Nutch and Lucene Framework

-Arjun Atreya V
RS-IITB
Outline

- Introduction
- Behavior of Nutch (Offline and Online)
- Lucene Features
Resources Used


- Nutch Wiki http://wiki.apache.org/nutch/
Introduction

- Nutch is an opensource search engine
- Implemented in Java
- Nutch is comprised of Lucene, Solr, Hadoop etc..
- Lucene is an implementation of indexing and searching crawled data
- Both Nutch and Lucene are developed using plugin framework
- Easy to customize
Where do they fit in IR?
Nutch – complete search engine

Nutch and Lucene Framework
Nutch – offline processing

- **Crawling**
  - Starts with set of seed URLs
  - Goes deeper in the web and starts fetching the content
  - Content need to be analyzed before storing
  - Storing the content
  - Makes suitable for searching

- **Issues**
  - Time consuming process
  - Freshness of the crawl (How often should I crawl?)
  - Coverage of content
Nutch – online processing

- Searching
  - Analysis of the query
  - Processing of few words (tokens) in the query
  - Query tokens matched against stored tokens (index)
- Fast and Accurate
- Involves ordering the matching results
- Ranking affects User’s satisfaction directly
- Supports distributed searching
Nutch – Data structures

- **Web Database or WebDB**
  - Mirrors the properties/structure of web graph being crawled

- **Segment**
  - Intermediate index
  - Contains pages fetched in a single run

- **Index**
  - Final inverted index obtained by “merging” segments (Lucene)
**Nutch –Crawling**

- **Inject: initial creation of CrawlDB**
  - Insert seed URLs
  - Initial LinkDB is empty

- **Generate new shard's fetchlist**
- **Fetch raw content**
- **Parse content (discovers outlinks)**
- **Update CrawlDB from shards**
- **Update LinkDB from shards**
- **Index shards**
Wide Crawling vs. Focused Crawling

- **Differences:**
  - Little technical difference in configuration
  - Big difference in operations, maintenance and quality

- **Wide crawling:**
  - (Almost) Unlimited crawling frontier
  - High risk of spamming and junk content
  - “Politeness” a very important limiting factor
  - Bandwidth & DNS considerations

- **Focused (vertical or enterprise) crawling:**
  - Limited crawling frontier
  - Bandwidth or politeness is often not an issue
  - Low risk of spamming and junk content
Crawling Architecture
Step1: Injector injects the list of seed URLs into the CrawlDB
Step2 : Generator takes the list of seed URLs from CrawlDB, forms fetch list, adds crawl_generate folder into the segments
Step 3: These fetch lists are used by fetchers to fetch the raw content of the document. It is then stored in segments.
Step 4: Parser is called to parse the content of the document and parsed content is stored back in segments.
Step 5: The links are inverted in the link graph and stored in LinkDB.
Step6 : Indexing the terms present in segments is done and indices are updated in the segments
Step 7: Information on the newly fetched documents are updated in the CrawlDB
Crawling: 10 stage process

```
bin/nutch crawl <urlfile> -dir <dir> -depth <n> >& crawl.log
```

1. **admin db –create**: Create a new WebDB.
2. **inject**: Inject root URLs into the WebDB.
3. **generate**: Generate a fetchlist from the WebDB in a new segment.
4. **fetch**: Fetch content from URLs in the fetchlist.
5. **updatedb**: Update the WebDB with links from fetched pages.
6. Repeat steps 3-5 until the required depth is reached.
7. **updatesegs**: Update segments with scores and links from the WebDB.
8. **index**: Index the fetched pages.
9. **dedup**: Eliminate duplicate content (and duplicate URLs) from the indexes.
10. **merge**: Merge the indexes into a single index for searching
De-duplication Algorithm

(MD5 hash, float score, int indexID, int docID, int urlLen)

for each page
to eliminate URL duplicates from a segmentsDir:
open a temporary file
for each segment:
  for each document in its index:
    append a tuple for the document to the temporary file with hash=MD5(URL)
close the temporary file
sort the temporary file by hash
for each group of tuples with the same hash:
  for each tuple but the first:
    delete the specified document from the index
URL Filtering

- URL Filters (Text file) *(conf/crawl-urlfilter.txt)*
  - Regular expression to filter URLs during crawling
  - E.g.
    - To ignore files with certain suffix:
      - `\.(gif|exe|zip|ico)$`
    - To accept host in a certain domain
      - `+^http://([a-z0-9]*\.)*apache.org/`
Few API’s

- Site we would crawl: http://www.iitb.ac.in
  - bin/nutch crawl <urlfile> -dir <dir> -depth <n> >& crawl.log

- Analyze the database:
  - bin/nutch readdb <db dir> –stats
  - bin/nutch readdb <db dir> –dumppageurl
  - bin/nutch readdb <db dir> –dumplinks
  - s=`ls -d <segment dir> /* | head -1` ; bin/nutch segread -dump $s
Map-Reduce Function

- Works in distributed environment
- map() and reduce() functions are implemented in most of the modules
- Both map() and reduce() functions uses <key, value> pairs
- Useful in case of processing large data (eg: Indexing)
- Some applications need sequence of map-reduce
  - Map-1 -> Reduce-1 -> ... -> Map-n -> Reduce-n
Map-Reduce Architecture

Node 1

File loaded from local HDFS stores

RecordReaders:

Input (k, v) pairs

Intermediate (k, v) pairs

Partitioner

"Shuffling" process

Intermediate (k, v) pairs exchanged by all nodes

(sort)

reduce

Final (k, v) pairs

Writeback to local HDFS store

OutputFormat

Node 2

File loaded from local HDFS stores

Partitioner

Intermediate (k, v) pairs

(sort)

reduce

Final (k, v) pairs

Writeback to local HDFS store

OutputFormat

Nutch and Lucene Framework
Nutch – Map-Reduce Indexing

- Map() just assembles all parts of documents
- Reduce() performs text analysis + indexing:
  - Adds to a local Lucene index

Other possible MR indexing models:

- Hadoop contrib/indexing model:
  - analysis and indexing on map() side
  - Index merging on reduce() side

- Modified Nutch model:
  - Analysis on map() side
  - Indexing on reduce() side
Nutch - Ranking

Nutch Ranking

\[
\text{score}(\vec{q}, \vec{d}) = \text{queryNorm}(\vec{q}) \times \text{coord}(\vec{q}, \vec{d}) \times \text{norm}(t, \vec{d}) \times \\
\sum_{t \in \vec{d}} (tf(t) \times idf(t) \times t.\text{boost}(t.\text{field}))
\]

- queryNorm() : indicates the normalization factor for the query
- coord() : indicates how many query terms are present in the given document
- norm() : score indicating field based normalization factor
- tf : term frequency and idf : inverse document frequency
- t.boost() : score indicating the importance of terms occurrence in a particular field
Lucene - Features

- Field based indexing and searching
- Different fields of a webpage are
  - Title
  - URL
  - Anchor text
  - Content, etc..
- Different boost factors to give importance to fields
- Uses inverted index to store content of crawled documents
- Open source Apache project
Lucene - Index

- Concepts
  - Index: sequence of documents (a.k.a. Directory)
  - Document: sequence of fields
  - Field: named sequence of terms
  - Term: a text string (e.g., a word)

- Statistics
  - Term frequencies and positions
IndexWriter writer =
    new IndexWriter(directory, analyzer, true);

Document doc = new Document();
    // add fields to document (next slide)
writer.addDocument(doc);
writer.close();
Adding Fields

doc.add(Field.Keyword("isbn", isbn));
doc.add(Field.Keyword("category", category));
doc.add(Field.Text("title", title));
doc.add(Field.Text("author", author));
doc.add(Field.UnIndexed("url", url));
doc.add(Field.UnStored("subjects", subjects, true));
doc.add(Field.Keyword("pubmonth", pubmonth));
doc.add(Field.UnStored("contents", author + " " + subjects));
doc.add(Field.Keyword("modified",
     DateField.timeToString(file.lastModified())));
Fields Description

- **Attributes**
  - Stored: original content retrievable
  - Indexed: inverted, searchable
  - Tokenized: analyzed, split into tokens

- **Factory methods**
  - Keyword: stored and indexed as single term
  - Text: indexed, tokenized, and stored if String
  - UnIndexed: stored
  - UnStored: indexed, tokenized

- **Terms are what matters for searching**
Searching an Index

```java
IndexSearcher searcher =
    new IndexSearcher(directory);

Query query = QueryParser.parse(queryExpression,
    "contents",analyzer);

Hits hits = searcher.search(query);
for (int i = 0; i < hits.length(); i++) {
    Document doc = hits.doc(i);
    System.out.println(doc.get("title"));
}
```
Analyzer

- **Analysis occurs**
  - For each tokenized field during indexing
  - For each term or phrase in QueryParser

- **Several analyzers built-in**
  - Many more in the sandbox
  - Straightforward to create your own

- **Choosing the right analyzer is important!**
WhiteSpace Analyzer

The quick brown fox jumps over the lazy dog.

[The] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog.]
The quick brown fox jumps over the lazy dog.

[the] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog]
Stop Analyzer

The quick brown fox jumps over the lazy dog.

[quick] [brown] [fox] [jumps] [over] [lazy] [dog]
The quick brown fox jumps over the lazy dog.

[the] [quick] [brown] [fox] [jump] [over] [the] [lazy] [dog]
Query Creation

- Searching by a term – TermQuery
- Searching within a range – RangeQuery
- Searching on a string – PrefixQuery
- Combining queries – BooleanQuery
- Searching by phrase – PhraseQuery
- Searching by wildcard – WildcardQuery
- Searching for similar terms - FuzzyQuery
## Lucene Queries

<table>
<thead>
<tr>
<th>Query expression</th>
<th>Matches documents that...</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>Contain the term <code>java</code> in the default field</td>
</tr>
<tr>
<td>java junit</td>
<td>Contain the term <code>java</code> or <code>junit</code>, or both, in the default field</td>
</tr>
<tr>
<td>java or junit</td>
<td>Contain both <code>java</code> and <code>junit</code> in the default field</td>
</tr>
<tr>
<td>+java +junit</td>
<td>Contain both <code>java</code> and <code>junit</code> in the default field</td>
</tr>
<tr>
<td>java AND junit</td>
<td>Contain both <code>java</code> and <code>junit</code> in the default field</td>
</tr>
<tr>
<td>title:ant</td>
<td>Contain the term <code>ant</code> in the title field</td>
</tr>
<tr>
<td>title:extreme</td>
<td>Have <code>extreme</code> in the title field and don’t have <code>sports</code> in the subject field</td>
</tr>
<tr>
<td>-subject:sports</td>
<td>Contain <code>methodology</code> and must also contain <code>agile</code> and/or <code>extreme</code>, all in the default field</td>
</tr>
<tr>
<td>title:extreme</td>
<td>Contain <code>methodology</code> and must also contain <code>agile</code> and/or <code>extreme</code>, all in the default field</td>
</tr>
<tr>
<td>subject:sports</td>
<td>Contain <code>methodology</code> and must also contain <code>agile</code> and/or <code>extreme</code>, all in the default field</td>
</tr>
<tr>
<td>title:extreme</td>
<td>Contain <code>methodology</code> and must also contain <code>agile</code> and/or <code>extreme</code>, all in the default field</td>
</tr>
<tr>
<td>AND NOT subject:sports</td>
<td>Contain the exact phrase “<code>junit in action</code>” in the title field</td>
</tr>
<tr>
<td>(agile OR extreme) AND methodology</td>
<td>Contain <code>methodology</code> and must also contain <code>agile</code> and/or <code>extreme</code>, all in the default field</td>
</tr>
<tr>
<td>title:“junit in action”</td>
<td>Contain the exact phrase “<code>junit in action</code>” in the title field</td>
</tr>
<tr>
<td>title:“junit action”~5</td>
<td>Contain the terms <code>junit</code> and <code>action</code> within five positions of one another</td>
</tr>
<tr>
<td>java*</td>
<td>Contain terms that begin with <code>java</code>, like <code>javaspace</code>, <code>javaserver</code>, and <code>java.net</code></td>
</tr>
<tr>
<td>java~</td>
<td>Contain terms that are close to the word <code>java</code>, such as <code>java</code></td>
</tr>
<tr>
<td>lastmodified: [1/1/04 TO 12/31/04]</td>
<td>Have lastmodified field values between the dates January 1, 2004 and December 31, 2004</td>
</tr>
</tbody>
</table>
Conclusions

- Nutch as a starting point
- Crawling in Nutch
- Detailed map-reduce architecture
- Different query formats in Lucene
- Built-in analyzers in Lucene
- Same analyzer need to be used both while indexing and searching
Thanks

- Questions ??