Information Retrieval

Index Construction

Adapted from several IR courses:
C. Manning, P. Raghavan and H. Schütze (http://nlp.stanford.edu/IR-book/)
W. Croft, D. Metzler, and T. Strohman (http://www.search-engines-book.com/)

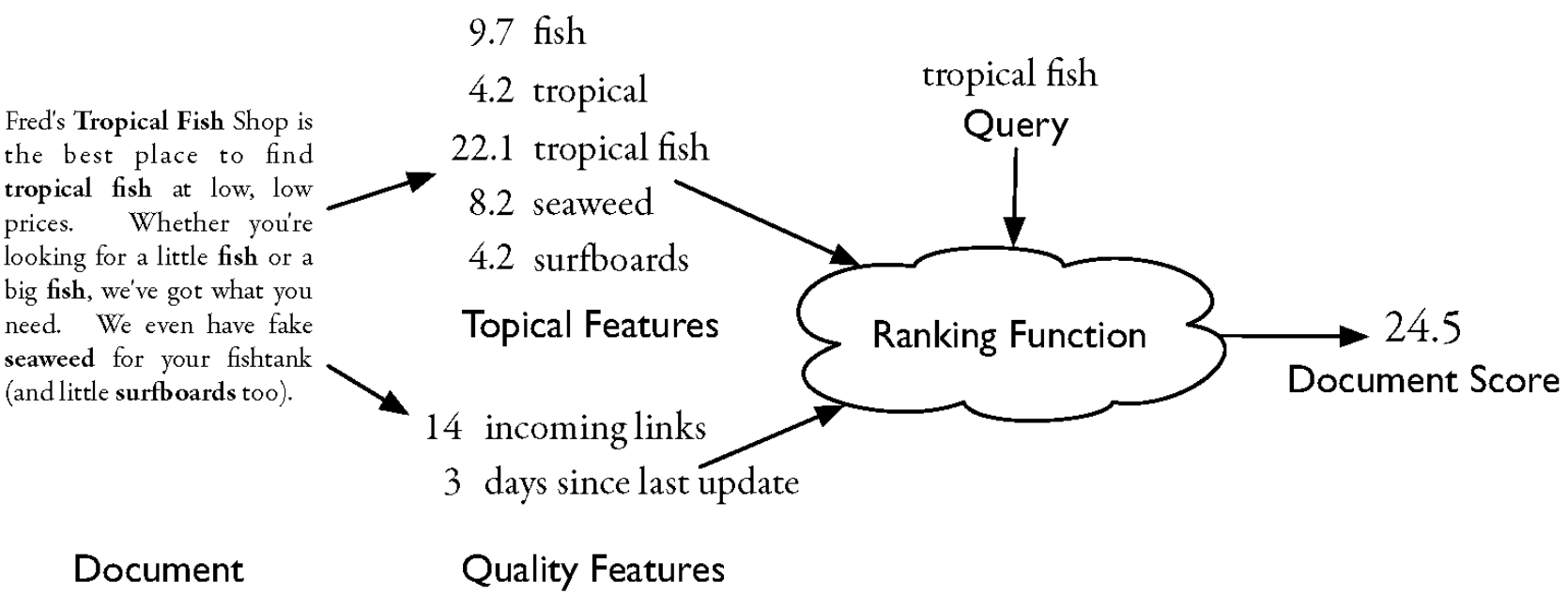
Indexes

- Indexes are data structures designed to make search faster
- Text search has unique requirements, which leads to unique data structures
- Most common data structure is inverted index
  - general name for a class of structures
  - “inverted” because documents are associated with words, rather than words with documents

Indexes and Ranking

- Indexes are designed to support search
  - faster response time, supports updates
- 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
- What is a reasonable abstract model for ranking?
  - enables discussion of indexes without details of retrieval model

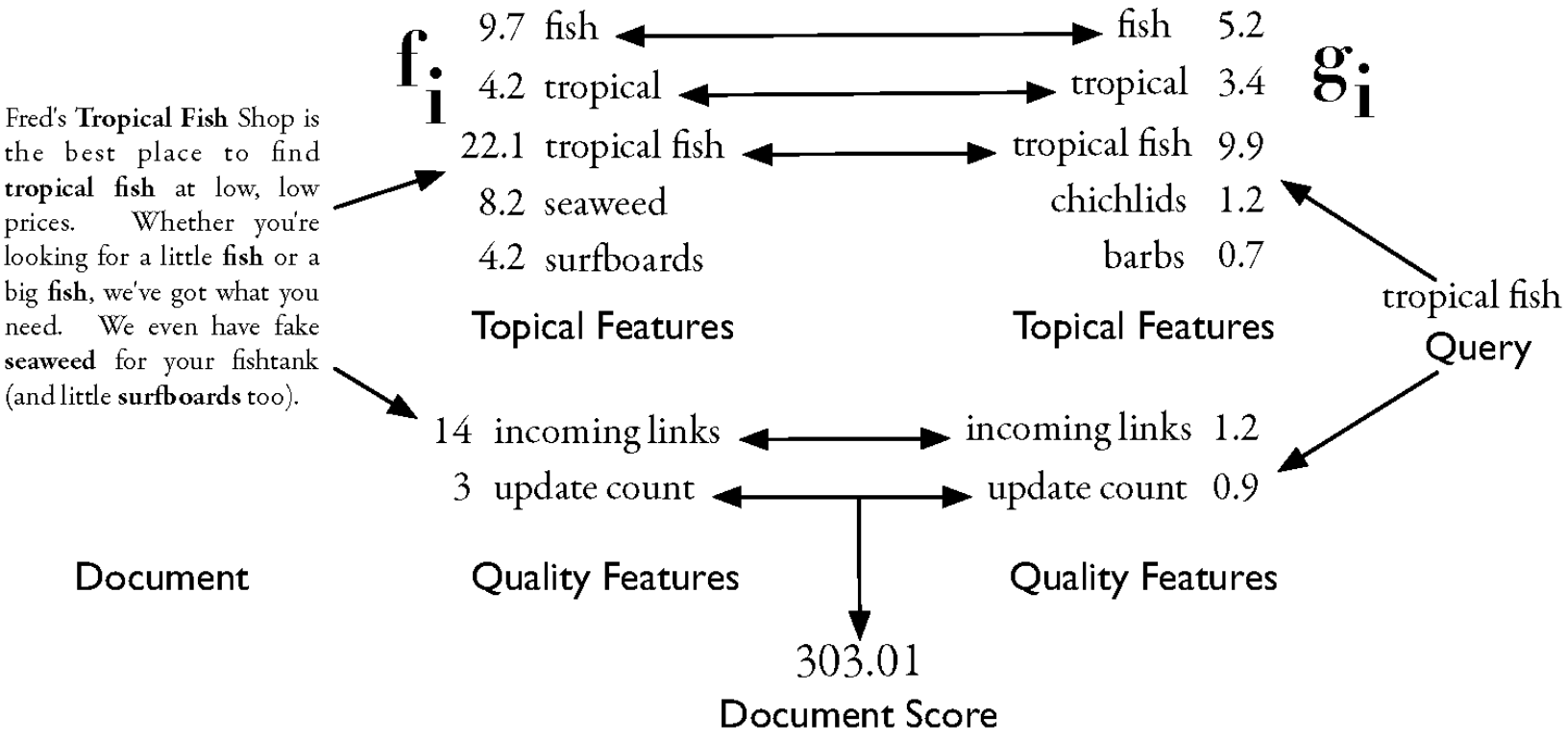
Abstract Model of Ranking



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)

Example "Collection"

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

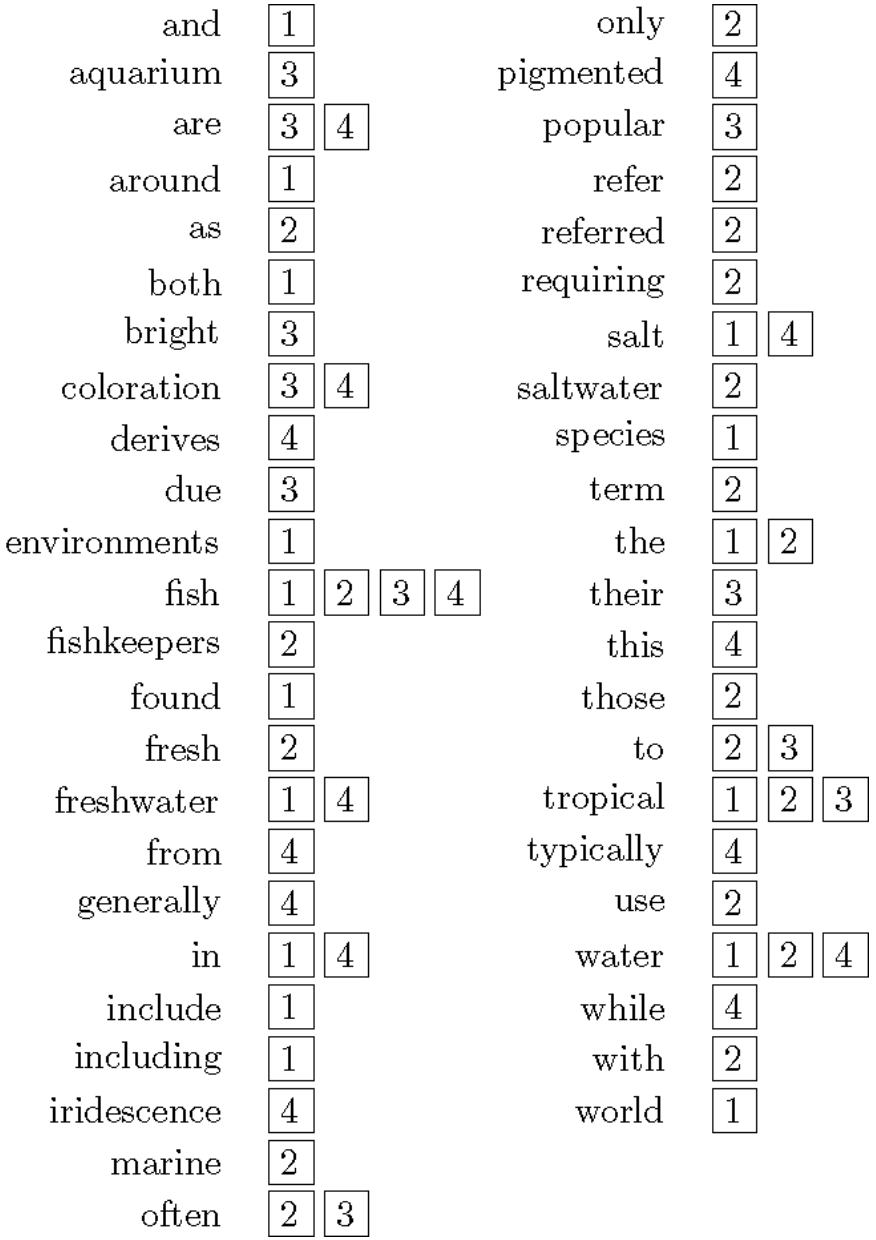
S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

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.

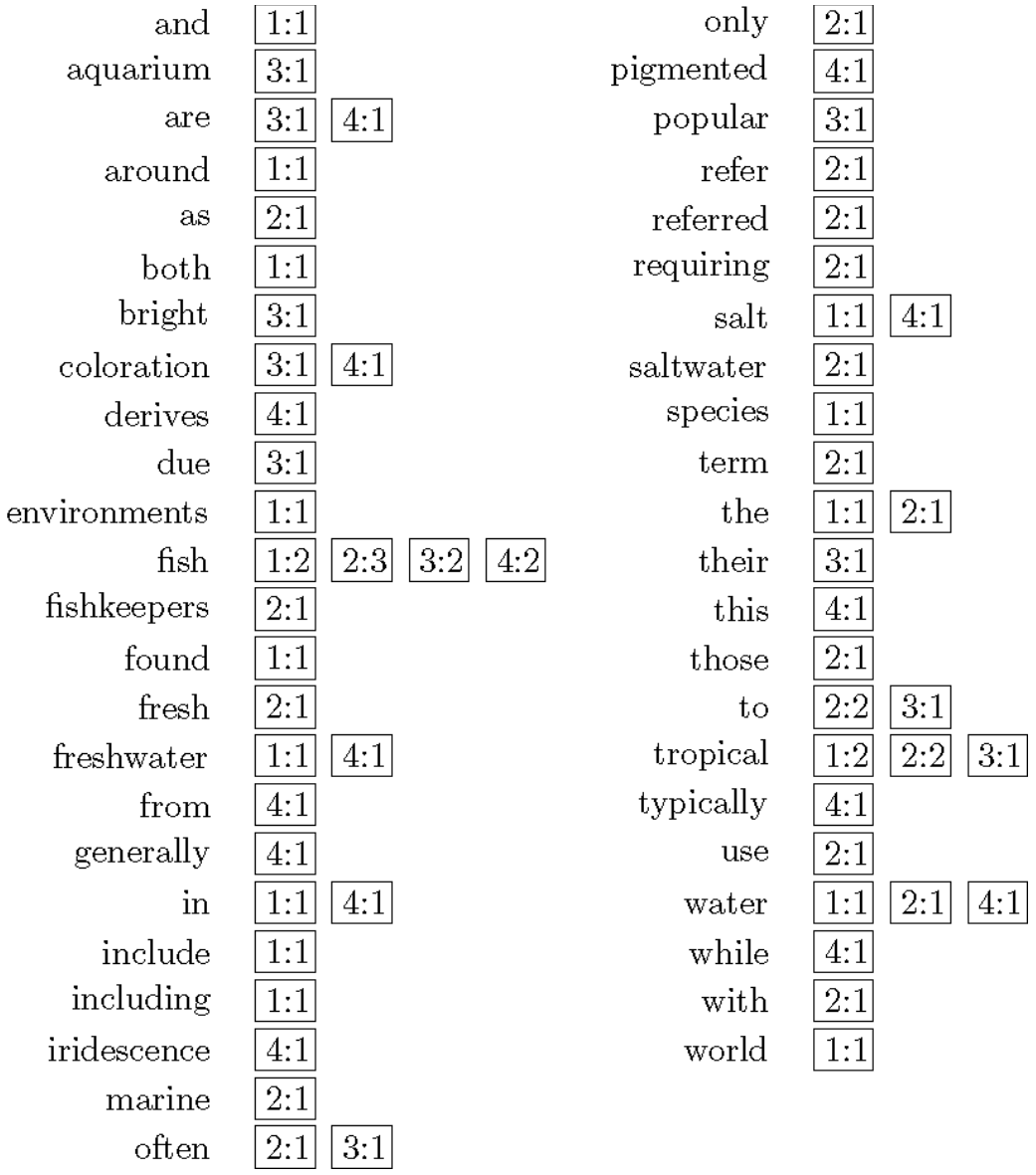
Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index



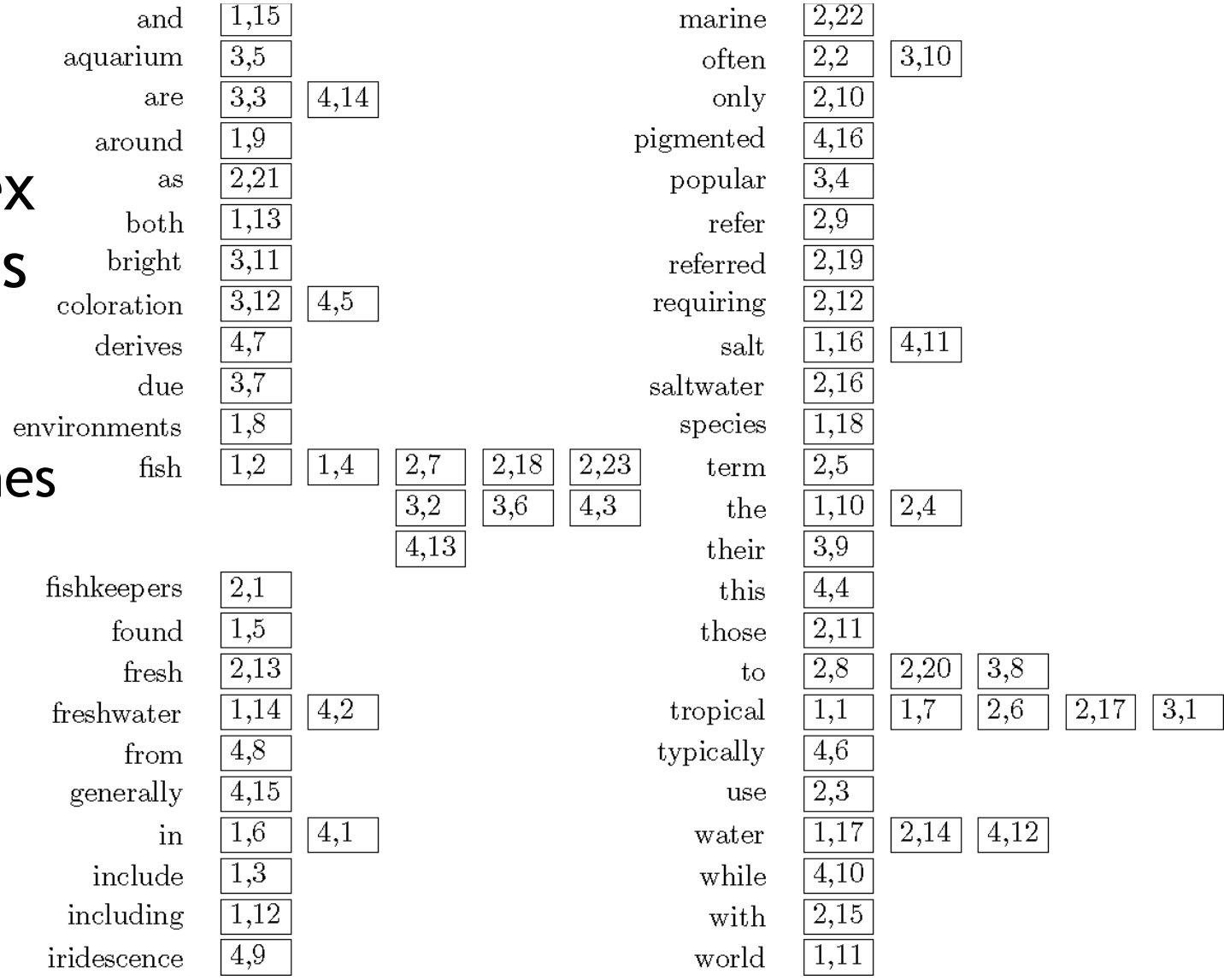
Inverted Index with counts

- supports better ranking algorithms



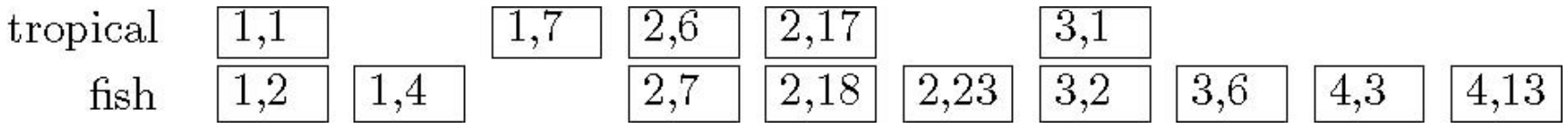
Inverted Index with positions

- supports proximity matches



Proximity Matches

- Matching phrases or words within a window
  - e.g., "tropical fish", or “find tropical within 5 words of fish”
- Word positions in inverted lists make these types of query features efficient
  - e.g.,

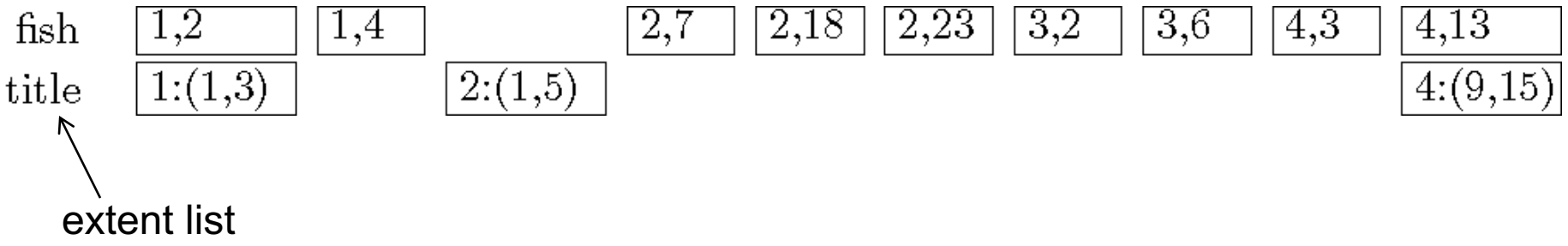


Fields and Extents

- Document structure is useful in search
  - field restrictions
    - **e.g., date, from:, etc.**
  - some fields more important
    - **e.g., title**
- Options:
  - separate inverted lists for each field type
  - add information about fields to postings
  - 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.,



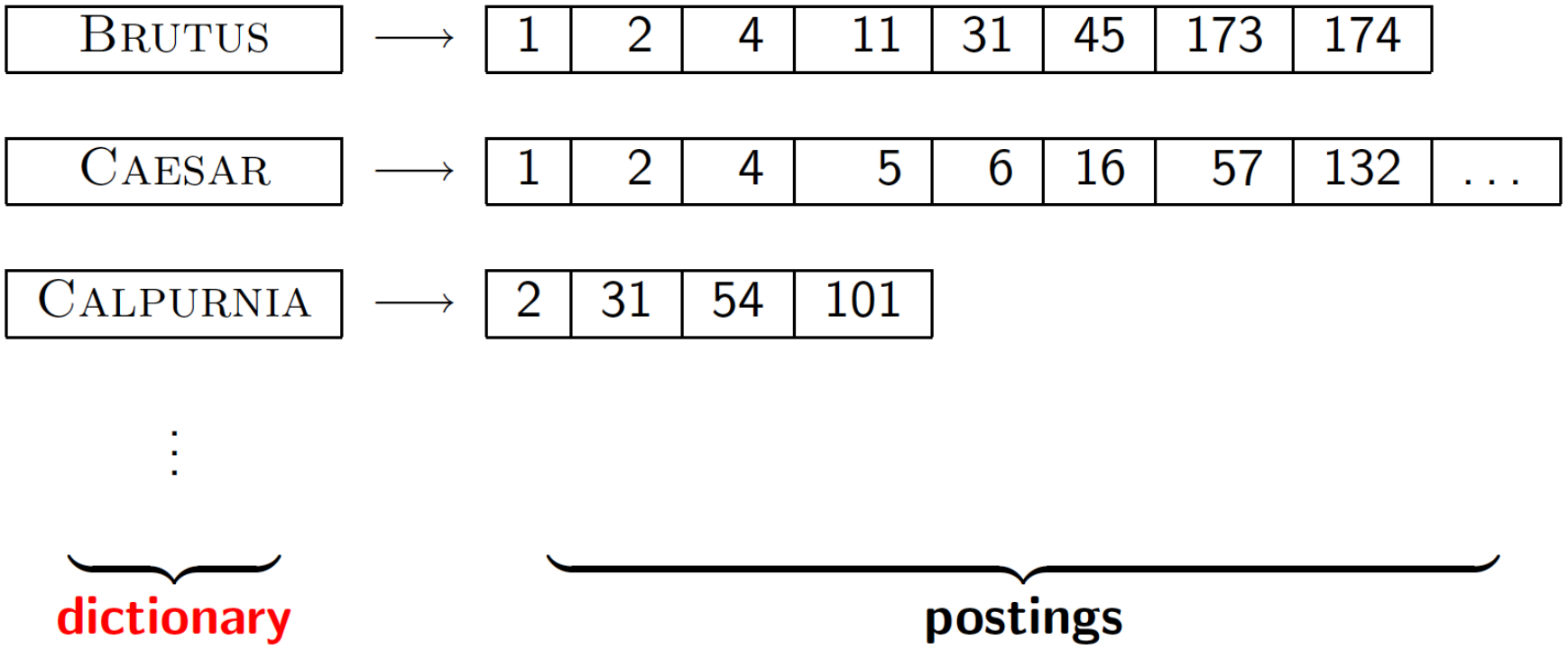
Other Issues

- 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
- Score-ordered lists
  - 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

Data Structures

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?



A naïve dictionary

- An array of struct:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

char[20] int Postings*

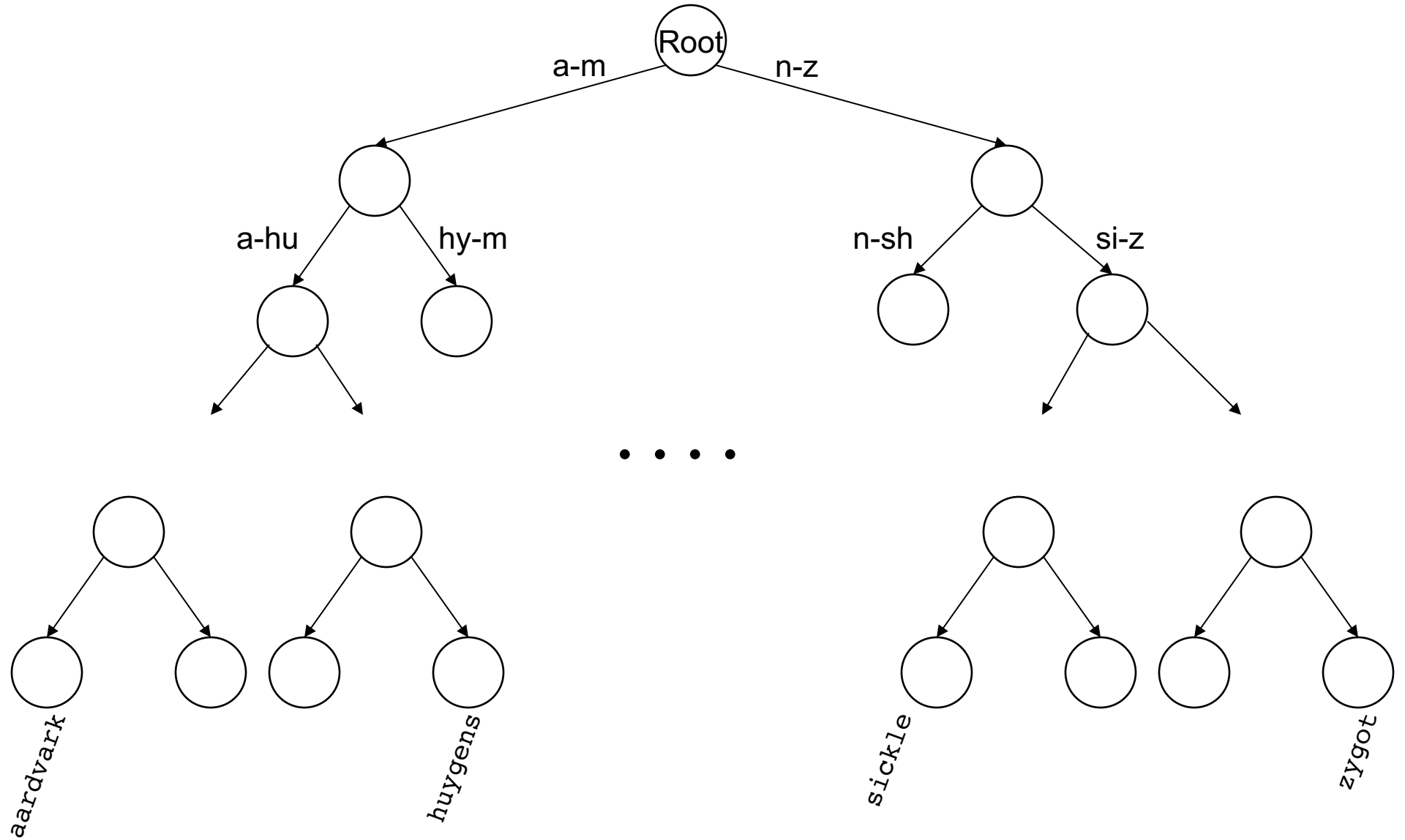
20 bytes 4/8 bytes 4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

Dictionary data structures

- Two main choices:
  - Tree (binary tree, B-tree, Patricia tree, Trie,..)
  - Hash table
- Some IR systems use trees, some hashes

Binary Search Tree (BST)

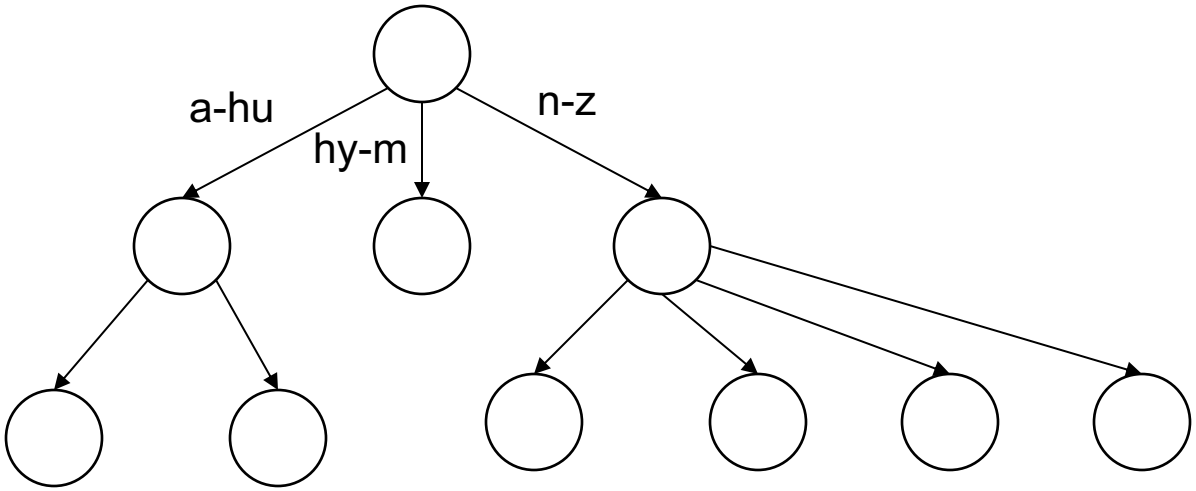


Balanced BST

- BST requires O($\log_2$ n) time in the average case, but..
  - 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 nodel has a number of children in the interval [a,b] where a, b are appropriate natural numbers, e.g., [2,4].

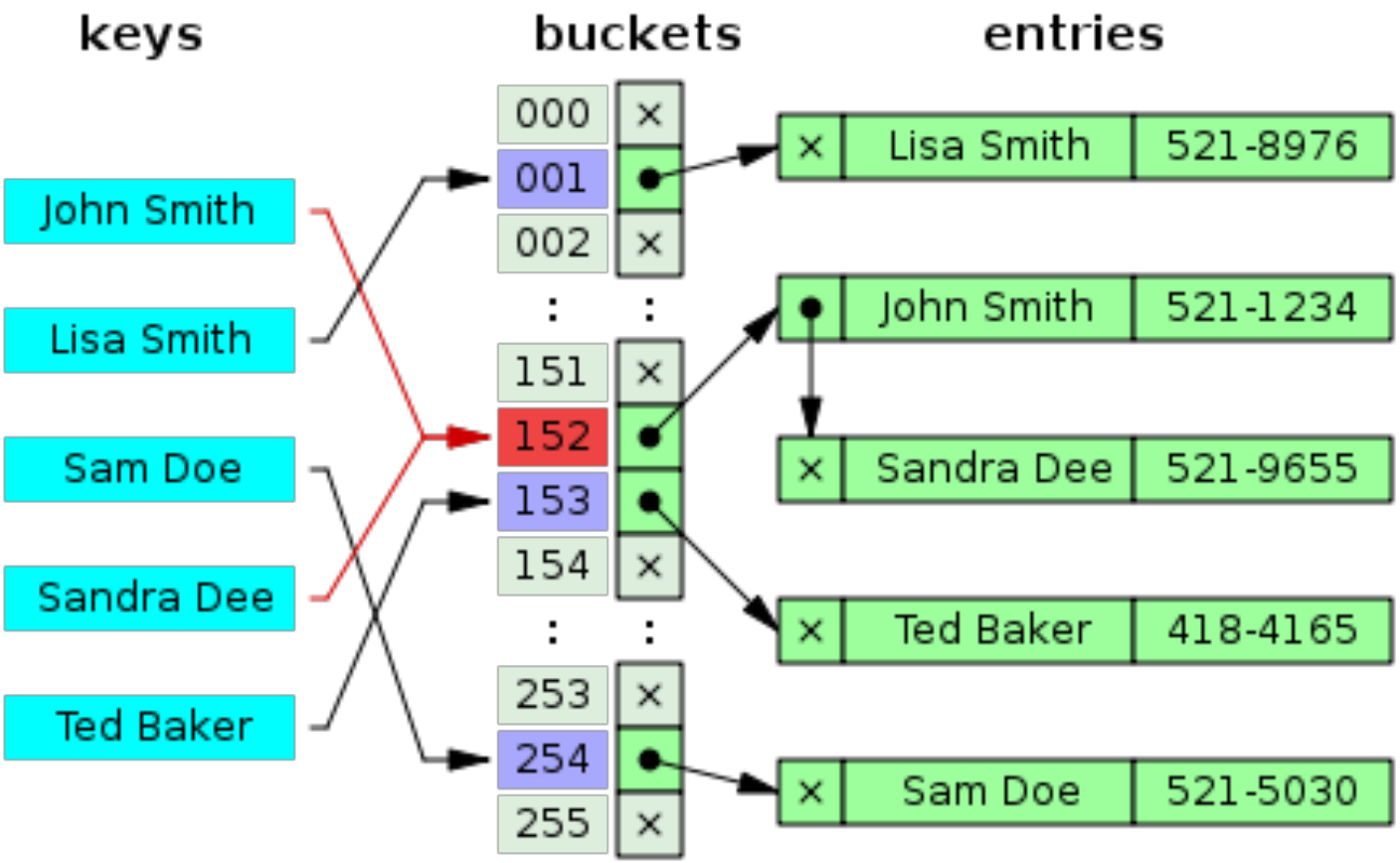


Trees

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

- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: $O(\log_2 N)$ [and this requires *balanced* tree]
  - Rebalancing binary 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 easy way to find minor variants:
    - **judgment/judgement**
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Index Construction

Index construction

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

Index construction

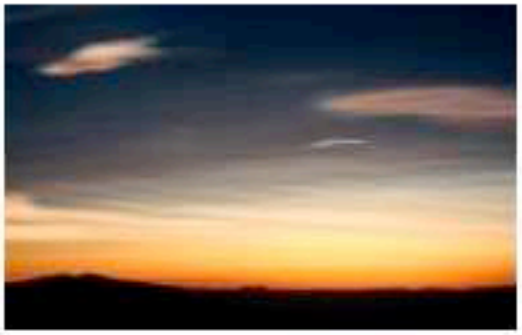
- Many design decisions in information retrieval are based on the characteristics of hardware
  - Access to data in memory is ***much*** faster than on disk
  - Disk seeks / latency (with mechanical drives)
  - Disk I/O is block-based
  - Reading one large chunk of data to memory is faster than reading 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

Reuters RCV1 collection

- Well-known collection in IR
- 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



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
	tokens	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	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

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 *(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 today, but typical collections are much larger. E.g. the *New York Times* provides an index of >150 years of newswire

Scaling index construction

- In-memory index construction does not scale
- 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 (Sorting with fewer disk seeks)

- 12-byte (4+4+4) records *(termID, 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

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

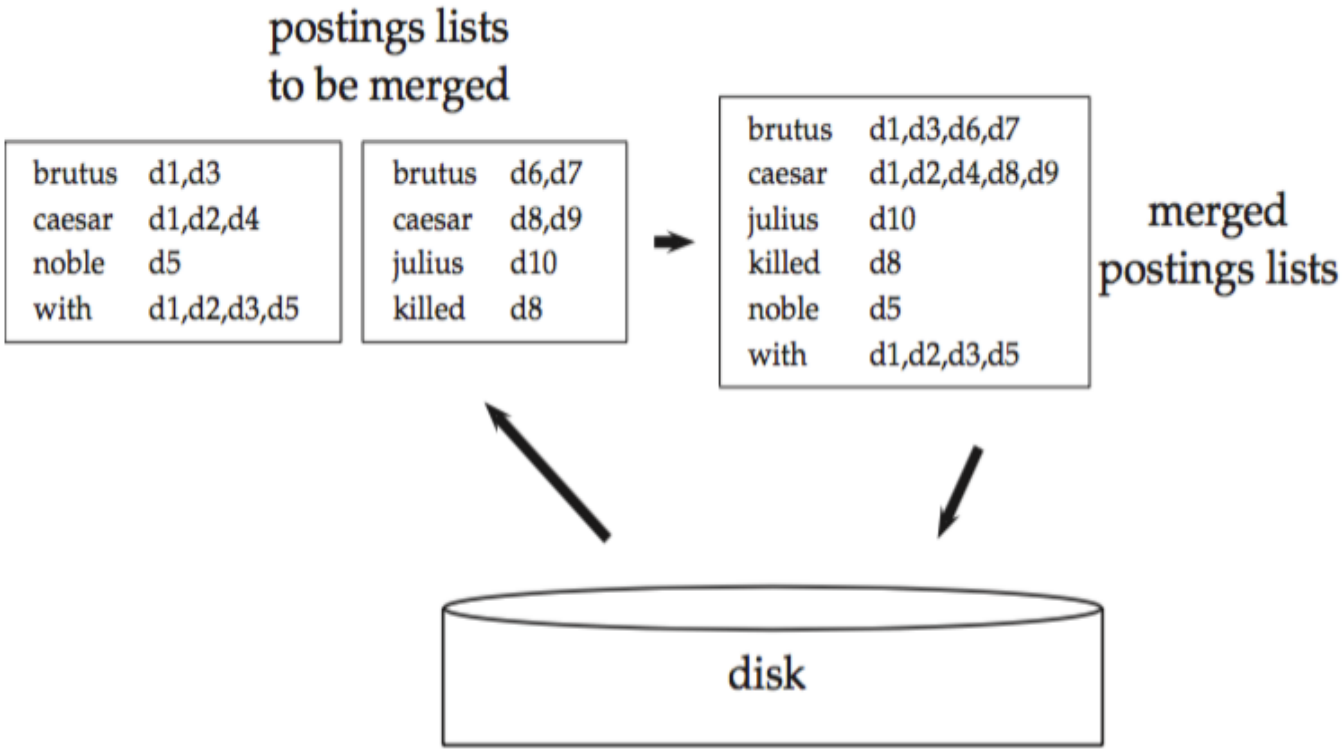
- Terms represented by TermID instead of strings
  - Unique serial number (*int*)
  - Build mapping
    - One-pass, while processing the collection
    - Two-pass approach: compile vocabulary, then construct inverted index
  - Dictionary kept in memory

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)
```

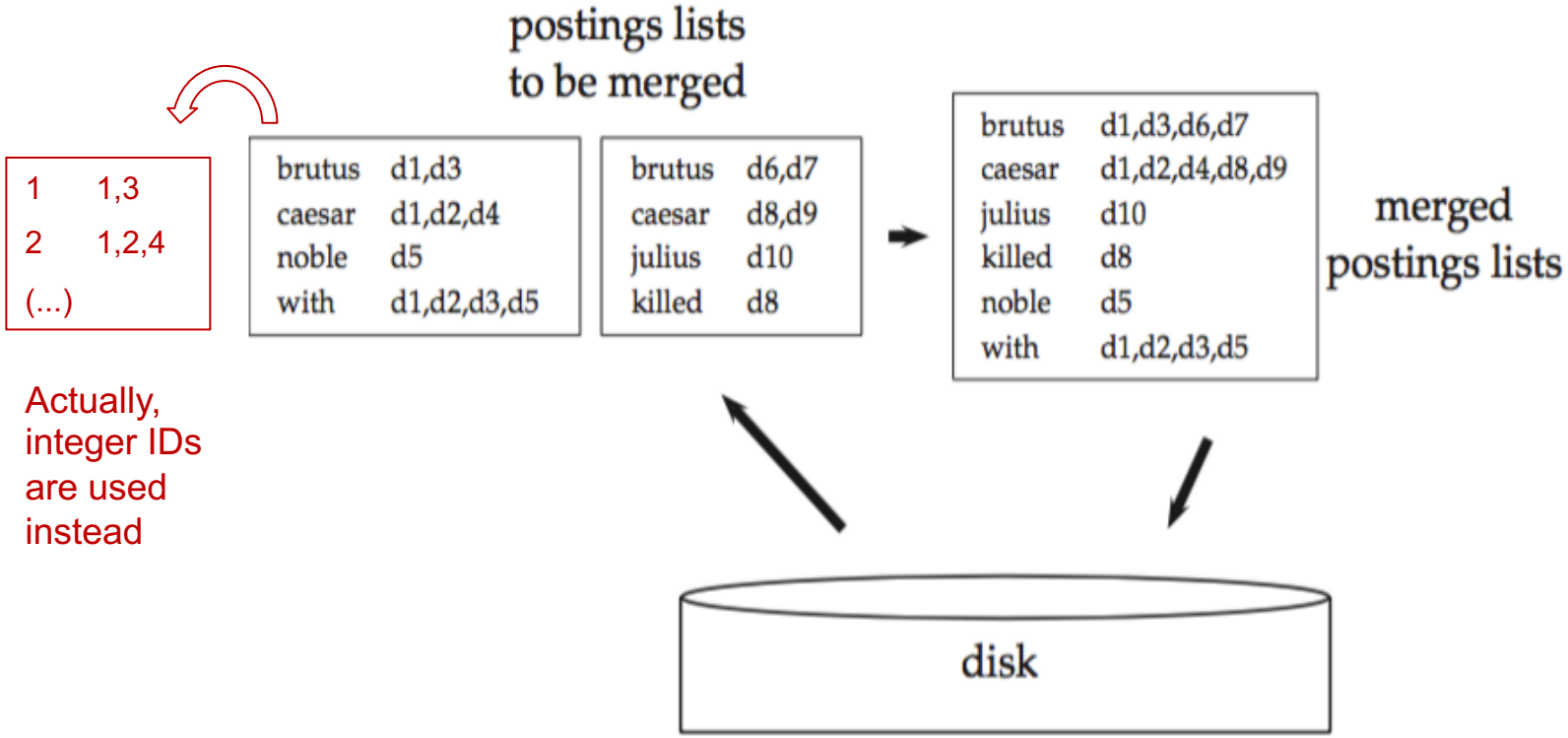
How to merge the sorted runs?

- 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



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.



How to merge the sorted runs?

- It is more efficient to do a n-way merge, where you are reading from all blocks simultaneously

- 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 very large. (We would end up with a scalable, but very slow index construction method.)

SPIMI: Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks
- Key idea 2: 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-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.

Distributed indexing

- For web-scale indexing
  - must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

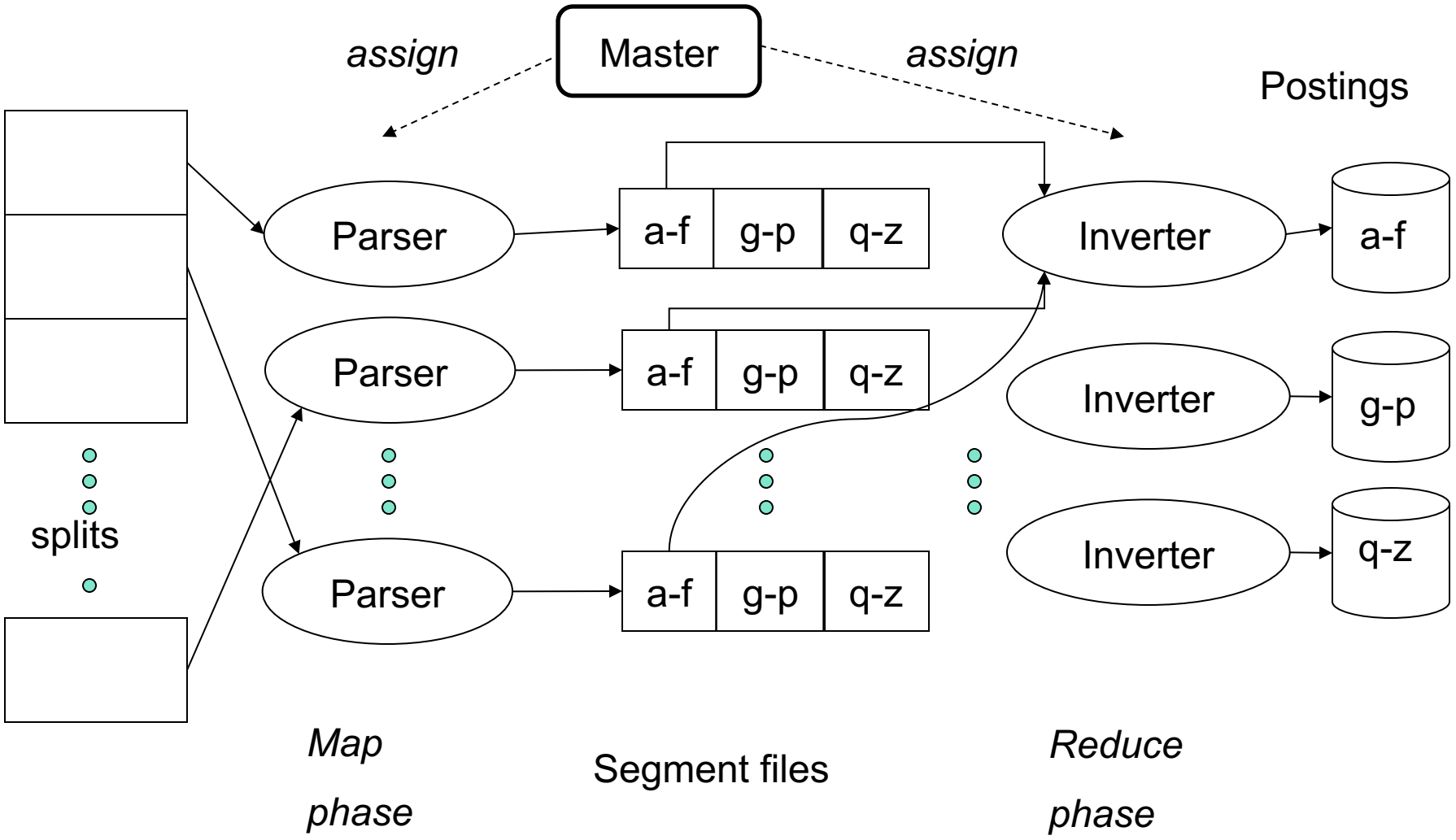
Distributed indexing

- 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)

Data flow

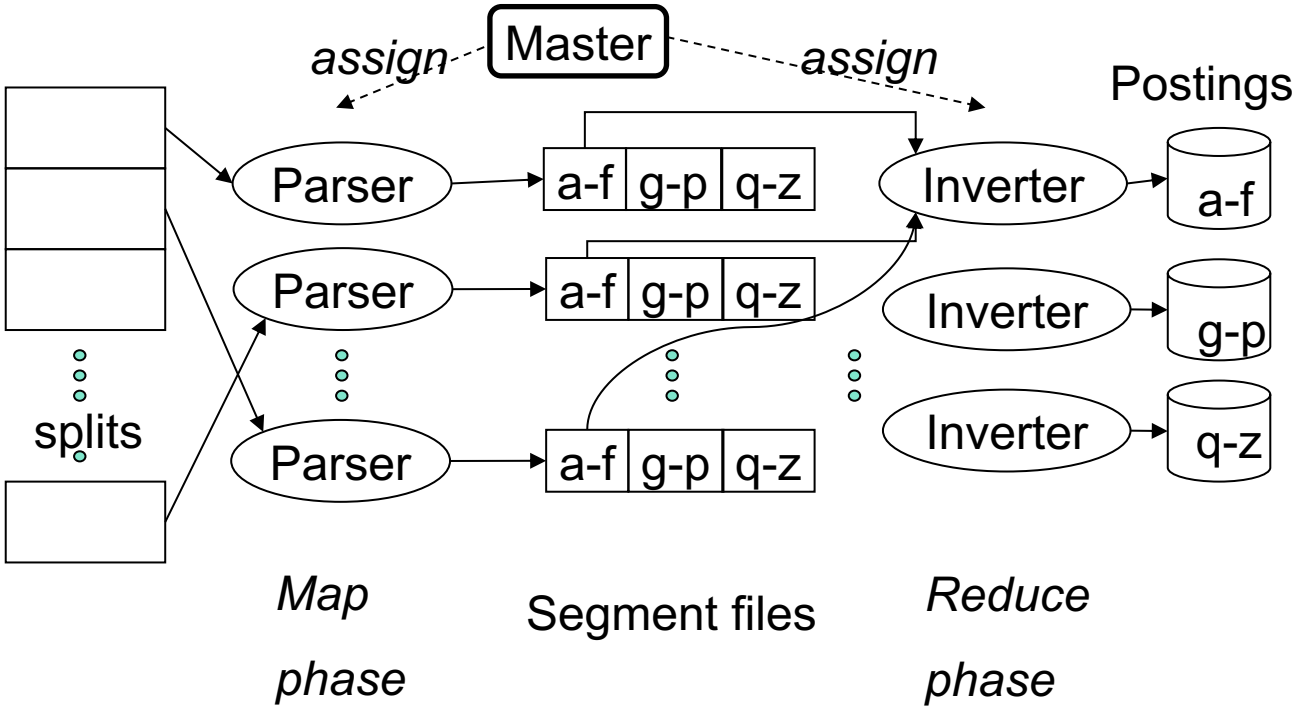

assign
Master
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map
phase
Segment files
Reduce
phase

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



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.
- They describe the Google indexing system as consisting of a number of phases, each implemented in MapReduce

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(termID, docID)

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

Example index construction in MapReduce

- **Map**

d1 : C came, C c'ed.

d2 : C died.

→

<C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>

- **Reduce**

(<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>)

→

(<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

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 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 Z1
- 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

- 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