Lecture 3 – Index Construction and Compression

Many thanks to Prabhakar Raghavan for sharing most content from the following slides
Recap of the previous lecture

- Tokenization
- Term equivalence
- Skip pointers
- Bi-word indexes for phrases
- Positional indexes for phrases/proximity queries
- Dictionary data structures
- Wild card queries
- Spell correction
This lecture

- Index construction
  - Doing sorting with limited main memory
  - Parallel and distributed indexing
- Index compression
  - Space estimation
  - Dictionary compression
  - Postings compression
Index construction
Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?
Many design decisions in information retrieval are based on the characteristics of hardware.

We begin by reviewing hardware basics.
Hardware basics

- Access to data in memory is **much** faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.
Hardware basics

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It’s much cheaper to use many regular machines rather than one fault tolerant machine.
## Hardware assumptions

<table>
<thead>
<tr>
<th>symbol</th>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>average seek time</td>
<td>$5 \text{ ms} = 5 \times 10^{-3} \text{ s}$</td>
</tr>
<tr>
<td>b</td>
<td>transfer time per byte</td>
<td>$0.02 \mu\text{s} = 2 \times 10^{-8} \text{ s}$</td>
</tr>
<tr>
<td></td>
<td>processor’s clock rate</td>
<td>$10^9 \text{ s}^{-1}$</td>
</tr>
<tr>
<td>p</td>
<td>low-level operation (e.g., compare &amp; swap a word)</td>
<td>$0.01 \mu\text{s} = 10^{-8} \text{ s}$</td>
</tr>
<tr>
<td></td>
<td>size of main memory</td>
<td>several GB</td>
</tr>
<tr>
<td></td>
<td>size of disk space</td>
<td>1 TB or more</td>
</tr>
</tbody>
</table>
RCV1: Our collection for this lecture

- Shakespeare’s collected works definitely aren’t large enough for demonstrating many of the points in this course.
- The collection we’ll use isn’t really large enough either, but it’s publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
  - [http://trec.nist.gov/data/reuters/reuters.html](http://trec.nist.gov/data/reuters/reuters.html)
- This is one year of Reuters newswire (part of 1995 and 1996)
Extreme conditions create rare Antarctic clouds

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.
# Reuters RCV1 statistics

<table>
<thead>
<tr>
<th>symbol</th>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>documents</td>
<td>800,000</td>
</tr>
<tr>
<td>L</td>
<td>avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>M</td>
<td>terms (= word types)</td>
<td>400,000</td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per token</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>(incl. spaces/punctuation)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per token</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>(without spaces/punctuation)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per term</td>
<td>7.5</td>
</tr>
<tr>
<td>T</td>
<td>non-positional postings</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>

4.5 bytes per word token vs. 7.5 bytes per word type: why?
Documents are parsed to extract words and these are saved with the Document ID.

Doc 1
I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious
Key step

- After all documents have been parsed, the inverted file is sorted by terms.

We focus on this sort step. We have 100M items to sort.
Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.
Sort-based index construction

- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks \((you \ can, \ but \ much \ more \ complex)\)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry \((\text{term}, \ \text{doc}, \ \text{freq})\), demands a lot of space for large collections.
- \(T = 100,000,000\) in the case of RCV1
  - So … we can do this in memory, but typical collections are much larger. E.g. the \textit{New York Times} provides an index of \(>150\) years of newswire
- Thus: We need to store intermediate results on disk.
Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?

No: Sorting $T = 100,000,000$ records on disk is too slow – too many disk seeks.

We need an external sorting algorithm.
Bottleneck

- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort $T=100M$ records

If every comparison took 2 disk seeks, and $N$ items could be sorted with $N \log_2 N$ comparisons, how long would this take?
BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

- 12-byte (4+4+4) records \((term, doc, freq)\).
- These are generated as we parse docs.
- Must now sort 100M such 12-byte records by \(term\).
- Define a Block ~ 10M such records
  - Can easily fit a couple into memory.
  - Will have 10 such blocks to start with.

Basic idea of algorithm:
- Accumulate postings for each block, sort, write to disk.
- Then merge the blocks into one long sorted order.
postings to be merged

brutus d3
caesar d4
noble d3
with d4

brutus d2
cæsar d1
julius d1
killed d2

merged postings

brutus d2
brutus d3
cæsar d1
cæsar d4
julius d1
killed d2
noble d3
with d4

disk
First, read each block and sort within:
- Quicksort takes $2N \ln N$ expected steps
- In our case $2 \times (10M \ln 10M)$ steps

**Exercise:** estimate total time to read each block from disk and quicksort it.

10 times this estimate – gives us 10 sorted *runs* of 10M records each.

Need 2 copies of data on disk
- But can optimize this
n = 0
while (all documents have not been processed)
do n = n + 1
    block = parseNextBlock();  // read postings until the block is full
    BSBI-Invert(block);  // sort all postings
    // collect all postings with same
termID into a list
    // results in a small inverted index

    writeBlockToDisk(block, fn);
mergeBlocks(f_1, f_2, ..., f_n; f_merged);
How to merge the sorted runs?

- Can do binary merges, with a merge tree of $\log_2 10 = 4$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.
Merge tree

Sorted runs.

1 run, 100M/run
1 run, 80M/run
2 runs, 40M/run
5 runs, 20M/run

Bottom level of tree.
How to merge the sorted runs?

- But it is more efficient to do a $n$-way merge, where you are reading from all blocks simultaneously.
- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you’re not killed by disk seeks.
Remaining problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)
SPIMI: Single-pass in-memory indexing

- **Key idea 1:** Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- **Key idea 2:** Don’t sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.
SPIMI-Invert

SPIMI-Invert\textit{(token\_stream)}
1 \hspace{1em} output\_file = NewFile()
2 \hspace{1em} dictionary = NewHash()
3 \hspace{1em} \textbf{while} (free memory available)
4 \hspace{2em} \textbf{do} token \leftarrow next(token\_stream)
5 \hspace{3em} \textbf{if} term(token) \notin dictionary
6 \hspace{4em} \textbf{then} postings\_list = AddToDictionary\textit{(dictionary, term(token))}
7 \hspace{4em} \textbf{else} postings\_list = GetPostingsList\textit{(dictionary, term(token))}
8 \hspace{3em} \textbf{if} full(postings\_list)
9 \hspace{4em} \textbf{then} postings\_list = DoublePostingsList\textit{(dictionary, term(token))}
10 \hspace{1em} AddToPostingsList\textit{(postings\_list, docID(token))}
11 \hspace{1em} sorted\_terms \leftarrow SortTerms\textit{(dictionary)}
12 \hspace{1em} WriteBlockToDisk\textit{(sorted\_terms, dictionary, output\_file)}
13 \hspace{1em} \textbf{return} output\_file

- Merging of blocks is analogous to BSBI.
SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
Distributed indexing

- For web-scale indexing: must 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?
Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world (in 2007)!??
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\%
- Calculate the number of servers failing per minute for an installation of 1 million servers.
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 collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)
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 $j = 3$.
- Now to complete the index inversion
Inverters

- An inverter collects all (term, doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists
Data flow

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

assign
assign

Master

Postings

Map phase
Segment files
Reduce phase

splits
The index construction algorithm we just described is an instance of MapReduce.

MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing …

… without having to write code for the distribution part.

They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.
Index construction was just one phase.

Another phase: transforming a term-partitioned index into a document-partitioned index.

- Term-partitioned: one machine handles a subrange of terms
- Document-partitioned: one machine handles a subrange of documents

As we discuss in the web part of the course, most search engines use a document-partitioned index ... better load balancing, etc.
Schema for index construction in MapReduce

**Schema of map and reduce functions**
- map: input → list(k, v)
- reduce: (k, list(v)) → output

**Instantiation of the schema for index construction**
- map: web collection → list (termID, docID)
- reduce: <termID1, list(docID)>, <termID2, list(docID)>, … → (postings list1, postings list2, …)

**Example for index construction**
- map:
  - d2: “Caesar died”. d1: “Caesar came, Caesar conquered”.
  - <Caesar, d2>, <died, d2>, <Caesar, d1>, <came, d1>, <Caesar, d1>, <conquered, d1>
- reduce:
  - <Caesar, d2>, <died, d2>, <Caesar, d1>, <came, d1>, <Caesar, d1>, <conquered, d1>
  - → (<Caesar, (d1:2, d2:1)>, <died, (d2:1)>, <came, (d1:1)>, <conquered, (d1:1)>)
MapReduce

- The infrastructure used today for this kind of task
  - Map: Parsing
  - Reduce: Inverting

- More on MapReduce theory here:
  - http://en.wikipedia.org/wiki/MapReduce

- A good MapReduce implementation is offered by Apache: http://hadoop.apache.org/mapreduce/
More on real life

- Same machine can do map, reduce, as well as other tasks in the same time (e.g., crawling the Web)
- Focus on minimizing I/O (network & disk)
Dynamic indexing

- Up to now, we have assumed that collections are static.

- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.

- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary
Simplest approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index
Issues with main and auxiliary indexes

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)
Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest ($Z_0$) in memory
- Larger ones ($I_0, I_1, \ldots$) on disk
- If $Z_0$ gets too big ($> n$), write to disk as $I_0$
  - or merge with $I_0$ (if $I_0$ already exists) as $Z_1$
- Either write merge $Z_1$ to disk as $I_1$ (if no $I_1$)
  - Or merge with $I_1$ to form $Z_2$
- etc.
LMERGEADDTOKEN(indexes, Z₀, token)

1. \( Z₀ \leftarrow \text{MERGE}(Z₀, \{\text{token}\}) \)
2. \text{if } |Z₀| = n
3. \text{then for } i \leftarrow 0 \text{ to } \infty
4. \text{do if } lᵢ \in \text{indexes}
5. \text{then } Zᵢ₊₁ \leftarrow \text{MERGE}(lᵢ, Zᵢ)
6. (\( Zᵢ₊₁ \) is a temporary index on disk.)
7. \text{indexes} \leftarrow \text{indexes} \setminus \{lᵢ\}
8. \text{else } lᵢ \leftarrow Zᵢ (Zᵢ \text{ becomes the permanent index } lᵢ.)
9. \text{indexes} \leftarrow \text{indexes} \cup \{lᵢ\}
10. \text{Break}
11. \( Z₀ \leftarrow \emptyset \)

LOGARITHMICMERGE()

1. \( Z₀ \leftarrow \emptyset \) (\( Z₀ \) is the in-memory index.)
2. \text{indexes} \leftarrow \emptyset
3. \text{while } \text{true}
4. \text{do LMERGEADDTOKEN(indexes, Z₀, getNextToken())}
Logarithmic merge

- Auxiliary and main index: index construction time is $O(T^2)$ as each posting is touched in each merge.
- Logarithmic merge: Each posting is merged $O(\log T)$ times, so complexity is $O(T \log T)$
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of $O(\log T)$ indexes
  - Whereas it is $O(1)$ if you just have a main and auxiliary index
Further issues with multiple indexes

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking
Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted
Google Dance Is Back? Plus Google’s First Live Chat Recap & Hyperactive Yahoo Slurp

Is the Google Dance back? Well, not really, but I am noticing Google Dance–like behavior from Google based on reading some of the feedback at a WebmasterWorld thread.

The Google Dance refers to how years ago, a change to Google’s ranking algorithm often began showing up slowly across data centers as they reflected different results, a sign of coming changes. These days Google’s data centers are typically always showing small changes and differences, but the differences between this data center and this one seem to be more like the extremes of the past Google Dances.

So either Google is preparing for a massive update or just messing around with our heads. As of now, these results have not yet moved over to the main Google.com results.
Other sorts of indexes

- Positional indexes
  - Same sort of sorting problem … just larger
- Building character n-gram indexes:
  - As text is parsed, enumerate $n$-grams.
  - For each $n$-gram, need pointers to all dictionary terms containing it – the “postings”.
  - Note that the same “postings entry” will arise repeatedly in parsing the docs – need efficient hashing to keep track of this.
    - E.g., that the trigram `you` occurs in the term `deciduous` will be discovered on each text occurrence of `deciduous`
    - Only need to process each term once
Building $n$-gram indexes

- Once all $(n$-gram $\in$ term) pairs have been enumerated, must sort for inversion
- For an average dictionary term of 8 characters
  - We have about 6 trigrams per term on average
  - For a vocabulary of 500K terms, this is about 3 million pointers – can compress
Index on disk vs. memory

- Most retrieval systems keep the dictionary in memory and the postings on disk
- Web search engines frequently keep both in memory
  - massive memory requirement
  - feasible for large web service installations
  - less so for commercial usage where query loads are lighter
Indexing in the real world

- Typically, don’t have all documents sitting on a local filesystem
  - Documents need to be *spidered*
  - Could be dispersed over a WAN with varying connectivity
  - Must schedule distributed spiders
  - Have already discussed distributed indexers
- Could be (secure content) in
  - Databases
  - Content management applications
  - Email applications
Content residing in applications

- Mail systems/groupware, content management contain the most “valuable” documents
- HTTP often not the most efficient way of fetching these documents - native API fetching
  - Specialized, repository-specific connectors
  - These connectors also facilitate *document viewing* when a search result is selected for viewing
Secure documents

- Each document is accessible to a subset of users
  - Usually implemented through some form of Access Control Lists (ACLs)
- Search users are authenticated
- Query should retrieve a document only if user can access it
  - So if there are docs matching your search but you’re not privy to them, “Sorry no results found”
  - E.g., as a lowly employee in the company, I get “No results” for the query “salary roster”
Users in groups, docs from groups

- Index the ACLs and filter results by them

- Often, user membership in an ACL group verified at query time – slowdown

Users

Documents

0/1

0 if user can’t read doc, 1 otherwise.
Exercise

- Can spelling suggestion compromise such document-level security?
- Consider the case when there are documents matching my query, but I lack access to them.
Compound documents

- What if a doc consisted of *components*
  - Each component has its own ACL.
- Your search should get a doc only if your query meets one of its components that you have access to.
- More generally: doc assembled from *computations* on components
  - e.g., in content management systems
- How do you index such docs?

No good answers …
“Rich” documents

- (How) Do we index images?
- Researchers have devised Query Based on Image Content (QBIC) systems
  - “show me a picture similar to this orange circle”
  - watch for lecture on vector space retrieval
- In practice, image search usually based on metadata such as file name e.g., monalisa.jpg
- New approaches exploit social tagging
  - E.g., flickr.com
Passage/sentence retrieval

- Suppose we want to retrieve not an entire document matching a query, but only a passage/sentence - say, in a very long document
- Can index passages/sentences as mini-documents – what should the index units be?
- This is the subject of XML search
Index compression
Why compression (in general)?

- Use less disk space
  - Saves a little money
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use
Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too

- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory

- We will devise various IR-specific compression schemes
Reuters RCV1 statistics (copy)

- symbol: statistic: value
  - N: documents: 800,000
  - L: avg. # tokens per doc: 200
  - M: terms (= word types): 400,000
  - avg. # bytes per token: 6
    (incl. spaces/punctuation)
  - avg. # bytes per token: 4.5
    (without spaces/punctuation)
  - avg. # bytes per term: 7.5
  - non-positional postings: 100,000,000
### Index parameters vs. what we index

<table>
<thead>
<tr>
<th>size of</th>
<th>word types (terms)</th>
<th>non-positional postings</th>
<th>positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dictionary</td>
<td>non-positional index</td>
<td>positional index</td>
</tr>
<tr>
<td>Size (K)</td>
<td>∆%</td>
<td>cumul %</td>
<td>Size (K)</td>
</tr>
<tr>
<td>Unfiltered</td>
<td>484</td>
<td></td>
<td>109,971</td>
</tr>
<tr>
<td>No numbers</td>
<td>474</td>
<td>-2</td>
<td>100,680</td>
</tr>
<tr>
<td>Case folding</td>
<td>392</td>
<td>-17</td>
<td>96,969</td>
</tr>
<tr>
<td>30 stopwords</td>
<td>391</td>
<td>-0</td>
<td>83,390</td>
</tr>
<tr>
<td>150 stopwords</td>
<td>391</td>
<td>-0</td>
<td>67,002</td>
</tr>
<tr>
<td>stemming</td>
<td>322</td>
<td>-17</td>
<td>63,812</td>
</tr>
</tbody>
</table>

Exercise: give intuitions for all the ‘0’ entries. Why do some zero entries correspond to big deltas in other columns?
Lossless vs. lossy compression

- **Lossless compression**: All information is preserved.
  - What we mostly do in IR.
- **Lossy compression**: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- **Optimization**: Prune postings entries that are unlikely to turn up in the top $k$ list for any query.
  - Almost no loss quality for top $k$ list.
Vocabulary vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode 😊
Vocabulary vs. collection size

- Heaps’ law: \( M = kT^b \)
- \( M \) is the size of the vocabulary, \( T \) is the number of tokens in the collection
- Typical values: \( 30 \leq k \leq 100 \) and \( b \approx 0.5 \)
- In a log-log plot of vocabulary size \( M \) vs. \( T \), Heaps’ law predicts a line with slope about \( \frac{1}{2} \)
  - It is the simplest possible relationship between the two in log-log space
  - An empirical finding (“empirical law”)
Heaps’ Law

For RCV1, the dashed line

$$\log_{10} M = 0.49 \log_{10} T + 1.64$$

is the best least squares fit.

Thus, $$M = 10^{1.64} T^{0.49}$$ so $$k = 10^{1.64} \approx 44$$ and $$b = 0.49$$.

Good empirical fit for Reuters RCV1!

For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms
Exercises

- What is the effect of including spelling errors, vs. automatically correcting spelling errors on Heaps’ law?

- Compute the vocabulary size $M$ for this scenario:
  - Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
  - Assume a search engine indexes a total of $20,000,000,000$ ($2 \times 10^{10}$) pages, containing 200 tokens on average.
  - What is the size of the vocabulary of the indexed collection as predicted by Heaps’ law?
Zipf’s law

- Heaps’ law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf’s law: The $i$th most frequent term has frequency proportional to $1/i$.
- $\text{cf}_i \propto 1/i = K/i$ where $K$ is a normalizing constant.
- $\text{cf}_i$ is collection frequency: the number of occurrences of the term $t_i$ in the collection.
If the most frequent term (the) occurs $c_1$ times
- then the second most frequent term (of) occurs $c_1/2$ times
- the third most frequent term (and) occurs $c_1/3$ times …

Equivalent: $c_f = K/i$ where $K$ is a normalizing factor, so
- $\log c_f = \log K - \log i$
- Linear relationship between $\log c_f$ and $\log i$

Another power law relationship
Zipf’s law for Reuters RCV1
Compression

- Now, we will consider compressing the space for the dictionary and postings
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes
DICTIONARY COMPRESSION
Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn’t in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important
Dictionary storage - first cut

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq.</th>
<th>Postings ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td></td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td></td>
</tr>
</tbody>
</table>

Dictionary search structure

20 bytes

4 bytes each
Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can’t handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.
- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn’t this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.
Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.

<table>
<thead>
<tr>
<th>Freq.</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total string length = $400K \times 8B = 3.2MB$

Pointers resolve $3.2M$ positions: $\log_23.2M = 22\text{bits} = 3\text{bytes}$
Space for dictionary as a string

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

Now avg. 11 bytes/term, not 20.
Blocking

- Store pointers to every $k$th term string.
  - Example below: $k=4$.
- Need to store term lengths (1 extra byte)

\[
\begin{array}{cccc}
\text{Freq.} & \text{Postings ptr.} & \text{Term ptr.} \\
33 & & \\
29 & & \\
44 & & \\
126 & & \\
7 & & \\
\end{array}
\]

\[\text{Save 9 bytes on 3 pointers.}\]
\[\text{Lose 4 bytes on term lengths.}\]

\[\ldots7\text{systile}9\text{syzygetic}8\text{syzygial}6\text{syzygy}11\text{szaibelyite}8\text{syczecin}9\text{szomo}\ldots\]
Net

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - $3 \times 4 = 12$ bytes,
  
  now we use $3 + 4 = 7$ bytes.

Shaved another $\sim 0.5$MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

We can save more with larger $k$.

Why not go with larger $k$?
Exercise

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 4, 8$ and $16$. 
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons $= (1+2\cdot2+4\cdot3+4)/8 \approx 2.6$

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?
Dictionary search with blocking

- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = \[(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3\] compares
Exercise

- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4, 8$ and $16$. 
Total space

- By increasing $k$, we could cut the pointer space in the dictionary, at the expense of search time; space 9.5MB $\rightarrow$ ~8MB
- Net – postings take up most of the space
  - Generally kept on disk
  - Dictionary compressed in memory
Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last $k-1$ in a block of $k$)

$$8\text{automata}8\text{automate}9\text{automatic}10\text{automation}$$

$$\rightarrow 8\text{automat}^*a1\diamond e2\diamond ic3\diamond ion$$

Encodes $\text{automat}$

Extra length beyond $\text{automat}$. Begins to resemble general string compression.
## RCV1 dictionary compression summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed width</td>
<td>11.2</td>
</tr>
<tr>
<td>Dictionary-as-String with pointers to every term</td>
<td>7.6</td>
</tr>
<tr>
<td>Also, blocking $k = 4$</td>
<td>7.1</td>
</tr>
<tr>
<td>Also, Blocking + front coding</td>
<td>5.9</td>
</tr>
</tbody>
</table>
POSTINGS COMPRESSION
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.
Postings: two conflicting forces

- A term like *arachnocentric* occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \sim 20$ bits.
- A term like *the* occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case
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 20 bits.
Three postings entries

<table>
<thead>
<tr>
<th>encoding</th>
<th>postings list</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td></td>
<td></td>
<td></td>
<td>283042</td>
<td></td>
<td>283043</td>
<td></td>
<td>283044</td>
<td></td>
<td>283045</td>
<td></td>
</tr>
<tr>
<td>docIDs</td>
<td>...</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPUTER</td>
<td></td>
<td></td>
<td></td>
<td>283047</td>
<td></td>
<td>283154</td>
<td></td>
<td>283159</td>
<td></td>
<td>283202</td>
<td></td>
</tr>
<tr>
<td>docIDs</td>
<td>...</td>
<td></td>
<td>107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td></td>
<td></td>
<td></td>
<td>252000</td>
<td></td>
<td>500100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>docIDs</td>
<td>252000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gaps</td>
<td>252000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>248100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Variable length encoding**

- **Aim:**
  - For *arachnocentric*, we will use ~20 bits/gap entry.
  - For *the*, we will use ~1 bit/gap entry.
- If the average gap for a term is $G$, we want to use ~$\log_2 G$ bits/gap entry.
- **Key challenge:** encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*.
- Variable length codes achieve this by using short codes for small numbers.
Variable Byte (VB) codes

- For a gap value $G$, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits.
- 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 lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm.
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$. 
# Variable Bytecode Example

<table>
<thead>
<tr>
<th>ID:</th>
<th>824</th>
<th>829</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap:</td>
<td>824</td>
<td>5</td>
</tr>
<tr>
<td>Encoding:</td>
<td>00000110 10111000 10000101</td>
<td></td>
</tr>
<tr>
<td>Decoding:</td>
<td>6*128 + (184 – 128)</td>
<td>(133 – 128)</td>
</tr>
</tbody>
</table>
### Example (cont.)

<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:
```
00000110 10000101 10000001 00001100 10110001 00001101 00001100 10110001
```

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

For a small gap (5), VB uses a whole byte.
Other variable unit codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).

- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.

- Variable byte codes:
  - Used by many commercial/research systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).

- There is also recent work on word-aligned codes that pack a variable number of gaps into one word
Unary code

- Represent $n$ as $n$ 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is
  11111111111111111111111111111111111111110.
- Unary code for 80 is:
  111111111111111111111111111111111111111111111
  111111111111111111111111111111111111111111110
- This doesn’t look promising, but....
Gamma codes

- We can compress better with **bit-level** codes
  - The Gamma code is the best known of these.
- Represent a gap $G$ as a pair $length$ and $offset$
- $offset$ is $G$ in binary, with the leading bit cut off
  - For example $13 \rightarrow 1101 \rightarrow 101$
- $length$ is the length of $offset$
  - For $13$ ($offset$ 101), this is 3.
- We encode $length$ with **unary code**: 1110.
- Gamma code of 13 is the concatenation of $length$ and $offset$: 1110101
## Gamma code examples

<table>
<thead>
<tr>
<th>number</th>
<th>length</th>
<th>offset</th>
<th>$\gamma$-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>10,0</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1</td>
<td>10,1</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>00</td>
<td>110,00</td>
</tr>
<tr>
<td>9</td>
<td>1110</td>
<td>001</td>
<td>1110,001</td>
</tr>
<tr>
<td>13</td>
<td>1110</td>
<td>101</td>
<td>1110,101</td>
</tr>
<tr>
<td>24</td>
<td>11110</td>
<td>1000</td>
<td>11110,1000</td>
</tr>
<tr>
<td>511</td>
<td>11111110</td>
<td>11111111</td>
<td>11111110,11111111</td>
</tr>
<tr>
<td>1025</td>
<td>1111111110</td>
<td>0000000001</td>
<td>1111111110,0000000001</td>
</tr>
</tbody>
</table>
Gamma code properties

- $G$ is encoded using $2 \lfloor \log G \rfloor + 1$ bits
  - Length of offset is $\lfloor \log G \rfloor$ bits
  - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
- Gamma code is parameter-free
Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost
Exercise

- Given the following sequence of $\gamma$-coded gaps, reconstruct the postings sequence:

111000111010101111101101111011

From these $\gamma$-decode and reconstruct gaps, then full postings.
## RCV1 compression

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>with blocking, ( k = 4 )</td>
<td>7.1</td>
</tr>
<tr>
<td>with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
<tr>
<td>collection (text, xml markup etc)</td>
<td>3,600.0</td>
</tr>
<tr>
<td>collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>Term-doc incidence matrix</td>
<td>40,000.0</td>
</tr>
<tr>
<td>postings, uncompressed (32-bit words)</td>
<td>400.0</td>
</tr>
<tr>
<td>postings, uncompressed (20 bits)</td>
<td>250.0</td>
</tr>
<tr>
<td>postings, variable byte encoded</td>
<td>116.0</td>
</tr>
<tr>
<td>postings, ( \gamma )-encoded</td>
<td>101.0</td>
</tr>
</tbody>
</table>
Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 4% of the total size of the collection.
- Only 10-15% of the total size of the text in the collection.
- However, we’ve ignored positional information.
- Hence, space savings are less for indexes used in practice.
  - But techniques substantially the same.
Resources

- Managing Gigabytes (still the best book out there on index construction/compression):