

Architecture of Enterprise Applications 20 Searching

Haopeng Chen

***RE**liable, **IN**telligent and **Scalable** Systems Group (**REINS**)*

Shanghai Jiao Tong University

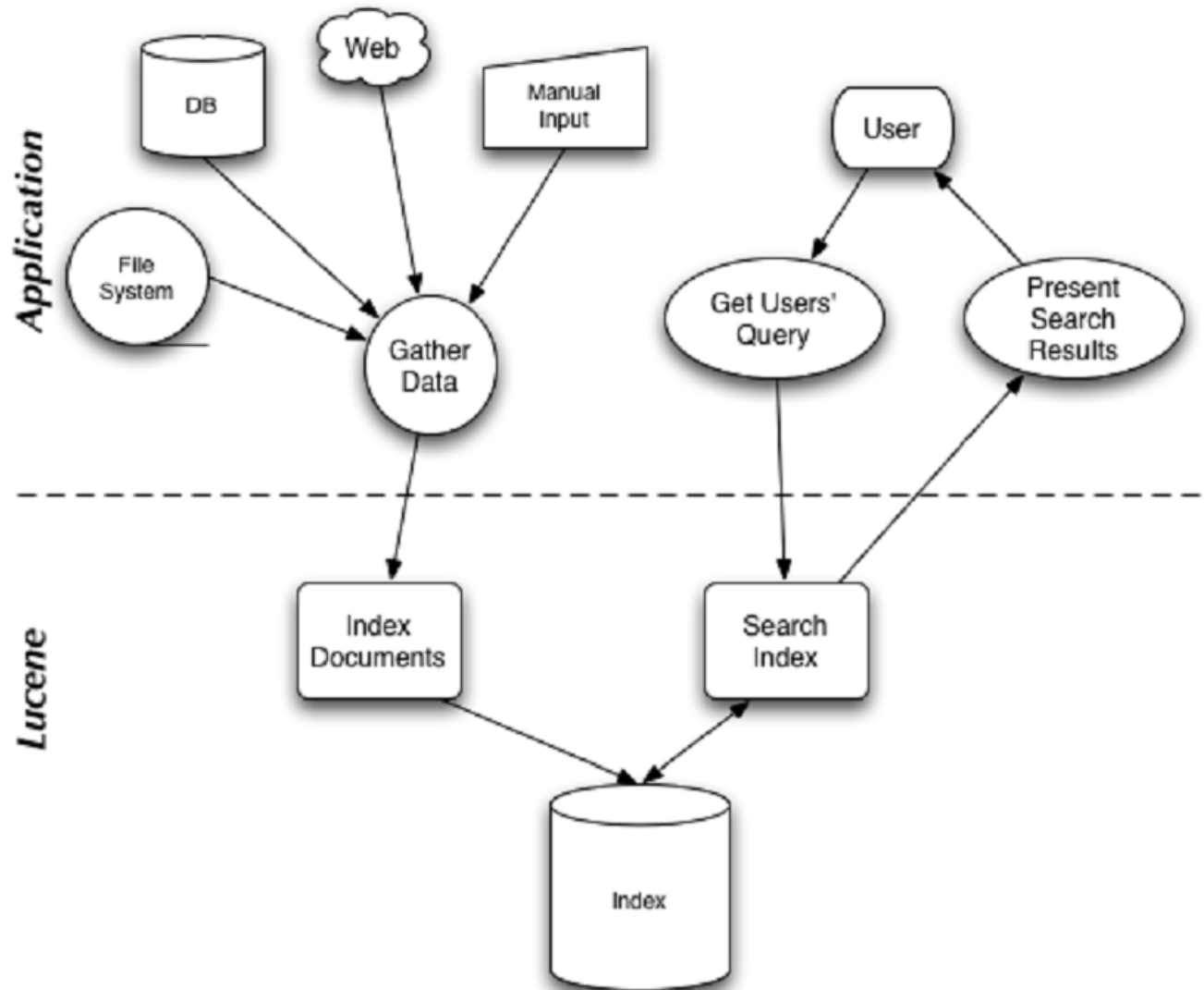
Shanghai, China

<http://reins.se.sjtu.edu.cn/~chenhp>

e-mail: chen-hp@sjtu.edu.cn

- Searching
 - Apache Lucene
 - Apache Solr

- 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.



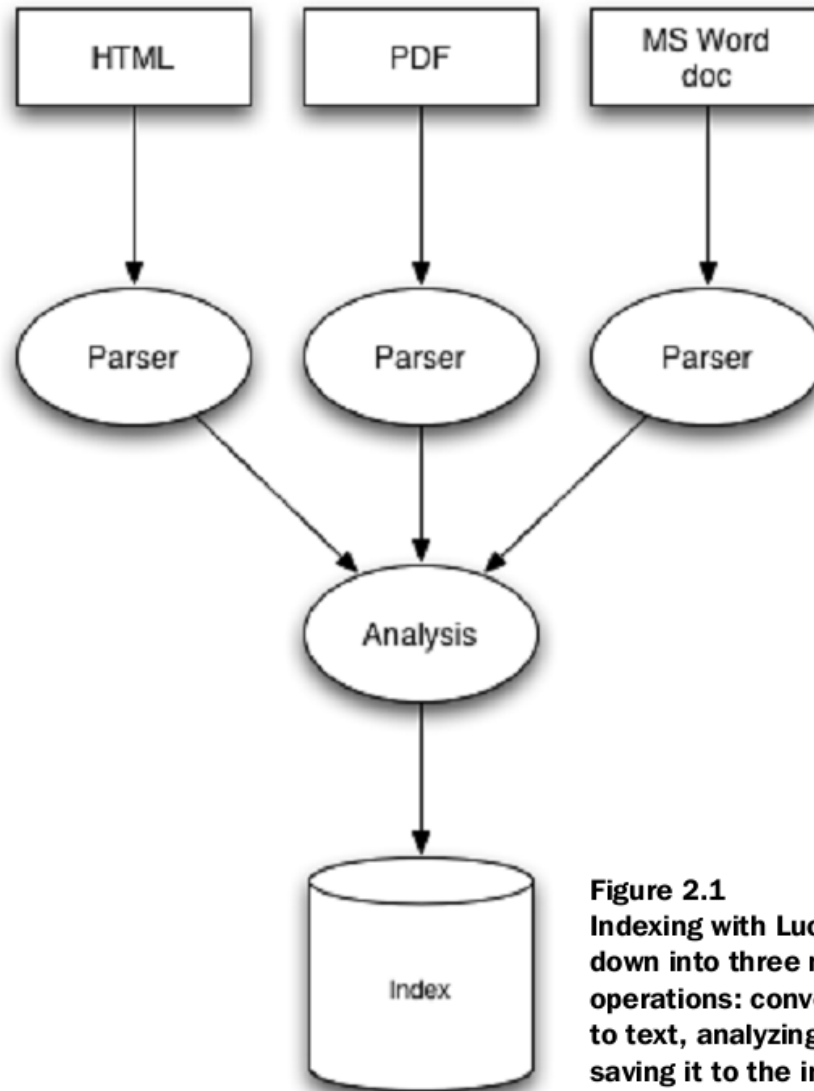
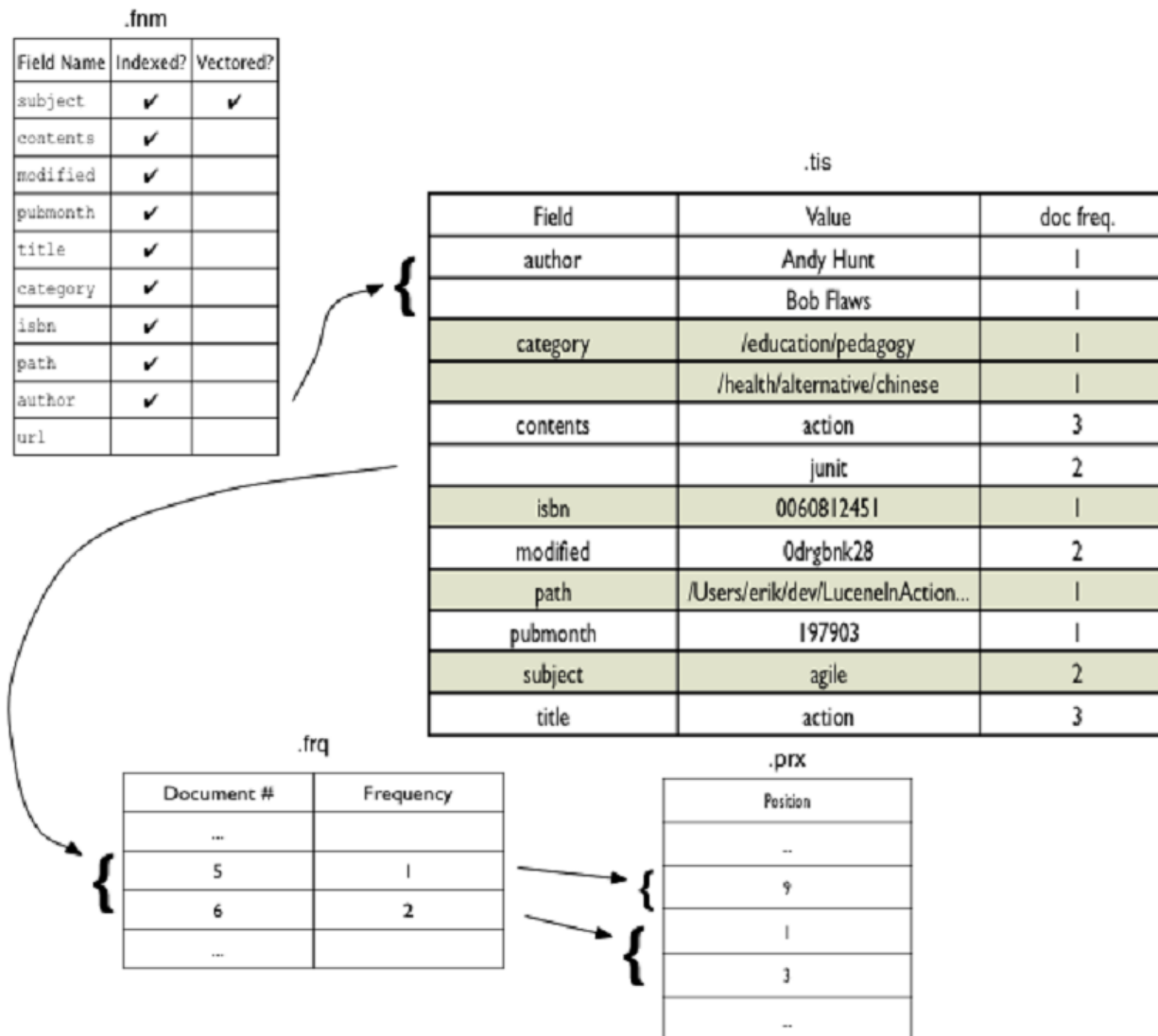


Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

- 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 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.

- 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.

```
/**
 * 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]); ← Create Lucene index in this directory
        File dataDir = new File(args[1]); ← Index files in this directory

        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");
    }
}
```

```
// 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);
```



Create Lucene index

```
indexDirectory(writer, dataDir);
```

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



Close 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

```
// 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
```

```
    }  
}
```

```
% 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
```

```
/**
 * 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]); ← Index directory created by Indexer
        String q = args[1]; ← Query string

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

        search(indexDir, q);
    }
}
```



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

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

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

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

```
%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

- **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.

- **IndexSearcher**
 - IndexSearcher is to searching what IndexWriter is 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

```
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



REliable, INtelligent & Scalable Systems

```
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;}
}
```

- 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.

- **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**.

- **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



REliable, INtelligent & Scalable Systems

```
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();
    }
}
```

```
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();  
}
```

- 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.

- “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();
```

- 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



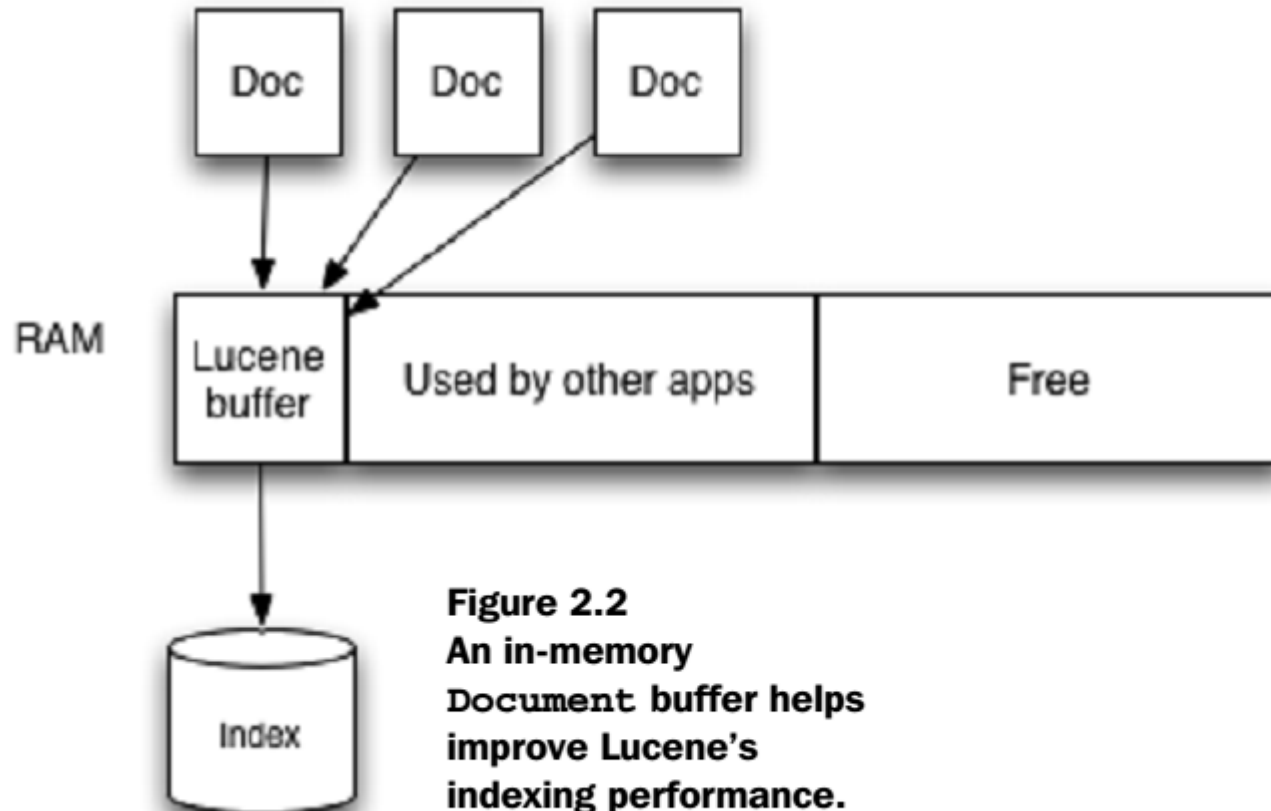
REliable, INtelligent & Scalable Systems

```
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);
```



Parallelizing indexing by working with multiple indexes

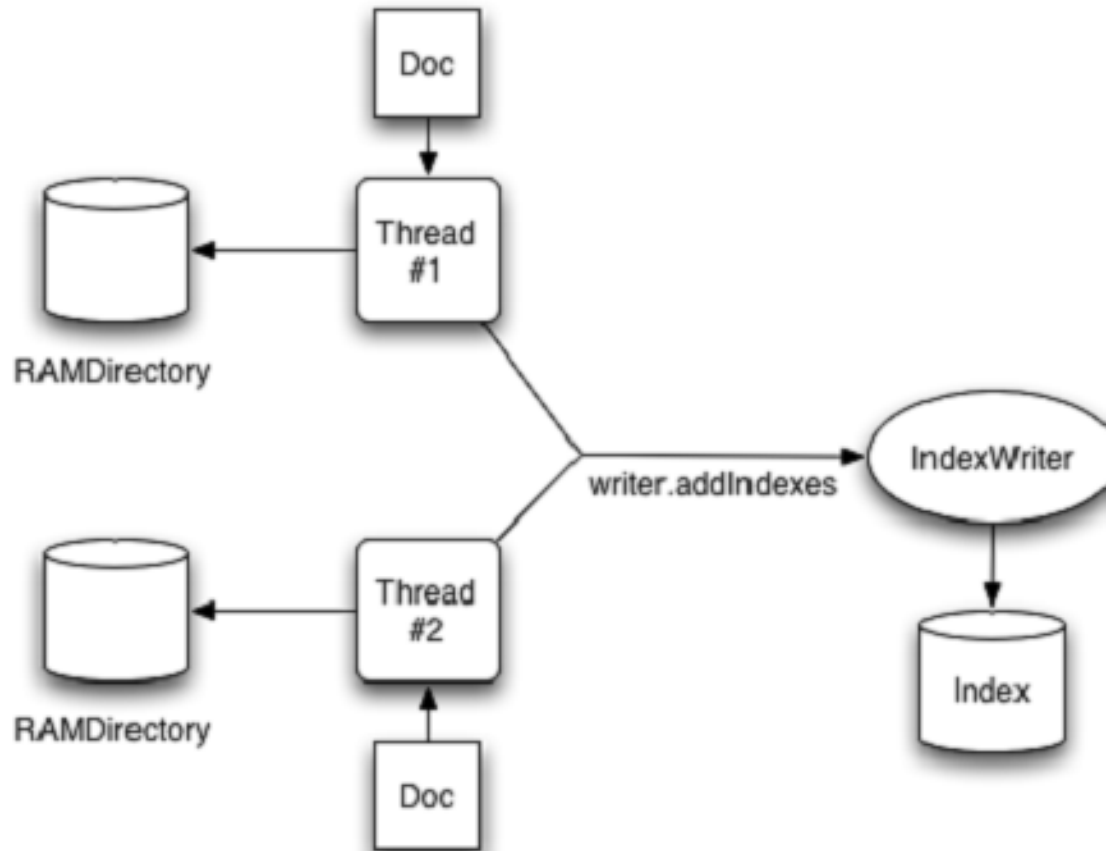


Figure 2.3 A multithreaded application that uses multiple RAMDirectory instances for parallel indexing.

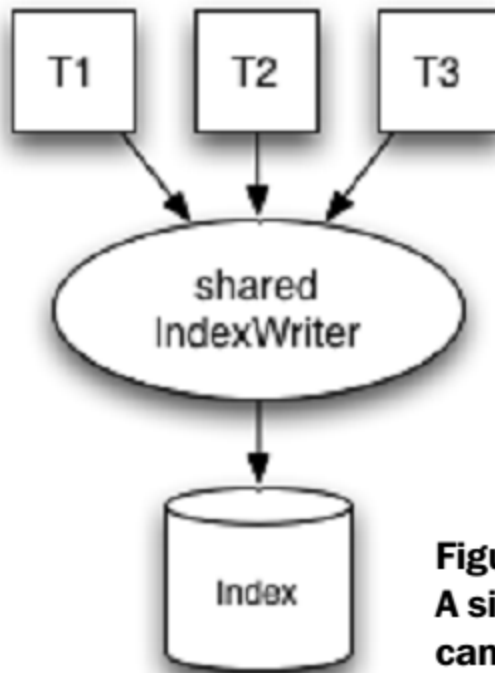


Figure 2.7
A single `IndexWriter` or `IndexReader` can be shared by multiple threads.

- 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



REliable, INtelligent & Scalable Systems

```
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);
```

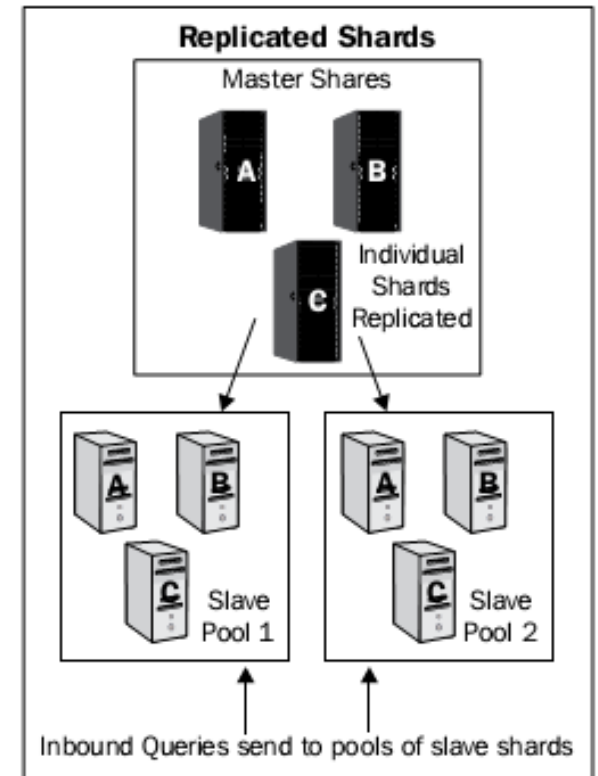
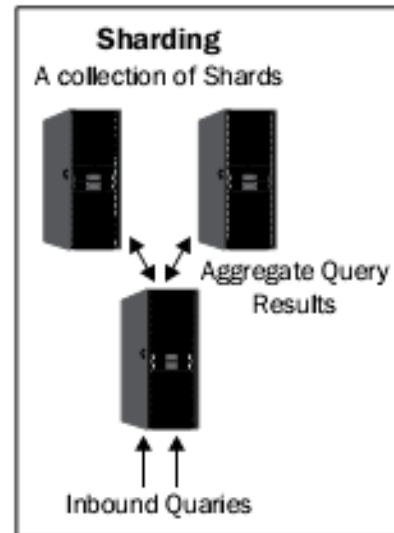
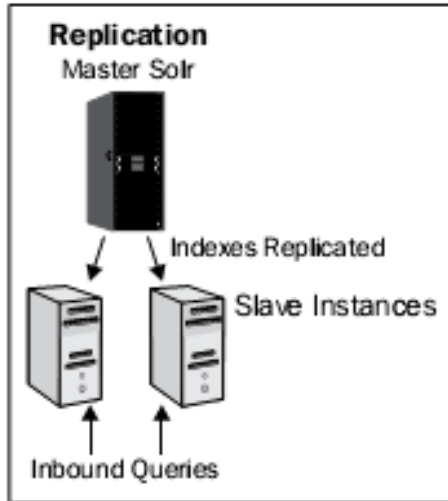
- The score is computed for each document (d) matching a specific.
 - This score is the raw score.
 - Scores returned from Hits aren't necessarily the raw score, however.
 - If the top-scoring document scores greater than 1.0, all scores are normalized from that score, such that all scores from Hits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

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

Table 3.5 Factors in the scoring formula

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

- 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 `dismax` that 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.



- 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.

- 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!