Part 4: Index Construction

Francesco Ricci

Most of these slides comes from the course:
Information Retrieval and Web Search, Christopher Manning and Prabhakar Raghavan
Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?
Hardware basics

- 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.

**Inside of Hard Drive video**
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.
Google Web Farm

- The best guess is that Google now has more than 2 Million servers (8 Petabytes of RAM $8 \times 10^6$ Gigabytes)
- Spread over at least 12 locations around the world
- Connecting these centers is a high-capacity fiber optic network that the company has assembled over the last few years. (video)

The Dalles, Oregon  
Dublin, Ireland
Hardware assumptions

- **symbol** | **statistic** | **value**
- s | average seek time | 5 ms = 5 x 10^{-3} s
- b | transfer time per byte | 0.02 µs = 2 x 10^{-8} s/B
- | processor’s clock rate | 10^9 s^{-1}
- p | low-level operation | 0.01 µs = 10^{-8} s
  (e.g., compare & swap a word)
- | size of main memory | several GB
- | size of disk space | 1 TB or more

Example: Reading 1GB from disk
- If stored in contiguous blocks: 2 x 10^{-8} s/B x 10^9 B = 20s
- If stored in 1M chunks of 1KB: 20s + 10^6 x 5 x 10^{-3}s = 5020 s = 1.4 h
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/punct.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per token</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>(without spaces/punct.)</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?
- Why T < N*L?
Recall IIR 1 index construction

- 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.

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc #</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
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 for Reuters RCV1 (after having removed duplicated docid for each term)
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 (*term, doc, freq*), demands a lot of space for large collections

- $T = 100,000,000$ in the case of RCV1 – so 1.2GB
  - So ... we can do this in memory in 2015, but typical collections are much larger - e.g. the *New York Times* provides an index of >150 years of newswire

- Thus: We need to store intermediate results on disk.
Use the same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, **but by using disk** instead of memory?
  - I.e. scan the documents, and for each term write the corresponding posting \((term, doc, freq)\) on a file
  - Finally sort the postings and build the postings lists for all the terms
- **No:** Sorting \(T = 100,000,000\) records \((term, doc, freq)\) on disk is too slow – too many disk seeks
  - See next slide
- We need an external sorting algorithm.
Bottleneck

- Parse and build postings entries one doc at a time
- Then 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?

<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>$p$</td>
<td>low-level operation</td>
<td>$0.01 \mu\text{s} = 10^{-8} \text{ s}$ (e.g., compare &amp; swap a word)</td>
</tr>
</tbody>
</table>
Solution

\[(2*\text{ds-time} + \text{comparison-time})*N\log_2 N \text{ seconds}
\]

\[= (2*5*10^{-3} + 10^{-8})* 10^8 \log_2 10^8 \]

\[\sim = (2*5*10^{-3})* 10^8 \log_2 10^8 \]

since the time required for the comparison is actually negligible (as the time for transferring data in the main memory)

\[= 10^6 * \log_2 10^8 = 10^6 * 26,5 = 2,65 * 10^7 \text{ s} = 307 \text{ days!} \]

- What can we do?
Gaius Julius Caesar

Divide et Impera
BSBI: Blocked sort-based Indexing
(Sorting with fewer disk seeks)

- 12-byte (4+4+4) records \( (\text{term-id}, \text{doc-id}, \text{freq}) \)
- These are generated as we parse docs
- Must now sort 100M such 12-byte records by \textit{term}

- Define a \textbf{Block} \( \sim 10M \) such records
  - Can easily fit a couple into memory
  - Will have 10 such blocks to start with (RCV1)

- \textbf{Basic idea of algorithm:}
  - Accumulate postings for each block (write on a file), (read and) sort, write to disk
  - Then merge the sorted blocks into one long sorted order.
Sec. 4.2

Blocks obtained parsing different documents

Blocks contain term-id instead

postings to be merged

<table>
<thead>
<tr>
<th>brutus</th>
<th>d3</th>
</tr>
</thead>
<tbody>
<tr>
<td>caesar</td>
<td>d4</td>
</tr>
<tr>
<td>noble</td>
<td>d3</td>
</tr>
<tr>
<td>with</td>
<td>d4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>brutus</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>caesar</td>
<td>d1</td>
</tr>
<tr>
<td>julius</td>
<td>d1</td>
</tr>
<tr>
<td>killed</td>
<td>d2</td>
</tr>
</tbody>
</table>

merged postings

<table>
<thead>
<tr>
<th>brutus</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>brutus</td>
<td>d3</td>
</tr>
<tr>
<td>caesar</td>
<td>d1</td>
</tr>
<tr>
<td>caesar</td>
<td>d4</td>
</tr>
<tr>
<td>julius</td>
<td>d1</td>
</tr>
<tr>
<td>killed</td>
<td>d2</td>
</tr>
<tr>
<td>noble</td>
<td>d3</td>
</tr>
<tr>
<td>with</td>
<td>d4</td>
</tr>
</tbody>
</table>
Sorting 10 blocks of 10M records

- First, read each block and sort (in memory) within:
  - Quicksort takes $2N \log_2 N$ expected steps
  - In our case $2 \times (10M \log_2 10M)$ steps

- Exercise: estimate total time to read each block from disk and quicksort it
  - Approximately 7 s

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

- Done straightforwardly, need 2 copies of data on disk
  - But can optimize this
Block sorted-based indexing

\texttt{BSBIndexConstruction()}

1. $n \leftarrow 0$
2. \textbf{while} (all documents have not been processed)
3. \textbf{do} $n \leftarrow n + 1$
4. \hspace{1em} $\text{block} \leftarrow \text{ParseNextBlock()}$
5. \hspace{1em} $\text{BSBI-Invert}(\text{block})$
6. \hspace{1em} $\text{WriteBlockToDisk}(\text{block}, f_n)$
7. \hspace{1em} $\text{MergeBlocks}(f_1, \ldots, f_n; f_{\text{merged}})$

$n = \text{number of generated blocks}$
How to merge the sorted runs?

- Open all block files and maintain small read buffers - and a write buffer for the final merged index
- In each iteration select the lowest termID that has not been processed yet
- All postings lists for this termID are read and merged, and the merged list is written back to disk
- Each read buffer is refilled from its file when necessary
- 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 larger - we would end up with a scalable, but slower index construction method.

Why?
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 the postings - accumulate postings in postings lists as they occur
  - *But at the end, before writing on disk, sort the terms*
- 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 (because terms are sorted).
SPIMI-Invert

SPIMI-Invert(token_stream)
1. output_file = NewFile()
2. dictionary = NewHash()
3. while (free memory available)
4. do token ← next(token_stream)
5. if term(token) ∉ dictionary
6. then postings_list = AddToDictionary(dictionary, term(token))
7. else postings_list = GetPostingsList(dictionary, term(token))
8. if full(postings_list)
9. then postings_list = DoublePostingsList(dictionary, term(token))
10. AddToPostingsList(postings_list, docID(token))
11. sorted_terms ← SortTerms(dictionary)
12. WriteBlockToDisk(sorted_terms, dictionary, output_file)
13. return output_file

- Then merging of blocks is analogous to BSBI (plus dictionary merging).

When the memory has been exhausted - write the index of the block (dictionary, postings lists) to disk.
SPIMI: Compression

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

- For web-scale indexing (don’t try this at home!): 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 2 million servers
- 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!??
Google data centers

- Consider a **non-fault-tolerant system** with 1000 nodes
- Each node has 99.9% availability (probability to be up in a time unit), what is the availability of the system?
  - *All of them should be simultaneously up*
- Answer: 37%
  - \((p \text{ of staying up})^{\# \text{ of server}} = (0.999)^{1000}\)
- 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-id, doc-id) **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 ...

![Diagram](attachment:image.png)

3 partitions
Inverters

- An **inverter collects** all (term-id, doc-id) pairs (= postings) for one term-partition (from the different segments produced by the parsers)
- **Sorts** and **writes** to postings lists.
Data flow: MapReduce

Parser
Parser
Parser
Master
Inverter
Inverter
Inverter
Postings

assign
assign

Map phase
Reduce phase
Segment files

splits

Sec. 4.4
MapReduce

- 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.
- Solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.
MapReduce

- Index construction was just one phase
- Another phase (not shown here): transforming a term-partitioned index into a document-partitioned index (for query processing)
  - Term-partitioned: one machine handles a subrange of terms
  - Document-partitioned: one machine handles a subrange of documents
- As we will 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 : "C died.", d1 : "C came, C c’ed.") → (<C, d2>, <died, d2>, <C, d1>, <came, d1>, <C, d1>, <c’ed, d1>
  - reduce: (<C,(d2,d1,d1)>, <died,(d2)>, <came,(d1)>, <c’ed,(d1)>) → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c’ed,(d1:1)>)

*we do not write term-ids for better readability*
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 – inefficient for the OS
- 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  if \(|Z₀| = n\)
3  then for \( i \leftarrow 0 \) to \( \infty \)
4    do if \( l_i \in \text{indexes} \)
5      then \( Z_{i+1} \leftarrow \text{MERGE}(l_i, Z_i) \)
6      \((Z_{i+1} \text{ is a temporary index on disk.})\)
7      indexes \leftarrow \text{indexes} - \{l_i\}
8    else \( l_i \leftarrow Z_i \) \((Z_i \text{ becomes the permanent index } l_i.)\)
9      indexes \leftarrow \text{indexes} \cup \{l_i\}
10     \text{Break}
11  \( Z₀ \leftarrow \emptyset \)

LOGARITHMICMERGE()
1  \( Z₀ \leftarrow \emptyset \) \((Z₀ \text{ is the in-memory index.})\)
2  indexes \leftarrow \emptyset
3  while true
4  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
    - Grillo, Crimea, ...
- 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 trends

http://www.google.com/trends/
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” (i.e., terms) will arise repeatedly in parsing the docs – need efficient hashing to keep track of this
    - E.g., that the trigram $\textit{you}$ occurs in the term $\textit{deciduous}$ will be discovered on each text occurrence of $\textit{deciduous}$
    - Only need to process each term once.
Reading Material

- Sections: Chapter 4 IIR