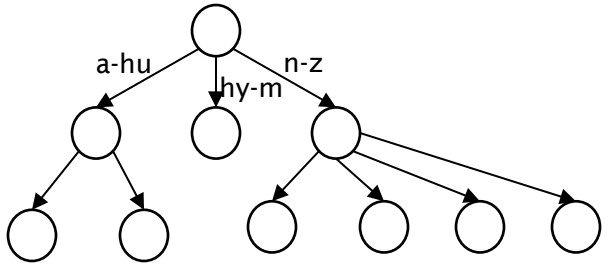
Introduction to

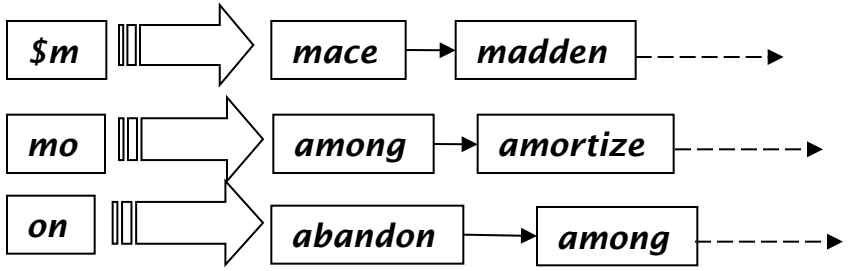
Information Retrieval

Index Construction

Plan

- Last lecture:
  - Dictionary data structures
  - Tolerant retrieval
    - Wildcards
    - Spell correction
    - Soundex
- This time:
  - Index construction





Index construction

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

Hardware basics

- Many design decisions in information retrieval are based on the characteristics of hardware e.g. size of main memory, disk and cache, read, write and other operations.
- 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 for this lecture

► **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

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

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.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > Article

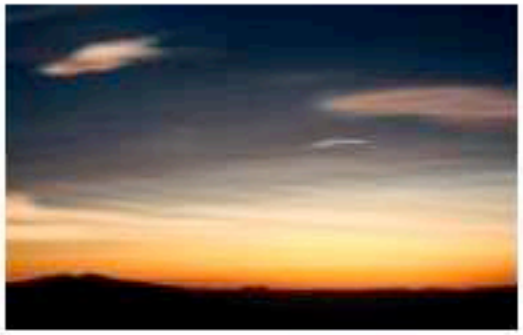
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

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

► **Table 4.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line ("case folding") in Table 5.1 (page 87).

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	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	tokens	100,000,000

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

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.
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

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 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 … 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.

Sort in "disk" or "memory"

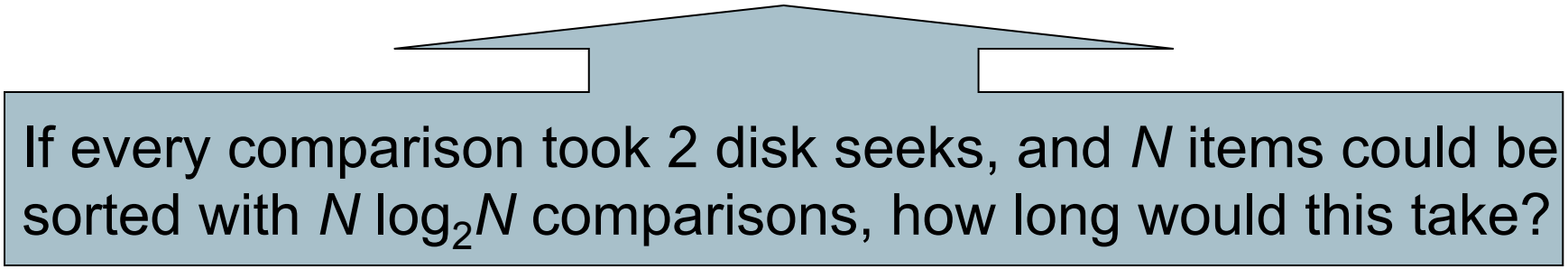
- Can we use the previous index construction algorithm (simple, positional index, or with skip pointers) for larger collections, but by using disk instead of memory?

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

- We need an *external* sorting algorithm.

Bottleneck

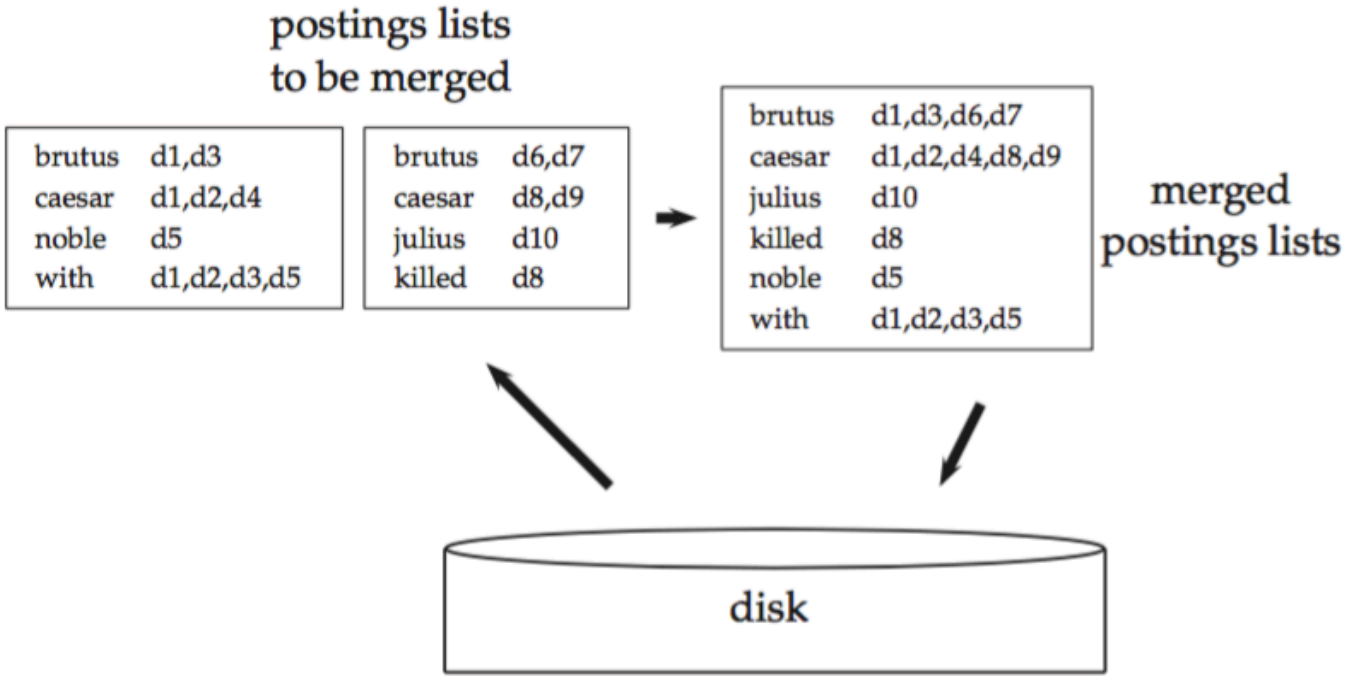
- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort T=100M records



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.

Sorting and merging of terms



► **Figure 4.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes 2*N* ln *N* expected steps (if is on disk)
  - In our case 2x(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.
  - Done straightforwardly, need 2 copies of data on disk
- But can optimize this

Algorithm description

- The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full (PARSENEXTBLOCK).
  - We choose the block size to fit comfortably into memory to permit a fast in-memory sort.
- The block is then inverted and written to disk.
  - *Inversion* involves two steps. First, we sort the termID–docID pairs. Next, we collect all termID–docID pairs with the same termID into a postings list, where a *posting* is simply a docID.
  - The result, an inverted index for the block we have just read, is then written to disk.

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

How to merge the sorted runs?

- To do the merging, open all block files simultaneously, and maintain small read buffers for the ten blocks we are reading.
- A write buffer for the final merged index we are writing.
- In each iteration, we select the lowest termID that has not been processed yet using a priority queue or a similar data structure.
- 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.

Remaining problem with BSBI

- Our assumption was to keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping (BSBI).
- Actually, we could work with term,docID postings (SPIMI) 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.)

Class Exercise

- **Exercise 4.1**
  If we need *T* log *T* comparisons (where *T* is the number of termID–docID pairs) and 2 two disk seeks for each comparison, how much time would index construction for Reuters-RCV1 take if we used disk instead of memory for storage and an unoptimized sorting algorithm (i.e., not an external sorting algorithm)? Use the system parameters in Table 4.1.

Solution

⇒Disk seek time = 5×10^{-3} s
2 x (5×10^{-3}) seconds per comparison
Transfer time = 2 x 10^{-8} s per byte
Low level operations = 10^{-8} seconds

⇒How long would it take to make $T(\log_2 T)$ comparisons with 2 disk seeks per comparison?

⇒$T(\log_2 T)$ x $2(5 \times 10^{-3}s)$
...consider transfer time and any low level operations

⇒Or see on the next slide

Solution #2

- 100,000,000 records
- $N\log_2(N)$ is = 2,657,542,475.91 comparisons
- 2 disk seeks per comparison = (2,657,542,475.91 x 0.005) = 13,287,712.38 seconds x 2
- = 26,575,424.76 seconds
- = 442,923.75 minutes
- = 7,382.06 hours
- = 307.59 days
- = 84% of a year
- = 1% of your life

Homework # 4 (a)

- **Exercise 4.2** [⋆] How would you create the dictionary in blocked sort-based indexing on the fly to avoid an extra pass through the data?

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)
 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
```

- Merging of blocks is analogous to BSBI.

SPIMI: Process

- SPIMI can index collections of any size as long as there is enough disk space available.
- The SPIMI algorithm is shown earlier. The part of the algorithm that parses documents and turns them into a stream of term–docID pairs, which we call *tokens* here, has been omitted.
- SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.
- Tokens are processed one by one (line 4). When a term occurs for the first time, it is added to the dictionary, and a new postings list is created (line 6). The call in line 7 returns this postings list for subsequent occurrences of the term.

SPIMI: Process

- A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list (line 10).
  - Instead of first collecting all termID–docID pairs and then sorting them (as we did in BSBI), each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings.
- This has two advantages:
  - It is faster and it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.
  - As a result, the blocks that individual calls of SPIMI-INVERT process are much larger and the index construction process as a whole is more efficient.
- Short spaced postings list initially and double the space each time it is full (lines 8–9).
  - Some memory is wasted. However, the overall memory requirements for the dynamically constructed index are still lower than in BSBI.

SPIMI: Process

- When memory has been exhausted, we write the index of the block (which consists of the dictionary and the postings lists) to disk (line 12).
- We have to sort the terms (line 11) before doing this because we want to write postings lists in lexicographic order to facilitate the final merging step.
- Each call of SPIMI-INVERT writes a block to disk, just as in BSBI.
- The last step of SPIMI (corresponding to line 7 in Figure 4.2; not shown in algorithm here) is then to merge the blocks into the final inverted index.
- The time complexity of SPIMI is $\Theta(T)$.
- Both the postings and the dictionary terms can be stored compactly on disk if we employ compression. Compression increases the efficiency further because we can process even larger blocks, and because the individual blocks require less space on disk (Section 4.7).

Distributed indexing

- Used for mainly 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?
  - By constructing distributed index that is partitioned across several 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 system?
  - Answer: 37% = $(99.9\%)^{1000}$
  - *Assumption: System is up if all nodes are up.
- Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?
- <2 minutes ((3*365*24*60)/1000000 = 1.5768)

Distributed indexing

- Maintain a *master* machine directing the indexing job.
- 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)

- First, the input data, in our case a collection of web pages, are split into *n splits* where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing.

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.

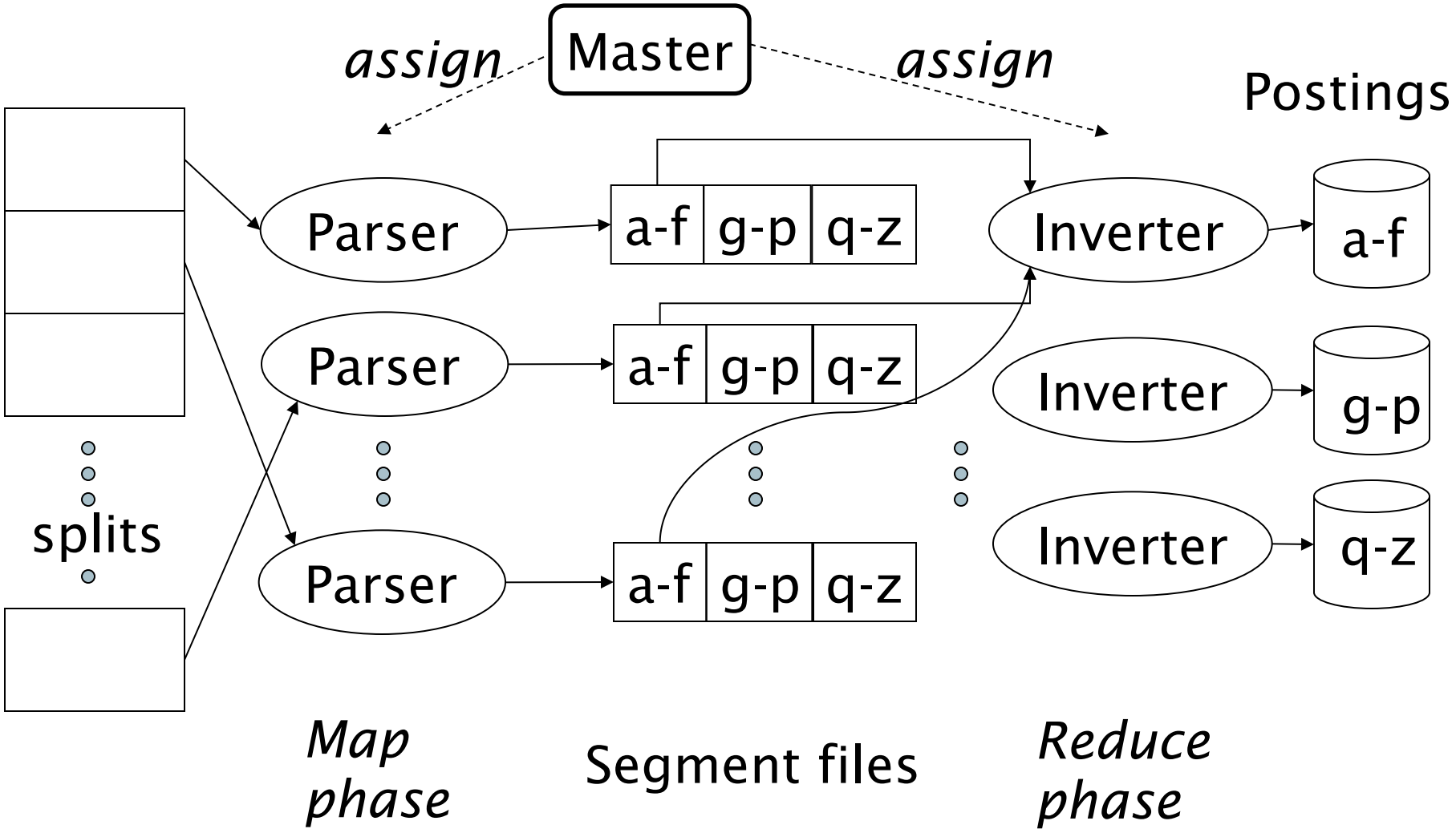
- Splits are not pre-assigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard (too slow) due to hardware problems, the split it is working on is simply reassigned to another machine.

Inverters

- An inverter collects all (term,docID) pairs (= postings) for one term-partition e.g. for a-f.
- Sorts and writes to postings lists.

- Each term partition (corresponding to *r* segment files, one on each parser) is processed by one inverter. Finally, the list of values (docIDs) is sorted for each key (term) and written to the final sorted postings list ("postings" in the figure). This completes the construction of the inverted index.

Data flow


assign
Master
assign
Postings
Parser
Parser
Parser
a-f g-p q-z
a-f g-p q-z
a-f g-p q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Segment files
Reduce phase

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.
- The original Google indexing system consisted of a number of phases, each implemented in MapReduce.

MapReduce (General)

- To minimize write times before inverters reduce the data, each parser writes its segment files to its *local disk*.
- In the reduce phase, the master communicates to an inverter the locations of the relevant segment files.
- Each segment file only requires one sequential read. This setup minimizes the amount of network traffic needed during indexing.
- The same machine can be a parser in the map phase and an inverter in the reduce phase.

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(terms, docIDs)
  - reduce: (<term1, list(docIDs)>, <term2, list(docIDs)>, …) → (postings list1, postings list2, …)

Example for index construction

- Map:
  - d2 : C died. d1 : C came, C c'ed.
  - → (⟨C, *d*2 ⟩, ⟨died,*d*2⟩, ⟨C,*d*1⟩, ⟨came,*d*1⟩, ⟨C,*d*1⟩, ⟨c'ed,*d*1⟩)
- Reduce:
  - (⟨C,(*d*2,*d*1,*d*1)⟩,⟨died,(*d*2)⟩,⟨came,(*d*1)⟩,⟨c'ed,(*d*1)⟩)
  - → (⟨C,(*d*1:2, *d*2:1)⟩, ⟨died,(*d*2:1)⟩, ⟨came,(*d*1:1)⟩, ⟨c'ed, (*d*1:1)⟩)

Exercise 4.3

- For n = 15 splits, r = 10 segments, and j = 3 term partitions, how long would distributed index creation take for Reuters-RCV1 in a MapReduce architecture? Base your assumptions about cluster machines on Table 4.1 & 4.2.

► **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	$5\text{ ms} = 5 \times 10^{-3}$ s
b	transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9\ \text{s}^{-1}$
p	lowlevel 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

► **Table 4.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line ("case folding") in Table 5.1 (page 87).

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	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	tokens	100,000,000

Solution

- We will be splitting by documents, so each split is roughly:
  split_documents = 800000/15 = 53333 documents
- Each split size is about:
  - Split_size = 53333documents x 200 token/document x 6 bytes/token
    = 63999600 bytes ≈ 61 MB
- **MAP phase:**
  - Time spent by a machine to read a split:
    Read_per_split = Split_size x (2×10^{-8} sec/byte) = 1.28 secs
  - Time spent to sort this split (algorithm complexity is $O(n\log_2 n)$:
    Sort_time = split_documents x 200 token/document x
    $\log_2$ (split_documents x 200 token/document) x (10^{-8} sec/byte) = 2.49 secs

Solution

- Time spent by a machine to write a split:
  Write_per_split = split_documents x 200 token/document x 4.5 x (2 x 10^{-8} sec/byte) = 0.96 secs -> note: here tokens are without spaces/punct. already
- MAP phase is read+sort+write:
  = 1.28 secs + 2.49 secs + 0.96 secs = 4.73 secs

There are 10 parser machines only since we have 10 segments. So to parse 15 splits we will need to do 2 passes of MAP.

Total MAP phase = 4.73 x 2 = 9.46 secs

- **REDUCE phase**

Index is split into 3 term partitions, so each term partition will hold about 100000000/3 tokens (this is a rough assumption, in reality term partition size could vary).

Solution

Each Inverter will need to read sort and write this amount of tokens.

- Term_ partition_size = 100000000/3 tokens x 4.5 bytes/tokens
  = 150000000 bytes ≈ 143 MB
- Time_reading = 150000000 bytes x (2×10^{-8} sec/byte) = 3 secs
- Time sorting = 100000000/3 tokens x $\log_2$ (100000000/3 tokens) x (10^{-8} sec/byte) = 8.33 secs
- Time_writing = 150000000 bytes x (2×10^{-8} sec/byte) = 3 secs
- Total REDUCE phase = 3 + 8.33 + 3 = 14.33 secs

Total time of Distributed Index creation = 9.46 secs + 14.33 secs = 23.79 secs

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 dynamic 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 – poor performance
- 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 remaining 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 merged Z_1 to disk as I_1 (if no I_1)
- Or merge with I_1 to form Z_2

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 2  if |Z_0| = n
 3     then for i ← 0 to ∞
 4          do if I_i ∈ indexes
 5               then Z_{i+1} ← MERGE(I_i, Z_i)
 6                    (Z_{i+1} is a temporary index on disk.)
 7                    indexes ← indexes − {I_i}
 8               else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
 9                    indexes ← indexes ∪ {I_i}
10                    BREAK
11          Z_0 ← ∅
```

```
LOGARITHMICMERGE()
1  Z_0 ← ∅    (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

Logarithmic merge

- Overall index construction time is $\Theta(T \log T)$ where T is total number of postings.
- We trade this efficiency gain for a slow down of query searching process;
- Due to complexity of dynamic indexing, some large search engines do not construct indexes dynamically. Instead, a new index is built from scratch periodically. Query processing is then switched to the new index and the old index is deleted.

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 deleted

Exercise

Exercise 4.4

For $n = 2$ and $1 \leq T \leq 30$, perform a step-by-step simulation of the algorithm in Figure 4.7. Create a table that shows, for each point in time at which $T = 2 * k$ tokens have been processed ($1 \leq k \leq 15$), which of the three indexes $I_0, \ldots, I_3$ are in use. The first three lines of the table are given below.

	I_3	I_2	I_1	I_0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0

Solution

	I3	I2	I1	I0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0
8	0	0	1	1
10	0	1	0	0
12	0	1	0	1
14	0	1	1	0
16	0	1	1	1
18	1	0	0	0
20	1	0	0	1
22	1	0	1	0
24	1	0	1	1
26	1	1	0	0
28	1	1	0	1
30	1	1	1	0

search engine land

Google Land | YAHOO! Land | Microsoft Land | Columns Land | Marketing Land | Searching Land | Ask, AOL & More Lands | Newsletters & Feeds | Confe & Wel

« Local Store And Inventory Data Poised To Transform "Online Shopping" | Main | SEO Company, Fathom Online, Acquired By Geary Interactive »

Mar 31, 2008 at 8:45am Eastern by Barry Schwartz

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.

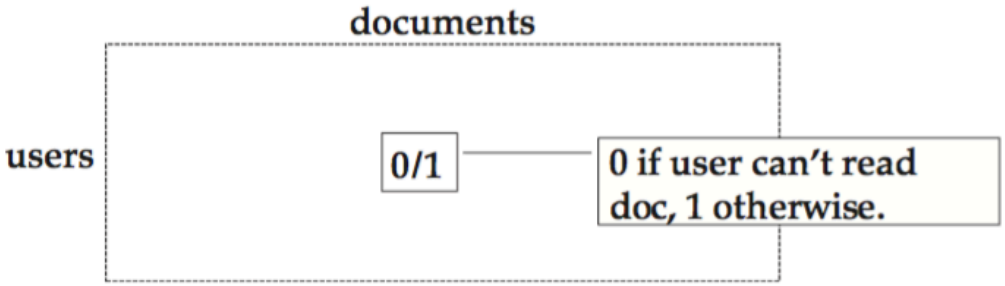
So either Google is preparing for a massive update or just messing around with our heads. As of now, these results have not yet moved over to the main Google.com results.

Other types of indexes

- Sorting algorithms discussed can all be applied to positional indexes.
- In ranked retrieval, postings are often ordered according to weight or impact, with the highest weighted postings occurring first.
- In a docID-sorted index, new documents are always inserted at the end of postings lists.
- In an impact-sorted index (will study next), the insertion can occur anywhere, thus complicating the update of the inverted index.

Other types of indexes

- *Security* is an important consideration for retrieval systems in corporations.
- User authorization is often mediated through *access control lists* or ACLs.



► **Figure 4.8** A user-document matrix for access control lists. Element (i, j) is 1 if user i has access to document j and 0 otherwise. During query processing, a user's access postings list is intersected with the results list returned by the text part of the index.

Other types of indexes

- The inverted ACL index has, for each user, a "postings list" of documents they can access – the user's access list.
- Search results are then intersected with this list. However, such an index is difficult to maintain when access permissions change.
- User membership is therefore often verified by retrieving access information directly from the file system at query time – even though this slows down retrieval.

Assignment #4(b)

- **Exercise 4.6**
- Total index construction time in blocked sort-based indexing is broken down in Ta- ble 4.3. Fill out the time column of the table for Reuters-RCV1 assuming a system with the parameters given in Table 4.1.

► **Table 4.3** The five steps in constructing an index for Reuters-RCV1 in blocked sort-based indexing. Line numbers refer to Figure 4.2.

	Step	Time
1	reading of collection (line 4)	
2	10 initial sorts of 10^7 records each (line 5)	
3	writing of 10 blocks (line 6)	
4	total disk transfer time for merging (line 7)	
5	time of actual merging (line 7)	
	total	

► **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	$5\text{ ms} = 5 \times 10^{-3}$ s
b	transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9\ \text{s}^{-1}$
p	lowlevel 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

Assignment #4(c)

- **Exercise 4.7**
- Repeat Exercise 4.6 for the larger collection in Table 4.4. Choose a block size that is realistic for current technology (remember that a block should easily fit into main memory). How many blocks do you need?

► **Table 4.4** Collection statistics for a large collection.

Symbol	Statistic	Value
N	# documents	1,000,000,000
L_{ave}	# tokens per document	1000
M	# distinct terms	44,000,000

► **Table 4.3** The five steps in constructing an index for Reuters-RCV1 in blocked sort-based indexing. Line numbers refer to Figure 4.2.

	Step	Time
1	reading of collection (line 4)	
2	10 initial sorts of 10^7 records each (line 5)	
3	writing of 10 blocks (line 6)	
4	total disk transfer time for merging (line 7)	
5	time of actual merging (line 7)	
	total	

► **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

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	lowlevel 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

Assignment #4(d)

- **Exercise 4.9**
- Assume that machines in MapReduce have 100 GB of disk space each. Assume fur- ther that the postings list of the term the has a size of 200 GB. Then the MapReduce algorithm as described cannot be run to construct the index. How would you modify MapReduce so that it can handle this case?

Articles and sources to be read

- Heinz and Zobel (2003) and Zobel and Moffat (2006) as up-do-date, in-depth treatments of index construction.
- Dynamic indexing methods are discussed in Büttcher et al. (2006) and Lester et al. (2006).
- Reuters' resources are available at the following link: https://trec.nist.gov/data/reuters/reuters.html

Programming Assignment #4(e)

- Visit the link: http://hadoop.apache.org/
- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

- Take any collection and run the map and reduce phase of hadoop to make an index with postings

Programming Assignment #4(f)

- Visit the link: http://lucene.apache.org/
- The Apache Lucene™ project provides Java-based indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/ tokenization capabilities.

- Take any collection and run the logarithmic merging of Lucene to make a dynamic index

References

- Heinz, Steffen, and Justin Zobel. 2003. Efficient single-pass index construction for text databases. *JASIST* 54(8):713–729. DOI: dx.doi.org/10.1002/asi.10268.
- Zobel, Justin, and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys* 38(2).
- Büttcher, Stefan, Charles L. A. Clarke, and Brad Lushman. 2006. Hybrid index main- tenance for growing text collections. In *Proc. SIGIR*, pp. 356–363. ACM Press. DOI: doi.acm.org/10.1145/1148170.1148233.
- Lester, Nicholas, Justin Zobel, and Hugh E. Williams. 2006. Efficient online index maintenance for contiguous inverted lists. *IP&M* 42(4):916–933. DOI: dx.doi.org/10.1016/j.ipm.2005.09.005.