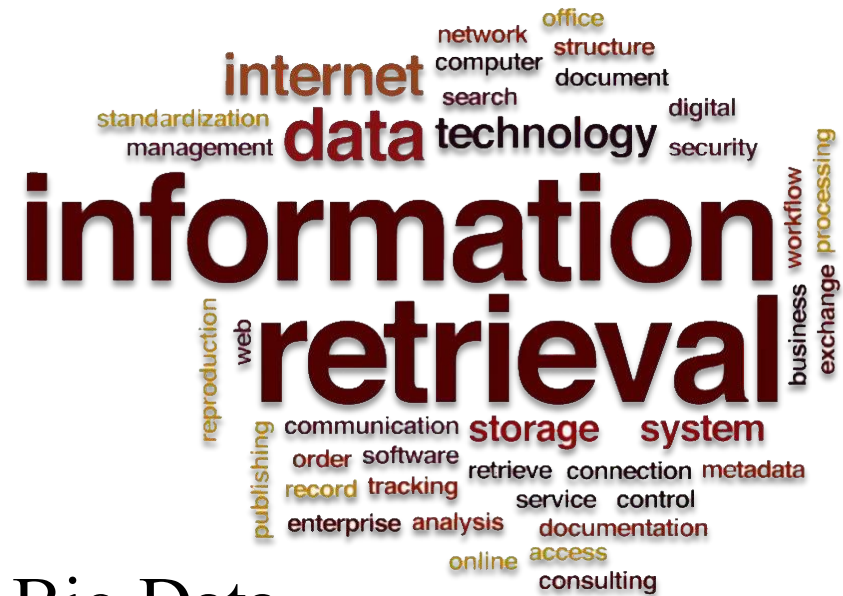




6CS030 Week 09

Indexing And Searching In Big Data



Fast Text Search: The Role of Indexing

- Searching large volumes of text efficiently **requires indexing**.
- **Indexing** converts raw text into a structured format called an **index**.
- This enables **rapid search**, avoiding slow sequential scans.
- Indexing is the **foundational** step in all search applications.



Goal of Indexing

Transform original data into an efficient cross-reference system for fast lookups.


The task is simpler when:

The content is already in text format.

Its location is known.

Searching

Searching uses this index to quickly find documents containing specific words—
No need to scan the entire text.

- ✓ Indexing builds the foundation.
-  Searching leverages it for speed and accuracy.












? What do you think powers fast searching in many applications?



Answer: Lucene



-  **High-performance, scalable**, full-text search library used by Facebook, twitter, LinkedIn, ..
-  Written by **Doug Cutting**, implemented in **100% Java** but **integrable on C/C++, Python, R**
-  Focuses on **indexing and searching documents**
-  **Easily embeddable** – no configuration files needed
-  No built-in **crawlers** or **document parsing**
-  Provides **core indexing and search services**
-  Competitive in:
 - **Engine performance**
 - **Search relevancy**
 - **Ease of code maintenance**

Lucene Architecture : Core Components

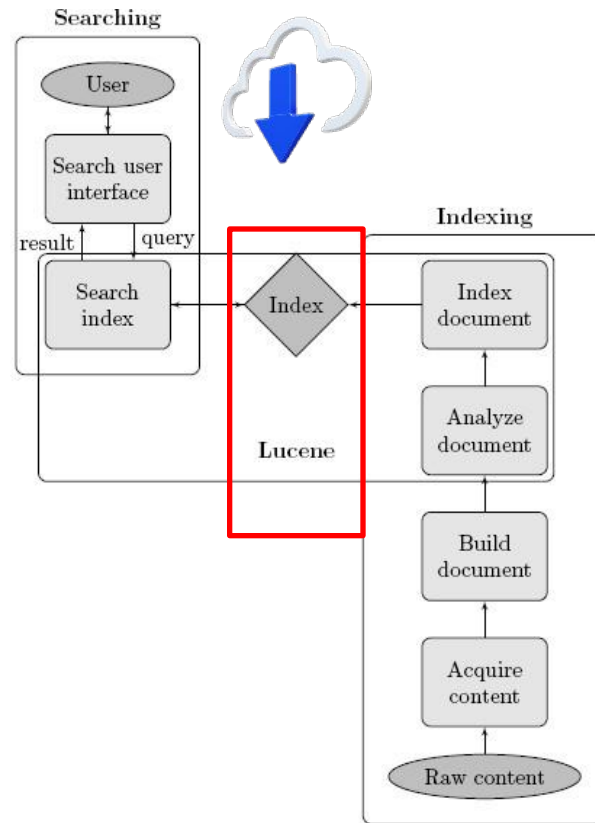


Figure 3.1: Typical components of search application architecture with Lucene components highlighted

Searching Side:

- User inputs a query through the interface.
- The query is processed against the indexed data.
- Results are returned to the user interface.

Indexing Side:

- Content Acquisition: Raw data is collected from various sources.
- Build Document: Content is structured into a format Lucene can process.
- Document Analysis: Documents are tokenized and filtered.
- Index Document: The actual search index is created and maintained, making content searchable.

Connection Point:




Index serves as the bridge between the indexing and searching processes, responsible for creating and maintaining efficient index structures for fast search retrieval. [Let's explore how it works!](#)

Indexing Approach

Linear String comparison **slow!**



“BigDataApplicationsInLogistics” vs
“BigDataApplicationsInHealthcare” means
scanning character by character until a
difference is found.

-  **Fast Lookups:** Jump directly to documents containing a term.
-  **Scales Well:** Handles large volumes of text efficiently.
-  **Enables Advanced Search:** Supports features like ranking, phrase search, wildcards, etc.

Query: **not**

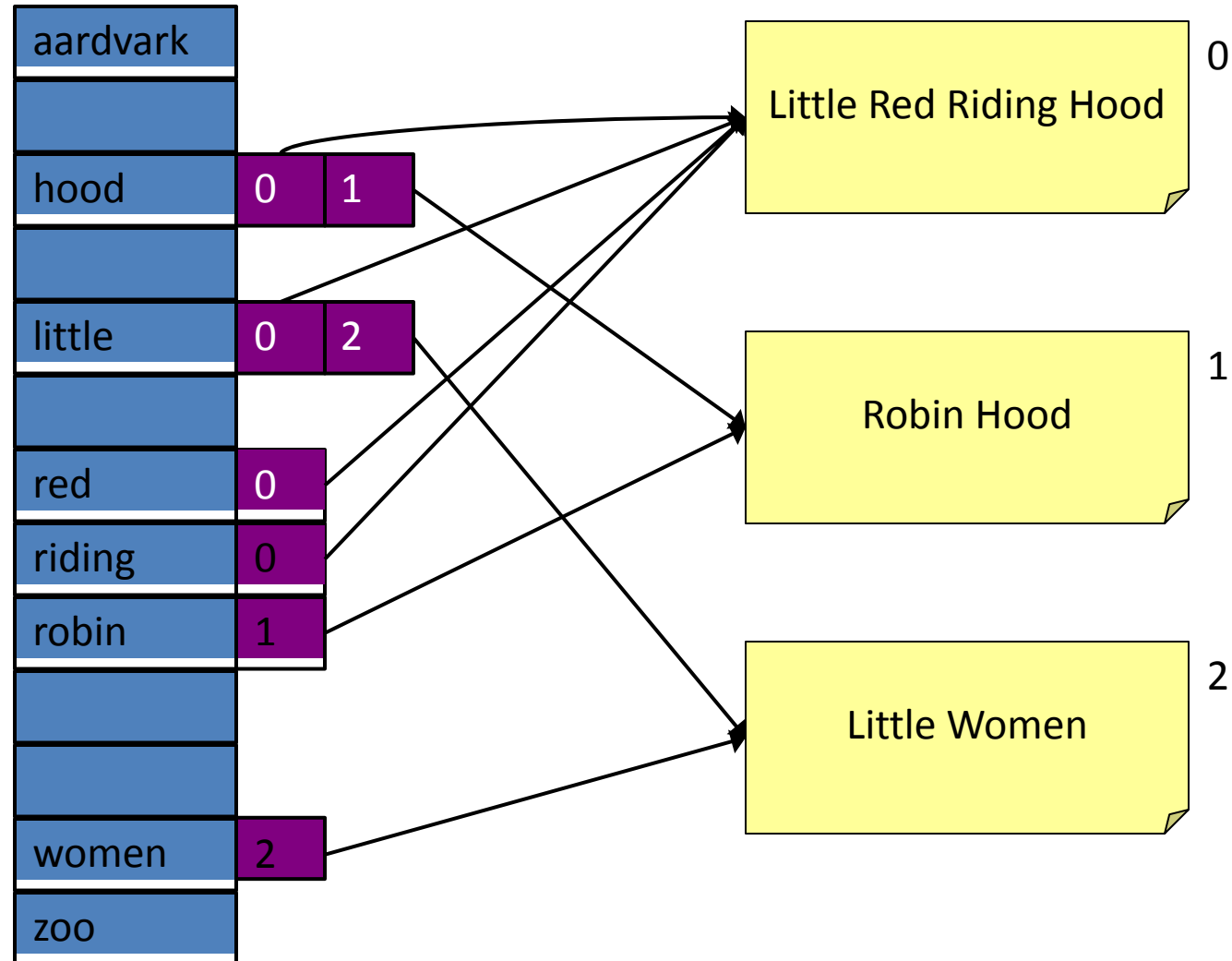
c:\docs\einstein.txt:

The important thing is not to
stop questioning.

Solution:
Inverted index

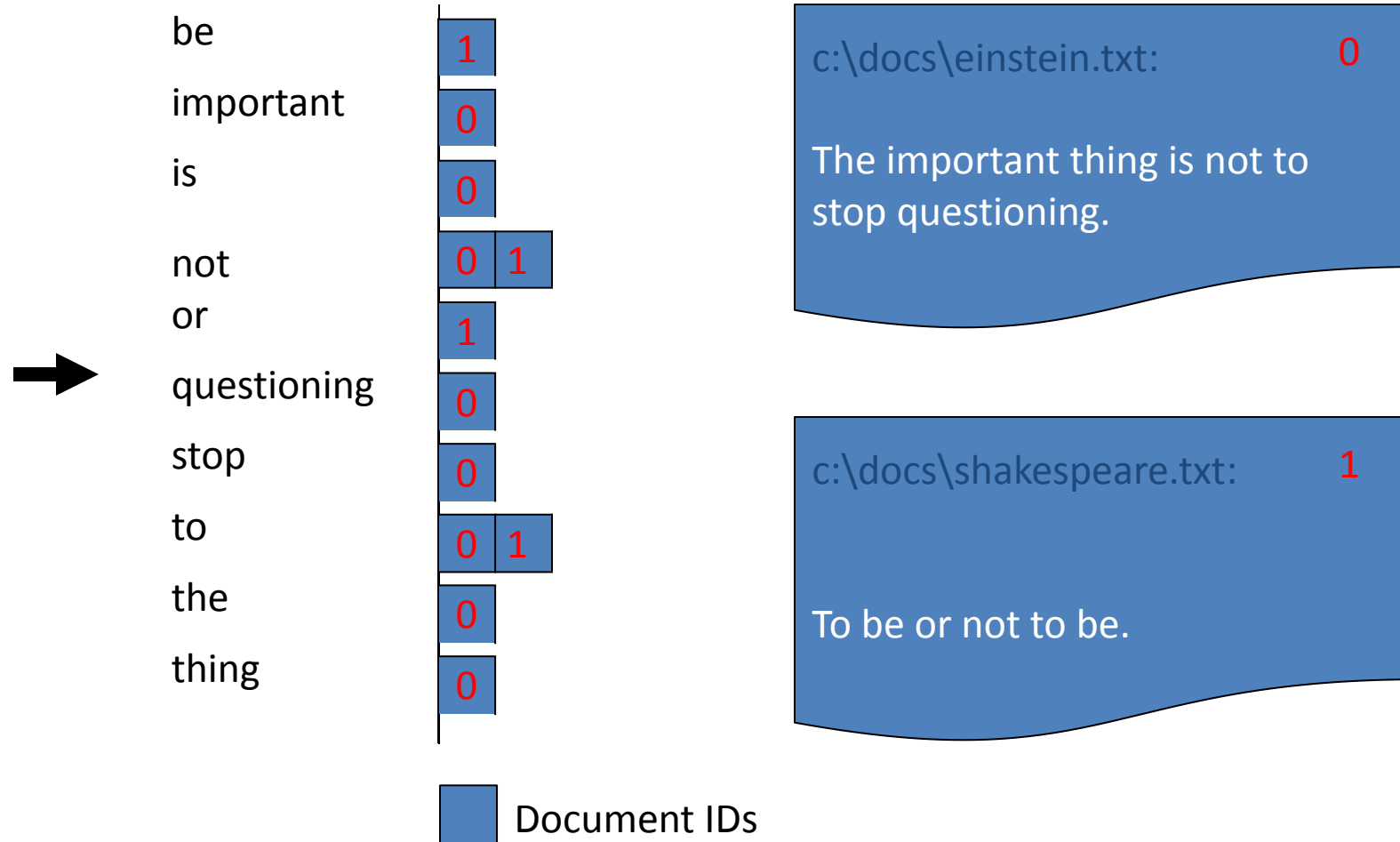
Indexing Method Used in Lucene

Inverted Index: Terms point to the documents containing them (instead of documents pointing to terms) enabling field specific searching.



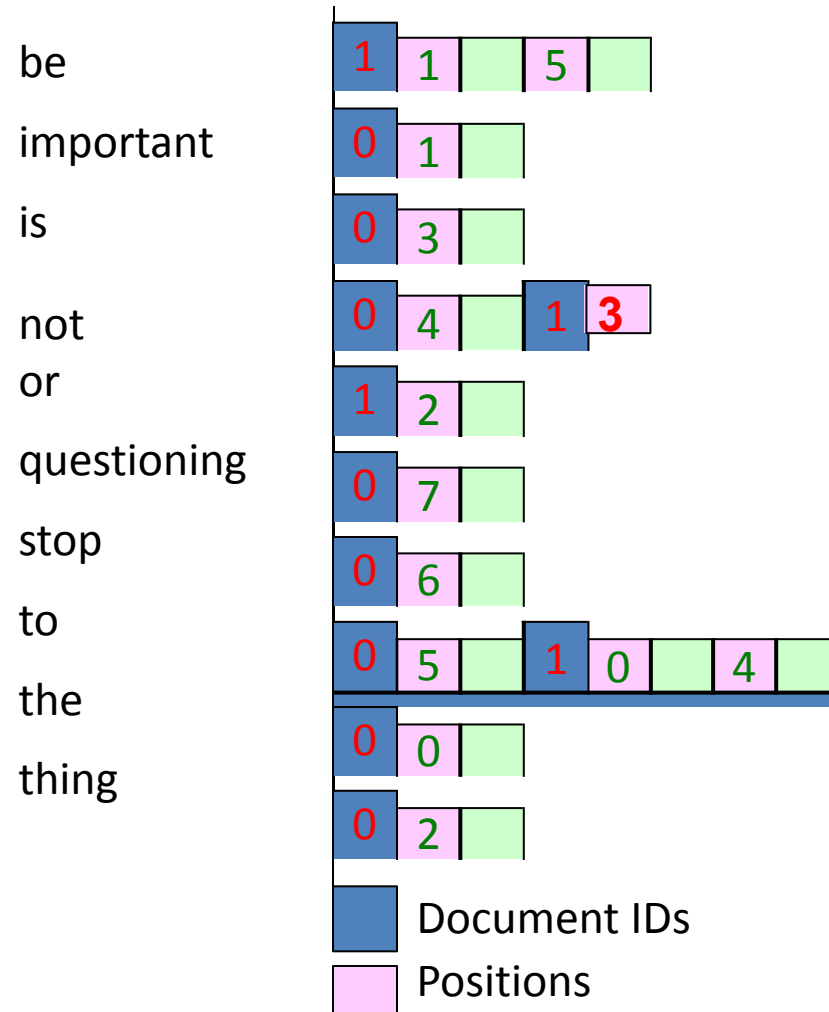
Inverted index

Query: not



But Order Matters so, lets enhancing the Inverted Index by adding Positions →

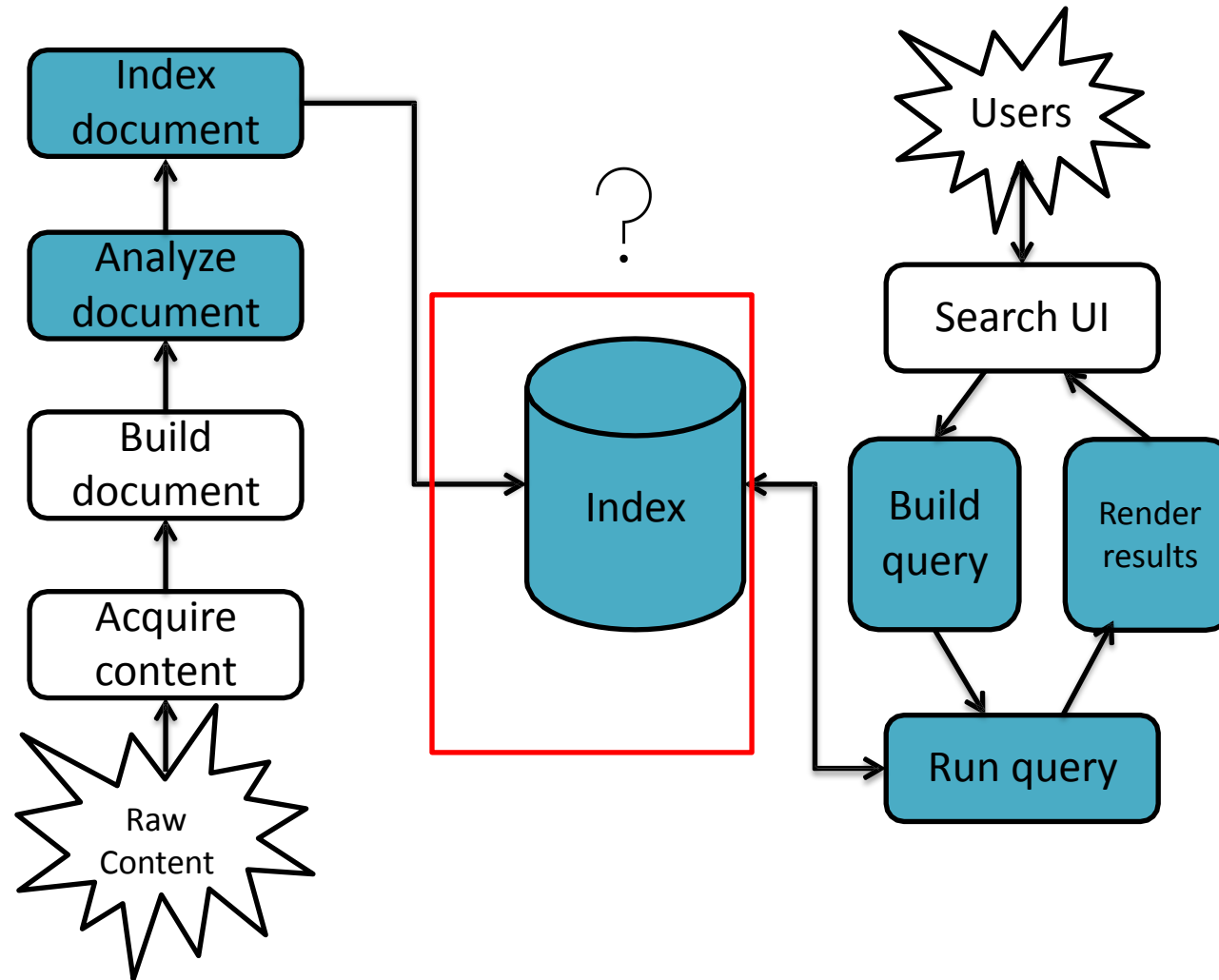
Inverted index with Payloads

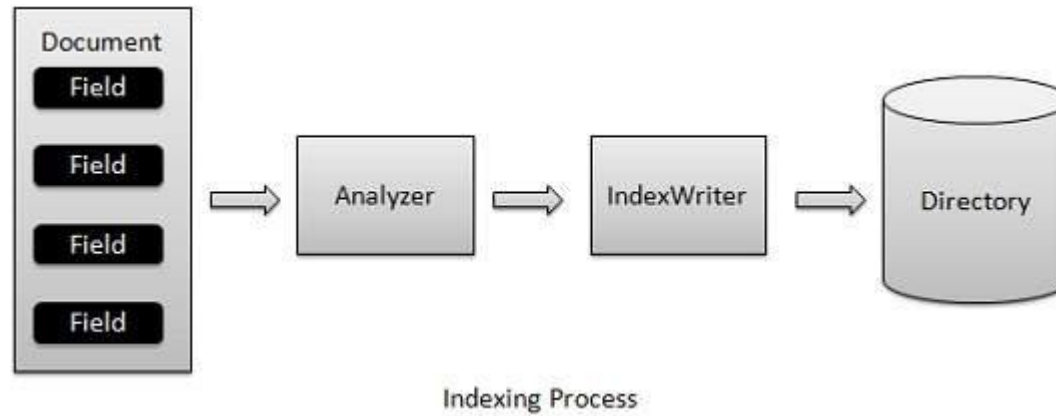


c:\docs\einstein.txt: 0
The 0 important 1 thing 2 is 3
not 4 to 5
stop 6 questioning 7.

c:\docs\shakespeare.txt: 1
To 0 be 1 or 2 not 3 to 4 be 5.

Let's Revisit the Lucene Architecture





Implementation of Searching with Lucene

Let's explore the set of utility classes available for indexing with Lucene toolkit

Lucene: Core **indexing** classes

- IndexWriter
- Document
- Analyzer
- Field
- Directory

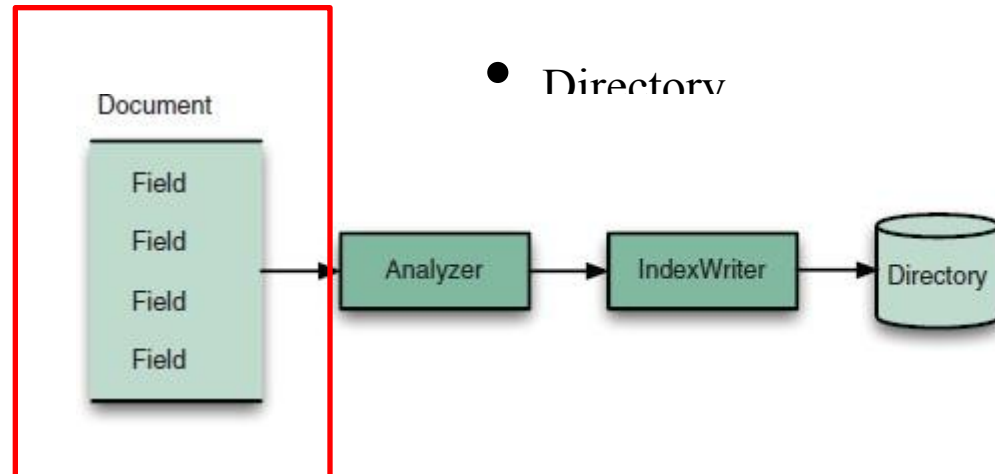


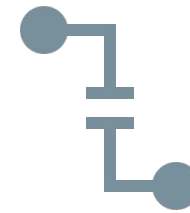
Figure 1.5 Classes used when indexing documents with Lucene

Lucene: Core **indexing** Document class



Document

Represents a collection of named Fields.
Text in these Fields are indexed.



Field

This is the lowest unit or the starting point of the indexing process. It represents the key value pair relationship where a key is used to identify the value to be indexed.

Lucene: Core **indexing** classes



IndexWriter

This class acts as a core component which creates/updates indexes during the indexing process.



Directory

This class represents the storage location of the indexes.



Analyzer

This class analyzes a document and get the tokens/words from the text to be indexed.

Analizers In Lucene

WhitespaceAnalyzer

- Splits tokens on whitespace

SimpleAnalyzer

- Splits tokens on non-letters, and then lowercases

StopAnalyzer

- Same as SimpleAnalyzer, but also removes stop words

KeywordAnalyzer

StandardAnalyzer

- Most sophisticated analyzer that knows about certain token types, lowercases, removes stop words, stems...

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]

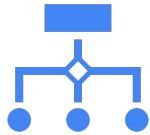
StopAnalyzer

- [quick] [brown] [fox] [jumped] [over] [lazy]
[dog]

StandardAnalyzer

- [quick] [brown] [fox] [jump] [over] [lazy] [dog]

Lucene: Core **searching** classes



IndexSearcher

This class acts as a core component which reads/searches indexes created after the indexing process. It takes directory instance pointing to the location containing the indexes.



Term

This class is the lowest unit of searching. It is like Field in indexing process.



Query

Query is an abstract class and contains various utility methods and is the parent of all types of queries that Lucene uses during search process.

Lucene: Core **searching** classes

TermQuery

- TermQuery is the most commonly-used query object and is the foundation of many complex queries that
- Lucene can make use of.

TopDocs

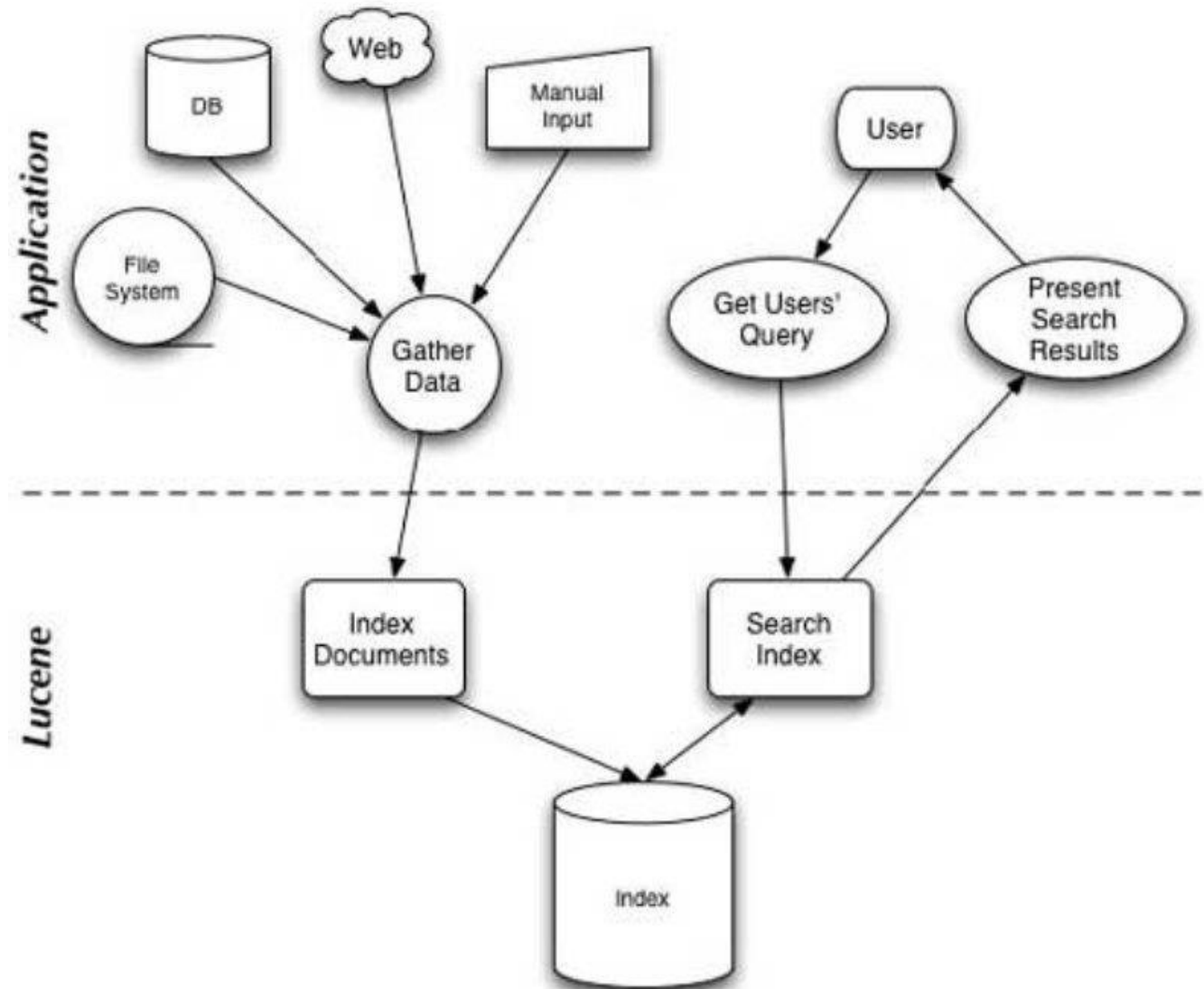
- TopDocs points to the top N search results which matches the search criteria. It is a simple container of pointers to point to documents which are the output of a search result.

```
// Define the term you want to search for in the "title" field
Term term = new Term("title", "Lucene");

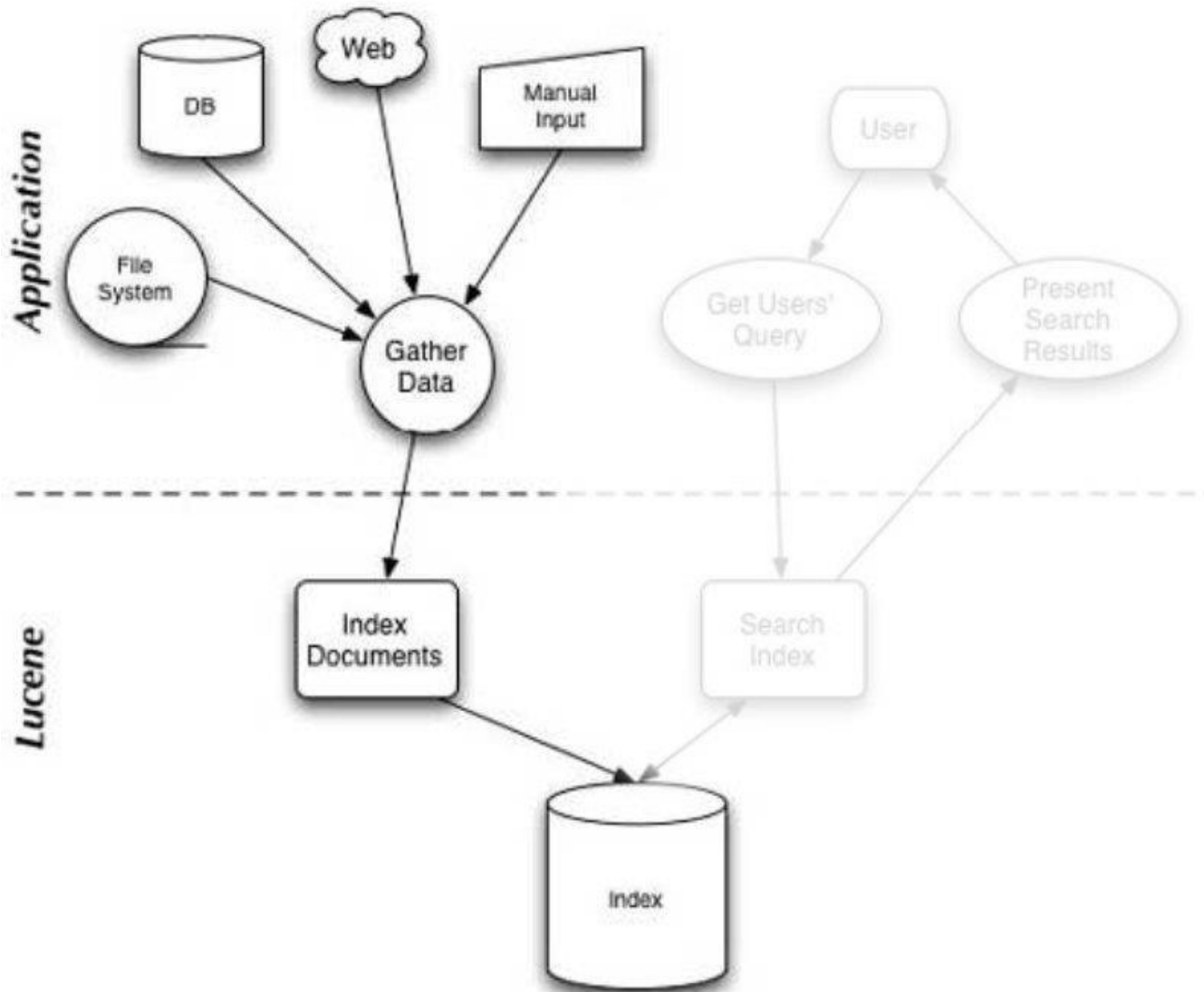
// Create the TermQuery using the term
Query query = new TermQuery(term);

// Perform the search and retrieve top 10 matching documents
TopDocs topDocs = searcher.search(query, 10);
```

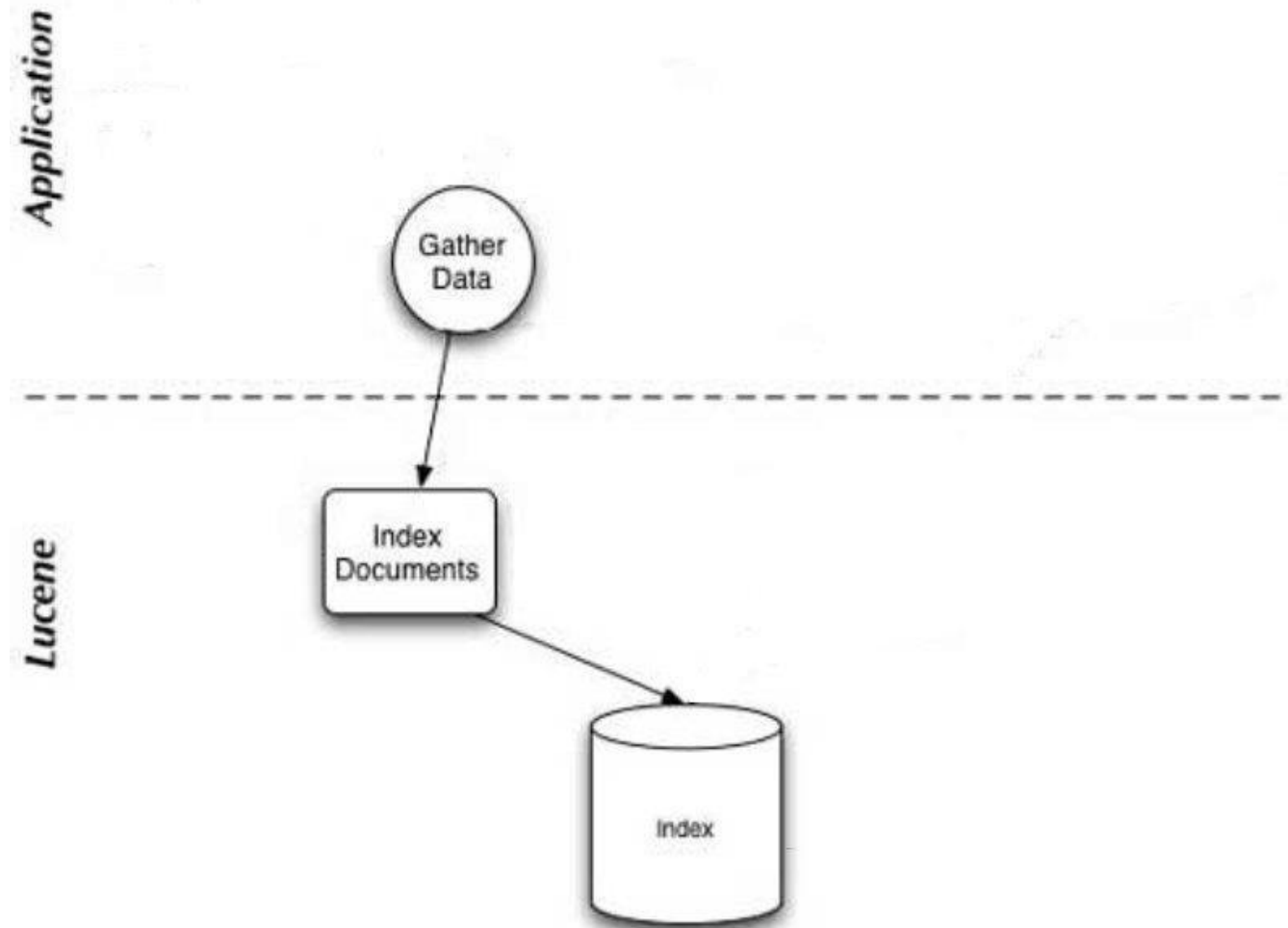
Lucene Implementation: Step By Step Walkthrough



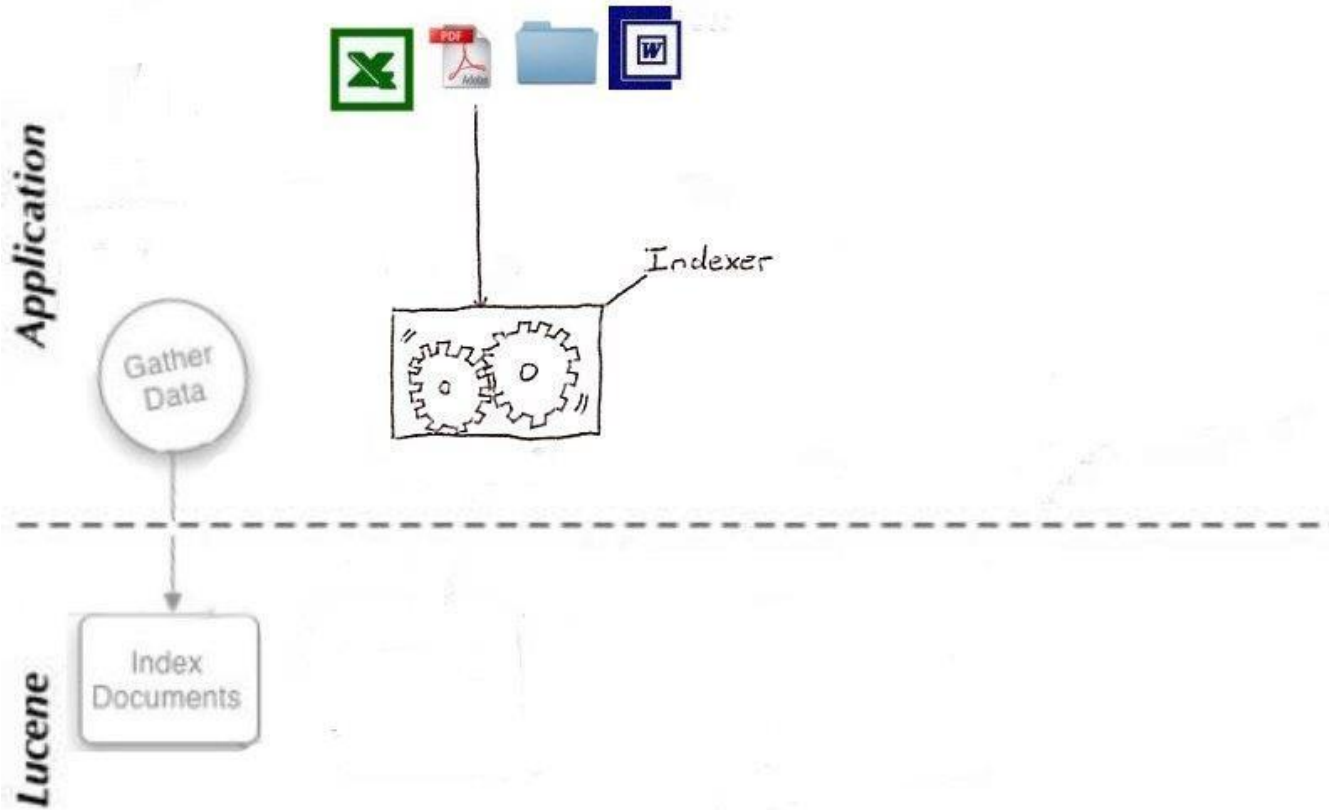
Lucene Implementation: Indexing



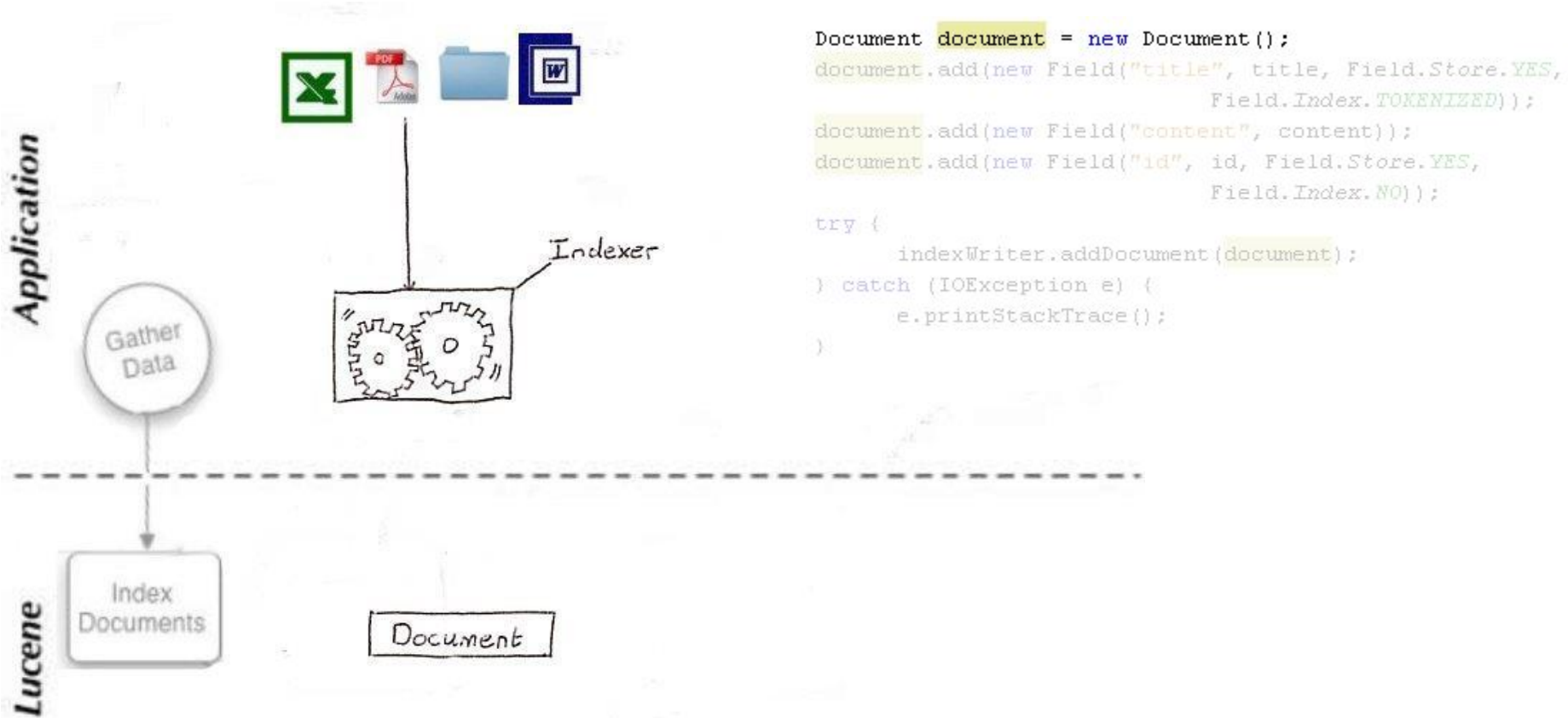
Lucene Implementation: Indexing



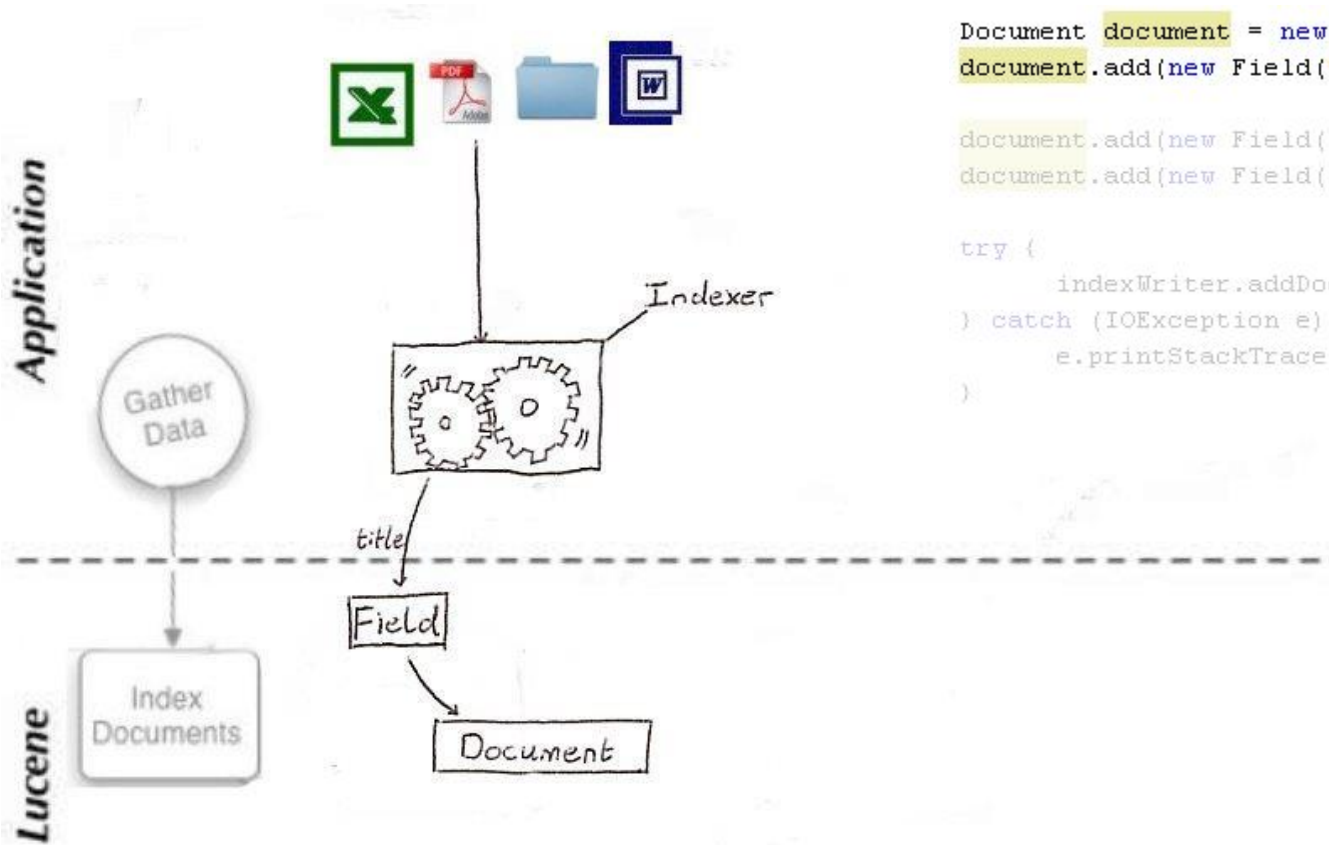
Lucene Implementation: Indexing



Lucene Implementation: Indexing



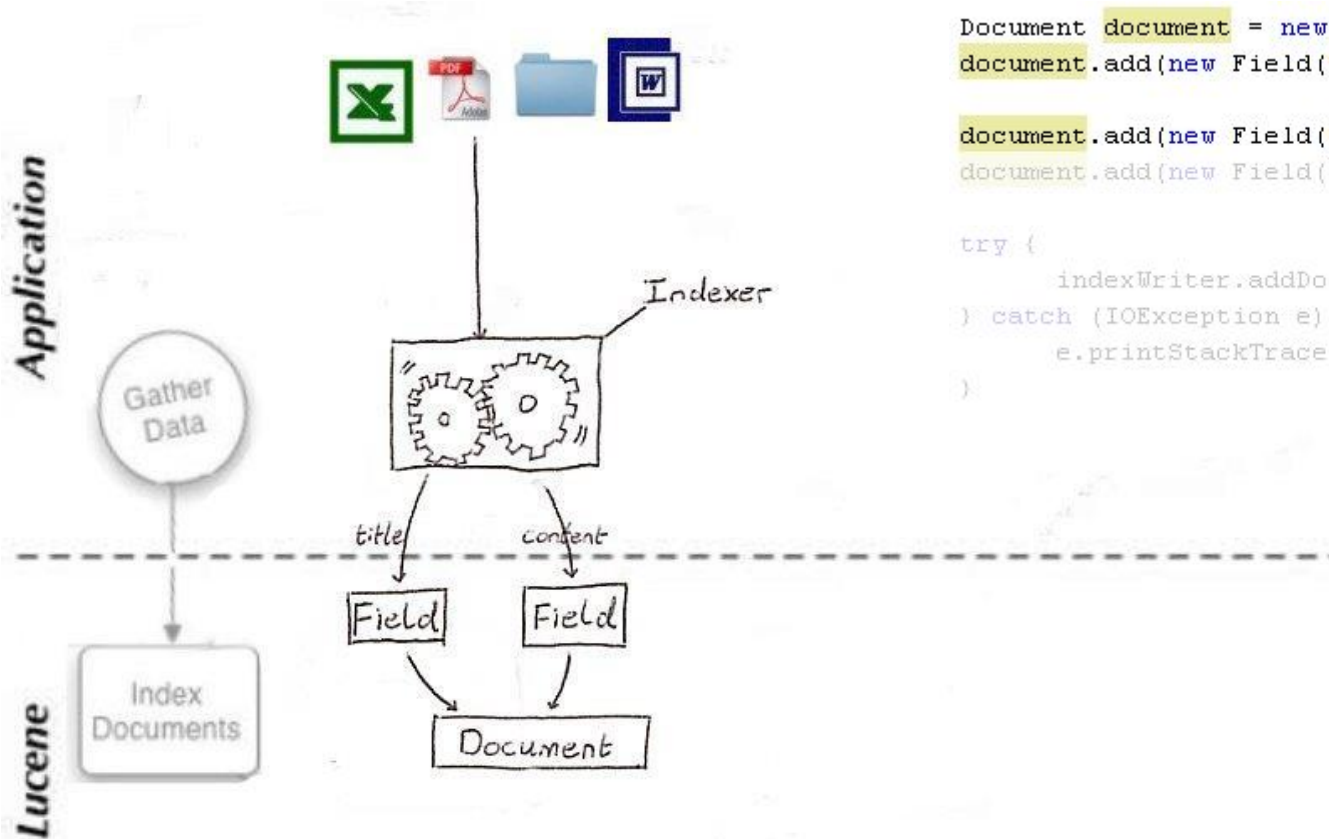
Lucene Implementation: Indexing



```
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();
}
```

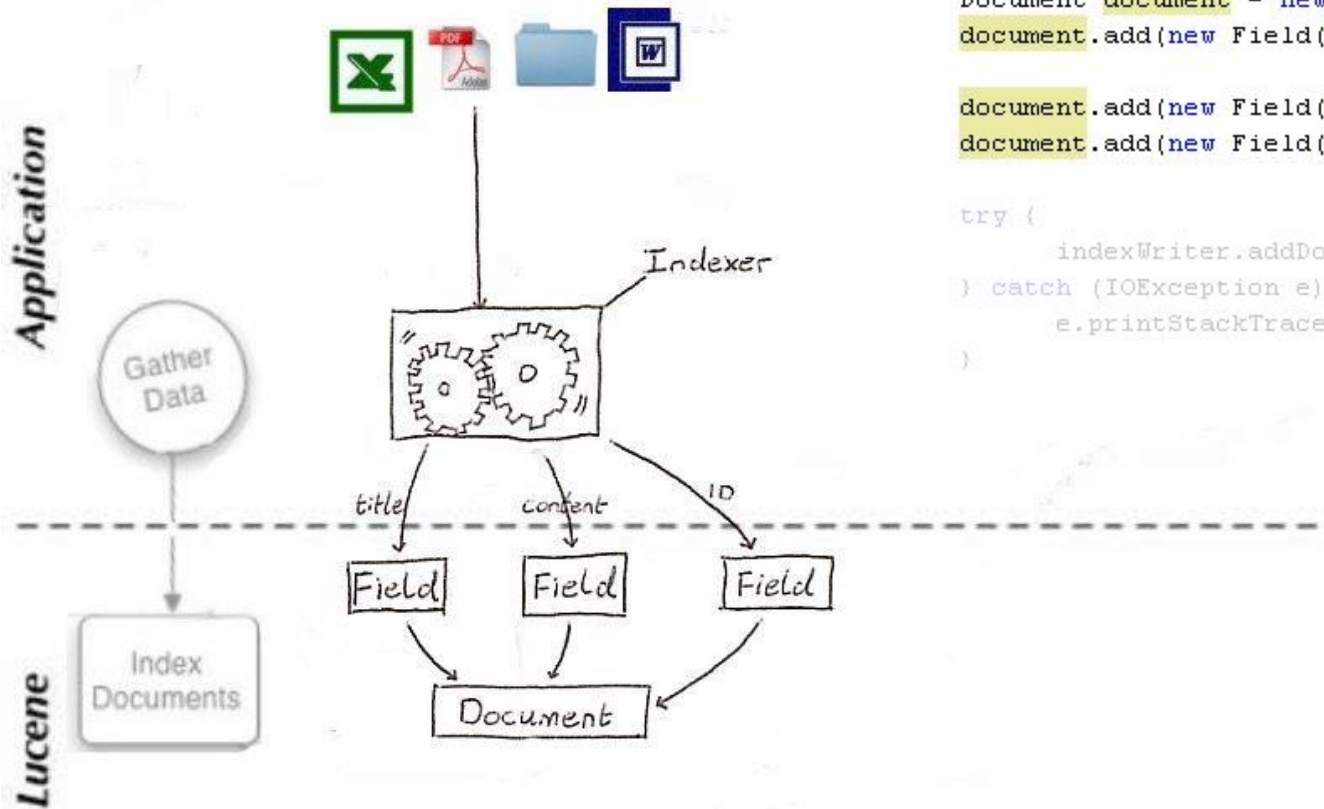
Lucene Implementation: Indexing



```
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();
}
```

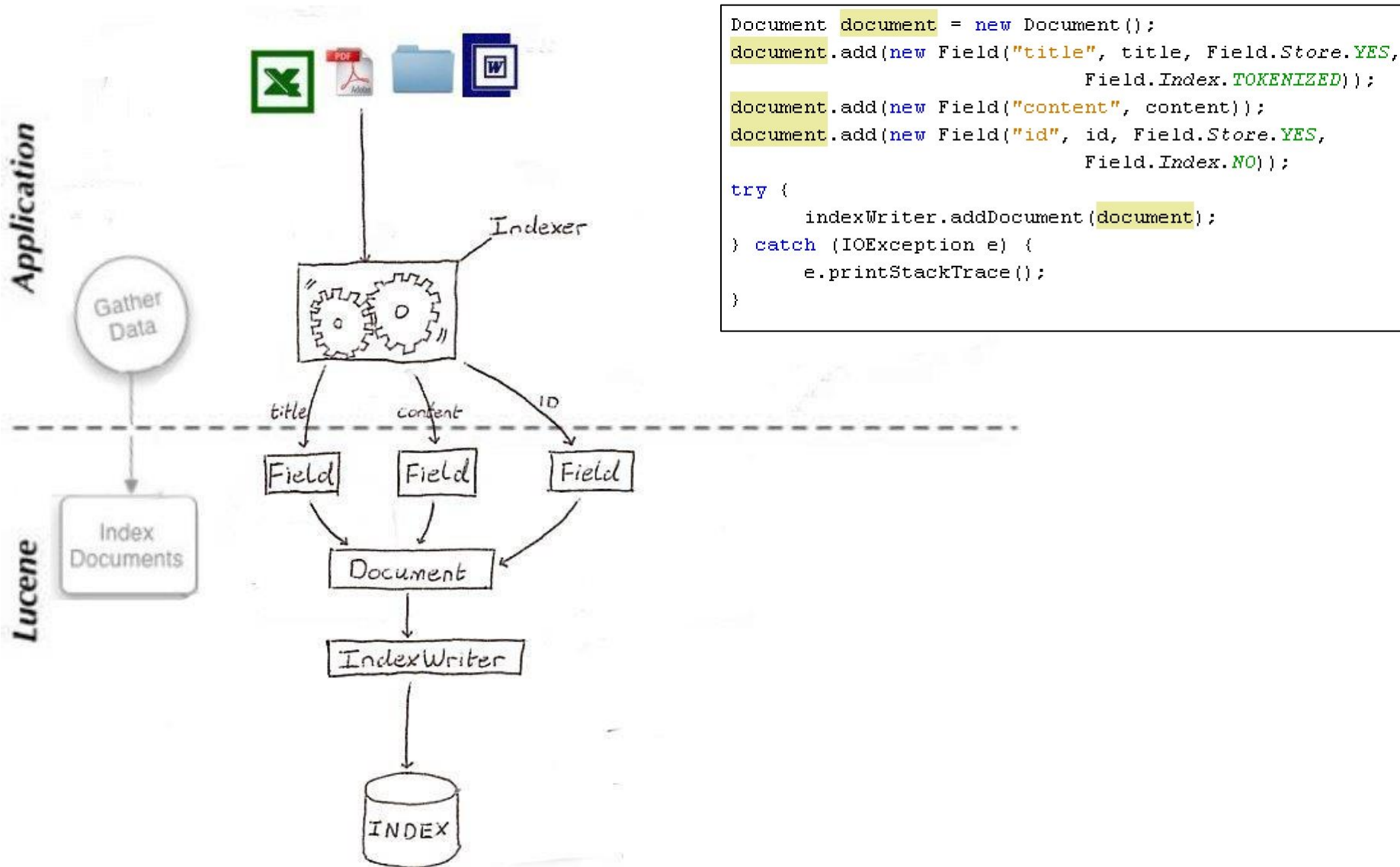
Lucene Implementation: Indexing



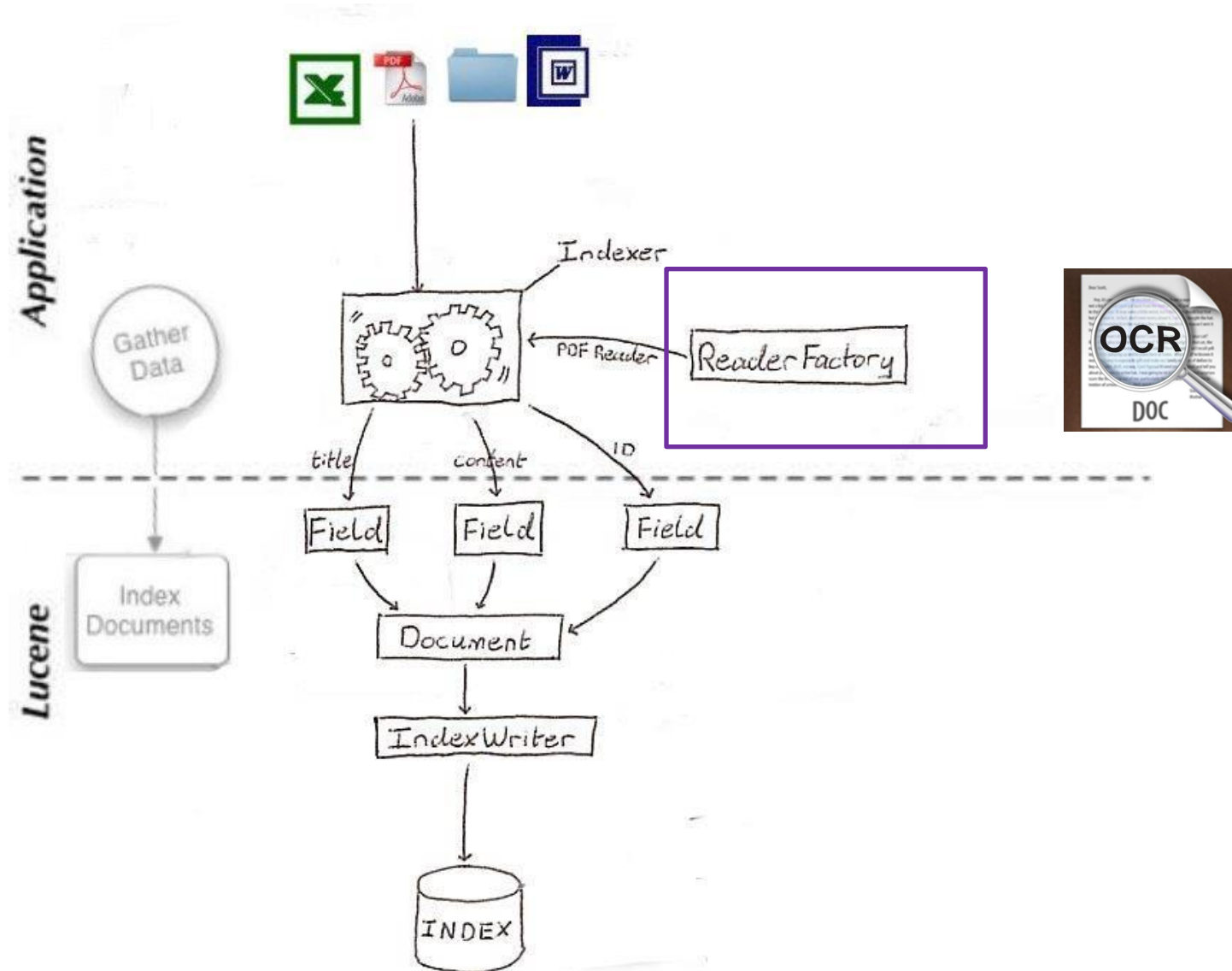
```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                      Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                      Field.Index.NO));

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

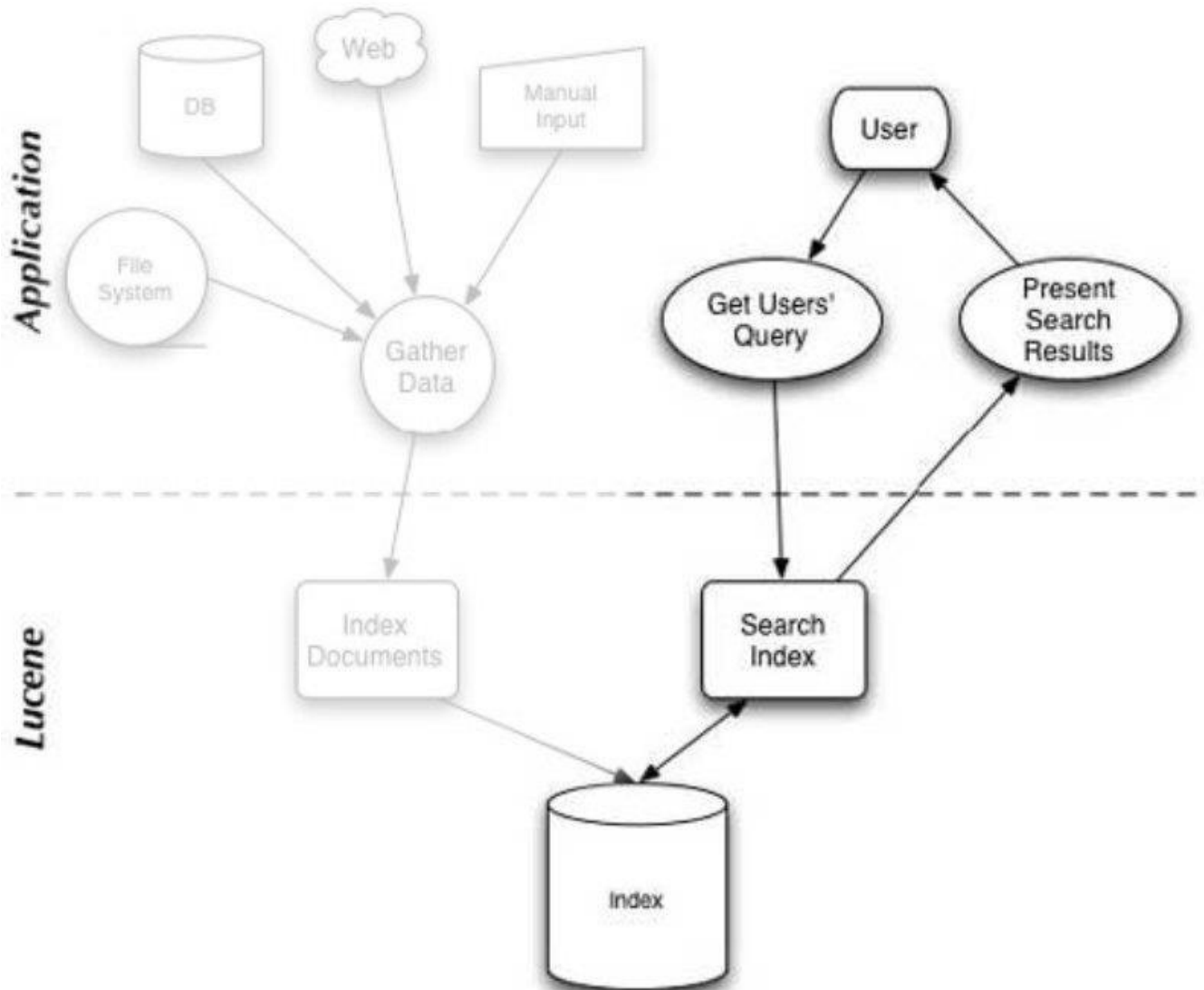
Lucene Implementation: Indexing



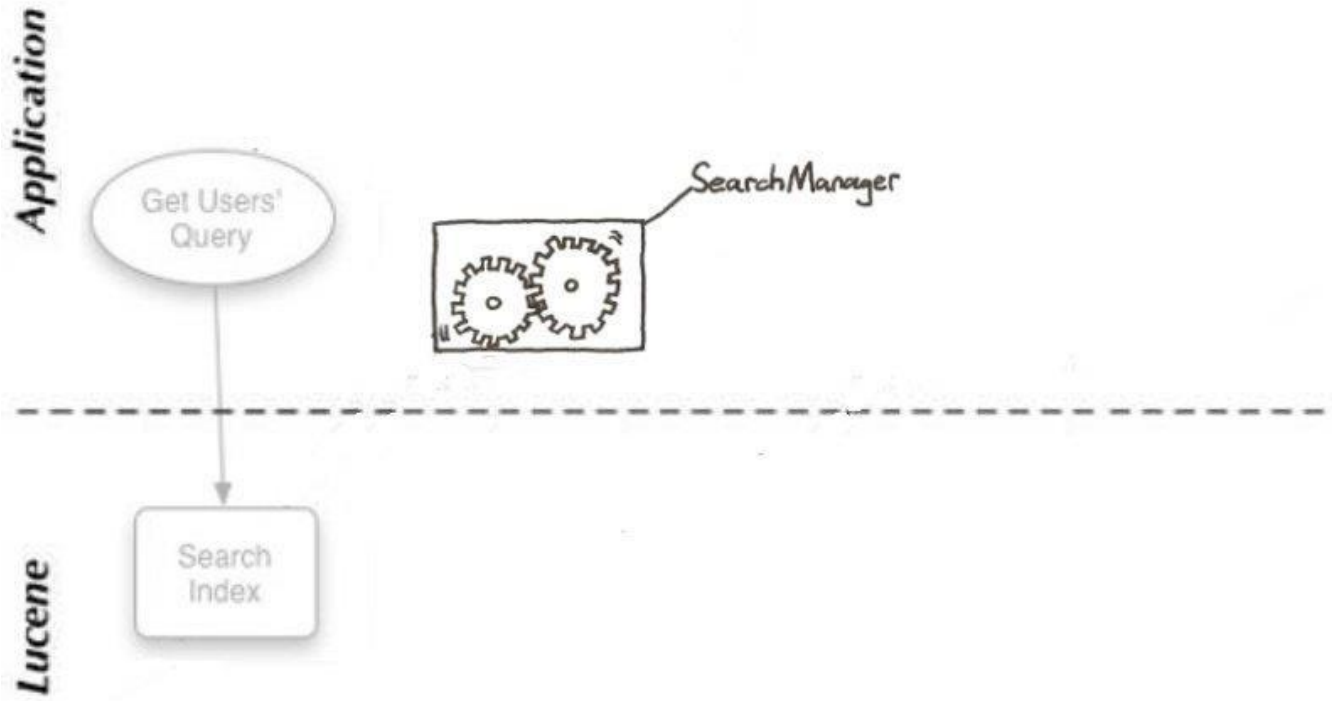
Lucene Implementation : Searching



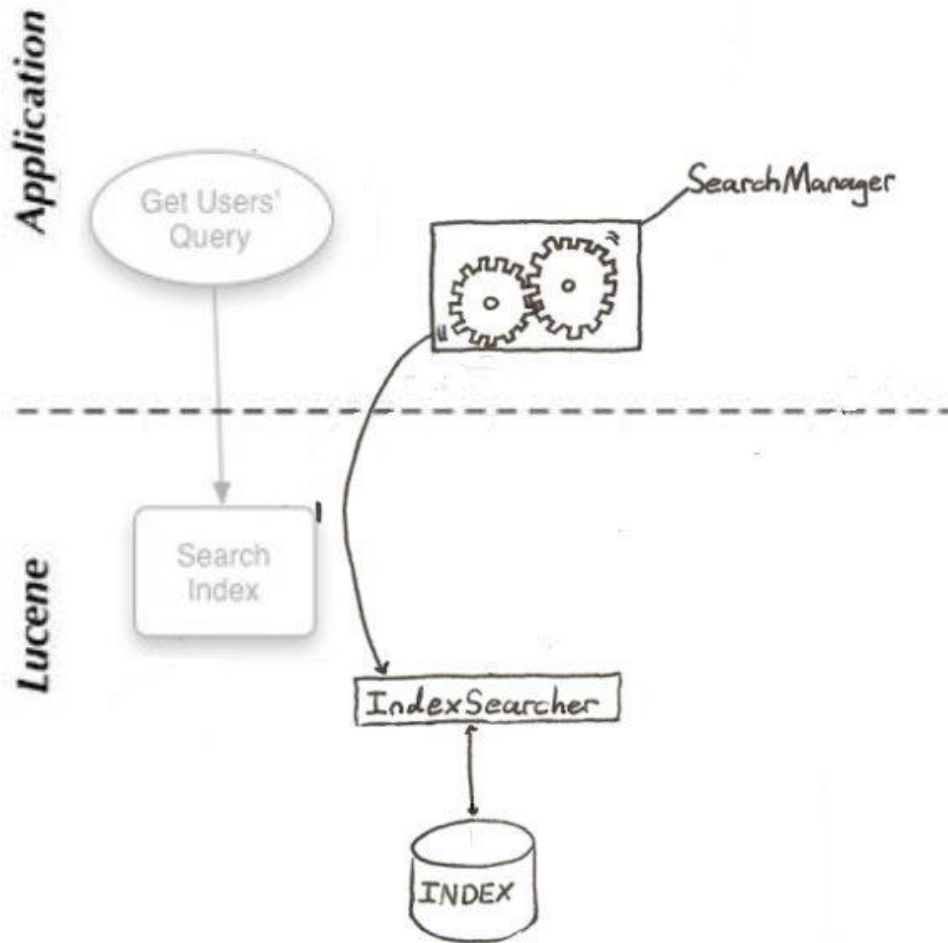
Lucene Implementation : Searching



Lucene Implementation : Searching

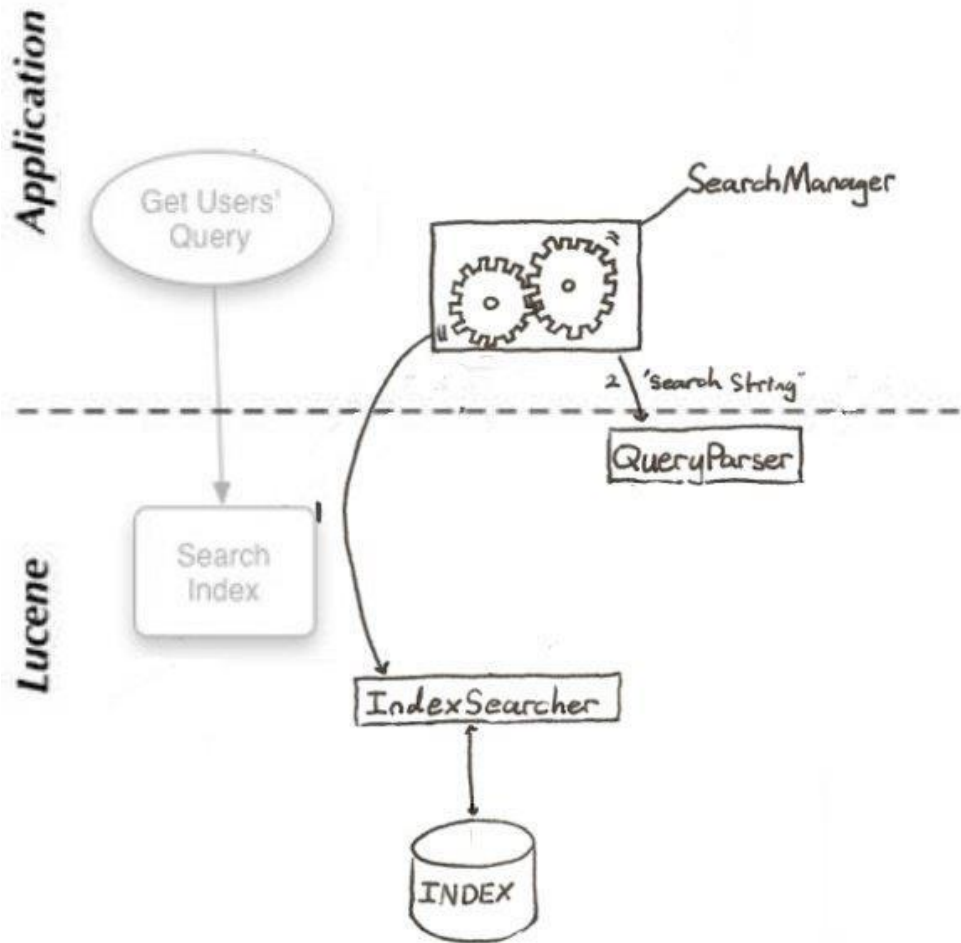


Lucene Implementation : Searching step 1 of 6



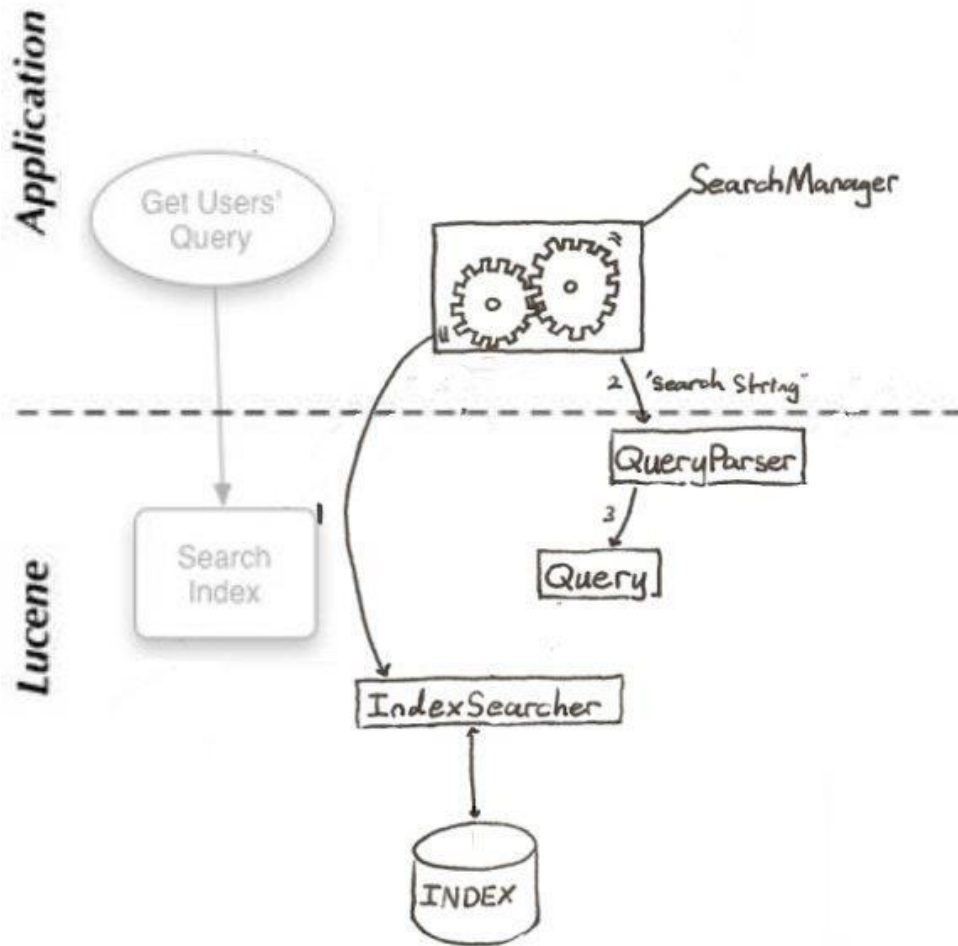
```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
```

Lucene Implementation : Searching step 2 of 6



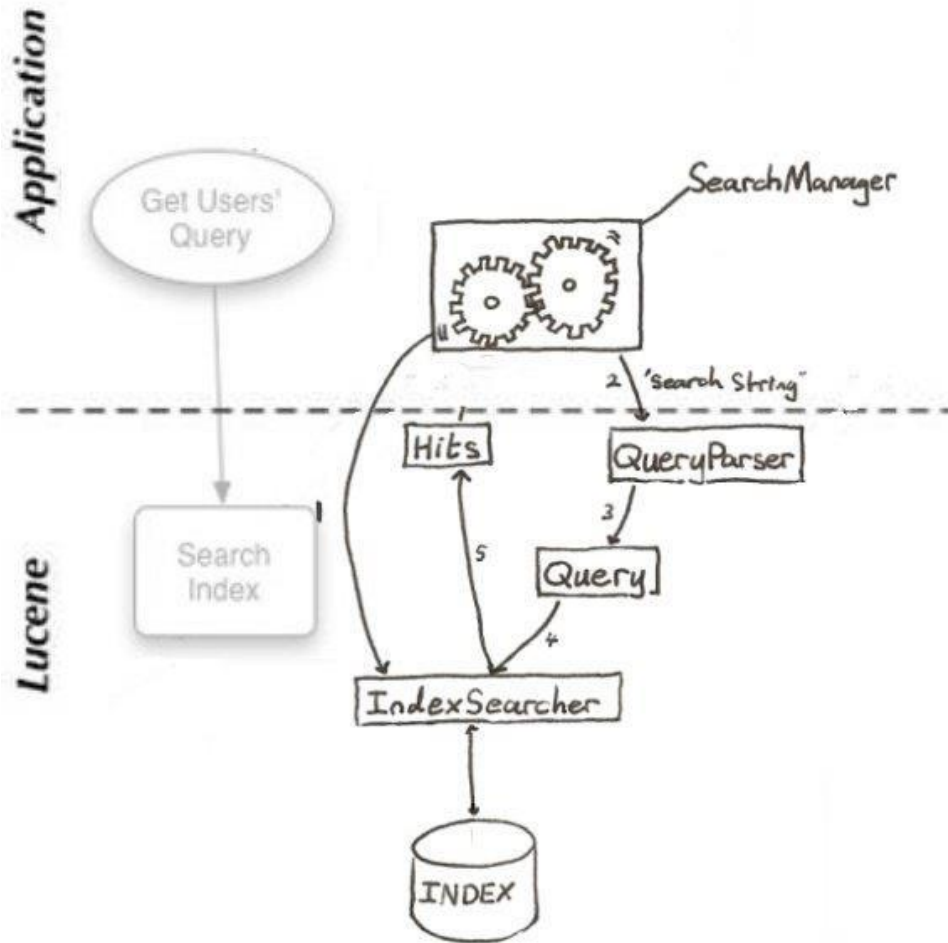
```
QueryParser queryParser = new QueryParser("content", analyzer);
```

Lucene Implementation : Searching step 3 of 6



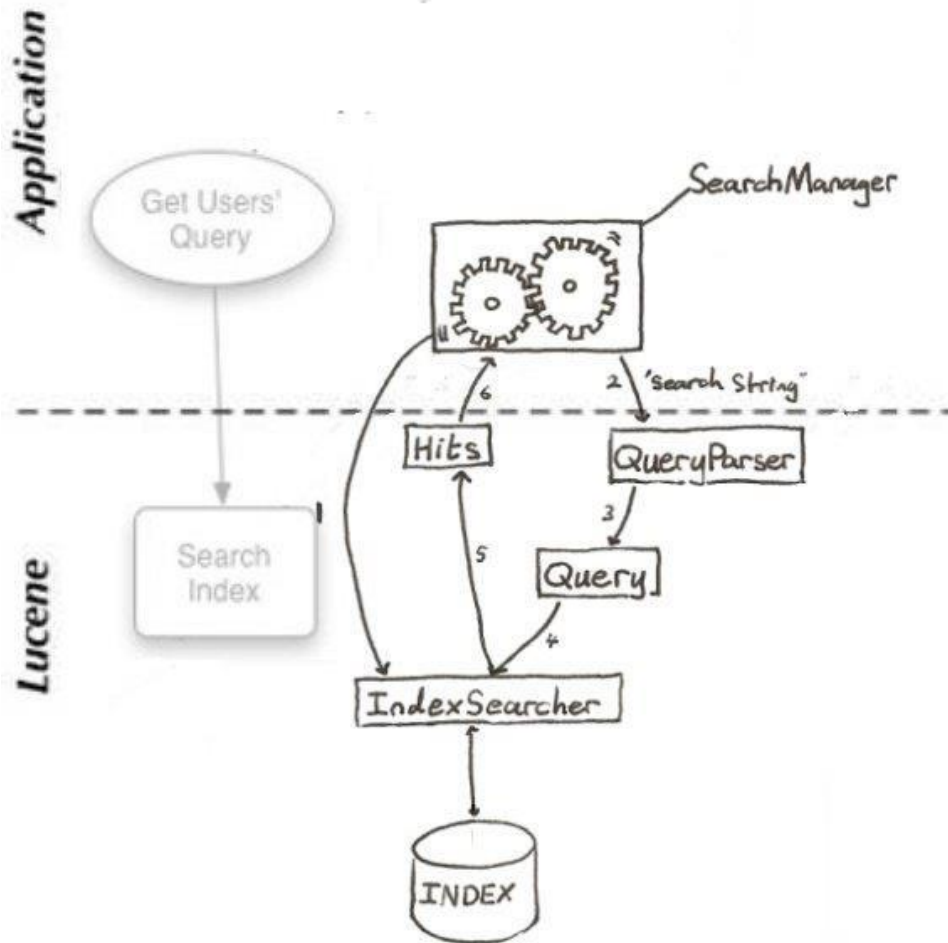
```
);  
Query query = null;  
try {  
    query = queryParser.parse("Search string");  
} catch (ParseException e) {  
    e.printStackTrace();  
}
```

Lucene Implementation : Searching step 4&5 of 6



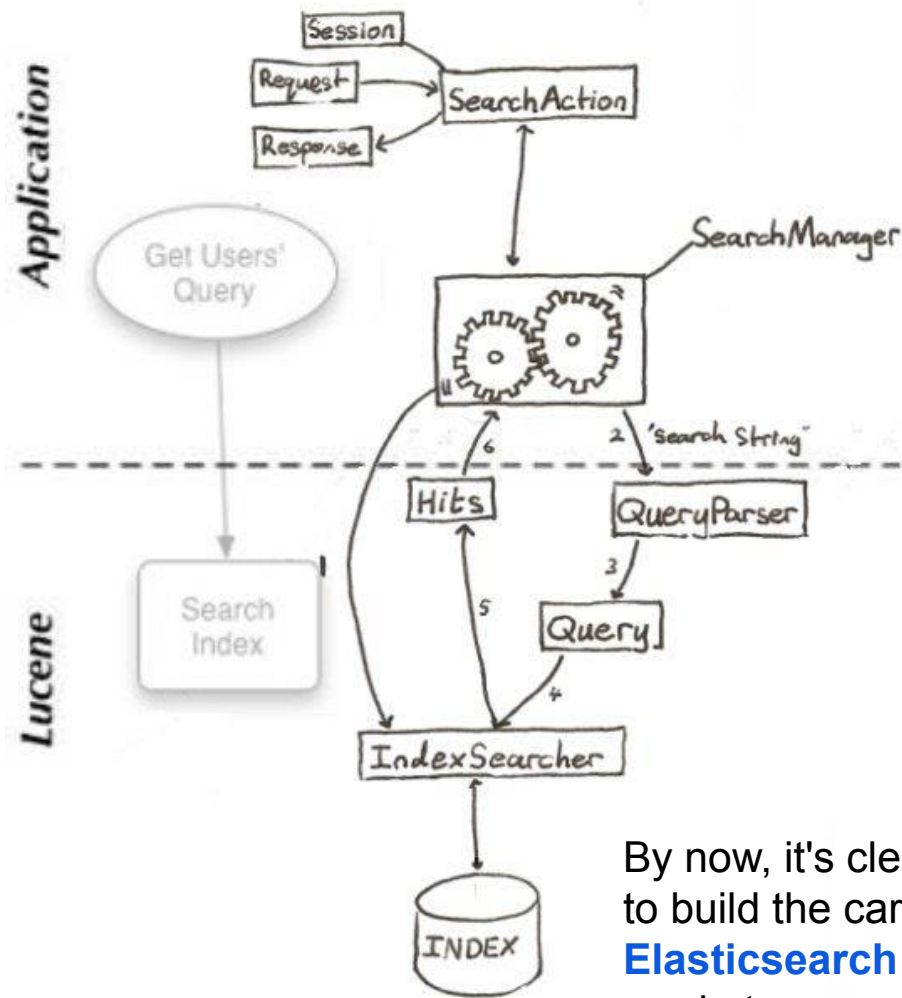
```
if(null != query && null != indexSearcher){  
    try {  
        Hits hits = indexSearcher.search(query);  
    }  
}
```

Lucene Implementation : 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();
    }
}
```

Lucene Implementation : 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();
    }
}
```

By now, it's clear: Lucene is a powerful engine, but you still need to build the car to start using it.

Elasticsearch does that for you — wrapping Lucene into a ready-to-use, scalable search server with APIs and clustering.

Distributed Searching with Elasticsearch

Elasticsearch is a highly scalable, open-source full-text search and analytics engine.

It enables fast, near real-time storage, search, and analysis of large volumes of data.

Often used as the backbone for applications with complex search requirements.



elasticsearch

Primary Elasticsearch Concepts



Elasticsearch delivers fast search responses by querying an **index** instead of the raw text, enabling quick document retrieval.



It's like looking up a keyword in a book's index to find relevant pages, rather than scanning every word on every page.




Elasticsearch uses Apache Lucene to create and manage this inverted index.

Data Representation in Elasticsearch

In Elasticsearch, a **Document** is the unit of search and index.



An index consists of one or more Documents, and a Document consists of one or more Fields.



In database terminology, a Document corresponds to a table row, and a Field corresponds to a table column.



Coming Up This Week

Exploring the ELK Stack

Get ready to dive into:

Elasticsearch: Powerful search & analytics engine

Logstash: Data collection & transformation pipeline

Kibana: Visualization and dashboard tool



We'll learn how they work together to process, store, and visualize large volumes of data in real-time.



Hands-on demos included!

Note : Please review the Final Assessment Guide thoroughly and reach out to your tutor if you have any questions or need clarification!