Search Engine Optimization

SWE 642, Spring 2008

Nick Duan

Overview

- Understanding Index Structure
  - Lucene Index Structure
  - Indexing APIs
  - Indexing with Metadata
- Distributed Indexing
  - Distributed indexing in Nutch
  - MapReduce Concepts
- Directory-based Search
  - Adding category information as a separate Field
  - TermQuery and BooleanQuery in Lucene
- Summary

Understanding Index Structure

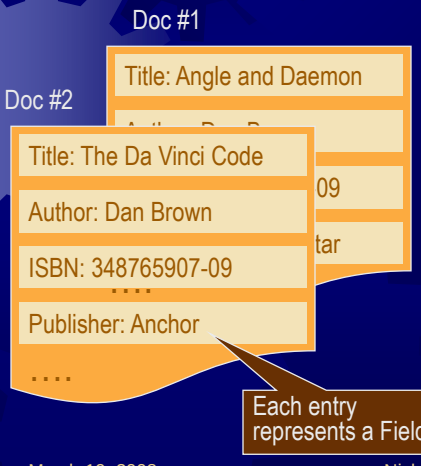
- Indexing of text beyond inverted index
  - Inverted index establishes a basic mechanism (reference pointers of term-to-document) for indexing text, but it is not sufficient to accommodate search needs
  - Terms have to be arranged to enable efficient searches (sorting, hashing, etc..)
  - Index file structure to be optimized (with respect to hardware specs) to speed up searches and save space
- Indexing of metadata to augment additional search options
  - Database: Any field can be indexed!
  - Textual Documents: User-defined metadata as fields
    - Lucene index file structure

Terminology of Lucene

- Document
  - A document, representing a index target or entity, consists of one or more fields. The text content of the document is one of the fields
- Field
  - A field is a name/value pair, with the value representing a sequence of terms
  - Indexing is performed on a field-by-field basis
- Term
  - A term is a word or string

Lucene Index Structure

Document as Index Card



Index File with Fields

Field	Value	Doc Freq
Title	Da Vinci	1
	code	1
	angle	1
	daemon	1
Author	Dan Brown	2
Publisher	Anchor	1
	Pocket Star	1

Lucene Index Files

- Segment: A physical grouping of index files. An index entity in Lucence consists of one or more segments
- Field names (.fnm): Containing the set of field names used in the index.
- Stored Field values (.fdx and .fdt): Containing a list of attribute-value pairs for each document, where the attributes are field names.
- Term dictionary (.tls): Containing all of the terms used in all of the indexed fields of all of the documents, as well as the number of documents which contain the term, and pointers to the term's frequency and proximity data.
- Term Frequency (.frq): Containing the numbers of all the documents that contain each of the terms defined in the dictionary, and the frequency of the term in that document.
- Term Proximity (.prx): For each term in the dictionary, the positions that the term occurs in each document.
- Normalization factors (.nrm): For each field in each document, a value (a single byte value) is stored that is multiplied into the score for hits on that field.
- Term Vectors (.tvx): A term vector consists of term text and term frequency for each field (to be dynamically configured via Field API)
- Deleted documents (.del): An optional file indicating which documents are deleted.

Index APIs in Lucene

- *org.apache.lucene.store.Directory*
  - A single Lucene Index, represented in different forms: *RAMDirectory, FSDirectory*
- *org.apache.lucene.document.Document*
  - A single index target
- *org.apache.lucene.document.Field*
  - A single name/value pair as the basic component of an index
  - Nested classes: *Field.Store, Field.Index, Field.TermVector*
- *org.apache.lucene.index.IndexWriter*
  - Create and maintain an index
- *org.apache.lucene.search.IndexSearcher*
  - Implements search on a single index
- *org.apache.lucene.queryParser.QueryParser*
  - Tokenizes and normalizes a query string
- *org.apache.lucene.analysis.Analyzer*
  - Builds token streams and performs analysis on the terms. Used by both *IndexWriter* and *IndexSearcher*

Integrating with Search Applications

- General Steps:
  - Create *Documents* by adding *Fields*;
  - Create an *IndexWriter* and add documents to it with *addDocument()*;
  - Call *QueryParser.parse()* to build a query from a string; and
  - Create an *IndexSearcher* and pass the query to its *search()* method.

Indexing Steps

```
Analyzer analyzer = new StandardAnalyzer();
Directory directory = new RAMDirectory();
//Directory directory = FSDirectory.getDirectory("/tmp/
testindex");
IndexWriter iwriter = new IndexWriter(directory, analy
zer, true);
iwriter.setMaxFieldLength(25000);
Document doc = new Document();
String text = "This is the text to be indexed.";
doc.add(new Field("fieldname", text, Field.Store.YES,
    Field.Index.TOKENIZED));
iwriter.addDocument(doc);
iwriter.optimize();
iwriter.close();
```

Store index in memory

Store index on file system

Search Steps

```
IndexSearcher  isearcher = new IndexSearcher(directory);
// Parse a simple query that searches for "text":
QueryParser parser = new QueryParser("fieldname", analyzer);
Query query = parser.parse("text");
Hits hits = isearcher.search(query);
// Iterate through the results:
for (int i = 0; i < hits.length(); i++) {
        Document hitDoc = hits.doc(i);
        System.out.println(hitDoc.get("fieldname"));
         // should print out  “This is the text to be indexed.“
}
isearcher.close();
directory.close();
```

The search term

The field name used during indexing

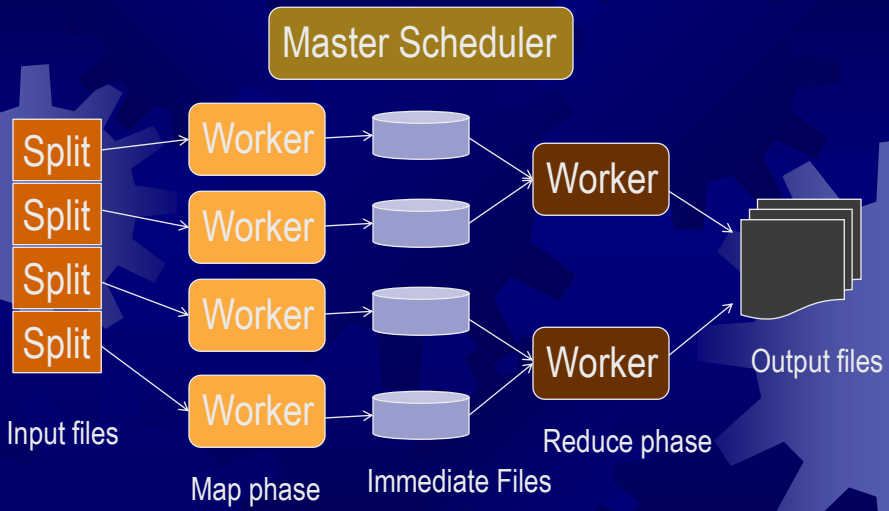
Distributed Indexing

- Index could be a lengthy and time-consuming task when dealing with millions of data records
- Simple solution: Divide and concur
  - Partition the index space into small sections
  - Assign each section to a single CPU or machine
  - Machines work in parallel to perform indexing
  - Sections can be re-assigned if a machine is deadlocked or slow

MapReduce Algorithm

- Used by Google for implementing large-scale, index clusters
- Inherit from the Map and Reducer data structure of Lisp
- Basic concept:
  - A map is defined as a list of key/value pairs. A key could have multiple values
  - Reduce the list to have each key correspond to its value list
  - Perform indexing on the reduced key/value list pairs
- Implemented in Nutch through its Hadoop subproject

MapReduce on Large Cluster



Implementing Directory-based Search

- Objective: Reduce search space by adding category information
- Intersection of both searches on content and category
- There are multiple implementations
  - Creating a where clause with two query fields in an AND condition
  - Creating an intersection of two result sets of two sub-queries
- Category information to be defined as a separate index field in addition to content
- Construct a BooleanQuery to search on both content and category fields (equivalent to constructing a where clause)

Define Category Field for Indexing

```
Document doc = new Document();
/* Define category as a index field without  tokenizing it
*/
doc.add(new Field("category", getCategory(),
    Field.Store.YES, Field.Index.UN_TOKENIZED));

/* Add the contents of the file to a field named "contents".
    Specify a Reader, so that the text of the file is tokenized
    and indexed, but not stored. Note that FileReader expects
    the file to be in the system's default encoding. If that's not
    the case searching for special characters will fail.
*/
 doc.add(new Field("contents", new FileReader(f)));
```


Construct a BooleanQuery

```
TermQuery contentQuery = new TermQuery(new
    Term("content", "bank");
TermQuery catQuery = new TermQuery (new
    Term("category", "news/financial");
BooleanQuery bQuery = new BooleanQuery();
bQuery.add(contentQuery,
    BooleanClause.Occur.MUST);
bQuery.add(catQuery,
    BooleanClause.Occur.SHOULD);
IndexSearcher searcher = new
    IndexSearcher(indexDir);
Hits hits = searcher.search(bQuery);
```

Online References

- Apache Lucene
  - http://lucene.apache.org/java/2_3_1
  - http://wiki.apache.org/lucene-java
- The Anatomy of a Large-Scale Hypertextual Web Search Engine, by Sergey Brin and Lawrence Page
  - http://infolab.standford.edu/~backrub/google.html
- Chapter 4 of Introduction to Information Retrieval, by Manning, et.al
  - http://www-csli.stanford.edu/~hinrich/information-retrieval-book.html
- Lucene in Action, by Otis Gospodnetic & Eric Hatcher, Manning publications, 2005 (examples are using old APIs)

Summary

- Search engine can be optimized by constructing the right index fields and search queries
  - What you search for is what you have indexed
  - Reducing search space and increase search accuracy by creating the right index fields
  - Using various Query types to construct the right search query based on indices
- Distributed Indexing is a common way to reduce the complexity and workload of indexing
  - MapReduce algorithm to divide and concur
  - Applicable to large computer clusters to perform parallel computing

Quiz

- What are the basic steps in Lucene to perform indexing on a directory of docs?
- What are the basic steps in Lucene to search a term in a content field?
- Download Lucene and write a simple program to index two or more documents in different categories, and perform search on both content and categories