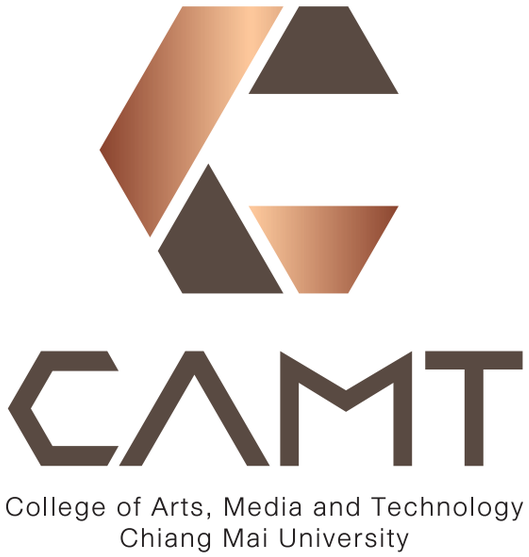
SE 481 Introduction to Information Retrieval (IR for SE)

Module #2 — Index Construction



Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

From assignment

```python
def inverse_indexing(parsed_description):
    sw_set = set(stopwords.words()) - {'c'}
    no_sw_description = parsed_description.apply(lambda x: [w for w in x if w not in sw_set])
    ps = PorterStemmer()
    stemmed_description = no_sw_description.apply(lambda x: set([ps.stem(w) for w in x]))
    all_unique_term = list(set.union(*stemmed_description.to_list()))

    invert_idx = {}
    for s in all_unique_term:
        invert_idx[s] = set(stemmed_description.loc[stemmed_description.apply(lambda x: s in x)].index)

    return invert_idx

def search(invert_idx, query):
    ps = PorterStemmer()
    processed_query = [s.lower() for s in query.split()]
    stemmed = [ps.stem(s) for s in processed_query]
    matched = list(set.intersection(*[invert_idx[s] for s in stemmed]))
    return matched

if __name__ == '__main__':
    parsed_description = parse_job_description()
    invert_idx = inverse_indexing(parsed_description)
    query = 'java oracle'
    matched = search(invert_idx, query)
    print(parsed_description.loc[matched].apply(lambda x: ' '.join(x)).head().to_markdown())
```

Preprocess = remove stop words and then stem

Create the inverted index

Preprocess query

Union = or, intersect = and

Search using index will perform 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:

- Discussion Points
  - What terminology should we use, including words, phrases, and entities?
  - Which specific terms should be incorporated?
  - Are all of the selected terms equally significant?
  - How should we handle numerical data?
  - How should we address variations in word 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 observations
  - Selecting an indexing unit with a certain level of **specificity** is essential.
  - Not all terms hold the same level of **importance**.
  - Valuable information may be **implied** rather than **explicitly** stated.
  - Some terms might have **multiple interpretations (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 an **automated tokenization 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 keywords
  - Remove stopwords
  - 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 keywords
  - Remove stopwords
  - 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 keywords
  - Remove stopwords
  - 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 keywords
  - Remove stopwords
  - Stemming

Tokenization — revisit

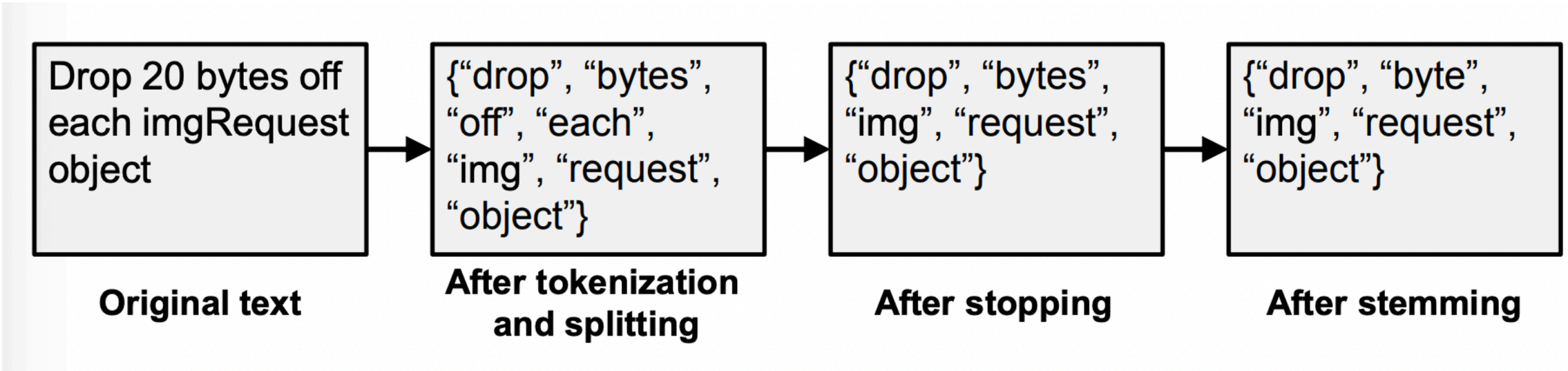
full job descript

~~**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 keywords
  - Remove stopwords
  - Stemming

Tokenization — more examples



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.

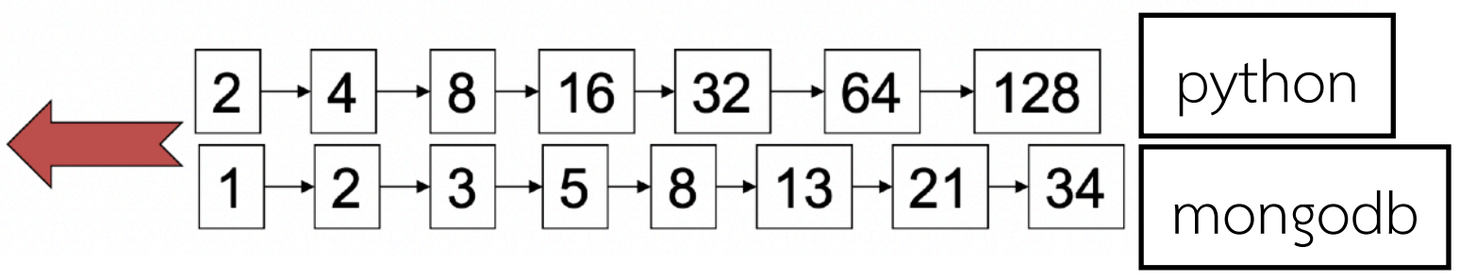
Why frequency?
Will discuss later.

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

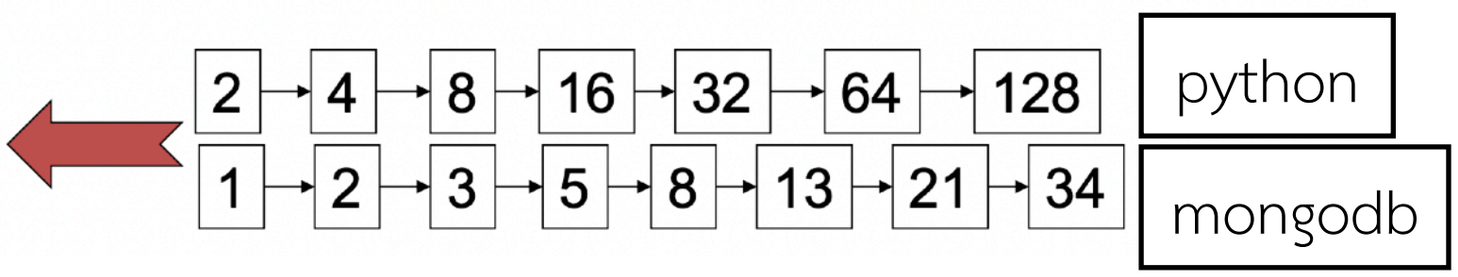
Querying

- 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>.**



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 build an index, we process the documents one by one.
- Storing the data in the format of (termID, docID) requires a substantial amount of memory.
  - For example, in the case of RCV1, the size of non-positional postings is 100,000,000.
  - While this may fit within today's memory capacities, it's important to note that the RCV1 dataset contains only one year's worth of data from 1994.
- Consequently, it becomes necessary to store some intermediate results on disk, which significantly slows down the process compared to using only the main memory (RAM).

Sort-based index construction

- In-memory index construction does not scale.
- How can we construct an index for a very large collection while considering hardware constraints?
- It's essential to carefully consider fault tolerance as well.

Reviews on some hardware basics

- Accessing data in memory is significantly faster than accessing data on disk.
- Disk seeks occur when the disk head is repositioning, leading to no data transfer during this time.
- Transferring a single large chunk of data from disk to memory is faster than transferring multiple smaller chunks.

What about sorting on disk?

- **Too slow** — Sorting 100,000,000 records on disk is too slow due to the high number of disk seeks involved.
- While SSDs are faster than traditional hard drives, they may not fully match the speed of RAM in a typical setup, especially without utilizing specialized hardware.
- Therefore, an **external sorting algorithm** is necessary.

External memory indexing

- **BSBI: Blocked sort-based Indexing**
  - Sorting with reduced disk seeks
  - Basic algorithm concept:
    - Accumulate postings for each block, sort them, and write to disk.
    - Merge the blocks into a single long sorted order.

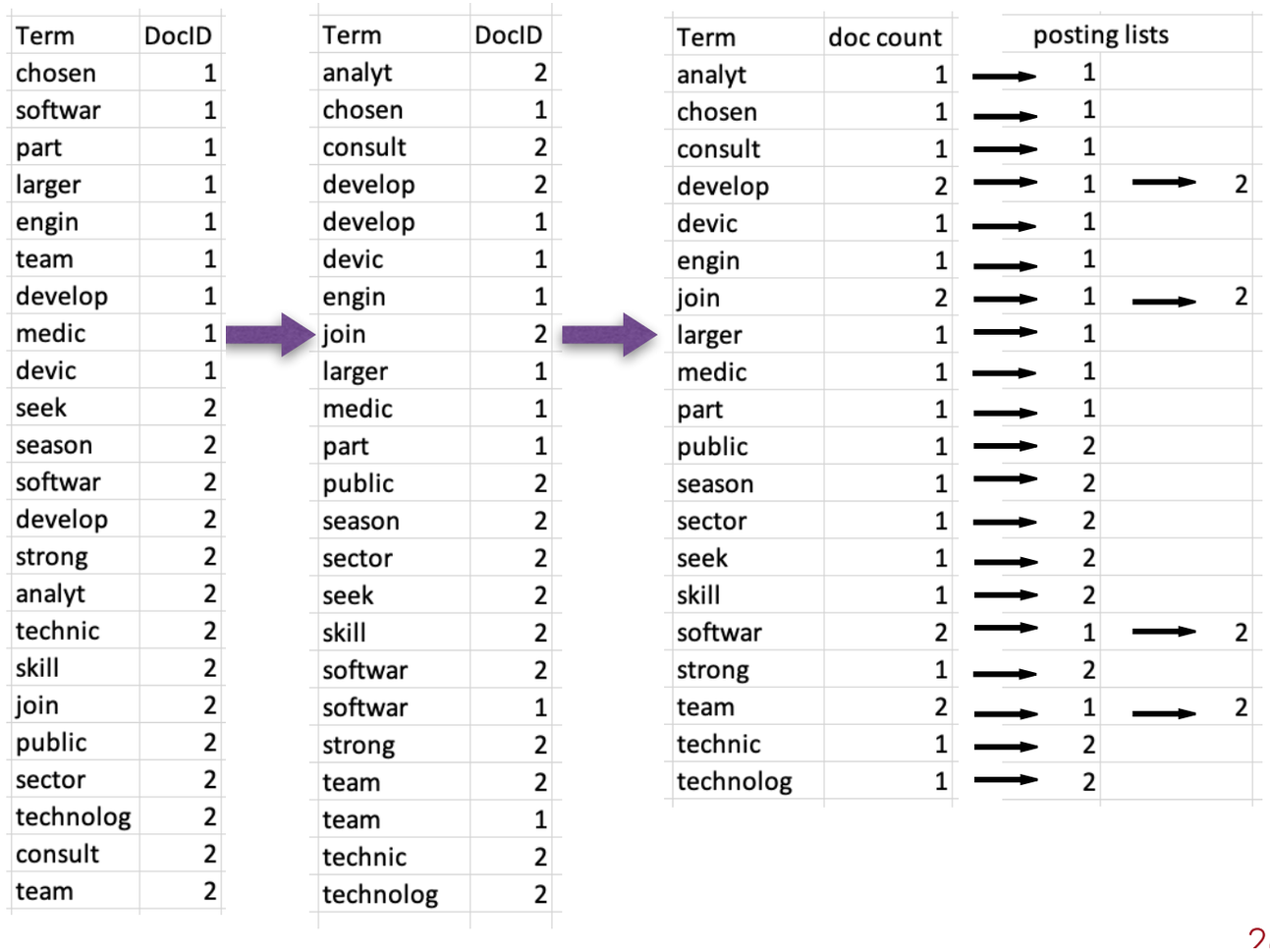
External memory indexing

- **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

E.g. https://nlp.stanford.edu/IR-book/html/htmledition/blocked-sort-based-indexing-1.html

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:**
  - Simultaneous Block Access:
    - Open all block files simultaneously.
    - Maintain read and write buffers for efficiency.
  - TermID Priority Queue:
    - Use a priority queue to select the lowest unused termID.
    - Helps streamline the merging process.
  - Efficient Merging:
    - Merge postings lists for the selected termID.
    - Write the merged result to the output file.
  - Reduced Disk Seeks:
    - Minimize disk seeks for quicker indexing.
    - Significantly improves overall performance.

Multi-Way Merge — example

- Block 1:
  - Term A: [Doc 1, Doc 3]
  - Term B: [Doc 2, Doc 4]
- Block 2:
  - Term A: [Doc 5]
  - Term C: [Doc 6]
- Block 3:
  - Term B: [Doc 7, Doc 8]
  - Term D: [Doc 9]
- Block 4:
  - Term C: [Doc 10, Doc 11]
  - Term D: [Doc 12]

Multi-Way Merge — process

- Open all block files and create read and write buffers.
- Initialize a priority queue for termIDs: {A, B, C, D}.
- Select the lowest termID, which is A.
- Merge postings lists for Term A from Block 1 and Block 2:
  - Term A: [Doc 1, Doc 3, Doc 5]
- Write the merged result to the output file.
- Update the priority queue: {B, C, D}.

Multi-Way Merge — process

- In concurrent with A, process B:
- Merge postings lists for Term B from Block 1 and Block 3:
  - Term B: [Doc 2, Doc 4, Doc 7, Doc 8]
- Write the merged result to the output file.
- Update the priority queue: {C, D}.

- In concurrent with A and B, process C:
- Merge postings lists for Term C from Block 2 and Block 4:
  - Term C: [Doc 6, Doc 10, Doc 11]
- Write the merged result to the output file.
- Update the priority queue: {D}.

Multi-Way Merge — process

- In concurrent with A B and C, process D:
- Merge postings lists for Term C from Block 3 and Block 4:
- Term C: [Doc 9, Doc 12]
- Write the merged result to the output file.

- Summarize the merged results into the final result list
  - Term A: [Doc 1, Doc 3, Doc 5]
  - Term B: [Doc 2, Doc 4, Doc 7, Doc 8]
  - Term C: [Doc 6, Doc 10, Doc 11]
  - Term D: [Doc 9, Doc 12]

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:
    - **Local Approach:** Generate separate dictionaries for each block, eliminating the need to maintain term-termID mapping across blocks.
    - **No Sorting:** Accumulate postings in postings lists as they occur, avoiding the sorting step.
    - **Merge at the End:** Merge these separate indexes into one comprehensive index at a later stage.

SPIMI — example

- **Document Collection:**
  - Document 1: Java is an object-oriented language.
  - Document 2: Python is widely used for data analysis
  - Document 3: JavaScript and TypeScript are commonly used for web development

- **SPIMI Process with Programming Languages:**
  1. **Initialization:** Create an empty postings list for each programming language encountered (e.g., Java, Python, JavaScript, TypeScript) and prepare separate dictionaries for each block. Also, maintain a master dictionary for the final merged index.

SPIMI — example

- **SPIMI Process with Programming Languages:**

  2. **Block 1 Processing (Documents 1 and 2):**

     - **Process Document 1:**

       - Term **Java** (new term): Add entry to Block 1 dictionary with DocID 1.
       - Term **object-oriented** (new term): Add entry to Block 1 dictionary with DocID 1.

     - **Process Document 2:**

       - Term **Python** (new term): Add entry to Block 1 dictionary with DocID 2.
       - Term **data analysis** (new term): Add entry to Block 1 dictionary with DocID 2.

SPIMI — example

- **SPIMI Process with Programming Languages:**
  3. **Block 2 Processing (Document 3):**
     - **Process Document 3:**
       - Term **Java** (already seen): Update Block 1 dictionary with DocID 3.
       - Term **Python** (already seen): Update Block 1 dictionary with DocID 3.
       - Term **web frontend** (new term): Add entry to Block 2 dictionary with DocID 3.
       - Term not commonly used (new term): Add entry to Block 2 dictionary with DocID 3.

SPIMI — example

- **SPIMI Process with Programming Languages:**

  4. **Merging Block 1 and Block 2:**
     - Merge Block 1 and Block 2 dictionaries into a single dictionary for the final index.
     - Resolve any term conflicts or updates by combining postings lists.

SPIMI — example

- **Final Inverted Index for this example:**
  - Java -> [1, 3]
  - object-oriented -> [1]
  - Python -> [2, 3]
  - data analysis -> [2]
  - web frontend -> [3]
  - not commonly used -> [3]

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**
  - Suppose 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%, do you think 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?
    - $0.999^{1000} \approx 0.368$ which is around 36.8%
    - meaning that 63% of the time one or more servers is down.

Distributed indexing

- **Fault tolerant matter**
  - 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% -> $0.99^{1000} \approx 0.0366$ which is around 3.66%
  - So, the downtime percentage for the entire system is approximately 96.34% which appears to be unacceptable.

Distributed indexing

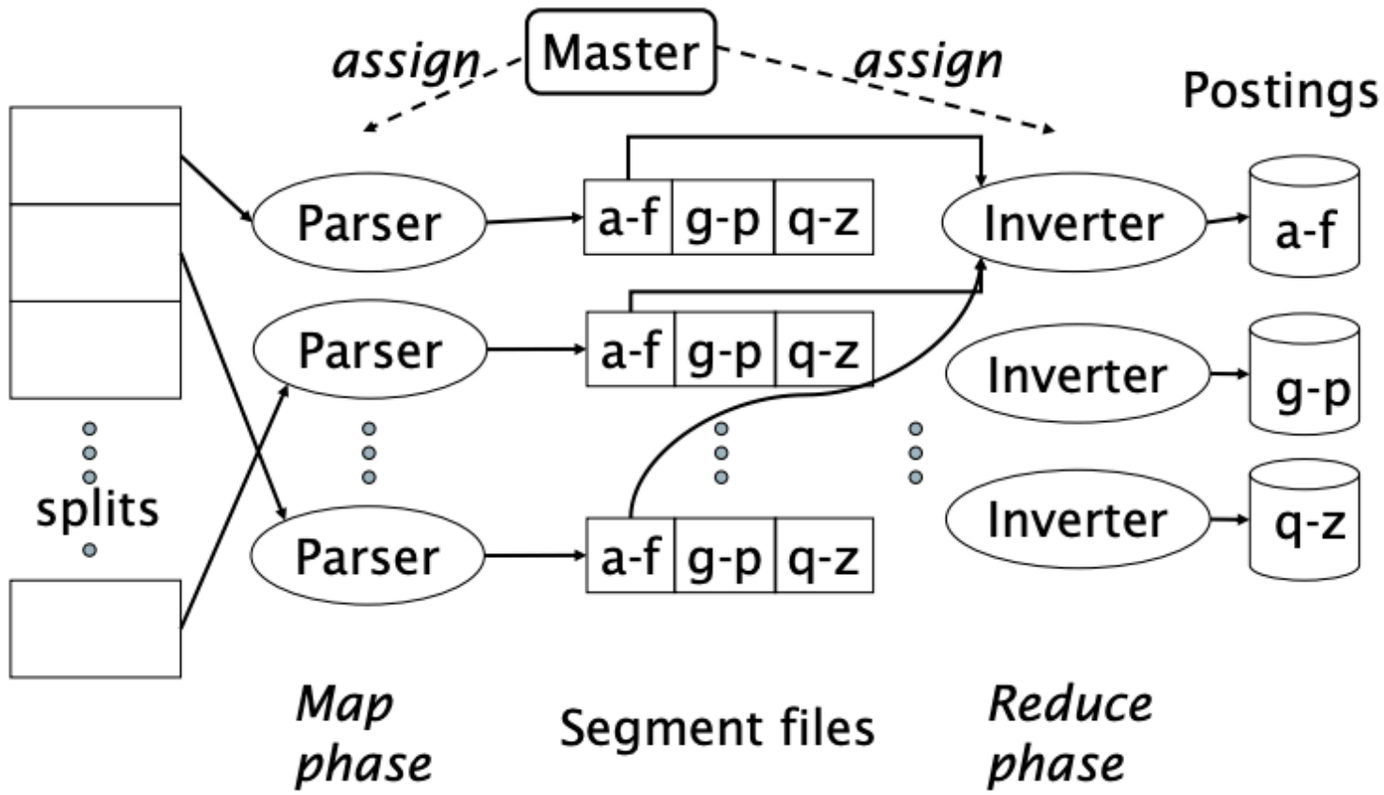
- **Key concept**
  - Maintain a master machine responsible for directing the indexing job.
  - Divide the indexing process into sets of parallel tasks.
  - The master machine assigns each task to an available machine from a pool of resources.

Distributed indexing

- **Parallel tasks**
  - We utilize two sets of parallel tasks:
    - Parsers
    - Inverters
  - The input document collection is divided into splits.
  - Each split represents a subset of documents (e.g., akin to blocks in BSBI).

Distributed indexing

- Data flow



Distributed indexing

- **Parsers**
  - The master assigns a split to an idle parser machine.
  - Each parser reads one document at a time and emits (term, doc) pairs.
  - Parsers write these pairs into designated partitions.
  - For example, partitions can be based on the first letters of terms.
    - Example with 3 partitions:
      - Partition 1: Terms starting with a-f
      - Partition 2: Terms starting with g-p
      - Partition 3: Terms starting with q-z
  - This process is a step toward completing the index inversion.

Distributed indexing

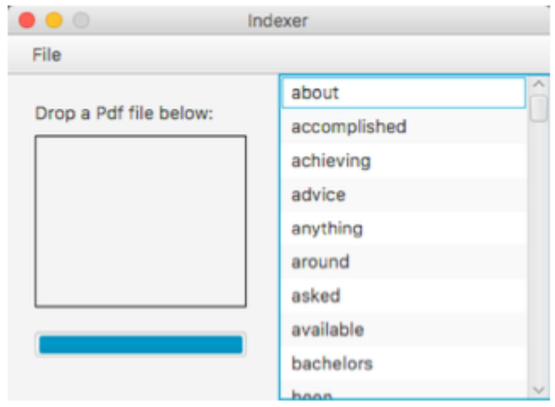
- **Inverters**
  - An inverter gathers all (term, doc) pairs (postings) for a specific term-partition.
  - It performs sorting and writes the postings to postings lists.

Distributed indexing

- **Index construction**
  - Index construction occurs in a single phase.
  - There is another phase involved: transforming a term-partitioned index into a document-partitioned index.
    - In term-partitioned indexing, each machine handles a subrange of terms.
    - In document-partitioned indexing, each machine handles a subrange of documents.
  - As we'll discuss in the web part of the course, most search engines opt for a document-partitioned index. This choice offers benefits such as improved load balancing and more efficient operations.

Distributed indexing

- **MapReduce**
  - The index construction algorithm we've discussed is an example of MapReduce, which we've previously explored in SE233.
  - MapReduce (Dean and Ghemawat, 2004) is a robust and conceptually straightforward framework for distributed computing, eliminating the need to write distribution-specific code.
  - The Google indexing system is described as comprising multiple phases, with each phase implemented using 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 — A simple approach**
  - Maintain one primary (big) main index.
  - New documents are added to a small auxiliary index.
  - For adding documents:
    - Search across both indexes.
    - Merge the results to provide comprehensive search results.
  - For deleting documents:
    - Use a filter based on an invalidation bit-vector derived from search results.
  - Periodically, re-index the data from the auxiliary index into the main index to maintain efficiency and coherency.

Distributed indexing

- **Dynamic indexing — Issues**
  - The problem of frequent merges can lead to poor performance during the merge process.
  - **In practice**:
    - Merging the auxiliary index into the main index is efficient if we maintain separate files for each postings list.
    - The merge operation becomes similar to a simple append operation.
    - However, using too many files can be inefficient for the operating system.
  - **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 major search engines employ dynamic indexing techniques.
  - Their indices constantly undergo incremental changes due to various factors:
    - News items, Updates to blogs, new topical web pages
  - However, at times (sometimes or typically), they also undertake the periodic reconstruction of the index from scratch.
  - During this reconstruction:
    - Query processing is switched to the new index.
    - The old index is subsequently deleted.

Index construction — Summary

- Sort-based indexing
  - Includes Naïve In-Memory Inversion.
  - Utilizes Blocked Sort-Based Indexing (BSBI) to minimize hard disk seeking.
- Single-Pass In-Memory Indexing (SPIMI)
  - Generates separate dictionaries for each block.
  - Accumulates postings in postings lists without sorting.
- Distributed indexing
  - Implemented using MapReduce for scalability.
- Dynamic indexing
  - Involves maintaining multiple indices with logarithmic merging for efficient updates.

Extra practices — Compression

- More storage
  - Inverted files and documents consume significant storage space.
  - Compression techniques enable the indexing of more documents within the available storage capacity.
- Faster access
  - Compressed blocks store more information efficiently.
  - Each read operation retrieves a larger amount of data, enhancing access speed and efficiency.

Extra practices — Compression

- Dictionary
  - The goal is to keep the dictionary small enough to fit in main memory.
  - Ideally, the dictionary should be so small that it allows for some postings lists to be kept in main memory as well, optimizing performance

- Postings file(s)
  - Compression reduces the disk space required for postings lists.
  - It also decreases the time needed to read postings lists from disk.
  - Large search engines often keep a substantial portion of the postings in memory for improved performance.

Simple compression approach - Sparse matrix

- A sparse matrix is a matrix in which most of the elements are zero (as defined on Wikipedia).
- Instead of storing the entire matrix as is, a more efficient approach is employed to only record the non-zero elements, saving storage space and computational resources.

	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.

Compression for Sparse matrices

- Sparse matrices are matrices with many zero elements.
- Storing them as if they were dense (with all elements) is inefficient.
- Wikipedia categorizes sparse matrix representations into two main types:
  - **For Constructing Matrices Only:**
    mainly used for constructing sparse matrices efficiently.
    Examples include DOK, LIL, and COO.
  - **For Constructing and Efficient Operations:**
    also support efficient access, arithmetic operations, and matrix-vector products.
    Examples include CSR and CSC.

Suggestion from Machine-Learning Communities

- Machine-learning communities recommend building a matrix in **COO (Coordinate List)** or **LIL (List of Lists)** format for efficient construction.
- After construction, it is advised to compress the matrix into **CSR (Compressed Sparse Row)** format, which supports efficient operations and is commonly used in machine learning.

Scipy example

```
01  # dense to sparse
02  from numpy import array
03  from scipy.sparse import coo_matrix, csr_matrix, csc_matrix, dok_matrix, lil_matrix
04  # create dense matrix
05  A = array([[1, 0, 0, 1, 0, 0], [0, 0, 2, 0, 0, 1], [0, 0, 0, 2, 0, 0]])
06  print(A)
07
08  # convert to sparse matrix (COO method)
09  S = coo_matrix(A)
10  print(S)
11
12  print(S.tocsr()[:,3])
13
14  # reconstruct dense matrix
15  B = S.todense()
16  print(B)
```

```
01  times = 100000
02  timeit.timeit(lambda : dok_matrix(B), number=times)/times
03  timeit.timeit(lambda : lil_matrix(B), number=times)/times
04  timeit.timeit(lambda : csr_matrix(B), number=times)/times
05  timeit.timeit(lambda : csc_matrix(B), number=times)/times
```

Example - The MovieLens 100K Dataset

- E.g., https://towardsdatascience.com/working-with-sparse-data-sets-in-pandas-and-sklearn-d26c1cfbe067
  - 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, limiting to just 1000 rows

```
00  from ordered_set import OrderedSet
01  #limit to just 1000 rows
02  cleaned_description = m1.get_and_clean_data()[:1000]
03
04  #replace non alphabets with spaces, and collapse spaces
05  cleaned_description = cleaned_description.apply(lambda s: re.sub(r'[^A-Za-z]', ' ', s))
06  cleaned_description = cleaned_description.apply(lambda s: re.sub(r'\s+', ' ', s))
07
08  #tokenize
09  tokenized_description = cleaned_description.apply(lambda s: word_tokenize(s))
10
11  #remove stop words
12  stop_dict = set(stopwords.words())
13  sw_removed_description = tokenized_description.apply(lambda s: list(OrderedSet(s) - stop_dict))
14  sw_removed_description = sw_removed_description.apply(lambda s: [word for word in s if len(word)>2])
15
16  #create stem caches
17  concated = np.unique(np.concatenate([s for s in tokenized_description.values]))
18  stem_cache = {}
19  ps = PorterStemmer()
20  for s in concated:
21      stem_cache[s] = ps.stem(s)
22
23  #stem
24  stemmed_description = sw_removed_description.apply(lambda s: [stem_cache[w] for w in s])
```

```
1   from sklearn.feature_extraction.text import CountVectorizer
2   cv = CountVectorizer(analyzer=lambda s: s)
3   vectorizer = cv.fit(stemmed_description)
4   X = vectorizer.transform(stemmed_description)
5   print(pd.DataFrame(X.toarray(), columns=cv.get_feature_names_out()))
```

Some walkthroughs

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

CSR

(0, 8386)	1
(0, 29613)	10
(0, 11225)	12
(0, 24119)	1
(0, 19514)	1
(0, 12960)	4
(0, 31394)	1
(0, 20988)	5
(0, 11544)	2
:	:
(0, 24920)	1

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

Some performance test

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

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

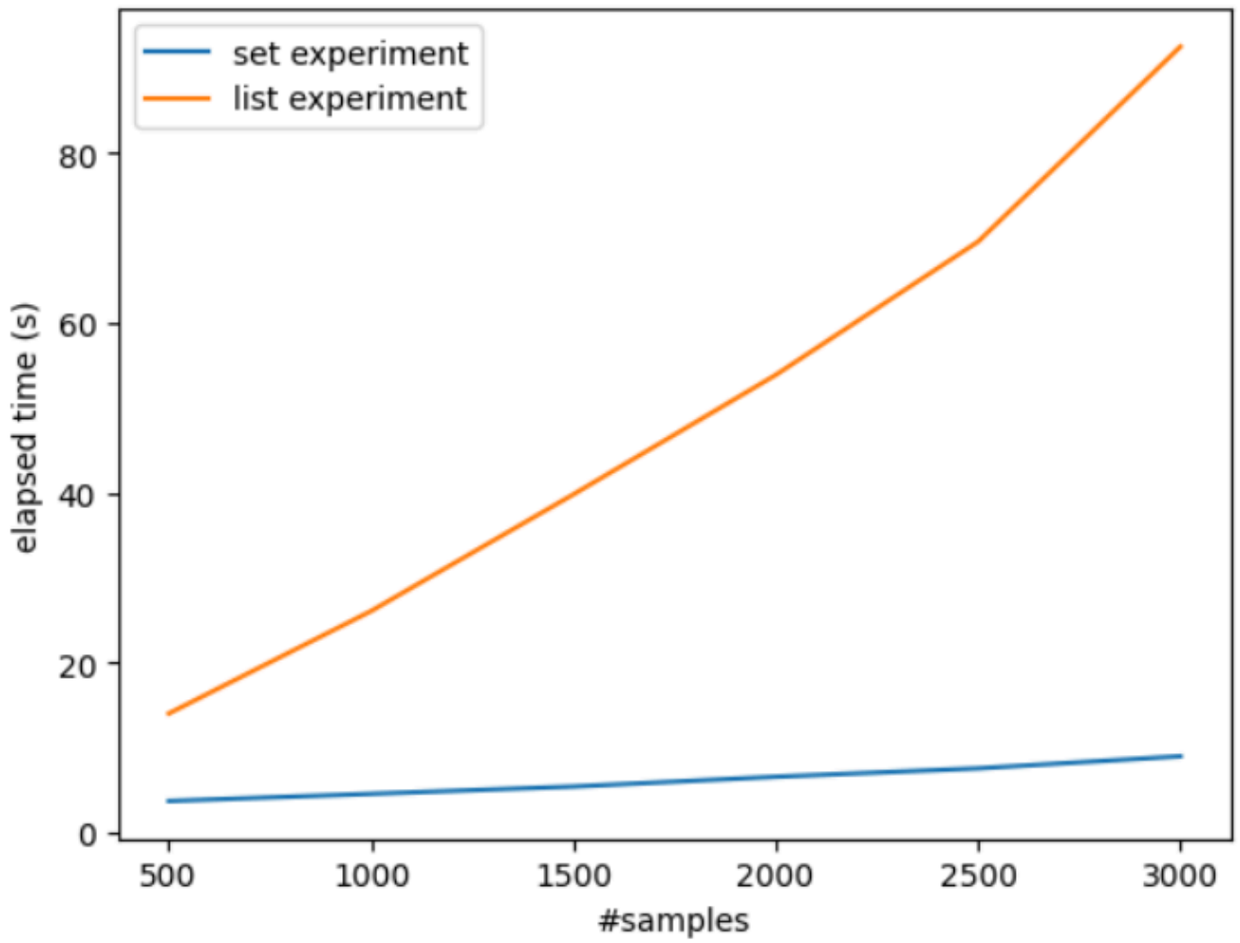
- Using the lecturer' laptop, it is ~4.0s vs 0.06s

Some performance test

```
>>>timeit.timeit(lambda: np.matmul(XX, XX.T), number=3)/3
4.024828208299975

>>>timeit.timeit(lambda: X.todok()*X.T.todok(),number=3)/3
0.1901305276745309 5

>>>timeit.timeit(lambda: X.tolil()*X.T.tolil(),number=3)/3
0.0833277776837349

>>>timeit.timeit(lambda: X.tocoo()*X.T.tocoo(),number=3)/3
0.0666958613631625 9

>>>timeit.timeit(lambda: X.tocsc()*X.T.tocsc(),number=3)/3
0.0665165973672022 4
```

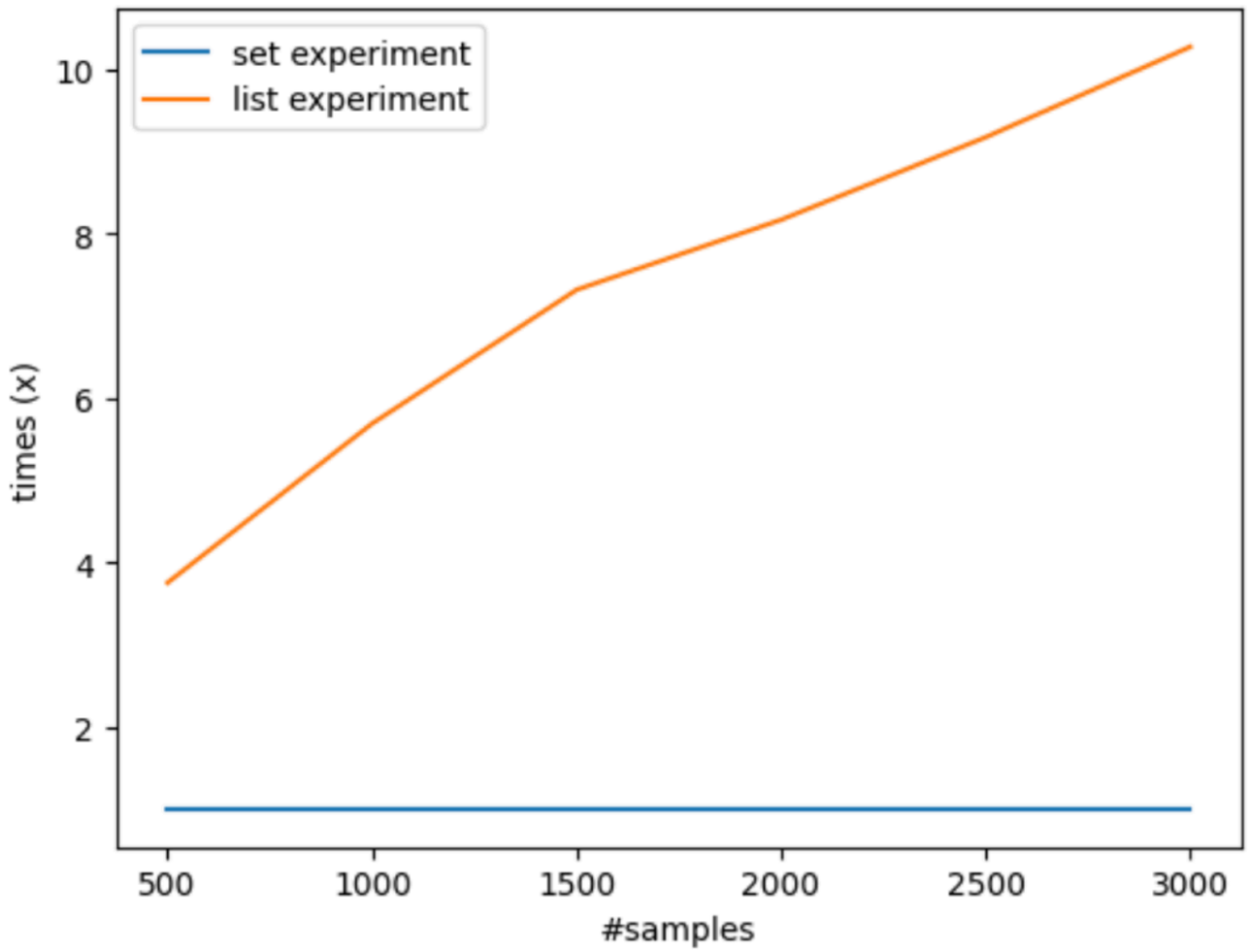
Activity — performance benchmark

- Set operations vs list operation in stemming
  - **list(OrderedSet(s) - stop_dict)** vs **[word for word in s if word not in stop_dict]**



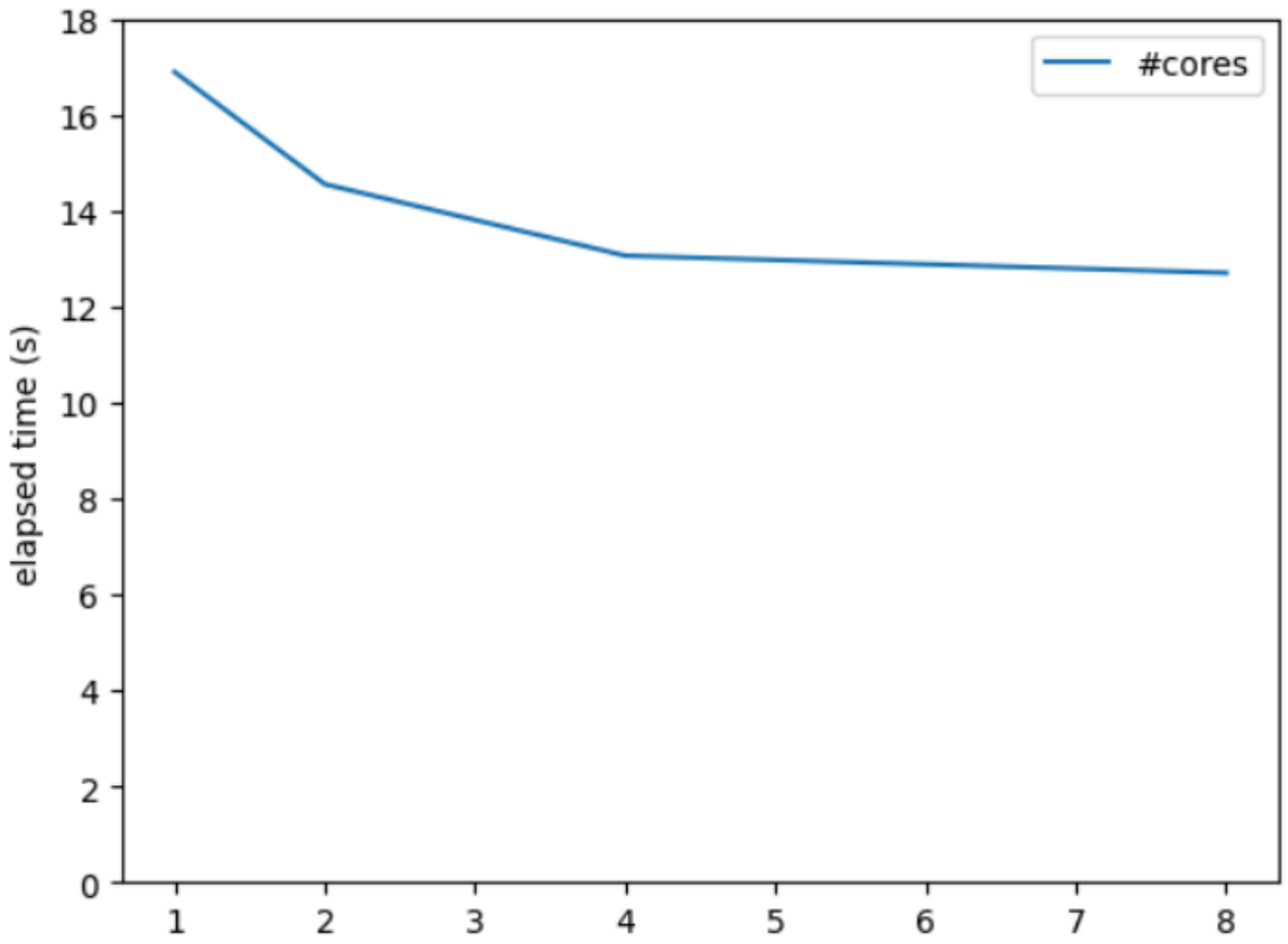
Activity — performance benchmark

- Set operations vs list operation in stemming
  - Convert time unit to **times (x)**



Activity — performance benchmark

- #cores using set experiments
  - **Set to maximum #rows**
  - **1 core - 2 cores - 4 cores .... your maximum #cores**



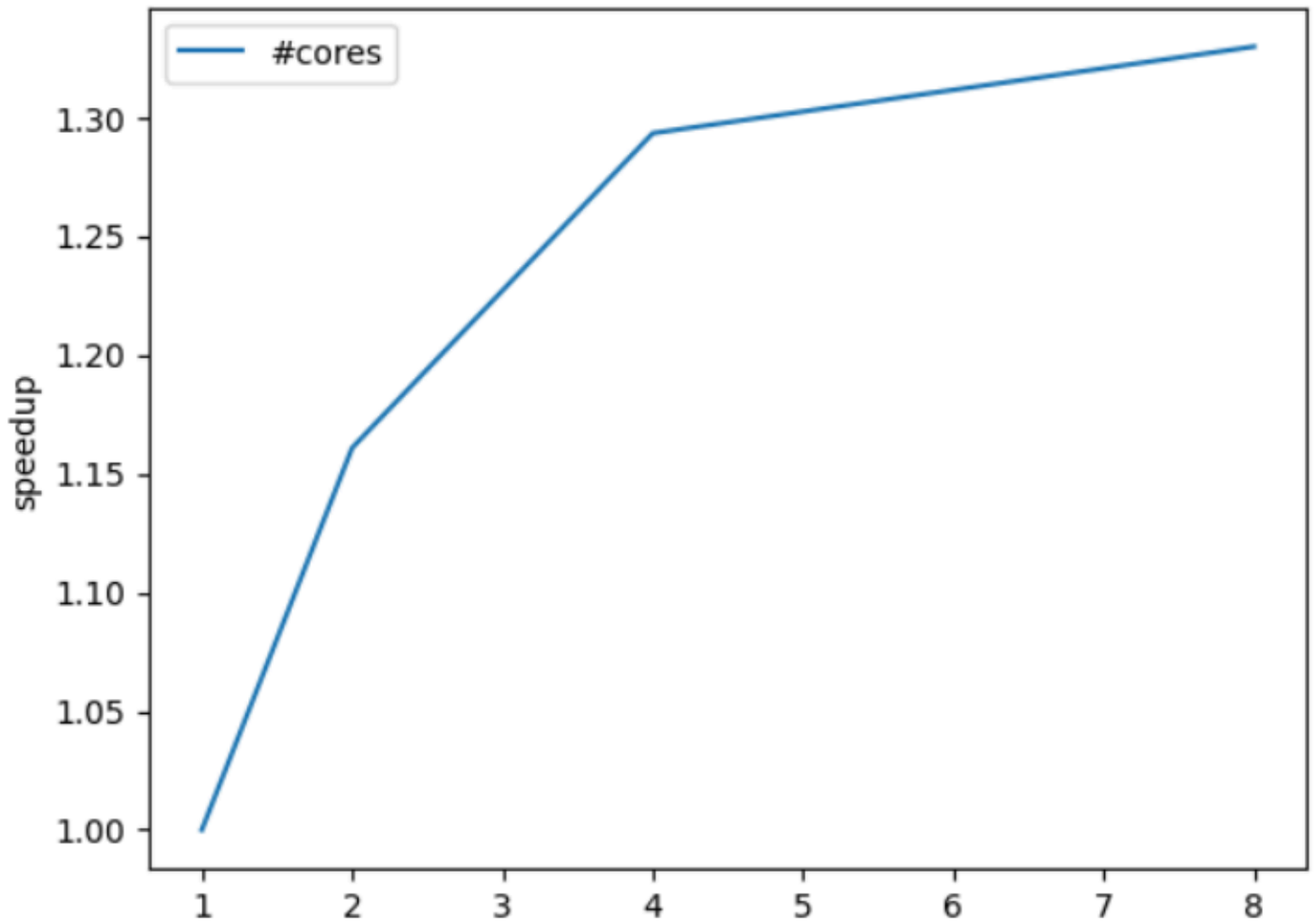
Activity — performance benchmark

- #cores using set experiments
  - **Set to maximum #rows**
  - **1 core - 2 cores - 4 cores …. your maximum #cores**
  - **Example**

```
import multiprocessing
parsed_description_split = np.array_split(parsed_description, multiprocessing.num_cpus)
with multiprocessing.pool.ThreadPool(c) as pool:
    pool.map(method_name, parsed_description_split)
```

Activity — performance benchmark

- Calculate speedup
  - Speedup = Execution time with 1 Cores / Execution time with N Core



Time for questions

Assignment

- Repeat the experiment from page #64-#67 using the full dataset
  - i.e., without limiting it to just 1000 rows.
  - Investigate whether the results maintain the same level of performance as observed when the dataset was restricted to 1000 rows, e.g., by visualizing the results
  - You have the flexibility to define a cutoff threshold, such as labeling durations longer than 2 or 3 hours as "too long."