Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

Basic inverted index construction

---



## Index construction

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

---



## Recall 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

| Term | Doc # |
|---|---|
| I | 1 |
| did | 1 |
| enact | 1 |
| julius | 1 |
| caesar | 1 |
| I | 1 |
| was | 1 |
| killed | 1 |
| i' | 1 |
| the | 1 |
| capitol | 1 |
| brutus | 1 |
| killed | 1 |
| me | 1 |
| so | 2 |
| let | 2 |
| it | 2 |
| be | 2 |
| with | 2 |
| caesar | 2 |
| the | 2 |
| noble | 2 |
| brutus | 2 |
| hath | 2 |
| told | 2 |
| you | 2 |
| caesar | 2 |
| was | 2 |
| ambitious | 2 |

---



## Key step

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

We focus on this sort step.

| Term | Doc # |
|---|---|
| I | 1 |
| did | 1 |
| enact | 1 |
| julius | 1 |
| caesar | 1 |
| I | 1 |
| was | 1 |
| killed | 1 |
| i' | 1 |
| the | 1 |
| capitol | 1 |
| brutus | 1 |
| killed | 1 |
| me | 1 |
| so | 2 |
| let | 2 |
| it | 2 |
| be | 2 |
| with | 2 |
| caesar | 2 |
| the | 2 |
| noble | 2 |
| brutus | 2 |
| hath | 2 |
| told | 2 |
| you | 2 |
| caesar | 2 |
| was | 2 |
| ambitious | 2 |

→

| Term | Doc # |
|---|---|
| ambitious | 2 |
| be | 2 |
| brutus | 1 |
| brutus | 2 |
| capitol | 1 |
| caesar | 1 |
| caesar | 2 |
| caesar | 2 |
| did | 1 |
| enact | 1 |
| hath | 1 |
| I | 1 |
| I | 1 |
| i' | 1 |
| it | 2 |
| julius | 1 |
| killed | 1 |
| killed | 1 |
| let | 2 |
| me | 1 |
| noble | 2 |
| so | 2 |
| the | 1 |
| the | 2 |
| told | 2 |
| you | 2 |
| was | 1 |
| was | 2 |
| with | 2 |

---



## RCV1: Our collection for this lecture

- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
  - This is one year of Reuters newswire (part of 1995 and 1996)
- The collection isn't really large enough, but it's publicly available and is a plausible example.

---



## A Reuters RCV1 document


REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

**Extreme conditions create rare Antarctic clouds**

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

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

| symbol | statistic | value |
|---|---|---|
| N | documents | 800,000 |
| L | avg. # tokens per doc | 200 |
| M | terms (= word types) | 400,000 |
| | avg. # bytes per token (incl. spaces/punct.) | 6 |
| | avg. # bytes per token (without spaces/punct.) | 4.5 |
| | avg. # bytes per term | 7.5 |
| | non-positional postings | 100,000,000 |

4.5 bytes per word token vs. 7.5 bytes per word type: why?



## Sort-based index construction

- As we build the index, we parse docs one at a time.
  - The final postings for any term are incomplete until the end.
- At 8 bytes per *(termID, docID)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
  - So … we can do this in memory today, 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.



## Scaling index construction

- In-memory index construction does not scale
  - Can't stuff entire collection into memory, sort, then write back
- How can we construct an index for very large collections?
- Taking into account hardware constraints. . .
  - Memory, disk, speed, etc.
- Let's review some hardware basics



## 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 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 assumptions (circa 2007)

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

## Sort using disk as "memory"?

- 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.

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search

External memory indexing



## BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

- 8-byte records *(termID, docID)*
- These are generated as we parse docs
- Must now sort 100M such 8-byte records by *termID*
- 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



```
BSBIndexConstruction()
  n ← 0
  while (all documents have not been processed)
  do n ← n + 1
     block ← ParseNextBlock()
     BSBI-Invert(block)
     WriteBlockToDisk(block, f_n)
  MergeBlocks(f_1, ..., f_n; f_merged)
```



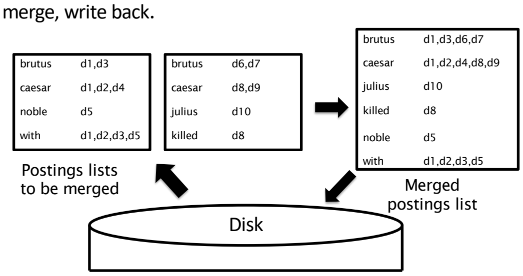
## Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes O(*N* ln *N)* expected steps
  - In our case *N=10M*
- 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



## 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.



## How to merge the sorted runs?

- But it is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously
  - Open all block files simultaneously and maintain a read buffer for each one and a write buffer for the output file
  - In each iteration, pick the lowest termID that hasn't been processed using a priority queue
  - Merge all postings lists for that termID and write it out
- 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.



## 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(token_stream)
  output_file = NEWFILE()
  dictionary = NEWHASH()
  while (free memory available)
  do token ← next(token_stream)
     if term(token) ∉ dictionary
       then postings_list = ADDTODICTIONARY(dictionary, term(token))
       else postings_list = GETPOSTINGSLIST(dictionary, term(token))
     if full(postings_list)
       then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
     ADDTOPOSTINGSLIST(postings_list, docID(token))
  sorted_terms ← SORTTERMS(dictionary)
  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
  return output_file
```

- Merging of blocks is analogous to BSBI.



## SPIMI in action

| Input token | Dictionary | Sorted dictionary |
|---|---|---|
| Caesar d1 | brutus d1 d3 | brutus d1 d3 |
| with d1 | | caesar d1 d2 d4 |
| Brutus d1 | with d1 d2 d3 d5 | noble d5 |
| Caesar d2 | noble d5 | with d1 d2 d3 d5 |
| with d2 | | |
| Brutus d3 | caesar d1 d2 d4 | |
| with d3 | | |
| Caesar d4 | | |
| noble d5 | | |
| with d5 | | |





## SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
- More on this later …

Original publication on SPIMI: Heinz and Zobel (2003)

Introduction to
# Information Retrieval

CS276: Information Retrieval and Web Search

Distributed indexing



## 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?



## Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)



## Massive data centers

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
- Answer: 37% - meaning, 63% of the time one or more servers is down.
- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.



## Distributed indexing

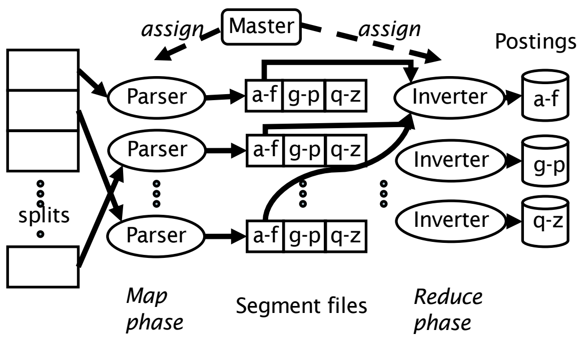
- Maintain a *master* machine directing the indexing job – considered "safe".
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.



## 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)

## Data flow






## 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
- Example: 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



## Example for index construction

Map: Caesar conquered

d1 : C came, C c' ed.

d2 : C died.

→

<C,d1>, <came,d1>, <C,d1>, <c' ed, d1>, <C, d2>, <died,d2>

Reduce:

(<C,(d1,d1,d2)>, <died,(d2)>, <came,(d1)>, <c' ed,(d1)>)

→

(<C,(d1:2,d2:1)><died,(d2:1)>, <came,(d1:1)>,<c' ed,(d1:1)>)



## Index construction

- 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' ll discuss in the web part of the course, most search engines use a document-partitioned index … better load balancing, etc.



## 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.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

## 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: collection → list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

Introduction to

# Information Retrieval

CS276: Information Retrieval and Web Search
Dynamic indexing



## 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

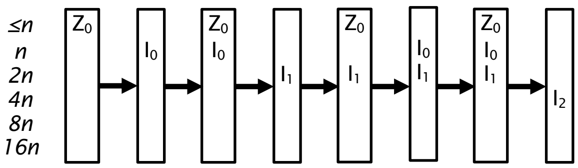
- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for 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
  - At any time, some of these powers of 2 are instantiated
- Keep smallest ($Z_0$) in memory
- Larger ones ($I_0$, $I_1$, …) 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 $Z_2$

## Logarithmic merge in action








```
LMergeAddToken(indexes, Z0, token)
  Z0 ← Merge(Z0, {token})
  if |Z0| = n
    then for i ← 0 to ∞
         do if Ii ∈ indexes
              then Zi+1 ← Merge(Ii, Zi)
                     (Zi+1 is a temporary index on disk.)
                   indexes ← indexes − {Ii}
              else Ii ← Zi    (Zi becomes the permanent index Ii.)
                   indexes ← indexes ∪ {Ii}
                   Break
         Z0 ← ∅

LogarithmicMerge()
  Z0 ← ∅    (Z0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMergeAddToken(indexes, Z0, getNextToken())
```



## Logarithmic merge

- Auxiliary and main index:
  - T/n merges where T is # of postings and n is size of auxiliary
  - Index construction time is $O(T^2/n)$ as in the worst case a posting is touched T/n times
- Logarithmic merge: Each posting is merged at most O(log (T/n)) times, so complexity is O(T log (T/n))
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of O(log (T/n)) 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 speak of spell-correction: which of several corrected alternatives do we present to the user?
  - We may want to 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
- 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 deleted



## Earlybird: Real-time search at Twitter

- Requirements for real-time search
  - Low latency, high throughput query evaluation
  - High ingestion rate and immediate data availability
  - Concurrent reads and writes of the index
  - Dominance of temporal signal

## Earlybird: Index organization

- Earlybird consists of multiple index segments
  - Each segment is relatively small, holding up to $2^{23}$ tweets
  - Each posting in a segment is a 32 bit word: 24 bits for the tweet id and 8 bits for the position in the tweet
- Only one segment can be written to at any given time
  - Small enough to be in memory
  - New postings are simply appended to the postings list
  - But the postings list is traversed backwards to prioritize newer tweets
- The remaining segments are optimized for read-only
  - Postings sorted in reverse chronological order (newest first)





## Other sorts of indexes

- Positional indexes
  - Same sort of sorting problem … just larger ⇐ Why?
- 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"



## Resources for today's lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)
- Earlybird: Busch et al, ICDE 2012