Index

Write search stats

Retrieve matching document

Display filename

Running Searcher


```
%java lia.meetlucene.Searcher build/index 'lucene'
Found 6 document(s) (in 66 milliseconds) that matched query 'lucene':
/lucene/README.txt
/lucene/src/jsp/README.txt
/lucene/BUILD.txt
/lucene/todo.txt
/lucene/LICENSE.txt
/lucene/CHANGES.txt
```

Core indexing classes

- IndexWriter
  - This class creates a new index and adds documents to an existing index.
- Directory
  - The Directory class represents the location of a Lucene index.
- Analyzer
  - The Analyzer, specified in the IndexWriter constructor, is in charge of extracting tokens out of text to be indexed and eliminating the rest.
- Document
  - A Document represents a collection of fields.
- Field
  - Each field corresponds to a piece of data that is either queried against or retrieved from the index during search.

Core searching classes

- IndexSearcher
  - IndexSearcher is to searching what IndexWriteris to indexing
- Term
  - A Term is the basic unit for searching.
- Query
  - Query is the common, abstract parent class. It contains several utility methods
- TermQuery
  - TermQuery is the most basic type of query supported by Lucene, and it's one of the primitive query types.
- Hits
  - The Hits class is a simple container of pointers to ranked search results

Adding documents to an index


```
public abstract class BaseIndexingTestCase extends TestCase {
  protected String[] keywords = {"1", "2"};
  protected String[] unindexed = {"Netherlands", "Italy"};
  protected String[] unstored = {"Amsterdam has lots of bridges",
                                 "Venice has lots of canals"};
  protected String[] text = {"Amsterdam", "Venice"};
  protected Directory dir;

  protected void setUp() throws IOException {
    String indexDir =
              System.getProperty("java.io.tmpdir", "tmp") +
              System.getProperty("file.separator") + "index-dir";
    dir = FSDirectory.getDirectory(indexDir, true);
    addDocuments(dir);
  }
```

Adding documents to an index


```
  protected void addDocuments(Directory dir) throws IOException {
    IndexWriter writer = new IndexWriter(dir, getAnalyzer(), true);
    writer.setUseCompoundFile(isCompound());
    for (int i = 0; i < keywords.length; i++) {
      Document doc = new Document();
      doc.add(Field.Keyword("id", keywords[i]));
      doc.add(Field.UnIndexed("country", unindexed[i]));
      doc.add(Field.UnStored("contents", unstored[i]));
      doc.add(Field.Text("city", text[i]));
      writer.addDocument(doc);
    }
    writer.optimize();
    writer.close();
  }
  protected Analyzer getAnalyzer() { return new SimpleAnalyzer();}
  protected boolean isCompound() { return true; }
}
```

Fields

- All fields consist of a name and value pair.
  - Keyword—Isn't analyzed, but is indexed and stored in the index verbatim.
  - UnIndexed—Is neither analyzed nor indexed, but its value is stored in the index as is.
  - UnStored—The opposite of UnIndexed. This field type is analyzed and indexed but isn't stored in the index.
  - Text—Is analyzed, and is indexed. This implies that fields of this type can be searched against, but be cautious about the field size.

Adding documents to an index

- **Heterogeneous Documents**
  - One handy feature of Lucene is that it allows Documents with different sets of Fields to coexist in the same index.
  - This means you can use a single index to hold Documents that represent different entities.
  - For instance, you could have Documents that represent retail products with Fields such as name and price, and Documents that represent people with Fields such as name, age, and gender.

Adding documents to an index

- **Appendable Fields**
- Suppose you have an application that generates an array of synonyms for a given word, and you want to use Lucene to index the base word plus all its synonyms.

- like this:

```
String baseWord = "fast";
String synonyms[] = String {"quick", "rapid", "speedy"};
Document doc = new Document();
doc.add(Field.Text("word", baseWord));
for (int i = 0; i < synonyms.length; i++) {
  doc.add(Field.Text("word", synonyms[i]));
}
```

- Internally, Lucene appends all the words together and index them in a single Field called word, allowing you to use any of the given words when searching.

Removing Documents from an index


```
public class DocumentDeleteTest extends BaseIndexingTestCase {
  public void testDeleteBeforeIndexMerge() throws IOException {
    IndexReader reader = IndexReader.open(dir);
    assertEquals(2, reader.maxDoc());
    assertEquals(2, reader.numDocs());
    reader.delete(1);
    assertTrue(reader.isDeleted(1));
    assertTrue(reader.hasDeletions());
    assertEquals(2, reader.maxDoc());
    assertEquals(1, reader.numDocs());
    reader.close();
    reader = IndexReader.open(dir);
    assertEquals(2, reader.maxDoc());
    assertEquals(1, reader.numDocs());
    reader.close();
  }
```

Removing Documents from an index


```
public void testDeleteAfterIndexMerge() throws IOException {
  IndexReader reader = IndexReader.open(dir);
  assertEquals(2, reader.maxDoc());
  assertEquals(2, reader.numDocs());
  reader.delete(1);
  reader.close();
  IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
  writer.optimize();
  writer.close();
  reader = IndexReader.open(dir);
  assertFalse(reader.isDeleted(1));
  assertFalse(reader.hasDeletions());
  assertEquals(1, reader.maxDoc());
  assertEquals(1, reader.numDocs());
  reader.close();
}
```

Undeleting Documents

- Because Document deletion is deferred until the closing of the IndexReader instance,
  - Lucene allows an application to change its mind and undelete Documents that have been marked as deleted.

- A call to IndexReader's undeleteAll()method undeletes all deleted Documents
  - by removing all .del files from the index directory.

- Subsequently closing the IndexReader instance therefore leaves all Documents in the index.
  - Documents can be undeleted only if the call to undeleteAll()was done using the same instance of IndexReader that was used to delete the Documents in the first place.

Updating Documents in an index

- "How do I update a document in an index?"
  - is a frequently asked question on the Lucene user mailing list.
- Lucene doesn't offer an update(Document)method;
  - instead, a Document must first be deleted from an index and then re-added to it

```
IndexReader reader = IndexReader.open(dir);
reader.delete(new Term("city", "Amsterdam"));
reader.close();

IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
Document doc = new Document();
doc.add(Field.Keyword("id", "1"));
doc.add(Field.UnIndexed("country", "Netherlands"));
doc.add(Field.UnStored("contents", "Amsterdam has lots of bridges"));
doc.add(Field.Text("city", "Haag"));
writer.addDocument(doc);
writer.optimize();
writer.close();
```

Boosting Documents and Fields

- Not all Documents and Fields are created equal
  - Document boosting is a feature that makes such a requirement simple to implement.
  - By default, all Documents have no boost—or, rather, they all have the same boost factor of 1.0.
  - By changing a Document's boost factor, you can instruct Lucene to consider it more or less important with respect to other Documents in the index.
  - The API for doing this consists of a single method, setBoost(float)

Boosting Documents and Fields


```
public static final String COMPANY_DOMAIN = "example.com";
public static final String BAD_DOMAIN = "yucky-domain.com";

Document doc = new Document();
String senderEmail = getSenderEmail();
String senderName = getSenderName();
String subject = getSubject();
String body = getBody();

doc.add(Field.Keyword("senderEmail", senderEmail));
doc.add(Field.Text("senderName", senderName));
doc.add(Field.Text("subject", subject));
doc.add(Field.UnStored("body", body));

if (getSenderDomain().endsWithIgnoreCase(COMPANY_DOMAIN)) {
    doc.setBoost(1.5);
}
else if (getSenderDomain().endsWithIgnoreCase(BAD_DOMAIN)) {
    doc.setBoost(0.1);
}
writer.addDocument(doc);
```

Tuning indexing performance

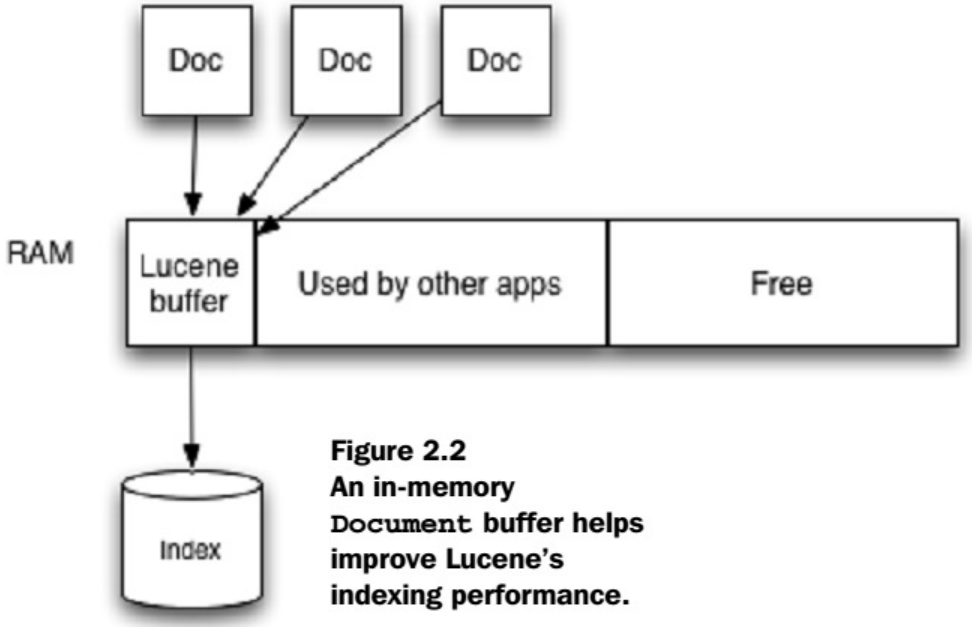


Figure 2.2
An in-memory `Document` buffer helps improve Lucene's indexing performance.

Parallelizing indexing by working with multiple indexes

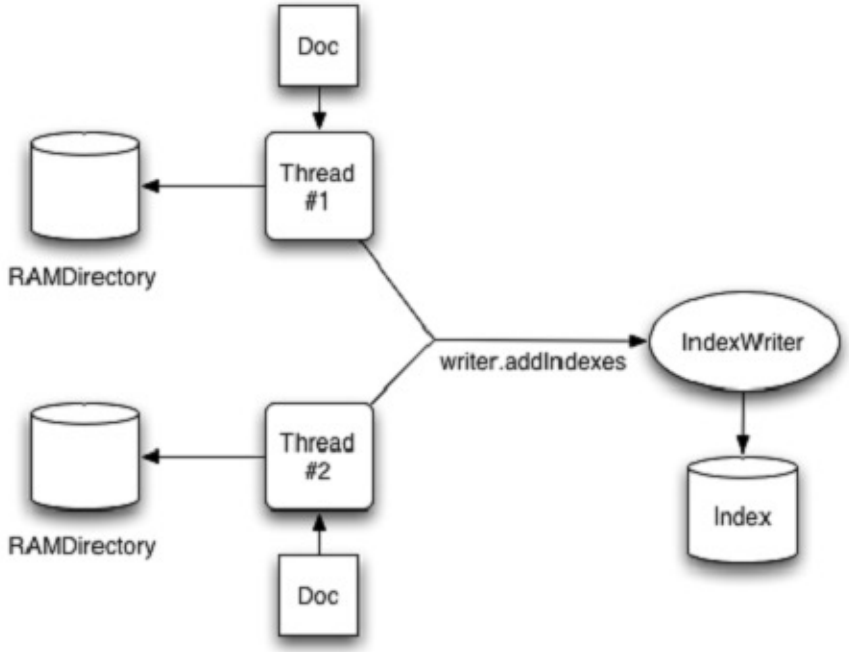


Figure 2.3 A multithreaded application that uses multiple `RAMDirectory` instances for parallel indexing.

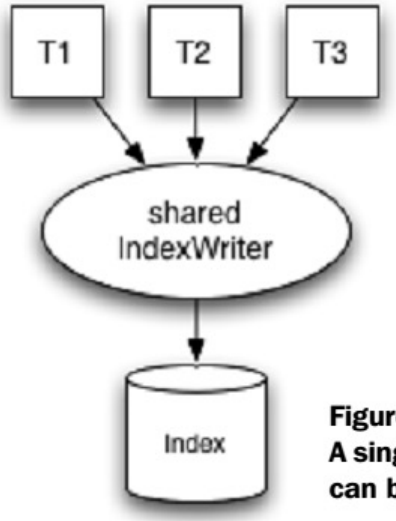


Figure 2.7
A single `IndexWriter` or `IndexReader` can be shared by multiple threads.

Searching for a specific term

- A term is a value that is paired with its containing field.

```
IndexSearcher searcher = new IndexSearcher(directory);

Term t = new Term("subject", "ant");
Query query = new TermQuery(t);
Hits hits = searcher.search(query);

t = new Term("subject", "junit");
hits = searcher.search(new TermQuery(t));

searcher.close();
```

Parsing a user-entered query expression: QueryParser


```
IndexSearcher searcher = new IndexSearcher(directory);
Query query = QueryParser.parse("+JUNIT +ANT -MOCK", "contents",
                                new SimpleAnalyzer());
Hits hits = searcher.search(query);

Document d = hits.doc(0);

query = QueryParser.parse("mock OR junit", "contents",
                                new SimpleAnalyzer());
hits = searcher.search(query);
```

Understanding Lucene scoring

- The score is computed for each document (d) matching a specific.
  - This score is the raw score.
  - Scores returned from Hits aren't necessarily the raw score, however.
  - If the top-scoring document scores greater than 1.0, all scores are normalized from that score, such that all scores from Hits are guaranteed to be 1.0 or less.

Understanding Lucene scoring

$$\sum_{t\ in\ q} tf(t\ in\ d) \cdot idf(t) \cdot boost(t.field\ in\ d) \cdot lengthNorm(t.field\ in\ d)$$

Table 3.5 Factors in the scoring formula

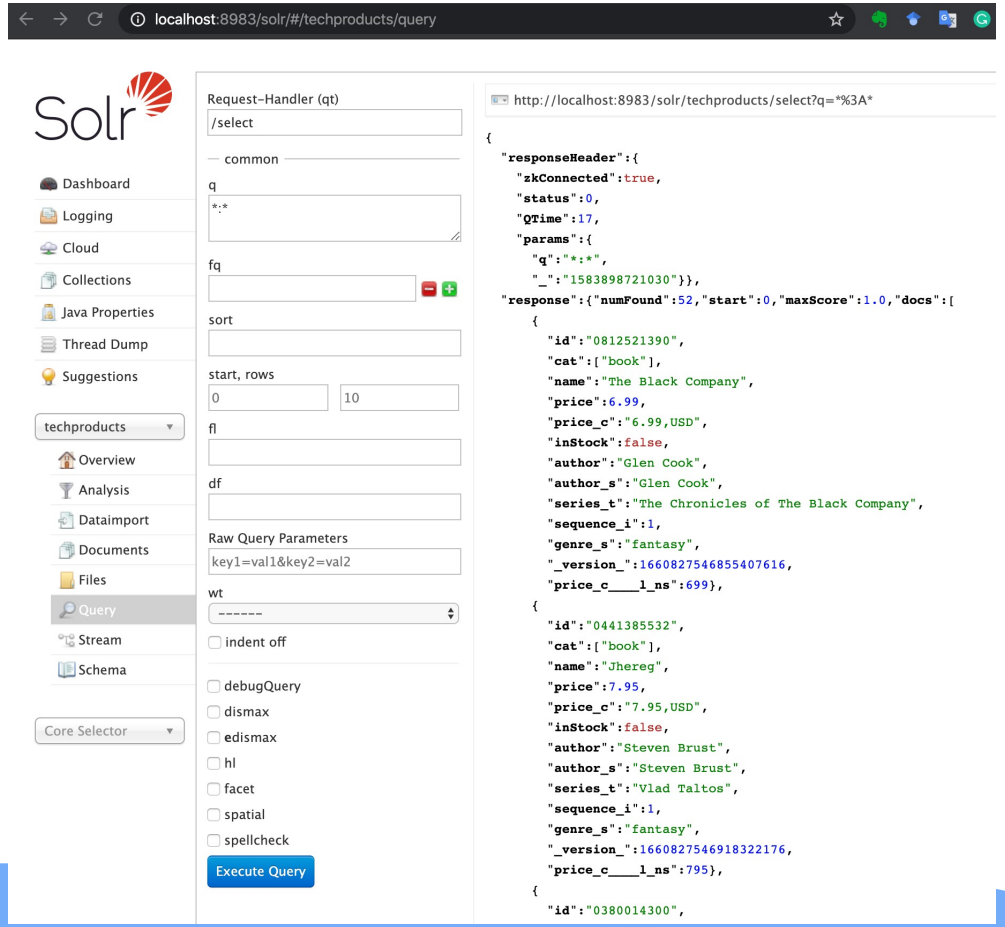
Factor	Description
`tf(t in d)`	Term frequency factor for the term (`t`) in the document (`d`).
`idf(t)`	Inverse document frequency of the term.
`boost(t.field in d)`	Field boost, as set during indexing.
`lengthNorm(t.field in d)`	Normalization value of a field, given the number of terms within the field. This value is computed during indexing and stored in the index.
`coord(q, d)`	Coordination factor, based on the number of query terms the document contains.
`queryNorm(q)`	Normalization value for a query, given the sum of the squared weights of each of the query terms.

Apache Solr

- APACHE SOLR™
  - is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene™.
  - is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more.
  - Solr powers the search and navigation features of many of the world's largest internet sites.

Solr

- Launch Solr in SolrCloud Mode
  - `$ ./bin/solr start -e cloud`



Using SolrJ

- SolrJ
  - is an API that makes it easy for applications written in Java (or any language based on the JVM) to talk to Solr.

- For projects built with Maven, place the following in your `pom.xml`:

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>8.4.1</version>
</dependency>
```

Using SolrJ

- Querying.java

```
public class Querying {
  public static void main(String[] args) throws IOException, SolrServerException {
    final SolrClient client = getSolrClient();

    final Map<String, String> queryParamMap = new HashMap<String, String>();
    queryParamMap.put("q", "*:*");
    queryParamMap.put("fl", "id, name");
    queryParamMap.put("sort", "id asc");
    MapSolrParams queryParams = new MapSolrParams(queryParamMap);

    final QueryResponse response = client.query("techproducts", queryParams);
    final SolrDocumentList documents = response.getResults();

    System.out.println("Found " + documents.getNumFound() + " documents");
    for (SolrDocument document : documents) {
      final String id = (String) document.getFirstValue("id");
      final String name = (String) document.getFirstValue("name");

      System.out.println("id: " + id + "; name: " + name);
    }
  }
```

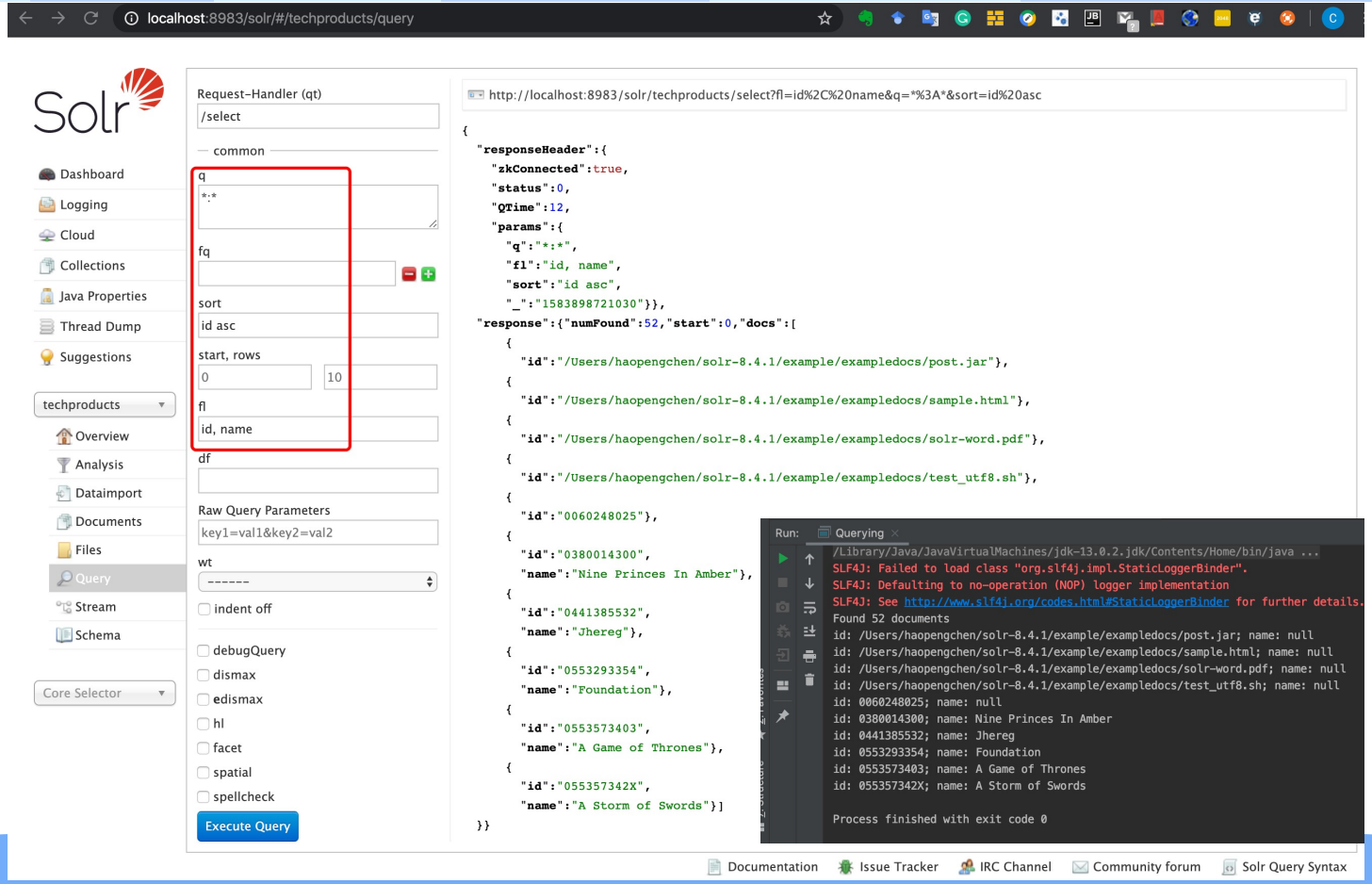
Using SolrJ

- Querying.java

```
    public static SolrClient getSolrClient() {
        final String solrUrl = "http://localhost:8983/solr";
        return new HttpSolrClient.Builder(solrUrl)
                .withConnectionTimeout(10000)
                .withSocketTimeout(60000)
                .build();
    }

}
```

Using SolrJ

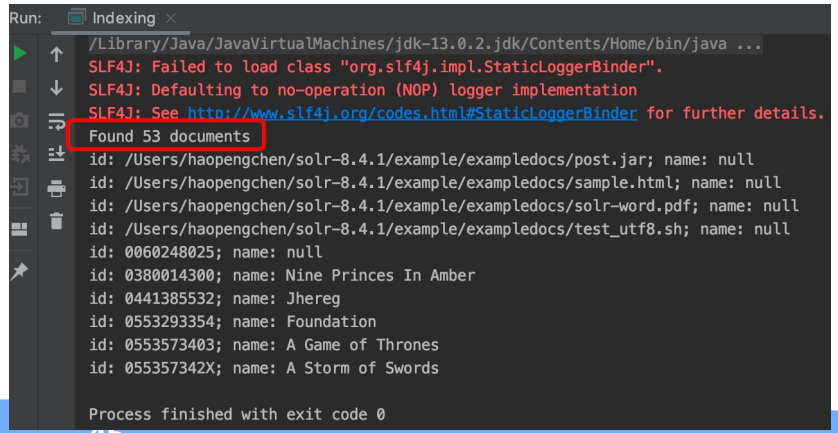
REin
REliable, INtelligent & Scalable Systems



Indexing

- Indexing.java

```
final SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", UUID.randomUUID().toString());
doc.addField("name", "Amazon Kindle Paperwhite");

final UpdateResponse updateResponse = client.add("techproducts", doc);
// Indexed documents must be committed
client.commit("techproducts");
```



Java Object Binding

- TechProduct.java

```
public class TechProduct {
   @Field
   public String id;
   @Field
   public String name;

   public TechProduct(String id, String name) {
      this.id = id;
      this.name = name;
   }

   public TechProduct() { }

   public String getId() {
      return this.id;
   }

   public String getName() {
      return this.name;
   }
}
```

Java Object Binding

- ObjectBinding.java

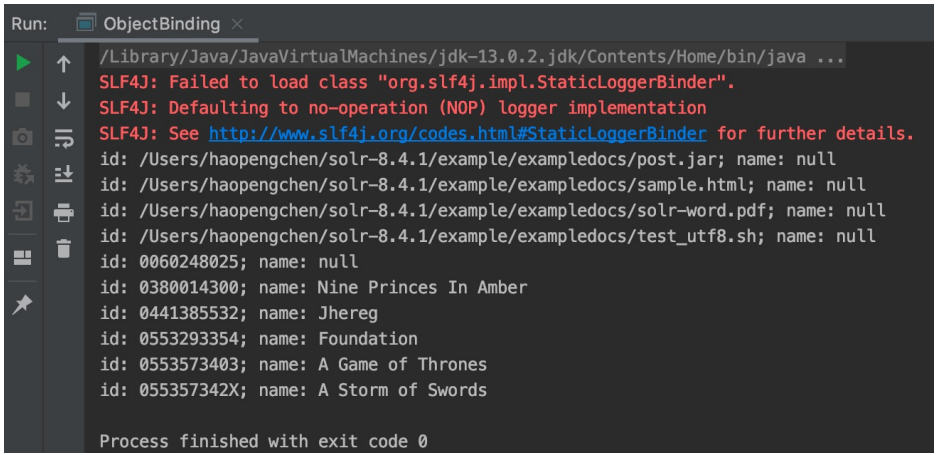
```
public class ObjectBinding {
  public static void main(String[] args) throws IOException, SolrServerException {
    final SolrClient client = getSolrClient();
    final TechProduct kindle = new TechProduct("kindle-id-4", "Amazon Kindle Paperwhite");
    final UpdateResponse response = client.addBean("techproducts", kindle);
    client.commit("techproducts");

    final SolrQuery query = new SolrQuery("*:*");
    query.addField("id");
    query.addField("name");
    query.setSort("id", SolrQuery.ORDER.asc);

    final QueryResponse responseOne = client.query("techproducts", query);
    final List<TechProduct> products = responseOne.getBeans(TechProduct.class);
    for (TechProduct product : products) {
      final String id = product.getId();
      final String name = product.getName();
      System.out.println("id: " + id + "; name: " + name);
    }
  }
```

Java Object Binding

REliable, INtelligent & Scalable Systems

- ObjectBinding.java

```
public static SolrClient getSolrClient() {
  final String solrUrl = "http://localhost:8983/solr";
  return new HttpSolrClient.Builder(solrUrl)
        .withConnectionTimeout(10000)
        .withSocketTimeout(60000)
        .build();
  }
}
```



Elasticsearch

- Elasticsearch is a highly scalable open-source full-text search and analytics engine.
  - It allows you to store, search, and analyze big volumes of data quickly and in near real time.
  - It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.

  - https://www.elastic.co

Elasticsearch – Basic Concepts

- Near Real Time
  - Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time you index a document until the time it becomes searchable.
- Document
  - A document is a basic unit of information that can be indexed.
  - This document is expressed in JSON (JavaScript Object Notation) which is a ubiquitous internet data interchange format.
- Index
  - An index is a collection of documents that have somewhat similar characteristics.
  - An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it.

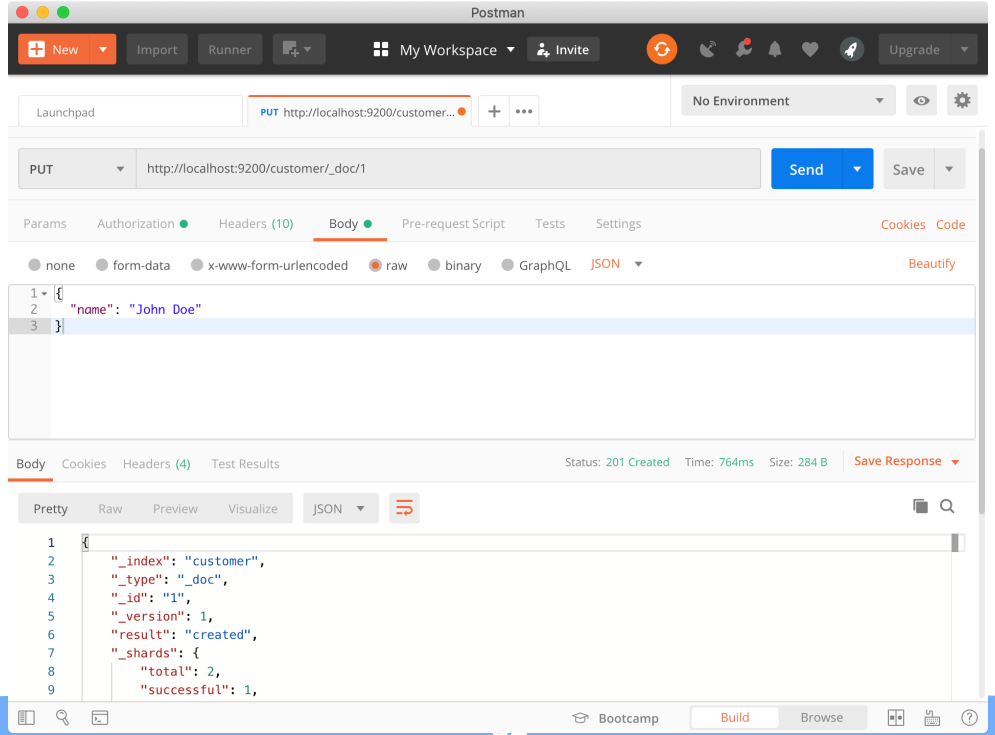
Elasticsearch – Basic Concepts

- Shards & Replicas
  - An index can potentially store a large amount of data that can exceed the hardware limits of a single node.
  - To solve this problem, Elasticsearch provides the ability to subdivide your index into multiple pieces called shards.
- Sharding is important for two primary reasons:
  - It allows you to horizontally split/scale your content volume
  - It allows you to distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput
- Replication is important for two primary reasons:
  - It provides high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
  - It allows you to scale out your search volume/throughput since searches can be executed on all replicas in parallel.

Elasticsearch

- Index some documents

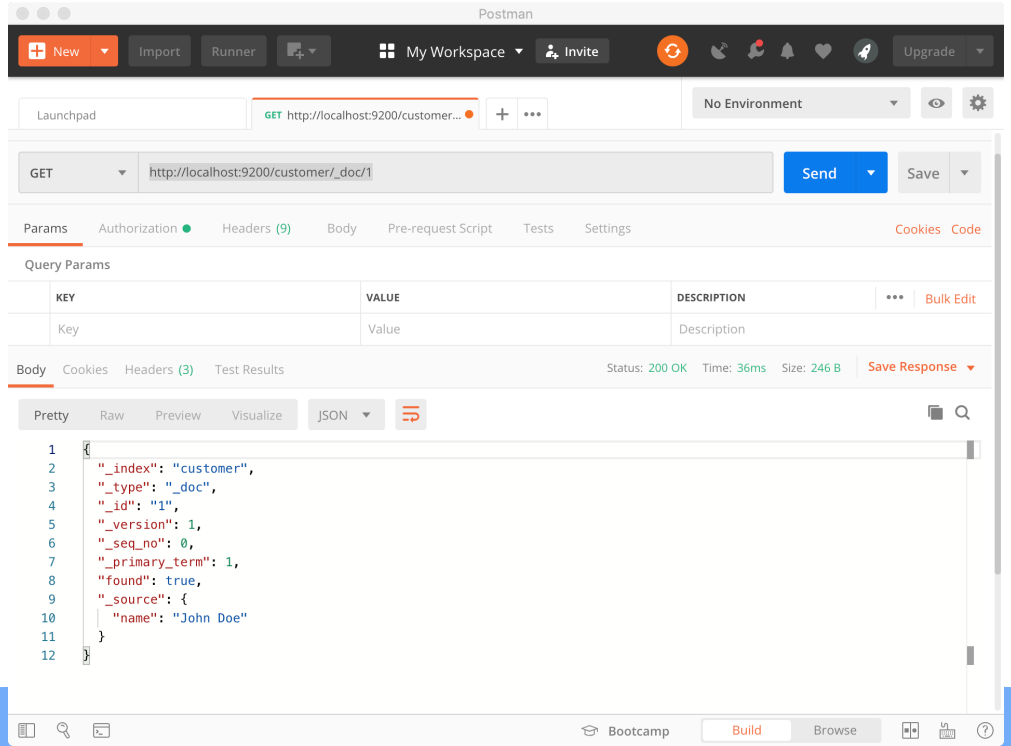
  PUT /customer/_doc/1 (http://localhost:9200/customer/_doc/1)



Elasticsearch

- Index some documents

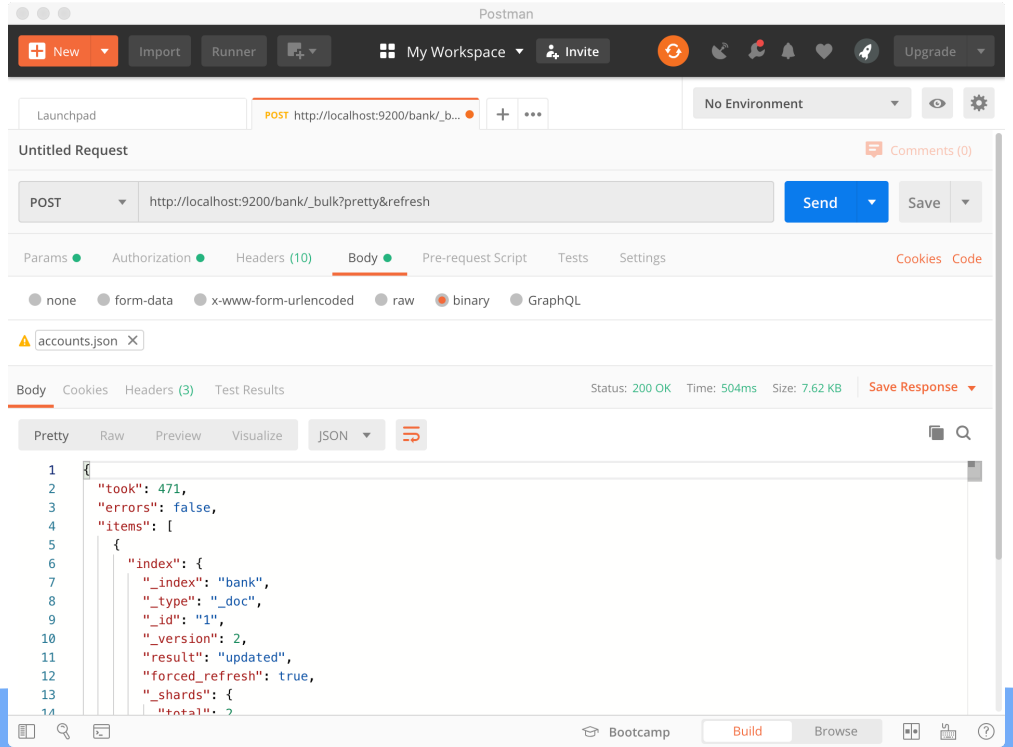
  GET /customer/_doc/1(http://localhost:9200/customer/_doc/1)



Elasticsearch

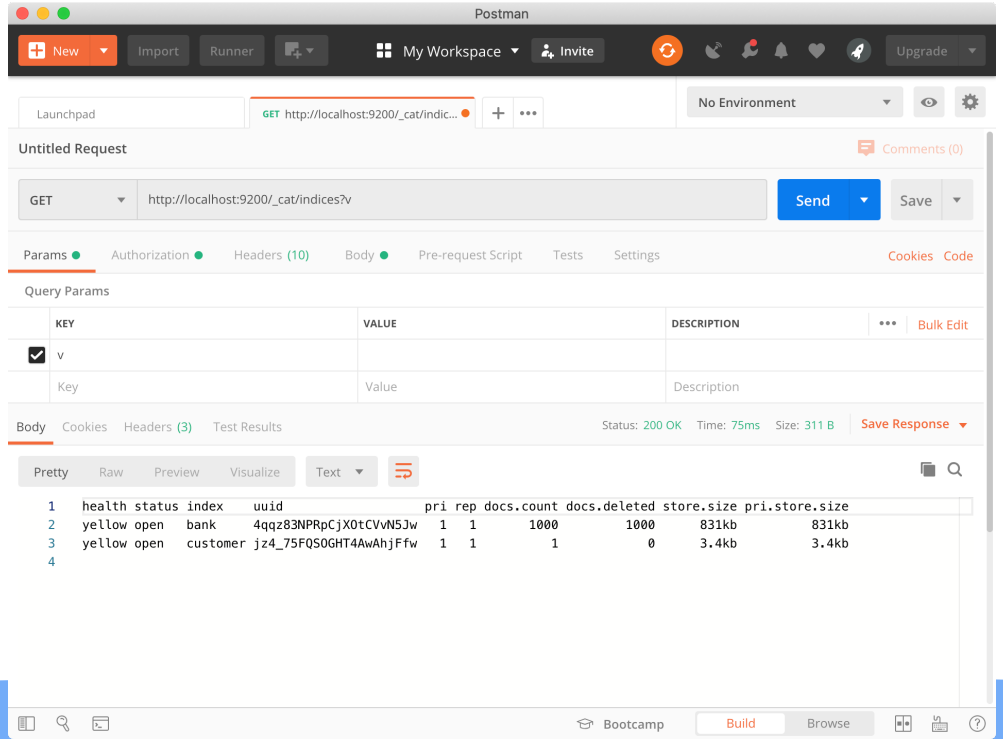
- Indexing documents in bulk

  http://localhost:9200/bank/_bulk?pretty&refresh



Elasticsearch

- Indexing documents in bulk
  - http://localhost:9200/_cat/indices?v



- Delete Index
  - Now let's delete the index that we just created and then list all the indexes again:

```
DELETE /customer?pretty
GET /_cat/indices?v
```

  - And the response:

```
health status index uuid pri rep docs.count docs.deleted store.size pri.store.size
```

  - Before we move on, let's take a closer look again at some of the API commands that we have learned so far:

```
PUT /customer
PUT /customer/_doc/1
{
  "name": "John Doe"
}
GET /customer/_doc/1
DELETE /customer
```

- Indexing and Replacing Documents
  - Let's recall that command again:

```
PUT /customer/_doc/1?pretty
{
  "name": "John Doe"
}
```

  - If we then executed the above command again with a different (or same) document, Elasticsearch will replace (i.e. reindex) a new document on top of the existing one with the ID of 1:

```
PUT /customer/_doc/1?pretty
{
  "name": "Jane Doe"
}
```

- Indexing and Replacing Documents
  - If, on the other hand, we use a different ID, a new document will be indexed and the existing document(s) already in the index remains untouched.

```
PUT /customer/_doc/2?pretty
{
  "name": "Jane Doe"
}
```

  - This example shows how to index a document without an explicit ID:

```
POST /customer/_doc?pretty
{
  "name": "Jane Doe"
}
```

- Deleting Documents
  - Deleting a document is fairly straightforward.

```
DELETE /customer/_doc/2?pretty
```

- Batch Processing

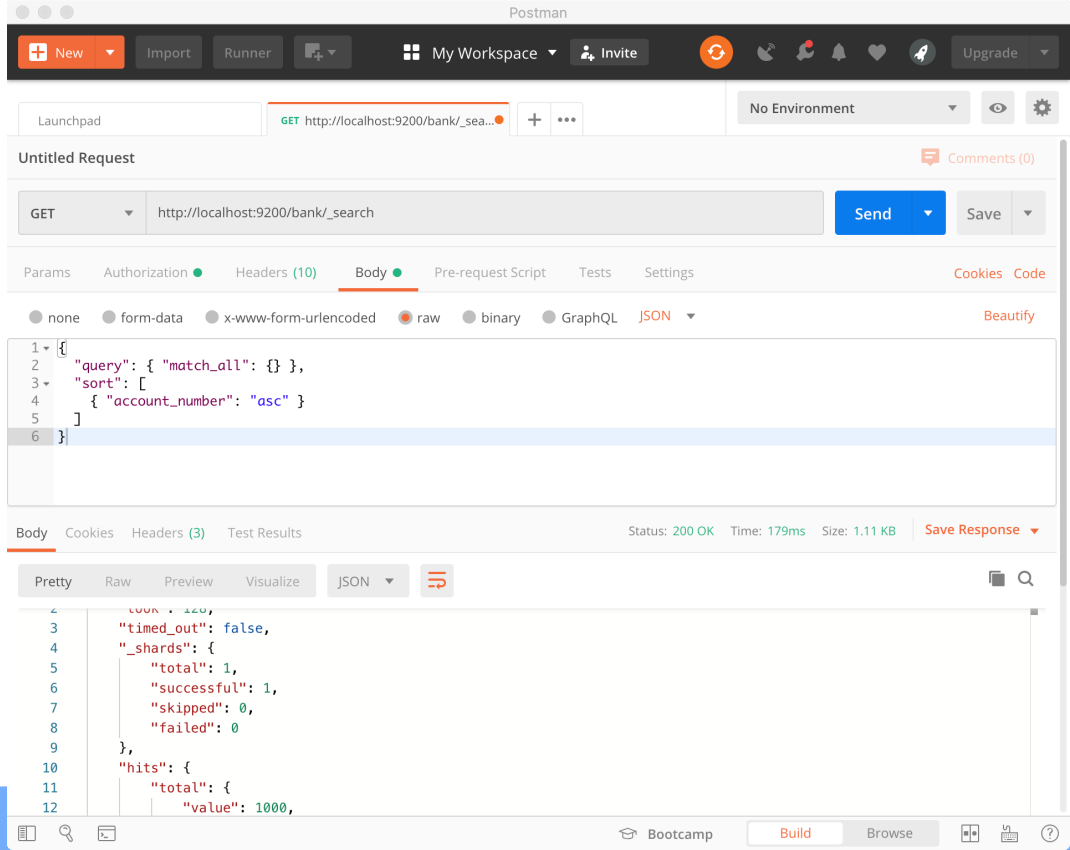
```
POST /customer/_doc/_bulk?pretty
{"index":{"_id":"1"}}
{"name": "John Doe" }
{"index":{"_id":"2"}}
{"name": "Jane Doe" }
```

```
POST /customer/_doc/_bulk?pretty
{"update":{"_id":"1"}}
{"doc": { "name": "John Doe becomes Jane Doe" } }
{"delete":{"_id":"2"}}
```

Elasticsearch

- Start searching
  - http://localhost:9200/bank/_search

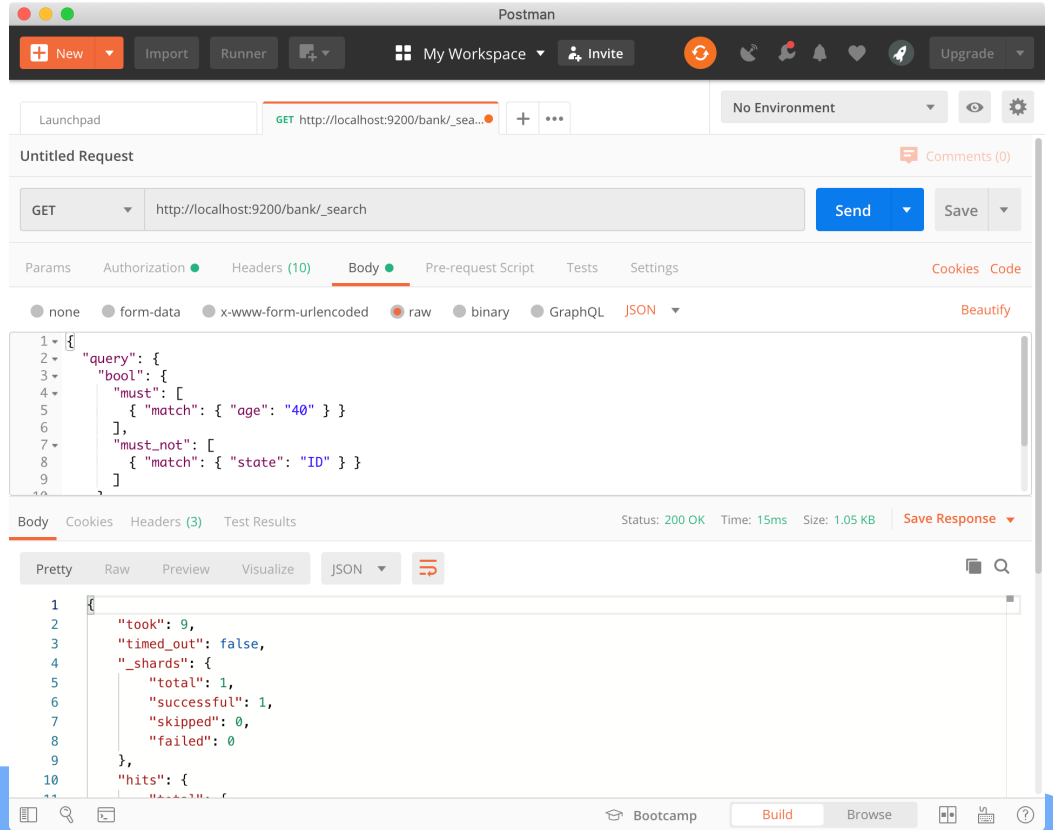
```
{
 "query": { "match_all": {} },
 "sort": [
  { "account_number": "asc" }
 ]
}
```



Elasticsearch

- Start searching
  - http://localhost:9200/bank/_search

```
{
 "query": {
  "bool": {
   "must": [
    { "match": { "age": "40" } }
   ],
   "must_not": [
    { "match": { "state": "ID" } }
   ]
  }
 }
}
```



作业四(第一部分)

- 请你在大二开发的E-Book系统的基础上，完成下列任务：
  1. 在你的项目中增加基于Solr或Lucene的针对书籍简介的全文搜索功能，用户可以在搜索界面输入搜索关键词，你可以通过全文搜索引擎找到书籍简介中包含该关键词的书籍列表。为了实现起来方便，你可以自己设计文本文件格式来存储书籍简介信息。例如，你可以将所有书籍的简介信息存储成为JSON对象，包含书的ID和简介文本，每行存储一本书的JSON对象。

  – 请将你编写的相关代码整体压缩后上传，请勿压缩整个工程提交。

- 评分标准：
  1. 能够正确实现上述搜索功能，并且集成到工程中(3分)

References

- Apache Lucene
  - http://lucene.apache.org/
- Lucene in Action
  - By Otis Gospodnetic & Erik Hatcher
  - MANNING Publishing
- Lucene 8.4.1 demo API
  - https://lucene.apache.org/core/8_4_1/demo/overview-summary.html#overview.description
- Apache Solr
  - https://lucene.apache.org/solr/
- Solr Tutorial
  - https://lucene.apache.org/solr/guide/8_4/solr-tutorial.html#launch-solr-in-solrcloud-mode
- Using SolrJ
  - https://lucene.apache.org/solr/guide/8_4/using-solrj.html
- Getting started with Elasticsearch
  - https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html
- Java API [7.6] » Document APIs
  - https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/java-docs.html

Thank You!