Lecture 3: Index Representation and Tolerant Retrieval

Information Retrieval
Computer Science Tripos Part II

Simone Teufel

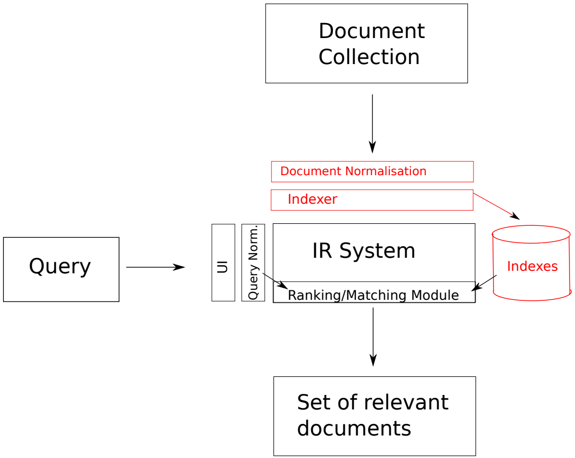
Natural Language and Information Processing (NLIP) Group
UNIVERSITY OF CAMBRIDGE

Simone.Teufel@cl.cam.ac.uk

Overview

1. Recap
2. Reuters RCV1 and Heap's Law
3. Dictionaries
4. Wildcard queries
5. Spelling correction
6. Distributed Index construction
   - BSBI algorithm
   - SPIMI and MapReduce

IR System components



Last time: The indexer

Type/token distinction

- Token an instance of a word or term occurring in a document
- Type an equivalence class of tokens

> In June, the dog likes to chase the cat in the barn.

- 12 word tokens
- 9 word types

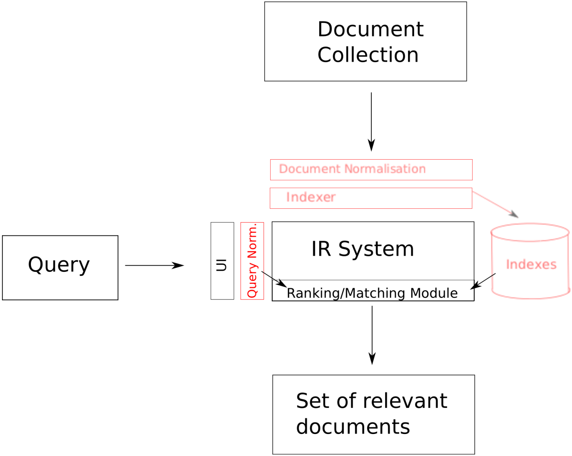
Problems with equivalence classing

- A term is an equivalence class of tokens.
- How do we define equivalence classes?
- Numbers (3/20/91 vs. 20/3/91)
- Case folding
- Stemming, Porter stemmer
- Morphological analysis: inflectional vs. derivational
- Equivalence classing problems in other languages

Positional indexes

- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions
- Example query: "to_1 be_2 or_3 not_4 to_5 be_6"
- With a positional index, we can answer
  - phrase queries
  - proximity queries

IR System components



Today: more indexing, some query normalisation

Upcoming

- Reuters RCV1 collection
- Tolerant retrieval: What to do if there is no exact match between query term and document term
- Data structures for dictionaries
  - Hashes
  - Trees
  - k-term index
  - Permuterm index
- Spelling correction
- Algorithms for large-scale indexing
  - BSBI; SPIMI
  - MapReduce

Overview

RCV1 collection

- Shakespeare's collected works are not large enough to demonstrate scalable index construction algorithms.
- Instead, we will use the Reuters RCV1 collection.
- English newswire articles published in a 12 month period (1995/6)

N	documents	800,000
M	terms (= word types)	400,000
T	non-positional postings	100,000,000

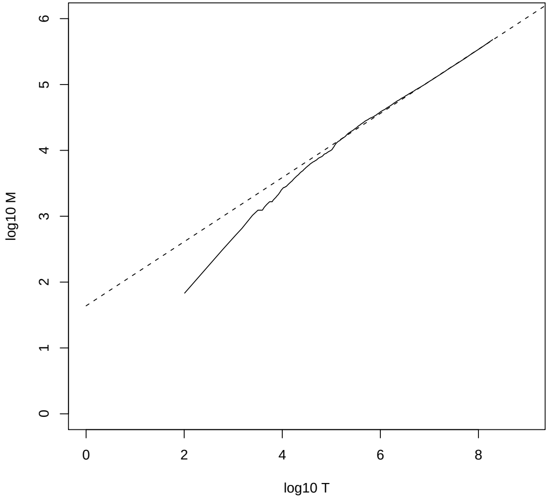
Effect of preprocessing for Reuters

size of	word types (terms)			non-positional postings			positional postings (word tokens)		
	dictionary			non-positional index			positional index		
	size	Δ	cml	size	Δ	cml	size	Δ	cml
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2	-2	100,680,242	-8	-8	179,158,204	-9	-9
case folding	391,523	-17	-19	96,969,056	-3	-12	179,158,204	-0	-9
30 stopw's	391,493	-0	-19	83,390,443	-14	-24	121,857,825	-31	-38
150 stopw's	391,373	-0	-19	67,001,847	-30	-39	94,516,599	-47	-52
stemming	322,383	-17	-33	63,812,300	-4	-42	94,516,599	-0	-52

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with collection size.
- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps' law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
  - Empirical law

Heaps' law for Reuters



Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit. Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.

Overview

Inverted Index

Brutus 8 → 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

Caesar 9 → 1 → 2 → 4 → 5 → 6 → 16 → 57 → 132 → 179

Calpurnia 4 → 2 → 31 → 54 → 101

Dictionaries

- The dictionary is the data structure for storing the term vocabulary.
- Term vocabulary: the data
- Dictionary: the data structure for storing the term vocabulary

Dictionaries

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list

How do we look up a query term q_i in the dictionary at query time?

Data structures for looking up terms

- Two main classes of data structures: hashes and trees
- Some IR systems use hashes, some use trees.
- Criteria for when to use hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?

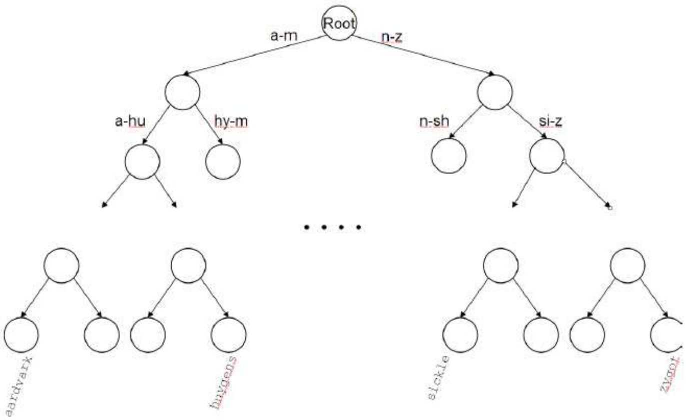
Hashes

- Each vocabulary term is hashed into an integer, its row number in the array
- At query time: hash query term, locate entry in fixed-width array
- Pros: Lookup in a hash is faster than lookup in a tree. (Lookup time is constant.)
- Cons
  - no way to find minor variants (resume vs. résumé)
  - no prefix search (all terms starting with automat)
  - need to rehash everything periodically if vocabulary keeps growing

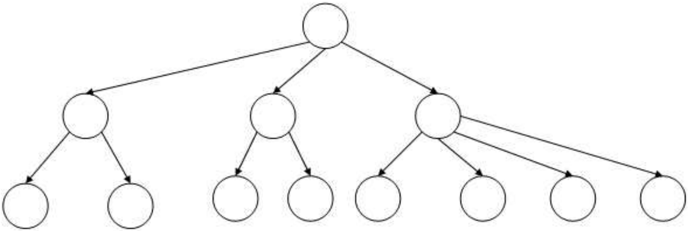
Trees

- Trees solve the prefix problem (find all terms starting with automat).
- Simplest tree: binary tree
- Search is slightly slower than in hashes: O(logM), where M is the size of the vocabulary.
- O(logM) only holds for balanced trees.
- Rebalancing binary trees is expensive.
- B-trees mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval [a, b] where a, b are appropriate positive integers, e.g., [2, 4].

Binary tree



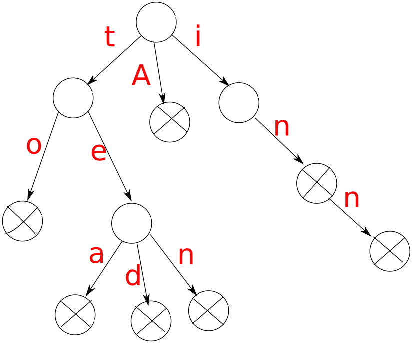
B-tree



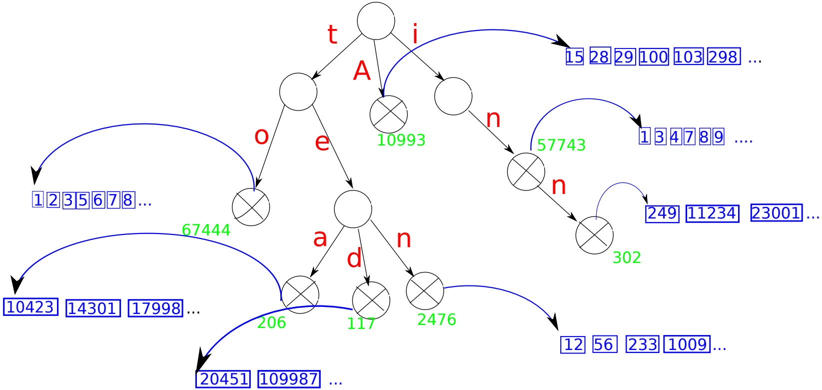
Trie

- An ordered tree data structure that is used to store an associative array
- The keys are strings
- The key associated with a node is inferred from the position of a node in the tree
  - Unlike in binary search trees, where keys are stored in nodes.
- Values are associated only with with leaves and some inner nodes that correspond to keys of interest (not all nodes).
- All descendants of a node have a common prefix of the string associated with that node $\rightarrow$ tries can be searched by prefixes
- The trie is sometimes called radix tree or prefix tree

Trie



A trie for keys "A", "to", "tea", "ted", "ten", "in", and "inn".


t
i
A
15 28 29 100 103 298 ...
10993
o
e
n
57743
1 3 4 7 8 9 ....
1 2 3 5 6 7 8 ...
67444
n
249 11234 23001 ...
a
d
n
302
10423 14301 17998 ...
206
117
2476
12 56 233 1009 ...
20451 109987 ...

Overview

Wildcard queries

hel*

- Find all docs containing any term beginning with "hel"
- Easy with trie: follow letters h-e-l and then lookup every term you find there

*hel

- Find all docs containing any term ending with "hel"
- Maintain an additional trie for terms backwards
- Then retrieve all terms t in subtree rooted at l-e-h

In both cases:

- This procedure gives us a set of terms that are matches for wildcard query
- Then retrieve documents that contain any of these terms

How to handle * in the middle of a term

hel*o

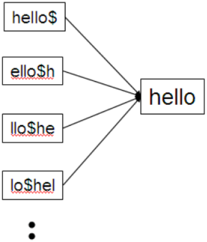
- We could look up "hel*" and "*o" in the tries as before and intersect the two term sets.
  - Expensive
- Alternative: permuterm index
- Basic idea: Rotate every wildcard query, so that the * occurs at the end.
- Store each of these rotations in the dictionary (trie)

Permuterm index

For term hello: add

hello$, ello$h, llohe, lohel, o$hell, $hello

to the trie where $ is a special symbol



for hel*o, look up o$hel*

Problem: Permuterm more than quadrupels the size of the dictionary compared to normal trie (empirical number).

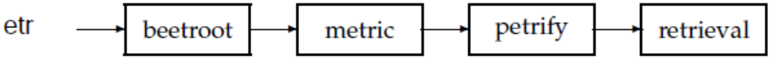
k-gram indexes

- More space-efficient than permuterm index
- Enumerate all character k-grams (sequence of k characters) occurring in a term

Bi-grams from April is the cruelest month

ap pr ri il l$ $i is s$ $t th he e$ $c cr ru ue el le es st t$ $m mo on nt th h$

- Maintain an inverted index from k-grams to the term that contain the k-gram



k-gram indexes

Note that we have two different kinds of inverted indexes:

- The term-document inverted index for finding documents based on a query consisting of terms
- The k-gram index for finding terms based on a query consisting of k-grams

Processing wildcard terms in a bigram index

- Query hel* can now be run as:

  $h AND he AND el

- ... but this will show up many false positives like heel.
- Postfilter, then look up surviving terms in term–document inverted index.
- k-gram vs. permuterm index
  - k-gram index is more space-efficient
  - permuterm index does not require postfiltering.

Overview

Spelling correction

an asterorid that fell form the sky

- In an IR system, spelling correction is only ever run on queries.
- The general philosophy in IR is: don't change the documents (exception: OCR'ed documents)
- Two different methods for spelling correction:
  - Isolated word spelling correction
    - Check each word on its own for misspelling
    - Will only attempt to catch first typo above
  - Context-sensitive spelling correction
    - Look at surrounding words
    - Should correct both typos above

Isolated word spelling correction

- There is a list of "correct" words – for instance a standard dictionary (Webster's, OED...)
- Then we need a way of computing the distance between a misspelled word and a correct word
  - for instance Edit/Levenshtein distance
  - k-gram overlap
- Return the "correct" word that has the smallest distance to the misspelled word.

informaton $\rightarrow$ information

Edit distance

- **Edit distance** between two strings s_1 and s_2 is the minimum number of basic operations that transform s_1 into s_2.
- **Levenshtein distance:** Admissible operations are insert, delete and replace

Levenshtein distance

dog	–	do	1 (delete)
cat	–	cart	1 (insert)
cat	–	cut	1 (replace)
cat	–	act	2 (delete+insert)

Levenshtein distance: Distance matrix

		s	n	o	w
	0	1	2	3	4
o	1	1	2	3	4
s	2	1	3	3	3
l	3	3	2	3	4
o	4	3	3	2	3

Edit Distance: Four cells

			s		n		o		w	
		0	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**
o		**1**	**1**	2	**2**	3	**2**	4	4	5
		1	2	**1**	**2**	**2**	3	**2**	**3**	**3**
s		**2**	**1**	2	**2**	3	**3**	**3**	**3**	4
		2	3	**1**	**2**	**2**	**3**	**3**	4	**3**
l		**3**	3	**2**	**2**	3	**3**	4	**4**	**4**
		3	4	**2**	3	**2**	**3**	**3**	**4**	**4**
o		**4**	4	**3**	**3**	**3**	**2**	4	4	5
		4	5	**3**	4	**3**	4	**2**	**3**	**3**

Each cell of Levenshtein matrix

Cost of getting here from my upper left neighbour (by copy or replace)	Cost of getting here from my upper neighbour (by delete)
Cost of getting here from my left neighbour (by insert)	Minimum cost out of these

Dynamic Programming

Cormen et al:

- Optimal substructure: The optimal solution contains within it subsolutions, i.e, optimal solutions to subproblems
- Overlapping subsolutions: The subsolutions overlap and would be computed over and over again by a brute-force algorithm.

For edit distance:

- Subproblem: edit distance of two prefixes
- Overlap: most distances of prefixes are needed 3 times (when moving right, diagonally, down in the matrix)

Example: Edit Distance OSLO – SNOW

			s		n		o		w	
		0	1	1	2	2	3	3	4	4
o		1	1	2	2	3	2	4	4	5
		1	2	1	2	2	3	2	3	3
s		2	1	2	2	3	3	3	3	4
		2	3	1	2	2	3	3	4	3
l		3	3	2	2	3	3	4	4	4
		3	4	2	3	2	3	3	4	4
o		4	4	3	3	3	2	4	4	5
		4	5	3	4	3	4	2	3	3

Edit distance OSLO–SNOW is 3! How do I read out the editing operations that transform OSLO into SNOW?

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n

Using edit distance for spelling correction

- Given a query, enumerate all character sequences within a preset edit distance
- Intersect this list with our list of “correct” words
- Suggest terms in the intersection to user.

k-gram indexes for spelling correction

- Enumerate all k-grams in the query term

Misspelled word bordroom

bo – or – rd – dr – ro – oo – om

- Use k-gram index to retrieve "correct" words that match query term k-grams
- Threshold by number of matching k-grams
- Eg. only vocabularly terms that differ by at most 3 k-grams

BO → aboard → about → boardroom → border

OR → border → lord → morbid → sordid

RD → aboard → ardent → boardroom → border

Context-sensitive Spelling correction

One idea: hit-based spelling correction

flew form munich

- Retrieve correct terms close to each query term

flew → flea
form → from
munich → munch

- Holding all other terms fixed, try all possible phrase queries for each replacement candidate

flea form munich – 62 results
flew from munich –78900 results
flew form munch – 66 results

Not efficient. Better source of information: large corpus of queries, not documents

General issues in spelling correction

- User interface
  - automatic vs. suggested correction
  - "Did you mean" only works for one suggestion; what about multiple possible corrections?
  - Tradeoff: Simple UI vs. powerful UI
- Cost
  - Potentially very expensive
  - Avoid running on every query
  - Maybe just those that match few documents

Overview

Hardware basics

- Access to data is much faster in memory than on disk. (roughly a factor of 10)
- Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
- To optimize transfer time from disk to memory: one large chunk is faster than 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 many GBs of main memory and TBs of disk space.
- Fault tolerance is expensive: It's cheaper to use many regular machines than one fault tolerant machine.

Goal: construct the inverted index

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 |

$\vdots$

$\underbrace{\hspace{3em}}_{\textbf{dictionary}}$ $\underbrace{\hspace{12em}}_{\textbf{postings}}$

Index construction: Sort postings in memory

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

$\Longrightarrow$

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

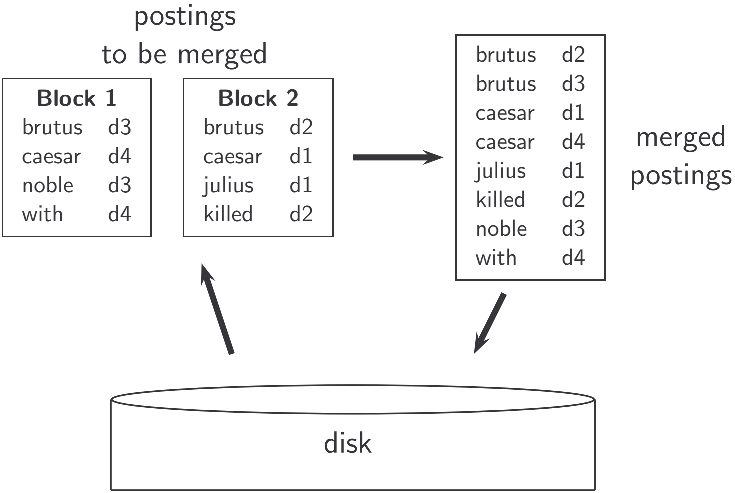
Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- But for large collections, we cannot keep all postings in memory and do the sort in-memory at the end
- We cannot sort very large sets of records on disk either (too many disk seeks)
- Thus: We need to store intermediate results on disk.
- We need an external sorting algorithm.

“External” sorting algorithm (using few disk seeks)

- We must sort $T = 100{,}000{,}000$ non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, term frequency).
- 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 BSBI algorithm:
  - For each block:
    - accumulate postings
    - sort in memory
    - write to disk
  - Then merge the blocks into one long sorted order.

Merging two blocks



Problem with BSBI 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.

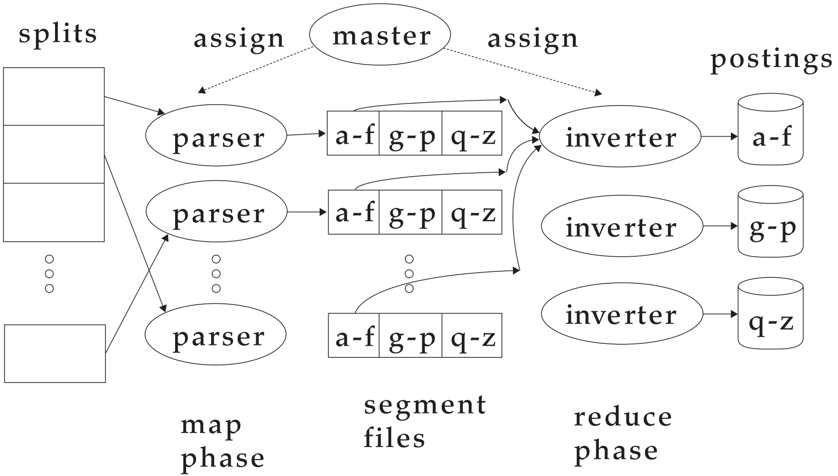
Single-pass in-memory indexing

- Abbreviation: SPIMI
- 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.

Distributed indexing– fault-tolerant indexing

- Maintain a master machine directing the indexing job – considered "safe"
- Break up indexing into sets of parallel tasks
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Master machine assigns each task to an idle machine from a pool.
- There are two sets of parallel tasks, and two types of machines are deployed to solve them:
  - Parser: reads a document at a time and emits (term,docID)-pairs. Writes these pairs into j term-partition; e.g., a-f, g-p, q-z (here: $j = 3$).
  - Inverter: collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f), sorts them and writes to postings lists

MapReduce


splits
assign
master
assign
postings
parser
a-f
g-p
q-z
inverter
a-f
parser
a-f
g-p
q-z
inverter
g-p
parser
a-f
g-p
q-z
inverter
q-z
map
phase
segment
files
reduce
phase

Schema of map and reduce functions

map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output

Instantiation of the schema for index construction

map:	web collection	$\rightarrow$ list(termID, docID)
reduce:	$(\langle \text{termID}_1, \text{list(docID)}\rangle, \langle \text{termID}_2, \text{list(docID)}\rangle, \ldots)$	$\rightarrow$ $(\text{postings\_list}_1, \text{postings\_list}_2, \ldots)$

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow (\langle \text{C}, d_2\rangle, \langle \text{DIED}, d_2\rangle, \langle \text{C}, d_1\rangle, \langle \text{CAME}, d_1\rangle, \langle \text{C}, d_1\rangle, \langle \text{C'ED}, d_1\rangle)$
reduce:	$(\langle \text{C}, (d_2, d_1, d_1)\rangle, \langle \text{DIED}, (d_2)\rangle, \langle \text{CAME}, (d_1)\rangle, \langle \text{C'ED}, (d_1)\rangle)$	$\rightarrow (\langle \text{C}, (d_1{:}2, d_2{:}1)\rangle, \langle \text{DIED}, (d_2{:}1)\rangle, \langle \text{CAME}, (d_1{:}1)\rangle, \langle \text{C'ED}, (d_1{:}1)\rangle)$

Takeaway

- What to do if there is no exact match between query term and document term
- Datastructures for tolerant retrieval:
  - Dictionary as hash, B-tree or trie
  - k-gram index and permuterm for wildcards
  - k-gram index and edit-distance for spelling correction
- Distributed, large-scale indexing
  - BSBI and SPIMI
  - MapReduce: distributed index construction

Reading

- Wikipedia article "trie"
- MRS chapter 3.1, 3.2, 3.3
- MRS Chapters 4.2-4.4