EECS 395/495 Lecture 3
Scalable Indexing, Searching, and Crawling

Doug Downey
Based partially on slides by Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze
Announcements

• Project proposals due tonight
• Sign up for project meetings
Last time: Inverted index

Brutus
Calpurnia
Caesar

Dictionary

Posting

Postings lists

2 → 4 → 8 → 16 → 32 → 64 → 128
1 → 2 → 3 → 5 → 8 → 13 → 21 → 34
13 → 16
How Does Web Search Work?

• Inverted Indices
  – Inverted Indices enable fast IR
  – Tokenization & Linguistic issues
  – Handling phrase queries
    – *Scalable, distributed construction & deployment*

• Web Crawlers

• Ranking Algorithms
Today’s Outline

• Index compression
• Distributed Indexing
  – MapReduce
• Distributed Searching
Index Compression

Dictionary

Brutus
Calpurnia
Caesar

Posting

Postings lists

Sec. 1.2
Postings compression

• The postings file is much larger than the dictionary, factor of at least 10.

• **Key goal:** store each posting compactly.
  – A posting for our purposes is a docID.
  – Naïve: use $\log_2 20,000,000,000 \approx 34$ bits per docID.

• Our goal: use a lot less than 34 bits per docID.
Postings: two conflicting forces

• Common terms (e.g. *the*)
  – 34 bits/posting is too expensive
  – Prefer 0/1 document incidence vector in this case
    • If *the* appears in half the docs => savings of 17x!

• Rare terms (e.g. *arachnocentric*)
  – Say it occurs once every million docs, on average
  – 0/1 document incidence vector is a disaster
    • 34 bits/occurrence uses 29,000x less space!
Postings file entry

• We store the list of docs containing a term in increasing order of docID.
  – *computer*: 33,47,154,159,202 ...
• Consequence: it suffices to store gaps.
  – 33,14,107,5,43 ...
• Hope: most gaps can be encoded/stored with far fewer than 34 bits.
Example: Three postings entries

<table>
<thead>
<tr>
<th>encoding</th>
<th>postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td>docIDs</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>283042</td>
</tr>
<tr>
<td></td>
<td>283043</td>
</tr>
<tr>
<td></td>
<td>283044</td>
</tr>
<tr>
<td></td>
<td>283045</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>docIDs</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>283047</td>
</tr>
<tr>
<td></td>
<td>283154</td>
</tr>
<tr>
<td></td>
<td>283159</td>
</tr>
<tr>
<td></td>
<td>283202</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
<tr>
<td></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td>docIDs</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Preserves linear-time merge
– just keep track of sum as you traverse postings
Variable length encoding

• Aim:
  – For *arachnocentric*, use ~20 bits per gap entry
  – For *the*, use ~1 bit per gap entry
  – More generally:
    • to encode a gap with integral value $G$, use $\sim \log_2 G$ bits

• Standard, fixed *n*-bit *int* won’t work here
  – Solution: *Variable length codes*
Variable Byte (VB) codes

• For a gap value $G$
  – Begin with one byte to store $G$ and dedicate 1 bit in it to be a continuation bit $c$
  – If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
  – Else
    • Encode $G$’s lowest-order 7 bits, put in rightmost byte
    • Encode $G$’s higher-order bits in additional bytes to left
    • When finished, set continuation bits: last byte $c = 1$; other bytes $c = 0$
Example

<table>
<thead>
<tr>
<th>docIDs</th>
<th>824</th>
<th>829</th>
<th>215406</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaps</td>
<td></td>
<td>5</td>
<td>214577</td>
</tr>
<tr>
<td>VB code</td>
<td>00000110 10111000</td>
<td>10000101</td>
<td>00001101 00001100 10110001</td>
</tr>
</tbody>
</table>

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.
Benefits from index compression

• Shrink index to ~15% the size of original text!
• ...but we ignored positional information
  – In practice savings are more limited
    (about 30%-50% of original text)
  – Techniques are substantially the same
Today’s Outline

• Index compression

• **Distributed Indexing**
  – **MapReduce**

• Distributed Searching
Indexing at Web-scale....

• Sort by terms.

Core indexing step.

And we have to sort postings too...how do we do this with 20 billion documents?
Distributed indexing

• For web-scale indexing:
  use a distributed computing cluster

• Individual machines are fault-prone
  – Can unpredictably slow down or fail

• How do we exploit such a pool of machines?
**Data centers**

- Mainly commodity machines
- Distributed around the world
- Estimates for Google:
  - At least 1 million servers, est. 2% of worldwide supply (datacenterknowledge.com, 2010)
  - Approximately 200,000 new servers each quarter, based on expenditures of $1.6B/year


- “According to Microsoft research chief Rick Rashid, around 20 percent of all the servers sold around the world each year are now being bought by a small handful of internet companies - he named Microsoft, Google, Yahoo and Amazon.”
Google data centers

• If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?

• Answer: 63%
Distributed indexing

- Maintain a *master* machine directing the indexing job – considered “safe”.
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.
Parallel tasks

• We will use two sets of parallel tasks
  – Parsers
  – Inverters
• Break the input document corpus into *splits*
• Each split is a subset of documents
Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into $j$ partitions
- Each partition is for a range of terms’ first letters
  - (e.g., a-f, g-p, q-z) – here $j=3$. 
Inverters

• An inverter collects all (term,doc) pairs (= postings) for one term-partition.
• Sorts and writes to postings lists
• Important: Each partition is small enough such that this sort can be done on a single machine – so in practice we’d use large $j$ (many partitions) e.g., al-aq rather than a-j
Data flow

splits

assign
Master
assign

Parser -> a-f, g-p, q-z
Parser -> a-f, g-p, q-z
Parser -> a-f, g-p, q-z

Inverter -> a-f
Inverter -> g-p
Inverter -> q-z

Postings

Segment files
MapReduce

• The index construction algorithm we just described is an instance of **MapReduce**
  – a robust and conceptually simple *framework* for distributed computing [Dean and Ghemawat 2004]
  – Handles the “distributed” part

• The Google indexing system (ca. 2002) consists of a number of phases, each until recently implemented in MapReduce.
Schema for index construction in MapReduce

- **Schema of map and reduce functions**
  - **map**: input $\rightarrow$ list($<k, v>$)  
    **reduce**: ($k$, list($v$)) $\rightarrow$ output

- **Instantiation of the schema for index construction**
  - **map**: web collection $\rightarrow$ list(termID, docID)
  - **reduce**: ($<\text{termID1}, \text{list(docID)>}$, $<\text{termID2}, \text{list(docID)>}$, ...) $\rightarrow$ (postings list1, postings list2, ...)
MapReduce Example for Index Construction

• **map:**

\[ \begin{align*}
    & d2: \text{C died. d1: C came, C c’ed.} \\
    & \rightarrow \\
    & \langle \text{C},d2\rangle, \langle \text{died},d2\rangle, \langle \text{C},d1\rangle, \langle \text{came},d1\rangle, \langle \text{C},d1\rangle, \langle \text{c’ed},d1\rangle \\
\end{align*} \]

• **reduce:**

\[ \begin{align*}
    & (\langle \text{C},(d2,d1,d1)\rangle, \langle \text{c’ed},(d1)\rangle, \langle \text{came},(d1)\rangle, \langle \text{died},(d2)\rangle) \\
    & \rightarrow \\
    & \langle \text{C},(d1:2,d2:1)\rangle, \langle \text{c’ed},(d1:1)\rangle, \langle \text{came},(d1:1)\rangle, \\
    & \langle \text{died},(d2:1)\rangle, \\
\end{align*} \]

In key-sorted order.
Gee, it’s easy

• You write two methods:
  – map:
    takes a data object and emits (key, value) pairs
  – reduce:
    takes (key, value) pairs for one key and merges them

• You specify:
  – A partition function for reduce (which keys go where)
  – Parameters $M$ (number of map tasks) and $R$ (number of reduce tasks – same as $j$)
Data flow

Map phase

Reduce phase

Parser

Master

Parser

Parser

Parser

assign

assign

Segment files

Postings
MapReduce Code Example

• Word Counting
  – In a set of a million documents, how often does each word appear?
    • **Map**: for a given doc, emit \(<\text{word}, 1>\) for each \text{word} that appears
    • **Reduce**: Output sum
// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;

            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
            if (start < i)
                Emit(text.substr(start, i-start), "1");
        }
    }
};
REGISTER_MAPPER(WordCounter);
// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);
Fun with MapReduce (1 of 4)

• Distributed grep
  – In a set of files, which lines include the word *arachnocentric*?
    • **Map**: scan text and output matching lines
    • **Reduce**: Copy the input
Fun with MapReduce (2 of 4)

• Monte Carlo simulation
  – E.g., Given computer players of backgammon A and B, how often will A beat B?
    • **Map**: play a game and emit \(<A, 1>\) or \(<B, 1>\) based on who wins
    • **Reduce**: Sum
  – post-processing: compute the ratio
Fun with MapReduce (3 of 4)

• Boolean Satisfiability
  – Given a boolean formula in 3-CNF, e.g.:
    \((x_1 \lor \neg x_3 \lor x_7) \land (x_4 \lor x_5 \lor \neg x_6) \land \ldots\)
  How many distinct assignments to variables (i.e. \(x_i = \text{true} | \text{false}\)) make the formula true?
  – Create \(M = 2^k\) splits, one for each assignment
    • Map: Output \(<\text{whatever}, 1>\> \text{iff the assignment makes the formula true}
    • Reduce: Sum
Fun with MapReduce (4 of 4)

• Link Graph Index
  – Build an index to answer “which pages link to me” queries (i.e. Google’s \textit{link:} operator)
    • \textbf{Map}: For document \textit{d1}, output $<dj, d1>$ for all $<a \text{href} =dj>$ tags therein
    • \textbf{Reduce}: Sort by values into posting lists
  – Partition function (which \textbf{map} keys are grouped together)
    • Sorted, e.g. ranges of domains (\texttt{www.aa*-www.ai*})
Today’s Outline

• Index compression
• Distributed Indexing
  – MapReduce
• Distributed Searching
Distributed Searching

• We described indexing into a term-partitioned structure
  – Each machine handles some fraction of terms (e.g. al-aq)

• Another possibility:
  – Document-partitioned: one machine handles a subset of documents
Back to MapReduce

• How do we change our implementation to be document-partitioned?

• One possibility:
  – **map**: output \(<\text{doc-group} \mid \text{word-id}, \text{docID}>\)
  – **reduce**: sum and output posting list as before
  – **partition function**: by doc-group
Next

• Web Crawlers, Link Analysis, Information Extraction