Information Retrieval and Organisation

Dell Zhang

Birkbeck, University of London

Index Construction

Hardware

- In this chapter we will look at how to construct an inverted index
- Many design decisions for indexing (and information retrieval in general) are based on hardware constraints
- We begin by reviewing hardware basics in this lecture

Hardware Basics

- Access to data is much faster in memory than on disk: roughly 10x.
- 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
- Servers used in IR systems typically have
  - several GB of main memory, sometimes tens of GB;
  - a few orders of magnitude larger disk space.

Some Numbers

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	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

Reuters RCV1 Collection

- Shakespeare's Collected Works are not large enough for demonstrating many of the points in this lecture.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection that consists of English newswire articles sent over the wire in 1995 and 1996.

Reuters RCV1 Collection

REUTERS

You are here: Home > News > Science > Article

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

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 Collection

- Corpus statistics

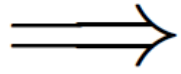
symbol	statistic	value
N	documents	800,000
L	avg. # word tokens per document	200
M	terms (= word types)	400,000
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Index Construction

- Straightforward approach:
  1. Make a pass through the collection assembling all term-docID pairs
  2. Sort pairs (using term as the dominant key, docID as the secondary key)
  3. Organize docIDs for each term into a postings list (and compute statistics like term and document frequencies)

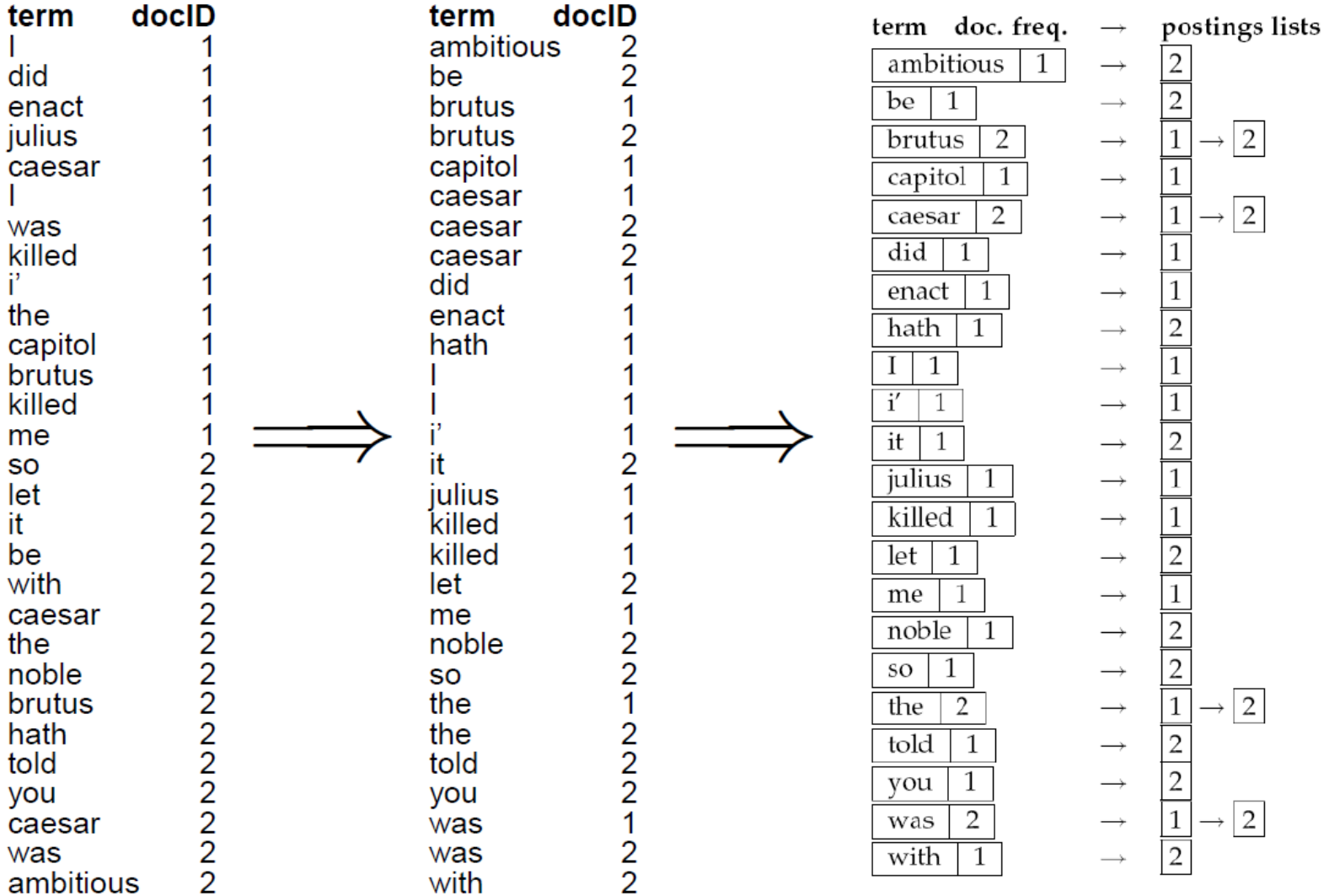
Example

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	docID
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

Example



term	docID
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	docID
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

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Sort-based Index Construction

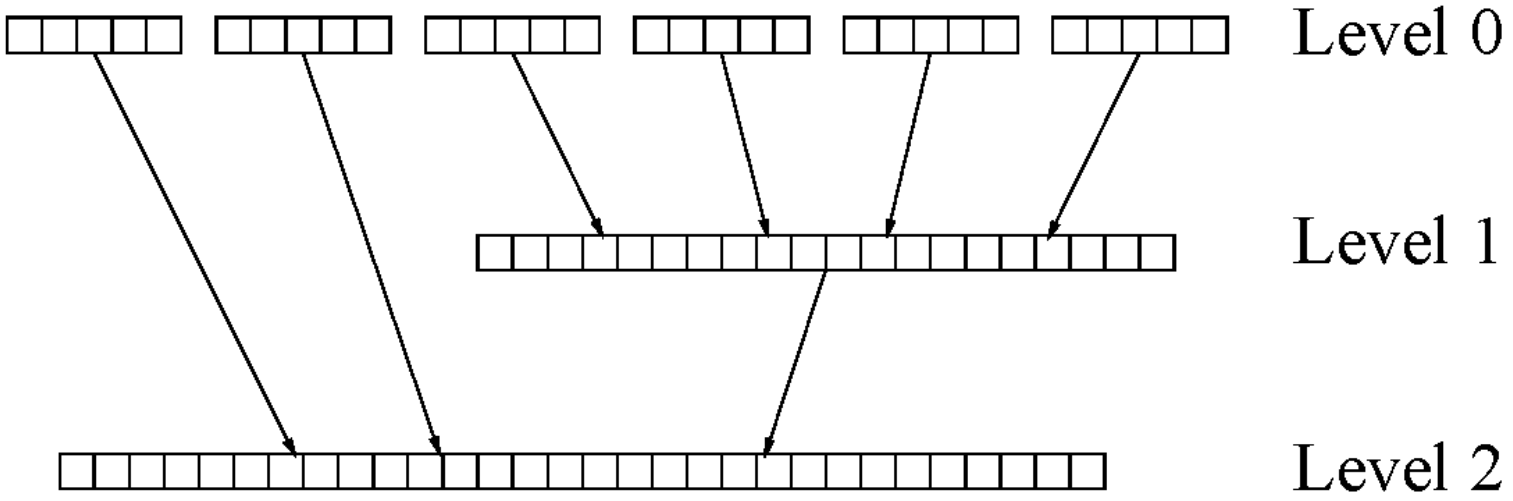
- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- At 10–12 bytes per postings entry, it demands a lot of space for large collections.
  - $T = 100{,}000{,}000$ in the case of RCV1
  - Actually, we can probably do 100,000,000 in memory, but typical collections are even larger than RCV1.
- Thus, we need to store intermediate results on disk.

Sort-based Index Construction

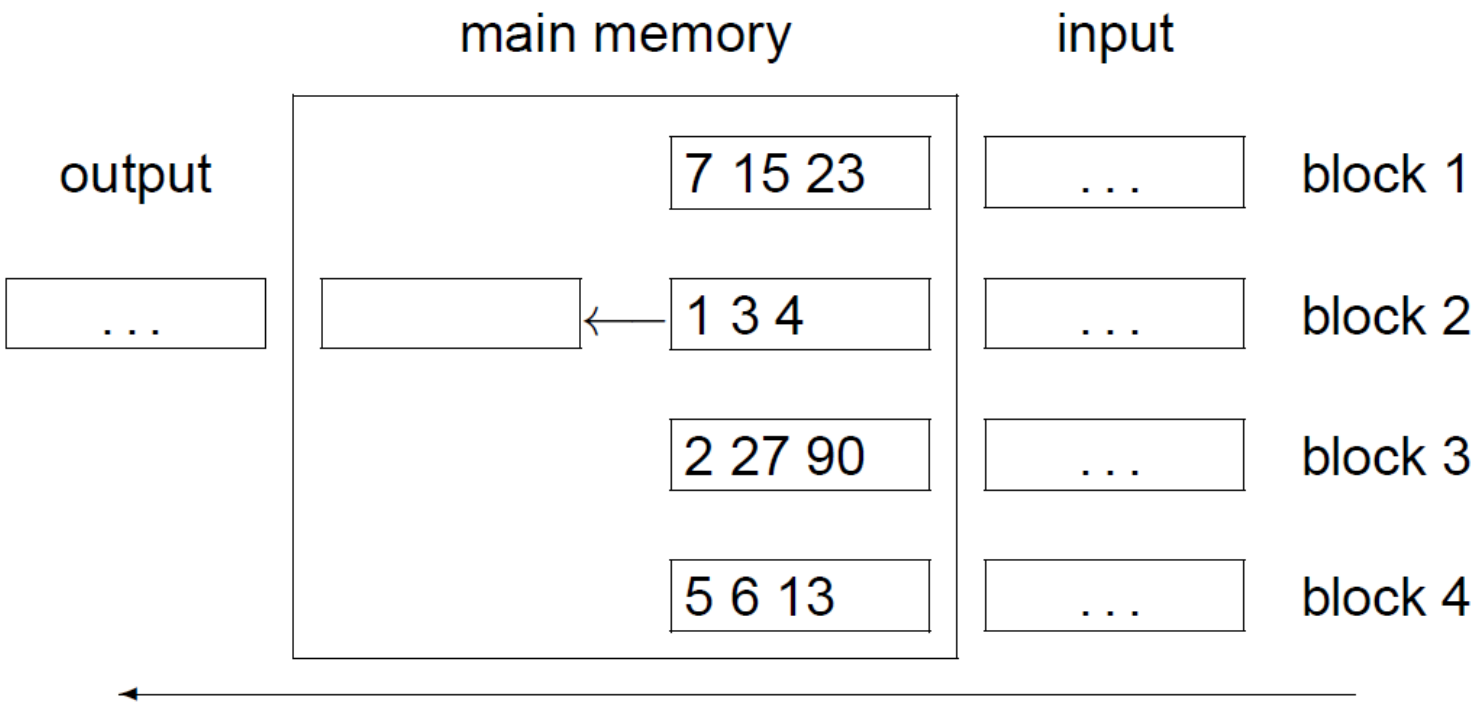
- To make index construction more efficient, we represent terms as termIDs (instead of strings)
- Build mapping from terms to termIDs on the fly (or do a two-pass approach, first compiling the vocabulary)
- Sorting $T = 100{,}000{,}000$ records on disk using standard in-memory algorithms is too slow — too many disk seeks
- We need an external sorting algorithm that minimizes the amount of random I/O

External Sorting

- Divide up the data into blocks that can fit in main memory
- Sort each block in main memory
- Sort the data by merging two (or more) blocks in separate steps



Merging Blocks



Replacement Selection

- It's even possible to sort blocks that are larger than main memory:

output						main memory				input						
						10	20	30	40	25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
				10	20	25	30	40	73	16	26	33	50	31		
			10	20	25	(16)	30	40	73	26	33	50	31			
		10	20	25	30	(16)	(26)	40	73	33	50	31				
	10	20	25	30	40	(16)	(26)	(33)	73	50	31					
10	20	25	30	40	73	(16)	(26)	(33)	(50)	31						
					16	26	31	33	50							

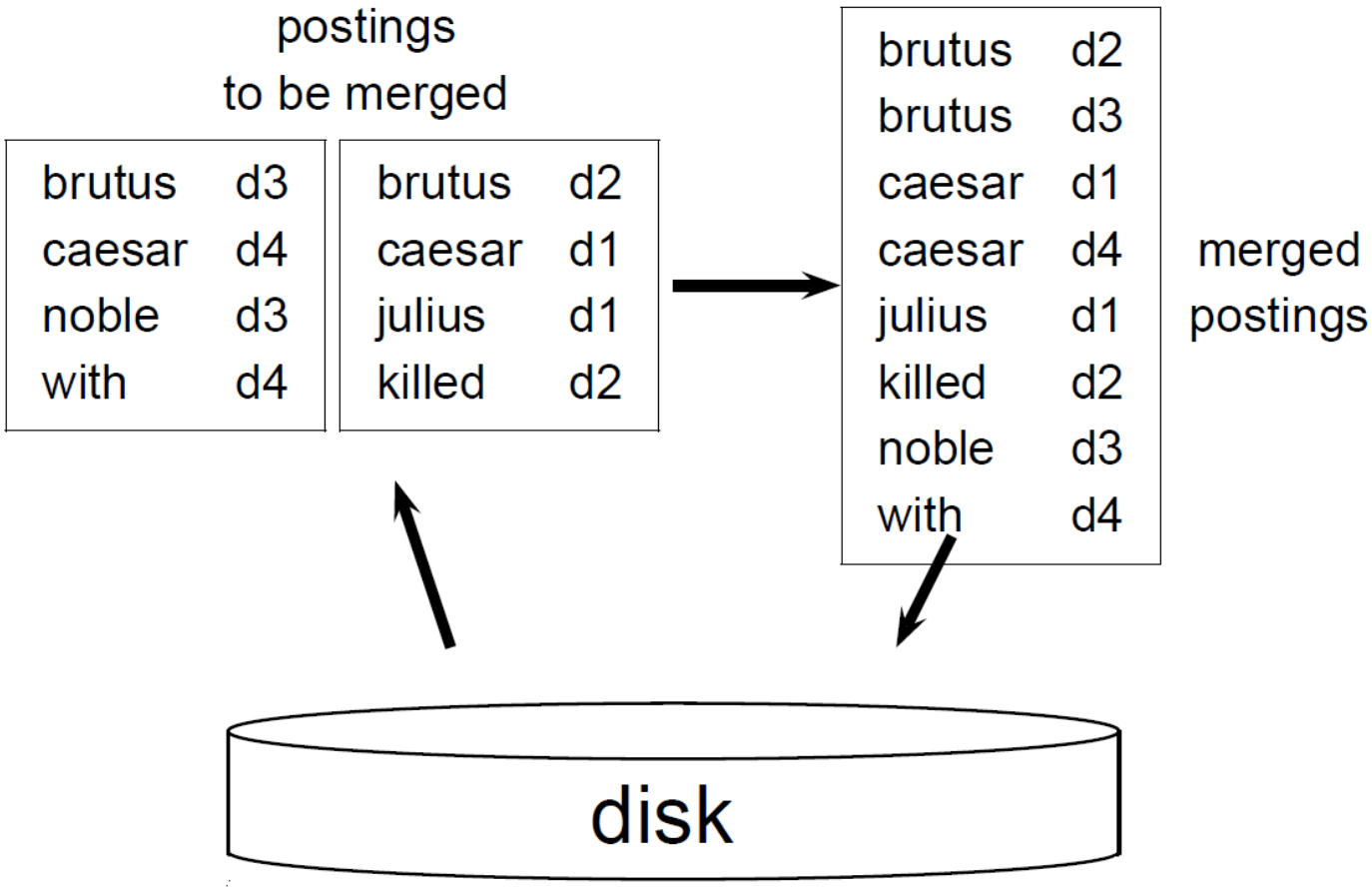
External Sorting and Inverted Indexes

- 12-byte (4+4+4) postings: (termID, docID, document frequency)
  - To simplify things a bit, we're looking at non-positional indexes
  - Techniques can be extended to positional ones
- Must now sort $T = 100{,}000{,}000$ such 12-byte postings by $\langle$termID,docID$\rangle$

External Sorting and Inverted Indexes

- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks for RCV1.
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.

Merging Postings Blocks



Another Problem

- 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.
  - So we need to come up with another solution

Single-Pass In-Memory Indexing

- Abbreviation: SPIMI
- Key ideas
  - (1) Generate separate dictionaries for each block: no need to maintain term-termID mapping across blocks.
  - (2) Don't sort immediately: 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.

Single-Pass In-Memory Indexing

- Sketch of algorithm:
  - Scan through all documents
  - If the term occurs for the first time, add it to the dictionary and allocate a new (short) postings list
  - If the term already exists in the dictionary, append docID at the end of its postings list.
    - When there is no space left, double the allocated space for postings list.
  - When we run out of memory, sort dictionary, sort postings lists, write them to disk, and then begin a new block
  - When we are through with scanning documents, merge all blocks

Single-Pass In-Memory Indexing

```
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
```

Distributed Indexing

- For Web-scale indexing, we must use a distributed computer cluster
  - Individual machines are fault-prone: they can unpredictably slow down or fail.
  - How do we exploit such a pool of machines? MapReduce.
- Now we have a new module *Cloud Computing* to cover such topics.

Dynamic Indexing

- Up to now, we have assumed that collections are static.
- However, they rarely are (especially for Web search engines).
- Documents are inserted, deleted and modified.
- This means that the dictionary and the postings lists have to be modified.
- Periodically reconstructing the entire index is usually too expensive

Quick Fix

- Maintain "big" main index on disk
- New docs go into "small" auxiliary index in memory
- Search across both, merge results
- Periodically, merge auxiliary index into one main index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this invalidation bit-vector: only return "valid" docs to user.

Issue with Auxiliary Indexes

- Frequent merges; poor performance during merge
- Actually, merging of the auxiliary index into the main index could be efficient if we keep a separate file for each postings list
  - But then we would need a lot of files (there can be millions of postings lists in an IR system).
  - Handling this number of files usually not efficient
- For the moment, let's assume that we store index in one big file
- Reality lies somewhere in between
  - e.g., split very large postings lists, collect postings lists of length 1 in one file

Costs of Merging

- Assume that we will process a total of T postings and have an auxiliary index of size n
  - i.e., there will be $\lfloor T/n \rfloor$ merges
- Each posting will be handled at least once (when merging it into the main index)
  - i.e., costs for handling postings in the order of T^2/n
- We can do better than that by doing *logarithmic merging*
- Compromises between having one big main index and millions of small ones

Logarithmic Merging

- Start with an in-memory auxiliary index Z_0 of size n
- When limit n is reached, an index I_0 of size $2^0 \times n$ is created on disk
- Next time Z_0 is full, it is merged with I_0 to create an index Z_1 of size $2^1 \times n$
  - If I_1 doesn't exist yet, Z_1 is stored as I_1
  - Otherwise, Z_1 is merged with I_1 to create Z_2 of size $2^2 \times n$ (sizes double with each step)
  - Continue with Z_2 in the same way

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
Z_0					

First Few Steps ...

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	...
$Z_0 \rightarrow I_0$					

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
	l_0				

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
Z_0	I_0				

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
	$Z_0 \cup I_0 \rightarrow Z_1 \rightarrow I_1$				

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
		I_1			

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
Z_0		I_1			

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
$Z_0 \rightarrow I_0$		I_1			

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
	I_0	I_1			

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
Z_0	I_0	I_1			

First Few Steps ...

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	...
	$Z_0 \cup I_0 \rightarrow Z_1$	I_1			

First Few Steps ...

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	...
		$Z_1 \cup I_1 \to Z_2 \to I_2$			

First Few Steps . . .

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	. . .
			l_2		

First Few Steps (Overview)

n	$2^0 \times n$	$2^1 \times n$	$2^2 \times n$	$2^3 \times n$	...
Z_0 $Z_0 \to I_0$	 I_0				
Z_0	I_0 $Z_0 \cup I_0 \to Z_1 \to I_1$	 I_1			
Z_0 $Z_0 \to I_0$	 I_0	I_1 I_1 I_1			
Z_0	I_0 $Z_0 \cup I_0 \to Z_1$	I_1 I_1 $Z_1 \cup I_1 \to Z_2 \to I_2$	 I_2		

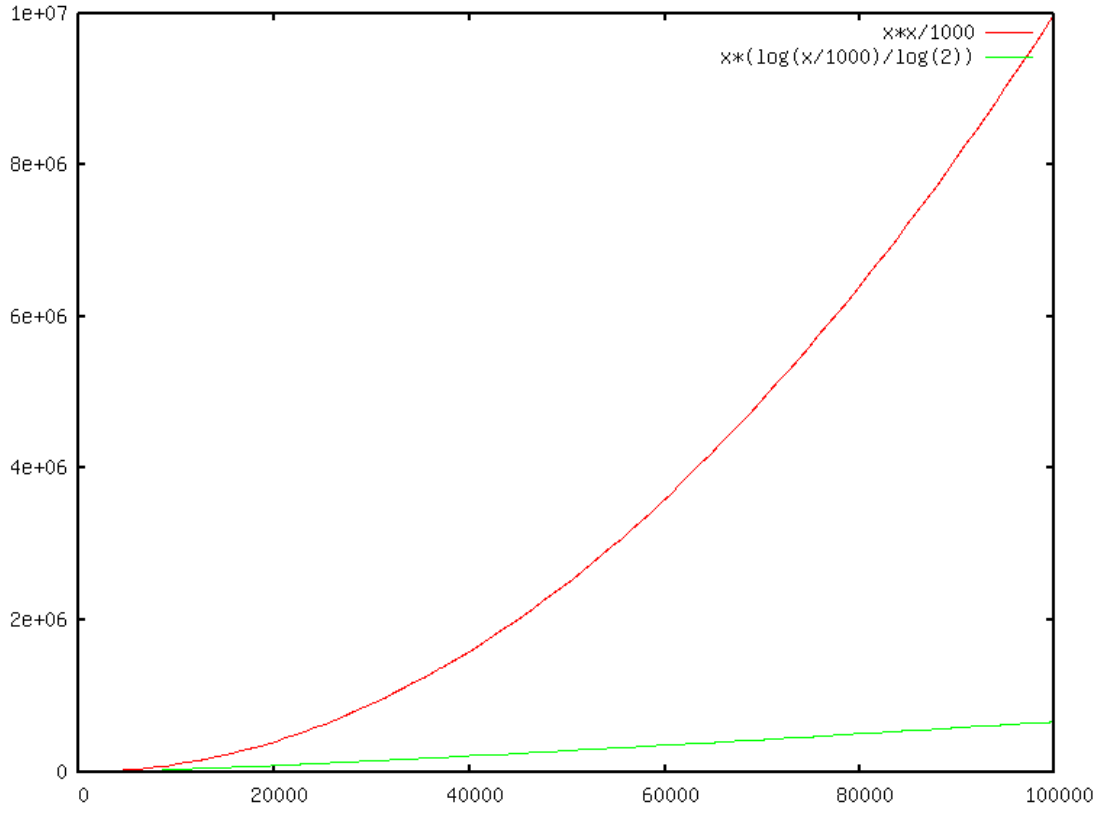
Algorithm of Logarithmic Merging

```
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 ← ∅
```

Costs of Logarithmic Merging

- What do we gain from doing logarithmic merging?
  - We have $\log_2(T/n)$ levels in indexing scheme
  - Each posting is processed once on each level
  - So, total costs are $T \log_2(T/n)$

Example for n=1000



Disadvantages?

- Slow-down of query processing:
  - we have to merge results from $\log_2(T/n)$ indexes (as opposed to just two)
- We still need to merge very large indexes occasionally. However:
  - this will happen less frequently
  - the merged indexes will on average be smaller
- Having multiple indexes complicates maintenance of collection-wide statistics
- Sometimes, rebuilding index from scratch can be better
  - depends on the frequency of updates

Summary

- Hardware puts constraints on index construction
  - Main memory is scarce
  - We want to avoid random I/O on disk
- Additional challenges for distributed and dynamic indexing