Implementing Web Intelligence: A Search Framework

NetDB

CS, NTHU,

Fall, 2013

Where Are We?

- You have learned how to analyze text efficiently
  - Based on streams
  - Flyweight tokens
- But how can we capitalize on this analyzer framework to build a recommendation engine?
  - Given a document, we need a search similar documents efficiently

Today's topic

1. Create a Document class that is general enough to contain the searchable content
   - E.g., the content of a Definition object, a WiKi page, etc.
2. Implement an Analyzer that extracts the “terms” from a Document object (doc for short)
3. An **IndexWriter** that calculates the statistics about the terms/docs (e.g., term frequencies normalized by doc norms) and stores them into an **Index**
   - Basically, an index is a map from the terms to document statistics and IDs
4. An **IndexSearcher** that to find the IDs of the most similar docs to a given doc from index
   - Calculate cosine similarity by looking up the index term-by-term

The Search Framework

3.1. Models docs as vectors

- Tokens (from the analyzer framework) as dimensions

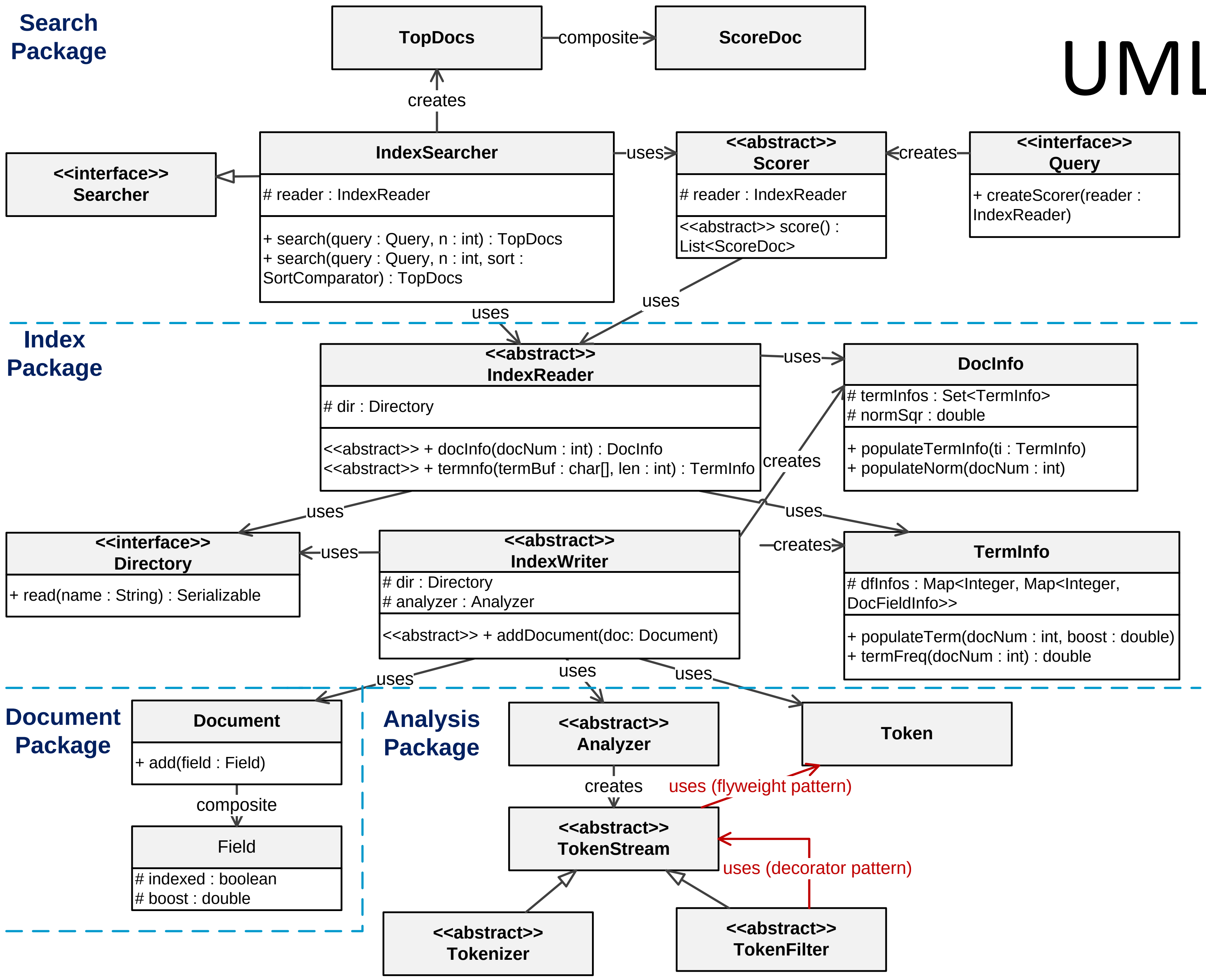
3.2 **IndexWriter** calculates and stores *tf* and *idf* of each vector into the **TermInfo** and **DocInfo** objects, which is kept inside an **IndexFile** in some **Directory**

4.1. Implements a **MoreLikeThisQuery**

- "This" can be a doc (in a recommendation engine) or an ad hoc text (in a search engine)
- Still a vector

4.2. The **IndexSearcher** answers the query by returning a list of **ScoreDoc** objects, each points to a doc ID and the score (i.e., relevance) of the doc

UML


Search Package
TopDocs
composite
ScoreDoc
creates
<<interface>> Searcher
IndexSearcher
# reader : IndexReader
+ search(query : Query, n : int) : TopDocs
+ search(query : Query, n : int, sort : SortComparator) : TopDocs
uses
<<abstract>> Scorer
# reader : IndexReader
<<abstract>> score() : List<ScoreDoc>
creates
<<interface>> Query
+ createScorer(reader : IndexReader)
uses
uses
Index Package
<<abstract>> IndexReader
# dir : Directory
<<abstract>> + docInfo(docNum : int) : DocInfo
<<abstract>> + termnfo(termBuf : char[], len : int) : TermInfo
uses
DocInfo
# termInfos : Set<TermInfo>
# normSqr : double
+ populateTermInfo(ti : TermInfo)
+ populateNorm(docNum : int)
creates
uses
uses
<<interface>> Directory
+ read(name : String) : Serializable
uses
<<abstract>> IndexWriter
# dir : Directory
# analyzer : Analyzer
<<abstract>> + addDocument(doc: Document)
creates
TermInfo
# dfInfos : Map<Integer, Map<Integer, DocFieldInfo>>
+ populateTerm(docNum : int, boost : double)
+ termFreq(docNum : int) : double
uses
uses
uses
Document Package
Document
+ add(field : Field)
composite
Field
# indexed : boolean
# boost : double
Analysis Package
<<abstract>> Analyzer
Token
creates
uses (flyweight pattern)
<<abstract>> TokenStream
uses (decorator pattern)
<<abstract>> Tokenizer
<<abstract>> TokenFilter

Core Classes (1/5)

- IndexWriter
  - Creates and maintains an index
  - addDocument(doc) writes the statistics of a doc into the index
    - Returns a docNum
- Directory
  - The place where an index is physically placed
    - E.g., RamDirectory, FileDirectory, JdbcDirectory, etc.
  - Serializes index into byte[]
- IndexReader
  - Provides access to the statistics in an index

Core Classes (2/5)

- TermInfo
  - Stores statistics of a term (across docs)
  - termFreq(docNum) returns tf inside a doc
  - docFreq() return doc frequency
- DocInfo
  - Stores statistics of a doc
  - termInfos() returns all TermInfo objects corresponding to the terms appearing in this doc

Core Classes (3/5)

- Query
  - Specifies a reference vector
- MoreLikeThisQuery
  - Specifies a doc in an index as a reference vector
  - Constructor: MoreLikeThisQuery(docNum)

Core Classes (4/5)

- Searcher
  - Can capitalize on an Indexreader: IndexSearcher(reader)
  - search(query) returns all similar docs in an index
    - In decreasing order along the scores
- Scorer
  - Performs the actual scoring for each doc
  - Created by Query rather than Searcher
    - Why?

Core Classes (5/5)

- TopDocs
  - Gathers the top score docs
- ScoreDoc
  - Contains docNum and score of a document
  - Does **not** contains the Document object
    - To retrieve the Document object, Call docInfo(docNum).document() instead
  - Why?

Storing the Term Statistics (1/3)

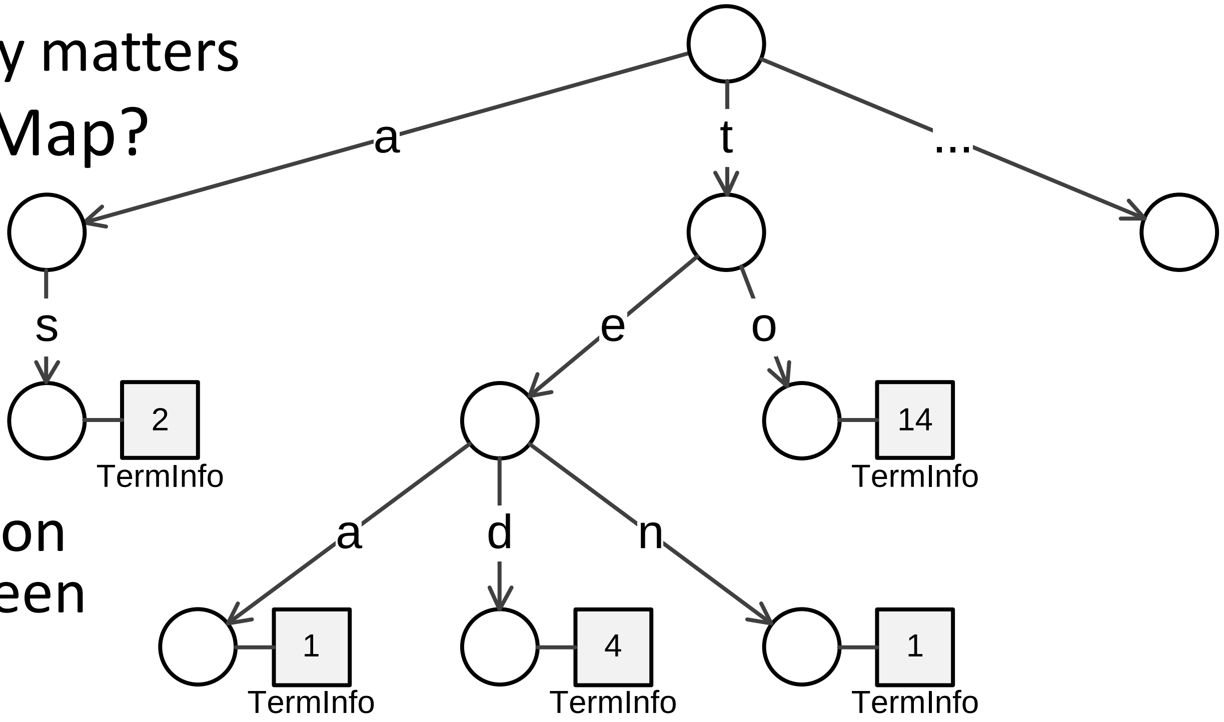
- There can be many, many statistic objects
  - Due to many, many terms
- Unlike the case in analyzer, the flyweight pattern does not help
  - We need to store all these statistics
  - To be used later to score docs when answering a query
- In the memory or files?

Storing the Term Statistics (2/3)

- Strategy 1:
  - Write statistic objects into a file system
  - Only load objects of the currently working term into memory
- Pros:
  - Generally, there is no space limitation in a file system
- Cons:
  - Complex; needs sophisticated file structure design and good I/O and caching implementation to be fast

Storing the Term Statistics (3/3)

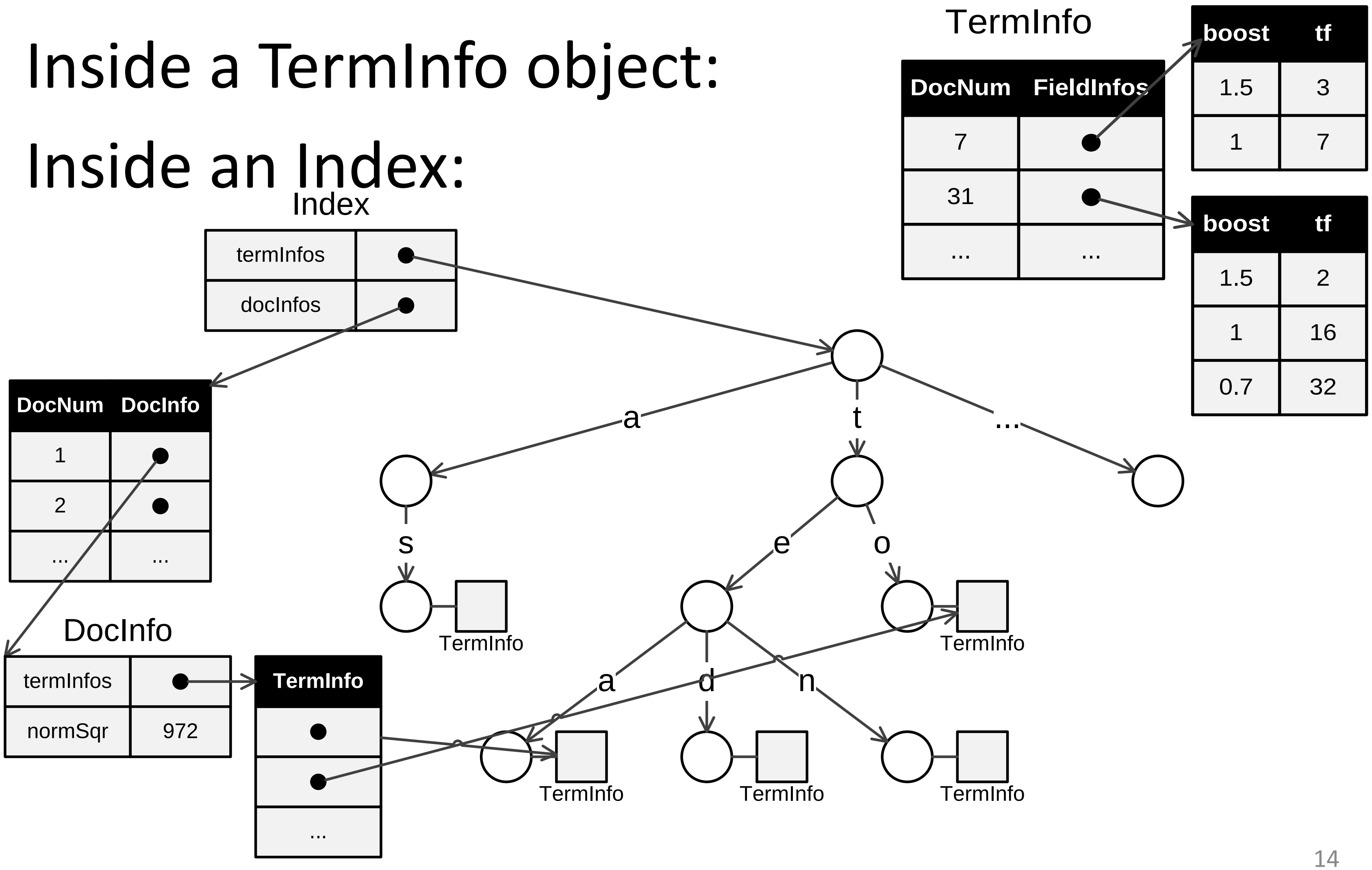
- Strategy 2: Write statistic objects into the memory
- Pros & cons?
  - Space complexity matters
- Can we use HashMap?
- Better choice?



  - Share the common characters between terms
  - Known as a **Trie**

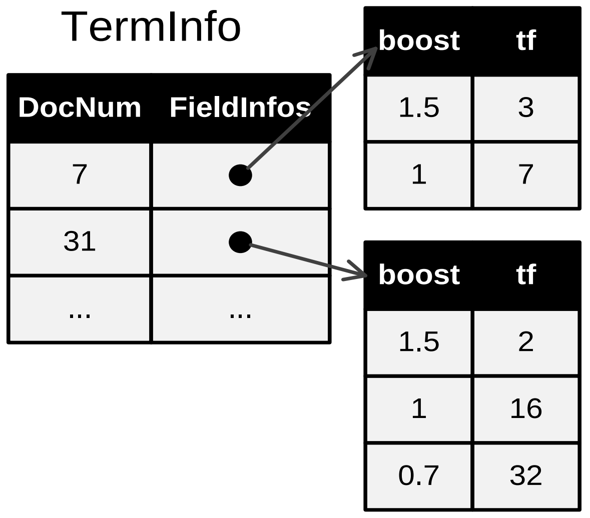
What’s Inside an Index?

- Inside a TermInfo object:
- Inside an Index:



Searching

- With the above index structure, there is no need to examine every doc in an index to get TopDocs
- How?
- To search a term, we can get its TermInfo which has a list of documents containing this term



Notes about the Search Framework

- Not tightly coupled with text analyzers
  - A “term” can be a user action (“like”, rating, etc.)
  - You can design a new index to store these statistics efficiently
- Supports various query types
  - E.g., term queries, range queries, composite queries (Q3 = Q1 + Q2 + ...)
  - You can design new scoring function
- You can see Apache Lucene for more sophisticated searching functions

Lab (1/3)

- Checkout the project “recommender” from svn
- Implement the TermQuery that accepts an arbitrary string (e.g., "apache lucene", or "I like Software Studio", etc.) from user such that the IndexSearcher will find the most relevant docs for this string
  - This type of queries makes our framework a search engine
  - Can be reused in your final project

Lab (2/3)

- Any change to scoring?
- Some observations:
  - The terms in a query string is equally important. You don't need to compute tf, idf and norm for a query string
  - The score of a document in TermQuery is the sum of the scores of all its matching terms to the query
  - The score of a matching term is computed by the tf, idf, and norm of the document

Lab (3/3)

- Tips:
  - We have implemented a template TermQuery for you already!
- What should you fill into the blank?
  1. Computes the score of each term of each document recorded in the TermInfo
  2. Store this term score into scoreDocs so different term scores can be accumulated to form the score of a document