Introduction to Lucene

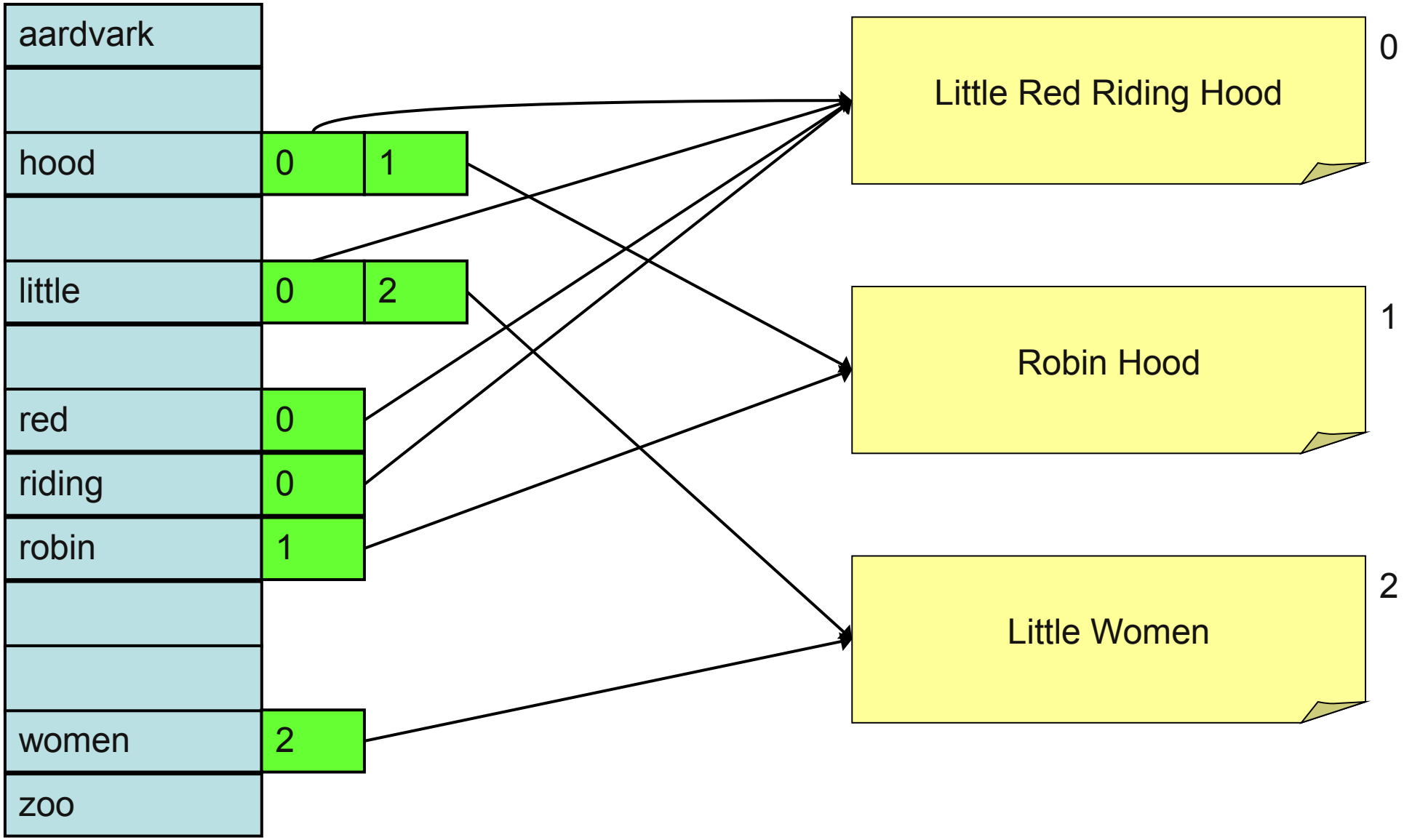
Overview

- What is Lucene?
- Vector Space Model
- Lucene tutorial
- Summary

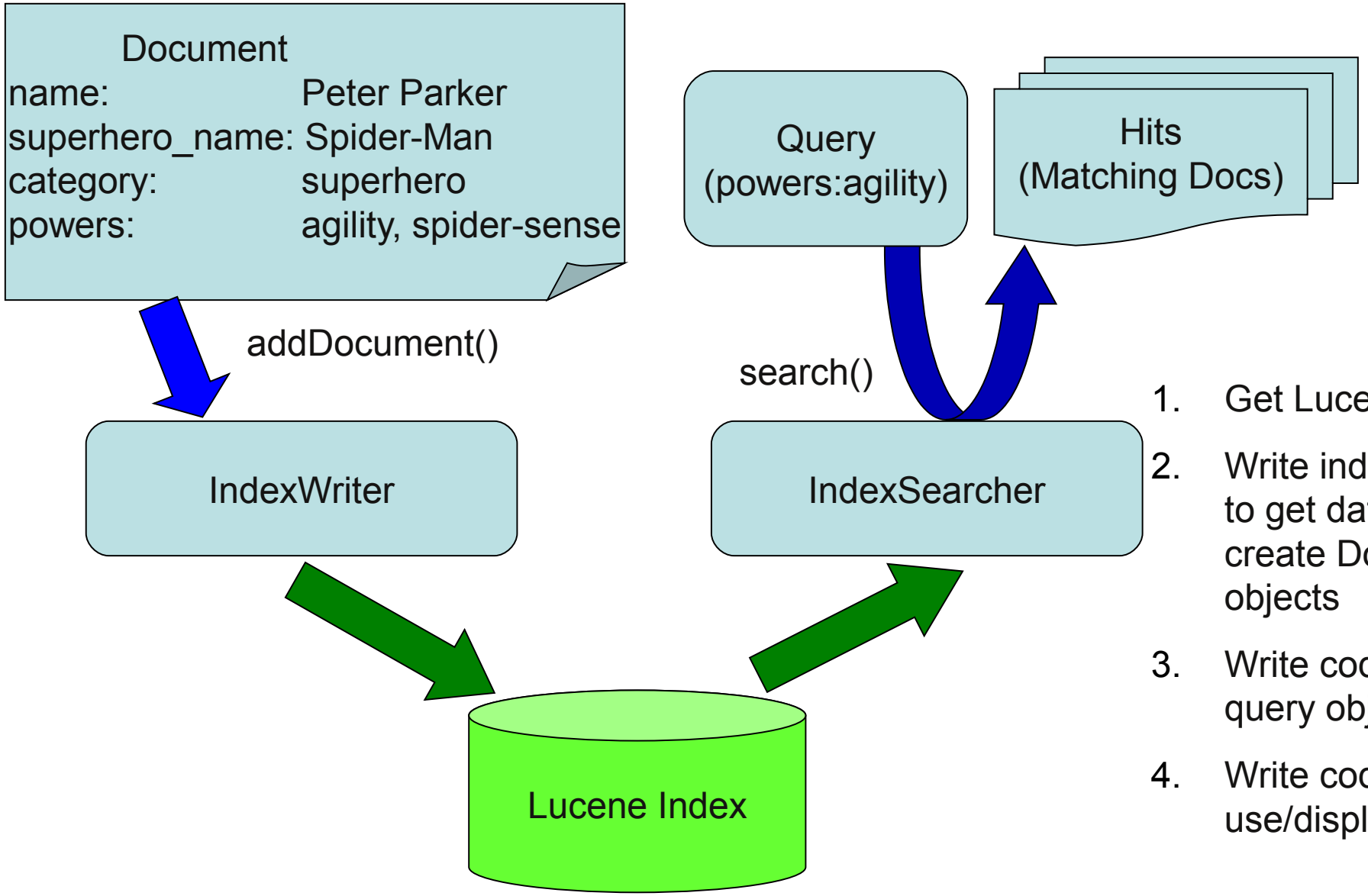
What is  ?

- Free-text indexing library
- Implements standard IR/search functionality
  - Query models, ranking, indexing
- Core API is implemented in Java
  - Bindings for C++/C, Ruby, Python, etc.

Inverted Index



Basic Application



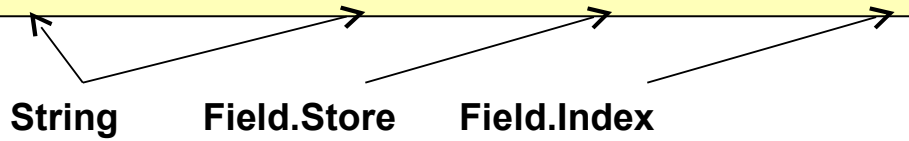
1. Get Lucene jar file
2. Write indexing code to get data and create Document objects
3. Write code to create query objects
4. Write code to use/display results

Lucene: Documents

- a Document is the basic unit for indexing and searching (**note**: it is different from the notion of document as file)
- each Document is a list of Field(s)
- each Field has a name and a text value
- **It is up to us to decide what to include in a Document**

JAVA Code

```
Document d = new Document();
d.add(new Field(nome_campo, valore, storeable, indexable));
```



Lucene: Field

- a field is the basic unit of Documents are made
  - remember: each Document is a list of Field(s)
- For each field, you need to specify
  - name
  - value
  - whether to store it
  - whether to index it

JAVA Code

```
Field f1 = new Field(“name”, “my_doc1”, Field.Store.YES, Field.Index.NO);
Field f2 = new Field(“term”, “Lucene”, Field.Store.YES, Field.Index.NOT_ANALYZED);
Field f3 = new Field(“term”, “Oggi ho assistito”, Field.Store.YES, Field.Index.ANALYZED);
```

Field Options

- Indexed
  - Necessary for searching or sorting
- Tokenized
  - Text analysis done before indexing
- Stored
  - You get these back on a search “hit”
- Compressed
- Binary
  - Currently for stored-only fields

Lucene: Directory

- At its core, a list of files
- RAMDirectory
  - in-memory volatile dir
  - useful for “on-the-fly” indexes
  - RAMDirectory dir= new RAMDirectory();
- FSDirectory
  - file-based, persistent dir
  - FSDirectory dir = FSDirectory.open(new File("tmp"));

Indexing Documents

```
IndexWriter writer =
        new IndexWriter(directory, analyzer, true);
Document doc = new Document();
doc.add(new Field(“super_name", “Sandman",
              Field.Store.YES, Field.Index.TOKENIZED));
doc.add(new Field(“name", “William Baker",
              Field.Store.YES, Field.Index.TOKENIZED));
doc.add(new Field(“name", “Flint Marko",
             Field.Store.YES, Field.Index.TOKENIZED));
// [...]
writer.addDocument(doc);
writer.close();
```

Searching an index

```
IndexSearcher searcher =
      new IndexSearcher(directory);
QueryParser parser =
      new QueryParser("defaultField", analyzer);
Query query = parser.parse(“powers:agility");
Hits hits = searcher.search(query);

System.out.println(“matches:" + hits.length());
Document doc = hits.doc(0); // look at first match
System.out.println(“name=" + doc.get(“name"));

searcher.close();
```

Searching an index: example 2

1. Open index

JAVA Code

```
IndexSearcher is = new IndexSearcher(indexDir);
```

2. Create the query

JAVA Code

```
QueryParser parser = new QueryParser(Version.LUCENE_30, “term”, analizzatore);
Query q = parser.parse(“lezione”);
```

3. Search

JAVA Code

```
ScoreDoc[] docs = searcher.search(q, <numHits>).scoreDocs;
```

4. Read the result

JAVA Code

```
for (ScoreDoc doc:docs)
{
        Document d = is.doc(doc.doc);      // ottiene il documento
        float score =doc.score; // punteggio del documento
        // ...
}
```

Query Construction: QueryParser

- **QueryParser**
  - does text analysis (more later) and constructs appropriate queries
  - not all query types supported

JAVA Code

```
QueryParser parser  =  new QueryParser(Version.LUCENE_30, “powers”, analizzatore);
Query q = parser.parse(“agility AND spider-sense”);
```

- **TermQuery**
  - explicit, no escaping necessary
  - no text analysis

JAVA Code

```
BooleanQuery q = new BooleanQuery();
q.add(new TermQuery(new Term(“powers”, “agility”)), BooleanClause.Occur.MUST);
q.add(new TermQuery(new Term(“powers”, “spider-sense”)), BooleanClause.Occur.MUST);
```

QueryParser: examples

Query expression	Matches documents that...
`java`	Contain the term *java* in the default field
`java junit` `java or junit`	Contain the term *java* or *junit*, or both, in the default field[a]
`+java +junit` `java AND junit`	Contain both *java* and *junit* in the default field
`title:ant`	Contain the term *ant* in the `title` field

`title:extreme`
`-subject:sports`
`title:extreme`
`AND NOT subject:sports`
	Have *extreme* in the `title` field and don't have *sports* in the `subject` field

`(agile OR extreme) AND` `methodology`	Contain *methodology* and must also contain *agile* and/or *extreme*, all in the default field
`title:"junit in action"`	Contain the exact phrase "*junit in action*" in the `title` field
`title:"junit action"~5`	Contain the terms *junit* and *action* within five positions of one another
`java*`	Contain terms that begin with *java*, like *javaspaces*, *javaserver*, and *java.net*
`java~`	Contain terms that are close *to* the word *java*, such as *lava*
`lastmodified:` `[1/1/04 TO 12/31/04]`	Have `lastmodified` field values between the dates January 1, 2004 and December 31, 2004

Scoring documents

- Scoring is performed using the so-called Vector Space Model (VSM)
  - A model to represent text as vectors of terms
  - Represents documents as BOWs
  - Scoring is typically performed using *tf*idf*
- **Let's review the VSM!**

Boolean search

- Based on Boolean queries
  - documents either match or don’t
  - good for expert users with precise understanding of their needs and the collection (e.g. library search)
  - not good for the majority of users
    - most users incapable of writing Boolean queries

Boolean search: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
  - "*standard user dlink 650*" → 200,000 hits
  - "*standard user dlink 650 no card found*": 0 hits
- It takes skill to come up with a query that produces a manageable number of hits.
- With a ranked list of documents, it does not matter how large the retrieved set is.

Scoring as the basis of ranked retrieval

- We wish to return *in order* the documents most likely to be useful to the searcher

- How can we rank-order the documents in the collection with respect to a query?

- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query “match”.

Query-document matching scores

- We need a way of assigning a score to a query/document pair

- Let's start with a one-term query
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)

Take 1: Jaccard coefficient

- a commonly used measure of overlap of two sets *A* and *B*

$$\text{jaccard}(A,B) = |A \cap B| \,/\, |A \cup B|$$

$$\text{jaccard}(A,A) = 1$$

$$\text{jaccard}(A,B) = 0 \text{ if } A \cap B = 0$$

- *A* and *B* don't have to be the same size.
- always assigns a number between *0* and *1*.

Jaccard coefficient: example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?

- Query: *ides of march*
- Document 1: *caesar died in march*
- Document 2: *the long march*

Issues with Jaccard for scoring

- It doesn't consider term frequency (how many times a term occurs in a document)
- It doesn't consider document/collection frequency (rare terms in a collection are more informative than frequent terms)

Binary term-document incidence matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

Term-document count matrices

- Consider the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^v$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Bag of words model

- Vector representation doesn't consider the ordering of words in a document
  - *John is quicker than Mary* and *Mary is quicker than John* have the same vectors

- This is called the bag of words model.

Term frequency *tf*

- The *term frequency* $tf_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
- *Raw* term frequency is *not* what we want:
  - A document with 10 occurrences of the term may be more relevant than a document with one occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms *t* in both *q* and *d*:
- score $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$

Document frequency

- Rare terms are more informative than frequent terms
  - Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
  - A document containing this term is very likely to be relevant to the query *arachnocentric*
  - → We want a higher weight for rare terms like *arachnocentric*.

Document frequency, continued

- Consider a query term that is frequent in the collection (e.g., *high, increase, line*)
  - A document containing such a term is more likely to be relevant than a document that doesn't, *but it's not a sure indicator of relevance.*
  - → For frequent terms, we want positive weights for words like *high, increase, and line*, but lower weights than for rare terms.
- We will use document frequency (df) to capture this in the score.
- df ($\leq N$) is the number of documents that contain the term

idf weight

- df_t is the document frequency of *t*: the number of documents that contain *t*
  - df is a measure of the informativeness of *t*
- We define the idf (inverse document frequency) of *t* by

$$\text{idf}_t = \log_{10} N/\text{df}_t$$

  - We use log N/df_t instead of N/df_t to "dampen" the effect of idf.

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log N / \mathrm{df}_t$$

  - Best known weighting scheme in information retrieval
  - Increases with the number of occurrences within a document
  - Increases with the rarity of the term in the collection

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

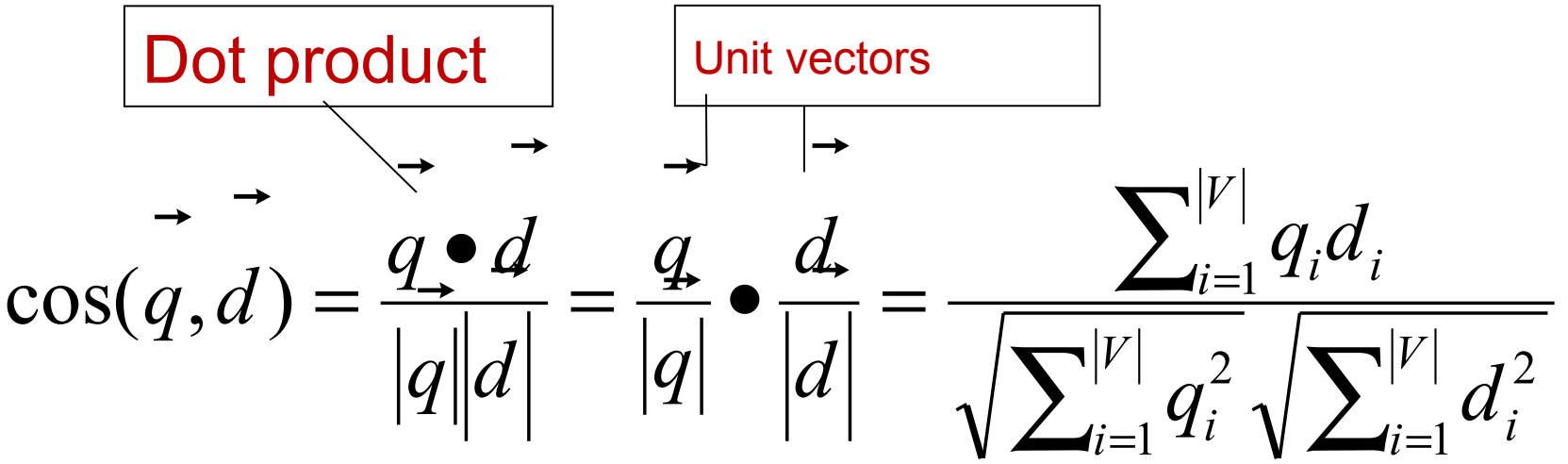
Each document is now represented by a real-valued vector of tf-idf weights $\in R^{|V|}$

Documents as vectors

- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space

- Very high-dimensional: hundreds of millions of dimensions when you apply this to a web search engine
- This is a very sparse vector - most entries are zero.

Computing cosine scores



q_i is the tf-idf weight of term i in the query

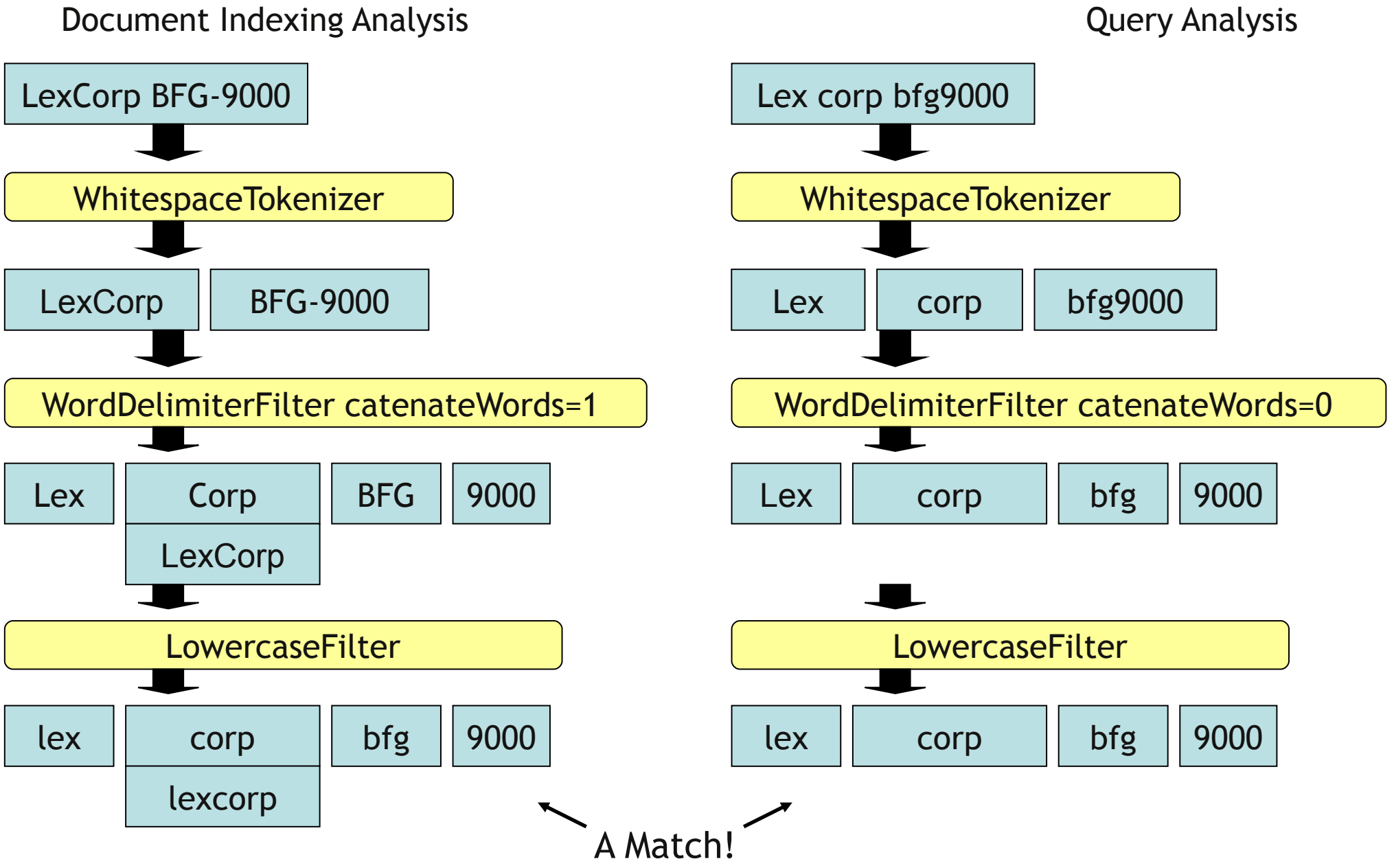
d_i is the tf-idf weight of term i in the document

cos(q,d) is the cosine similarity of q and d … or, equivalently, the cosine of the angle between q and d.

Computing cosine scores

```
CosineScore(q)
 1  float Scores[N] = 0
 2  float Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5     for each pair(d, tf_{t,d}) in postings list
 6     do Scores[d] += w_{t,d} × w_{t,q}
 7  Read the array Length
 8  for each d
 9  do Scores[d] = Scores[d]/Length[d]
10  return Top K components of Scores[]
```

Analysis & Search Relevancy



Performing text analisys

- Analyser
  - instantiate a tokenizer
  - applies a set of filters to the input

JAVA Code

```
Analyzer analyser = new StandardAnalyzer();
IndexWriter writer = new IndexWriter(indexDir, analyser,
                                     bCrea, MaxFieldLength.UNLIMITED);
```

Tokenizers

Tokenizers break field text into tokens

- input: “full-text lucene.apache.org”
  - StandardTokenizer
    - => “full” “text” “lucene.apache.org”
  - WhitespaceTokenizer
    - => “full-text” “lucene.apache.org”
  - LetterTokenizer
    - => “full” “text” “lucene” “apache” “org”

Some analysers

- WhitespaceAnalyzer
  - splits on whitespace
- SimpleAnalyzer
  - splits on whitespace and special characters; applies lowercase
- StopAnalyzer
  - SimpleAnalyzer + stopword removal (the, an, a, ecc.)
- StandardAnalyzer
  - the most complete (Whitespace+Stop+misc)
- SnowballAnalyzer
  - performs also stemming

Analysers: examples

- The quick brown fox jumped over the lazy dogs

```
WhitespaceAnalyzer:
  [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

SimpleAnalyzer:
  [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

StopAnalyzer:
  [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

StandardAnalyzer:
  [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]
```

- 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]
```

Customized analysers

- … or how to create your own analyser:

JAVA Code

```
class MyAnalyzer extends Analyzer
{
    private Set stopWords = StopFilter.makeStopSet(StopAnalyzer.ENGLISH_STOP_WORDS);

    public TokenStream tokenStream(String fieldName, Reader reader)
    {
        TokenStream ts = new StandardTokenizer(reader);
        ts = new StandardFilter(ts);
        ts = new LowerCaseFilter(ts);
        ts = new StopFilter(ts, stopWords);
        return ts;
    }
}
```

Luke: a graphical user interface

- In a nutshell, a GUI for Lucene indexes
  - Search for ID or term
  - Visualizes documents
  - Visualizes results
  - Index optimization
  - etc.

Other useful stuff

- Nutch
  - A search engine built on top of Lucene
- Solr
  - A high-performance search server built on top of Lucene
- Mahout
  - libraries for large (i.e. Web-) scale learning and processing

Bringing it all together...

- Let's fire Eclipse …
- … and write some Java code!