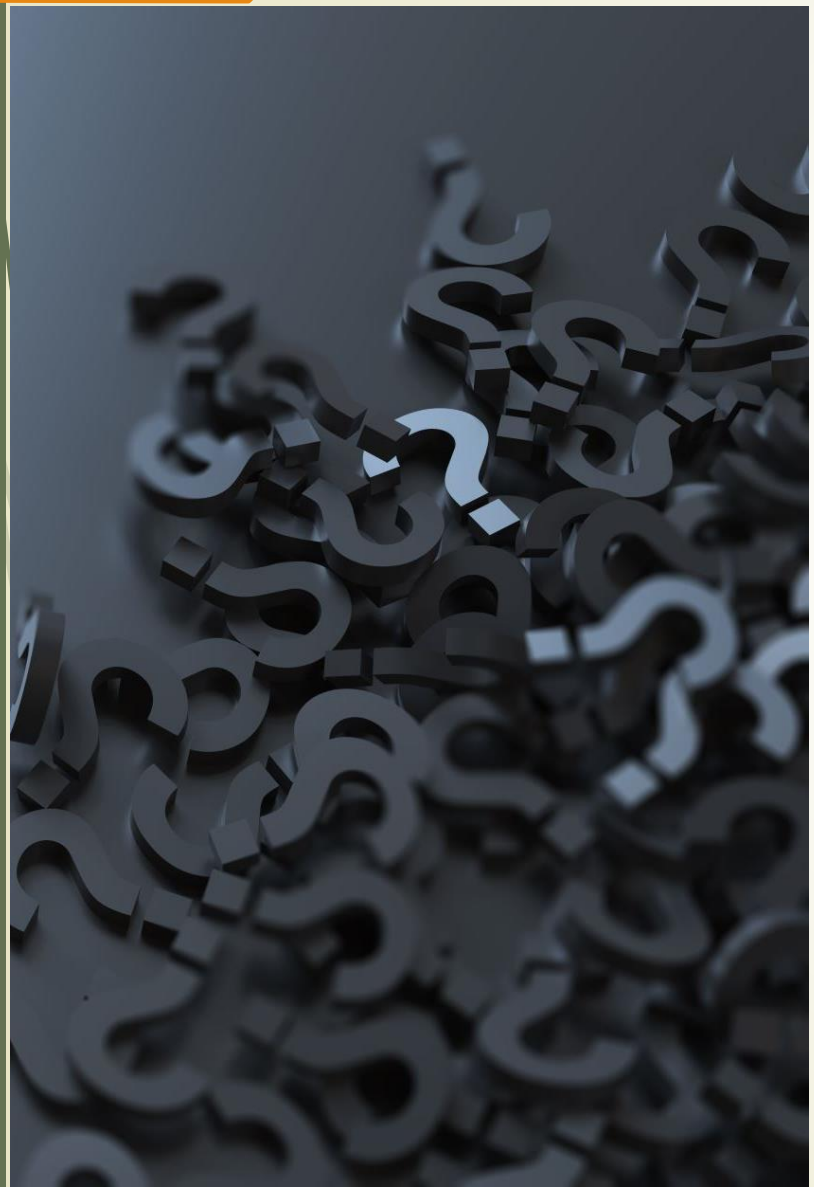
Chapter 5
Searching and Indexing

Bal Krishna Nyaupane
Assistant Professor
Department of Electronics and Computer Engineering
Institute of Engineering, Tribhuvan University
bkn@wrc.edu.np

Indexing



- To search ***large amounts of text quickly***, one must first ***index that text*** and convert it into a format that will let one search it rapidly, eliminating the slow sequential scanning process. This conversion process is ***called indexing***, and its ***output is called an index***.
- Indexing is the initial part of all search applications.
- Goal of indexing is to process the original ***data into a highly efficient cross-reference lookup*** in order to facilitate rapid searching.
- The job is simple when the ***content is already textual in nature*** and its location is known.

Searching

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

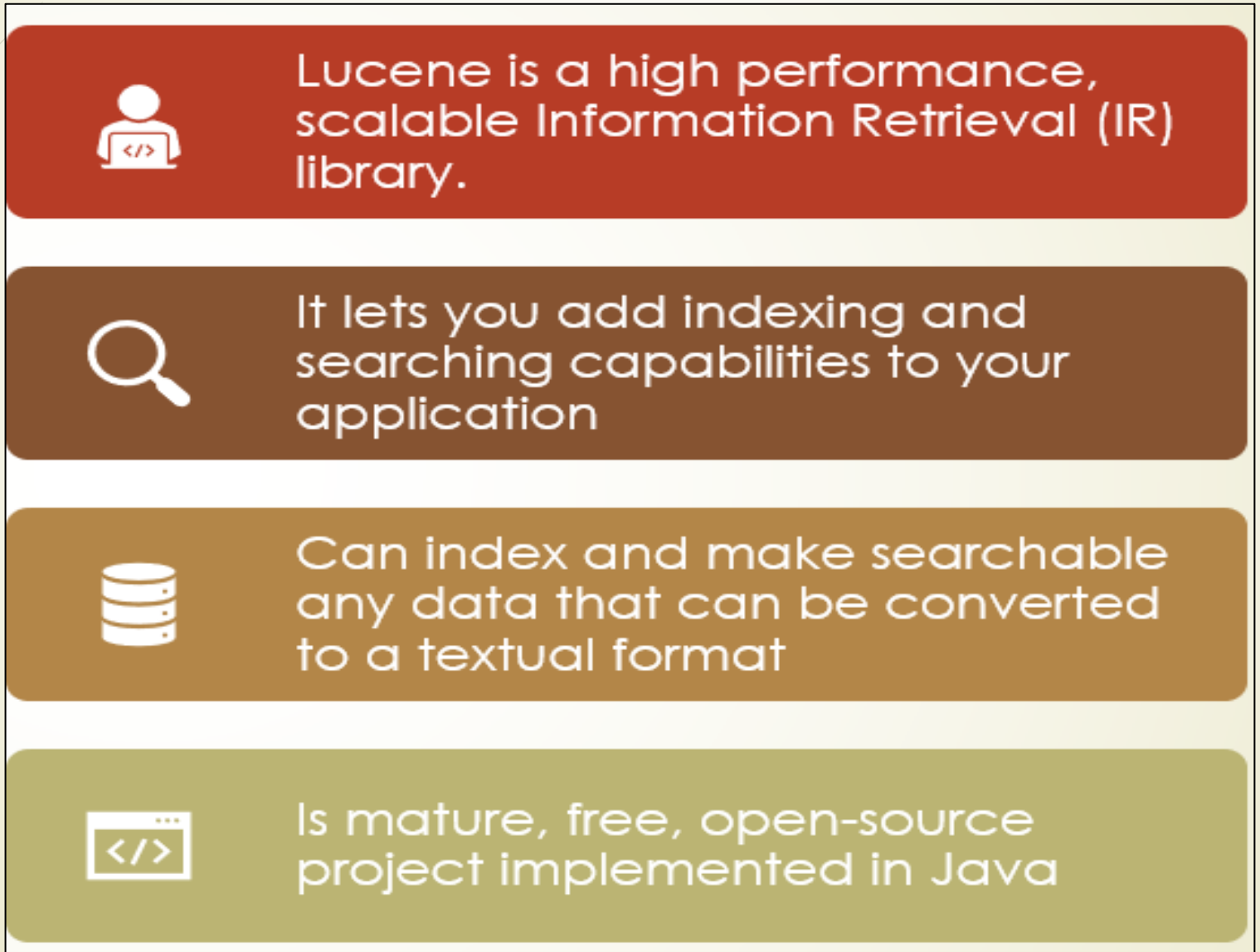
What is Lucene?

- Apache Lucene is a ***free and open-source search engine*** software library, originally written in ***Java by Doug Cutting***.
- It is supported by the ***Apache Software Foundation*** and is released under the ***Apache Software License***. Lucene is widely used as a standard foundation for non-research search applications.
- Lucene has been ported to ***other programming languages*** including ***Object Pascal, Perl, C#, C++, Python, Ruby and PHP***.
- Lucene is the search core of both ***Apache Solr™ and Elasticsearch™.***
- Lucene ***Core is a Java library*** providing powerful ***indexing and search features***, as well as ***spellchecking, hit highlighting and advanced analysis/tokenization*** capabilities.

What is Lucene?

- Lucene is a ***high-performance, scalable information retrieval (IR) library.***
- Lucene lets you ***add searching capabilities to your applications***. It's a mature, free, open source project implemented in Java, and a project in the Apache Software Foundation.
- Lucene's website, at ***http://lucene.apache.org/java*** , is a great place to learn more about the current status of Lucene.
- There you'll find the tutorial, Javadocs for Lucene's API for all recent releases, an issue-tracking system, links for downloading releases, and Lucene's wiki (***http://wiki.apache.org/lucene-java***), which contains many community-created and -maintained pages.

What is Lucene?



Who use Lucene?

Lucene Architecture

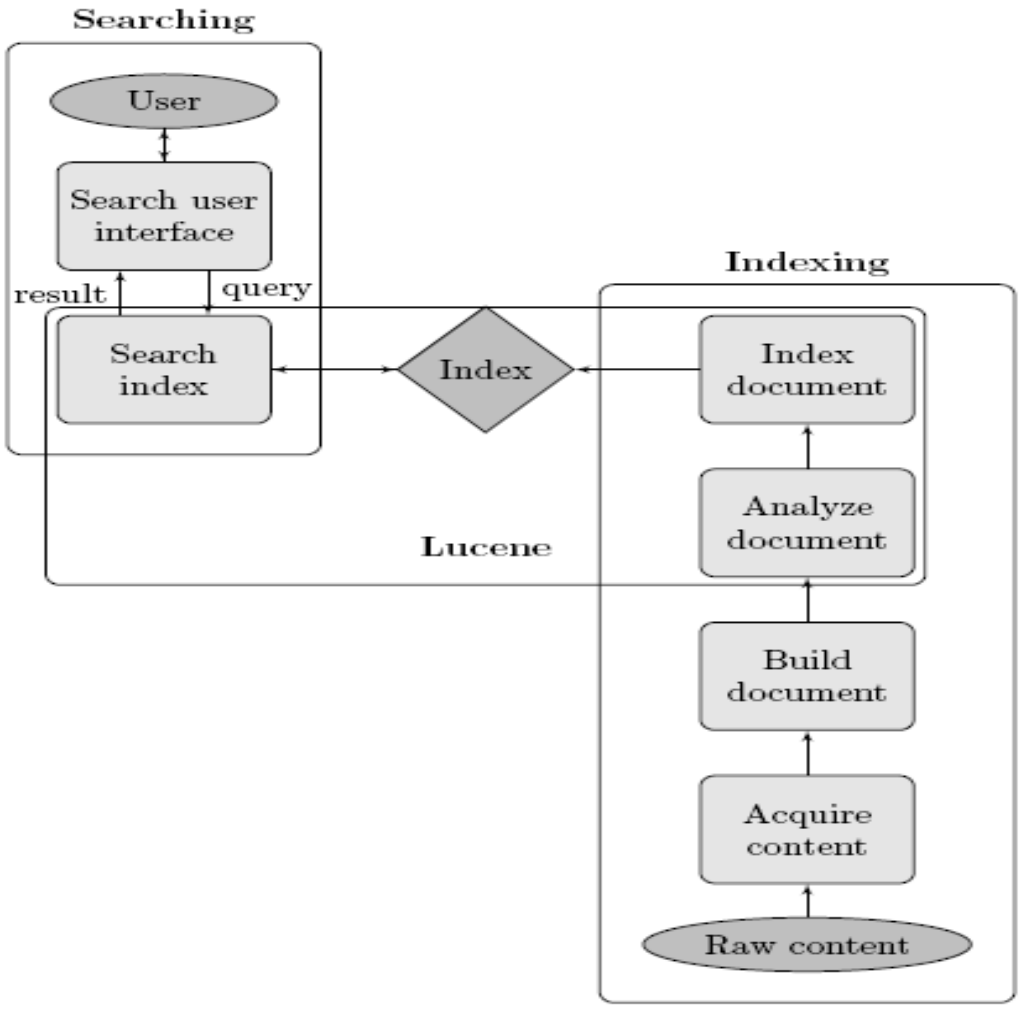
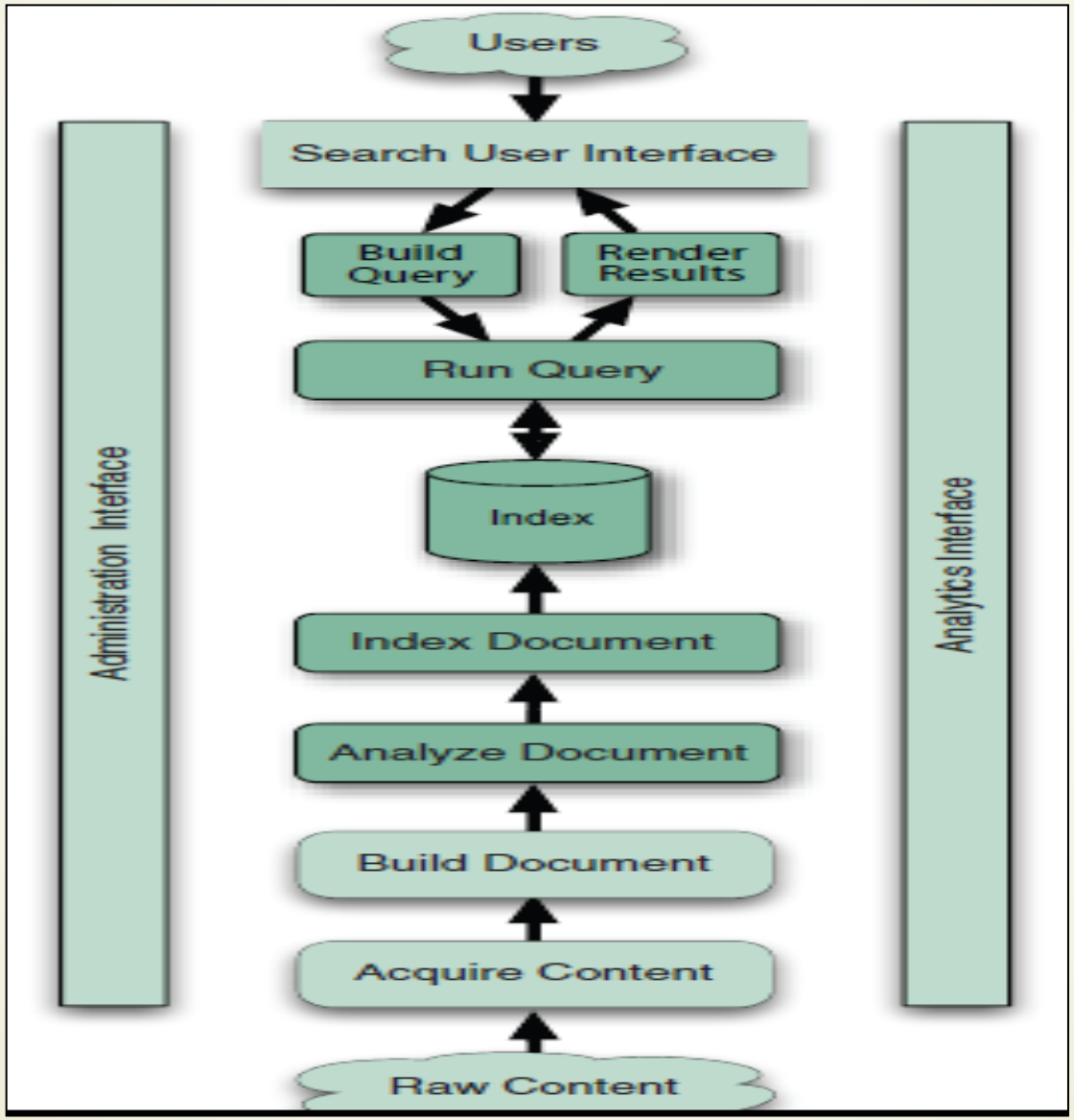


Figure 3.1: Typical components of search application architecture with Lucene components highlighted

Typical Components of Search Application



Typical Components of Search Application

- A ***common misconception*** is that Lucene is an ***entire search application***, when in fact it's ***simply the core indexing and searching component***.
- We'll see that a search application starts with an indexing chain, which in turn requires separate steps to retrieve the raw content; ***create documents from the content***, possibly ***extracting text from binary documents***; and ***index the documents***.
- ***Once the index is built***, the components required for ***searching*** are equally diverse, ***including a user interface, a means for building up a programmatic query, query execution*** (to retrieve matching documents), and ***results rendering.***

How Search Application works?

Acquire Raw Content	The first step of any search application is to collect the target contents on which search application is to be conducted.
Build the document	The next step is to build the document(s) from the raw content, which the search application can understand and interpret easily.
Analyze the document	Before the indexing process starts, the document is to be analyzed as to which part of the text is a candidate to be indexed. This process is where the document is analyzed.

How Search Application works?

Indexing the document

Once documents are ***built and analyzed***, the next step is to ***index them*** so that this document can be ***retrieved based on certain keys*** instead of the entire content of the document. Indexing process is similar to indexes in the end of a book ***where common words are shown with their page numbers*** so that these words can be tracked quickly instead of searching the complete book.

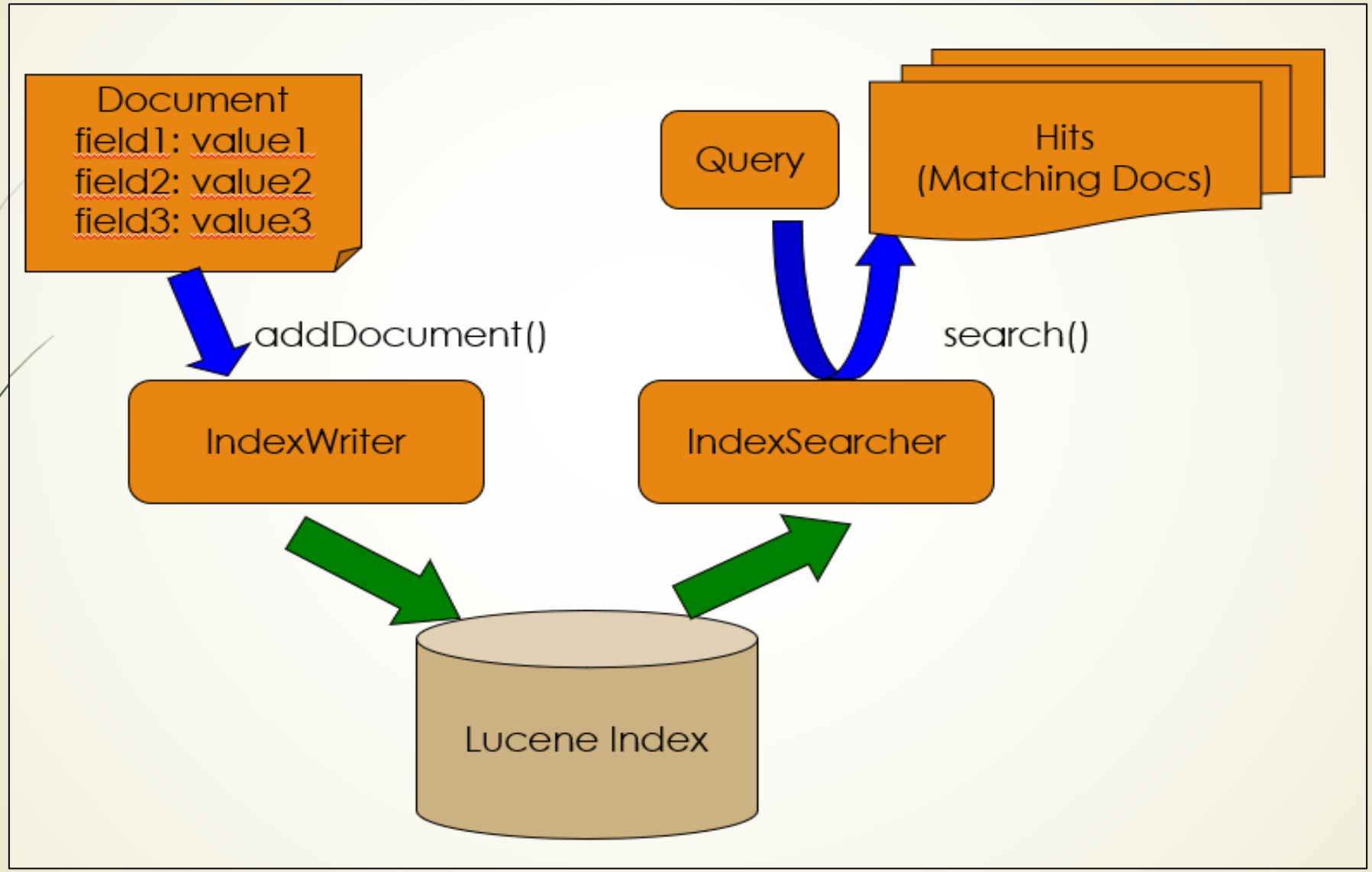
User Interface for Search

Once a database of indexes is ready then the application can make any search. To facilitate a user to make a search, the application must provide a ***user a mean*** or a ***user interface*** where a user can enter text and start the search process.

How Search Application works?

Build Query	Once a user makes a request to search a text, the application should prepare a Query object using that text which can be used to inquire index database to get the relevant details.
Search Query	Using a query object, the index database is then checked to get the relevant details and the content documents.
Render Results	Once the result is received, the application should decide on how to show the results to the user using User Interface. How much information is to be shown at first look and so on.

Basic Application



Inverted Index

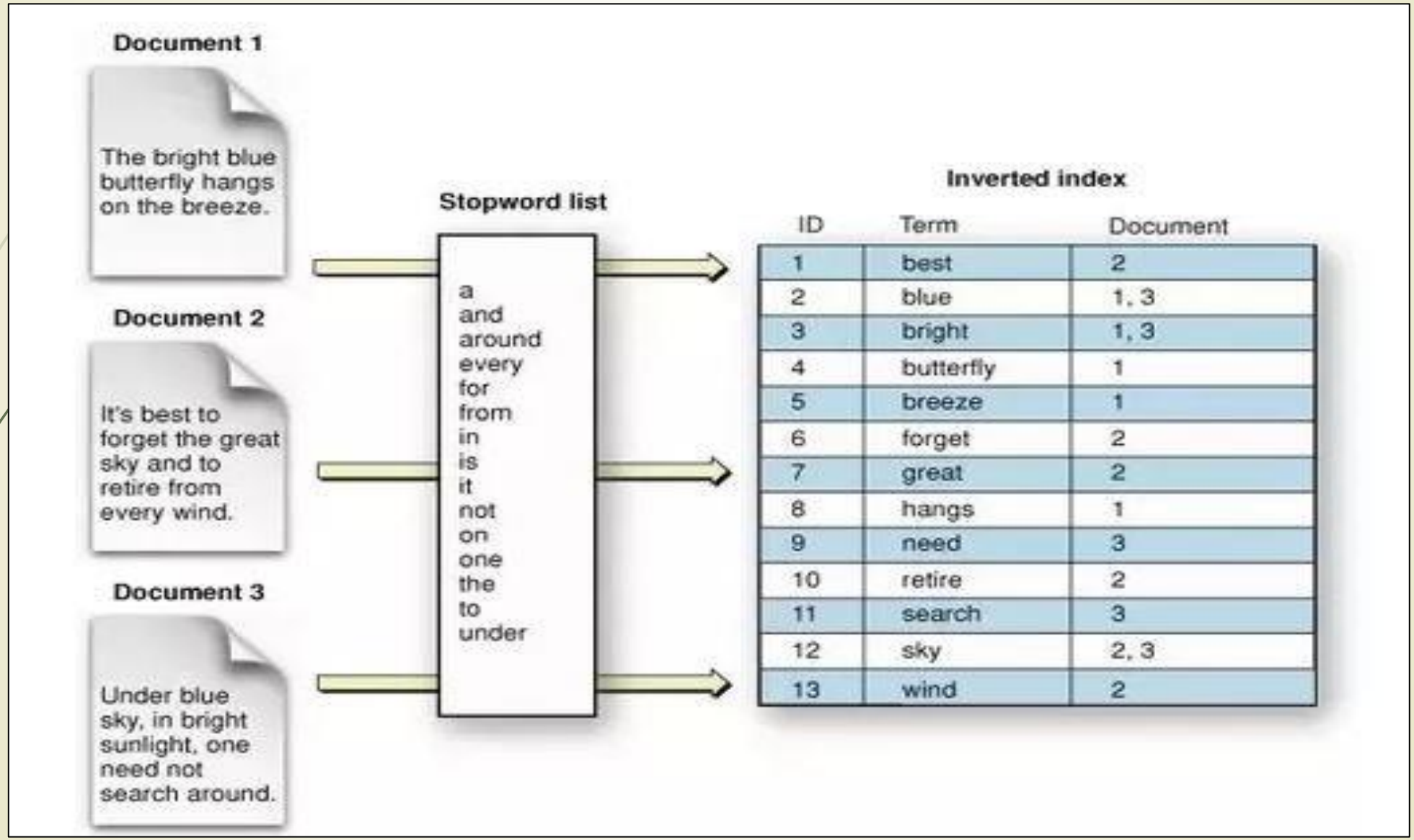
- An inverted index is an ***index data structure*** storing ***a mapping from content***, such ***as words or numbers, to its locations in a document or a set of documents***.
- The purpose of an ***inverted index is to allow fast full-text searches***, at a cost of increased processing when a document is added to the database.
- The ***inverted file may be the database file itself***, rather than its index. ***It is the most popular data structure used in document retrieval systems***, used on a large scale for example in search engines.
- Additionally, several significant general-purpose mainframe-based database management systems have used inverted list architectures.

Inverted Index

Document	Text
1	Pease porridge hot, pease porridge cold
2	Pease porridge in the pot
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot
6	Nine days old

Text	(Document; Word)
cold	(1; 6), (4; 8)
days	(3; 2), (6; 2)
hot	(1; 3), (4; 4)
in	(2; 3), (5; 4)
it	(4; 3, 7), (5; 3)
like	(4; 2, 6), (5; 2)
nine	(3; 1), (6; 1)
old	(3; 3), (6; 3)
pease	(1; 1, 4), (2; 1)
porridge	(1; 2, 5), (2; 2)
pot	(2; 5), (5; 6)
some	(4; 1, 5), (5; 1)
the	(2; 4), (5; 5)

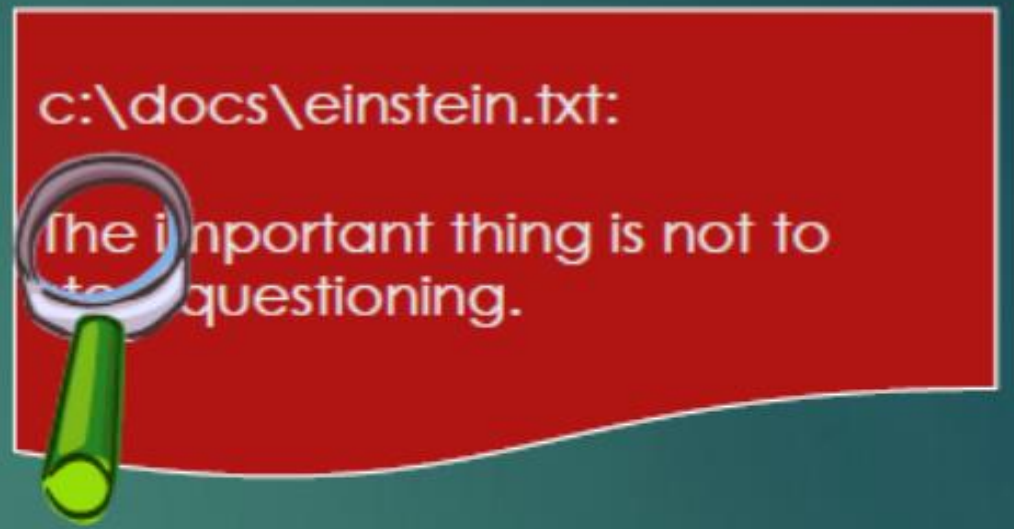
Inverted Index

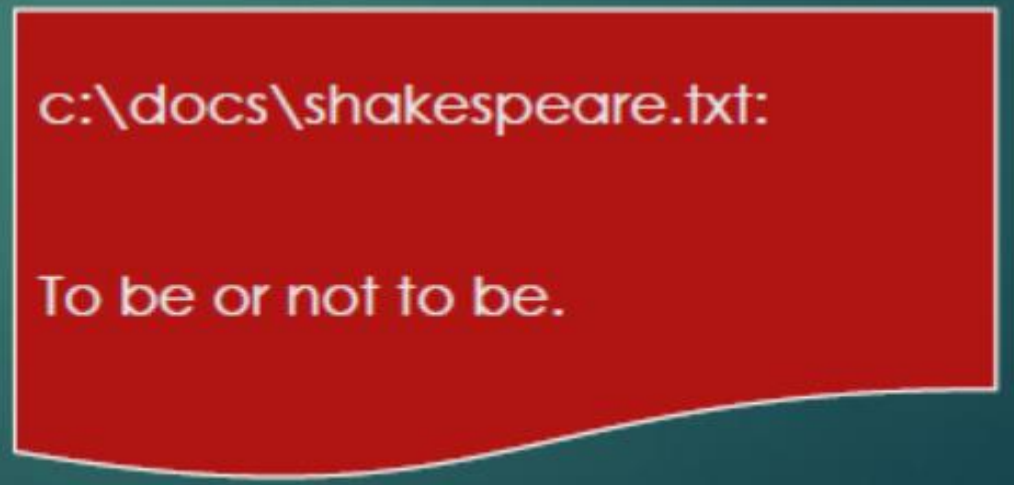


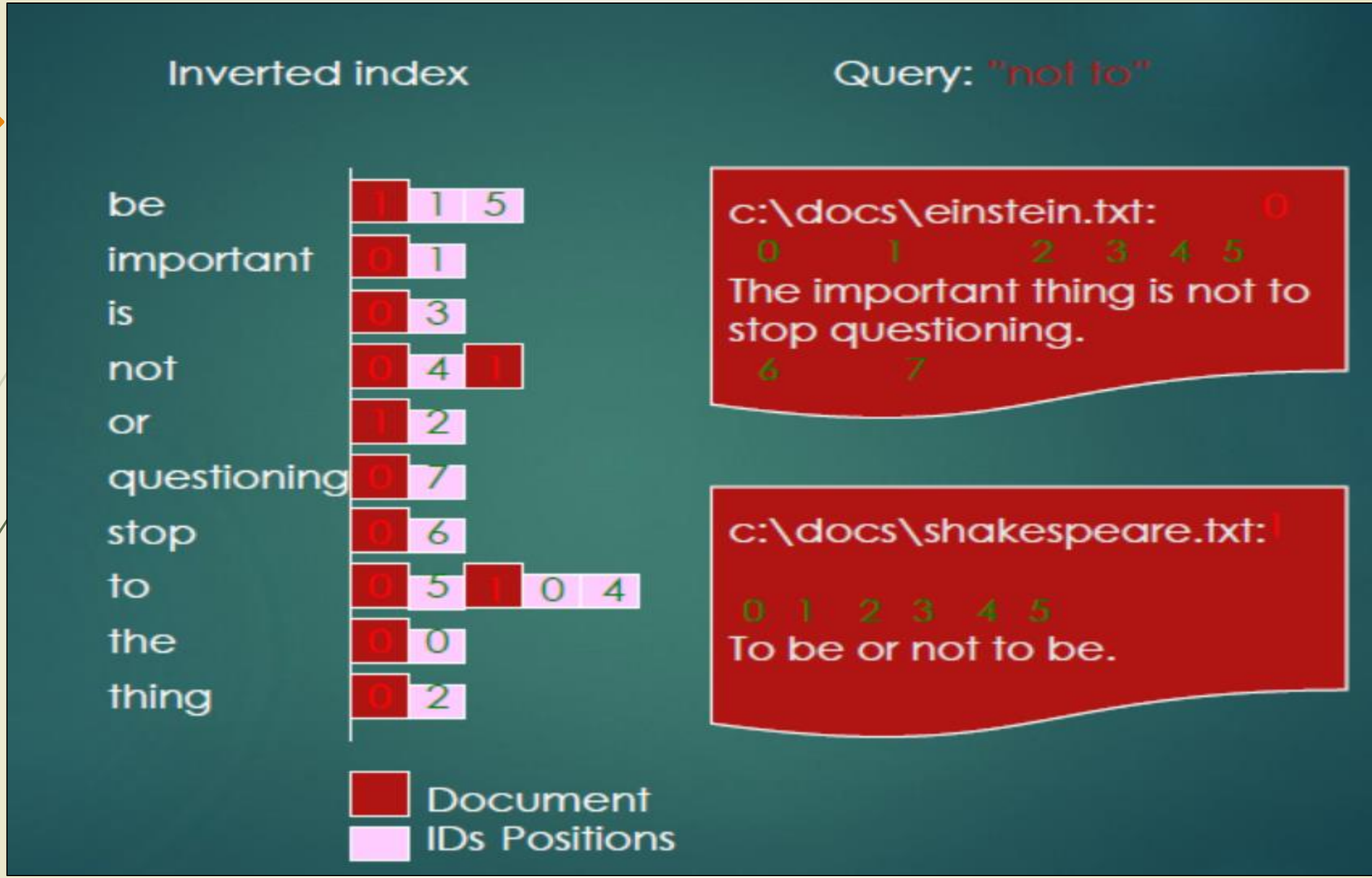
Query: not

String comparison slow!

Solution: Inverted index






Inverted index
Query: "not to"
be 1 1 5
important 0 1
is 0 3
not 0 4 1
or 1 2
questioning 0 7
stop 0 6
to 0 5 1 0 4
the 0 0
thing 0 2
Document
IDs Positions
c:\docs\einstein.txt: 0
0 1 2 3 4 5
The important thing is not to stop questioning.
6 7
c:\docs\shakespeare.txt: 1
0 1 2 3 4 5
To be or not to be.

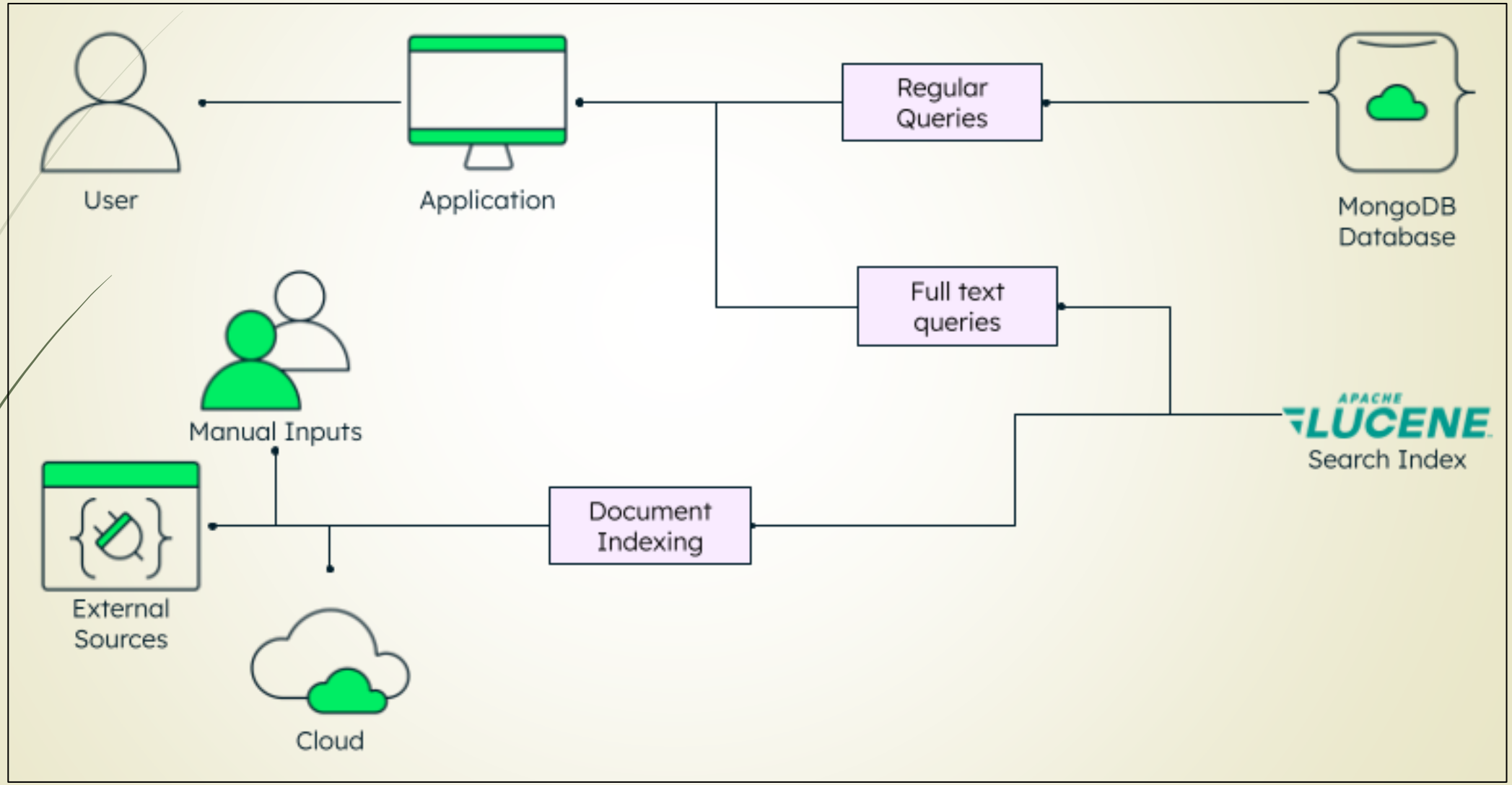
Full-text search

- A full-text search is a ***comprehensive search method*** that compares ***every word of the search*** request against ***every word within the document*** or database.
- Web ***search engines and document*** editing software ***make extensive use of the full-text search technique*** in functions for searching a text ***database stored on the Web*** or on the local ***drive of a computer***; it lets the user ***find a word or phrase*** anywhere within ***the database*** or document.
- Full-text search is the ***most common technique*** used in ***Web search engines and Web pages.***
- ***Each page is searched*** and ***indexed***, and if any ***matches are found***, they are ***displayed via the indexes***. Parts of original ***text are displayed against the user's query*** and then the full text.
- Full-text search ***reduces the hassle of searching for a word in huge amounts of metadat***a, such as the World Wide Web and commercial-scale databases.

Full-text search

- A common question from ***non-Full-Text users*** is, ***"If Full-Text search is about looking for words inside text, then XQuery already does that with the contains function. So what's missing?***" The contains function does not do a Full-Text search – it does a substring search.
- The ***main difference is that a Full-Text search will generally match only a complete word***, and not just part of a string. For example, a Full-Text search for ***"dent"*** will not match a piece of text that contains the word "students," but a substring search will.
- Also, when running a Full-Text search, there is generally an assumption that the ***match will be case-insensitive***,2 so that "dent" will match "DENT" as well as "dent" (and "Dent" and "dEnt" and "DEnt" and so on). ***With substring queries, matching is usually case-sensitive*** (depending on the collation used), so that the text being searched has to match the case of the search term.

Full-text search



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 is responsible to analyze a document and get the tokens/words from the text which is to be indexed. Without analysis done, IndexWriter cannot create index.
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.

Primary Analyzers available 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, ...

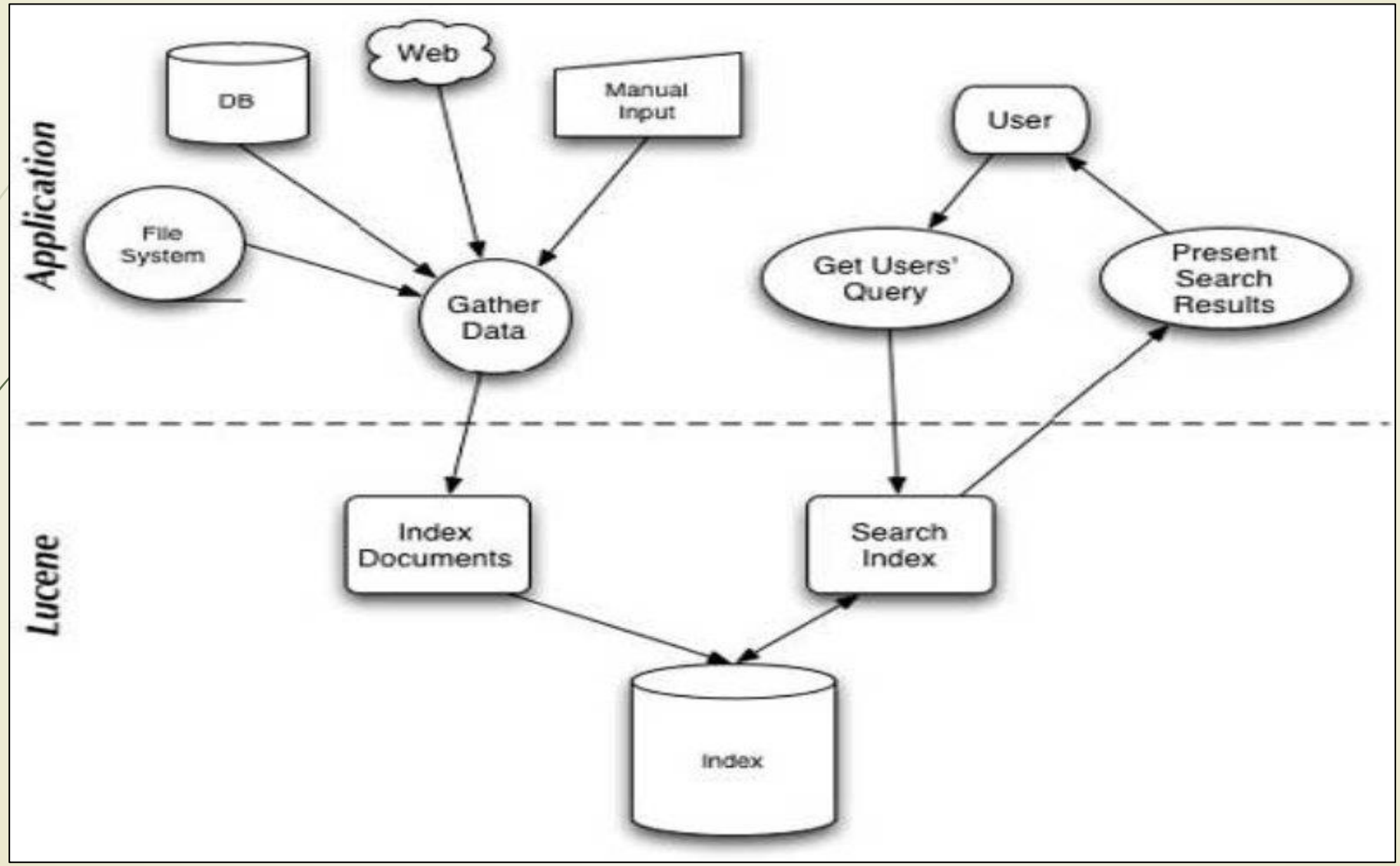
Analysis examples

Example Text	• "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]

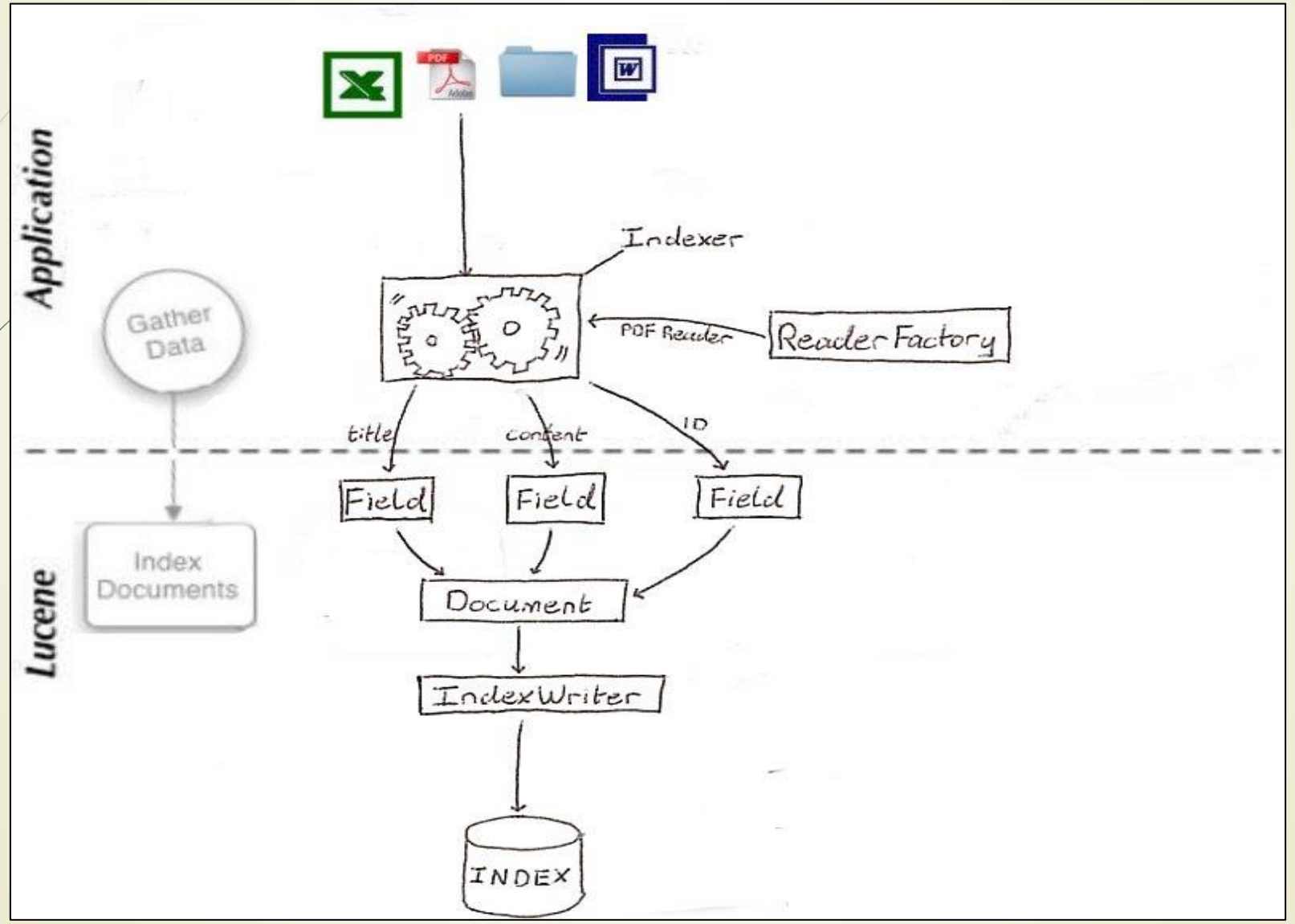
Core searching classes

- ***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.
- ***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 similar to 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.

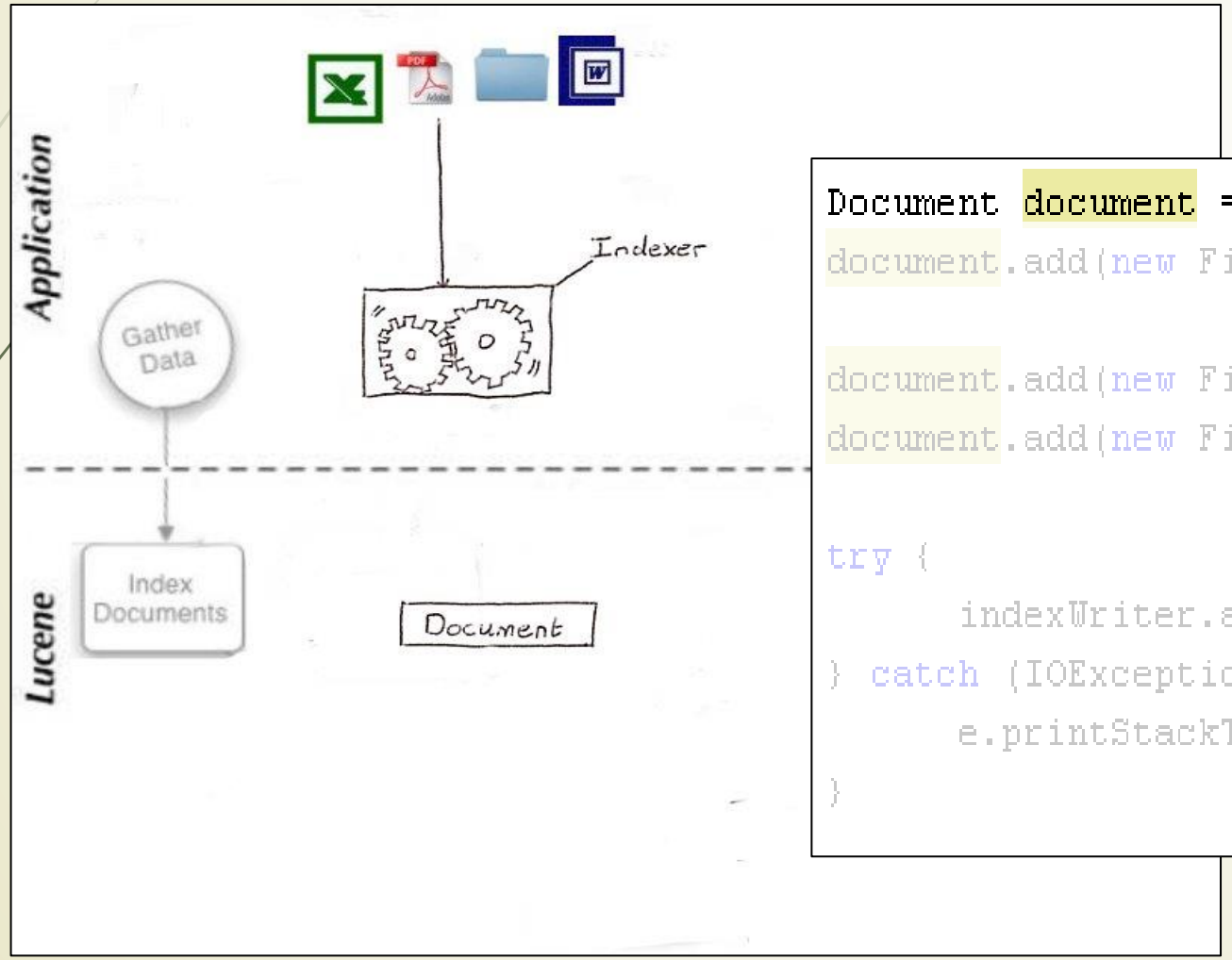
Lucene Implementation



Lucene Indexing

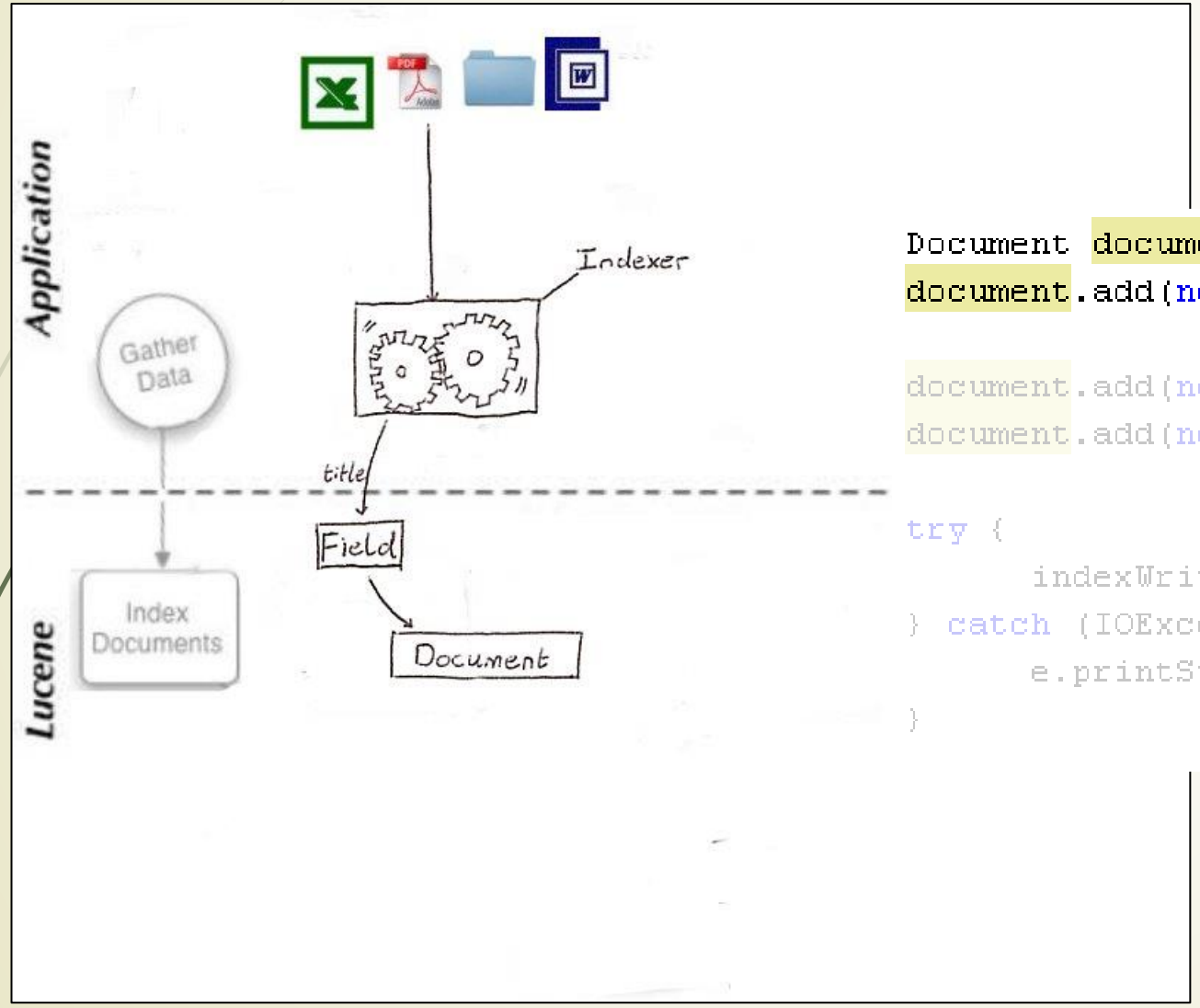


Lucene Indexing Step 1 of 5



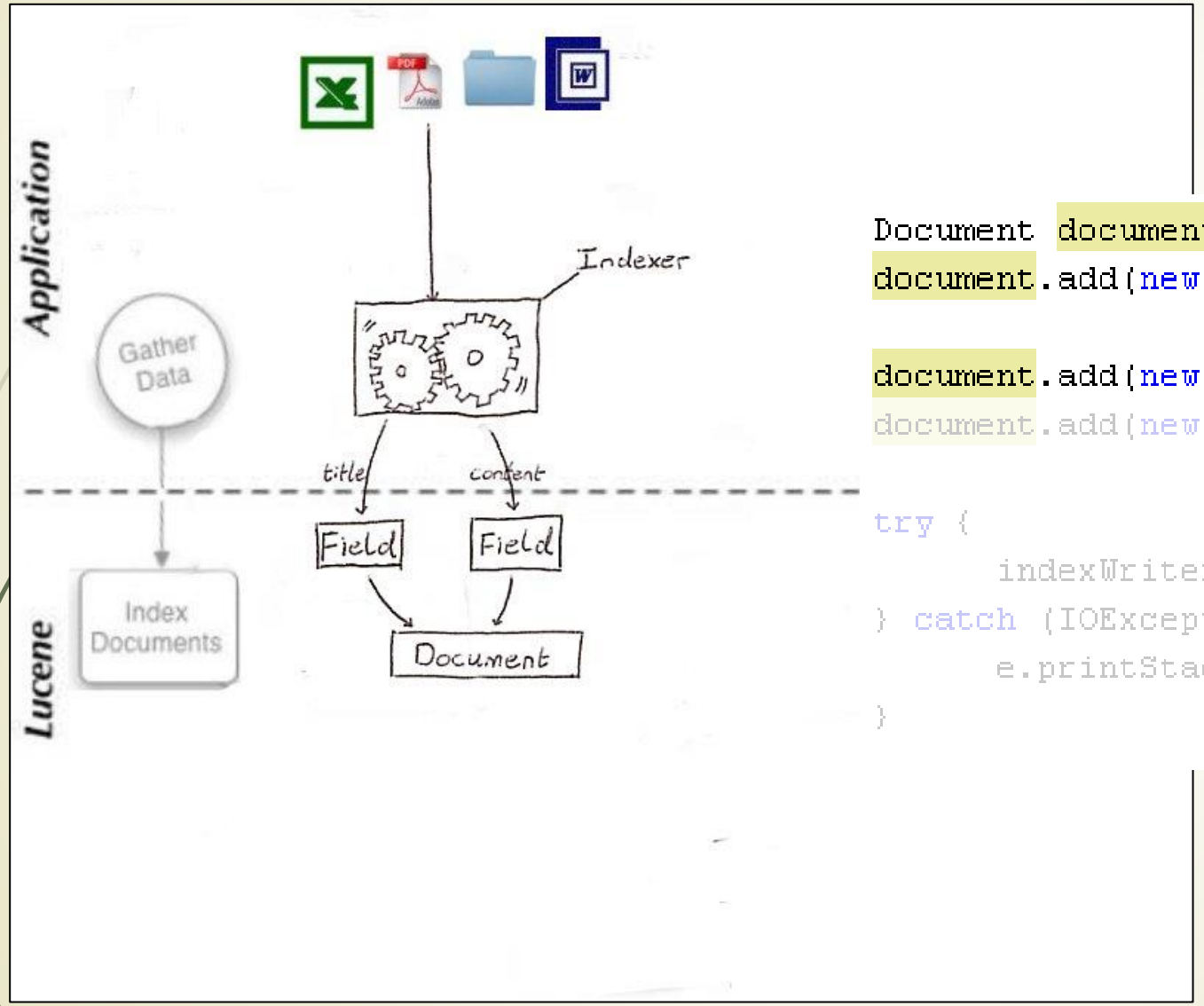

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                    Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                    Field.Index.NO));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

Lucene Indexing Step 2 of 5



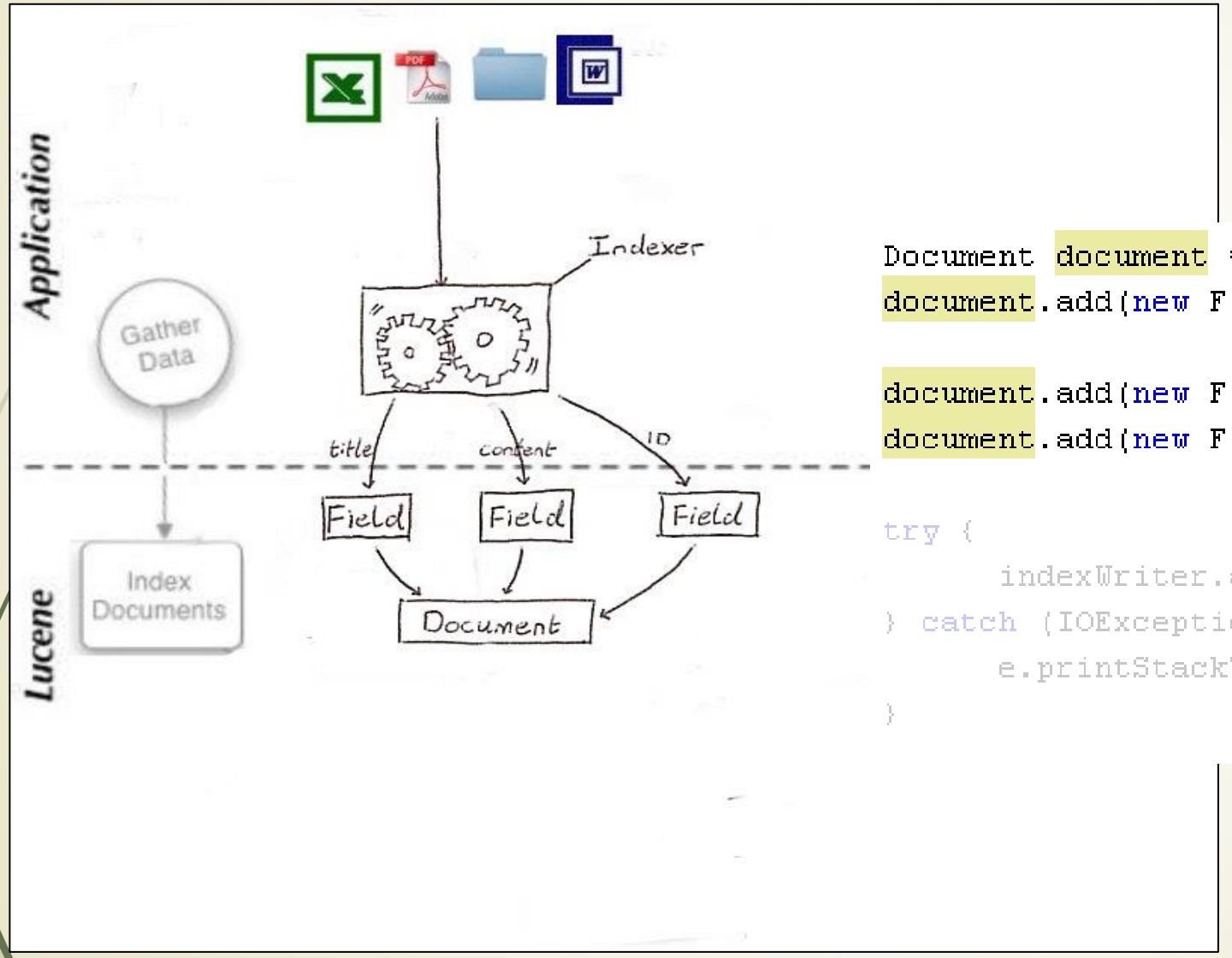

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                        Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                        Field.Index.NO));

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

Lucene Indexing Step 3 of 5



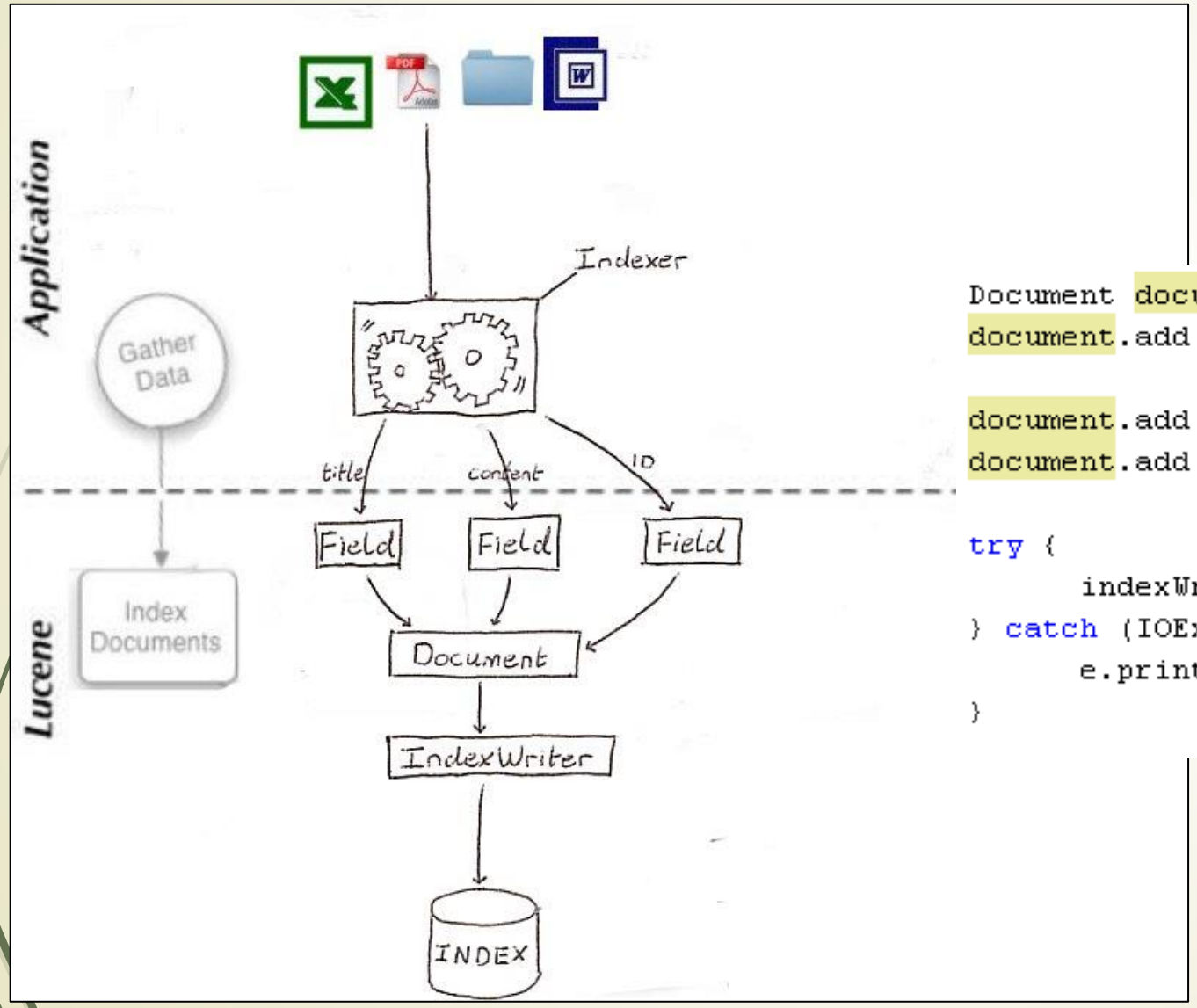

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                       Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                 Field.Index.NO));

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

Lucene Indexing Step 4 of 5



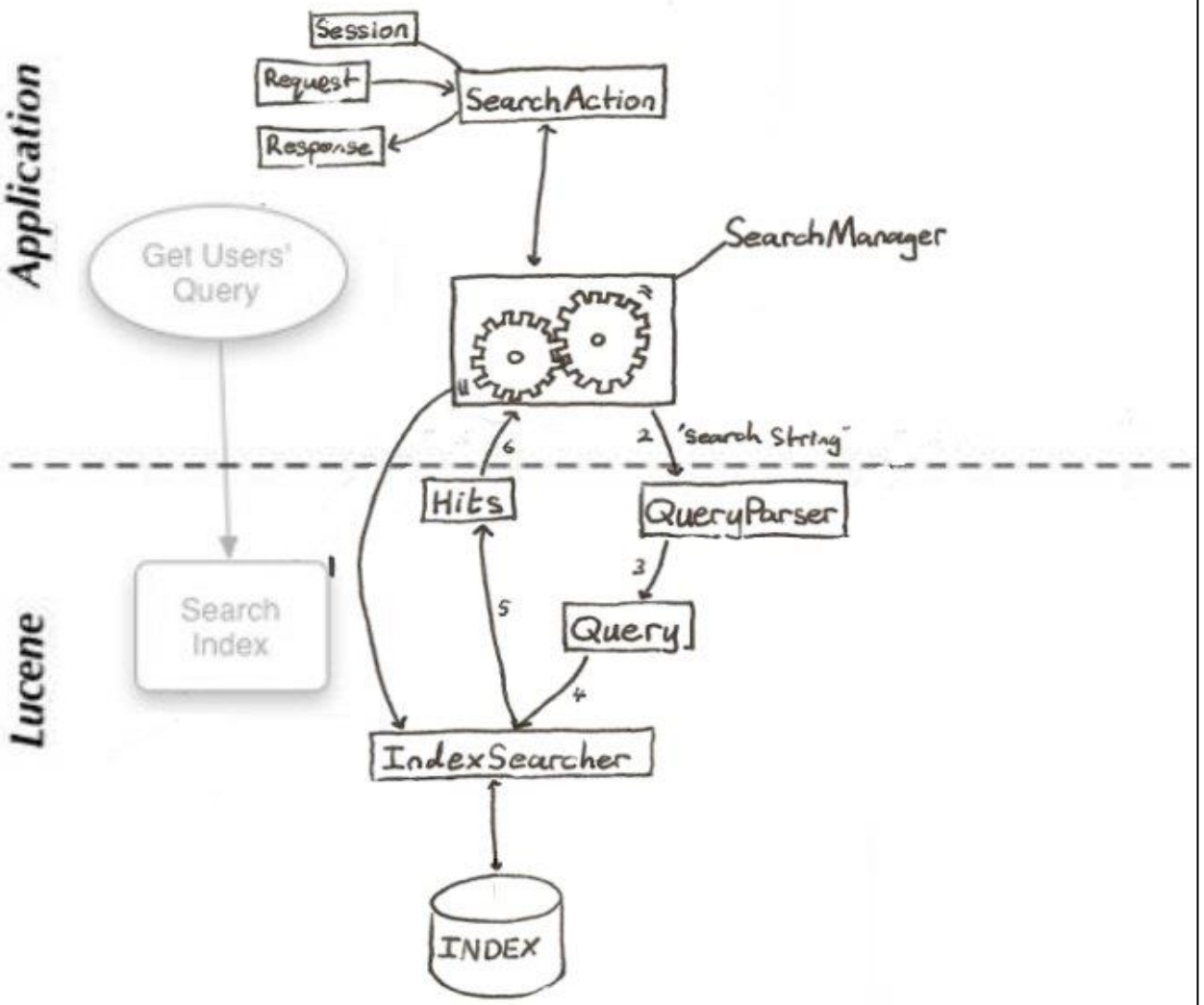

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                        Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                        Field.Index.NO));

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

Lucene Indexing Step 5 of 5



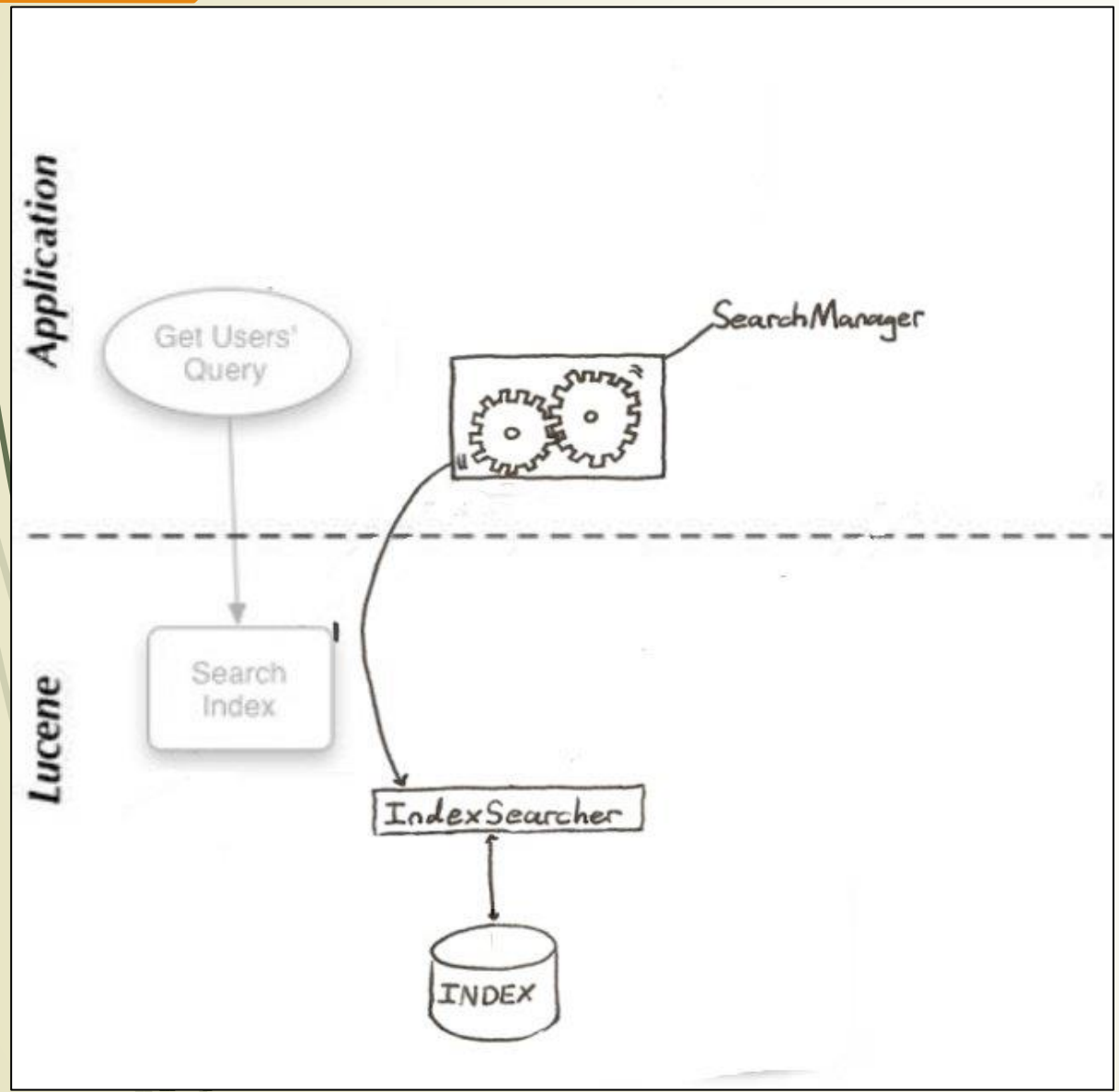

```
Document document = new Document();
document.add(new Field("title", title, Field.Store.YES,
                                        Field.Index.TOKENIZED));
document.add(new Field("content", content));
document.add(new Field("id", id, Field.Store.YES,
                                        Field.Index.NO));

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

Searching



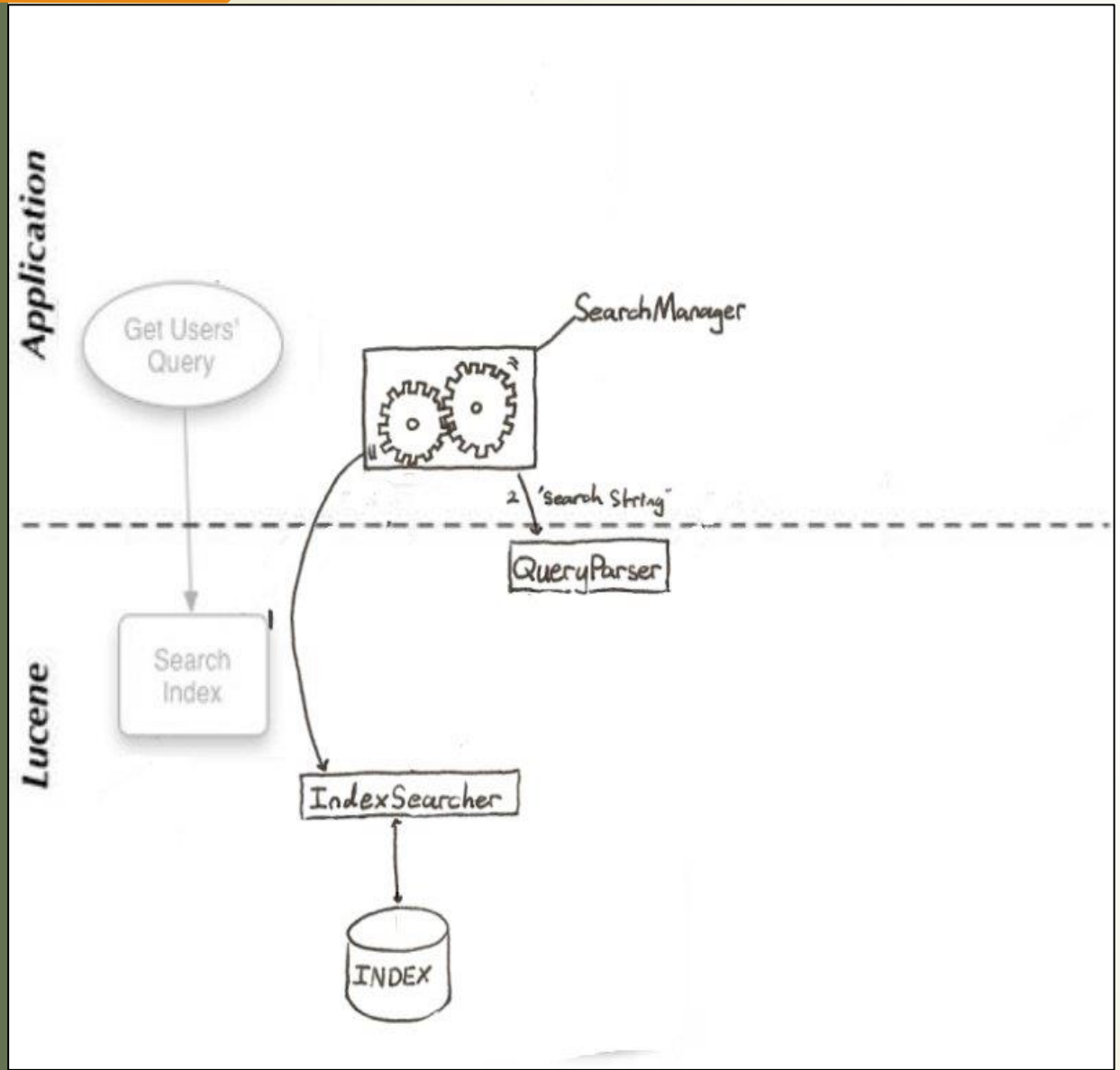

```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
QueryParser queryParser = new QueryParser("content",analyzer);
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (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();
    }
}
```

Searching: Step 1 of 6



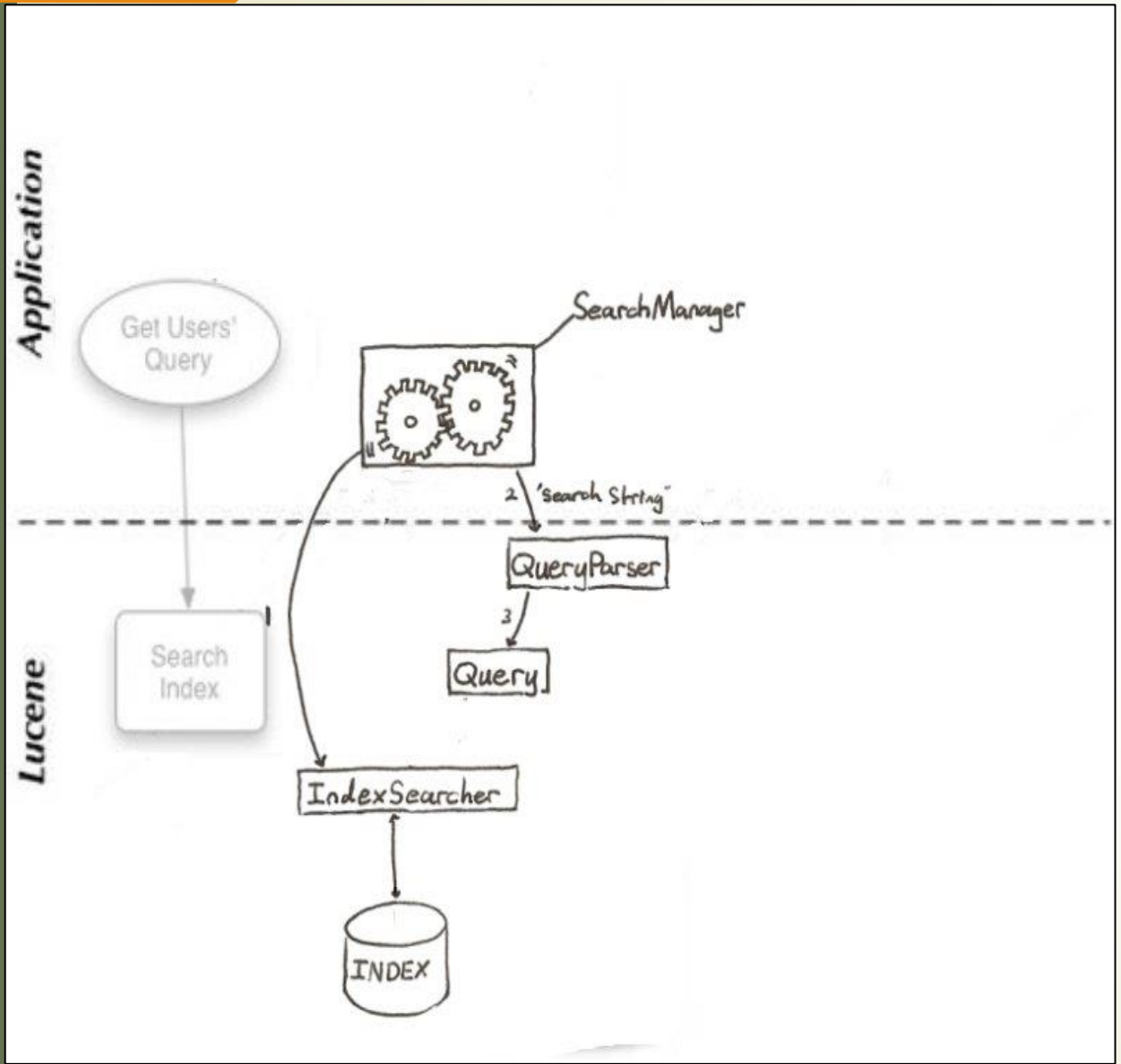

```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
QueryParser queryParser = new QueryParser("content",analyzer);
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (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();
    }
}
```

Searching: Step 2 of 6



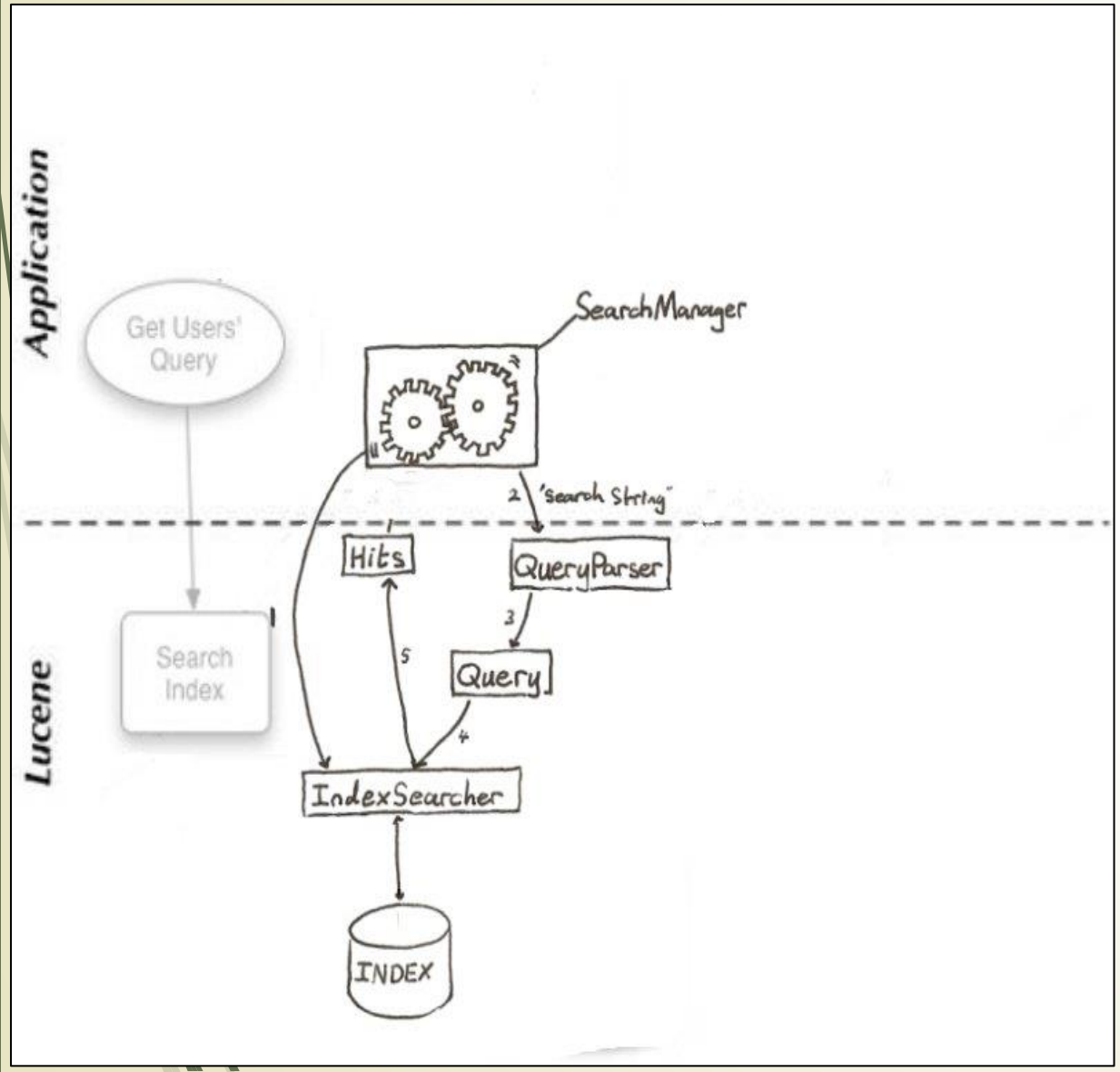

```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
QueryParser queryParser = new QueryParser("content",analyzer);
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (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();
    }
}
```

Searching: Step 3 of 6



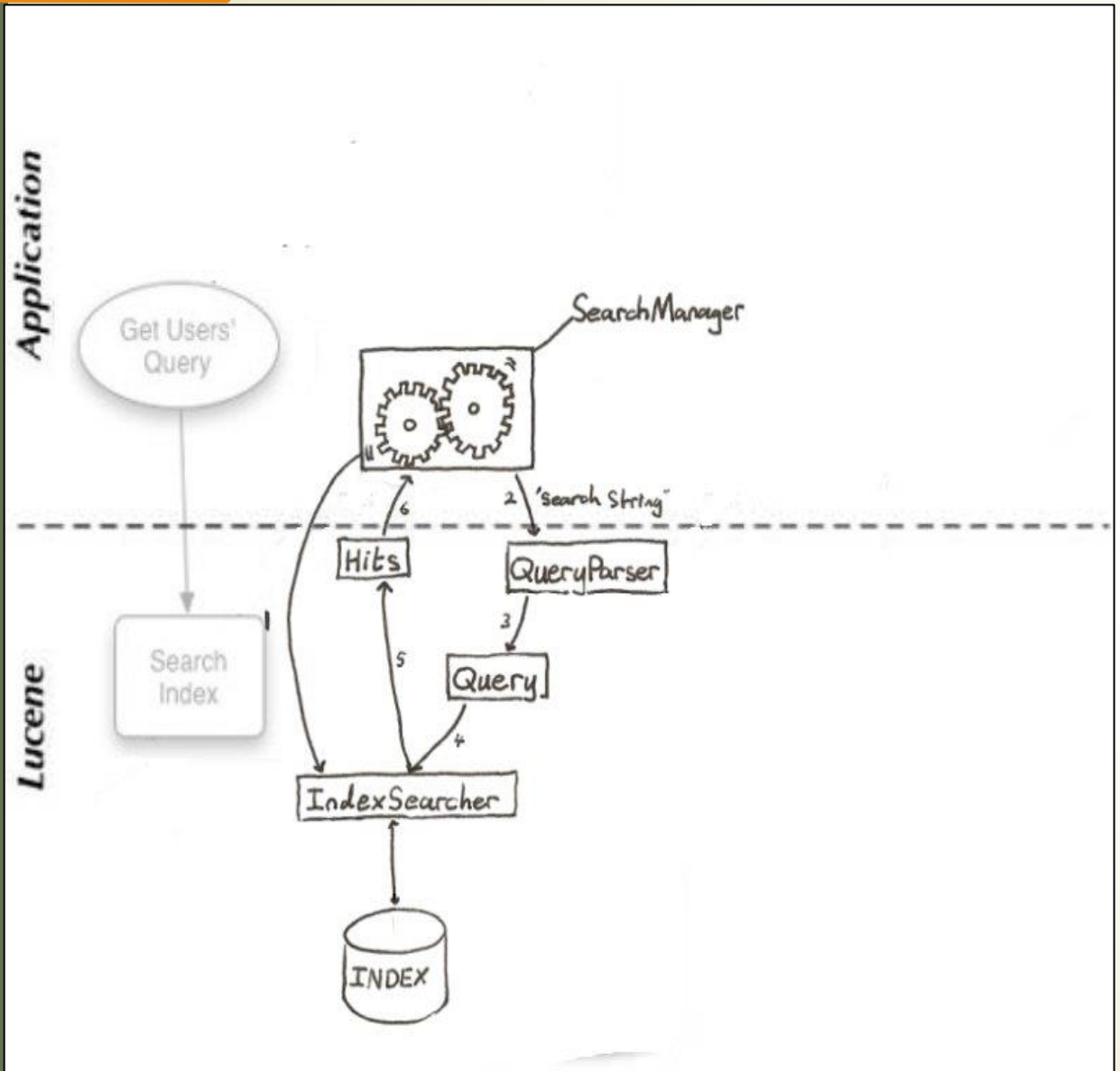

```
IndexSearcher indexSearcher = null;
try{
    indexSearcher = new IndexSearcher("/opt/lucene/index");
}catch(IOException ioe){
    ioe.printStackTrace();
}
QueryParser queryParser = new QueryParser("content",analyzer);
Query query = null;
try {
    query = queryParser.parse("Search string");
} catch (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();
    }
}
```

Searching: Step 4 and 5 of 6




```
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();
    }
}
```

Searching: Step 6 of 6




```
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();
    }
}
```

Elasticsearch

- Elasticsearch is a ***distributed, open-source*** search and analytics engine built on ***Apache Lucene*** and ***developed in Java***.
- It started as a ***scalable version of the Lucene*** open-source search framework then ***added the ability to horizontally scale*** Lucene indices.
- Elasticsearch ***allows you to store, search, and analyze huge volumes of data quickly*** and in ***near real-time*** and give back answers in milliseconds.
- It's able to ***achieve fast search responses because*** instead of searching the text directly, ***it searches an index***. It uses a structure based on documents instead of tables and schemas and ***comes with extensive REST APIs for storing and searching*** the data.
- At its core, you can think of ***Elasticsearch as a server*** that can process ***JSON requests and give you back JSON data***.

Elasticsearch

- ElasticSearch is ***able to achieve fast search*** responses because, instead of searching the text directly, ***it searches an index instead***. This is like retrieving pages in a book related to a keyword by scanning the ***index at the back of a book***, as opposed to searching every word of every page of the book.
- This type of index is called an ***inverted index***, because it inverts a page-centric data structure (page->words) to a keyword-centric data structure (word->pages).
- 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.

Elasticsearch

- RESTful API is an interface that two computer systems use to exchange information securely over the internet.
- ***Features***
  - Real time data,
  - Real time analytics,
  - Distributed, high availability, multi-tenancy, full text search,
  - Document oriented, conflict management, schema free,
  - RESTful API per-operation persistence, apache 2 open source license, build on top of apache Lucene.

Why Elasticsearch?

- Easy to deploy (minimum configuration)
- Scales vertically and horizontally
- Easy to use API
- Modules for most programming/scripting languages
- Actively developed with good online documentation
- It’s free.

Thank You

???