SE 481 Introduction to Information Retrieval

Module #2 — Index Construction

College of Arts, Media and Technology
Chiang Mai University

Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

Last assignment — difficult path

```
01  parsed_description = parse_job_description()
02
03  sw_set = set(stopwords.words()) - {'c'}
04  no_sw_description = parsed_description.apply(lambda x: [w for w in x if w not in sw_set])
05  ps = PorterStemmer()
06  stemmed_description = no_sw_description.apply(lambda x: set([ps.stem(w) for w in x]))
07  all_unique_term = list(set.union(*stemmed_description.to_list()))
08
09  invert_idx = {}
10  for s in all_unique_term:
11      invert_idx[s] = set(stemmed_description.loc[stemmed_description.apply(lambda x: s in x)].index)
12
13  to_search_db = ['mongodb']
14  stemmed_db = np.unique([ps.stem(w) for w in to_search_db])
15  searched_db = set.union(*[invert_idx[s] for s in stemmed_db])
16
17  to_search_lang = ['java']
18  stemmed_lang = np.unique([ps.stem(w) for w in to_search_lang])
19  searched_lang = set.union(*[invert_idx[s] for s in stemmed_lang])
20
21  appear_both = searched_db.intersection(searched_lang)
22
23  print(parsed_description.loc[appear_both].apply(lambda x: ' '.join(x)).tail().to_markdown())
```

Preprocess = remove stop word and then stem

Create the invert index

Preprocess

Preprocess

Search will be super fast

Agenda

- Tokenization — revisit
- How do we construct an index?
- How can we save the memory usage?

Tokenization — revisit

- Example — How will we tokenize this:

Full Job Description

The Role:

Autopilot is of critical importance to Tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. As a member of Tesla's Autopilot Simulation team, you will be in a unique position to accelerate the pace at which Autopilot improves over time. The main ways in which the simulation team realizes this include:

- Some discussion questions
  - What terms to use: words, phrases, entities?
  - Which terms to include?
  - Are all terms equally important?
  - How to deal with numbers?
  - How to deal with word variations?

Tokenization — revisit

- Example — How will we tokenize this:

Full Job Description

The Role:

Autopilot is of critical importance to Tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. As a member of Tesla's Autopilot Simulation team, you will be in a unique position to accelerate the pace at which Autopilot improves over time. The main ways in which the simulation team realizes this include:

- Some observations
  - You chose an indexing unit, with a certain **specificity**
  - Some terms are more **important** than others
  - Important information is **implicit**
  - Terms can be **ambiguous**
  - How **consistent** do you think you are?

Tokenization — revisit

- Example — How will we tokenize this:

Full Job Description

The Role:

Autopilot is of critical importance to Tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. As a member of Tesla's Autopilot Simulation team, you will be in a unique position to accelerate the pace at which Autopilot improves over time. The main ways in which the simulation team realizes this include:

- Thus, we need to automate the tokenizing process.

Tokenization — revisit

Full Job Description

The Role:

Autopilot is of critical importance to Tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. As a member of Tesla's Autopilot Simulation team, you will be in a unique position to accelerate the pace at which Autopilot improves over time. The main ways in which the simulation team realizes this include:

- Tokenization
  - Lowercase text
  - Extract words
  - Stopword removal
  - Stemming

Tokenization — revisit

full job description

the role:

autopilot is of critical importance to tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. as a member of tesla's autopilot simulation team, you will be in a unique position to accelerate the pace at which autopilot improves over time. the main ways in which the simulation team realizes this include:

- Tokenization
  - **Lowercase text**
  - Extract words
  - Stopword removal
  - Stemming

Tokenization — revisit

full job description

the role:

autopilot is of critical importance to tesla's mission. It is safer, makes driving more enjoyable, and will ultimately deliver on the promise of self-driving cars. as a member of tesla's autopilot simulation team, you will be in a unique position to accelerate the pace at which autopilot improves over time. the main ways in which the simulation team realizes this include:

- Tokenization
  - Lowercase text
  - Extract words
  - Stopword removal
  - Stemming

Tokenization — revisit

full job description

~~the~~ role:

autopilot ~~is~~ ~~of~~ critical importance ~~to~~ tesla's mission. It ~~is~~ safer, makes driving more enjoyable, ~~and~~ ~~will~~ ultimately deliver ~~on~~ ~~the~~ promise ~~of~~ self-driving cars. ~~as~~ ~~a~~ member ~~of~~ tesla's autopilot simulation team, you ~~will~~ ~~be~~ ~~in~~ ~~a~~ unique position ~~to~~ accelerate ~~the~~ pace ~~at~~ ~~which~~ autopilot improves ~~over~~ time. ~~the~~ main ways ~~in~~ ~~which~~ ~~the~~ simulation team realizes ~~this~~ include:

- Tokenization
  - Lowercase text
  - Extract words
  - Stopword removal
  - Stemming

Tokenization — revisit

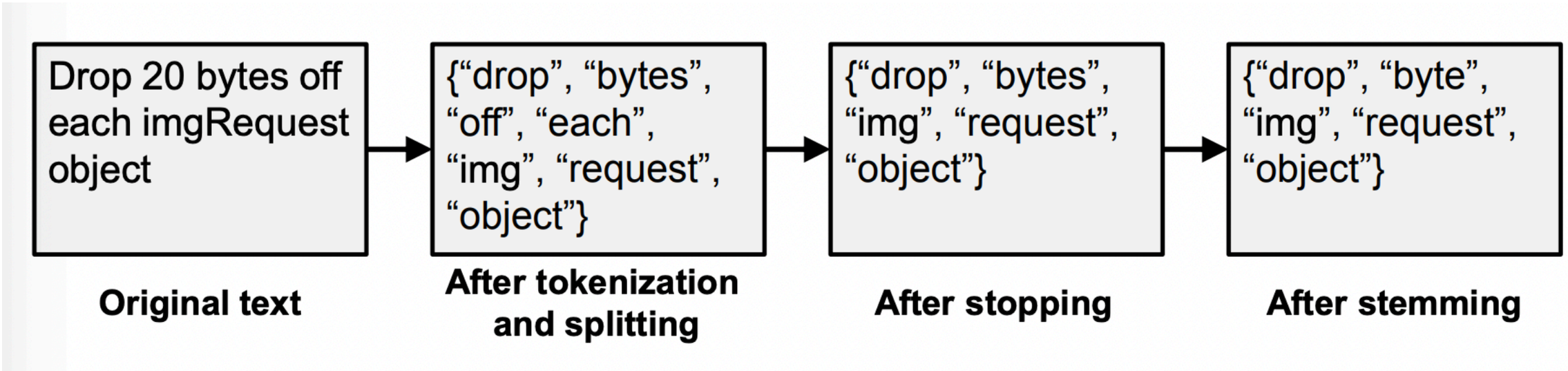
full job descript

~~**the**~~ **role:**

autopilot ~~is~~ ~~of~~ **critic import** ~~to~~ tesla's mission. It ~~is~~ **safe**, makes **driv** ~~more~~ **enjoy**, ~~and~~ ~~will~~ **ultim deliv** ~~on~~ ~~the~~ **promis** ~~of~~ self-**driv** cars. ~~as~~ ~~a~~ member ~~of~~ tesla's autopilot **simul** team, you ~~will~~ ~~be~~ ~~in~~ ~~a~~ **uniqu posit** ~~to~~ **acceler** ~~the~~ pace ~~at~~ ~~which~~ autopilot **improv** ~~over~~ time. ~~the~~ main ways ~~in~~ ~~which~~ ~~the~~ **simul** team **realiz** ~~this~~ **includ**:

- Tokenization
  - Lowercase text
  - Extract words
  - Stopword removal
  - **Stemming**

Tokenization — more example



Index construction — Revisit

- Indexer steps: Token sequence
  - Sequence of (Modified token, Document ID) pairs.

Doc 1

The chosen software developer will be part of a larger engineering team developing software for medical devices

Doc 2

we are seeking a seasoned software developer with strong analytical and technical skills to join our public sector technology consulting team

Term	DocID
chosen	1
softwar	1
part	1
larger	1
engin	1
team	1
develop	1
medic	1
devic	1
seek	2
season	2
softwar	2
develop	2
strong	2
analyt	2
technic	2
skill	2
join	2
public	2
sector	2
technolog	2
consult	2
team	2

Key step — Sorting : Compute intensive

- Sort by terms
  - At least conceptuall
    - And then docID

Core indexing step

Term	DocID
chosen	1
softwar	1
part	1
larger	1
engin	1
team	1
develop	1
medic	1
devic	1
seek	2
season	2
softwar	2
develop	2
strong	2
analyt	2
technic	2
skill	2
join	2
public	2
sector	2
technolog	2
consult	2
team	2

Term	DocID
analyt	2
chosen	1
consult	2
develop	2
develop	1
devic	1
engin	1
join	2
larger	1
medic	1
part	1
public	2
season	2
sector	2
seek	2
skill	2
softwar	2
softwar	1
strong	2
team	2
team	1
technic	2
technolog	2

Each component

- Indexer steps: Dictionary & Postings
  - Multiple term entries in a single document are merged.
  - Split into Dictionary and Postings
  - Doc. frequency information is added.

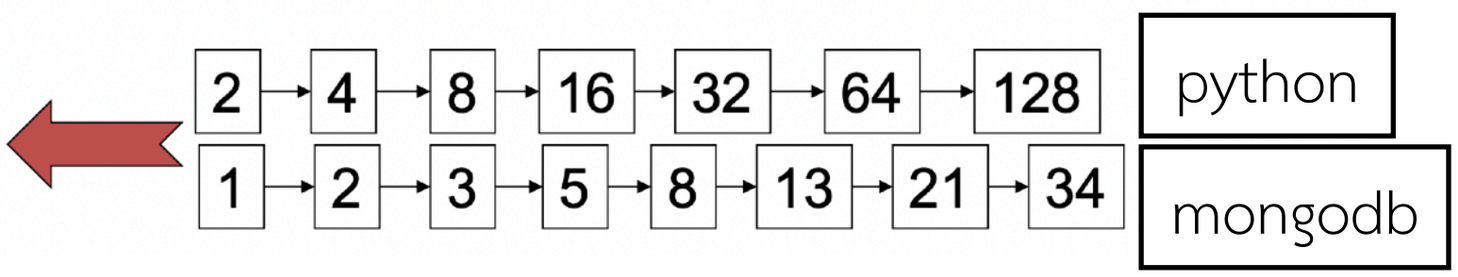
Why frequency?
Will discuss later.

Term	DocID
analyt	2
chosen	1
consult	2
develop	2
develop	1
devic	1
engin	1
join	2
larger	1
medic	1
part	1
public	2
season	2
sector	2
seek	2
skill	2
softwar	2
softwar	1
strong	2
team	2
team	1
technic	2
technolog	2

Term	doc count	posting lists
analyt	1	1
chosen	1	1
consult	1	1
develop	2	1 → 2
devic	1	1
engin	1	1
join	2	1 → 2
larger	1	1
medic	1	1
part	1	1
public	1	2
season	1	2
sector	1	2
seek	1	2
skill	1	2
softwar	2	1 → 2
strong	1	2
team	2	1 → 2
technic	1	2
technolog	1	2

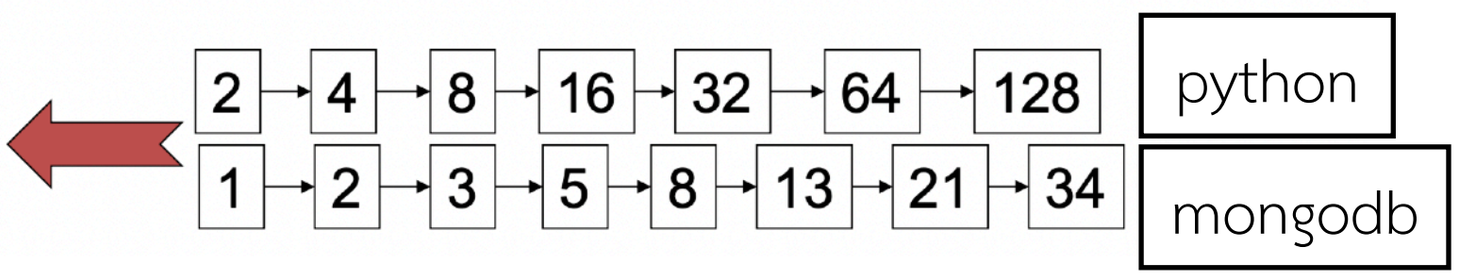
Querying

- AND
  - E.g., **Python** and **MongoDB**
    - Locate **python** in the **Dictionary** and get its **Posting list**;
    - Locate **mongodb** in the **Dictionary** and get its **Posting list**;
    - Merge the two **Posting lists, i.e., set intersection**.



Querying

- OR
  - E.g., **Python** <u>or</u> **MongoDB**
    - Locate **python** in the **Dictionary** and get its **Posting list**;
    - Locate **mongodb** in the **Dictionary** and get its **Posting list**;
    - Merge the two **Posting lists, i.e., set <u>union</u>.**



Running through a classic example

- Reuters RCV1 collection
  - one year of Reuters newswire (part of 1995 and 1996)


REUTERS

You are here: Home > News > Science > Article

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

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.

Lewis, D. D., Yang, Y., Rose, T. G., & Li, F. (2004). RCV1: A new benchmark collection for text categorization research. The Journal of Machine Learning Research, 5, 361-397.

Some RCV1 statistics

- 800,000 documents
- 400,000 unique terms
- 200 tokens is the average tokens per document
- The size of non-positional postings is 100,000,000

Sort-based index construction

- To construct the index, we parse the documents one at a time.
- Storing in the form of (termID, docID) demands huge amount of memory.
- For RCV1, the size of non-positional posting = 100,000,000
  - It may fit today's memory, but this RCV1 dataset contains only one-year of data back in 1994.
- In other words, it is unavoidable to store some intermediate results on disk
  - Much slower than using only the main memory (RAM).

Sort-based index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account hardware constraints.
- Fault tolerance also requires a careful consideration.

Review of 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.
- Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.

What about sorting on disk as memory ?

- **Too slow** — Sorting 100,000,000 records **in disk** require too many disk seeks.
- SSD is now much faster than disk but may not fully compete RAM in a typical setup.
- Thus, we need an external sorting algorithm.

External memory indexing

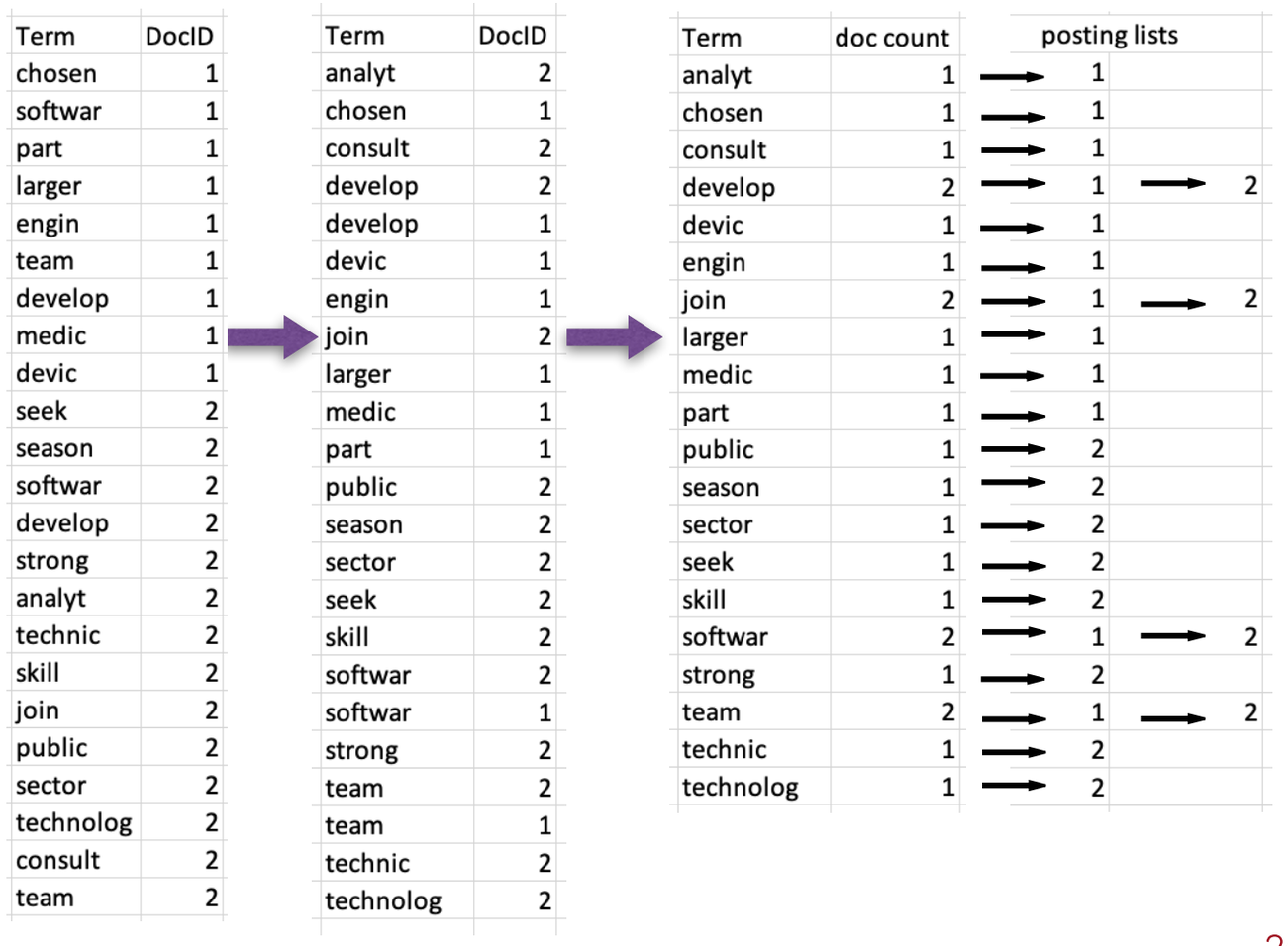
- **BSBI: Blocked sort-based Indexing**
  - Sorting with fewer disk seeks
  - Basic idea of algorithm:
    - Accumulate postings for each block, sort, write to disk.
    - Merge the blocks into one long sorted order.

External memory indexing

- **Example: Sorting 10 blocks of 10M records**
  - First, read each block and sort within:
    - O(N log N)
  - 10 blocks = 10 runs
  - Without further optimization, we need two copy of data on the disks

External memory indexing

- **Example: Merging 10 blocks of 10M records**
  - Binary merge of 10 blocks will require 4 layers, i.e., $\log_2 10$
  - During each layer, read into memory runs in blocks of 10M, merge, write back.



Term	DocID
chosen	1
softwar	1
part	1
larger	1
engin	1
team	1
develop	1
medic	1
devic	1
seek	2
season	2
softwar	2
develop	2
strong	2
analyt	2
technic	2
skill	2
join	2
public	2
sector	2
technolog	2
consult	2
team	2

Term	DocID
analyt	2
chosen	1
consult	2
develop	2
develop	1
devic	1
engin	1
join	2
larger	1
medic	1
part	1
public	2
season	2
sector	2
seek	2
skill	2
softwar	2
softwar	1
strong	2
team	2
team	1
technic	2
technolog	2

Term	doc count	posting lists
analyt	1	1
chosen	1	1
consult	1	1
develop	2	1 → 2
devic	1	1
engin	1	1
join	2	1 → 2
larger	1	1
medic	1	1
part	1	1
public	1	2
season	1	2
sector	1	2
seek	1	2
skill	1	2
softwar	2	1 → 2
strong	1	2
team	2	1 → 2
technic	1	2
technolog	1	2

External memory indexing

- **Optimizing the sort-based algorithm**
  - multi-way merge, e.g., reading all blocks simultaneously
    - Open all block files simultaneously and maintain a read buffer for each one and a write buffer for the output file;
    - In each iteration, pick the lowest termID that hasn't been processed using a priority queue;
    - Merge all postings lists for that termID and write it out
  - At least, #disk seek is reduced

External memory indexing

- **Remaining problems with the sort-based algorithm**
  - Static in-memory dictionary

External memory indexing

- **SPIMI: Single-pass in-memory indexing**
  - Basic idea of algorithm:
    - **More local** —> Generate separate dictionaries for each block = no need to maintain term-termID mapping across blocks.
    - **Do not sort** —> Accumulate postings in postings lists as they occur.
    - **Merge at once** —> These separate indexes can then be merged into one big index.

Distributed indexing

- **Using computing cluster**
  - for scaling the indexing process

Distributed indexing

- **Web search engine data centers**
  - Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
  - Data centers are distributed around the world.
    - As of Gartner in 2007,
      - Google ~1 million servers, 3 million processors/cores

Distributed indexing

- **Fault tolerant matter**
  - If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
  - Then, if the up time is 99%, can you whether the difference is large or small?

Distributed indexing

- **Fault tolerant matter**
  - If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
    - 0.999^{1000} = 37%
    - meaning that 63% of the time one or more servers is down.

Distributed indexing

- **Fault tolerant matter**
  - If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
    - 0.999^{1000} = 37%
      - meaning that 63% of the time one or more servers is down.

- 99% -> 0.99^{1000} = 0.0043%
  - >99.9% of the time one or more servers is down.

Distributed indexing

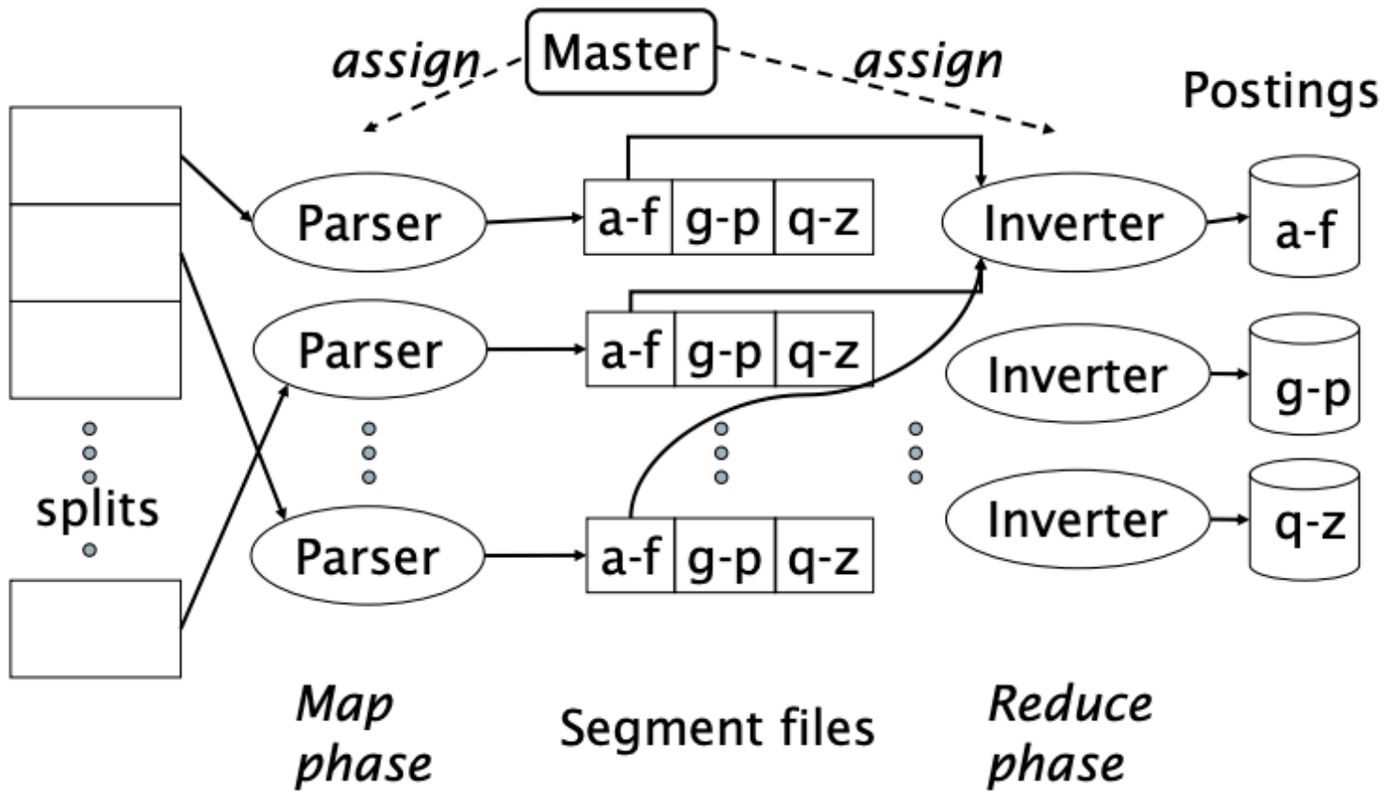
- **Key concept**
  - Maintain a master machine directing the indexing job
  - Break up indexing into sets of (parallel) tasks.
  - Master machine assigns each task to an idle machine from a pool.

Distributed indexing

- **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 (e.g., corresponding to blocks in BSBI)

Distributed indexing

- Data flow



Distributed indexing

- **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
    - **Example:** Each partition is for a range of terms' first letters
      - e.g., 3 partitions : a-f, g-p, q-z
  - Now to complete the index inversion

Distributed indexing

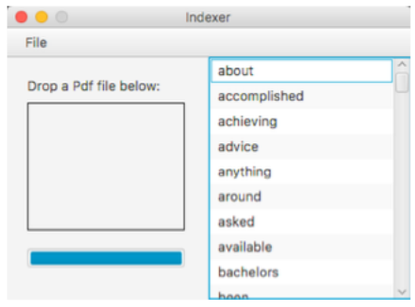
- **Inverters**
  - An inverter collects all (term, doc) pairs (= postings) for one term-partition.
  - Sorts and writes to postings lists

Distributed indexing

- **Index construction**
  - 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
  - As well discuss in the web part of the course, most search engines use a document-partitioned index … better load balancing, etc.

Distributed indexing

- **MapReduce**
  - The index construction algorithm we just described is an instance of MapReduce. — We played once in SE233
  - 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.



Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters.

Distributed indexing

- **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, …)

Distributed indexing

- **Dynamic indexing — Simplest approach**
  - Maintain one (big) main index
  - New docs go into small auxiliary index
  - Adding — Search across both, merge results
  - Deleting — Filter docs output on a search result by this invalidation bit-vector
  - Periodically, re-index into one main index

Distributed indexing

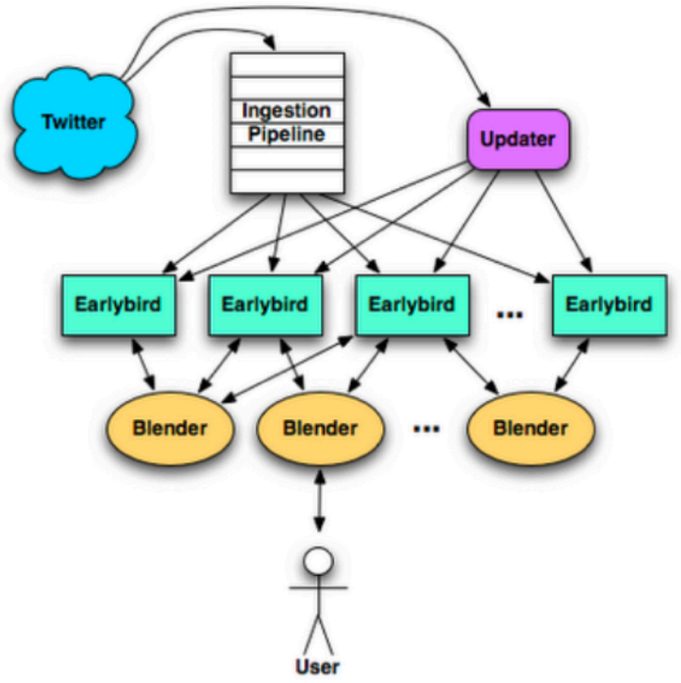
- **Dynamic indexing — Issues**
  - Problem of frequent merges — poor performance during merge
  - **Actually**:
    - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
    - Merge is the same as a simple append.
    - But then we would need a lot of files – inefficient for OS.
- **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.)

Distributed indexing

- **Dynamic indexing — Real world in search engine**
  - 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 deleted

Earlybird: Real-time search at Twitter

- **Requirements for real-time search**
  - Low latency, high throughput query evaluation
  - High ingestion rate and immediate data availability
  - Concurrent reads and writes of the index
  - Dominance of temporal signal



Ref: https://blog.twitter.com/engineering/en_us/a/2011/the-engineering-behind-twitter-s-new-search-experience.html

Earlybird: Real-time search at Twitter

- **Index organization**
  - Earlybird consists of multiple index segments
    - Each segment is relatively small, holding up to 2^{23} tweets
    - Each posting in a segment is a 32 bit word:
      - 24 bits for the tweet id and
      - 8 bits for the position in the tweet
  - Only one segment can be written to at any given time
    - Small enough to be in memory
  - The remaining segments are optimized for read-only
    - Postings sorted in reverse chronological order (newest first)

Index construction — Summary

- Sort-based indexing
  - Naïve in-memory inversion
  - Blocked Sort-Based Indexing (BSBI)
    - Avoid hard disk seeking
- Single-Pass In-Memory Indexing (SPIMI)
    - No global dictionary — Generate separate dictionary for each block
    - Don’t sort postings — Accumulate postings in postings lists as they occur
- Distributed indexing — using MapReduce
- Dynamic indexing — Multiple indices, logarithmic merge

Extra practices — Compression

- More storage
  - inverted file and documents use lots of space
  - Compression lets us index more document
- Faster access
  - A compressed block stores more information
  - Each read brings in more da

Extra practices — Compression

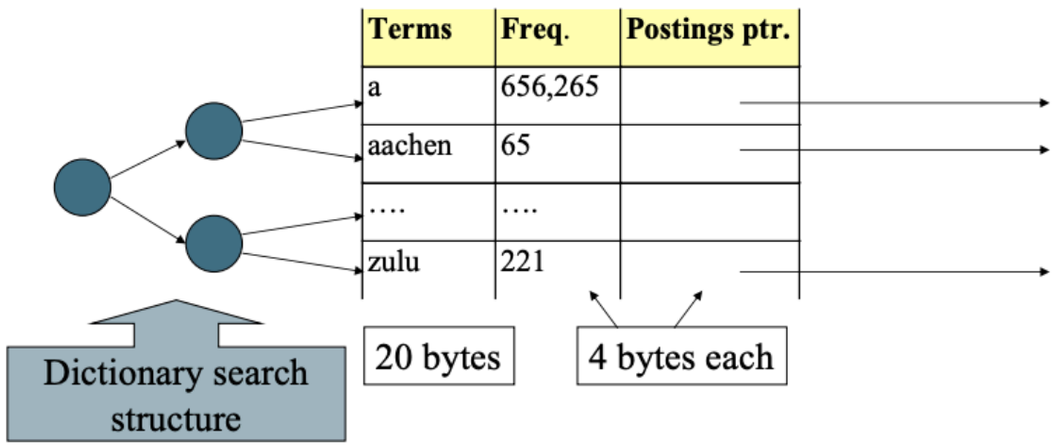
- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.

Dictionary compression

- Example — Dictionary of RCV1

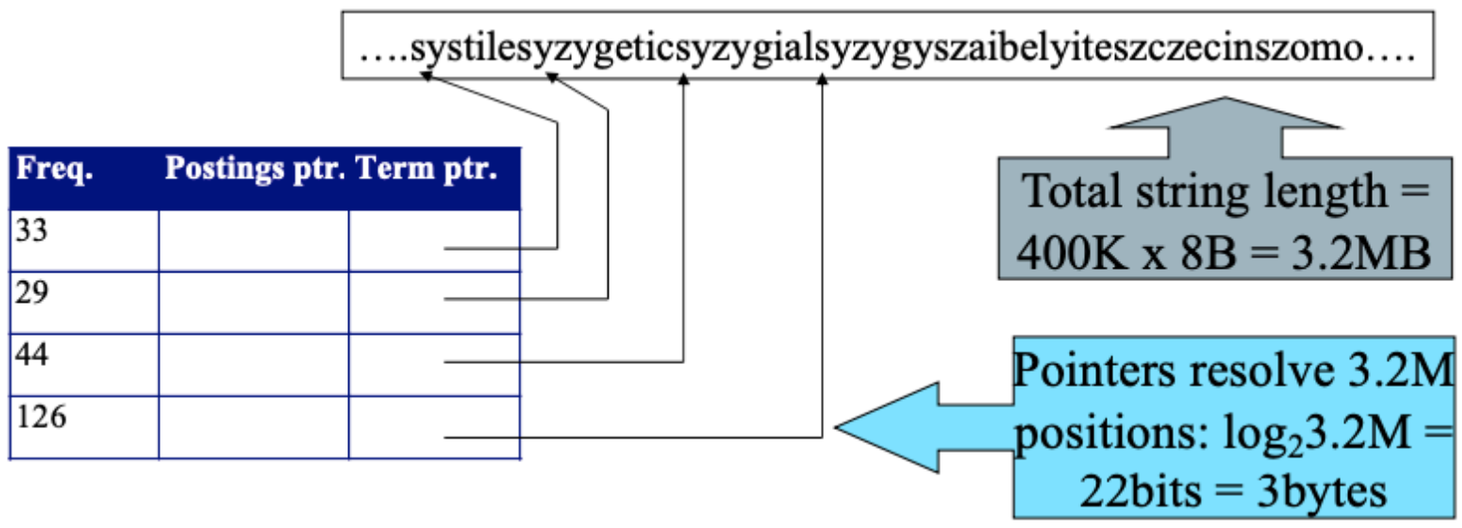
- 800,000 documents
- 400,000 unique terms
- 200 tokens is the average tokens per document
- The size of non-positional postings is 100,000,000

- Without compression
- ~400,000 terms @ e.g., 28bytes/term = 11.2 MB



Dictionary compression

- Compressing the term list: Dictionary-as-a-String
  - Store dictionary as a (long) string of characters:
    - Pointer to next word shows end of current word
    - Average dictionary word in English: ~8 characters

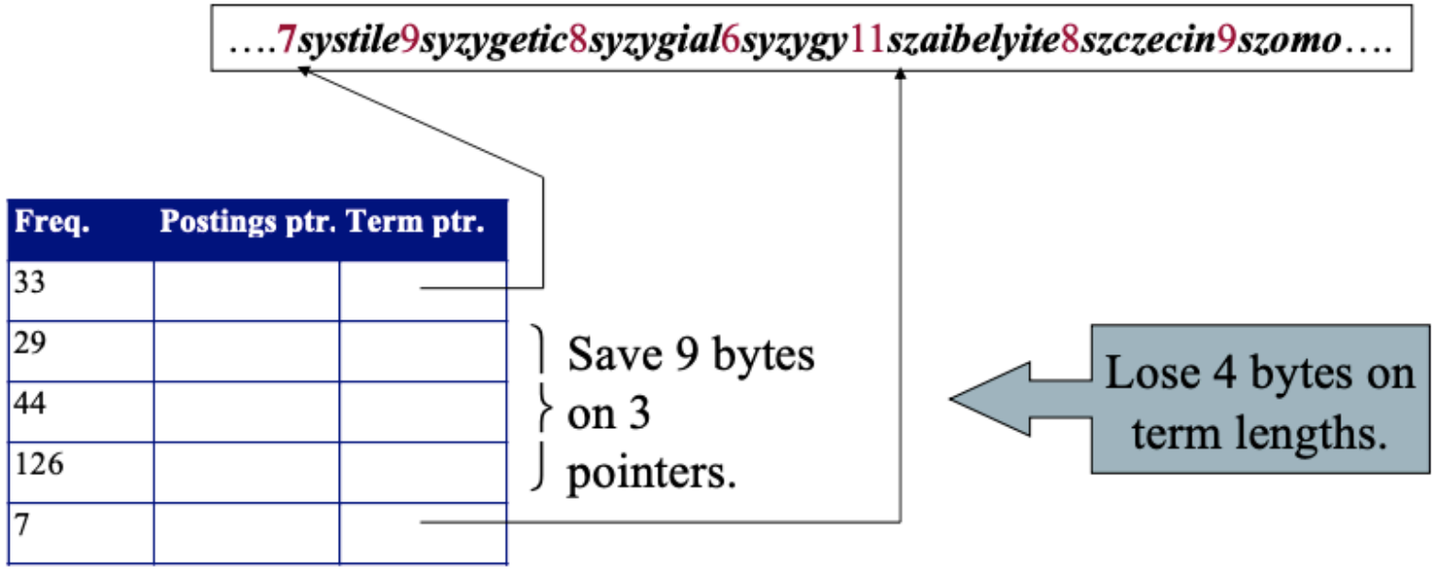


Dictionary compression

- Compressing the term list: Dictionary-as-a-String
  - Estimated space
    - 4 bytes per term for frequencies.
    - 4 bytes per term for pointer to Postings.
    - 3 bytes per term pointer
    - Average 8 bytes per term in term string
    - **400K terms x 19 —> 7.6 MB (against 11.2MB for fixed width)**

Dictionary compression

- Compressing the term list: Blocking
  - Store pointers to every k^{th} term string, e.g., k=4
  - Need to store term lengths (1 extra byte)

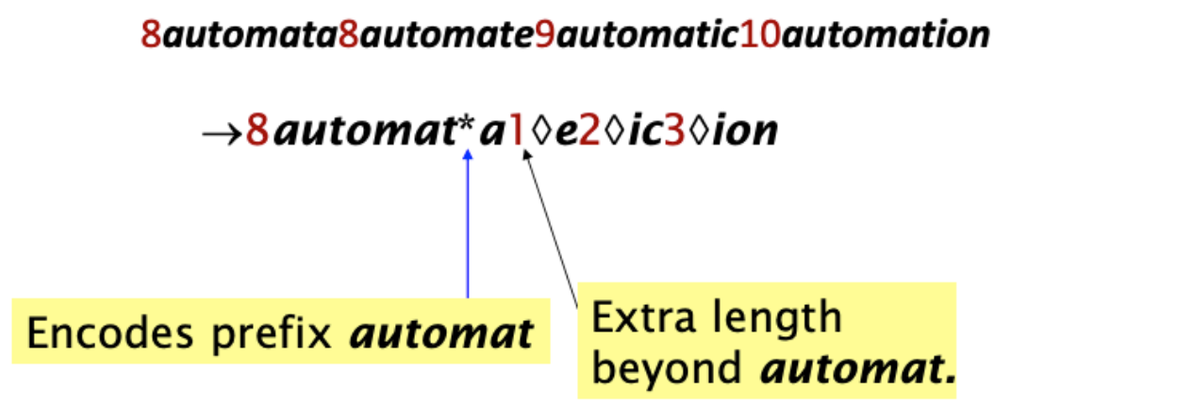


Dictionary compression

- Compressing the term list: Blocking
  - Estimated space
    - Example for block size k = 4
      - Without blocking
        - 3 bytes per pointer —> 12 bytes
      - With blocking
        - 3 + 4 —> 7 bytes
    - **Slightly save ~0.5 Mb more space, now 7.1 Mb**

Dictionary compression

- Compressing the term list: Front coding
  - Sorted words commonly have long common prefix
    - store differences only
    - (for last k-1 in a block of k)



Begins to resemble general string compression.

Dictionary compression

- RCV 1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
+ blocking, $k = 4$	7.1
+ blocking + front coding	5.9

Postings compression

- The postings file is much larger than the dictionary,
  - factor of at least 10, often over 100 times larger
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents)
  - We would use 32 bits per docID when using 4-byte integers.
  - That is, we can use log2 800,000 ≈ 20 bits per docID.
  - Our goal: use far fewer than 20 bits per docID.

Postings compression

- Example — Gap encoding
  - We store the list of docs containing a term in increasing order of docID.
    - e.g., **computer:** 33,47,154,159,202,..
    - Become 33,14,107,5,43
  - Despite being very simple, it helps a lot for common words.

Postings compression

- Example — Variable length encoding
  - Use smaller variable size per gap entry, e.g.,
    - for less word like ***arachnocentric*** we will use ~20bits/gap entry
    - for more word like ***cat*** we will use ~1bits/gap entry
  - Unary code
    - Represent n as n 1s with a final 0, e.g.,
    - 3 is 1110.
  - An example practice is Elias' Gamma codes

Postings compression

- Example — Elias' Gamma codes
- Store integer in two parts
  - 1 + floor($\log_2 x$) in unary
  - $x - 2^{\text{floor}(\log_2 x)}$ in binary
- In total it use

  ~ 1 + 2 $\log_2 x$ bits

number	2^n	output
1	2^0+0	1
2	2^1+0	010
3	2^1+1	011
4	2^2+0	00100
5	2^2+1	00101
6	2^2+2	00110
7	2^2+3	00111
8	2^3+0	0001000
9	2^3+1	0001001
10	2^3+2	0001010
11	2^3+3	0001011
12	2^3+4	0001100
13	2^3+5	0001101
14	2^3+6	0001110
15	2^3+7	0001111
16	2^4+0	000010000
17	2^4+1	000010001

Example

$42=2^5+10$

00000101010

Dictionary compression

- RCV 1 compression summary

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ–encoded	101.0

More comprehensible way to compress

- Sparse matrix
  - Matrix in which most of elements are zero. (Wikipedia)
  - Not a good idea to store the entire matrix as is.

	java	python	...	kotlin
0	0	0	...	0
1	0	0	...	0
2	0	0	...	0
3	0	0	...	0
4	0	0	...	0
	...	...	...	...
7582	1	0	0	0

Sparse matrix representation

- Scipy library –

Sparse matrices (`scipy.sparse`)¶

SciPy 2-D sparse matrix package for numeric data.

Contents

Sparse matrix classes

bsr_matrix(arg1[, shape, dtype, copy, blocksize])	Block Sparse Row matrix
coo_matrix(arg1[, shape, dtype, copy])	A sparse matrix in COOrdinate format.
csc_matrix(arg1[, shape, dtype, copy])	Compressed Sparse Column matrix
csr_matrix(arg1[, shape, dtype, copy])	Compressed Sparse Row matrix
dia_matrix(arg1[, shape, dtype, copy])	Sparse matrix with DIAgonal storage
dok_matrix(arg1[, shape, dtype, copy])	Dictionary Of Keys based sparse matrix.
lil_matrix(arg1[, shape, dtype, copy])	Row-based list of lists sparse matrix
spmatrix([maxprint])	This class provides a base class for all sparse matrices.

More comprehensible way to compress

- According to Wikipedia — sparse matrices are categorized as
  - Those for just constructing the matrices
    - **DOK**, **LIL**, and **COO**
  - Those for constructing the matrices and computing — supports efficient access, arithmetic operation, and matrix-vector products
    - **CSR** and **CSC**

- Machine-learning communities suggest to build a matrix in **COO** or **LIL** format then compress it in **CSR** format.

More comprehensible way to compress

- E.g., https://towardsdatascience.com/working-with-sparse-data-sets-in-pandas-and-sklearn-d26c1cfbe067
  - Modelling with the MovieLens 100K Dataset (https://grouplens.org/datasets/movielens/100k/)
    - Without compressing it took 3.06 + 0.82 s to complete the task.
    - Compressing in LIL -> CSR decreases the time to 1.58 + 0.05 s
  - We should also note that matrix construction and compression also take time.
    - Good for repetitive tasks.

Some walkthroughs

```
01  from nltk.tokenize import word_tokenize
02  from nltk.corpus import stopwords
03  from nltk.stem import PorterStemmer
04
05  cleaned_description = get_and_clean_data()
06  cleaned_description = cleaned_description.iloc[:2]
07
08  tokenized_description = cleaned_description.apply(lambda s: word_tokenize(s))
09  sw_removed_description = tokenized_description.apply(lambda s: [word for word in s if word not in
stopwords.words()])
10  sw_removed_description = sw_removed_description.apply(lambda s: [word for word in s if
len(word)>2])
11
12  ps = PorterStemmer()
13  stemmed_description = sw_removed_description.apply(lambda s: [ps.stem(w) for w in s])
```

```
1   from sklearn.feature_extraction.text import CountVectorizer
2   cv = CountVectorizer(analyzer=lambda x: x)
3   X = cv.fit_transform(stemmed_description)
```

Some walkthroughs

```
print(pd.DataFrame(X.toarray()))
```

Dense

	0	1	2	3	4	5	6	...	280	281	282	283	284	285	286
0	0	1	1	1	1	0	1	...	1	2	1	0	5	1	0
1	1	0	0	0	0	1	0	...	0	1	0	1	1	0	1

Some walkthroughs

```
print(X.tocsr()[0,:])
```

CSR

(0, 43)	1
(0, 239)	10
(0, 76)	12
(0, 182)	1
(0, 139)	1
(0, 202)	1
(0, 104)	1
(0, 47)	1
(0, 260)	1
:	:
(0, 189)	1

```
print(X.tocoo()[0, :])
>>> error
```

Some performance test

```
import timeit
timeit.timeit(lambda: np.matmul(X.toarray(), X.toarray().T),number=1)
np.shape(np.matmul(X.toarray(),X.toarray().T))
```

```
timeit.timeit(lambda: X*X.T,number=1)
np.shape(X*X.T))
```

- Using the lecturer' laptop, it is ~4s vs 0.08s

Some performance test

```
>>>timeit.timeit(lambda: X*X.T,number=1)
0.08053383399999348

>>>timeit.timeit(lambda: X.todok()*X.T.todok(),number=1)
0.2057406669999864

>>>timeit.timeit(lambda: X.tolil()*X.T.tolil(),number=1)
0.12067987500000754

>>>timeit.timeit(lambda: X.tocoo()*X.T.tocoo(),number=1)
0.09625850000003311

>>>timeit.timeit(lambda: X.tocsc()*X.T.tocsc(),number=1)
0.08785583300004873
```

Time for questions