

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 **450,000 servers** (2 Petabytes of RAM 2×10^6 Gigabytes)
- ❑ Spread over at least **25 locations** around the world
- ❑ Connecting these centers is a high-capacity fiber optic network that the company has assembled over the last few years.



J. Markoff, NYT, June 2006

Google is building two computing centers, top and left, each the size of a football field, in The Dalles, Ore.

Hardware assumptions

□ symbol	statistic	value
□ s	average seek time	5 ms = 5×10^{-3} s
□ b	transfer time per byte	$0.02 \mu\text{s} = 2 \times 10^{-8}$ s
□	processor's clock rate	10^9 s^{-1}
□ p	low-level operation (e.g., compare & swap a word)	$0.01 \mu\text{s} = 10^{-8}$ s
□	size of main memory	several GB
□	size of disk space	1 TB or more
<p>□ Example: Reading 1GB from disk</p> <ul style="list-style-type: none"> ■ If stored in contiguous blocks: $2 \times 10^{-8} \text{ s} \times 10^9 = 20\text{s}$ ■ If stored in 1M chunks of 1KB: $20\text{s} + 10^6 \times 5 \times 10^{-3}\text{s} = 5020 \text{ s} = 1.4 \text{ h}$ 		

A Reuters RCV1 document



You are here: [Home](#) > [News](#) > [Science](#) > [Article](#)

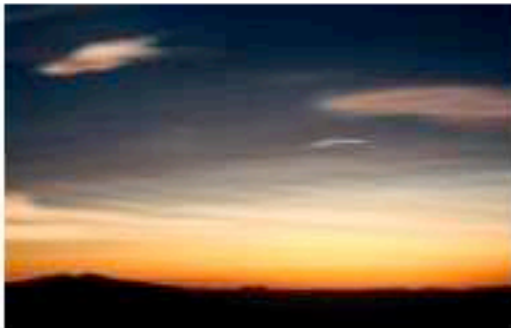
Go to a Section: [U.S.](#) [International](#) [Business](#) [Markets](#) [Politics](#) [Entertainment](#) [Technology](#) [Sports](#) [Oddly Enough](#)

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
T	non-positional postings	100,000,000

- 4.5 bytes per word token vs. 7.5 bytes per word type: why?
- Why $T \neq N * L$?

Recall IIR 1 index construction

- Documents are parsed to extract words and these are saved with the Document ID.

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

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
for Reuters RCV1

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

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 2009, 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 token write the corresponding posting (*term*, *doc*, *freq*) on a new 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
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort $T = 100\text{M}$ 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?

- $2 * ds * N \log_2 N$ s = $2 * 5 * 10^{-3} * 10^8 \log_2 10^8 = 10^6 * \log_2 10^8 = 10^6 * 26,5 = 2,65 * 10^7$ s = 307 days!
- What can we do?

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 (RCV1)
- **Basic idea of algorithm:**
 - Accumulate postings for each block, sort, write to disk
 - Then merge the sorted blocks into one long sorted order.

postings to be merged

brutus	d3
caesar	d4
noble	d3
with	d4

brutus	d2
caesar	d1
julius	d1
killed	d2



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

merged
postings

Blocks obtained
parsing different
documents



Sorting 10 blocks of 10M records

- First, read each block and sort (**in memory**) 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*
 - *Aproximately 7s*
- 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

BSBINDEXCONSTRUCTION()

1 $n \leftarrow 0$

2 **while** (all documents have not been processed)

3 **do** $n \leftarrow n + 1$

4 $block \leftarrow \text{PARSENEXTBLOCK}()$

5 $\text{BSBI-INVERT}(block)$

6 $\text{WRITEBLOCKTODISK}(block, f_n)$

7 $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$

Keeping the
dictionary in memory

n = 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
```

When the memory has been exhausted - write the index of the block (*dictionary*, *postings lists*) to disk

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

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 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!?!

Google data centers

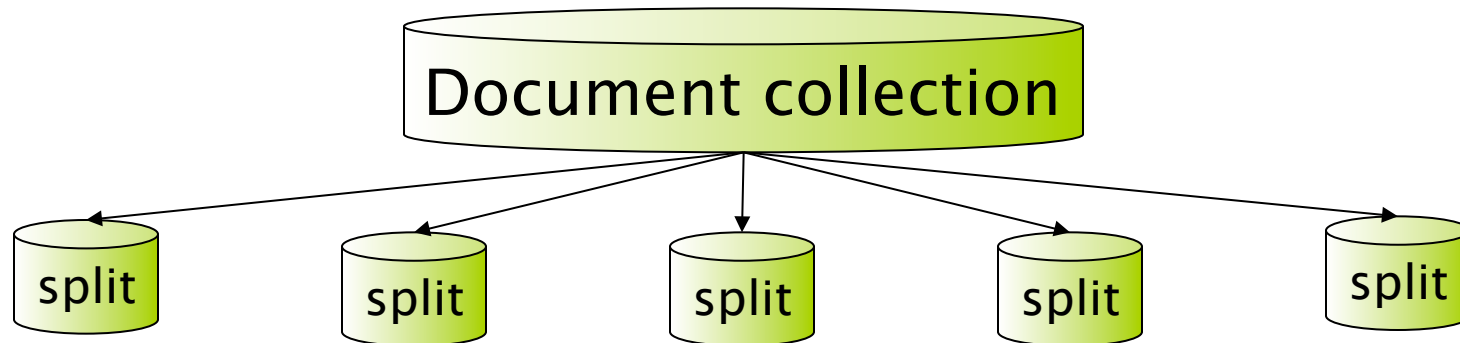
- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?
 - All of them should be simultaneously up
- Answer: 37%
 - $(p \text{ of staying up})^{\# \text{ of server}} = (0.999)^{1000}$
- Calculate the number of servers failing per minute for an 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

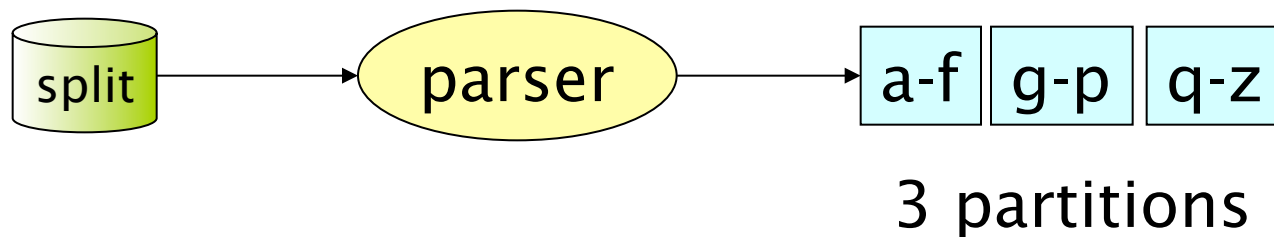
- We will use two sets of parallel tasks
 - **Parsers**
 - **Inverters**
- Break the input document collection into *splits*



- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

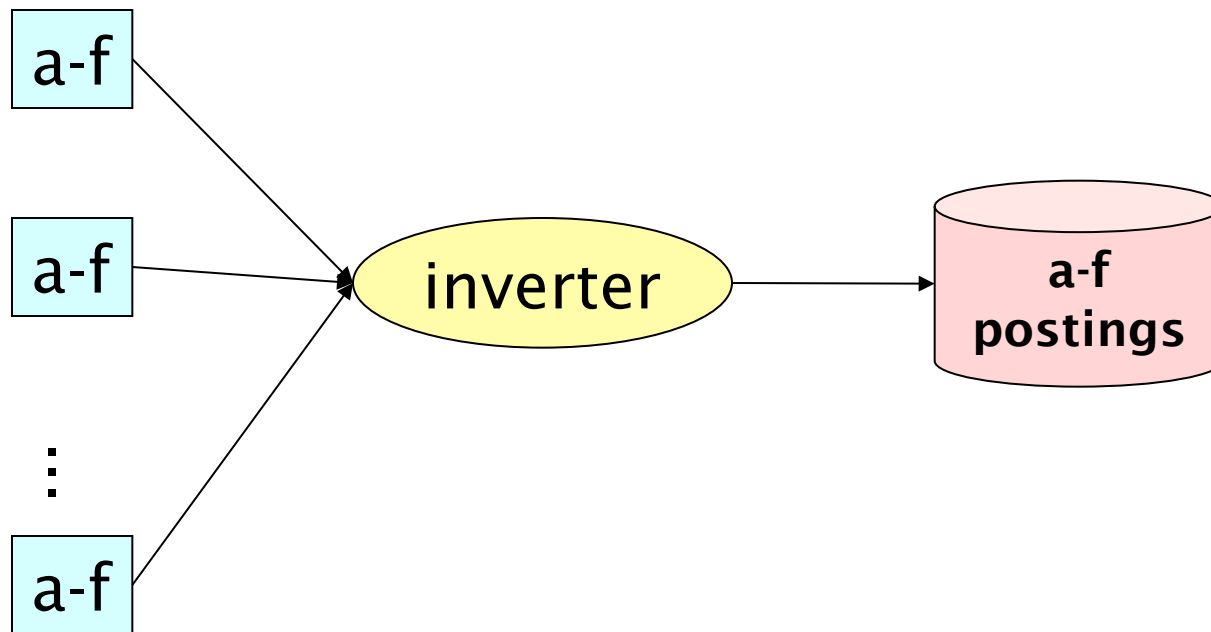
Parsers

- **Master** assigns a split to an **idle parser** machine
- Parser **reads a document** at a time and **emits (term, 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 ...

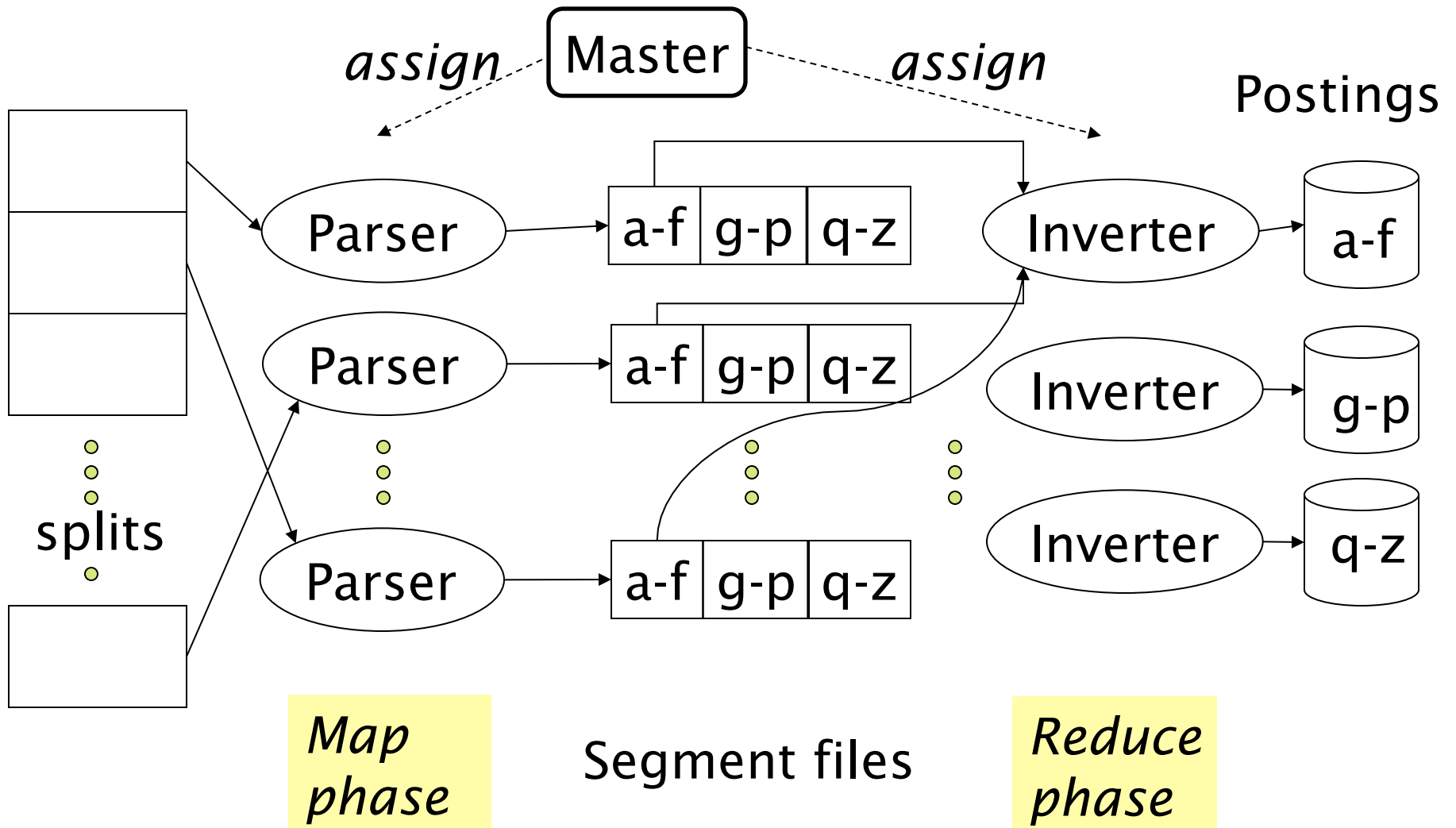


Inverters

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



Data flow: MapReduce



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*
 - *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: $\text{input} \rightarrow \text{list}(k, v)$ reduce: $(k, \text{list}(v)) \rightarrow \text{output}$
- **Instantiation of the schema for index construction**
 - map: $\text{web collection} \rightarrow \text{list}(\text{termID}, \text{docID})$
 - reduce: $(\langle \text{termID1}, \text{list}(\text{docID}) \rangle, \langle \text{termID2}, \text{list}(\text{docID}) \rangle, \dots) \rightarrow (\text{postings list1}, \text{postings list2}, \dots)$
- **Example for index construction**
 - map: $(d2 : \text{"C died."}, d1 : \text{"C came, C c'ed."}) \rightarrow (\langle C, d2 \rangle, \langle \text{died}, d2 \rangle, \langle C, d1 \rangle, \langle \text{came}, d1 \rangle, \langle C, d1 \rangle, \langle \text{c'ed}, d1 \rangle)$
 - reduce: $(\langle C, (d2, d1, d1) \rangle, \langle \text{died}, (d2) \rangle, \langle \text{came}, (d1) \rangle, \langle \text{c'ed}, (d1) \rangle) \rightarrow (\langle C, (d1:2, d2:1) \rangle, \langle \text{died}, (d2:1) \rangle, \langle \text{came}, (d1:1) \rangle, \langle \text{c'ed}, (d1:1) \rangle)$

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.

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, \dots) 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*, Z_0 , *token*)

```

1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{\text{token}\})$ 
2  if  $|Z_0| = n$ 
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                  ( $Z_{i+1}$  is a temporary index on disk.)
7                   $\text{indexes} \leftarrow \text{indexes} - \{l_i\}$ 
8              else  $l_i \leftarrow Z_i$     ( $Z_i$  becomes the permanent index  $l_i$ .)
9                   $\text{indexes} \leftarrow \text{indexes} \cup \{l_i\}$ 
10                 BREAK
11          $Z_0 \leftarrow \emptyset$ 

```

LOGARITHMICMERGE()

```

1   $Z_0 \leftarrow \emptyset$     ( $Z_0$  is the in-memory index.)
2   $\text{indexes} \leftarrow \emptyset$ 
3  while true
4  do LMERGEADDTOKEN(indexes,  $Z_0$ , 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 trends

Google trends

bolzano, trento

Search Trends

Tip: Use commas to compare multiple search terms.

[Sign in](#) to see and export additional Trends data.

Searches [Websites](#)

All regions

All years

● bolzano ● trento



Rank by

- A ['Beast of Bolzano' extradited to Italy](#)
Toronto Star - Feb 16 2008
 - B [Classical scenes from Trento and Verona](#)
International Herald Tribune - Oct 3 2008
 - C [Nazi war criminal Michael Seifert, 'the beast of Bolzano,' dies at 86 in Italy, officials say](#)
Winnipeg Free Press - Nov 6 2010
- [More news 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 uou occurs in the term ***deciduous*** will be discovered on each text occurrence of ***deciduous***
 - Only need to process each term once.



Reading Material

- ▣ Sections: Chapter 4 IIR