CMPE 493
INTRODUCTION TO
INFORMATION RETRIEVAL

Index Construction

Department of Computer Engineering, Boğaziçi University
November 30, 2020

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.

Hardware basics

- Servers used in IR systems now typically have several GB of main memory.
- Available disk space is several orders of magnitude larger.
- Fault tolerance is very expensive: It is much cheaper to use many regular machines rather than one fault tolerant machine.

RCV1: Our collection for this lecture

- Shakespeare's collected works definitely are not large enough for demonstrating many of the points in this course.
- The collection we will use isn't really large enough either, but it is 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.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

REUTERS

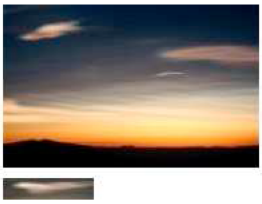
You are here: Home > News > Science > **Article**

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
T	tokens	100,000,000

Recall sort-based index construction (Lec1)

- 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

9

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.

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

10

Sort-based index construction

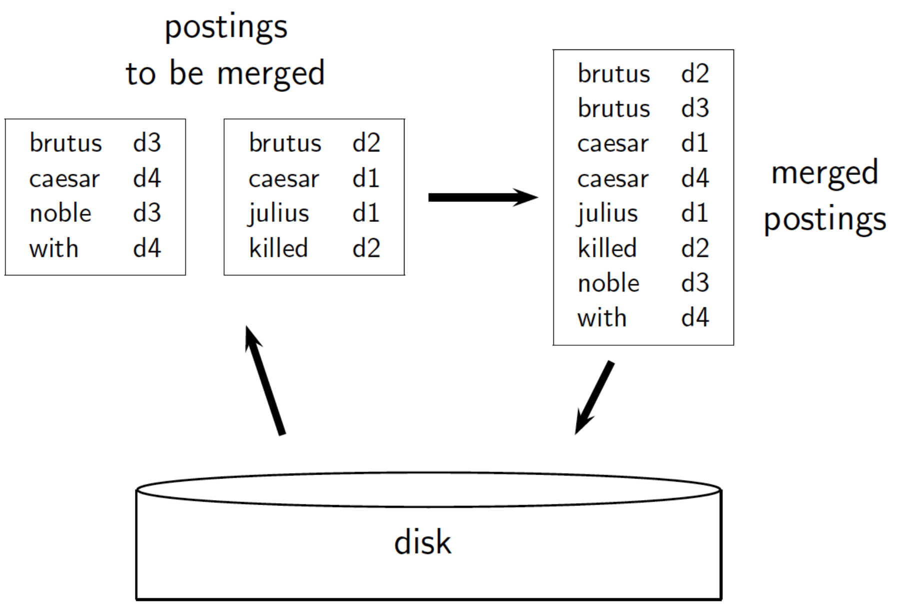
- As we build the index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry *(termID, docID, freq)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
  - So … we can do this in memory (1.2 GB), 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. (Need to use an external sorting algorithm).

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.

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

- 12-byte (4+4+4) records *(termID, docID, freq)*.
- These are generated as we parse docs.
- Must 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.



Blocked sort-based Indexing

BSBINDEXCONSTRUCTION()
$n \leftarrow 0$
while (all documents have not been processed)
do $n \leftarrow n + 1$
    $block \leftarrow$ PARSENEXTBLOCK()
    BSBI-INVERT($block$)
    WRITEBLOCKTODISK($block$, f_n)
MERGEBLOCKS($f_1, \ldots, f_n; f_{\text{merged}}$)

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: 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: 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?

Web search engine data centers

- Web search engine (e.g. Google, Bing, Baidu) data centers mainly contain commodity machines.
- Data centers are distributed around the world.

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)

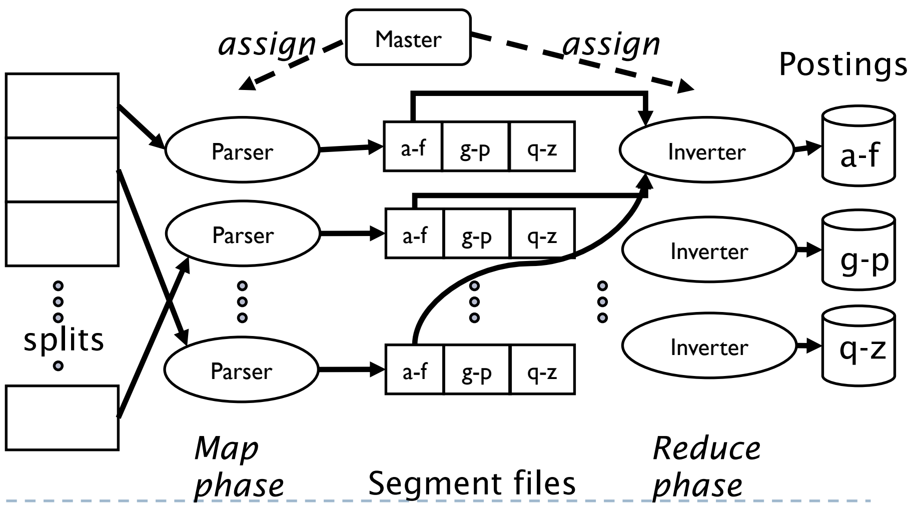
Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, docID) 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,docID) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists

Data flow



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.

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:	$(\langle \text{termID}_1, \text{list(docID)}\rangle, \langle \text{termID}_2, \text{list(docID)}\rangle, \ldots)$	$\rightarrow (\text{postings\_list}_1, \text{postings\_list}_2, \ldots)$

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow (\langle \text{C}, d_2\rangle, \langle \text{DIED}, d_2\rangle, \langle \text{C}, d_1\rangle, \langle \text{CAME}, d_1\rangle, \langle \text{C}, d_1\rangle, \langle \text{C'ED}, d_1\rangle)$
reduce:	$(\langle \text{C}, (d_2, d_1, d_1)\rangle, \langle \text{DIED}, (d_2)\rangle, \langle \text{CAME}, (d_1)\rangle, \langle \text{C'ED}, (d_1)\rangle)$	$\rightarrow (\langle \text{C}, (d_1{:}2, d_2{:}1)\rangle, \langle \text{DIED}, (d_2{:}1)\rangle, \langle \text{CAME}, (d_1{:}1)\rangle, \langle \text{C'ED}, (d_1{:}1)\rangle)$

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

29

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

30

Issues with main and auxiliary indexes

- Problem of frequent merges
- 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 O/S.
- 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.)

31

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

32

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 then deleted

33

Building Positional Indexes

- Basically the same problem except that the intermediate data structures are large.

34

Resources

- *Slides adapted from the lecture notes at the book's web site: Introduction to Information Retrieval*, chapter 4.
  - http://nlp.stanford.edu/IR-book/information-retrieval-book.html
- Challenges in Building Large-Scale Information Retrieval Systems, Jeff Dean (Google Fellow)
  - http://videolectures.net/wsdm09_dean_cblirs/
- Information about Caffeine on Google's official blog:
  - http://googleblog.blogspot.com.tr/2010/06/our-new-search-index-caffeine.html
- Original MapReduce paper by Dean and Ghemawat, 2004 posted on Moodle.
- SPIMI paper by Heinz and Zobel, 2003 posted on Moodle.