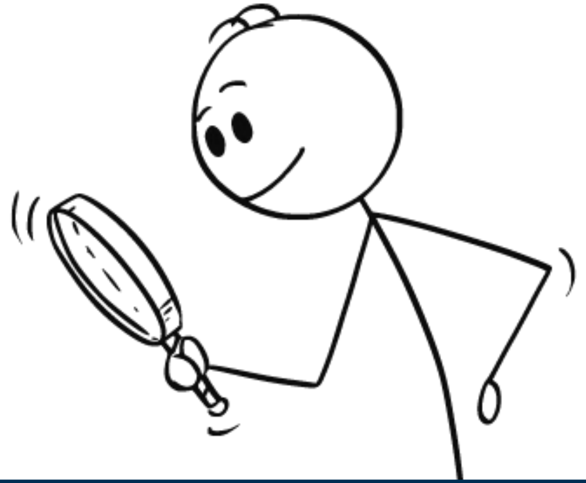


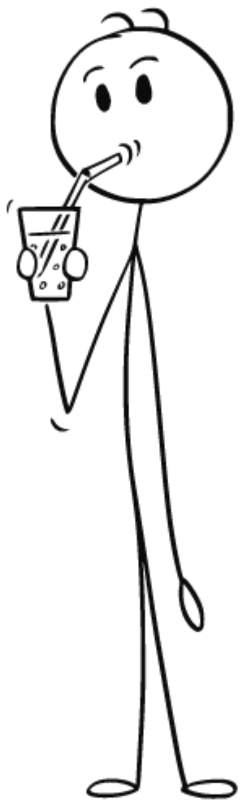
CSE 4/535
Information Retrieval

Sayantan Pal
PhD Student, Department of CSE
338Z Davis Hall

University at Buffalo
Department of CSE

Before we start

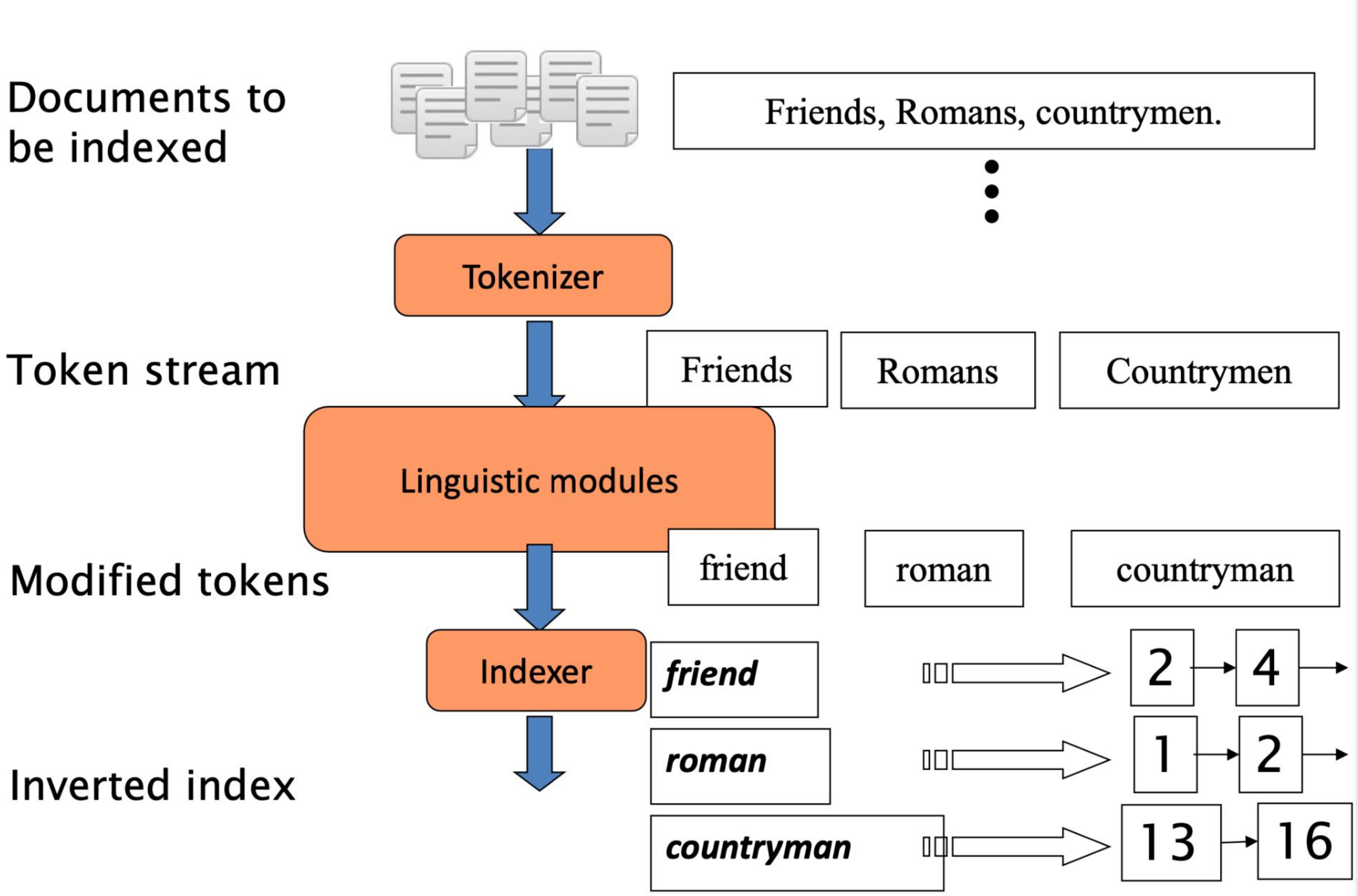
1. Project 1 released, due 27th September.
2. Join office hours if you have questions



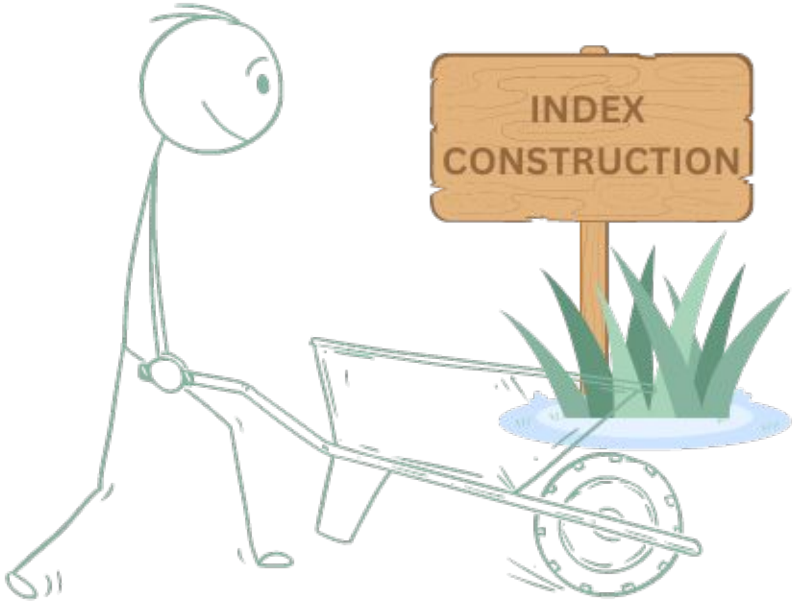
Recap - Previous Class

1. What is tolerant Retrieval?
2. Why spelling corrections are important?
3. Soundex

Recall the basic indexing pipeline



Index Construction



Index Construction… Hardwares?

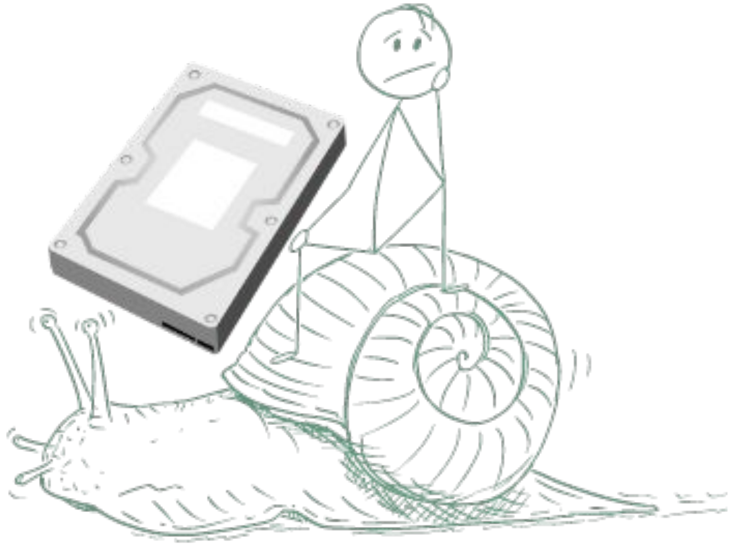
- How important is system memory (RAM)?

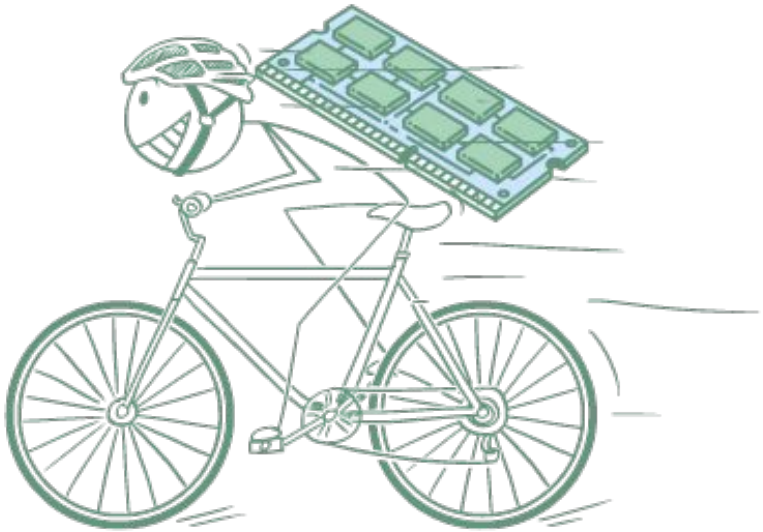
  Where are the index stored when you search?

  Do we have enough memory?

Index Construction… Hardwares?

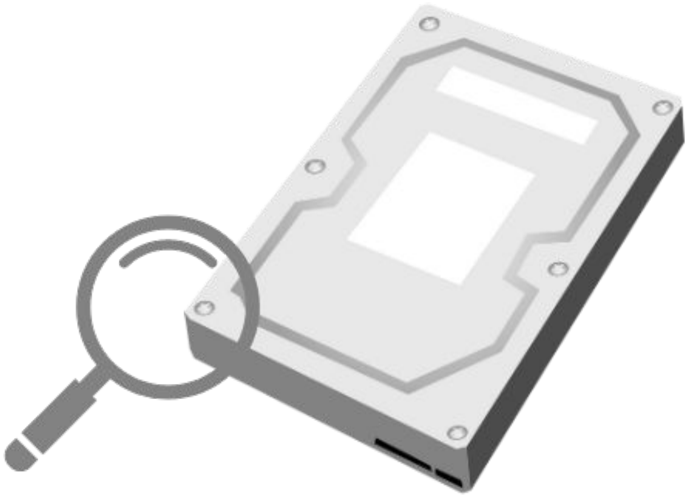
- How important is system memory (RAM)?
  Where are the index stored when you search?
  Do we have enough memory?
- Access to data in memory is much faster than access to data on disk.





Index Construction... Hardwares?

- How important is system memory (RAM)?
- 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.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.



Analogy - Reading a book…

Assume 2 situations:

1. You are reading a book and the information required is in the same page
2. You are reading a book and the information required is in different pages. The time required to find the page is similar to the disk seek time.

Let's talk about real world data

- RCV1 (Reuters Corpus Volume 1)
  - It is a collection of newswire articles produced by Reuters in 1996-1997.
  - It contains 804,414 manually labeled newswire documents.

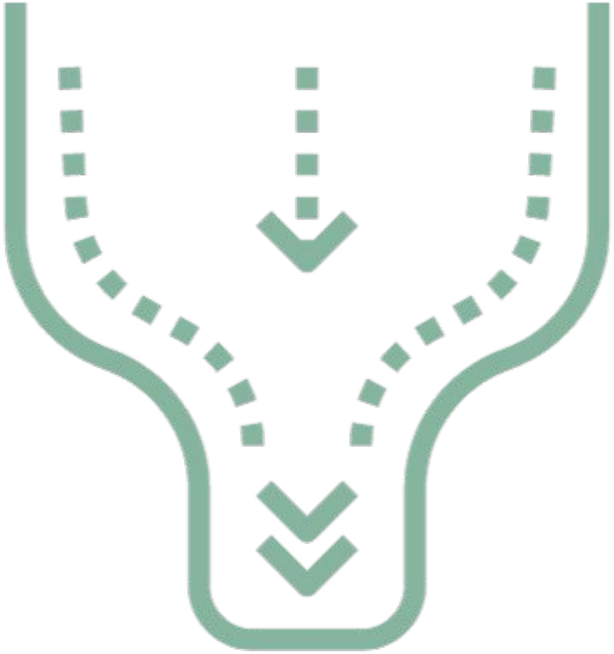
RCV1 Corpus Stats

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per document	200
M	term types	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term type	7.5
	non-positional postings	100,000,000

- Why #byter per token < #bytes per term type?
- What is non-positional posting?

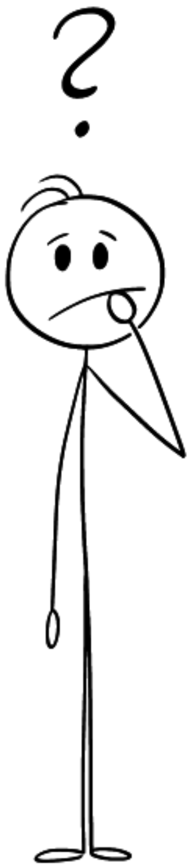
Main Bottleneck…

1. Except for small collections, basic sort cannot be done in memory
2. Need some form of external memory sort
3. Will also look at distributed and incremental versions



What if we do everything on disk?

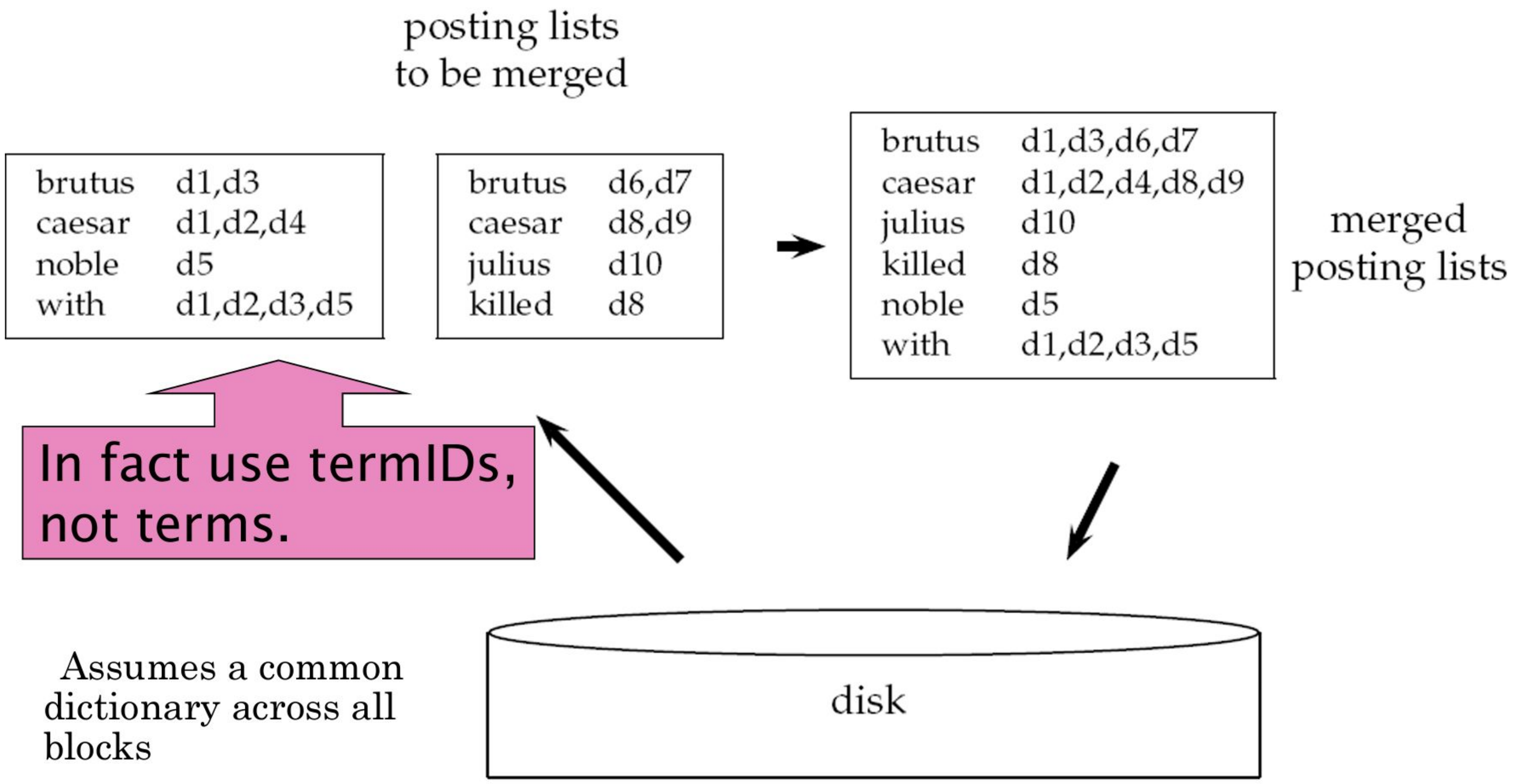
1. The disk seek time is in milliseconds
2. If you have 100 million records, you will end up in months to sort the records
3. Do we have a faster alternative?



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

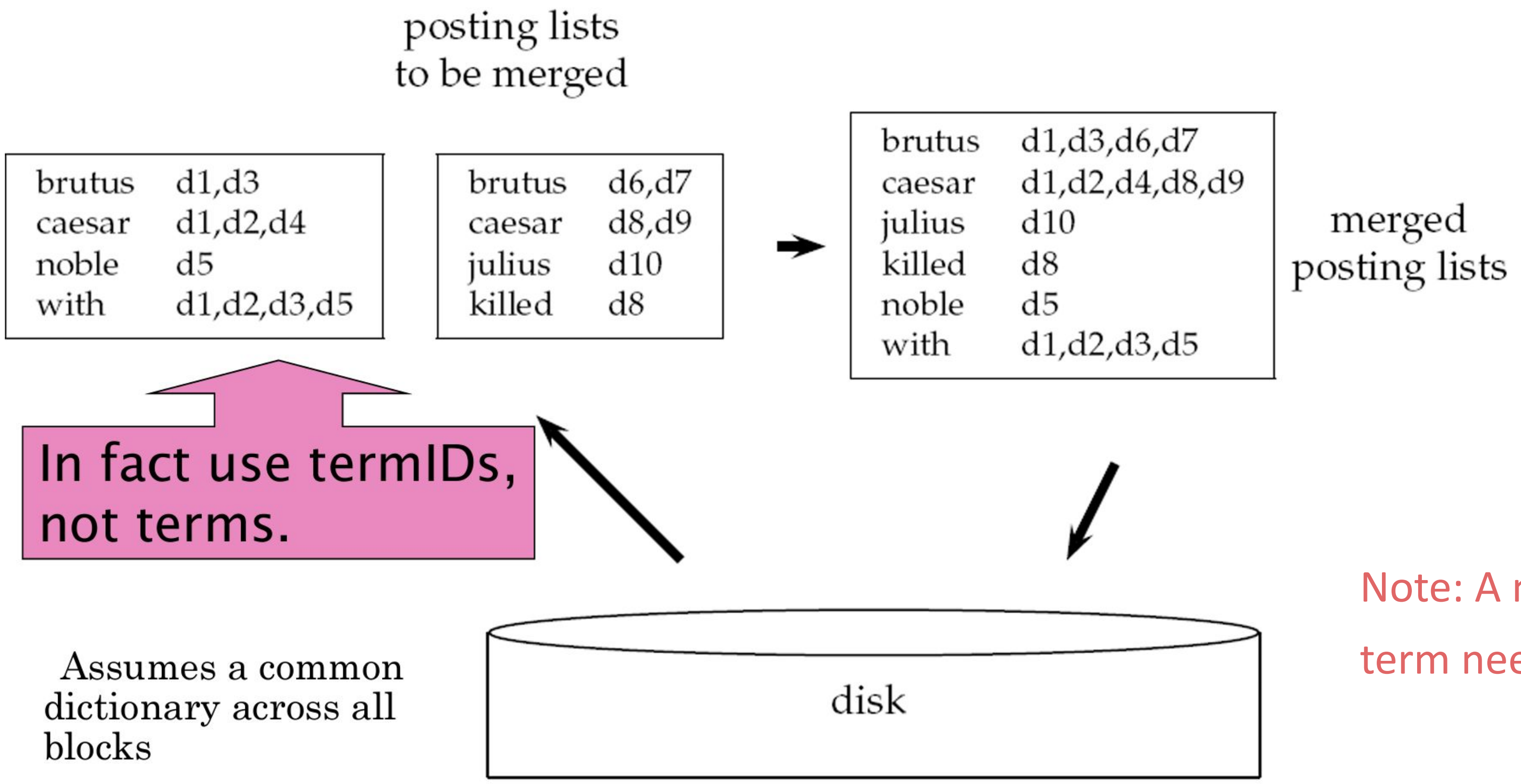
1. Basic idea of algorithm:
   a. We create blocks that fit into the memory
   b. Accumulate postings for each block, sort, write to disk.
   c. Then merge the blocks into one long sorted order.

BSBI - Blocked sort-based Indexing



Assumes a common dictionary across all blocks

BSBI - Blocked sort-based Indexing



Note: A mapping of term ids to term needs to be maintained

BSBI - Blocked sort-based Indexing

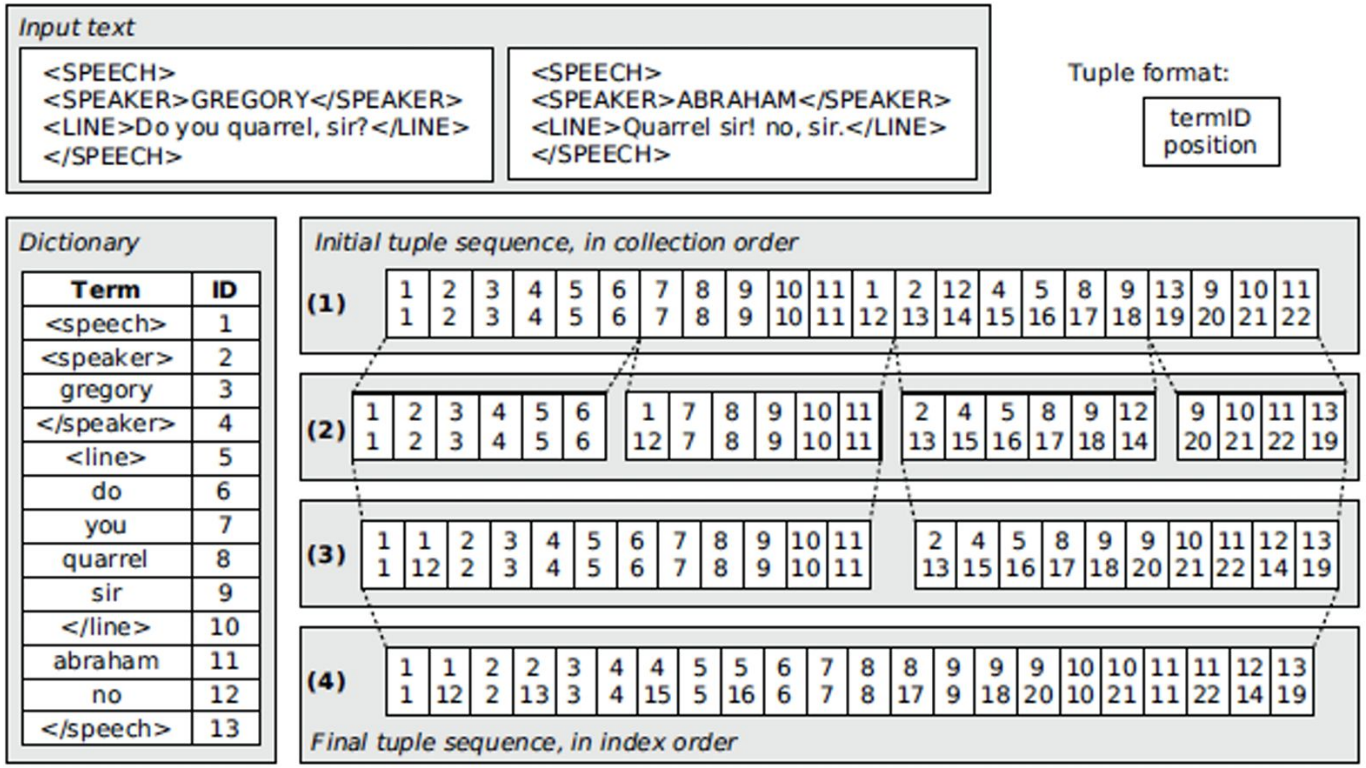


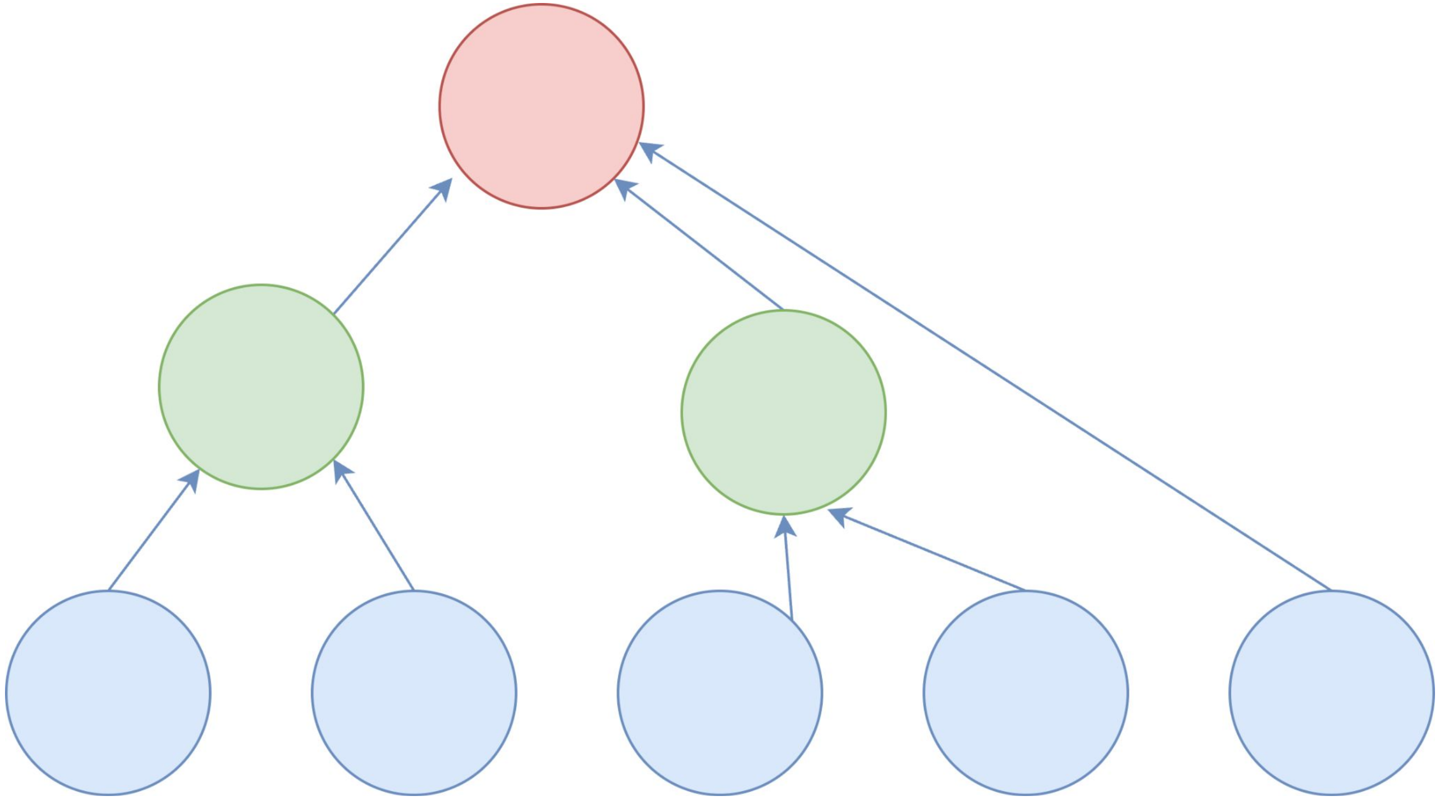
Figure 4.11 Sort-based index construction with global term IDs. Main memory is large enough to hold 6 (*termID*, *position*) tuples at a time. (1)→(2): sorting blocks of size ≤ 6 in memory, one at a time. (2)→(3) and (3)→(4): merging sorted blocks into bigger blocks.

BSBI - Blocked sort-based Indexing - Algorithm

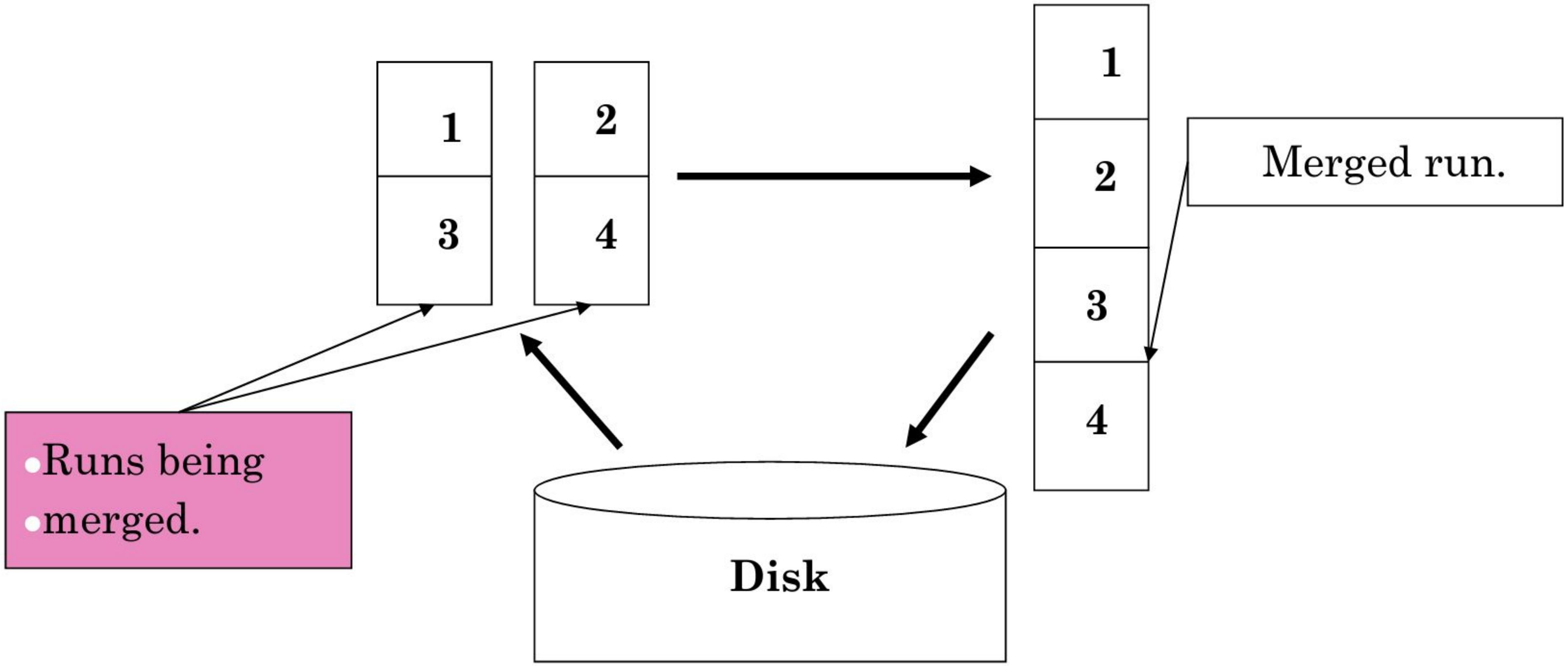
```
BSBIndexConstruction()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block)
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_merged)
```

BSBI - Blocked sort-based Indexing - Merging

1. Two ways
   a. Binary pairs and merge (has more disk seeks)
   b. Priority Queue or Multiway Merge where you are reading from all blocks simultaneously



BSBI - Blocked sort-based Indexing - Binary Merging

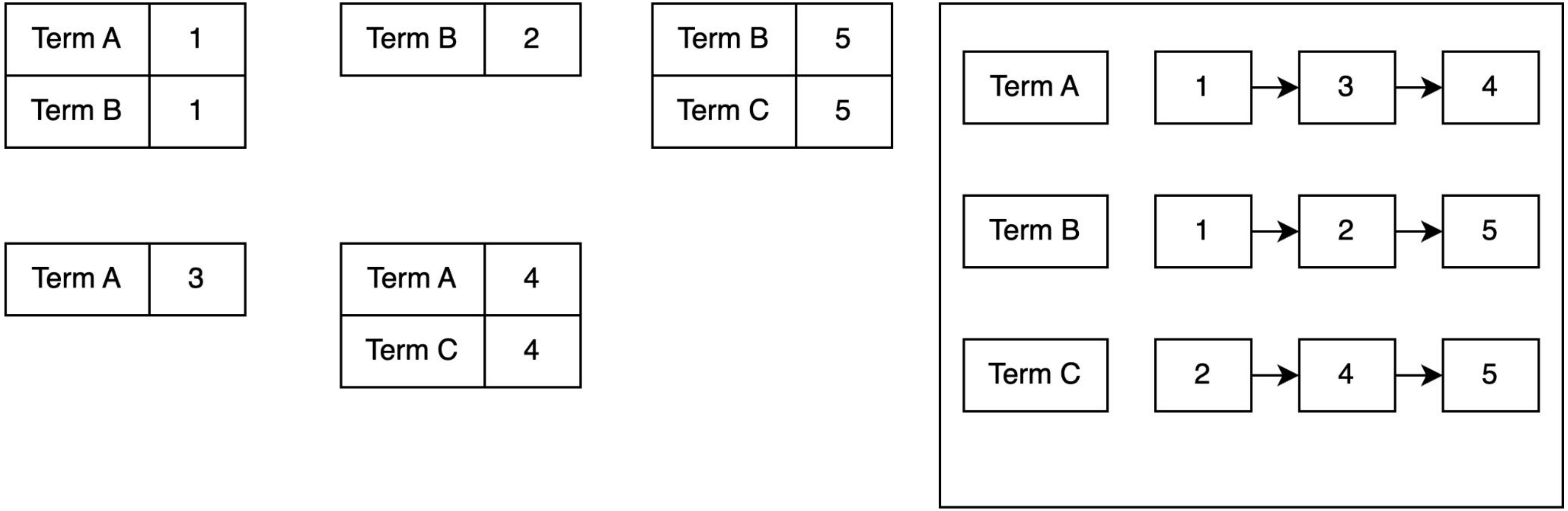


BSBI - Blocked sort-based Indexing - N-way Merging

1. But it is more efficient to do a n-way merge, where you are reading from all blocks simultaneously
2. Providing you read decent-sized chunks of each block into memory, you're not killed by disk seeks

Single pass In-memory Indexing (SPIMI)

1. Key Idea: Generate separate dictionaries for each block, no need for term IDs
2. No sorting needed, accumulate postings as they occur
3. Finally merge
4. TC: O(T)



Single pass In-memory Indexing (SPIMI) - Algorithm

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
   if term(token) ∉ dictionary
     then postings_list = ADDTODICTIONARY(dictionary, term(token))
     else postings_list = GETPOSTINGSLIST(dictionary, term(token))
   if full(postings_list)
     then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
   ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Merging of blocks is analogous to BSBI.

Indexing Times

Table 4.7 Building a schema-independent index for various text collections, using merge-based index construction with 512 MB of RAM for the in-memory index. The indexing-time dictionary is realized by a hash table with 2^{16} entries and move-to-front heuristic. The extensible in-memory postings lists are unrolled linked lists, linking between groups of postings, with a pre-allocation factor $k = 1.2$.

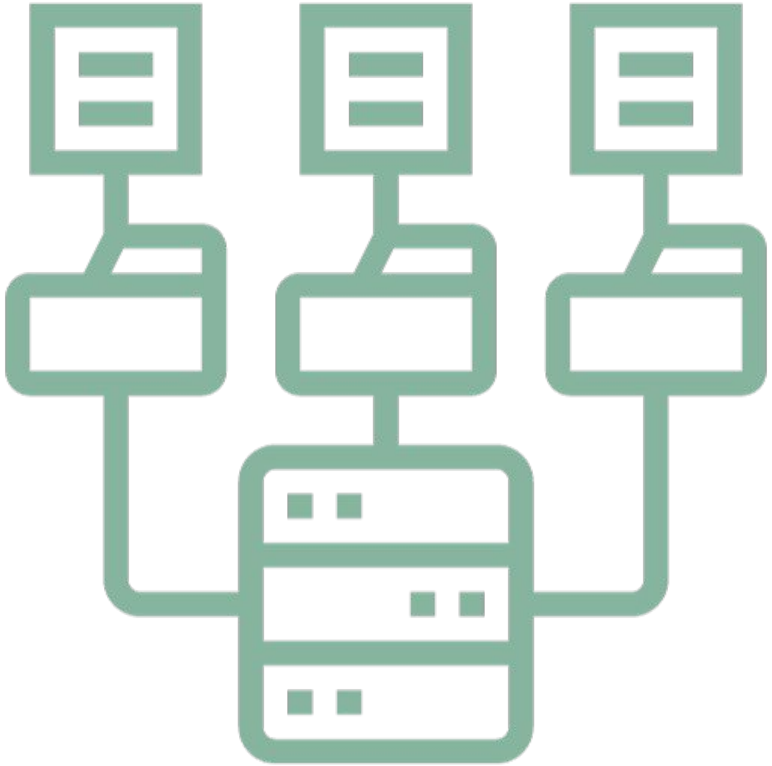
	Reading, Parsing & Indexing	Merging	Total Time
Shakespeare	1 sec	0 sec	1 sec
TREC45	71 sec	11 sec	82 sec
GOV2 (10%)	20 min	4 min	24 min
GOV2 (25%)	51 min	11 min	62 min
GOV2 (50%)	102 min	25 min	127 min
GOV2 (100%)	205 min	58 min	263 min

GOV2 is 426GB of text; using SPIMI

Note that parsing, dictionary construction and indexing all happen in conjunction with each other

Distributed Indexing

1. Data that cannot fit into a single hard disk :(
2. For web-scale indexing (don’t try this at home!)
    a. must use a distributed computing cluster
3. Individual machines are fault-prone
    a. Can unpredictably slow down or fail
4. How do we exploit such a pool of machines?



Google data centers

1. Google data centers mainly contain commodity machines.
2. Data centers are distributed around the world (around 15 data centers?)
3. Estimate: a total of 2.5 million servers?
4. Estimate: Google installs 100,000 servers each quarter.
    a. Spent $30B on data centers from 2014-2017
5. This would be 10% of the computing capacity of the world!?!

Google data centers

1. Google data centers mainly contain commodity machines.
2. Data centers are distributed around the world (around 15 data centers?)
3. Estimate: a total of 2.5 million servers?
4. Estimate: Google installs 100,000 servers each quarter.
   a. Spent $30B on data centers from 2014-2017
5. This would be 10% of the computing capacity of the world!?!

Exercise: If in a non-fault-tolerant system with 1,000 nodes, each node has 99.9% uptime, what is the uptime of the system?

Answer: ~37%

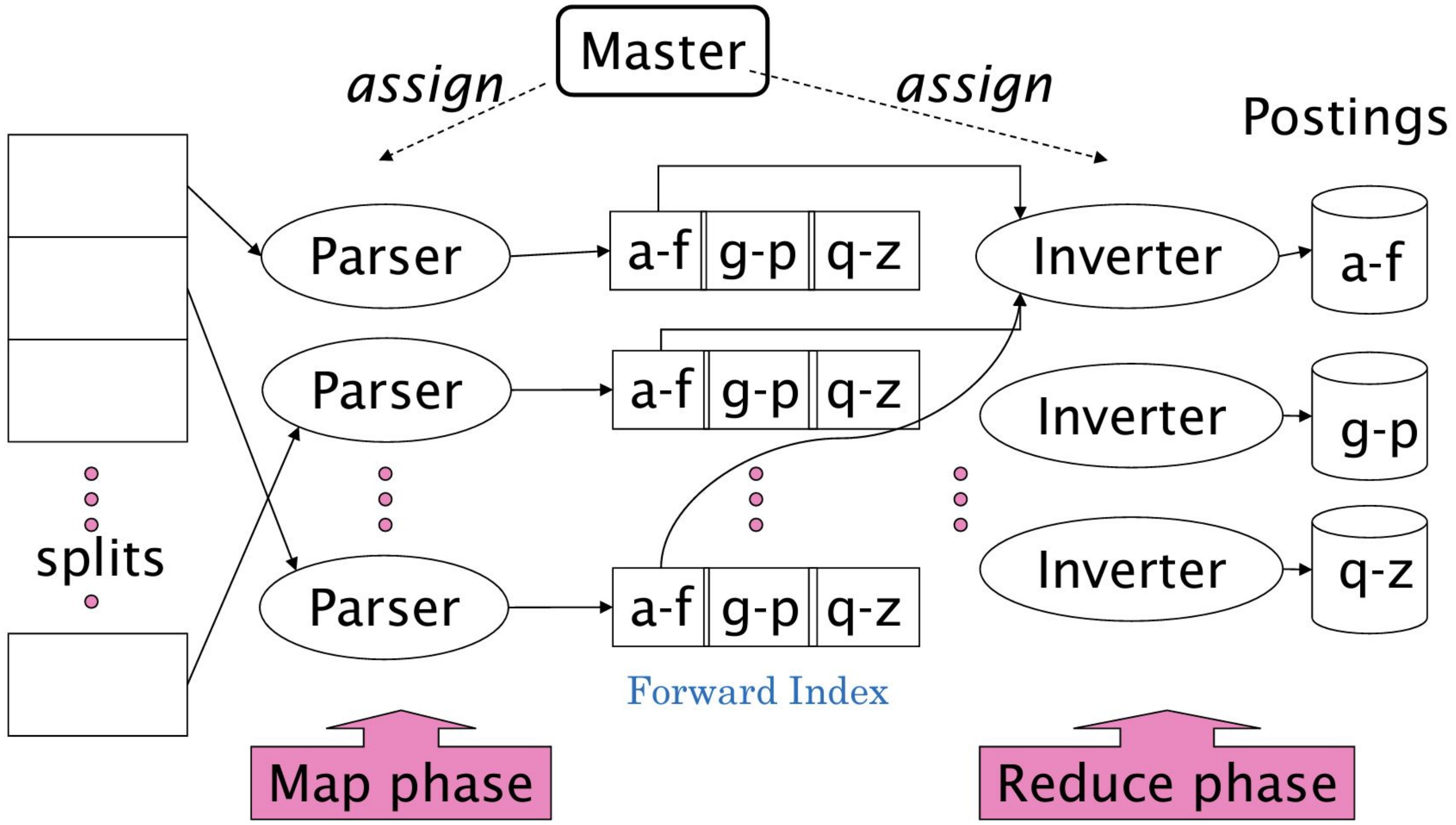
Parallel tasks

- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into splits
- Each split is a subset of documents
- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits <termID,docID> pairs
- Parser writes pairs into j partitions
- Each for a range of terms' first letters
  - (e.g., a-f, g-p, q-z) here j=3.

Inverter

- Collect all (term, doc) pairs for a partition
- Sorts and writes to postings list
- Each partition contains a set of postings

Data Flow


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

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.
- Assumption for the rest of the lecture: The index is one big file.
- 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 I0(if I_0 already exists) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
- Or merge with I_1 to form Z_2, etc.
- Search Request: query in memory Z_0 and all currently valid indexes I_i on disk; merge results

Logarithmic Merge Algorithm

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

```
Z_0 ← Merge(Z_0, {token})
if |Z_0| = n
  then for i ← 0 to ∞
         do if I_i ∈ indexes
              then Z_{i+1} ← Merge(I_i, Z_i)
                     (Z_{i+1} is a temporary index on disk.)
                     indexes ← indexes − {I_i}
              else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
                     indexes ← indexes ∪ {I_i}
                     Break
         Z_0 ← ∅
```

$\textsc{LogarithmicMerge}()$

```
Z_0 ← ∅    (Z_0 is the in-memory index.)
indexes ← ∅
while  true
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

Further issues with multiple indexes

- Corpus-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking

References

1. Slides provided by Sougata Saha (Instructor, Fall 2022 - CSE 4/535)
2. Materials provided by Dr. Rohini K Srihari
3. https://nlp.stanford.edu/IR-book/information-retrieval-book.html