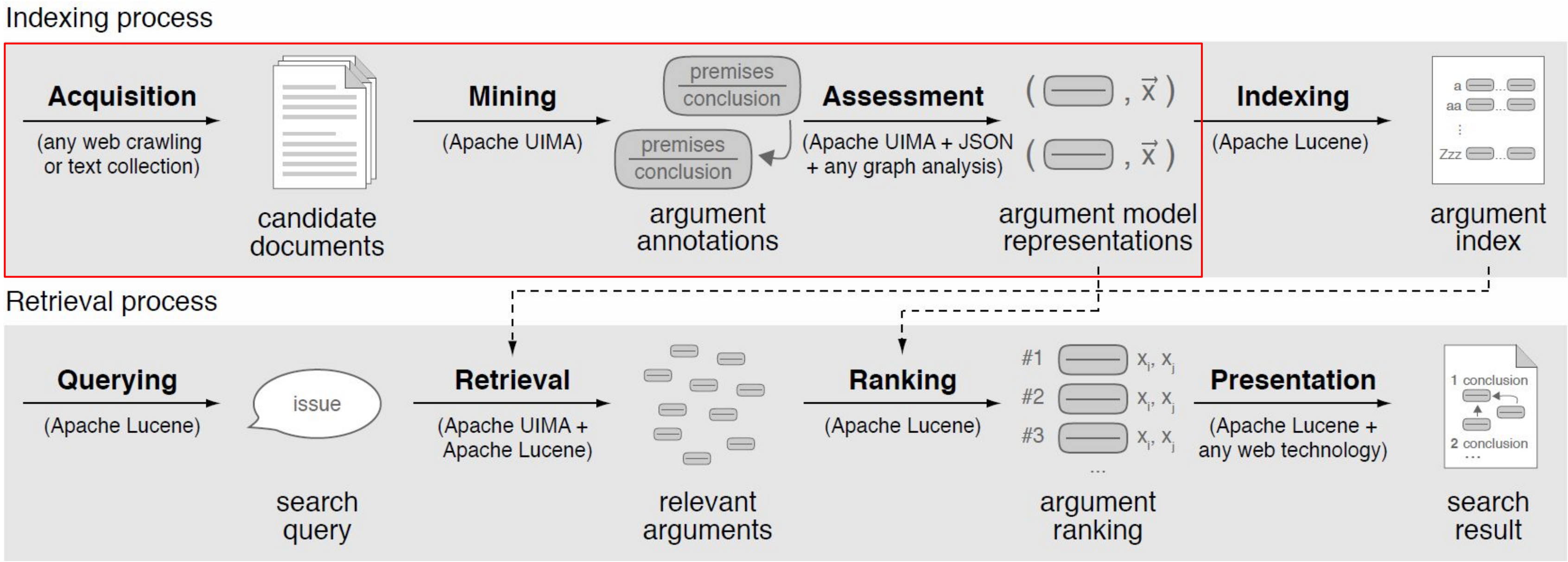
Args.me Architecture

Args.me Architecture

Args.me search engine structure



Indexing process

Acquisition

(any web crawling or text collection)

candidate documents

Mining

(Apache UIMA)

premises
conclusion

premises
conclusion

argument annotations

Assessment

(Apache UIMA + JSON + any graph analysis)

(, $\vec{x}$)

(, $\vec{x}$)

argument model representations

Indexing

(Apache Lucene)

a ...
aa ...
Zzz ...

argument index

Retrieval process

Querying

(Apache Lucene)

issue

search query

Retrieval

(Apache UIMA + Apache Lucene)

relevant arguments

Ranking

(Apache Lucene)

#1 x_i, x_j
#2 x_i, x_j
#3 x_i, x_j
...

argument ranking

Presentation

(Apache Lucene + any web technology)

1 conclusion
2 conclusion
...

search result

Args.me Architecture / Indexing

de.webis.args.framework.indexing.IIndexGenerator.java

public void initializeIndexer(Properties config)
public void createIndex()

Interface

de.webis.args.framework.ranking.SampleLuceneIndexer.java

Implements IIndexGenerator.java

public void createIndex()
Checks whether a new index is being created or an existing one being updated,
Creates a new IndexWriter (lucene class) and uses it to index documents from the docDir path

Args.me Architecture / Query Parser

de.webis.args.framework.presentation.GenericQueryGenerator.java

Can be used to prepare queries for argument-retrieval

public GenericQueryGenerato (String configPath)

➔ *Load properties for querying*

de.webis.args.framework.ranking.querying.java

Can be used to create the interface for presentation of Query parser

public interface IQueryCreator

➔ *public void initializeQuery(Properties config);*

➔ *public List<PreparedQuery> createQuery(String jsonSrcPath);*

Args.me Architecture / Query Parser

de.webis.args.framework.presentation.PreparedQuery.java

result class after executing the query step

public class PreparedQuery

- ➔ *Constructor for Query string*
- ➔ *Getter and Setter for query string*

Args.me Architecture / Retrieval

de.webis.args.framework.retrieval.GenericArgumentsGenerator.java

public List <Set<Argument>> retrieveArguments(List<...>prepQueries)

Iterates over a list of queries and returns all arguments

- ➔ for (PreparedQuery query : prepQueries)
  retrieveArguments(query)

- ➔ Fully configurable via properties file

Args.me Architecture / Ranking

de.webis.args.framework.ranking.SampleLuceneRanking.java

public List <Argument> computeScoresAndSort(retrievedArguments, preparedQuery)

Iterates over all retrieved Arguments and sets a score for every single one (using Lucene):

- ➔ *argument.setScore(argument.luceneScore())*
- ➔ Argument Object contains: URL, Conclusion, Premise, etc.

de.webis.args.framework.ranking.GenericRankingGenerator.java

Can be used to rank the retrieved arguments, configurable via properties file

public List<Argument> generateRanking(retrievedArguments)

- ➔ *mRankingType.computeScoresAndSort(retrievedArguments, preparedQueries)*
- ➔ mRankingType configured by (custom) config file

Args.me Architecture / Presentation

de.webis.args.framework.presentation.GenericPresiGenerator.java

Can be used to create the presentation of the retrieved arguments for the web interface

public void generatePresi(rankedArguments, preparedQueries)

- ➔ Creates presentation components for earch Argument and combines them
    - → *contentSnippet, titleSnippet, polishedPremise, urlSnippet, finalExplanation*
- ➔ Fully configurable via properties file

Apache Lucene

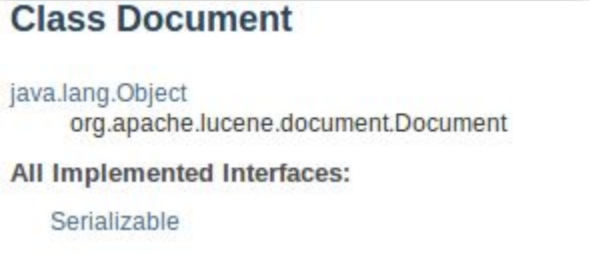
- Indexing & retrieving
- Query Syntax
- Scoring & ranking

Indexing & retrieving in Lucene

indexing documents:

- ➢ Indices in Lucene are based around four major components:
  - ○ The index, which contains a sequence of documents
  - ○ The document, which is a sequence of fields
  - ○ The field, which is a named sequence of terms
  - ○ The term, which is a sequence of bytes

- ➢ Lucene’s index belongs to the inverted index family

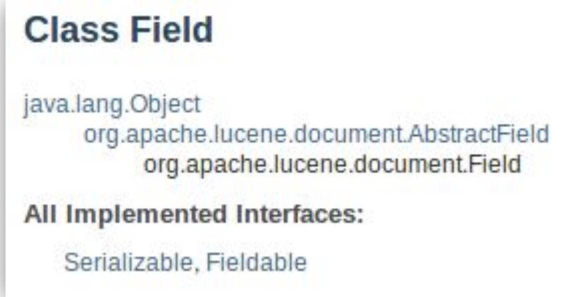


Class Document

java.lang.Object
    org.apache.lucene.document.Document

All Implemented Interfaces:

Serializable



Class Field

java.lang.Object
    org.apache.lucene.document.AbstractField
        org.apache.lucene.document.Field

All Implemented Interfaces:

Serializable, Fieldable

Indexing & retrieving in Lucene

- An index may be composed of multiple sub-indices (segments)

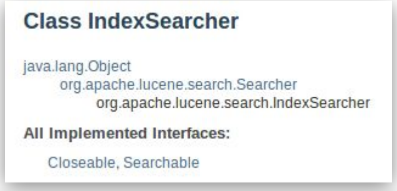
- Fields may be stored (non inverted
  - May be indexed (inverted)
  - May be stored (non inverted)
  - Tokenized (separated into terms to be indexed)
- Segments
  - Independent from one another
  - Documents are numbered by segment

Indexing & retrieving in Lucene

Index/segment structure:

- Segment info (metadata)
- Field names
- Stored field values
- Term dictionary
- Term frequency data
- Term proximity data
- Normalization factors
- Term vectors
- Per-document values
- Live documents
- Point values

Indexing & retrieving in Lucene



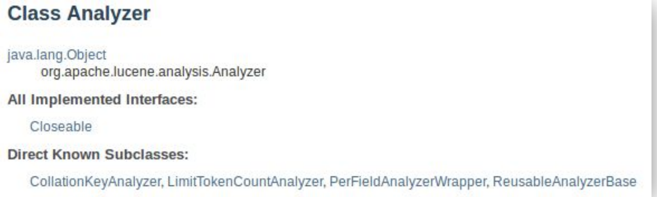
Class IndexSearcher

java.lang.Object
    org.apache.lucene.search.Searcher
        org.apache.lucene.search.IndexSearcher

All Implemented Interfaces:

Closeable, Searchable

Retrieving documents:



Class Analyzer

java.lang.Object
    org.apache.lucene.analysis.Analyzer

All Implemented Interfaces:

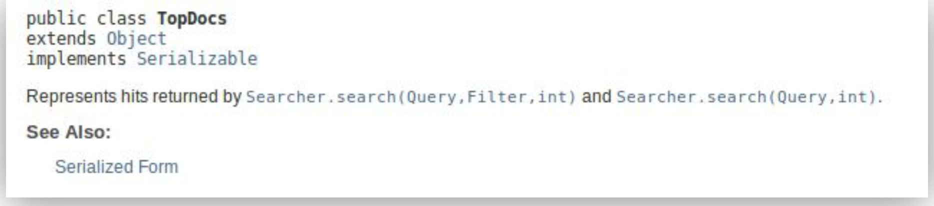
Closeable

Direct Known Subclasses:

CollationKeyAnalyzer, LimitTokenCountAnalyzer, PerFieldAnalyzerWrapper, ReusableAnalyzerBase

- ➢ The search function is made up of multiple components
  - ○ The index searcher
  - ○ The analyzer
  - ○ The query parser

- ➢ The search function returns the documents according to the highest score



```
public class TopDocs
extends Object
implements Serializable
```

Represents hits returned by `Searcher.search(Query,Filter,int)` and `Searcher.search(Query,int)`.

See Also:

Serialized Form

Apache Lucene

- Indexing & retrieving
- Query Syntax
- Scoring & ranking

Query Parser

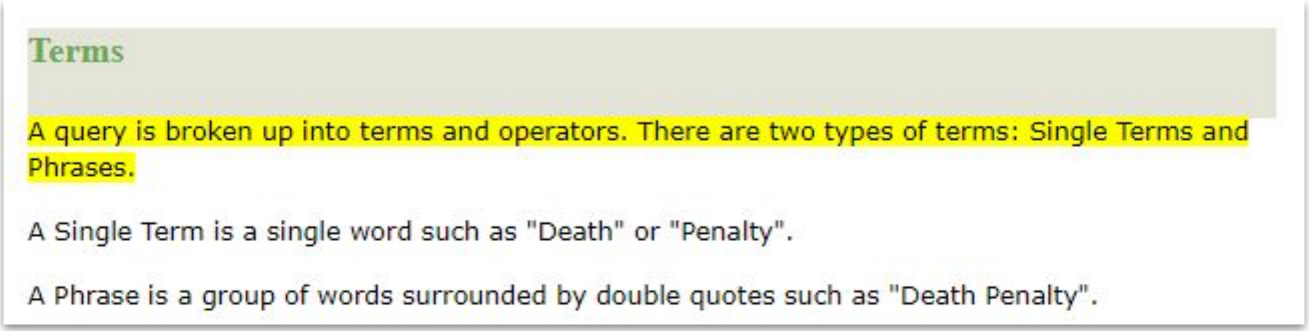
How does it work?

- **Lexer** which interprets a string into a Lucene Query using JavaCC
- Human- entered text, not for program-generated text
- All others, such as date ranges, keywords, etc. are better added directly through the **query API**

Query Parser

How does it work?

- **Lexer** which interprets a string into a Lucene Query using JavaCC
- Human- entered text, not for program-generate text
- All others, such as date ranges, keywords, etc. are better added directly through the **query API**



Terms

A query is broken up into terms and operators. There are two types of terms: Single Terms and Phrases.

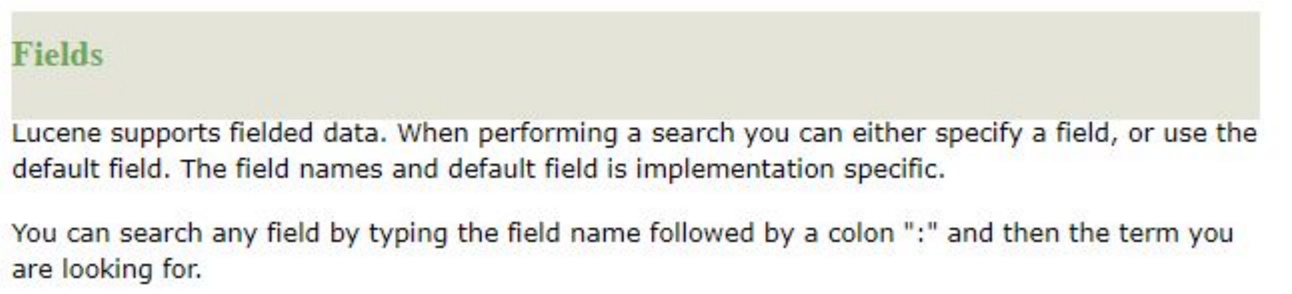
A Single Term is a single word such as "Death" or "Penalty".

A Phrase is a group of words surrounded by double quotes such as "Death Penalty".

Query Parser

How does it work?

- **Lexer** which interprets a string into a Lucene Query using JavaCC
- Human- entered text, not for program-generate text
- All others, such as date ranges, keywords, etc. are better added directly through the **query API**



Fields

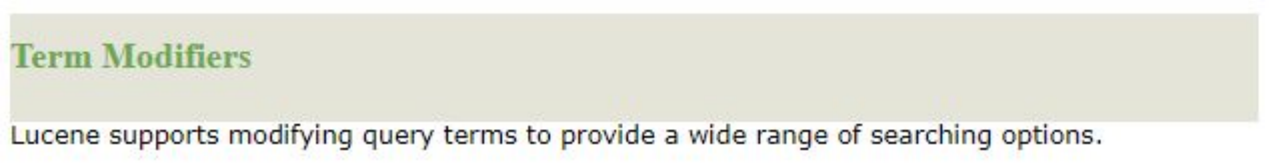
Lucene supports fielded data. When performing a search you can either specify a field, or use the default field. The field names and default field is implementation specific.

You can search any field by typing the field name followed by a colon ":" and then the term you are looking for.

Query Parser

How does it work?

- **Lexer** which interprets a string into a Lucene Query using JavaCC
- Human- entered text, not for program-generate text
- All others, such as date ranges, keywords, etc. are better added directly through the **query API**



Term Modifiers

Lucene supports modifying query terms to provide a wide range of searching options.

Query Parser

How does it work?

- **Lexer** which interprets a string into a Lucene Query using JavaCC
- Human- entered text, not for program-generate text
- All others, such as date ranges, keywords, etc. are better added directly through the **query API**

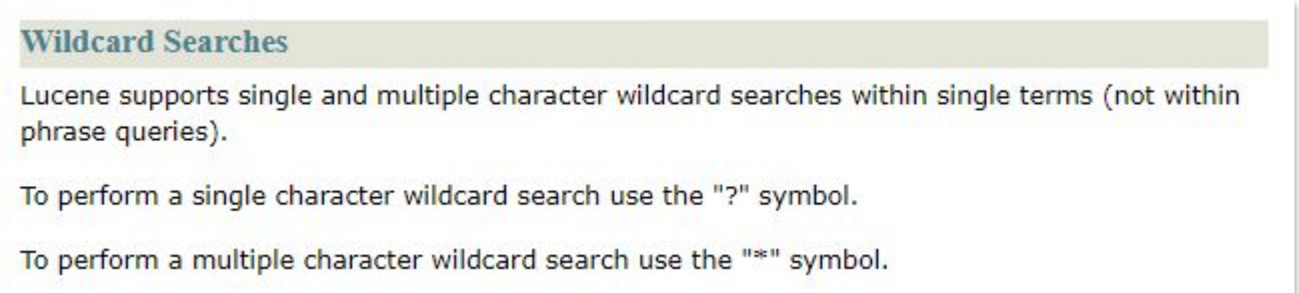
Term Modifiers

- Wildcard Searches
- Fuzzy Searches
- Proximity Searches
- Range Searches
- Boosting a Term

Query Parser

Term Modifiers

1



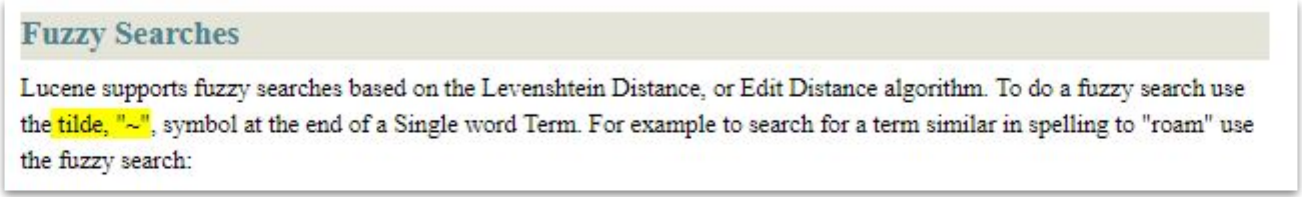
Wildcard Searches

Lucene supports single and multiple character wildcard searches within single terms (not within phrase queries).

To perform a single character wildcard search use the "?" symbol.

To perform a multiple character wildcard search use the "*" symbol.

2



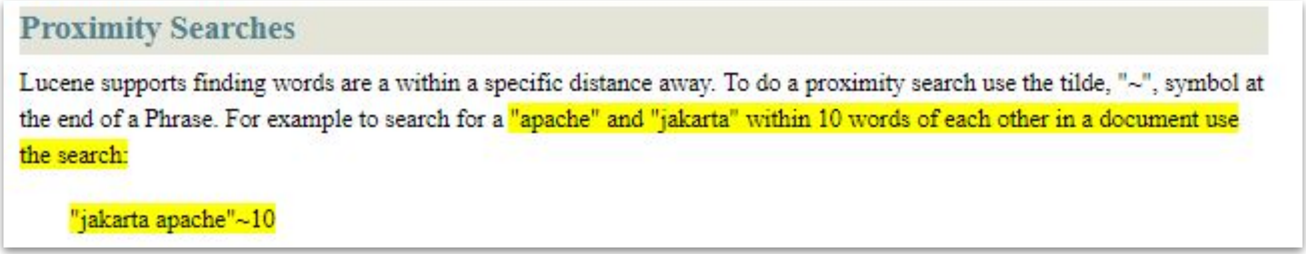
Fuzzy Searches

Lucene supports fuzzy searches based on the Levenshtein Distance, or Edit Distance algorithm. To do a fuzzy search use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search:

Query Parser

Term Modifiers

3

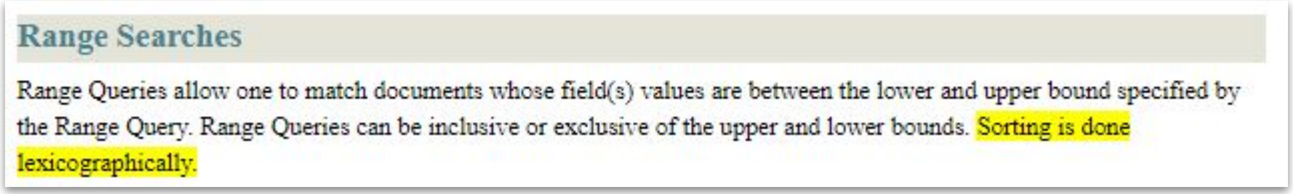


Proximity Searches

Lucene supports finding words are a within a specific distance away. To do a proximity search use the tilde, "~", symbol at the end of a Phrase. For example to search for a "apache" and "jakarta" within 10 words of each other in a document use the search:

```
"jakarta apache"~10
```

4

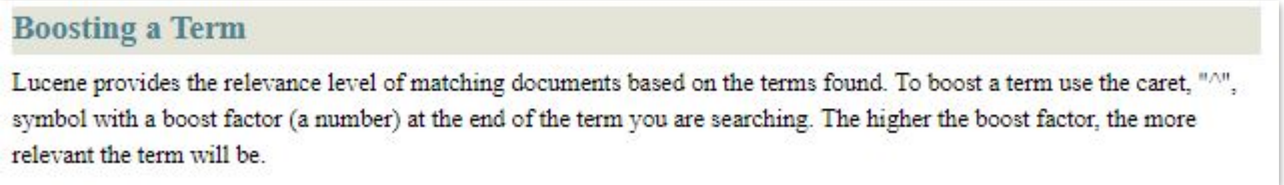


Range Searches

Range Queries allow one to match documents whose field(s) values are between the lower and upper bound specified by the Range Query. Range Queries can be inclusive or exclusive of the upper and lower bounds. Sorting is done lexicographically.

Query Parser

Term Modifiers



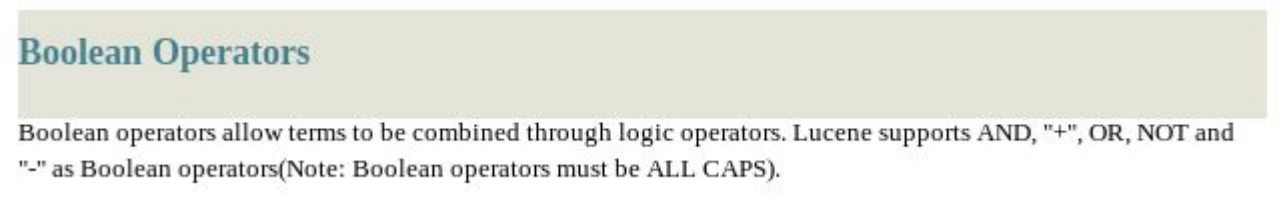
Boosting a Term

Lucene provides the relevance level of matching documents based on the terms found. To boost a term use the caret, "^", symbol with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.

Query Parser

Boolean Operators

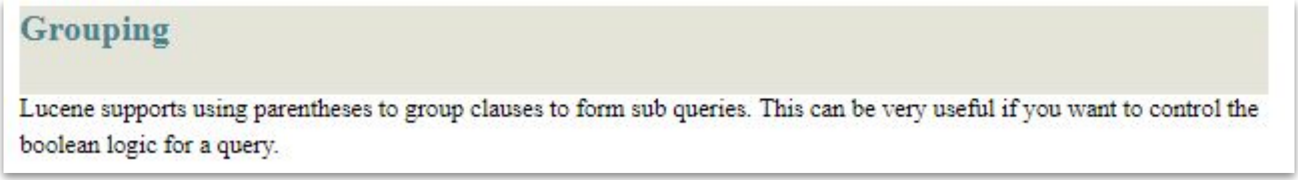
1



Boolean Operators

Boolean operators allow terms to be combined through logic operators. Lucene supports AND, "+", OR, NOT and "-" as Boolean operators(Note: Boolean operators must be ALL CAPS).

2



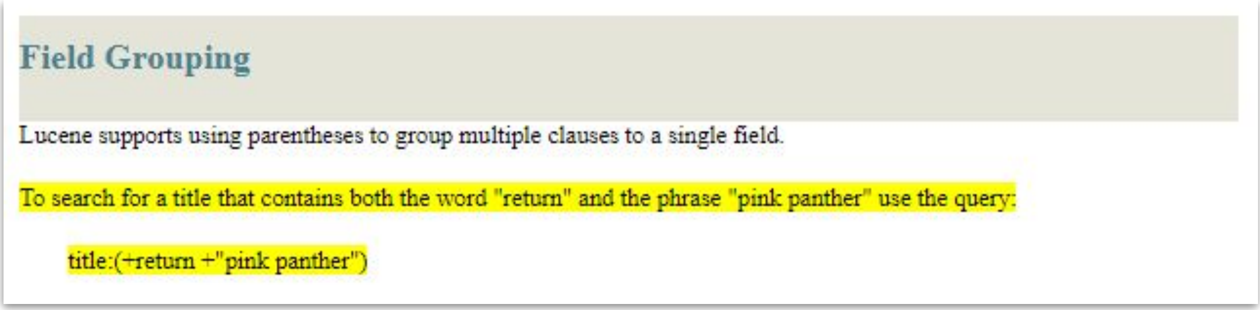
Grouping

Lucene supports using parentheses to group clauses to form sub queries. This can be very useful if you want to control the boolean logic for a query.

Query Parser

Boolean operators

3



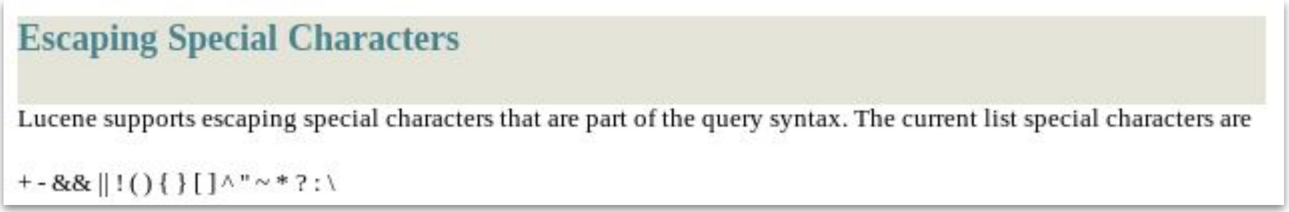
Field Grouping

Lucene supports using parentheses to group multiple clauses to a single field.

To search for a title that contains both the word "return" and the phrase "pink panther" use the query:

```
title:(+return +"pink panther")
```

6



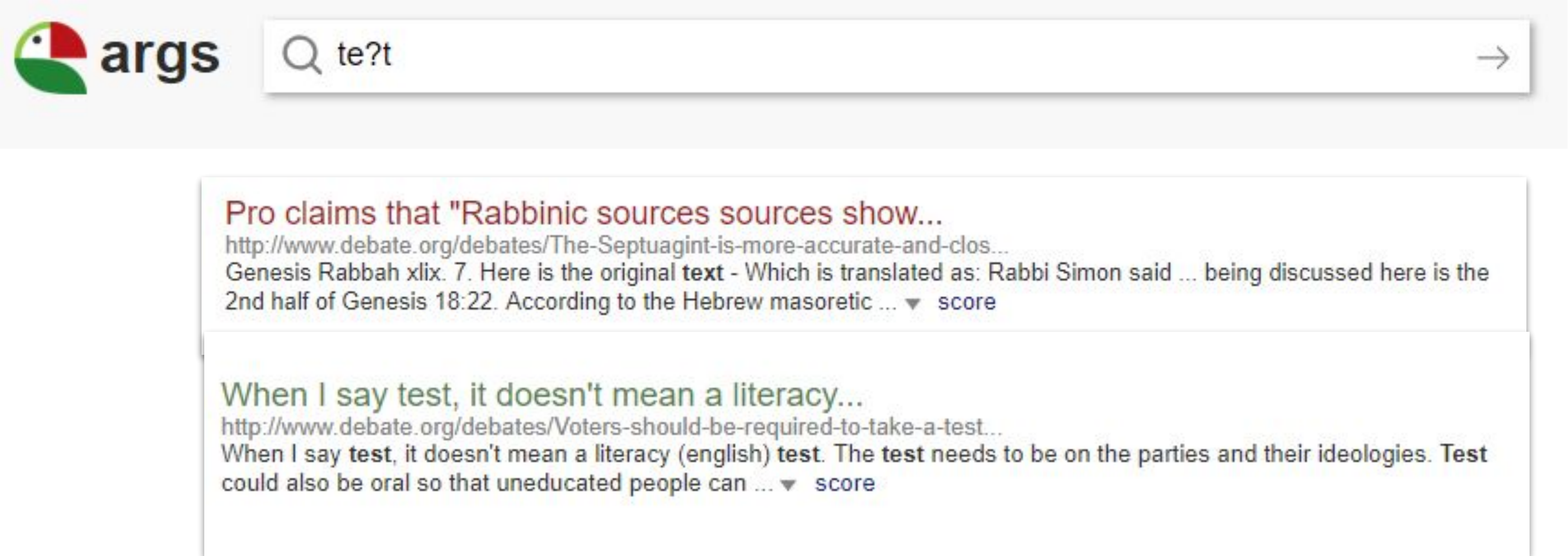
Escaping Special Characters

Lucene supports escaping special characters that are part of the query syntax. The current list special characters are

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \
```

Query Parser

Examples



Pro claims that "Rabbinic sources sources show...

http://www.debate.org/debates/The-Septuagint-is-more-accurate-and-clos...

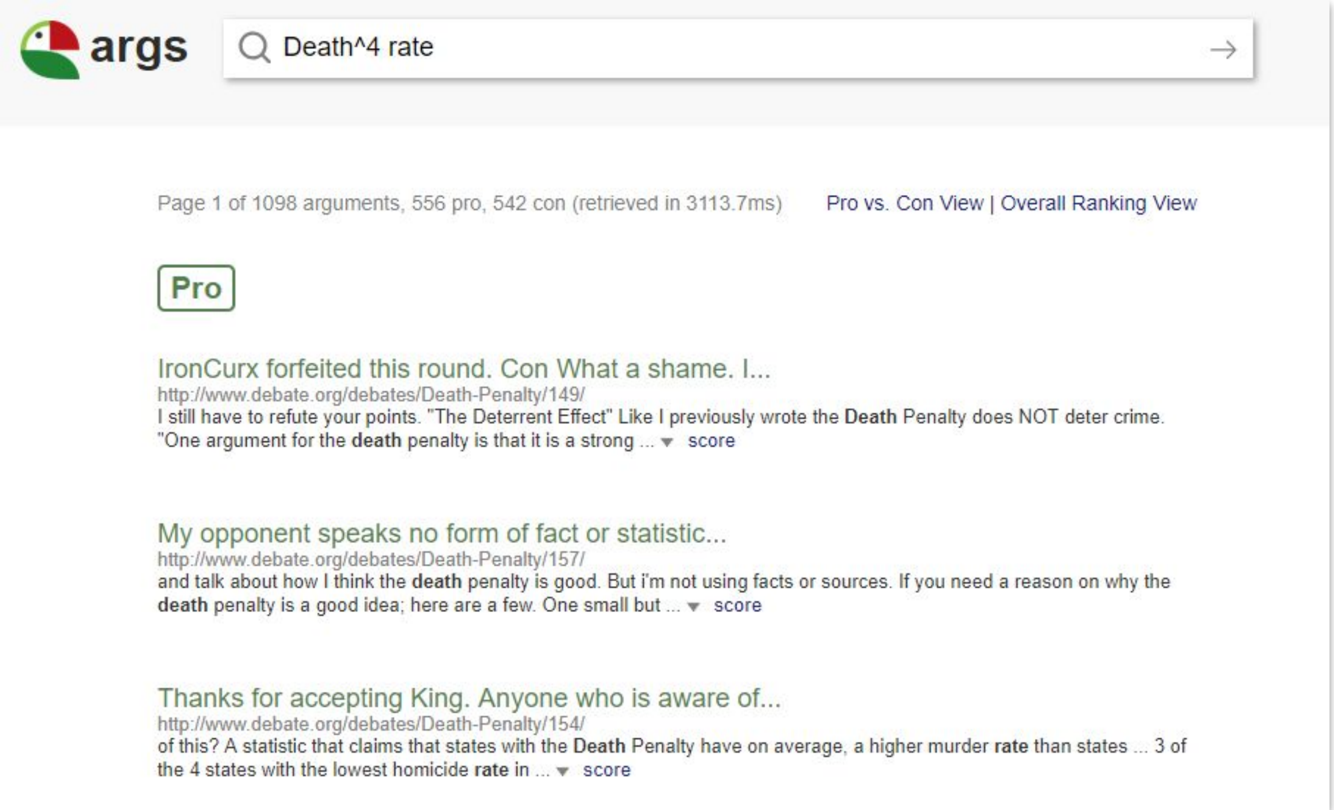
Genesis Rabbah xlix. 7. Here is the original **text** - Which is translated as: Rabbi Simon said ... being discussed here is the 2nd half of Genesis 18:22. According to the Hebrew masoretic ... ▾ score

When I say test, it doesn't mean a literacy...

http://www.debate.org/debates/Voters-should-be-required-to-take-a-test...

When I say **test**, it doesn't mean a literacy (english) **test**. The **test** needs to be on the parties and their ideologies. **Test** could also be oral so that uneducated people can ... ▾ score

Query Parser



Death^4 rate

Page 1 of 1098 arguments, 556 pro, 542 con (retrieved in 3113.7ms)

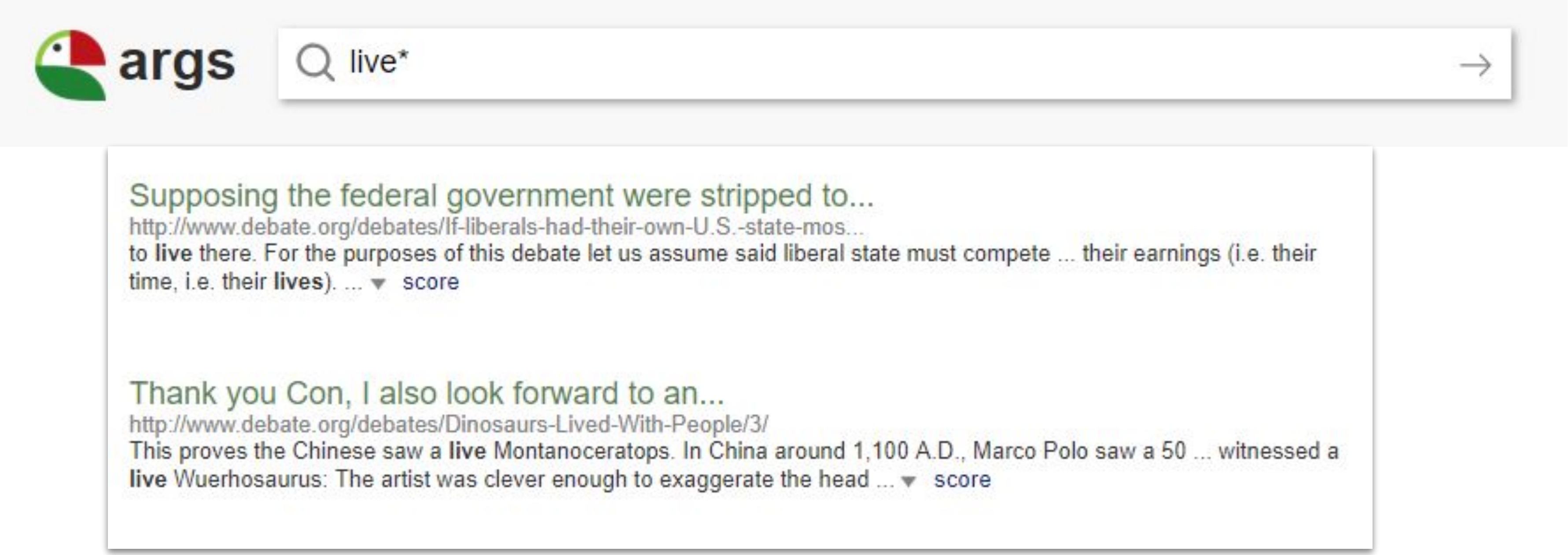
IronCurx forfeited this round. Con What a shame. I...
http://www.debate.org/debates/Death-Penalty/149/
I still have to refute your points. "The Deterrent Effect" Like I previously wrote the **Death** Penalty does NOT deter crime. "One argument for the **death** penalty is that it is a strong ... ▾ score

My opponent speaks no form of fact or statistic...
http://www.debate.org/debates/Death-Penalty/157/
and talk about how I think the **death** penalty is good. But i'm not using facts or sources. If you need a reason on why the **death** penalty is a good idea; here are a few. One small but ... ▾ score

Thanks for accepting King. Anyone who is aware of...
http://www.debate.org/debates/Death-Penalty/154/
of this? A statistic that claims that states with the **Death** Penalty have on average, a higher murder **rate** than states ... 3 of the 4 states with the lowest homicide **rate** in ... ▾ score

Query Parser

Examples



live*

Supposing the federal government were stripped to...

http://www.debate.org/debates/If-liberals-had-their-own-U.S.-state-mos...

to **live** there. For the purposes of this debate let us assume said liberal state must compete ... their earnings (i.e. their time, i.e. their **lives**). ... ▾ score

Thank you Con, I also look forward to an...

http://www.debate.org/debates/Dinosaurs-Lived-With-People/3/

This proves the Chinese saw a **live** Montanoceratops. In China around 1,100 A.D., Marco Polo saw a 50 ... witnessed a **live** Wuerhosaurus: The artist was clever enough to exaggerate the head ... ▾ score

Apache Lucene

- Indexing & retrieving
- Query Syntax
- Scoring & ranking

Scoring & ranking in Lucene

How does it work?

- Combination of the **Vector Space Model (VSM)** and the **Boolean Model** of Information Retrieval

<u>VSM</u>

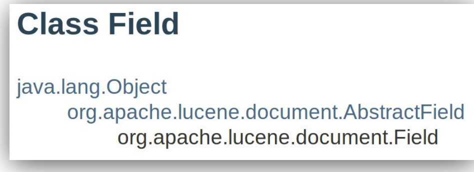
- ➢ Docs represented as multi-dimensional weighted vectors
    - ○ Dimensions = All index terms
    - ○ Weights = Tf-idf values
- ➢ Tf-idf = Term frequency - inverse document frequency
- ➢ “Rare” terms get weighted more

<u>Boolean Model</u>

- ➢ Based on Set theory and boolean logic
- ➢ Documents and user query conceived as a set of terms
- ➢ Narrows down documents based on query/terms

Scoring & ranking in Lucene

Scoring documents:



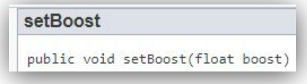
Class Field

java.lang.Object
    org.apache.lucene.document.AbstractField
        org.apache.lucene.document.Field

- ➢ A document is a collection of Fields
  - ○ Each Field specifies how it is created and stored
    - → Field.Index (whether and how it should be indexed)
    - → Field.Store (whether and how it should be stored)

- ➢ Lucene Scoring works on fields and combines the results to return documents

- ➢ Lucene allows influencing search results by boosting
  - ○ Boosting works on documents, fields and queries
  - ○ Multiplies the **documents/fields** by a given float value
  - ○ Multiplies the score of documents matching a **query** clause by a float value

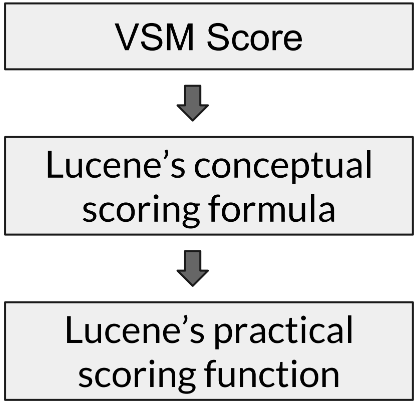


setBoost

```
public void setBoost(float boost)
```

Scoring & ranking in Lucene

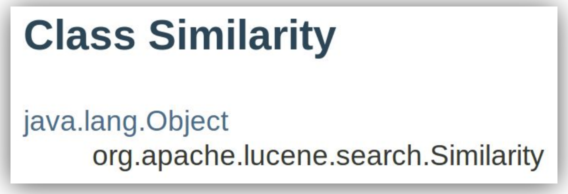
Scoring-formula:



VSM Score

Lucene’s conceptual scoring formula

Lucene’s practical scoring function

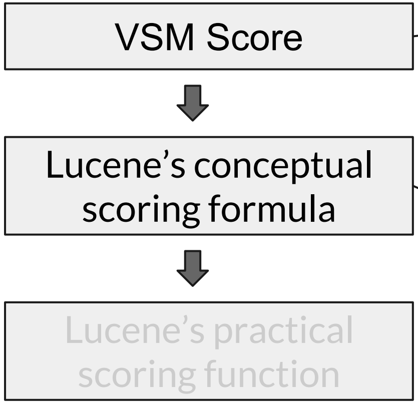


Class Similarity

java.lang.Object
org.apache.lucene.search.Similarity

Scoring & ranking in Lucene

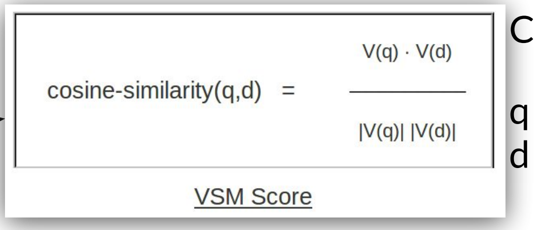
Scoring-formula:



VSM Score

Lucene’s conceptual scoring formula

Lucene’s practical scoring function

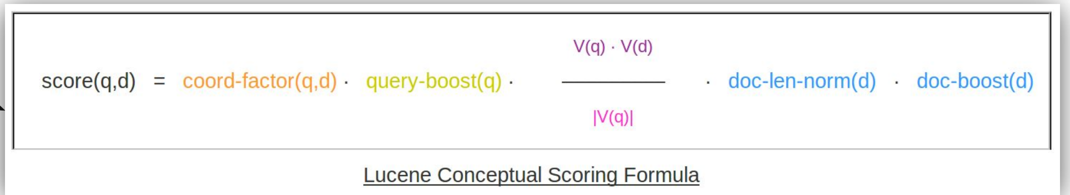


$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{|V(q)|\ |V(d)|}$$

VSM Score

Core algorithm of Lucene Scoring

q = query
d = document



$$\text{score}(q,d) = \text{coord-factor}(q,d) \cdot \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

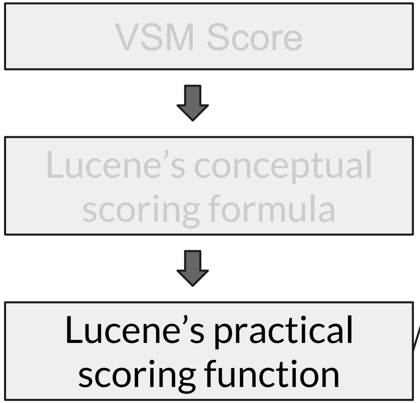
Lucene Conceptual Scoring Formula

Refines VSM Score

- ➢ Different length normalization factor (to avoid problems)
- ➢ Document-, field-, query-boosts
- ➢ Field based
- ➢ Document may match a multi term query without containing all the terms of the query

Scoring & ranking in Lucene

Scoring-formula:

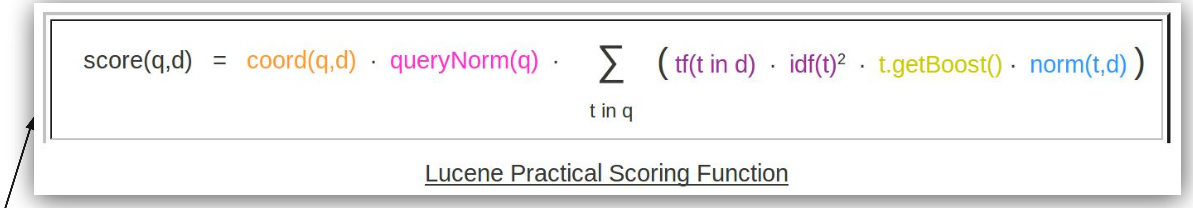


VSM Score

Lucene's conceptual scoring formula

Lucene's practical scoring function

q = query d = document t = term



$$\mathrm{score}(q,d) = \mathrm{coord}(q,d) \cdot \mathrm{queryNorm}(q) \cdot \sum_{t \text{ in } q} \left(\mathrm{tf}(t \text{ in } d) \cdot \mathrm{idf}(t)^2 \cdot t.\mathrm{getBoost}() \cdot \mathrm{norm}(t,d) \right)$$

Lucene Practical Scoring Function

coord(q,d):
How many query terms are found in document *d*?

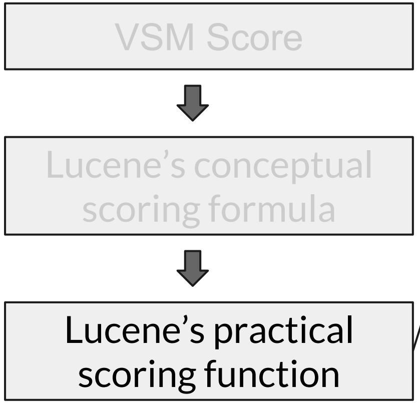
queryNorm(q):
normalizing factor to make scores between queries comparable

tf(t in d):
Frequency of term *t* in document *d*

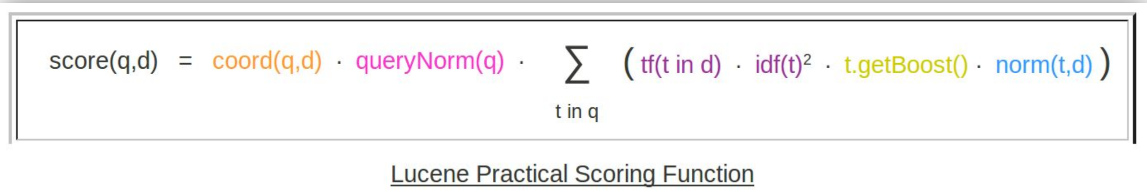
idf(t)²:
Inverse of number of documents in which term *t* appears
(appears in both query and document)

Scoring & ranking in Lucene

Scoring-formula:


VSM Score
Lucene's conceptual scoring formula
Lucene's practical scoring function

q = query   d = document   t = term



$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} \left(\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost}() \cdot \text{norm}(t,d) \right)$$

Lucene Practical Scoring Function

- Lucene is fast, because a lot of pre-calculation can be done before computing specific scores
  - Query boosts for each query term are known before search starts
  - Query Euclidean Norm can be computed before search starts
  - Document length norm and document boost are known at indexing time

Scoring & ranking in Lucene

Package org.apache.lucene.search

Code to search indices.

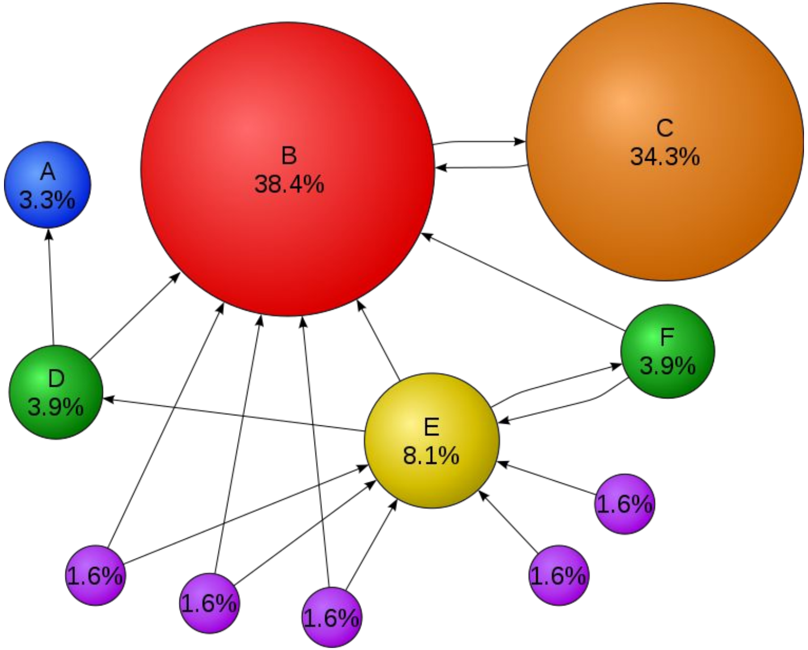
Important parts of Lucene Scoring:

- Use and interaction between Query classes. Several unique Query-classes that can be used:
  - TermQuery (***fieldName***, ***term***)
    - → Returns all documents that have a field named ***fieldName*** containing the word ***term***
    - → Most often used & most simple Query class
  - BooleanQuery
    - → Multiple TermQuery instances combined with boolean operators (should, must, must not)
  - FuzzyQuery
    - → Returns documents that contain terms *similar* to the specified term
- Implementations can be combined in a wide variety of ways
- Developers can implement their own Query Class

Popular Ranking Algorithms

PageRank:

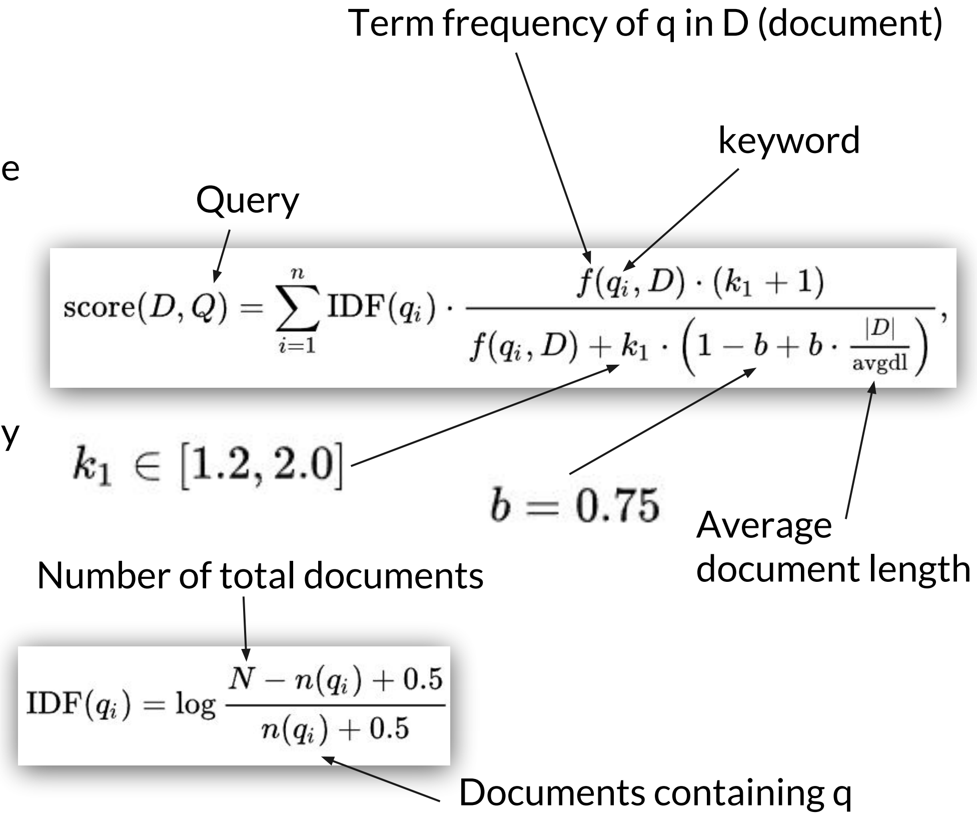
- ➢ Counts the number and quality of links to a page
- ➢ Then determines a rough estimate of how important the website is
- ➢ More important websites are likely to receive more links from other websites
- ➢ Was used by Google but not updated anymore



Popular Ranking Algorithms

Okapi BM25:

- ➢ Uses “bag of words”-model (Vector Space Model)
- ➢ Ranks documents based on the query terms appearing in each document
- ➢ Ignores inter-relationship between query terms in a document (e.g.: their relative proximity)
- ➢ Represents TF-IDF-like retrieval functions
- ➢ **BM25F** interprets documents as fields



Term frequency of q in D (document)

keyword

Query

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)},$$

$$k_1 \in [1.2, 2.0]$$

$$b = 0.75$$

Average document length

Number of total documents

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Documents containing q