Architecture of Enterprise Applications 18 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

Agenda

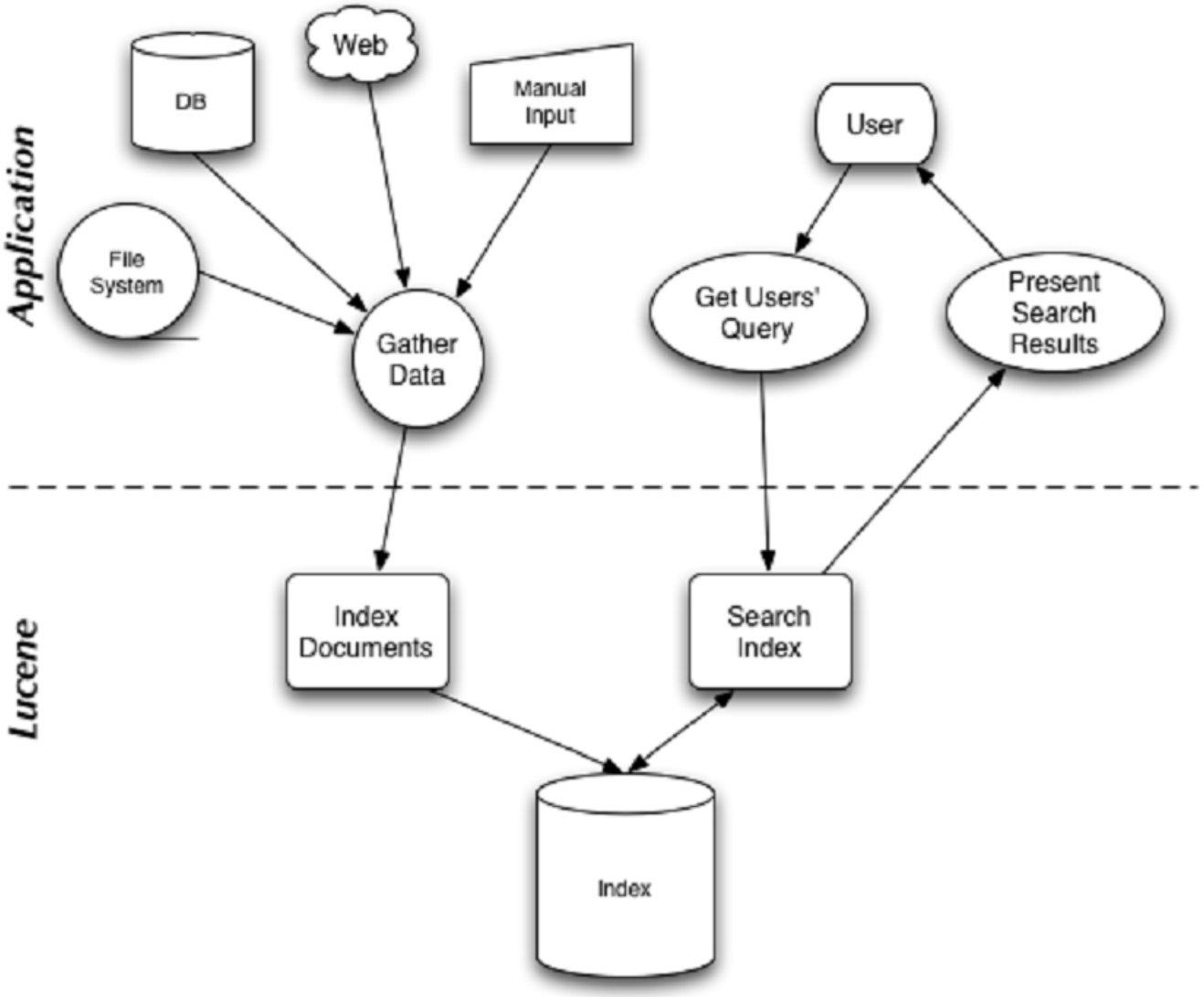
- Searching
  - Apache Lucene
  - Apache Solr

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

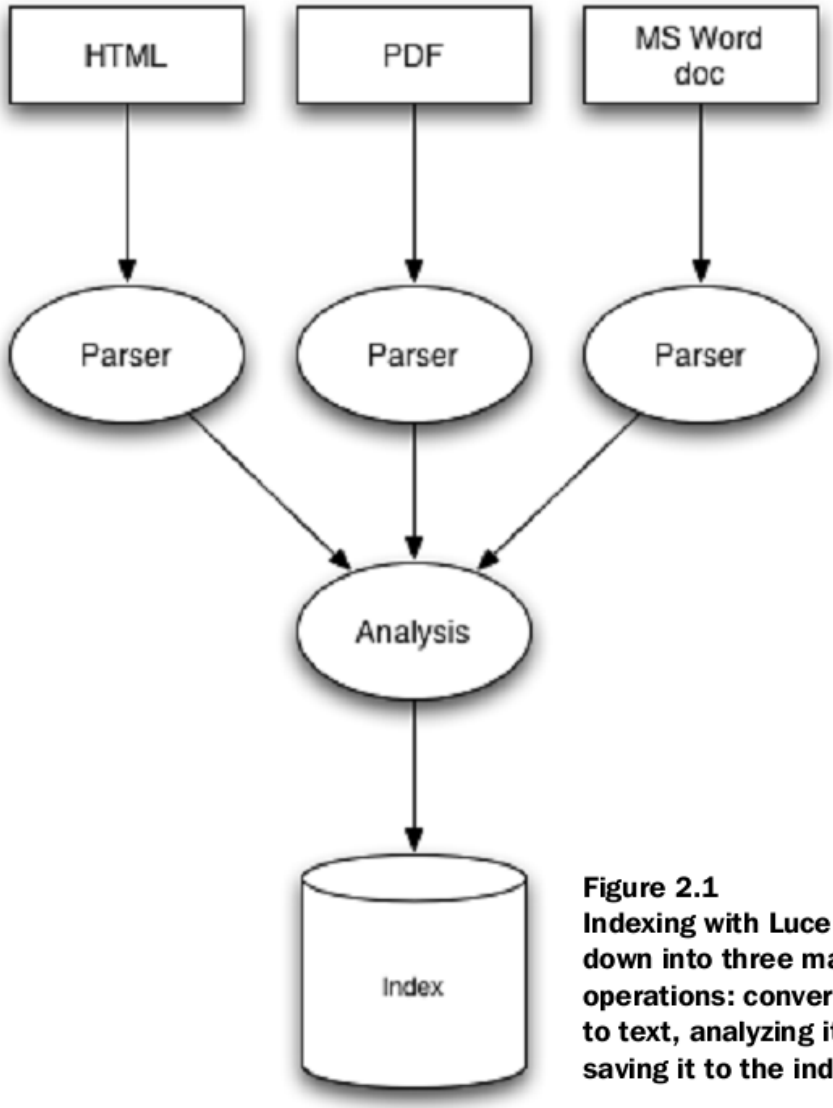


Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

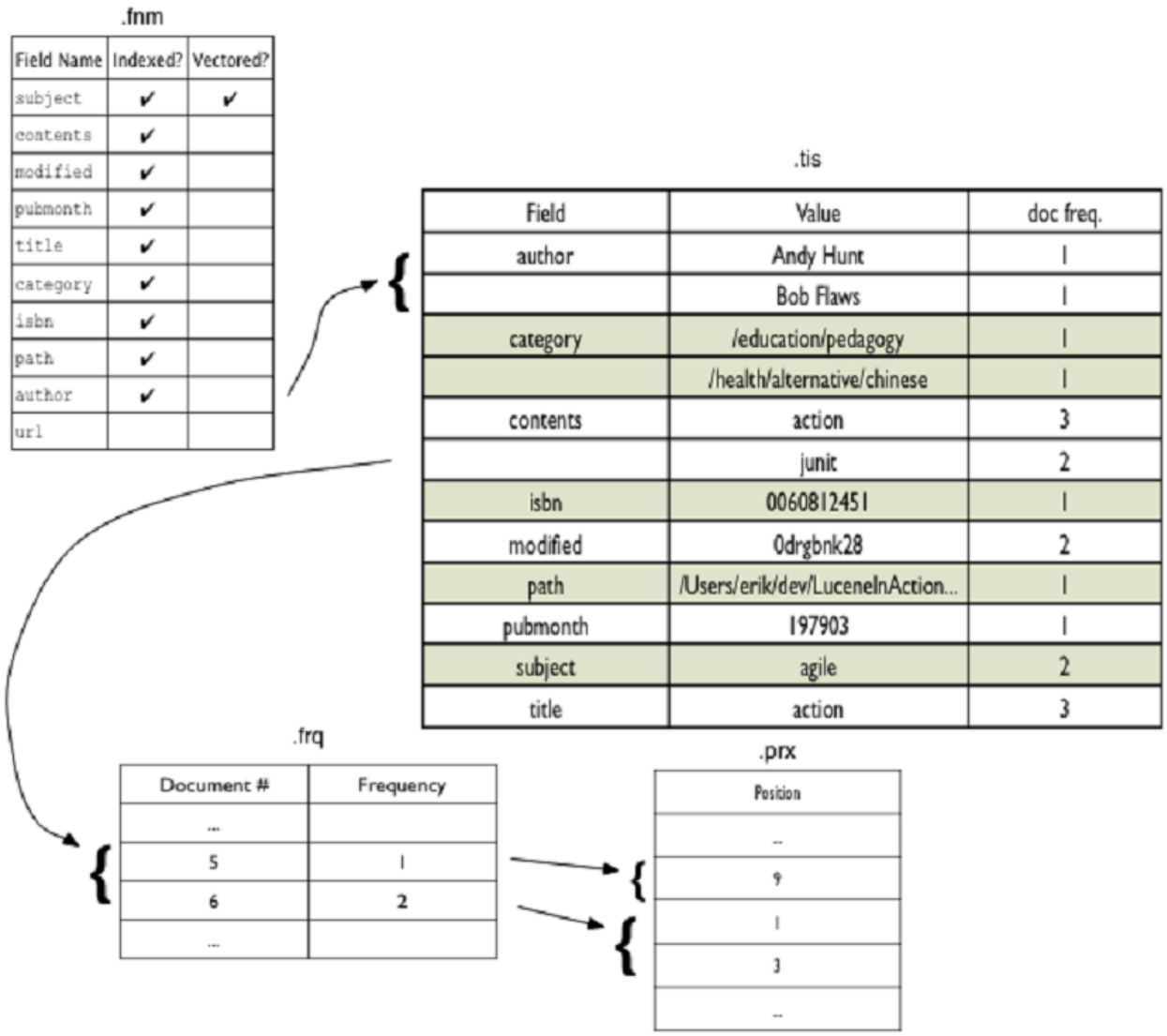
indexing

- 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



.fnm

Field Name	Indexed?	Vectored?
subject	✔	✔
contents	✔	
modified	✔	
pubmonth	✔	
title	✔	
category	✔	
isbn	✔	
path	✔	
author	✔	
url		

.tis

Field	Value	doc freq.
author	Andy Hunt	1
	Bob Flaws	1
category	/education/pedagogy	1
	/health/alternative/chinese	1
contents	action	3
	junit	2
isbn	0060812451	1
modified	0drgbnk28	2
path	/Users/erik/dev/LuceneInAction...	1
pubmonth	197903	1
subject	agile	2
title	action	3

.frq

Document #	Frequency
...	
5	1
6	2
...	

.prx

Position
...
9
1
3
...

- 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))); <-- Index file content

    doc.add(Field.Keyword("filename", f.getCanonicalPath())); <-- Index file name
    writer.addDocument(doc); <-- 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

REin
REliable, INtelligent & Scalable Systems

```
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 Documentdeletion 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 Documentmust 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

REliable, INtelligent & Scalable Systems

```
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

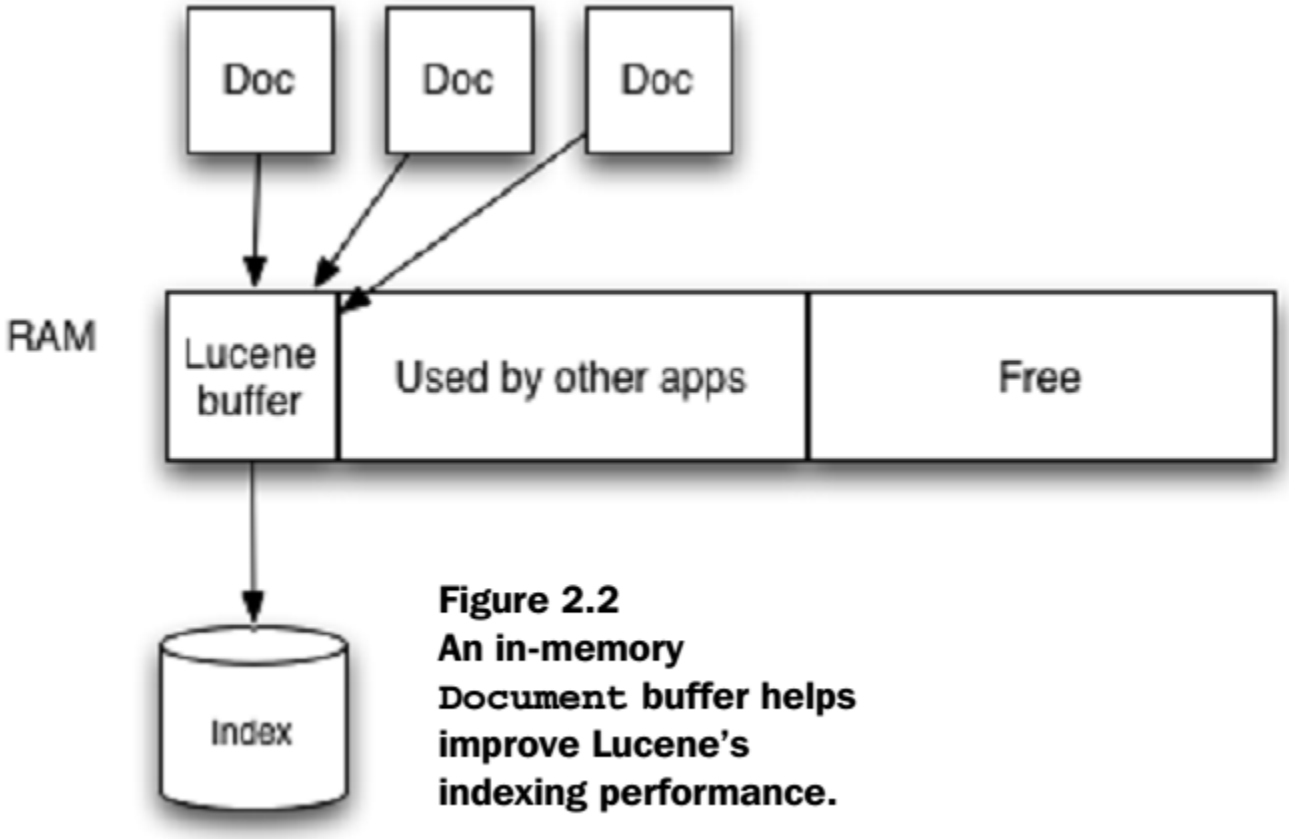


Figure 2.2
An in-memory `Document` buffer helps improve Lucene's indexing performance.

Parallelizing indexing by working with multiple indexes

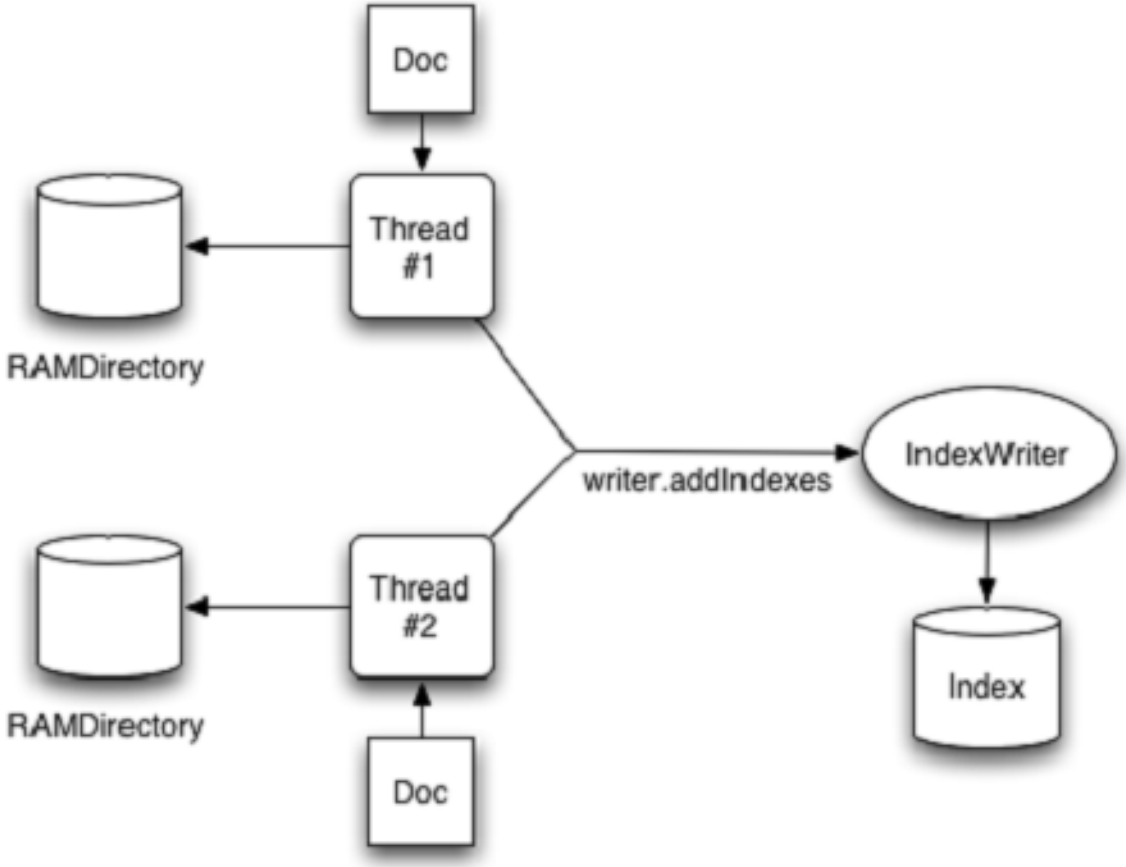


Figure 2.3 A multithreaded application that uses multiple `RAMDirectory` instances for parallel indexing.

Concurrency

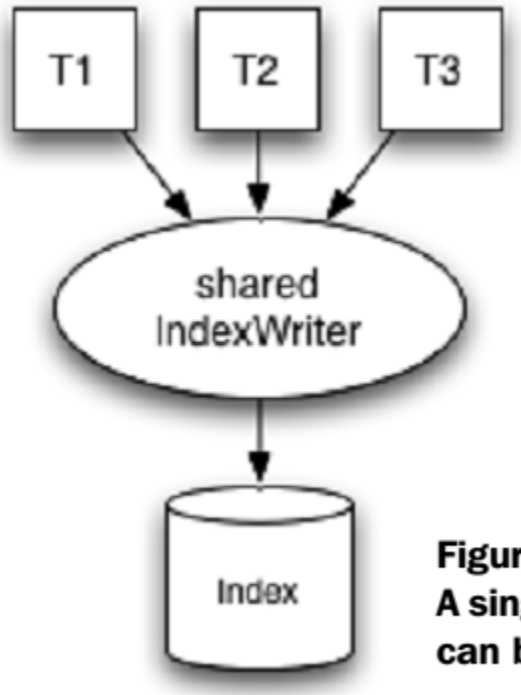


Figure 2.7
A single `IndexWriter` or `IndexReader` can be shared by multiple threads.

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

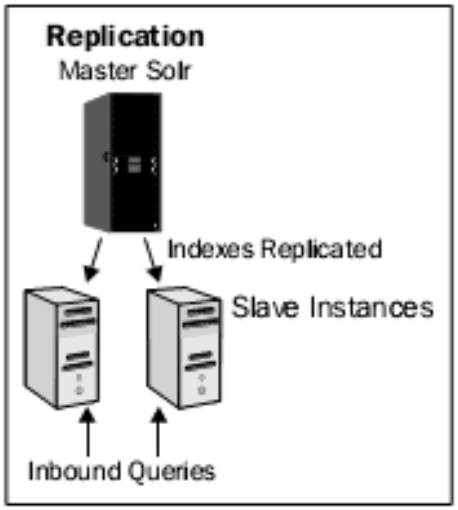
Factor	Description
`tf(t in d)`	Term frequency factor for the term (`t`) in the document (`d`).
`idf(t)`	Inverse document frequency of the term.
`boost(t.field in d)`	Field boost, as set during indexing.
`lengthNorm(t.field in d)`	Normalization value of a field, given the number of terms within the field. 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.

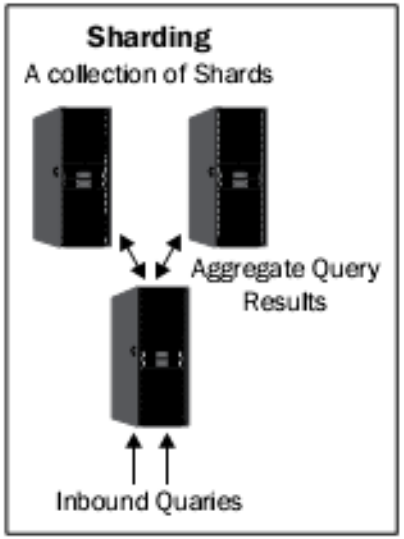
Apache Solr

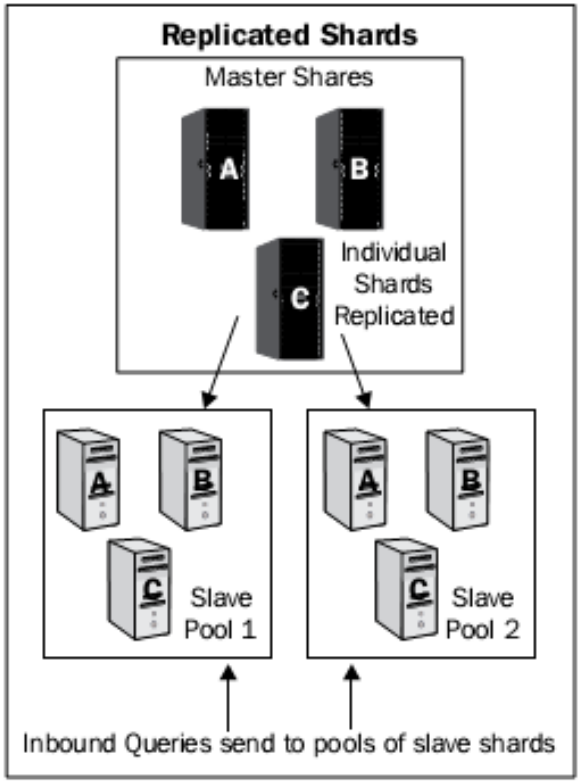
- Apache Solr is an enterprise search server based on Lucene.

- Some of Solr's most notable features beyond Lucene are:
  - A server that communicates over HTTP via XML and JSON data formats.
  - Configuration files, most notably for the index's schema, which defines the fields and configuration of their text analysis.
  - Several caches for faster search responses.
  - A web-based administrative interface including:
    - Runtime search and cache performance statistics.
    - A schema browser with index statistics on each field.
    - A diagnostic tool for debugging text analysis.
  - Faceting of search results.
  - A query parser called dismaxthat is more usable for parsing end user queries than Lucene's native query parser.
  - Geospatial search for filtering and sorting by distance.
  - Distributed-search support and index replication for scaling Solr.
  - Solritas: A sample generic web search UI demonstrating many of Solr's search features.

Apache Solr


Replication
Master Solr
Indexes Replicated
Slave Instances
Inbound Queries


Sharding
A collection of Shards
Aggregate Query Results
Inbound Quaries


Replicated Shards
Master Shares
A
B
C
Individual Shards Replicated
A
B
C
Slave Pool 1
A
B
C
Slave Pool 2
Inbound Queries send to pools of slave shards

Project

- Requirement
  - Try Lucene or Solr if you can.
  - Use Lucene or Solr to do the full-text searching in all the blogs to find the desired topics.

References

- Apache Lucene
  - http://lucene.apache.org/
- Lucene in Action
  - By Otis Gospodnetic & Erik Hatcher
  - MANNING Publishing
- Solr: Ultra-fast Lucene-based Search Server
  - http://lucene.apache.org/solr/
- Apache Solr 3 Enterprise Search Server
  - By David Smiley & Eric Pugh
  - PACKT Publishing

Thank You!