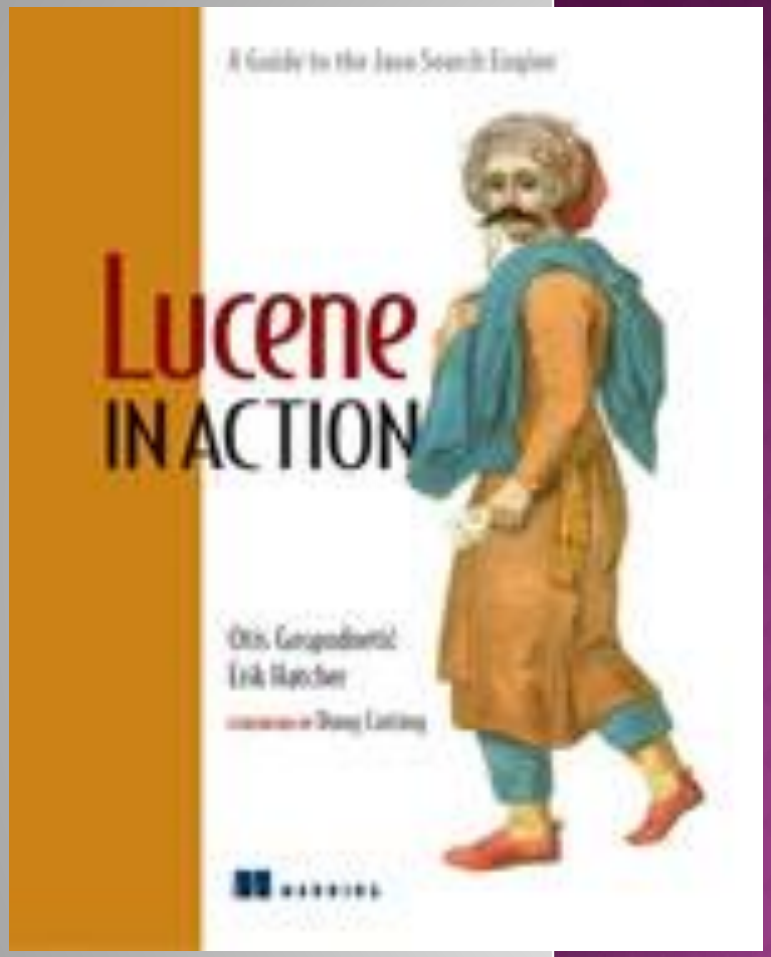


SEARCHING AND INDEXING BIG DATA

-By Jagadish Rouniyar

WHAT IS IT?

- Doug Cutting's grandmother's middle name
- A open source set of Java Classses
  - Search Engine/Document Classifier/Indexer
    - http://lucene.sourceforge.net/talks/pisa/
  - Developed by Doug Cutting 1996
    - Xerox/Apple/Excite/Nutch
    - Wrote several papers in IR

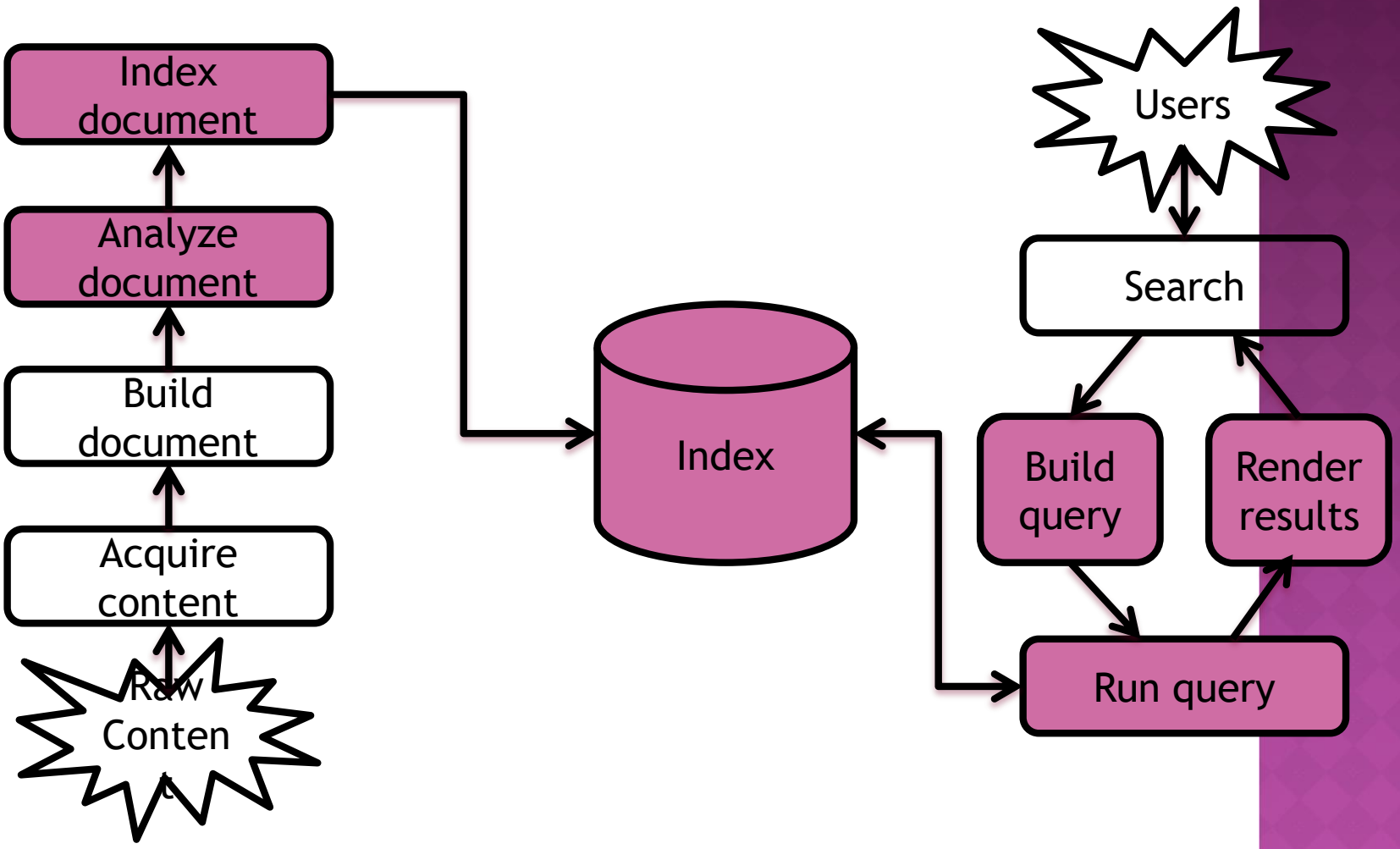
LUCENE

- Open source Java library for indexing and searching
  - Lets you add search to your application
  - High performance, scalable, full-text search library
  - Not a complete search system by itself
  - Written by Doug Cutting
- Used by LinkedIn, Twitter, …
  - …and many more (see http://wiki.apache.org/lucene-java/PoweredBy)
- Ports/integrations to other languages
  - C/C++, C#, Ruby, Perl, Python, PHP, …

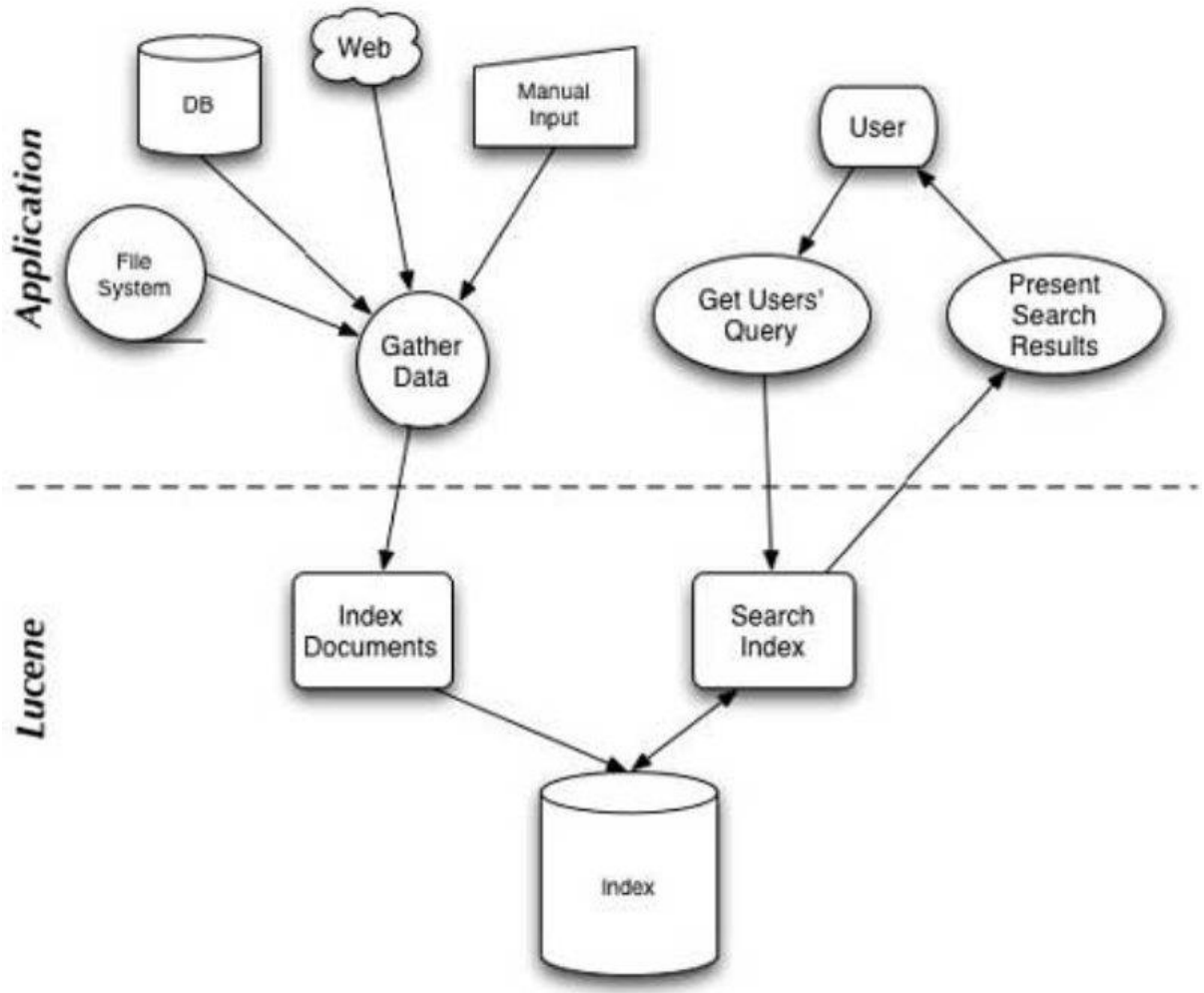
RESOURCES

- Lucene: http://lucene.apache.org/core/
- Lucene in Action: http://www.manning.com/hatcher3/
  - Code samples available for download
- Ant: http://ant.apache.org/
  - Java build system used by “Lucene in Action” code

LUCENE IN A SEARCH SYSTEM



LUCENE IMPLEMENTATION



Lucene Index: Concepts

The theory

Concepts

- Index: sequence of documents (a.k.a. Directory)
- Document: sequence of fields
- Field: named sequence of terms
- Term: a *text string (e.g., a word)*

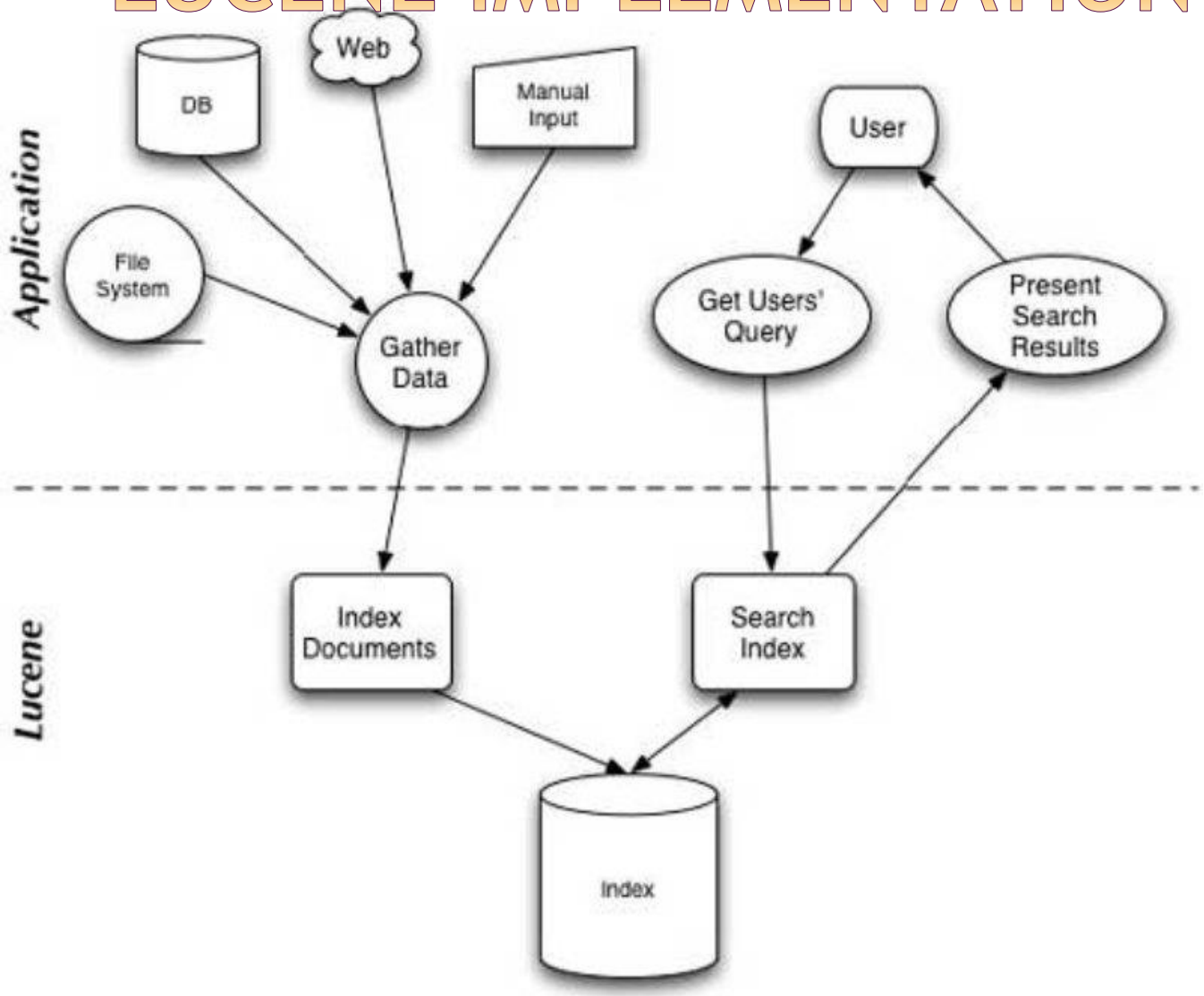
Statistics

- Term frequencies and positions

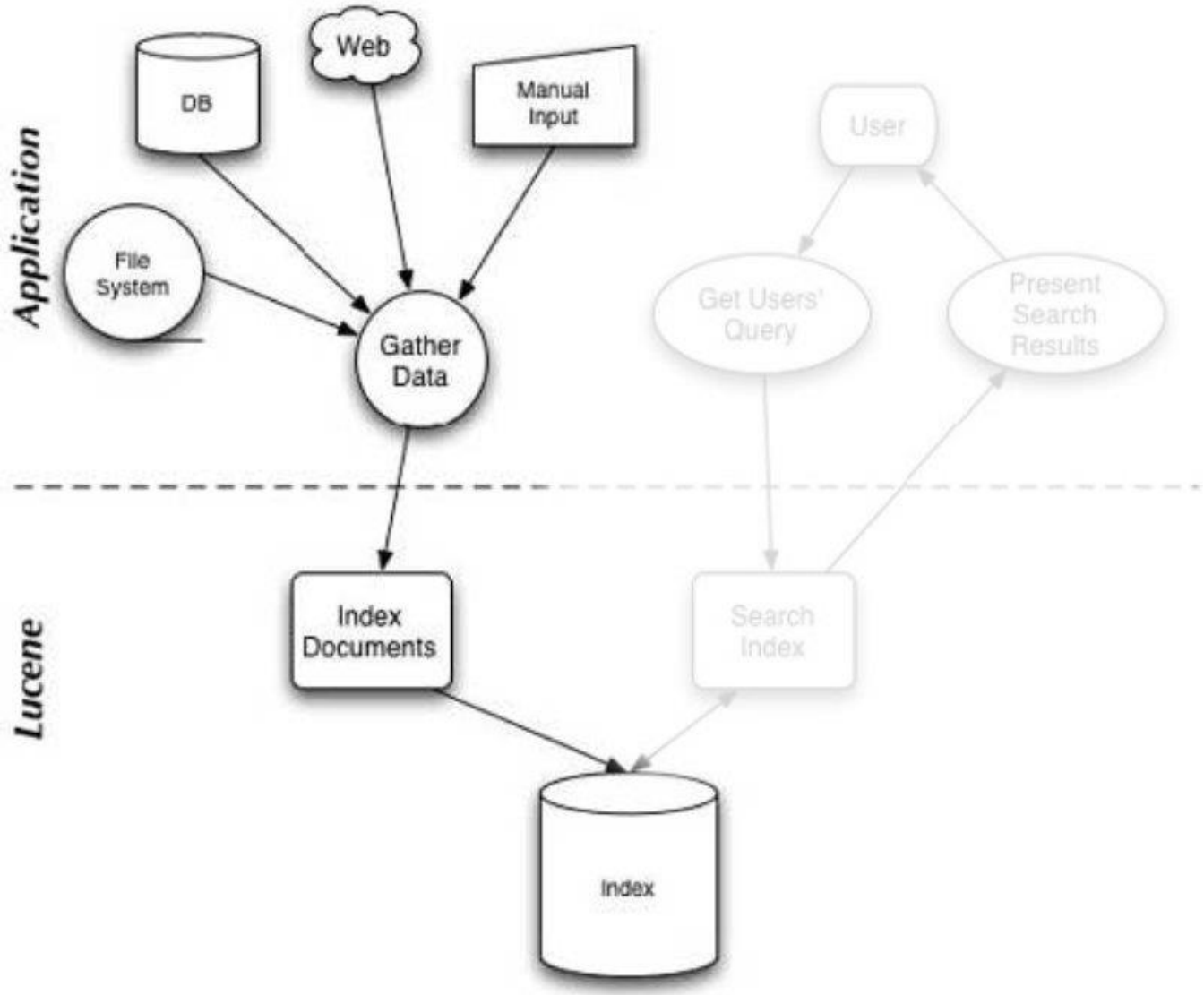
CORE INDEXING CLASSES

- `IndexWriter`
  - Central component that allows you to create a new index, open an existing one, and add, remove, or update documents in an index
- `Directory`
  - Abstract class that represents the location of an index
- `Analyzer`
  - Extracts tokens from a text stream

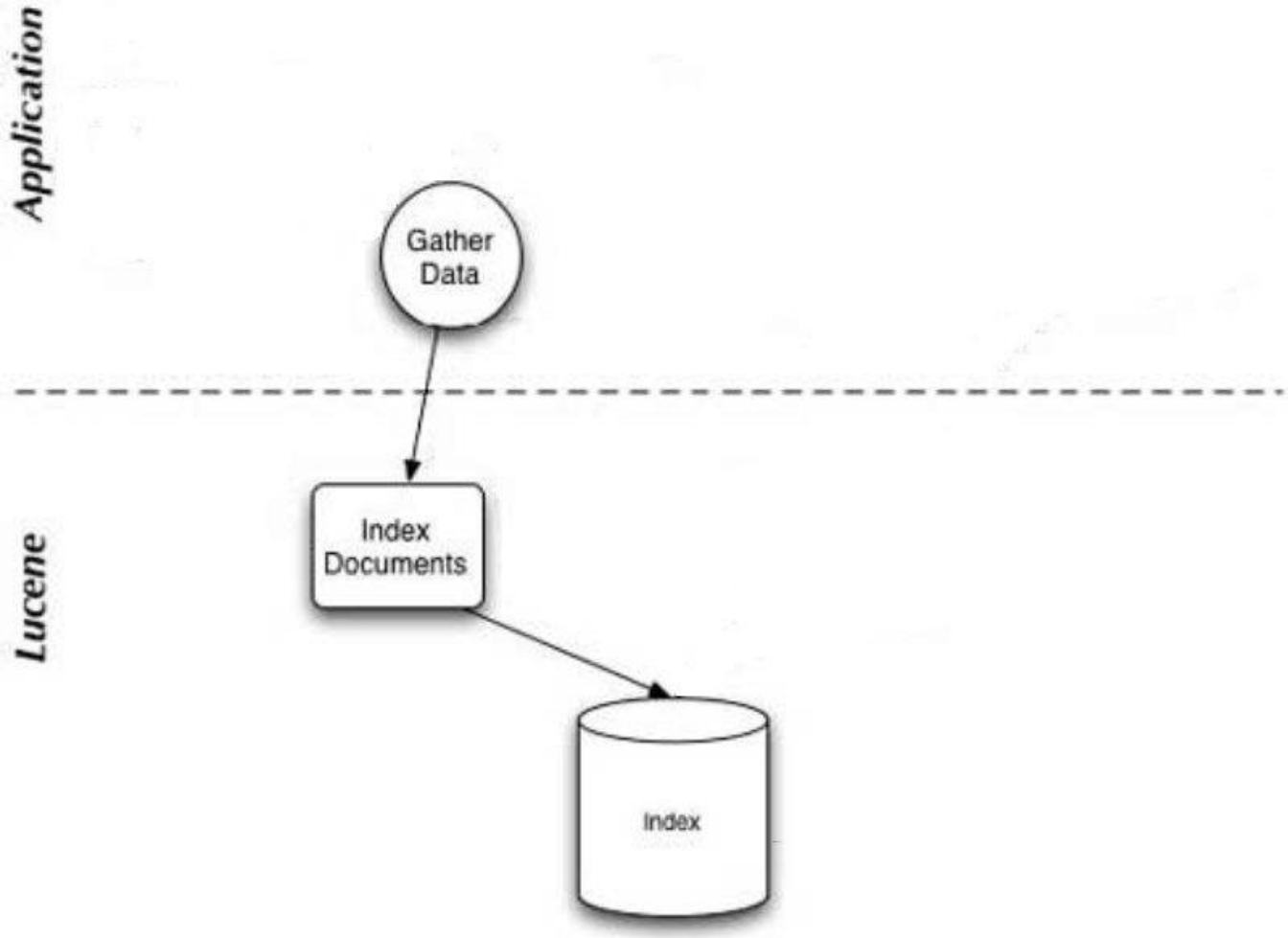
LUCENE IMPLEMENTATION



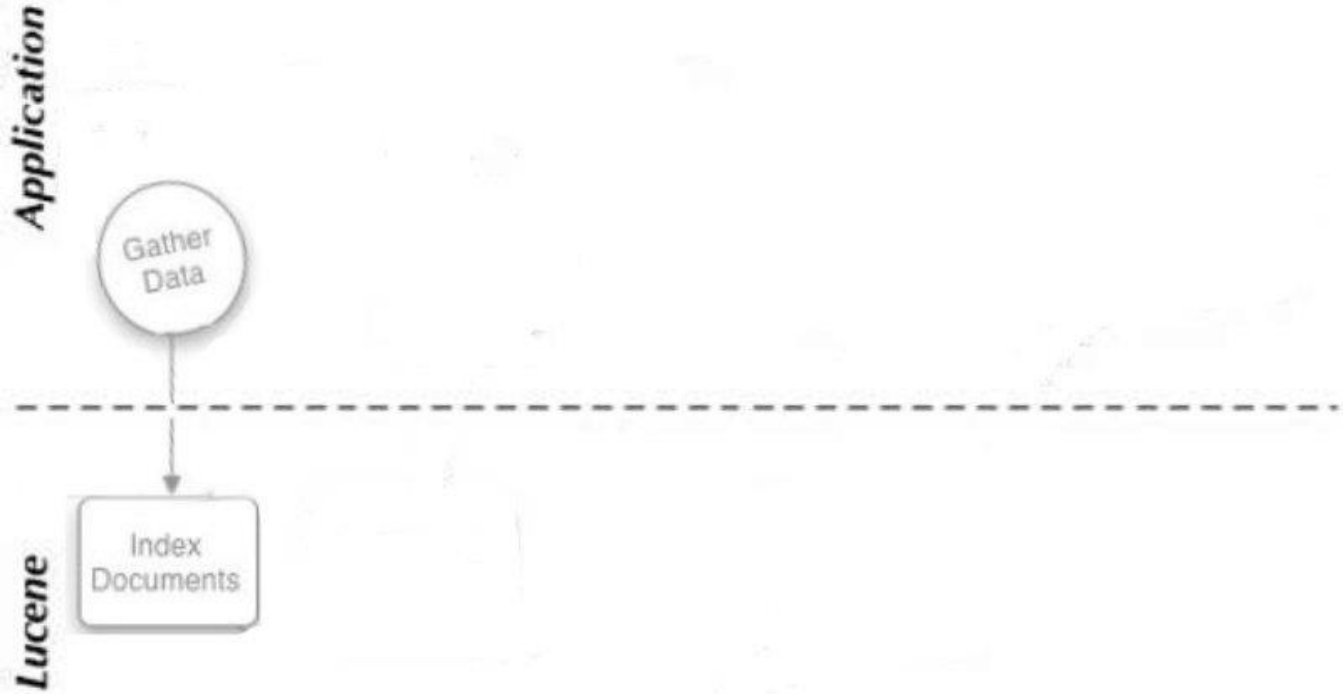
LUCENE IMPLEMENTATION: INDEXING



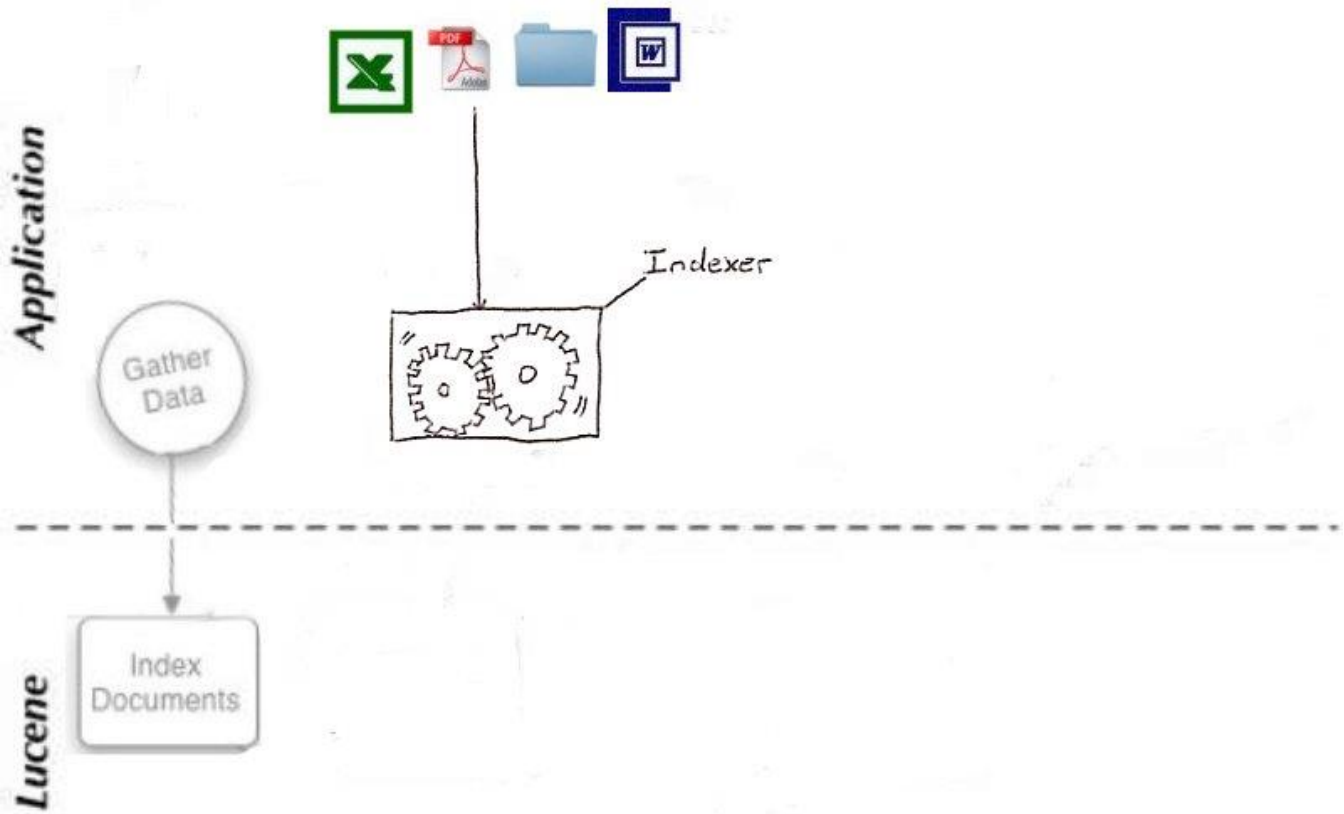
LUCENE IMPLEMENTATION: INDEXING



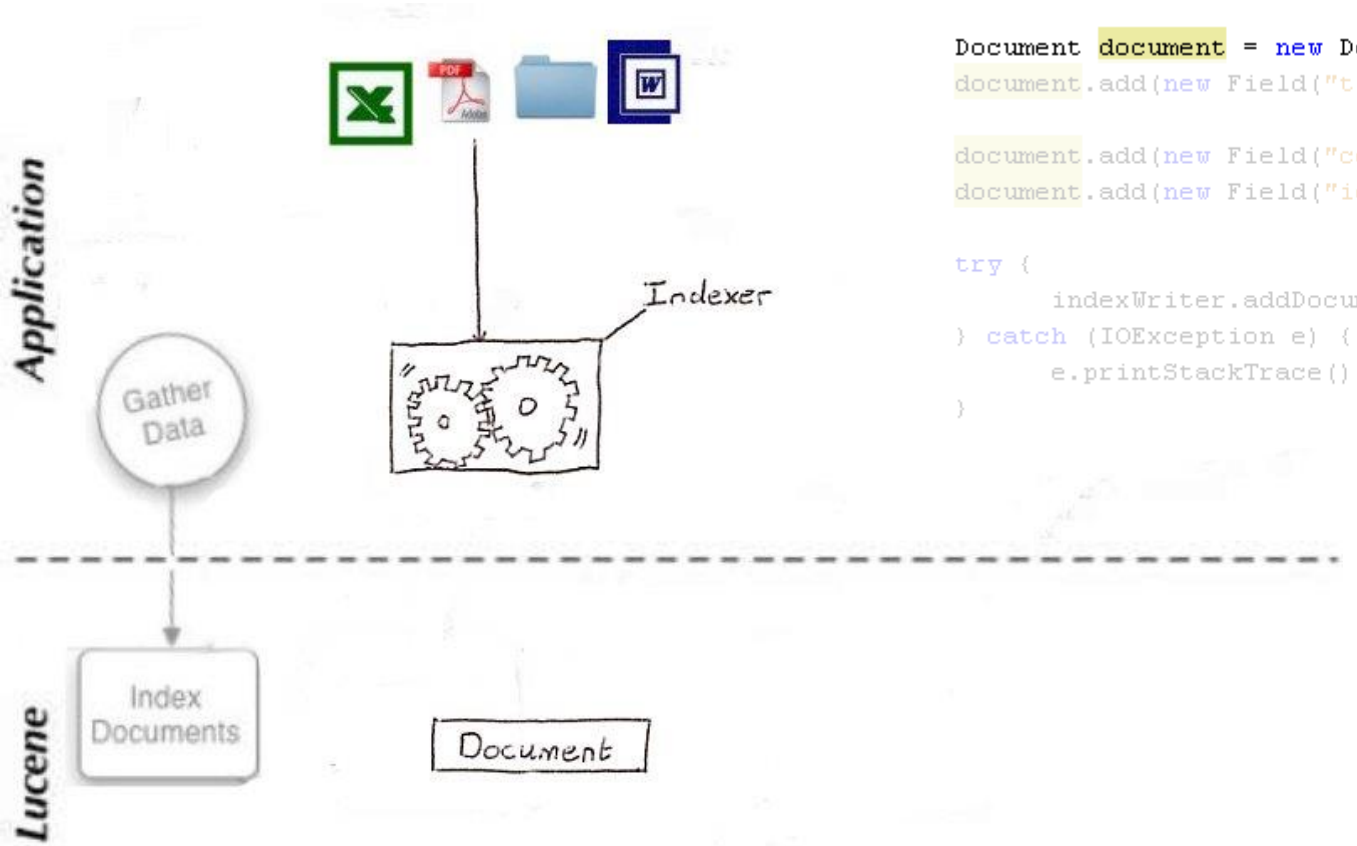
LUCENE IMPLEMENTATION: INDEXING



LUCENE INDEXING

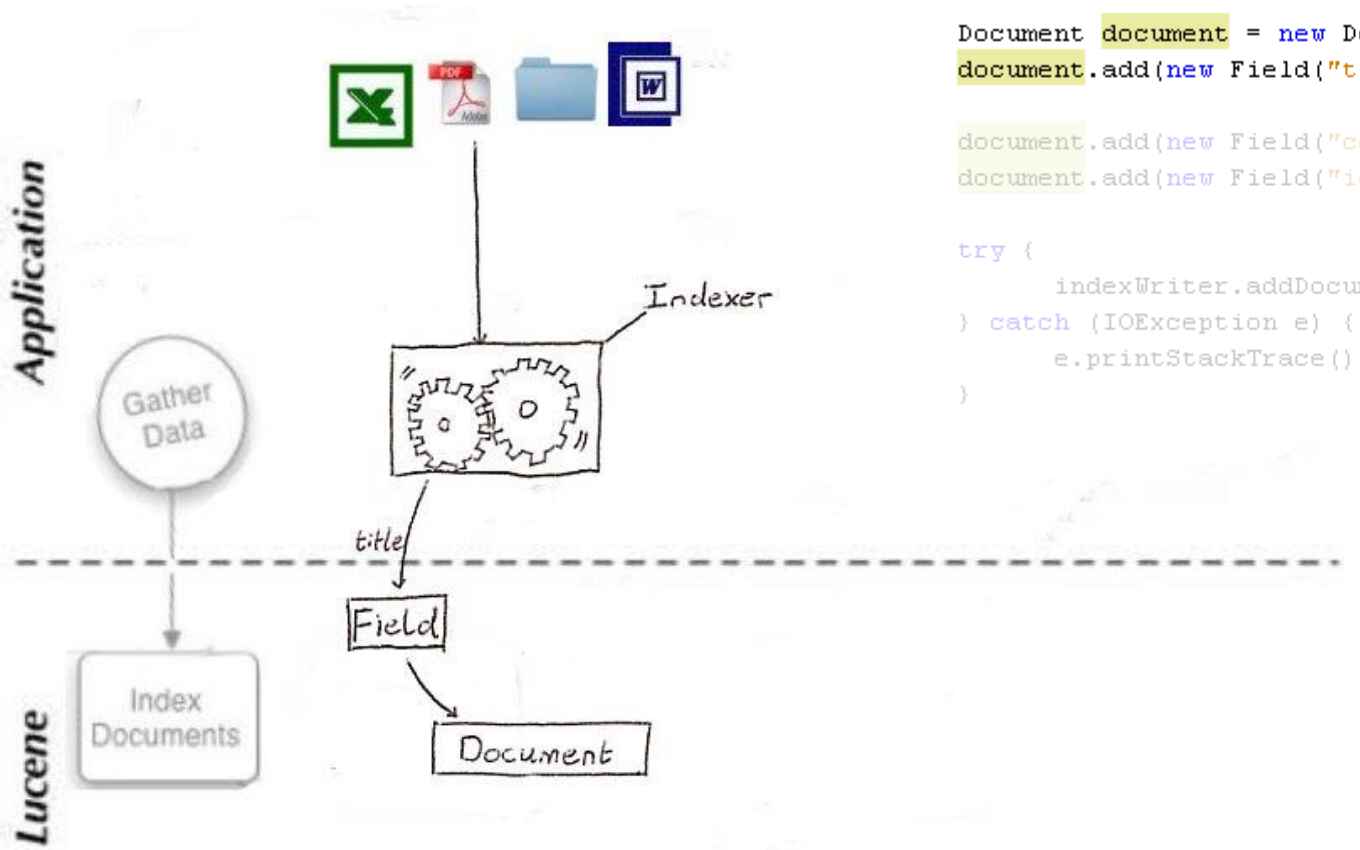


LUCENE INDEXING STEP 1 OF 5



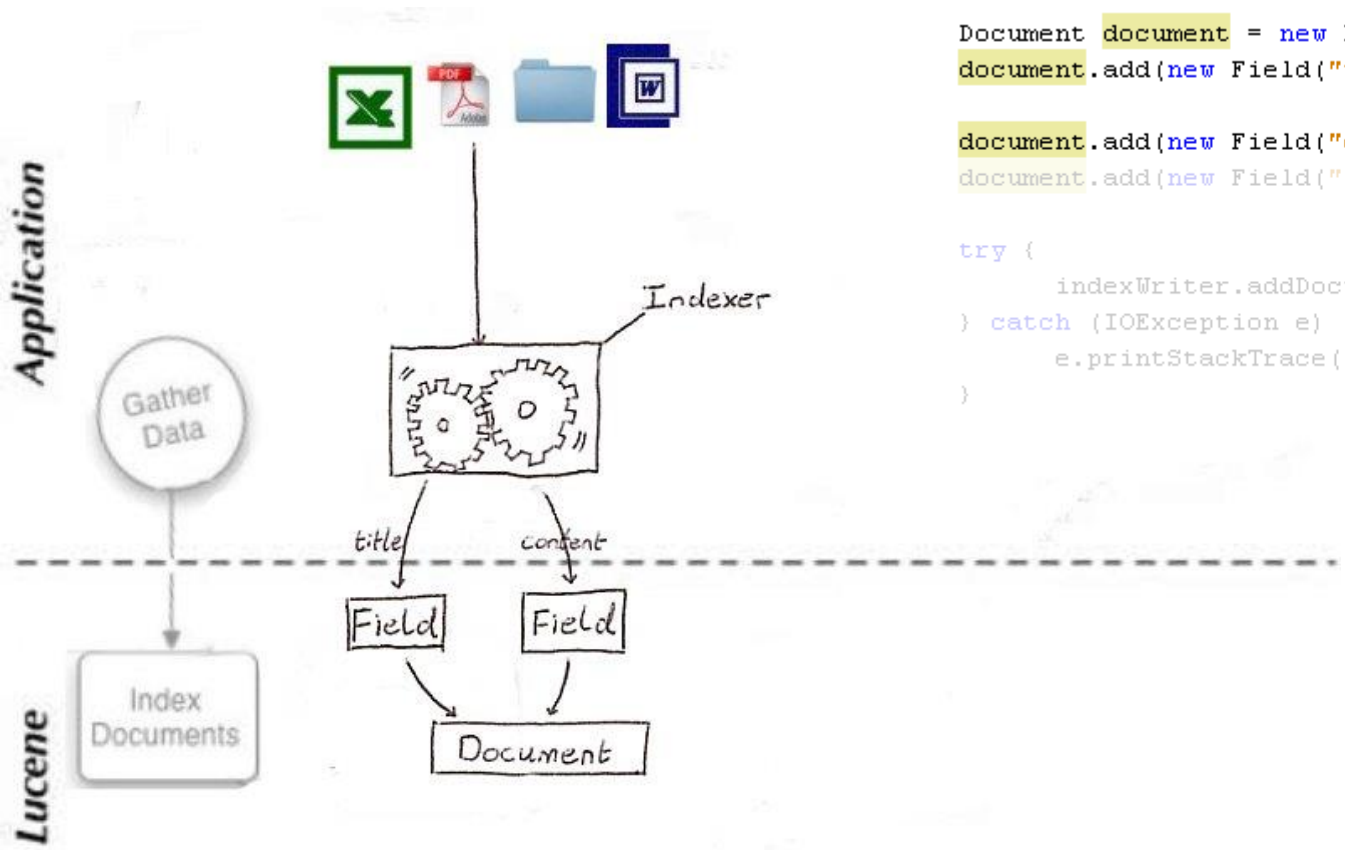

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));

try {
      indexWriter.addDocument(document);
} catch (IOException e) {
      e.printStackTrace();
}
```

LUCENE INDEXING STEP 2 OF 5



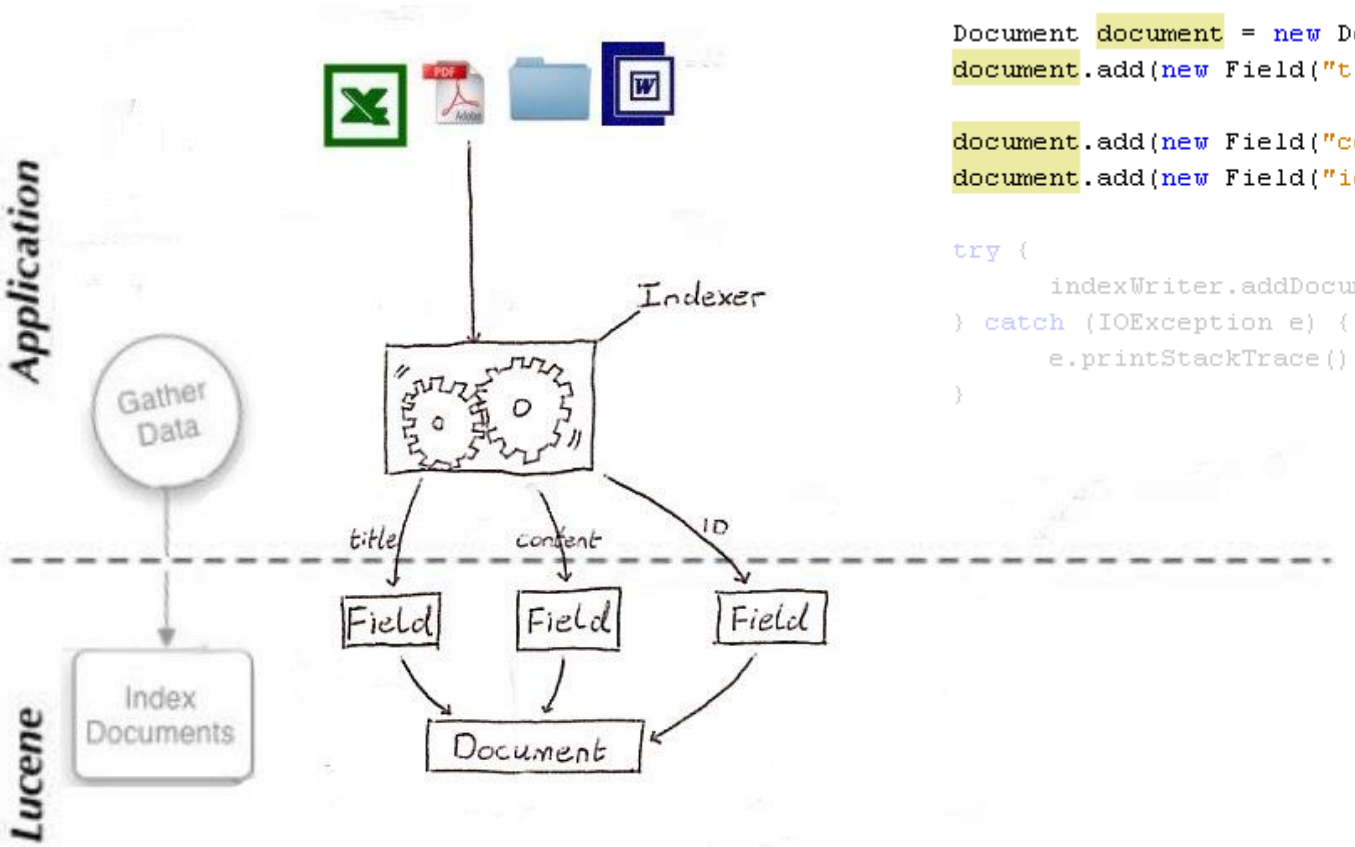

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));
try {
      indexWriter.addDocument(document);
} catch (IOException e) {
      e.printStackTrace();
}
```

LUCENE INDEXING STEP 3 OF 5



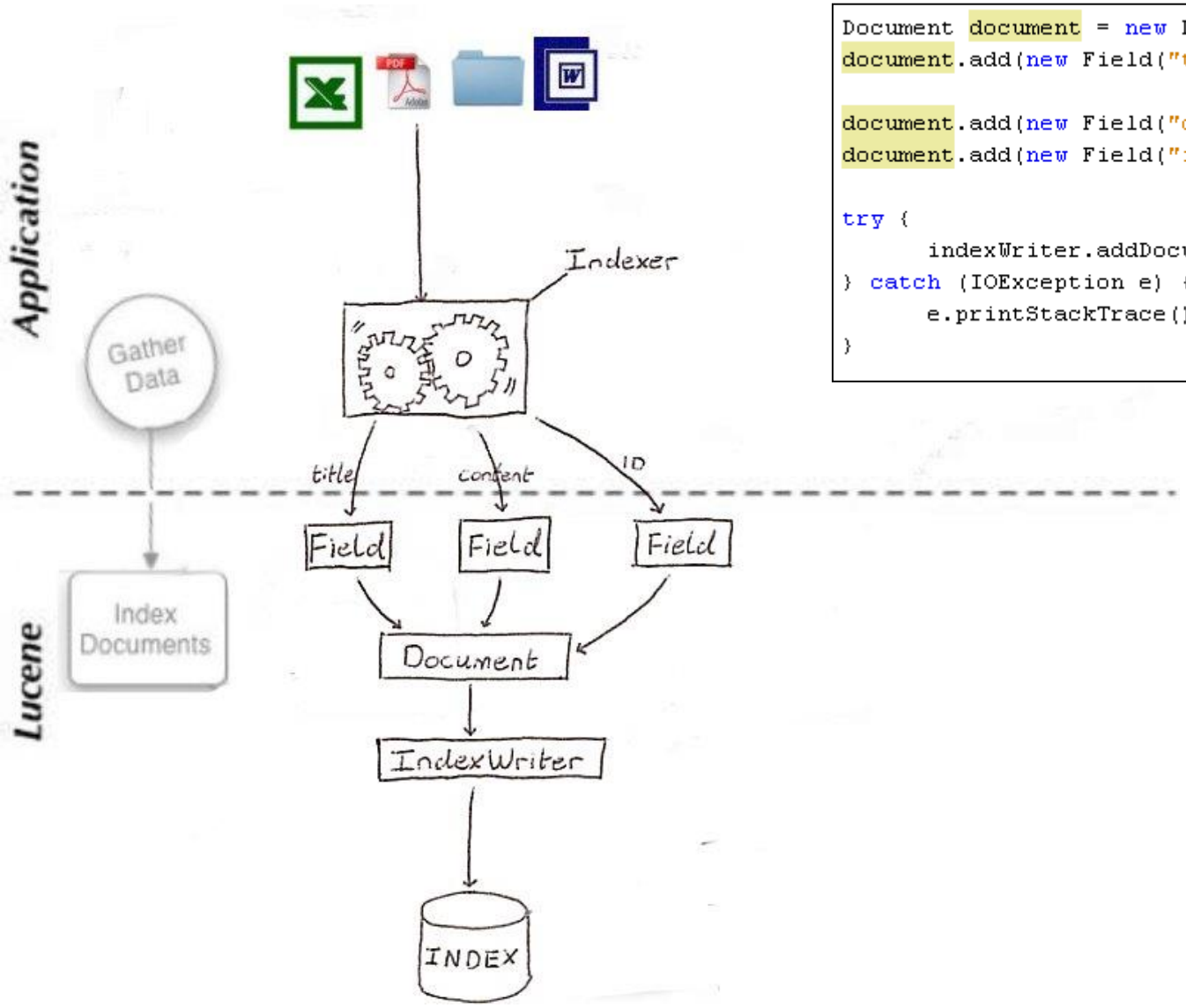

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));
try {
      indexWriter.addDocument(document);
} catch (IOException e) {
      e.printStackTrace();
}
```

LUCENE INDEXING STEP 4 OF 5



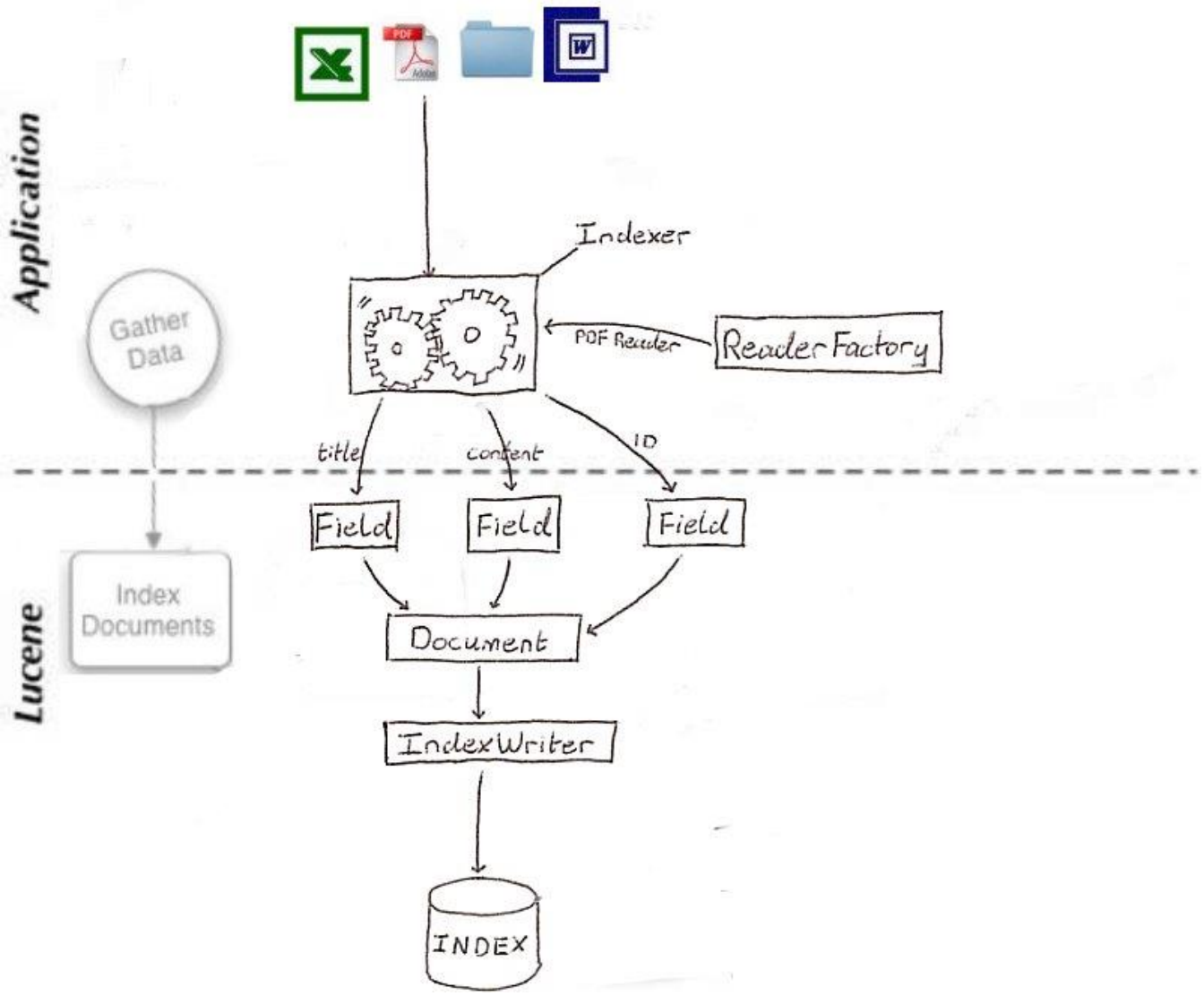

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));

try {
      indexWriter.addDocument(document);
} catch (IOException e) {
      e.printStackTrace();
}
```

LUCENE INDEXING STEP 5 OF 5

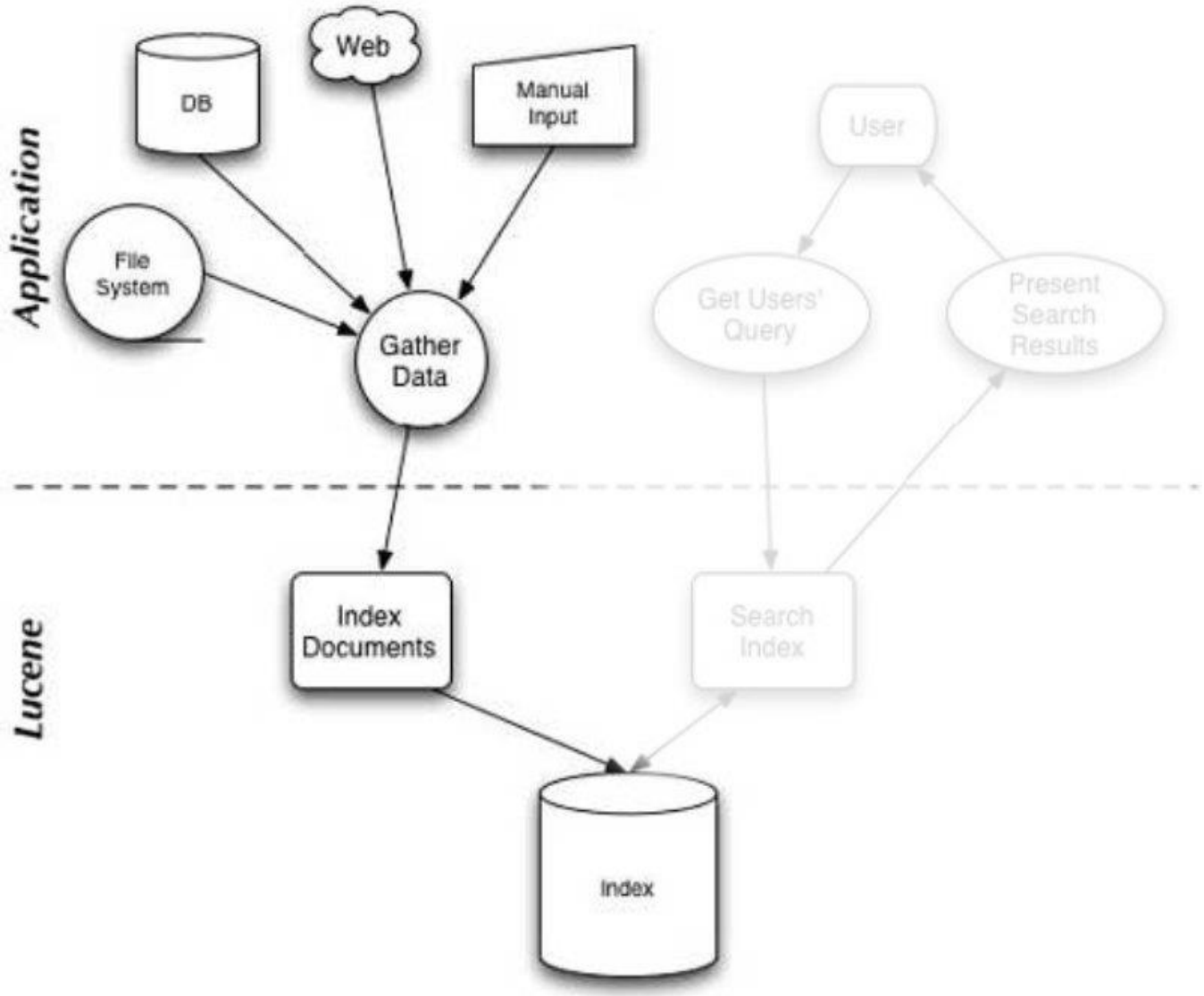



```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));
try {
      indexWriter.addDocument(document);
} catch (IOException e) {
      e.printStackTrace();
}
```

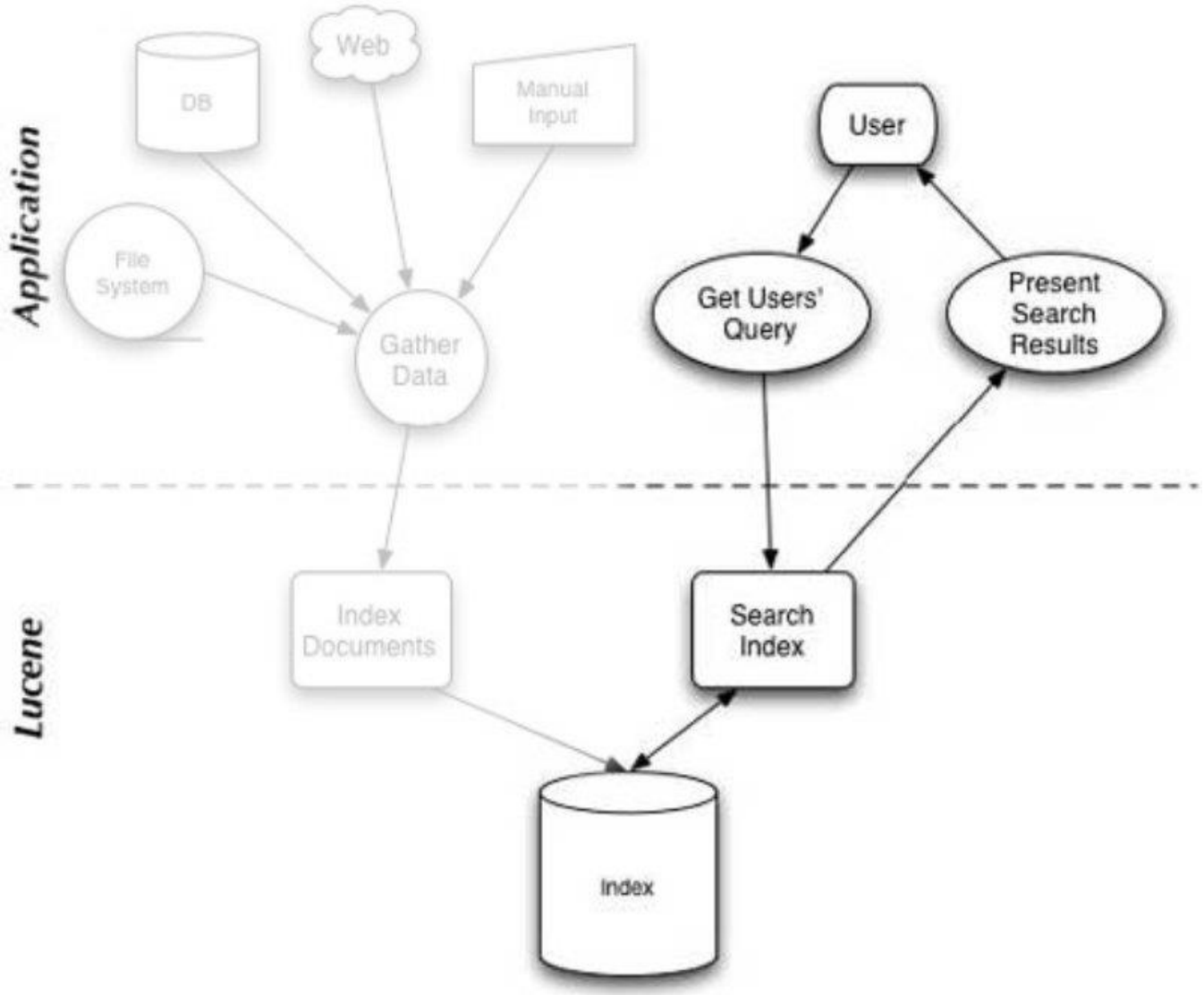
LUCENE INDEXING



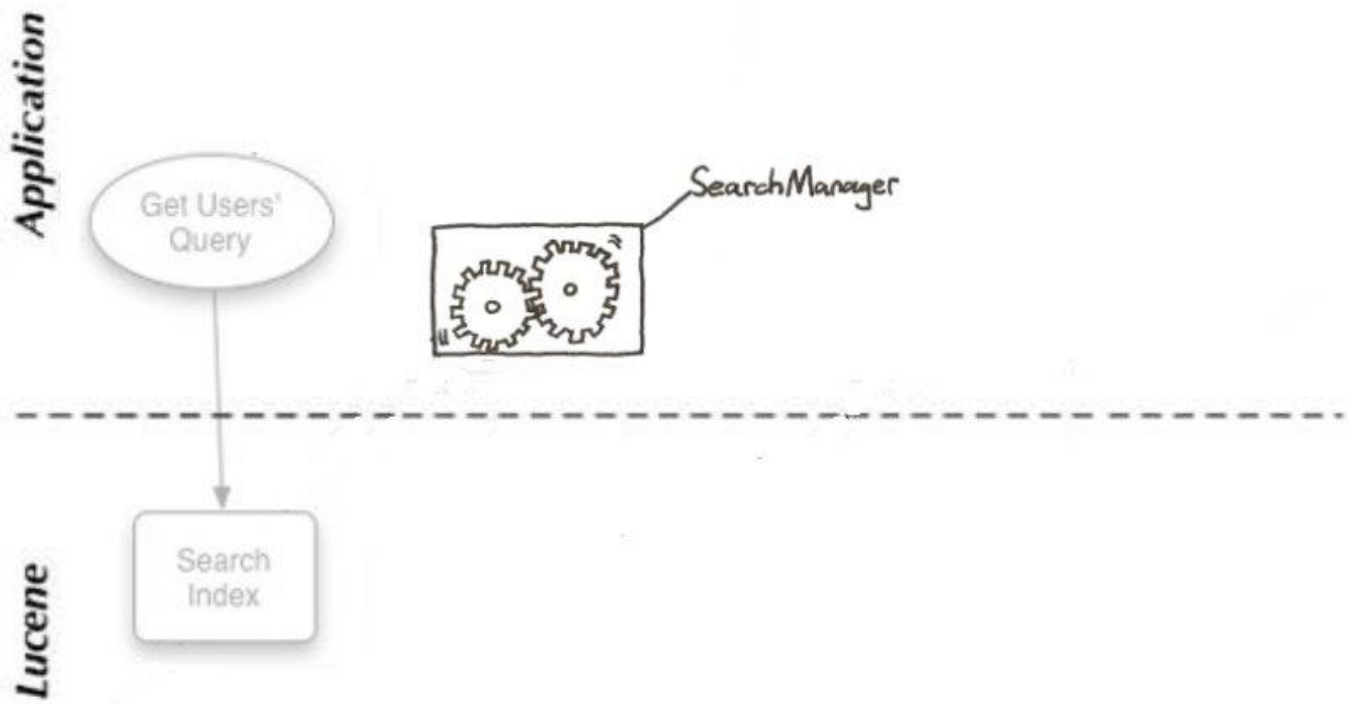
LUCENE IMPLEMENTATION



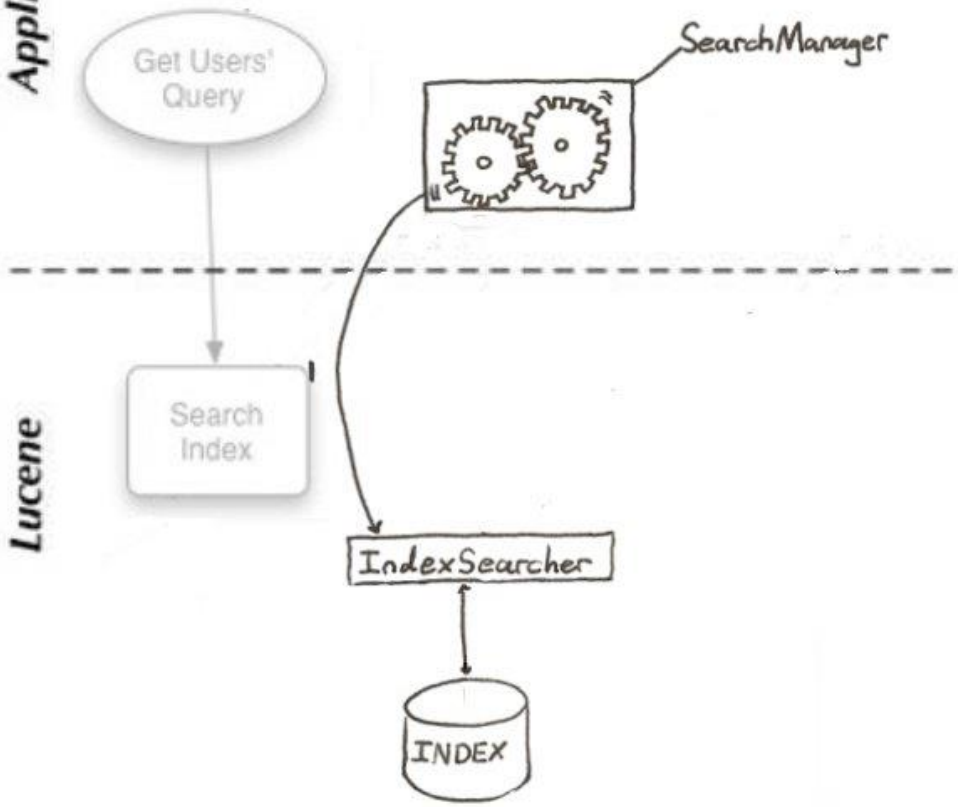
LUCENE IMPLEMENTATION



SEARCHING:


Application
Get Users' Query
SearchManager
Lucene
Search Index

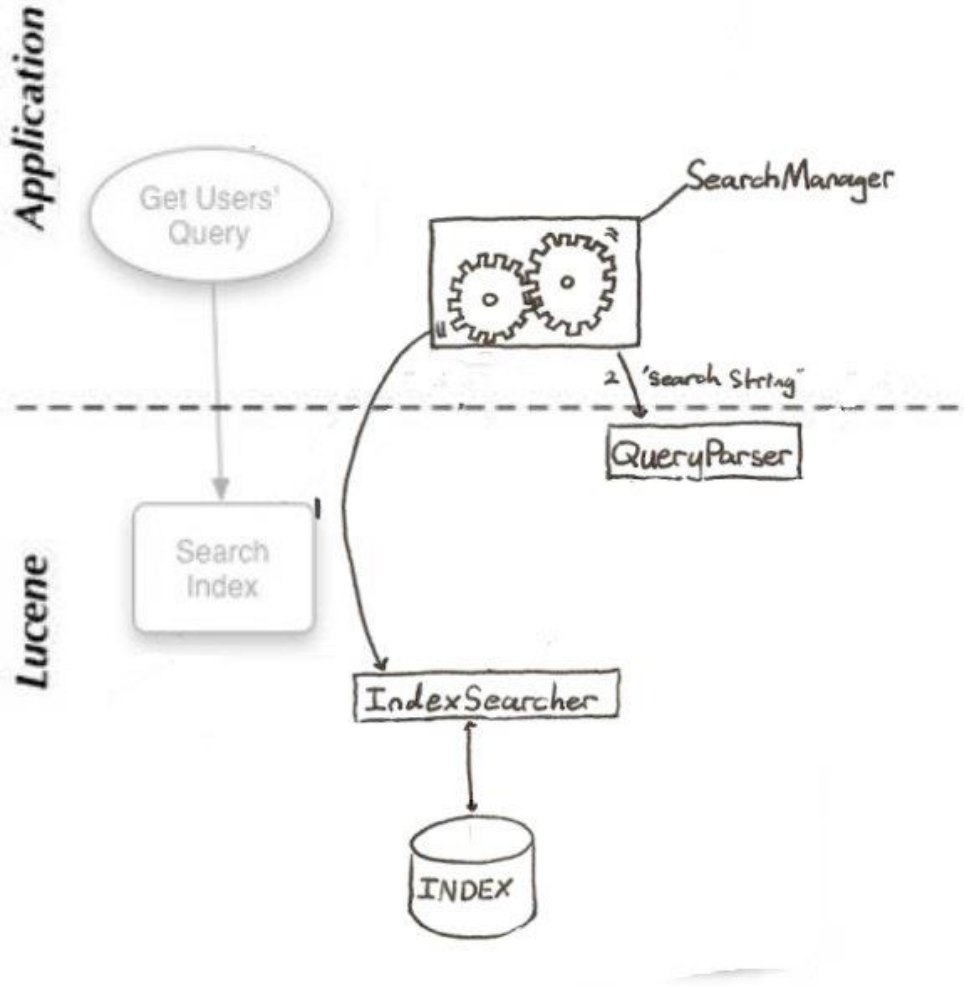
SEARCHING: STEP 1 OF 6



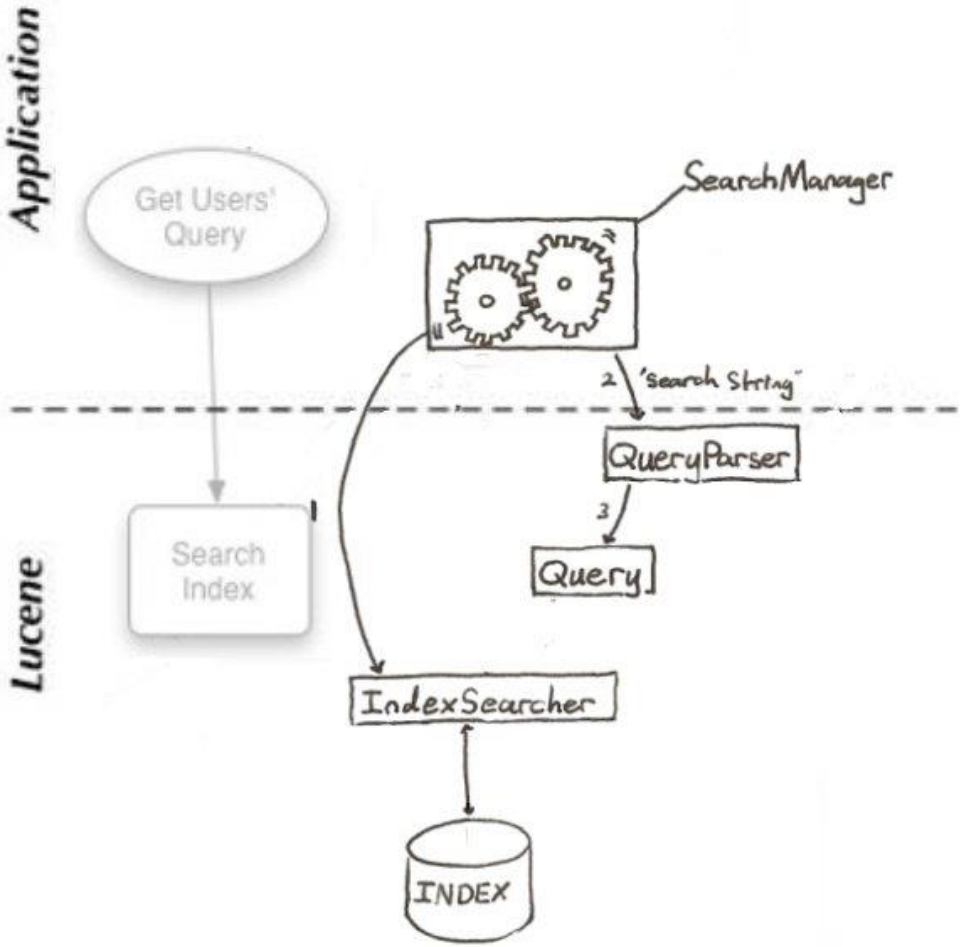

```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
```

SEARCHING: STEP 2 OF 6

```
QueryParser queryParser = new QueryParser("content",analyzer);
```

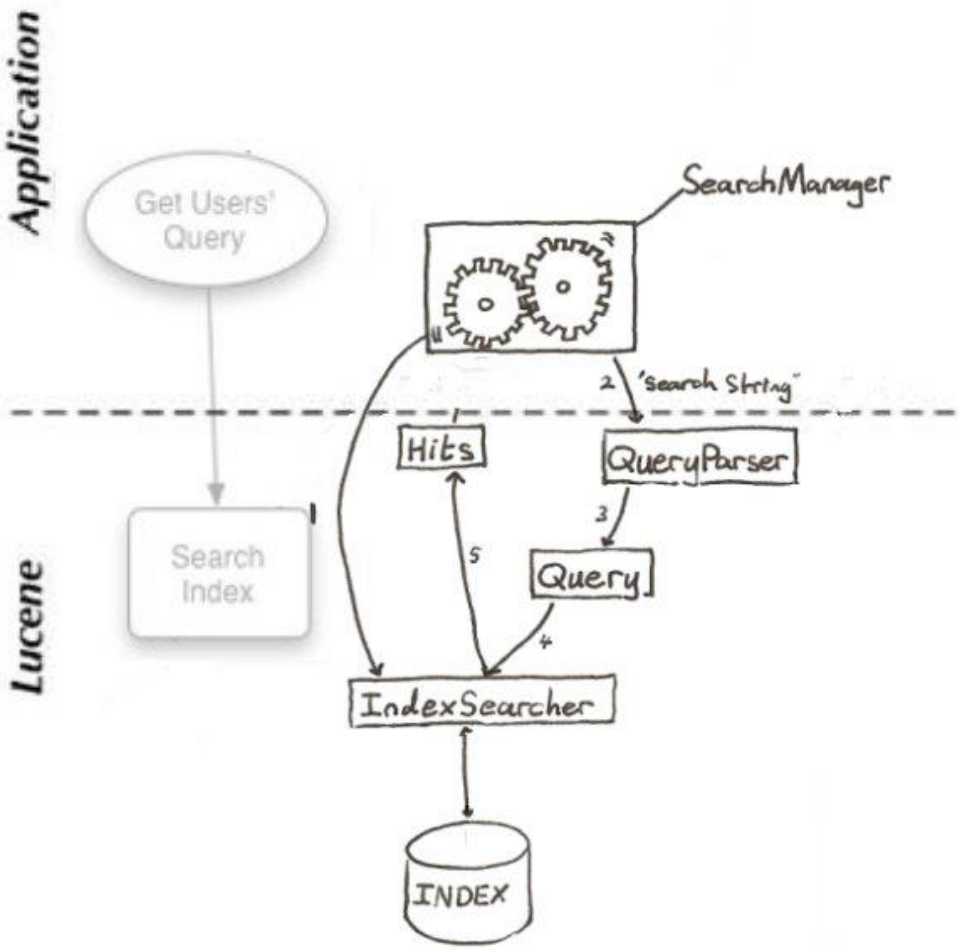


SEARCHING: STEP 3 OF 6



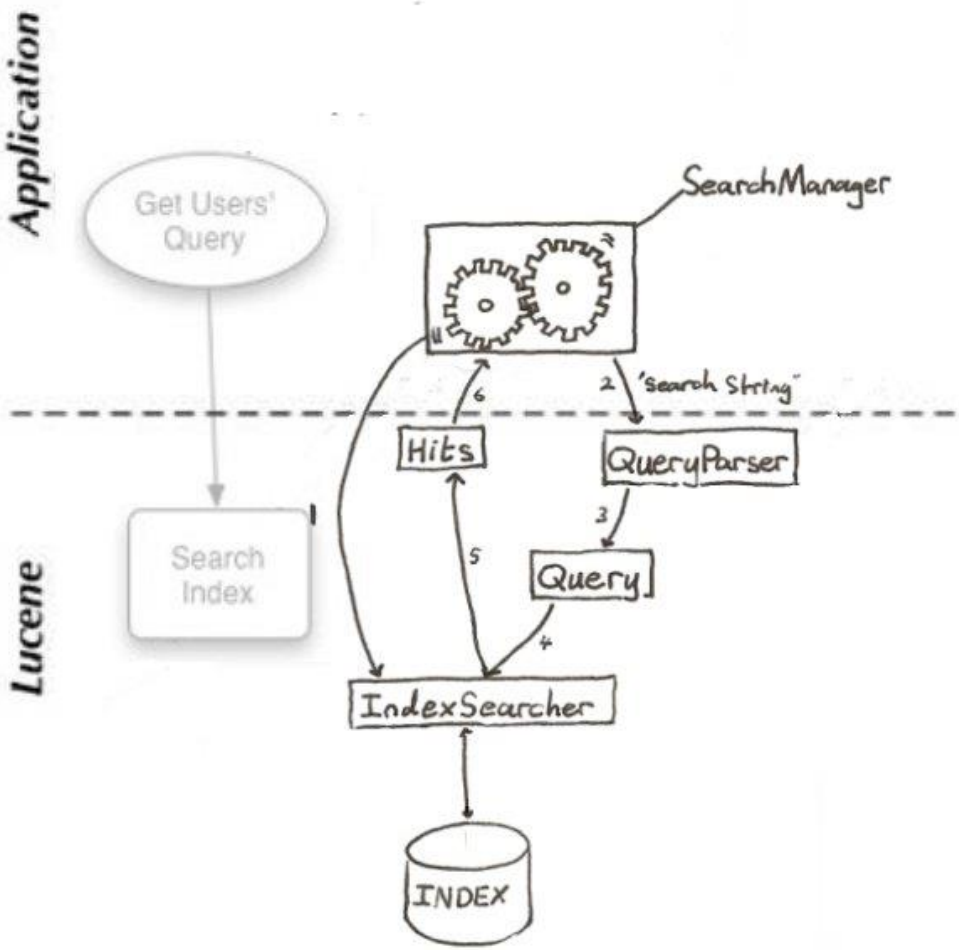

```
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (ParseException e) {
  e.printStackTrace();
}
```

SEARCHING: STEP 4&5 OF 6



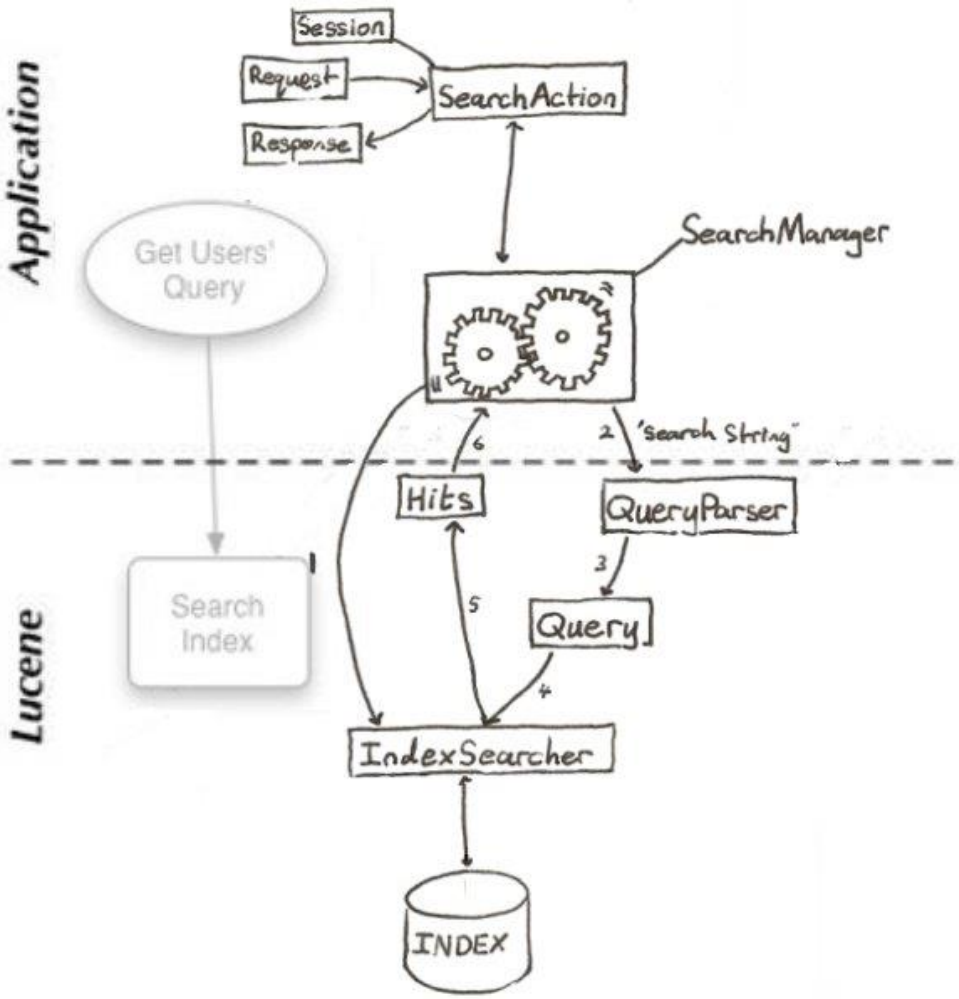

```
if(null != query && null != indexSearcher){
    try {
        Hits hits = indexSearcher.search(query);
```

SEARCHING: STEP 6 OF 6




```
if(null != query && null != indexSearcher){
    try {
        Hits hits = indexSearcher.search(query);
        for(int i = 0; i < hits.length(); i ++){
            System.out.print(hits.doc(i).get("id"));
            System.out.println(hits.doc(i).get("title"));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

SEARCHING




```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
QueryParser queryParser = new QueryParser("content",analyzer);
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (ParseException e) {
  e.printStackTrace();
}
if(null != query && null != indexSearcher){
    try {
        Hits hits = indexSearcher.search(query);
        for(int i = 0; i < hits.length(); i ++){
            System.out.print(hits.doc(i).get("id"));
            System.out.println(hits.doc(i).get("title"));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

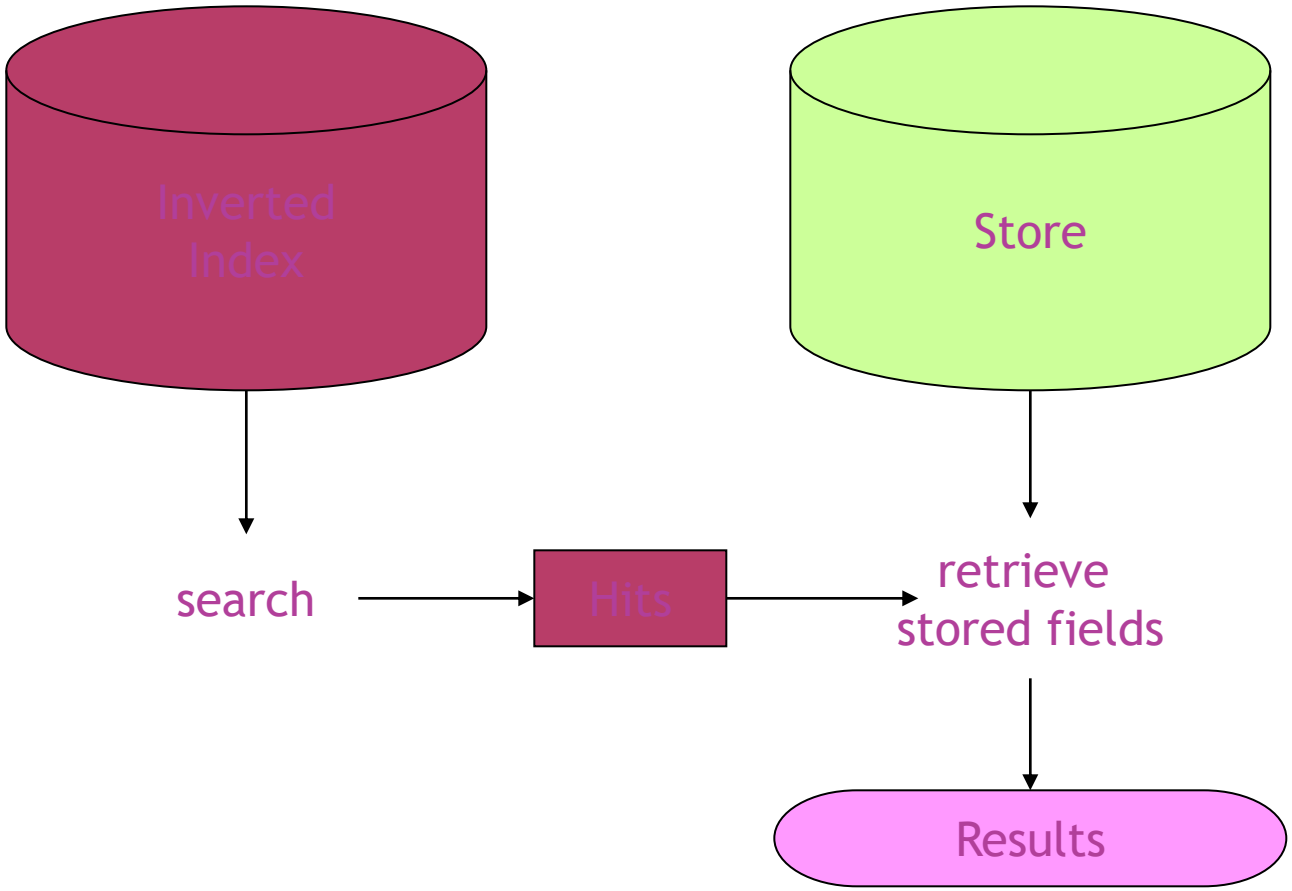
CREATING AN INDEXWRITER

```
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
...
private IndexWriter writer;
...
public Indexer(String indexDir) throws IOException {
  Directory dir = FSDirectory.open(new File(indexDir));
  writer = new IndexWriter(
                                  dir,
                          new
StandardAnalyzer(Version.LUCENE_30),
                                  true,

      IndexWriter.MaxFieldLength.UNLIMITED);
}
```

Core Indexing Classes (Contd.)

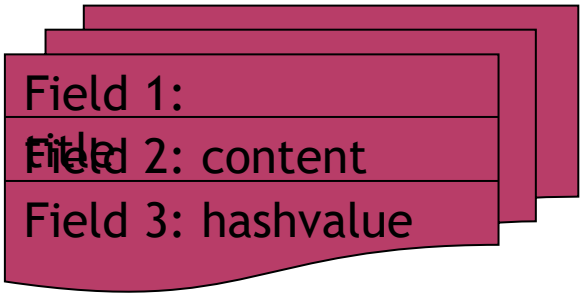
- `Document`
  - Represents a collection of named `Field`s. Text in these `Field`s are indexed.
- `Field`
  - Note: Lucene `Field`s can represent both “fields” and “zones”

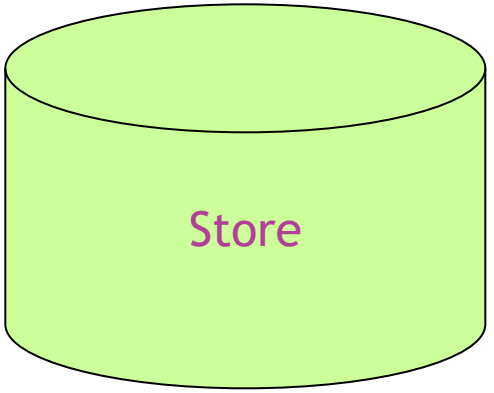
Lucene's data structures

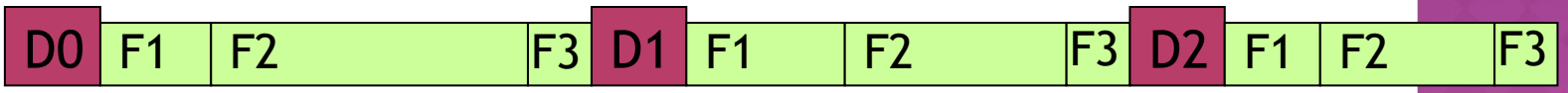


Store

Documents:







A DOCUMENT CONTAINS FIELDS

```
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
...
protected Document getDocument(File f) throws Exception {
        Document doc = new Document();
        doc.add(new Field("contents”, new FileReader(f)))
    doc.add(new Field("filename”, f.getName(),
                        Field.Store.YES,
                  Field.Index.NOT_ANALYZED));
    doc.add(new Field("fullpath”,

f.getCanonicalPath(),
                                                Field.Store.YES,

Field.Index.NOT_ANALYZED));
    return doc;
```

INDEX A DOCUMENT WITH INDEXWRITER

```
private IndexWriter writer;
...
private void indexFile(File f) throws
            Exception {
      Document doc = getDocument(f);
      writer.addDocument(doc);
}
```

INDEXING A DIRECTORY

```
private IndexWriter writer;
...
public int index(String dataDir,
                                 FileFilter filter)
              throws Exception {
       File[] files = new File(dataDir).listFiles();
       for (File f: files) {
              if (... &&
                     (filter == null ||
filter.accept(f))) {
                     indexFile(f);
              }
       }
       return writer.numDocs();
```

CLOSING THE INDEXWRITER

```
private IndexWriter writer;
...
public void close() throws IOException
{
     writer.close();
}
```

Indexing

- Attributes
  - Stored: original content retrievable
  - Indexed: inverted, searchable
  - Tokenized: analyzed, split into tokens
- Factory methods
  - Keyword: stored and indexed as single term
  - Text: indexed, tokenized, and stored if String
  - UnIndexed: stored
  - UnStored: indexed, tokenized
- Terms are what matter for searching

CORE SEARCHING CLASSES

- `IndexSearcher`
  - Central class that exposes several search methods on an index
- `Query`
  - Abstract query class. Concrete subclasses represent specific types of queries, e.g., matching terms in fields, boolean queries, phrase queries, …
- `QueryParser`
  - Parses a textual representation of a query into a `Query` instance

CREATING AN INDEXSEARCHER

```
import org.apache.lucene.search.IndexSearcher;
...
public static void search(String indexDir,

        String q)
            throws IOException, ParseException {
      Directory dir = FSDirectory.open(
                                                  new
File(indexDir));
      IndexSearcher is = new IndexSearcher(dir);
      ...
}
```

QUERY AND QUERYPARSER

```
import org.apache.lucene.search.Query;
import org.apache.lucene.queryParser.QueryParser;
...
public static void search(String indexDir, String q)
            throws IOException, ParseException
      ...
      QueryParser parser =
            new QueryParser(Version.LUCENE_30,

"contents”,
                                                new
StandardAnalyzer(

        Version.LUCENE_30));
```

Core Searching Classes (Contd.)

- `TopDocs`
  - Contains references to the top documents returned by a search
- `ScoreDoc`
  - Represents a single search result

search() returns TopDocs

```
import org.apache.lucene.search.TopDocs;
...
public static void search(String indexDir,

                String q)
            throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    Query query = ...;
    ...
    TopDocs hits = is.search(query, 10);
}
```

TopDocs contain ScoreDocs

```
import org.apache.lucene.search.ScoreDoc;
...
public static void search(String indexDir, String q)
            throws IOException, ParseException
      ...
      IndexSearcher is = ...;
      ...
      TopDocs hits = ...;
      ...
      for(ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = is.doc(scoreDoc.doc);
            System.out.println(doc.get("fullpath"));
      }
}
```

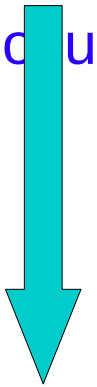
CLOSING INDEXSEARCHER

```
public static void search(String indexDir,

                          String q)
                   throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    is.close();
}
```

LUCENE CODE EXAMPLE: INDEXING

```
01 Analyzer analyzer = new StandardAnalyzer();
02 IndexWriter iw = new IndexWriter("/tmp/testindex", analyzer, t
rue);
03
04 Document doc = new Document();
05 doc.add(new Field("body", "This is my TEST document",
     Field.Store.YES, Field.Index.TOKENIZED));
07 iw.addDocument(doc);
08
09 iw.optimize();
10 iw.close();
```

loop



StandardAnalyzer: my, test, document

LUCENE CODE EXAMPLE: SEARCHING

```
01 Analyzer analyzer = new StandardAnalyzer();
02 IndexSearcher is = new IndexSearcher("/tmp/testindex");
03
04 QueryParser qp = new QueryParser("body", analyzer);
05 String userInput = "document AND test";
06 Query q = qp.parse(userInput);
07 Hits hits = is.search(q);
08 for (Iterator iter = hits.iterator(); iter.hasNext();) {
09    Hit hit = (Hit) iter.next();
10    System.out.println(hit.getScore() + " " + hit.get("body"));
11 }
12
13 is.close();
```

How Lucene models content

- A `Document` is the atomic unit of indexing and searching
  - A `Document` contains `Field`s
- `Field`s have a name and a value
  - You have to translate raw content into `Field`s
  - Examples: Title, author, date, abstract, body, URL, keywords, ...
  - Different documents can have different fields
  - Search a field using name:term, e.g., title:lucene

Fields

- `Field`s may
  - Be indexed or not
    - Indexed fields may or may not be analyzed (i.e., tokenized with an `Analyzer`)
      - Non-analyzed fields view the entire value as a single token (useful for URLs, paths, dates, social security numbers, ...)
  - Be stored or not
    - Useful for fields that you'd like to display to users
  - Optionally store term vectors
    - Like a positional index on the `Field`'s terms
    - Useful for highlighting, finding similar documents, categorization

Field Construction
Lots of Different Constructors

```
import org.apache.lucene.document.Field

Field(String name,
              String value,
              Field.Store store,  // store or
not
              Field.Index index,  // index or
not
              Field.TermVector termVector);
```

`value` can also be specified with a `Reader`, a `TokenStream`, or a `byte[]`

Field Options

- `Field.Store`
  - `NO` : Don't store the field value in the index
  - `YES` : Store the field value in the index
- `Field.Index`
  - `ANALYZED` : Tokenize with an `Analyzer`
  - `NOT_ANALYZED` : Do not tokenize
  - `NO` : Do not index this field
  - `Couple of other advanced options`
- `Field.TermVector`
  - `NO` : Don't store term vectors
  - `YES` : Store term vectors
  - Several other options to store positions and offsets

USING FIELD OPTIONS

Index	Store	TermVector	Example usage
NOT_ANALYZED	YES	NO	Identifiers, telephone/SSNs, URLs, dates, ...
ANALYZED	YES	WITH_POSITIONS_OFFSETS	Title, abstract
ANALYZED	NO	WITH_POSITIONS_OFFSETS	Body
NO	YES	NO	Document type, DB keys (if not used for searching)
NOT_ANALYZED	NO	NO	Hidden keywords

DOCUMENT

```
import org.apache.lucene.document.Field
```

- Constructor:
  - `Document();`
- Methods
  - `void add(Fieldable field); // Field implements // Fieldable`
  - `String get(String name);  // Returns value of Field with given // name`
  - `Fieldable getFieldable(String name);`
  - ... and many more

MULTI-VALUED FIELDS

- You can add multiple `Field`s with the same name
  - Lucene simply concatenates the different values for that named Field

```
        Document doc = new Document();
   doc.add(new Field("author",
                                                    "chris
manning",
Field.Store.YES,
Field.Index.ANALYZED));
         doc.add(new Field("author",
                                                    "prabhakar
raghavan",
Field.Store.YES,
Field.Index.ANALYZED));
         ...
```

Analyzers

- Tokenizes the input text
- Common `Analyzer`s
  - `WhitespaceAnalyzer`
    Splits tokens on whitespace
  - `SimpleAnalyzer`
    Splits tokens on non-letters, and then lowercases
  - `StopAnalyzer`
    Same as `SimpleAnalyzer`, but also removes stop words
  - `StandardAnalyzer`
    Most sophisticated analyzer that knows about certain token types, lowercases, removes stop words, ...

Analysis Examples

- "The quick brown fox jumped over the lazy dog"
- `WhitespaceAnalyzer`
  - [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- `SimpleAnalyzer`
  - [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- `StopAnalyzer`
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
- StandardAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]

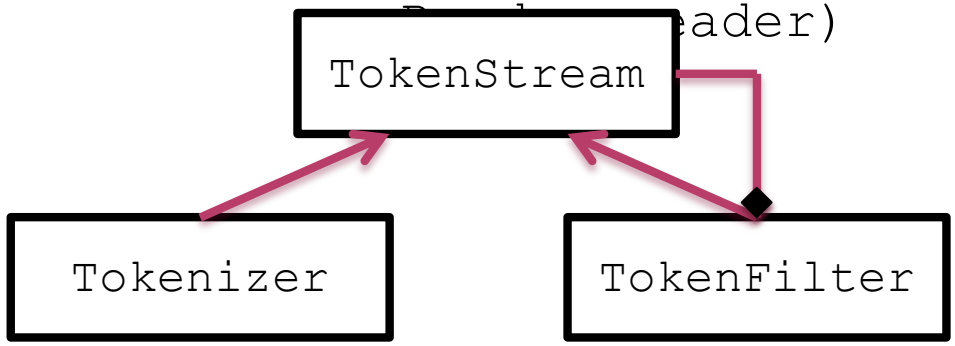
MORE ANALYSIS EXAMPLES

- “XY&Z Corporation - xyz@example.com”
- `WhitespaceAnalyzer`
  - [XY&Z] [Corporation] [-] [xyz@example.com]
- `SimpleAnalyzer`
  - [xy] [z] [corporation] [xyz] [example] [com]
- `StopAnalyzer`
  - [xy] [z] [corporation] [xyz] [example] [com]
- StandardAnalyzer
  - [xy&z] [corporation] [xyz@example.com]

WHAT'S INSIDE AN ANALYZER?

- Analyzers need to return a `TokenStream`

```
public TokenStream tokenStream(String
fieldName,
                               eader)
```



TOKENIZERS AND TOKENFILTERS

- Tokenizer
  - WhitespaceTokenizer
  - KeywordTokenizer
  - LetterTokenizer
  - StandardTokenizer
  - ...
- TokenFilter
  - LowerCaseFilter
  - StopFilter
  - PorterStemFilter
  - ASCIIFoldingFilter
  - StandardFilter
  - ...

IndexWriter Construction

```
// Deprecated
IndexWriter(Directory d,
                        Analyzer a,  //
default analyzer

IndexWriter.MaxFieldLength mfl);

// Preferred
IndexWriter(Directory d,
                        IndexWriterConfig c);
```

Adding/deleting documents to/from an IndexWriter

```
void addDocument(Document d);
void addDocument(Document d, Analyzer a);
```

Important: Need to ensure that `Analyzer`s used at indexing time are consistent with `Analyzer`s used at searching time

```
// deletes docs containing term or matching
// query.  The term version is useful for
// deleting one document.
void deleteDocuments(Term term);
void deleteDocuments(Query query);
```

INDEX FORMAT

- Each Lucene index consists of one or more segments
  - A segment is a standalone index for a subset of documents
  - All segments are searched
  - A segment is created whenever `IndexWriter` flushes adds/deletes
- Periodically, `IndexWriter` will merge a set of segments into a single segment
  - Policy specified by a `MergePolicy`
- You can explicitly invoke `optimize()` to merge segments

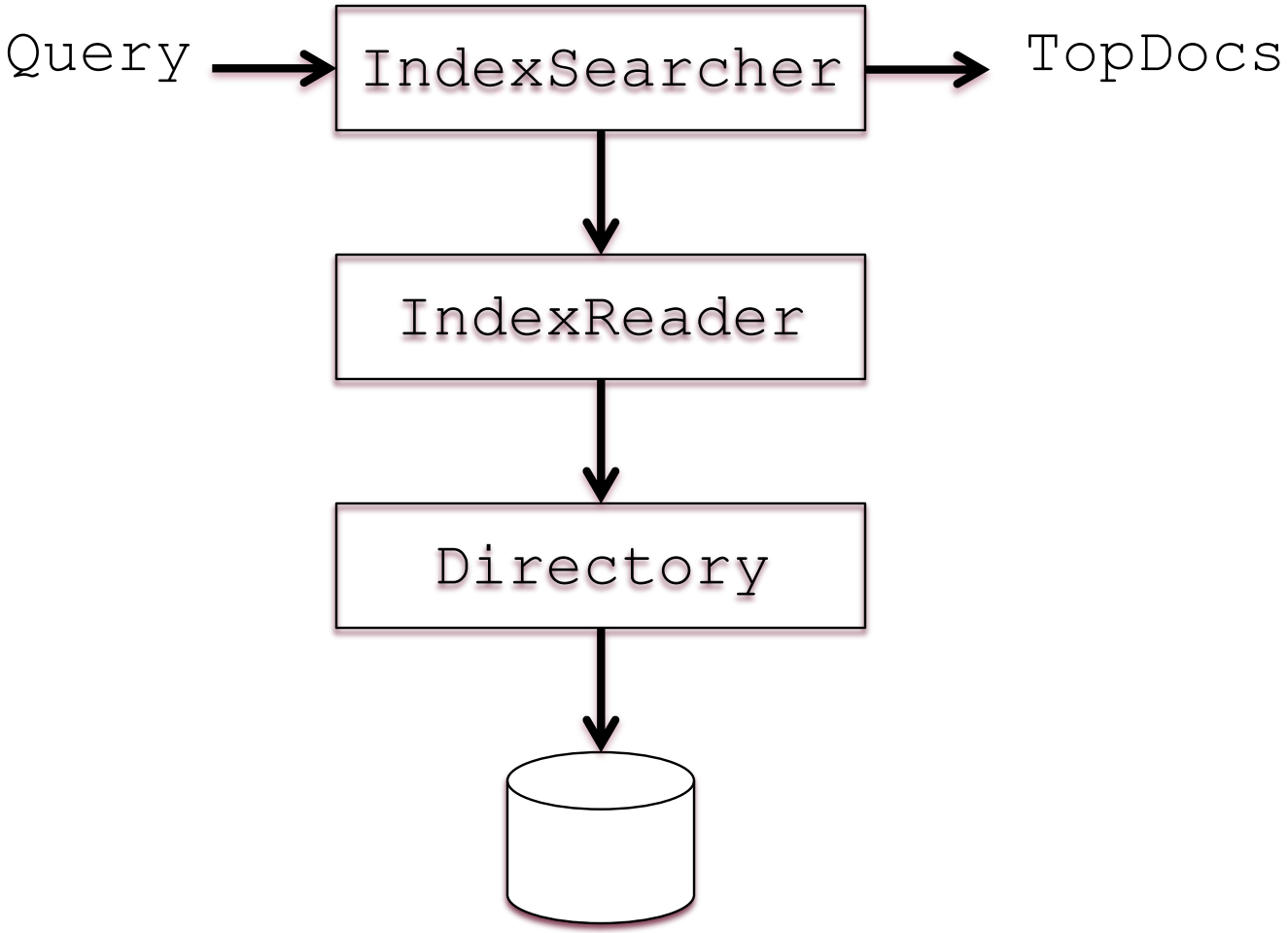
BASIC MERGE POLICY

- Segments are grouped into levels
- Segments within a group are roughly equal size (in log space)
- Once a level has enough segments, they are merged into a segment at the next level up

IndexSearcher

- Constructor:
  - `IndexSearcher(Directory d);`
    - deprecated

INDEXREADER



IndexSearcher

- Constructor:
  - `IndexSearcher(Directory d);`
    - deprecated
  - `IndexSearcher(IndexReader r);`
    - Construct an `IndexReader` with static method `IndexReader.open(dir)`

SEARCHING A CHANGING INDEX

```
Directory dir = FSDirectory.open(...);
IndexReader reader = IndexReader.open(dir);
IndexSearcher searcher = new IndexSearcher(reader);
```

Above `reader` does not reflect changes to the index unless you `reopen` it.

`Reopen`ing is more resource efficient than `open`ing a new `IndexReader`.

```
IndexReader newReader = reader.reopen();
If (reader != newReader) {
        reader.close();
        reader = newReader;
        searcher = new IndexSearcher(reader);
}
```

NEAR-REAL-TIME SEARCH

```
IndexWriter writer = ...;
IndexReader reader = writer.getReader();
IndexSearcher searcher = new IndexSearcher(reader);
```

Now let us say there's a change to the index using `writer`

```
// reopen() and getReader() force writer to flush
IndexReader newReader = reader.reopen();
if (reader != newReader) {
      reader.close();
      reader = newReader;
      searcher = new IndexSearcher(reader);
}
```

IndexSearcher

- Methods
  - `TopDocs search(Query q, int n);`
  - `Document doc(int docID);`

QueryParser

- Constructor
  - `QueryParser(Version matchVersion, String defaultField, Analyzer analyzer);`
- Parsing methods
  - `Query parse(String query) throws ParseException;`
  - ... and many more

QUERYPARSER SYNTAX EXAMPLES

Query expression	Document matches if…
java	Contains the term *java* in the default field
java junit java OR junit	Contains the term *java* or *junit* or both in the default field (*the default operator can be changed to* AND)
+java +junit java AND junit	Contains both *java* and *junit* in the default field
title:ant	Contains the term *ant* in the title field
title:extreme -subject:sports	Contains *extreme* in the title and not *sports* in subject
(agile OR extreme) AND java	Boolean expression matches
title:"junit in action"	Phrase matches in title
title:"junit action"~5	Proximity matches (within 5) in title
java*	Wildcard matches
java~	Fuzzy matches

Construct Querys Programmatically

- `TermQuery`
  - Constructed from a `Term`
- `TermRangeQuery`
- `NumericRangeQuery`
- `PrefixQuery`
- `BooleanQuery`
- `PhraseQuery`
- `WildcardQuery`
- `FuzzyQuery`
- `MatchAllDocsQuery`

TopDocs and ScoreDoc

- `TopDocs` methods
  - Number of documents that matched the search
    `totalHits`
  - Array of `ScoreDoc` instances containing results
    `scoreDocs`
  - Returns best score of all matches
    `getMaxScore()`
- `ScoreDoc` methods
  - Document id
    `doc`
  - Document score
    `score`

SCORING

- Scoring function uses
  - Programmable boost values for certain fields in documents
  - Length normalization
  - Boosts for documents containing more of the query terms
- `IndexSearcher` provides an `explain()` method that explains the scoring of a document

SEARCH AND ANALYTICS (USING ELASTICSEARCH)

- Search – what's the big deal?
- Basic/Metadata retrieval
- "Find banks with more then (x) accounts"
- "Find banks *near my location*"

SEARCH CATEGORIES

- Basic/Metadata retrieval : data stores
- search engines:
  - Full-text search
  - Highlighting
  - Geolocation
  - Fuzzy search (“did-you-mean”)
  - Natural Language

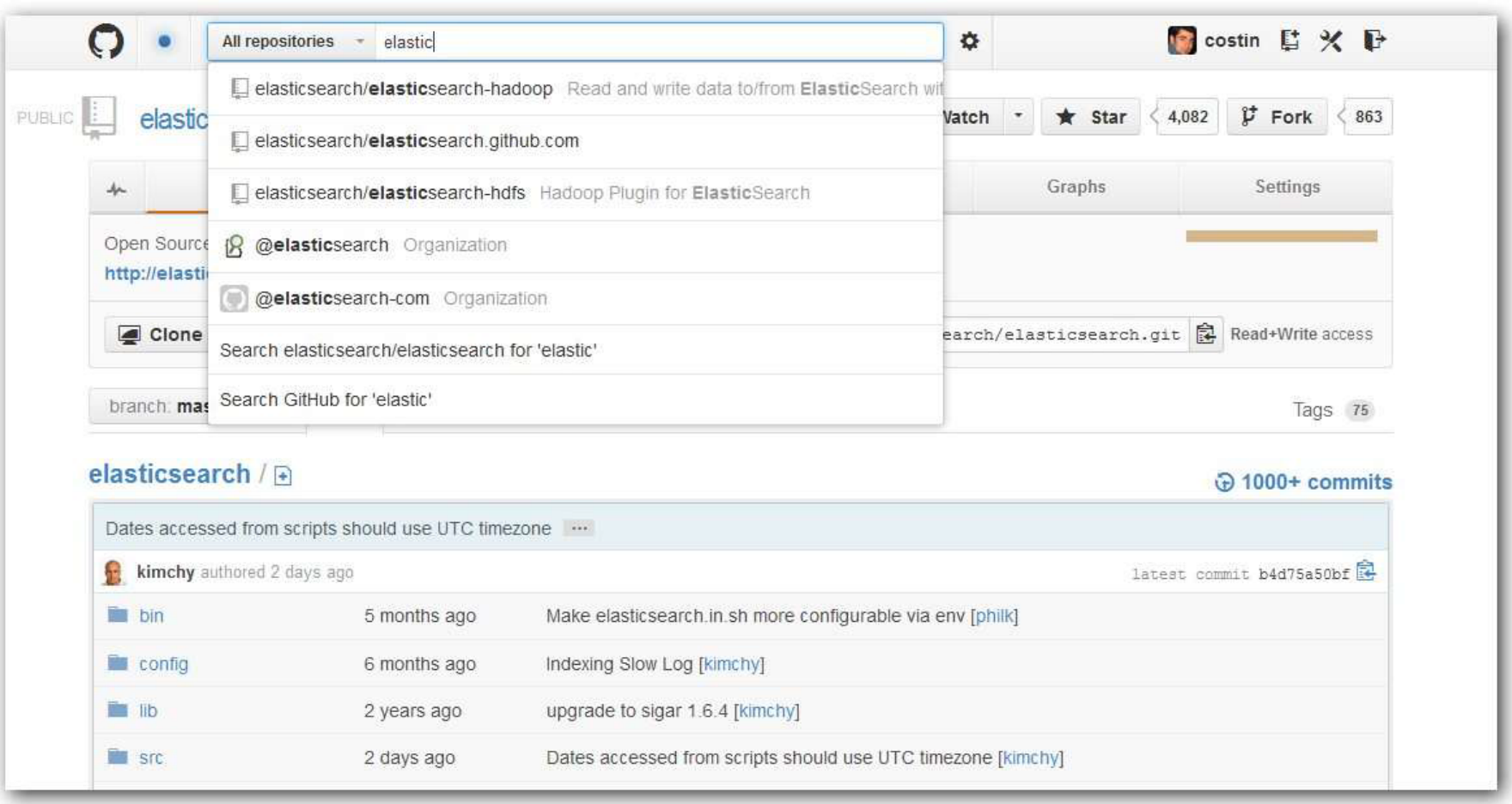
ELASTICSEARCH

- **ElasticSearch** is a Distributed, RESTful, free/open source search server based on Apache Lucene. It is developed by Shay Banon[1] and is released under the terms of the Apache License. ElasticSearch is developed in Java.
- Open-Source Search & Analytics engine
  - Structured & Unstructured Data
  - Real Time
  - Analytics capabilities (facets)
  - REST based
- Distributed
  - Designed for the Cloud
  - Designed for Big Data

ELASTICSEARCH

- **Advantages**:
  - ElasticSearch is distributed.
  - ElasticSearch fully supports the near real-time search of Apache Lucene.
  - Handling multitenancy is not a special configuration, whereas a more advanced setup is necessary with Solr.
  - ElasticSearch introduces the concept of the Gateway, which makes full backups easier.
- **Disadvantages**:
  - Vendor dependent

USE CASE - TEXT SEARCH



USE CASE - GEOLOCATION

- Searches 50,000,000 venues every day using Elasticsearch