The Lucene Full-Text Search Engine

• Topics
  • Finish up HITS/PageRank
  • Full text in databases
  • Lucene overview, architecture and algorithms

• Learning objectives
  • Explain how the Lucene search engine works.
  • Explain what data structures Lucene uses and why.
  • Identify cases where Lucene is the correct tool to get a job done.
HITS Algorithm

- Models the web as a directed graph.
- Each page is a node; each hyperlink is an edge.
- Sampling step: collect a set of pages called the base set.
  - Base set contains very relevant documents
  - May be quite large
- Iteration Step: find good authorities and hubs in the base set.
- Creating the base set:
  1. Perform a boolean query on search terms and retrieve list of all matching pages. Call this the root set.
  2. Add all link pages: pages that contain a link to an element of the root set or that are referenced by a link in the root set.
  3. Any page in this base set is called a base page.
The Iteration Step

- Goal: Find the best authorities and hubs.
  1. Assign a hub weight and authority weigh to each base page; weights describe a page’s quality as a hub/authority.
  2. Initialize all weights 1.
  3. On each iteration, update authority/hub weight for page p:
     1. \( a_p = \sum h_q \), summing over all base pages q that have a link to p
     2. \( h_p = \sum a_q \), summing over all base pages q that have a link to p
- Key is the links, not the actual terms on the pages in the base set.
- Google uses something called Page Rank instead of this hub/authority ranking function.
From HITS to Page Rank

• Page rank (PR) captures the behavior of a random user who:
  • Picks a page at random
  • Follows links (clicks on them) until he gets bored
  • Selects a new random page
• PR forms a probability distribution over all web pages (such that the sum of the probabilities of all pages sums to 1).
• Let $T_i$ be the $i^{th}$ page that points to page A
• Let $C(T_i)$ be the number of links on page $T_i$
• Let $d$ be a constant (equal to about .85 for Google)

$$PR(A) = (1-d) + \frac{D(PR(T_1)/C(T_1)) + \ldots + PR(T_n)/C(T_n))}{C(T_1)}$$
Text in Databases

• Extensions to SQL include a FullText data type.
• Methods associated with this type support:
  • searching for individual words in the text
  • searching for phrases
  • performing “sounds like” queries
  • Keywords
    • CONTAINS: checks if the FullText object contains the search term(s)
    • RANK: returns the relevance of a particular object to the search term
    • IS ABOUT: Indicates if the particular object is relevant to the query term.
  • Implementation of the ranking algorithm and processing of IS ABOUT is implementation specific.
Lucene Overview

• Part of the Apache project
• High-performance, full-featured text search engine:
  • Over 20 MB/minute indexing on 1.5GHz Pentium
  • Requires only 1 MB heap
  • Incremental indexing
  • Index is 20-30% of input text
  • Ranked search
  • Concurrent update/search
  • Field-searching, range-searching, sorting
• Embedded search engine: you supply the GUI/UI
• Written in Java
Functionality

• Ranked Searching
  • Phrase matching, “Harvard undergraduate admissions”
  • Keyword matching, hamster OR gerbil
  • Incorporates term frequency

• Flexible Queries
  • Phrases, Wildcards, etc...

• Field-specific Queries
  • title, artist, album (anything tagged in semi-structured data)

• Sorting (by fields in documents; default is sort by rank)
Who Uses Lucene?

• My perception: everyone who needs a full-text engine.
• Apple
• LinkedIn
• Nutch (open source web search engine)
• Eclipse IDE
• IBM OmniFind Personal email Search
• Salesforce.com search UI
• MIT Dspace
• Akamai website
• CNET reviews
Architecture (1)

- Document-type-specific parsers generate textual contents (e.g., HTML, Word, PDF, text, etc).
- Each document is assigned a unique ID as it is indexed.
- Documents are assigned to segments.
- Each segment is a self-contained index.
- Segments can be searched in parallel.
- Searches can involve multiple segments and/or multiple indexes.
Architecture (2)
Architecture (3)

- Indexes are based upon an inverted index structure.
- Recall what an inverted index is:
  - Keys are terms
  - Data is a list of documents that contain the term
- Key concepts: Index, document, field, term
  - Documents: A sequence of fields
  - Field: A named sequence of terms (handles semi-structured data)
  - Term: A string; a unit of indexing (name value pair where the value is the actual text)
Indexing

- **Segment**
  - Contains a set of indexed documents.
  - Number of segments determined by the number of documents indexed and the maximum number of documents per segment.

- Segments maintain the following information:
  - A set of names for the different fields used in the index of which this segment is a part.
  - Stored Field Values
  - Term Dictionary
  - Term Frequency data
  - Term Proximity data
  - Normalization factors
Definitions (1)

- **Stored Field Values**
  - Keyed by document number.
  - A list of attribute/value pairs.
  - Examples: URL, title
  - When a search returns a document, it returns these stored field values.

- **Term Dictionary**
  - One entry per term
  - All terms appearing in any indexed fields of any document.
  - Also contains the Document IDs of each document containing the term and pointers to the frequency and position information (think *Postings File*).
Definitions (2)

• **Term Frequency data**
  - One entry per term.
  - A set of pairs indicating the documents and frequency in those documents in which the term appears.

• **Term Proximity data**
  - One entry per term.
  - A set of tuples identifying the positions in each document in which the term appears.

• **Normalization factors**
  - One entry per field in each document.
  - A value that is multiplied into the score for hits on that field.
Concurrency

• Segments are never modified.
• Instead, create new segments or merge existing segments into a new segment.
• Searches never need read locks.
Building the Index

• Tokenize documents.
• Transform tokens: stemming, language translation, etc.
• Filter tokens:
  • Stop-word elimination
  • Application-specific elimination
• Then, an IndexWriter builds the index.
• The FSDirectory stores the index on disk.
• Incremental Index Creation
  • Maintain stack of segment indices
  • Create a segment index for a new document
  • Push new index on stack
  • Indexes will be of size $M^i$ where $M$ is the merge factor.
  • Pop batches of $M$ indexes (all of the same size) off the stack and merge them, push merged index back on.
  • End up with an index of $M \times \log_M(N)/2$ entries (on average)
Index Build: Example (1)

Let’s assume $M = 3$

Queue of segments
Index Build: Example (2)

Let’s assume $M = 3$
Index Build: Example (3)

Let’s assume $M = 3$
Using the Index: Search

- Key idea: Use parallel search (in segments) to compute document scores.
- Consider the vector-space model
  - Each document (and query) is a vector with each component representing a weighted value for a term in that document.
  - Vectors are sparse.
  - For each term in the query, find all documents containing that term.
  - For each document, accumulate a score for that document.
  - *Pure inverted index sort is space inefficient because you have to gradually grow the vectors and their scores for all documents in order to retain the top K.*
  - Some approaches prune early by calculating the max-possible weight of a document; these perform well, and save space by a factor of 50 or so.
Search (continued)

- Normalize vector weights.
- Similarity can be any metric on vectors: cosine, dot-product, Euclidean distance, etc.
- Typical inverted search is space inefficient, so the goal here is to do something better -- want to compute document vectors optimized for the query where the vector has only the relevant search terms.
An Optimized Vector

• Assume there are Q terms in the query.
• A posting stream is the list of entries associated with a term, sorted by document ID.
• Construct a sorted Queue containing the Q posting streams, sorted by document ID.
• Now, you process the head of the queue, pop off the first term (the smallest document ID), compute a partial score and then replace the posting stream back in the Queue in order.
• This means you process all the terms for document $D_i$ before $D_j$ where $i < j$.
• This has created your optimized document vector.
• Now, you just process all the documents and keep the K highest scores.
Mapping the Index onto the File System

• Lucene index consists of five meta-data files and the segments comprising the index.
  • Segment File
  • Fields Information File
  • Text Information File
  • Frequency File
  • Position File
Segment File

- Version: Version of the index files
- SegCount: The number of segments in the index
- NameCounter: Unique integer to name segment files
- SegName: One per each segment in the index; contains the name of the actual file containing the segment.
- SegSize: The size of each segment (one entry per segment).
Fields Information File

- FieldCount: Number of fields
- FieldName: Name of a field
- FieldBits: Flag values (e.g., indexed or non-indexed)
Text Information File

- TIVersion: Version of this file's format
- TermCount: Number of terms in this segment
- Term: Structure containing PrefixLength, Suffix, and Fieldnum; describes a term.
- DocFreq: Number of documents containing this term
- FreqDelta: Points to the frequency file
- ProxDelta: Points to the position file.
Frequency and Position File

- Frequency File
  - DocDelta: Determines the document number and term frequency.
  - Freq: Number of times term appears

- Position File
  - PositionDelta: Position at which each term appears in the document
Tuning

• Not surprisingly, without tuning, the indexing performance of Lucene can suffer, so there are tuning hooks.
  • mergeFactor: How many documents to keep in memory before writing to a new on-disk segment index. Also, once there are mergeFactor segment indexes on disk, they will be merged together. If you have lots of documents, this parameter needs to be increased.
  • minMergeDocs: How many documents are kept in memory before being written to disk. This should be as large as memory capacity will allow.
  • maxMergeDocs: Maximum number of documents per segment index. Larger indexes allow for speedier searches.