CSE 454 - Case Studies

Indexing & Retrieval in Google

Logistics

- **For next class**
  - Read: How to implement PageRank Efficiently

- **Projects due**
  - Before class

- **Project part II**
  - Alternatives…

BigTable

- **Customers**
- **Pros**
- **Cons**

Review: AltaVista

- A document is a region of location space
  - Contiguous
  - No overlap
  - Densely allocated (first doc is location 1)

- All document structure encoded with words
  - enddoc at last location of document
  - begintitle, endtitle mark document title

```
| Document 1 | Document 2 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Index Stream Readers (ISRs)

- **Interface for**
  - Reading result of query
  - Return ascending sequence of locations
  - Implemented using lazy evaluation

- **Methods**
  - loc(ISR) return current location
  - next(ISR) advance to next location
  - seek(ISR, X) advance to next loc after X
  - prev(ISR) return previous location

Advanced Search

- Field query: a in Title of page
- Let BT, ET be ISRP of words begintitle, endtitle
- Constraints:
  - loc(BT) ≤ loc(A)
  - loc(A) ≤ loc(ET)
  - prev(ET) ≤ loc(BT)
### Solver Algorithm

```plaintext
while (unsatisfied_constraints)
    satisfy_constraint(choose_unsat_constraint())
```

- **To satisfy:** \( \text{loc}(A) \leq \text{loc}(B) + K \)
  - Execute: seek\(B\), \(\text{loc}(A) - K\)
- **To satisfy:** \( \text{prev}(A) \leq \text{loc}(B) + K \)
  - Execute: seek\(B\), \(\text{prev}(A) - K\)
- **To satisfy:** \( \text{loc}(A) \leq \text{prev}(B) + K \)
  - Execute: seek\(B\), \(\text{loc}(A) - K\),
    - next\(B\)
- **To satisfy:** \( \text{prev}(A) \leq \text{prev}(B) + K \)
  - Execute: seek\(B\), \(\text{prev}(A) - K\)
    - next\(B\)

Heuristic: Which choice advances a stream the furthest?

---

### Course Overview

- **Info Extraction**
- **Ecommerce**
- **Datamining**
- **P2P**
- **Security**
- **Web Services**
- **Semantic Web**
- **Advt**
- **New Stuff**

#### Case Studies
- Nutch
- Google
- Alavista

- **Information Retrieval**
- **Precision vs Recall**
- **Inverted Indices**
- **Crawler Architecture**
- **Cluster Computing**

---

### Major Data Structures

- **Google File System**
- **Big Files**
  - virtual files spanning multiple file systems
  - addressable by 64 bit integers
  - handles allocation & deallocation of File Descriptions
    since the OS’s is not enough
  - supports rudimentary compression

- **More on this Later…**

---

### Google System Anatomy

- **Compresses Documents**
- **Create partially sorted forward index** (i.e. for a range of words)
- **Anchor text added as well**
- **Each barrel holds part of the forward index**
- **Sorts by wordID to creates inverted index**

---

### Major Data Structures (2)

#### Repository
- **Full HTML of every page**
- **Docs stored one after another**
  - Prefix: docID, length, URL
- **Compressed: Tradeoff between**
  - Speed
  - Compression ratio
- **Choose zlib (3 to 1)**
  - Rather than bzip (4 to 1)
- **Requires no other data structure to access it**
  - Robustness
  - Ease of dev
  - Can rebuild other structures

---

### Major Data Structures (3)

#### Document Index
- **Info about each document**
  - Status
  - Pointer to repository
  - Document checksum + statistics
    - If crawled,
      - pointer to var width file with URL, title
    - Else
      - Pointer to URL-list
- **Fixed width ISAM (index sequential access mode) index**
  - Ordered by docID
- **Compact data structure**
  - Can fetch a record in 1 disk seek during search
Major Data Structures (4)

URL's - docID file

- List of URL checksums with their docIDs
  - Sorted by checksums
- Used to convert URLs to docIDs
  - Given a URL a binary search is performed
- Additions done with batch merge

skip

Major Data Structures (5)

Lexicon

- Can fit in memory
  - currently 256 MB
  - contains 14 million words
- 2 parts
  - a list of words
    - Separated by nulls
  - a hash table of pointers
    - Into hit list (occurrences)

Major Data Structures (6)

Hit Lists (Occurrences)

- Includes position, font & capitalization
- Bulk of index size
- Tried 3 alternatives
  - Simple: triple of ints – too big
  - Huffman: too slow
  - Hand-optimized ☺
    - 2 bytes / hit

Major Data Structures (7)

Hit Lists Continued

- 64 Barrels
- Holds range

Crawling the Web

- Fast distributed crawling system
- URLserver & Crawlers are implemented in python
- Each Crawler keeps about 300 connections open
- Peak rate = 100 pages, 600K per second
- Cache DNS lookup internally
  - synchronized IO to handle events
  - number of queues
- Robust & Carefully tested
Parsing

• **Must handle errors**
  – HTML typos
  – KB of zeros in a middle of a TAG
  – Non-ASCII characters
  – HTML Tags nested hundreds deep
• **Developed their own Parser**
  – involved a fair amount of work
  – did not cause a bottleneck

Searching

• **Algorithm**
  – 1. Parse the query
  – 2. Convert word into wordIDs
  – 3. Seek to the start of the doclist in the short barrel for every word
  – 4. Scan through the doclists until there is a document that matches all of the search terms
  – 5. Compute the rank of that document
  – 6. If we’re at the end of the short barrels start at the doclists of the full barrel, unless we have enough
  – 7. If were not at the end of any doclist goto step 4
  – 8. Sort the documents by rank return the top K
    • (May jump here after 40k pages)

The Ranking System

• **The information**
  – Position, Font Size, Capitalization
  – Anchor Text
  – PageRank
• **Hits Types**
  – title ,anchor , URL etc..
  – small font, large font etc..

The Ranking System (2)

• **Each Hit type has it’s own weight**
  – Count weights increase linearly with counts at first but quickly taper off - this is the IR score of the doc
  – (IDF weighting??)
• **IR combined w/ PageRank to give the final Rank**
• **For multi-word query**
  – A proximity score for every set of hits with a proximity type weight
  – 10 grades of proximity

Storage Requirements

• **Using Compression on the repository**
  – About 55 GB for all the data used by the SE
• **Most of the queries can be answered by just the short inverted index**
• **“With better compression, a high quality SE can fit onto a 7GB drive of a new PC”**

Storage Statistics

<table>
<thead>
<tr>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size of Fetched Pages</td>
<td>147.8 GB</td>
</tr>
<tr>
<td>Compressed Repository</td>
<td>53.5 GB</td>
</tr>
<tr>
<td>Short Inverted Index</td>
<td>4.1 GB</td>
</tr>
<tr>
<td>Index File</td>
<td>6.6 GB</td>
</tr>
<tr>
<td>Anchor Data</td>
<td>9.7 GB</td>
</tr>
<tr>
<td>Temporary Document Index File</td>
<td>3.9 GB</td>
</tr>
<tr>
<td>Total Without Repository</td>
<td>55.2 GB</td>
</tr>
</tbody>
</table>

Web Page Statistics

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Web Pages Fetched</td>
<td>24 million</td>
</tr>
<tr>
<td>Number of URLs Seen</td>
<td>76.5 million</td>
</tr>
<tr>
<td>Number of Email Addresses</td>
<td>1.7 million</td>
</tr>
<tr>
<td>Number of 404’s</td>
<td>1.6 million</td>
</tr>
</tbody>
</table>

8 B pages in 2005
System Performance

- It took 9 days to download 26 million pages
- 48.5 pages per second
- The Indexer & Crawler ran simultaneously
- The Indexer runs at 54 pages per second
- The sorters run in parallel using 4 machines, the whole process took 24 hours

Link Spam

- Keyword stuffing
- Meta tag stuffing
- Multiple titles
- Tiny fonts
- Invisible text
  - <body bgcolor="FFFFFF">
  - <font color="#FFFFFF" size ="1">Your text here</font>
  - Problem: takes up space. Size=1? Bottom?
- Doorway / jump pages
  - Fast meta refresh
- Cloaking - Code swapping
- Domain spamming
- Pagerank spoofing