Information Retrieval

Index Construction

UNIVERSIDADE
DE AVEIRO

Previous lessons

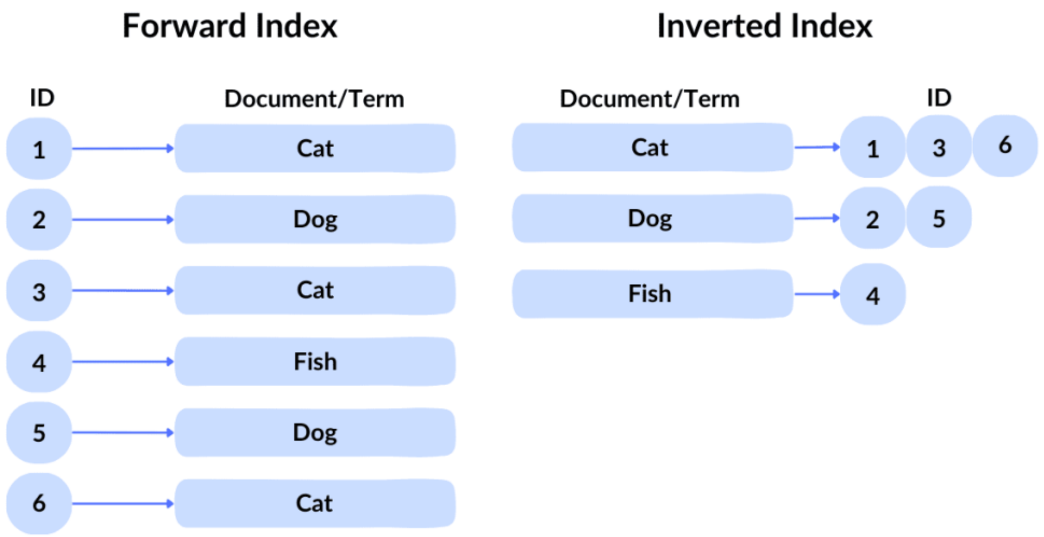
- Boolean indexing and search
- Term-document matrix
- Inverted index

- Documents
- Tokenization
- Token processing: stopping, normalization, stemming, lemmatization

- Handling phrases: bigram index, positional index

This lesson

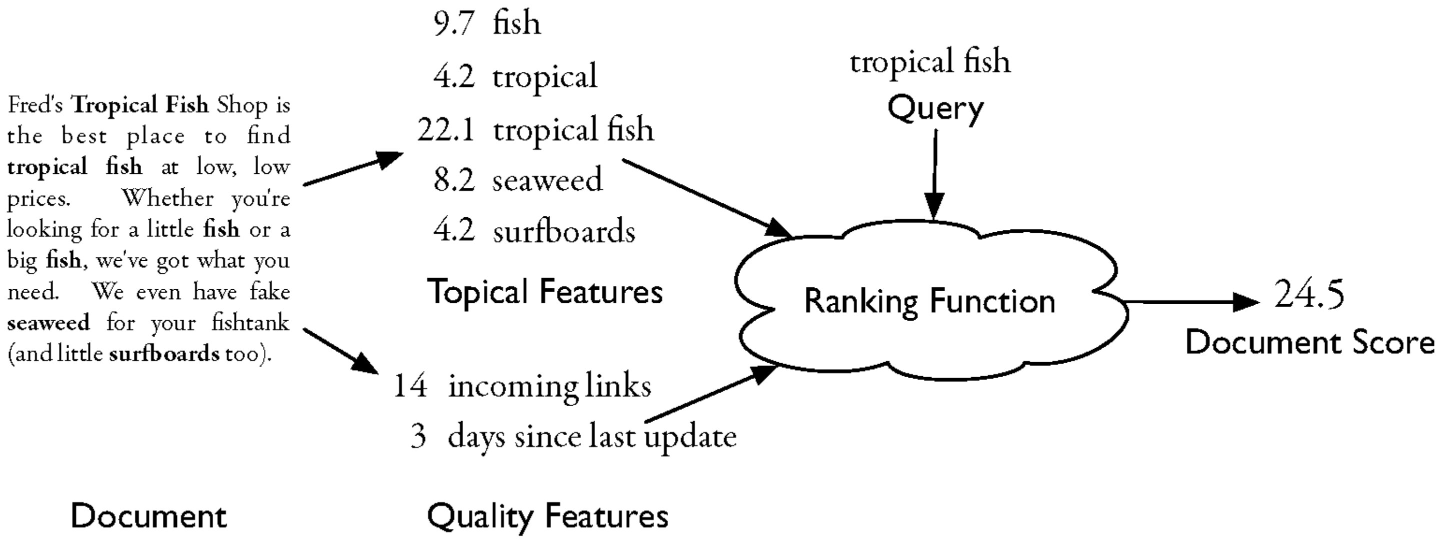
- Index construction
- Data structures
- Indexing strategies
- Distributed indexing



Indexes

- Indexes are data structures designed to make search faster
- Text search engines use a particular form of search: ranking
  - documents are retrieved in sorted order according to a score computing using the document representation, the query, and a ranking algorithm
- Requires specific data structure: most common is the inverted index
  - general name for a class of structures
  - "inverted" because documents are associated with words, rather than words with documents

Abstract Model of Ranking

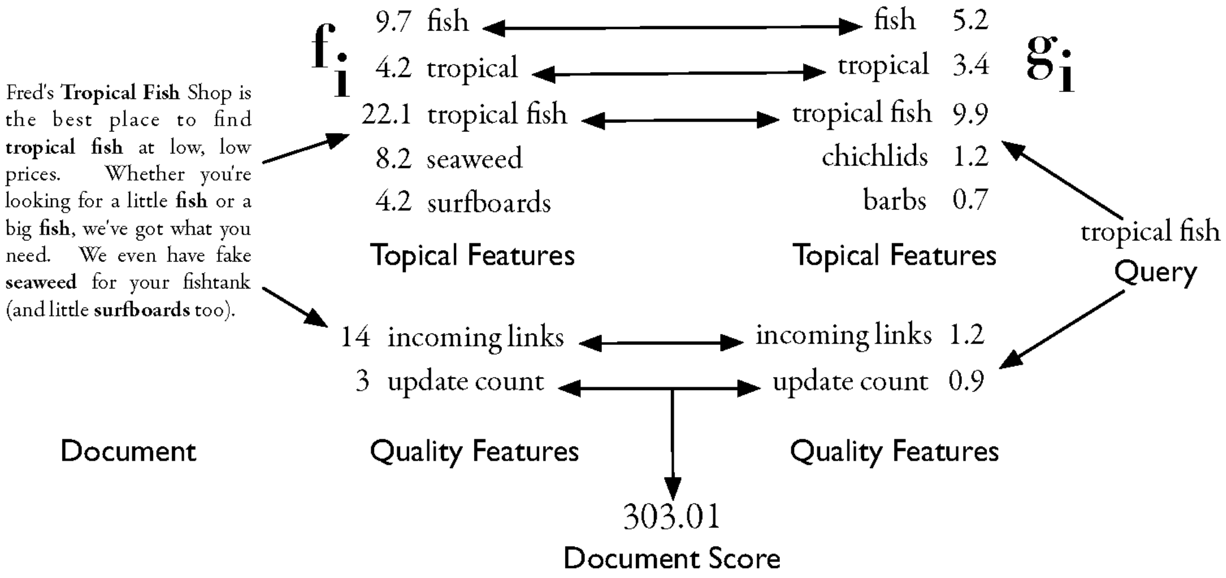


UNIVERSIDADE DE AVEIRO

More Concrete Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

f_i is a document feature function
g_i is a query feature function



Inverted Index

- Each index term is associated with an inverted list
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a posting
  - The part of the posting that refers to a specific document or location is called a pointer
  - Each document in the collection is given a unique number
  - Lists are usually document-ordered (sorted by document number)

Simple Inverted Index

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Collection

index term	posting list
and	1
aquarium	3
are	3, 4
around	1
as	2
both	1
bright	3
coloration	3, 4
derives	4
due	3
environments	1
fish	1, 2, 3, 4
fishkeepers	2
found	1
fresh	2
freshwater	1, 4
from	4
generally	4
in	1, 4
include	1
including	1
iridescence	4
marine	2
often	2, 3
only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1, 4
saltwater	2
species	1
term	2
the	1, 2
their	3
this	4
those	2
to	2, 3
tropical	1, 2, 3
typically	4
use	2
water	1, 2, 4
while	4
with	2
world	1

document-ordered

Simple Inverted Index

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- Query:
  - Tropical Fish

Term	Postings	Term	Postings
and	1	only	2
aquarium	3	pigmented	4
are	3, 4	popular	3
around	1	refer	2
as	2	referred	2
both	1	requiring	2
bright	3	salt	1, 4
coloration	3, 4	saltwater	2
derives	4	species	1
due	3	term	2
environments	1	the	1, 2
fish	1, 2, 3, 4	their	3
fishkeepers	2	this	4
found	1	those	2
fresh	2	to	2, 3
freshwater	1, 4	tropical	1, 2, 3
from	4	typically	4
generally	4	use	2
in	1, 4	water	1, 2, 4
include	1	while	4
including	1	with	2
iridescence	4	world	1
marine	2		
often	2, 3		

Simple Inverted Index

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- Issue
  - does not record the number of times each word appears
  - no reason to prefer any of these sentences over any other

Term	Postings	Term	Postings
and	1	only	2
aquarium	3	pigmented	4
are	3 4	popular	3
around	1	refer	2
as	2	referred	2
both	1	requiring	2
bright	3	salt	1 4
coloration	3 4	saltwater	2
derives	4	species	1
due	3	term	2
environments	1	the	1 2
fish	1 2 3 4	their	3
fishkeepers	2	this	4
found	1	those	2
fresh	2	to	2 3
freshwater	1 4	tropical	1 2 3
from	4	typically	4
generally	4	use	2
in	1 4	water	1 2 4
include	1	while	4
including	1	with	2
iridescence	4	world	1
marine	2		
often	2 3		

Inverted Index with counts

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- Query:
  - tropical fish
  - S2 should be preferred over S1 and S3
- Supports better ranking algorithms

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

Posting → a:b
a: document ID
b: # the word appears in ‘a’

Inverted Index with positions

one posting per word occurrence

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

and	1,15				
aquarium	3,5				
are	3,3	4,14			
around	1,9				
as	2,21				
both	1,13				
bright	3,11				
coloration	3,12	4,5			
derives	4,7				
due	3,7				
vironments	1,8				
fish	1,2	1,4	2,7	2,18	2,23
			3,2	3,6	4,3
			4,13		
fishkeepers	2,1				
found	1,5				
fresh	2,13				
freshwater	1,14	4,2			
from	4,8				
generally	4,15				
in	1,6	4,1			
include	1,3				
including	1,12				
iridescence	4,9				

marine	2,22				
often	2,2	3,10			
only	2,10				
pigmented	4,16				
popular	3,4				
refer	2,9				
referred	2,19				
requiring	2,12				
salt	1,16	4,11			
saltwater	2,16				
species	1,18				
term	2,5				
the	1,10	2,4			
their	3,9				
this	4,4				
those	2,11				
to	2,8	2,20	3,8		
tropical	1,1	1,7	2,6	2,17	3,1
typically	4,6				
use	2,3				
water	1,17	2,14	4,12		
while	4,10				
with	2,15				
world	1,11				

Posting → a:b
a: document ID
b: the word position in ‘a’

- Query:
  - tropical fish

tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

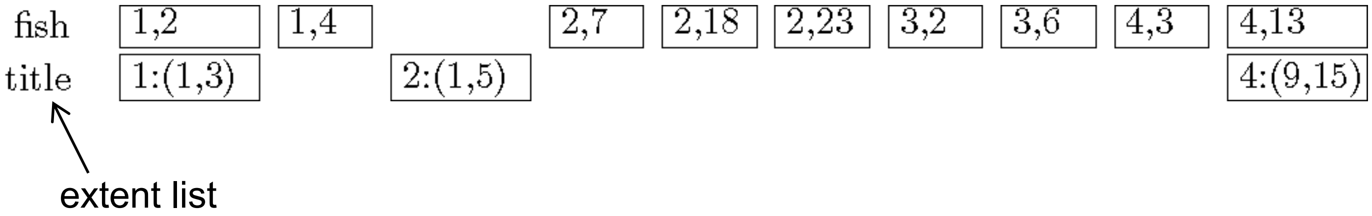
- Supports proximity matches

Fields and Extents

- Document structure is useful in the search
  - field restrictions
    - e.g., date, from:
  - some fields are more important
    - e.g., title, headings
- Options:
  - separate index (set of inverted lists) for each field type
    - e.g., one for title and one for body
    - Problem: General search must read multiple indexes
  - use extent lists

Extent Lists

- An extent is a contiguous region of a document
  - represent extents using word positions
  - record all extents for each field in an inverted list
  - e.g.,



- The title in document 1 starts from a word in position 1 and ends before position 3
  - fish in the title of document 1

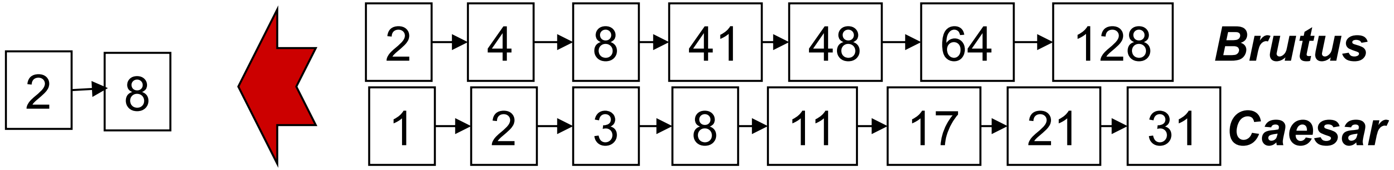
Other Issues

- Precomputed scores in inverted list
  - e.g., list for “fish” [(1:3.6), (3:2.2)], where 3.6 is total feature value for document 1
  - improves speed but reduces flexibility
    - Phrase information is lost here
- Score-ordered lists (not document-ordered)
  - only for indexes with precomputed scores
  - query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries

Faster postings merges: Skip pointers/Skip lists

Recall basic merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

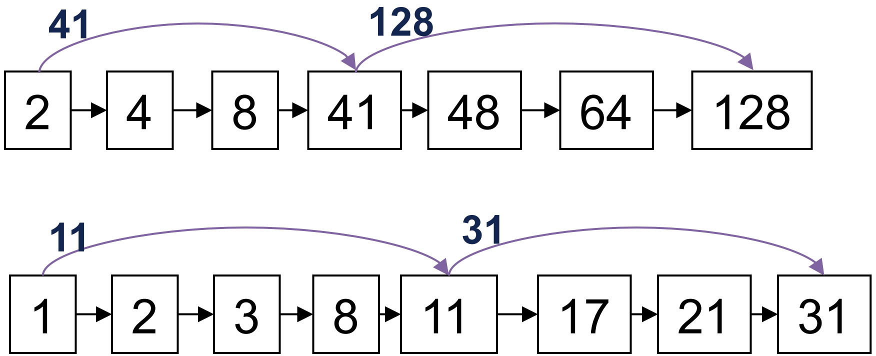


- If the list lengths are m and n, the merge takes O(m+n) operations.

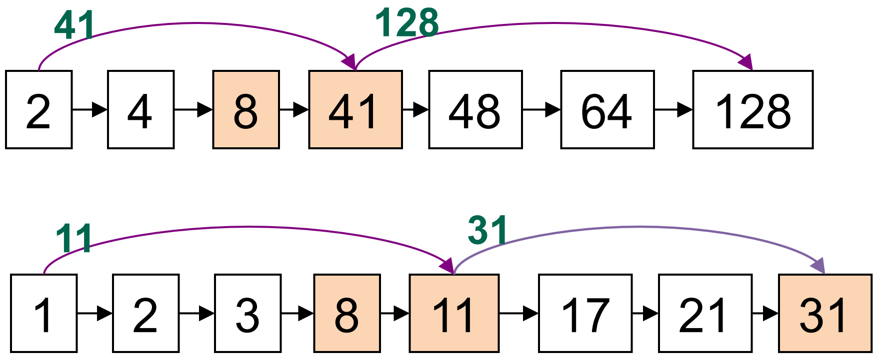
- Can we do better?

Augment postings with skip pointers (at indexing time)

- Why?
  - To skip postings that will not figure in the search results.



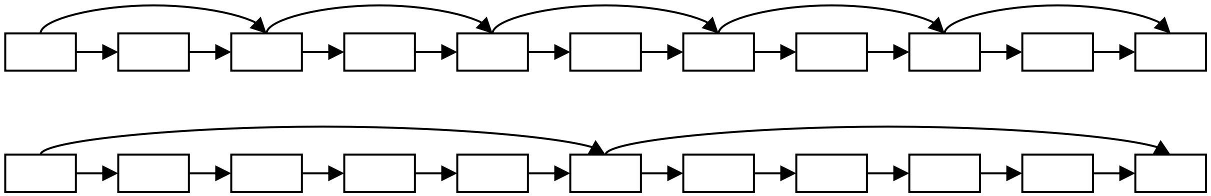
Query processing with skip pointers



- Suppose we've stepped through the lists until we process 8 on each list.
  - We match it and advance.
- We then have **41** and **11** on the lower. **11** is smaller.
- But the skip successor of **11** on the lower list is **31**, so we can skip ahead past the intervening postings.

Placing skips

- Tradeoff:
  - More skips $\rightarrow$ shorter skip spans $\Rightarrow$ more likely to skip.
    - But lots of comparisons to skip pointers.
  - Fewer skips $\rightarrow$ few pointer comparison,
    - but then long skip spans $\Rightarrow$ few successful skips.



- Simple heuristic:
  - for postings of length L, use $\sqrt{L}$ evenly-spaced skip pointers.
  - Easy if the index is relatively static; harder if L keeps changing because of updates.

Postings intersection with skip pointers

```
IntersectWithSkips(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then Add(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                      then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                              do p1 ← skip(p1)
                      else p1 ← next(p1)
              else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                      then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                              do p2 ← skip(p2)
                      else p2 ← next(p2)
return answer
```

Index Construction

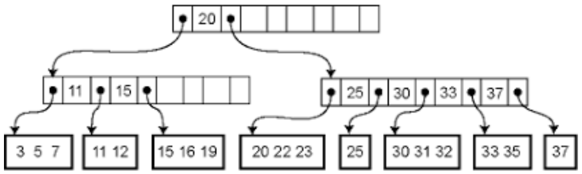
Dictionary data structures for inverted indexes

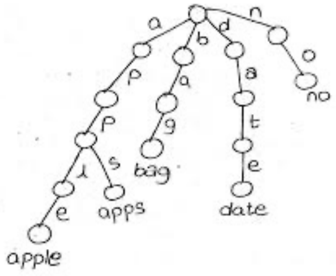
- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list …
  - in what data structure?

BRUTUS	$\longrightarrow$	1	2	4	11	31	45	173	174	
CAESAR	$\longrightarrow$	1	2	4	5	6	16	57	132	…
CALPURNIA	$\longrightarrow$	2	31	54	101					

Dictionary data structures

- Sorted array:
  - Binary search if can be kept in memory
  - High overhead for additions
- Hashing
  - Fast look-up
  - Collisions
- Binary tree (BST), B-Tree, B+Tree, Trie, ...
  - Maintain balance - always log look-up time
  - Can insert and delete
- Some IR systems use trees, some hashes

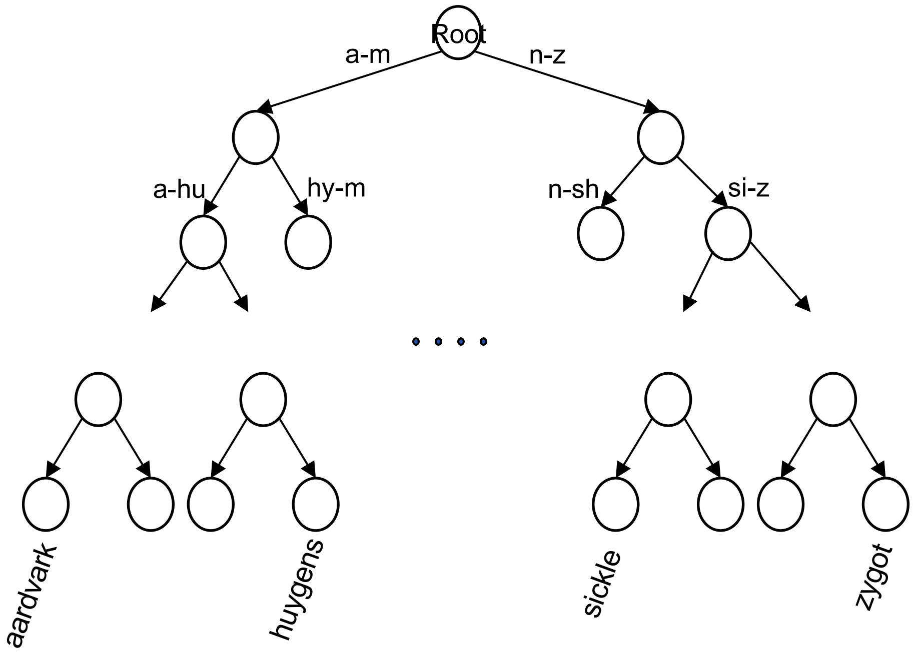




UNIVERSIDADE DE AVEIRO

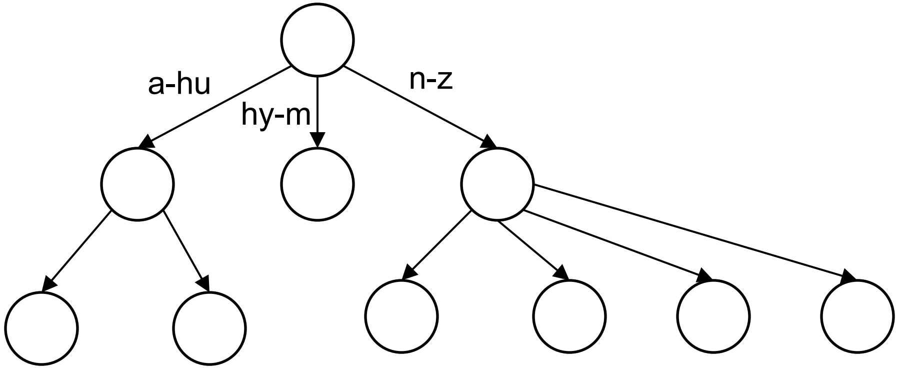
Balanced BST

- BST requires $O(log_2\ n)$ time in the average case
  - needs $O(n)$ time in the worst case, when the unbalanced tree resembles a linked list (degenerate tree)
- To keep the BST balanced we need to verify balance for each insert and delete
- BST is balanced if:
  - the difference between left and right branches is 0 or 1
  - all sub-tree are balanced



Tree: B-tree

- Definition:
  - Every internal node has several children in the interval [a,b] where a, b are appropriate natural numbers, e.g., [2,4].

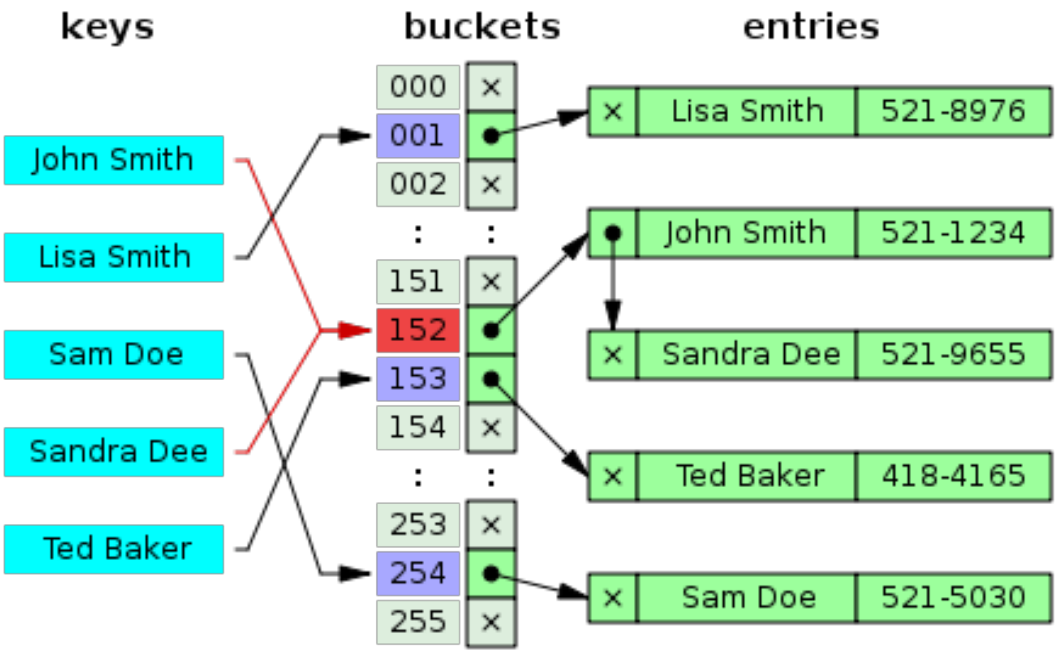


Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require ordering of strings

- Pros:
  - Solves the prefix problem
    - e.g., terms starting with "*univ*"
- Cons:
  - BST are slower: $O(\log_2 N)$
    - and this requires balanced tree
  - Rebalancing trees is expensive
    - but B-trees mitigate the rebalancing problem

Hash Table


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Hashes

- Each vocabulary term is hashed to an integer

- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No prefix search
  - No easy way to find minor variants:
    - judgment/judgement
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing everything

Index construction – A simple approach

procedure BUILDINDEX(D) ▷ D is a set of text documents
  $I \leftarrow$ HashTable() ▷ Inverted list storage
  $n \leftarrow 0$ ▷ Document numbering
  for all documents $d \in D$ **do**
    $n \leftarrow n + 1$
    $\boxed{T \leftarrow \text{Parse}(d)}$ ▷ Parse document into tokens
    Remove duplicates from T
    for all tokens $t \in T$ **do**
      if $I_t \notin I$ **then**
        $I_t \leftarrow$ Array()
      end if
      I_t.append(n)
    end for
  end for
  return I
end procedure

Parse each document into tokens

- Requires all the inverted lists in memory
- Is sequential, with no obvious way to parallelize its construction

UNIVERSIDADE DE AVEIRO

Index construction

- Many design decisions in information retrieval are based on the characteristics of hardware
  - Access to data in memory is much faster than on disk
  - Disk seeks / latency (with mechanical drives)
  - Disk I/O is block-based
  - Reading one large chunk of data to memory is faster than reading many small chunks
  - Servers used in IR systems now typically have tens or hundreds GB of main memory
  - Available disk space is several orders of magnitude larger

Dataset example: Reuters RCV1 collection

- Contains one year of Reuters newswire
  - between August 20, 1996 and August 19,1997
- Typical document:

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

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	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

Indexer steps: Sort

- 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.

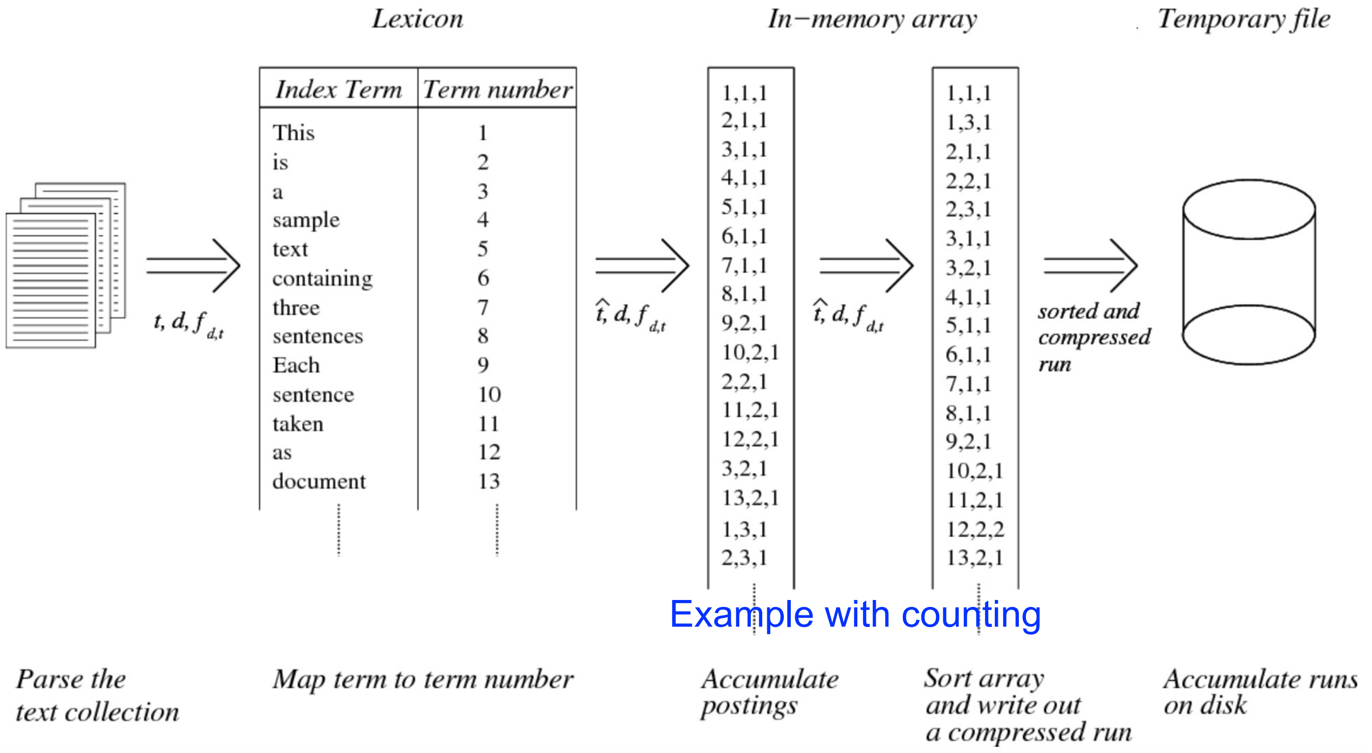
Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

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

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 (1.2GB)
  - So … we can do this in memory, but typical collections are much larger.
- How can we construct an index for very large collections?
  - Taking into account hardware constraints . . .
  - Memory, disk, speed, etc.
- We need to store intermediate results on disk
  - BSBI: Blocked sort-based Indexing
  - SPIMI: Single-pass in-memory indexing

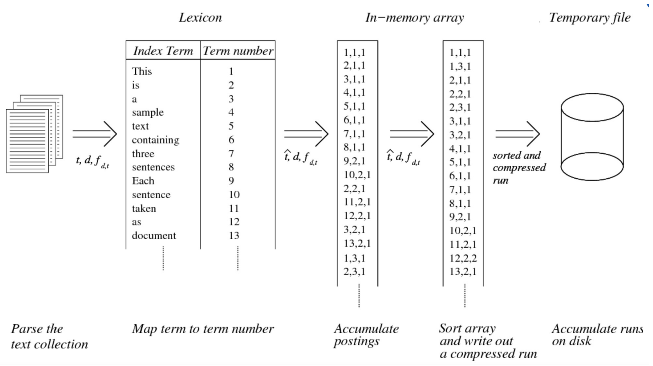
BSBI: Blocked Sort-Based Indexing



BSBI: Blocked Sort-Based Indexing

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

- Terms represented by TermID instead of strings
  - Unique serial number (int)
  - Build mapping
    - One-pass, while processing the collection
    - Two-pass approach: compile vocabulary, then construct inverted index
  - Dictionary kept in memory



SPIMI: Single-Pass In-Memory Indexing

- BSBI has excellent scaling properties, but it needs a data structure for mapping terms to termIDs.
  - For very large collections, this data structure does not fit into memory.

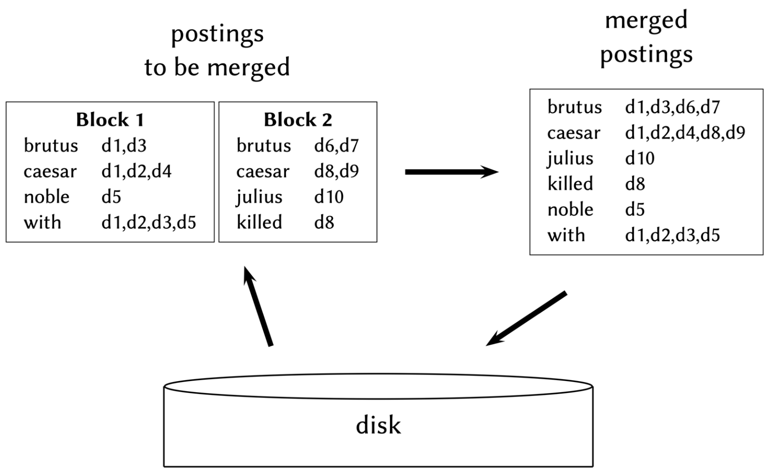
- A more scalable alternative is single-pass in-memory indexing (SPIMI),
  - SPIMI uses terms instead of termIDs
  - writes each block's dictionary to disk, and then
  - starts a new dictionary for the next block
  - SPIMI can index collections of any size as long as there is enough disk space available.

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 during indexing, instead sort when writing blocks to disk. 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

How to merge the sorted runs?

- Can do binary merges
  - For 10 blocks, merge tree will have 4 layers ($\log_2 10$)
- During each layer, read into memory runs in blocks of 10M, merge, write back



- It is more efficient to do a n-way merge, where you are reading from all blocks simultaneously
  - Reading chunks of each block into memory and then write out an output chunk

SPIMI-Invert

- Merging of blocks is analogous in BSBI and SPIMI.

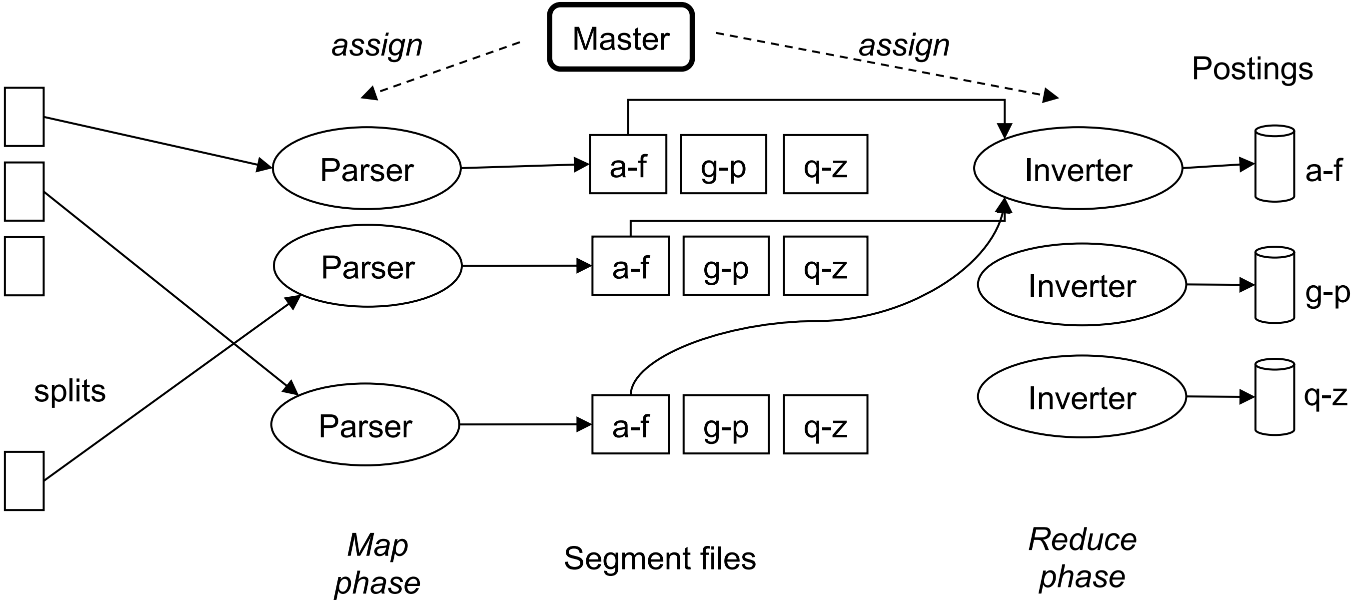
```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
   if term(token) ∉ dictionary
     then postings_list = ADDTODICTIONARY(dictionary, term(token))
     else postings_list = GETPOSTINGSLIST(dictionary, term(token))
   ~~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
```

Distributed indexing

- Distributed processing is driven by the need to index and analyze huge amounts of data
  - For large-scale indexing (e.g. web-scale)
- Quite common in some NoSQL databases
  - E.g., MongoDB, Cassandra

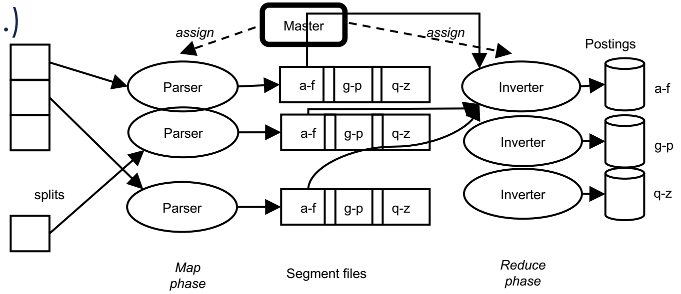
- Large numbers of inexpensive servers used rather than larger, more expensive machines
  - How do we exploit such a pool of machines?

Data flow


Master
assign
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

- MapReduce is a distributed programming tool designed for indexing and analysis tasks:
  - Map (parallel document processing)
    - Count the terms in each document (tokenize, stem, ...)
    - Produces (document, count) pairs
  - Shuffle
    - Send all pairs for the same term to the same reducer
  - Reduce (parallel term processing)
    - Build the postings file



- Fault-tolerant:
  - If a machine fails to complete, that task is restarted
- The Mapper or Reducer is called multiple times on the same input, the output will always be the same

MapReduce

- 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
- 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: input → list(k, v)
  - reduce: (k,list(v)) → output

MapReduce includes a shuffle/combining stage between map and reduce

- map:
  - collection → list(termID, docID)

```
d1 : Caesar came, Caesar conquered.
d2 : Caesar died.
→
< Caesar,d1>, <came,d1>, < Caesar,d1>, <conquered,d1>, < Caesar,d2>, <died,d2>
```

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

```
(<Caesar,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <conquered,(d1)>)
→
(<Caesar,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <conquered,(d1:1)>)
```

UNIVERSIDADE DE AVEIRO

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 the dictionary
  - New terms added to the 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

- The problem of frequent merges
  - We touch stuff a lot
  - Poor performance during the merge
- Actually:
  - Merging 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.
- 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, …) 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) to form 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
- …

$\textsc{LMergeAddToken}(indexes, Z_0, token)$

$Z_0 \leftarrow \textsc{Merge}(Z_0, \{token\})$
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
    do if $I_i \in indexes$
      then $Z_{i+1} \leftarrow \textsc{Merge}(I_i, Z_i)$
        (Z_{i+1} is a temporary index on disk.)
        $indexes \leftarrow indexes - \{I_i\}$
      else $I_i \leftarrow Z_i$ *(Z_i becomes the permanent index I_i.)*
        $indexes \leftarrow indexes \cup \{I_i\}$
        $\textsc{Break}$
    $Z_0 \leftarrow \emptyset$

$\textsc{LogarithmicMerge}()$

$Z_0 \leftarrow \emptyset$ *(Z_0 is the in-memory index.)*
$indexes \leftarrow \emptyset$
while true
do $\textsc{LMergeAddToken}(indexes, Z_0, \textsc{getNextToken}())$

Logarithmic merge

- Main and Auxiliary 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
- 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

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

Summary

- Index construction
- Data structures
- Indexing strategies
- Distributed indexing