Modern Information Retrieval

Index Construction

Hamid Beigy

Sharif university of technology

February 21, 2025

Table of contents

Introduction

Inverted index

1. The goal is constructing inverted index

For each term t, we store a list of all documents that contain t.

dictionary		postings								
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					
$\vdots$										

RCV1 collection

1. An example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
2. English newswire articles sent over the wire in 1995 and 1996 (a year).
3. RCV1 statistics
   - Number of documents (N): 800,000
   - Number of tokens per document (L): 200
   - Number of distinct terms (M) : 400,000
   - Bytes per token (including spaces): 6
   - Bytes per token (without spaces): 4.5
   - Bytes per term: 7.5
4. Why does the algorithm given in previous sections not scale to very large collections?

Sort-based index construction

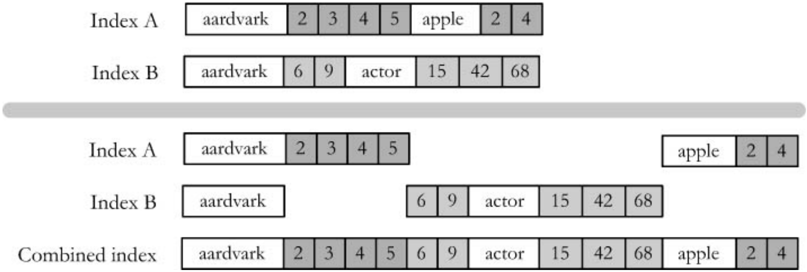
Sort-based index construction

1. As we build index, we parse docs one at a time.
2. The final postings for any term are incomplete until the end.
3. Can we keep all postings in memory and then do the sort in-memory at the end?

   No, not for large collections

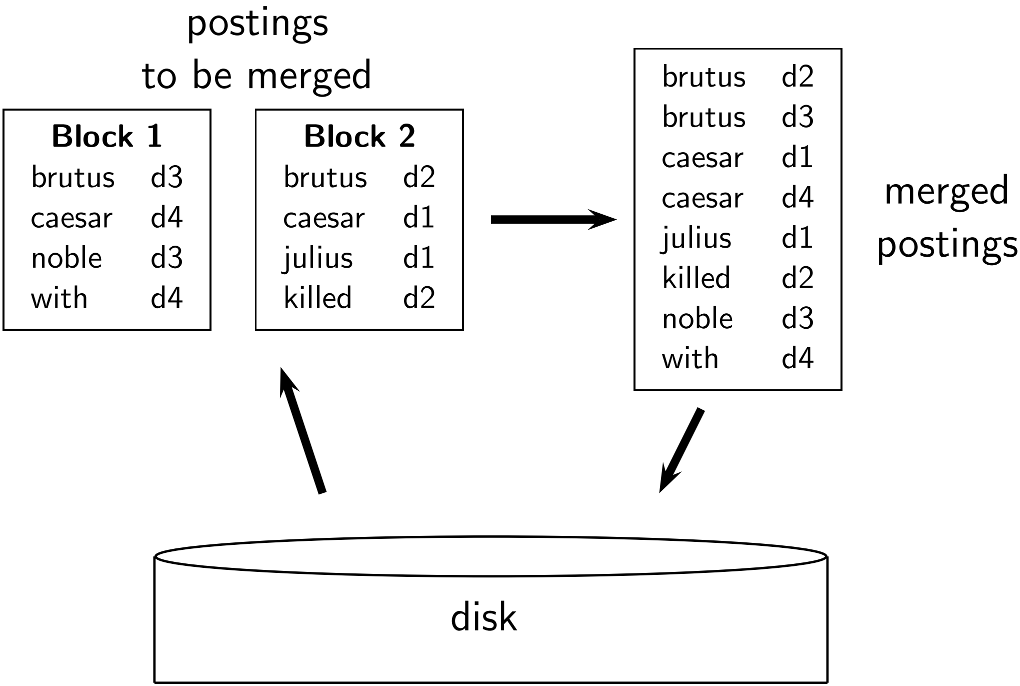
   Thus: We need to store intermediate results on disk.
4. Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?

   No: Sorting very large sets of records on disk is too slow– too many disk seeks.
5. We need an external sorting algorithm.



External sorting algorithm

1. We must sort $T = 100{,}000{,}000$ non-positional postings.
2. Each posting has size 12 bytes (4+4+4: termID, docID, term frequency).
3. Define a block to consist of 10,000,000 such postings
4. We can easily fit that many postings into memory.
5. Basic idea of algorithm:
6. For each block do
   - accumulate postings
   - sort in memory
   - write to disk
7. Then merge the blocks into one long sorted order.


postings
to be merged
Block 1
brutus d3
caesar d4
noble d3
with d4
Block 2
brutus d2
caesar d1
julius d1
killed d2
brutus d2
brutus d3
caesar d1
caesar d4
julius d1
killed d2
noble d3
with d4
merged
postings
disk

Problem with sort-based algorithm

1. The assumption was: we can keep the dictionary in memory.
2. We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
3. Actually, we could work with term,docID postings instead of termID,docID postings . . .
4. The intermediate files become very large.
5. **We would end up with a scalable, but very slow index construction method.**

Single–pass in-memory indexing (SPIMI)

Single–pass in-memory indexing (SPIMI)

1. **Key idea 1:** Generate separate dictionaries for each block - no need to maintain term–termID mapping across blocks.
2. **Key idea 2:** Don't sort. Accumulate postings in postings lists as they occur.
3. With these two ideas we can generate a complete inverted index for each block.
4. These separate indexes can then be merged into one big index.
5. **Compression makes SPIMI even more efficient.**
    - Compression of terms
    - Compression of postings

```
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.

How can we index web-scale data?

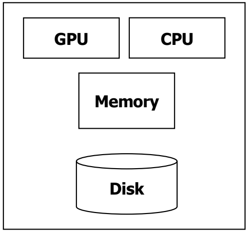
1. For web-scale indexing: **must use a distributed computer system**
2. Individual machines are fault-prone.

   Can unpredictably slow down or fail.
3. **How do we exploit such a pool of machines?**
4. Distributed index is partitioned across several machines - either
   - according to term or
   - according to document.

Distributed indexing

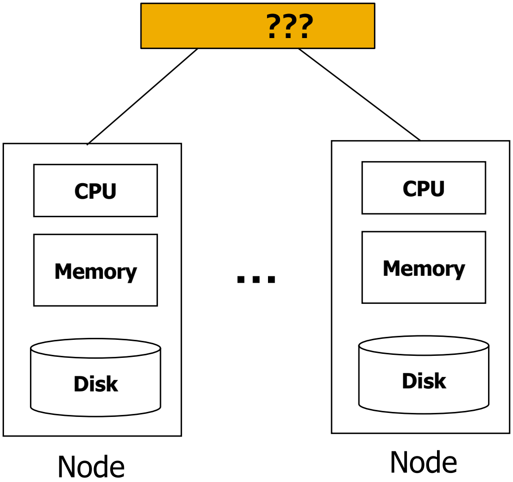
Google Example

1. **Crawling** and **indexing** the **web pages**.
   - The number of web pages: **10 billion**
   - Average size of web page: **20 KB**
   - The average size of the whole data: **200 TB**
   - The data is stored on a single disk and tends to be processed in CPU.

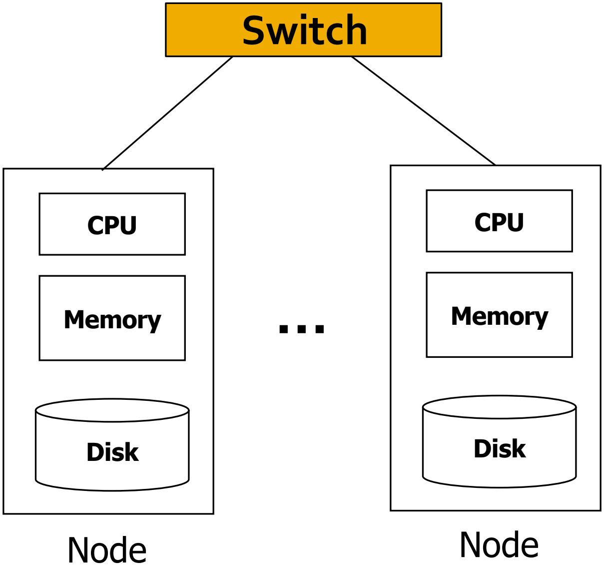


   - One computer reads 50 MB/sec from disk (disk read bandwidth).
   - Time to read: **4 million seconds $\approx$ 46 days**
2. **Even longer to do useful things with the data**.
3. Solution: **Cluster computing**

1. Every rack has **42-48 units**, containing **16-64 nodes**.
2. **Ex.** Each computer is **commodity Linux nodes**.

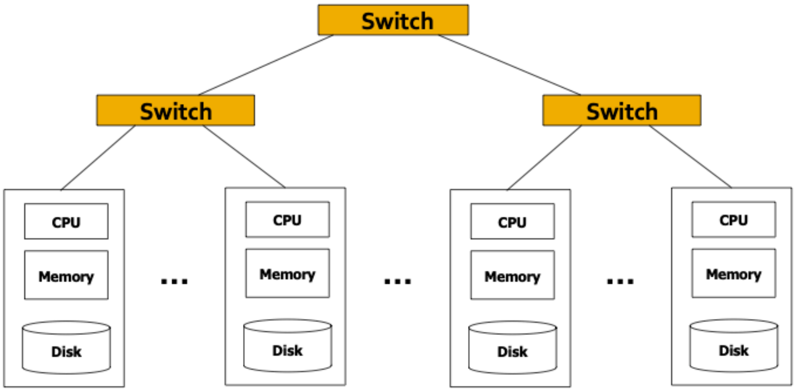


1. **Switch** connecting nodes
2. **Ex.** **10 GB/sec** bandwidth between any pair of nodes in a rack



Cluster architecture

1. **Backbone switch** connecting racks
2. **Ex.** **100 GB/sec** bandwidth between racks.



Cluster architecture

Node failures

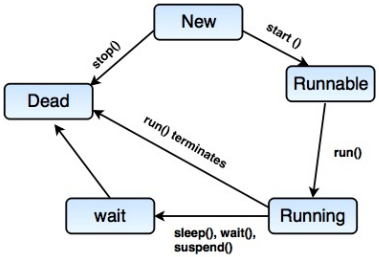
- one server can stay up 3 years (1,000 days)
- 1,000 servers in cluster → 1 failure/day
- 1M servers in cluster → 1,000 failure/day
- **What does it happen to its data and its computations?**

Network bottleneck

- Let network bandwidth: **1 GB/sec**
- Time for moving 10TB data: **1 day**

Distributed/parallel programming is hard

- Consider the life-cycle for Java threads.
- **A programming model that hides most of the complexity**.



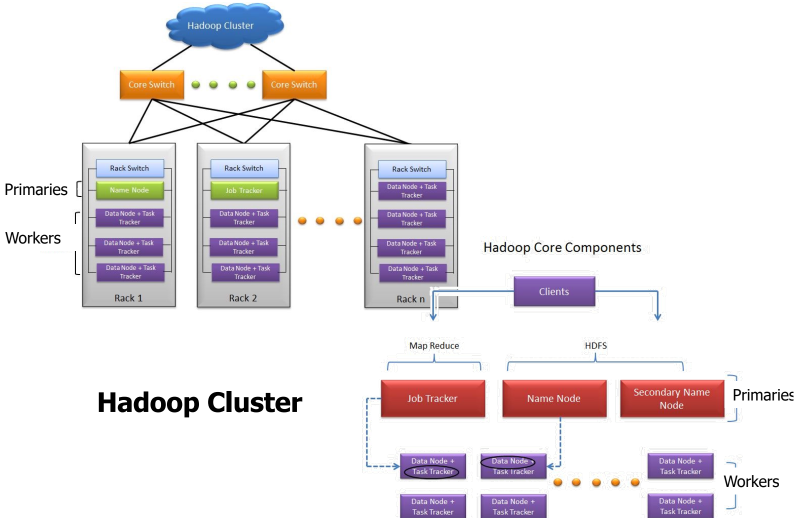
Map-Reduce addresses the challenges

Google data centers (Gartner estimates)

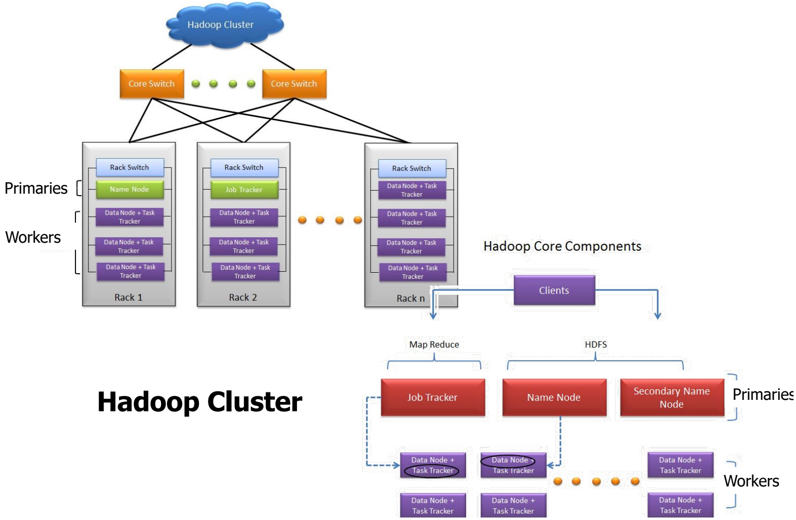
1. Google data centers mainly contain commodity machines. Data centers are distributed all over the world.
2. 1 million servers, 3 million processors/cores
3. Google installs 100,000 servers each quarter.
4. Based on expenditures of 200–250 million dollars per year. This would be 10% of the computing capacity of the world!
5. If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system (assuming it does not tolerate failures)?
6. Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?
7. Answer: Less than two minutes.

Map-Reduce addresses the challenges

1. **Node failure**: Store data redundantly on multiple nodes
2. **Network bottleneck**: Move computation close to data to minimize data movement
3. **Distributed programming:** Map function and Reduce functions



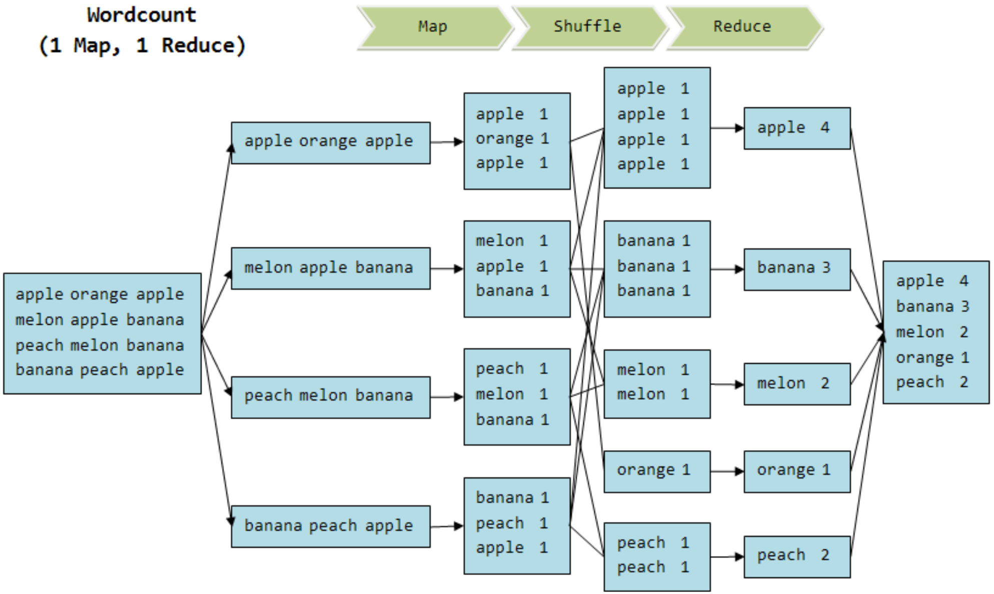
1. Maintain a master machine directing the indexing job – considered "safe"
2. Break up indexing into sets of parallel tasks: **Map** and **Reduce**
3. Master machine assigns each task to an idle machine from a pool.



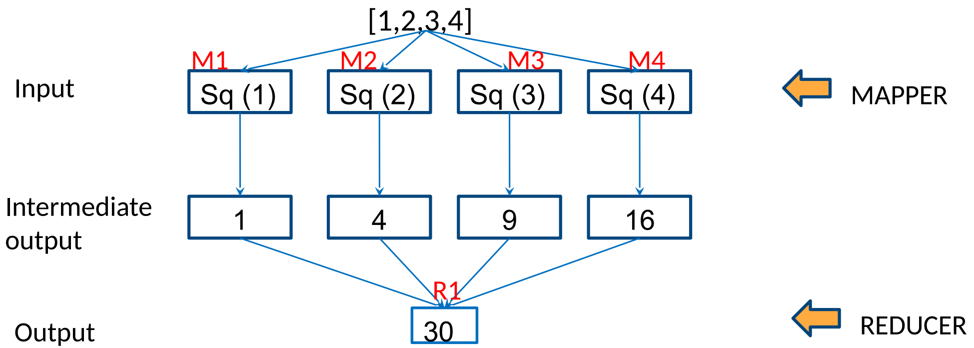
```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)


reduce(key, values):
// key: a word; value: an iterator over counts
        result = 0
        for each count v in values:
                result += v
        emit(result)
```

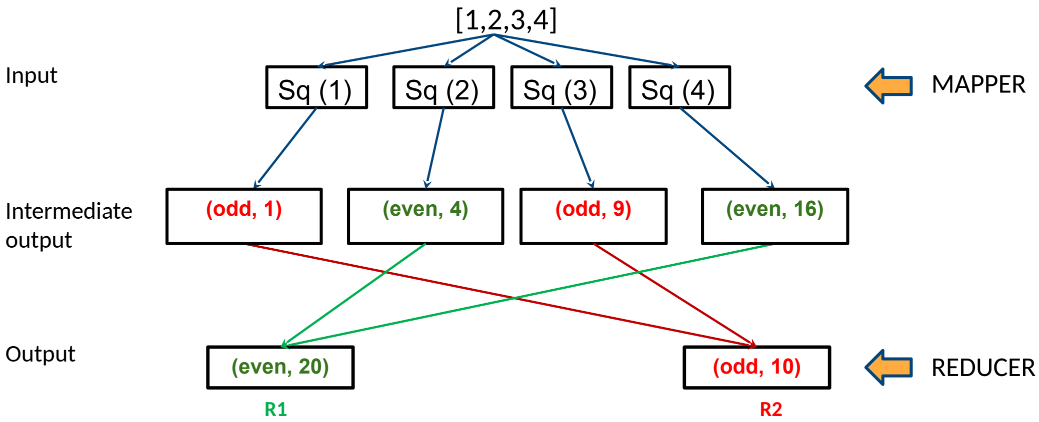
Map-Reduce: word count example



Map-Reduce: Sum of squares



Map-Reduce: Sum of squares of even and odd

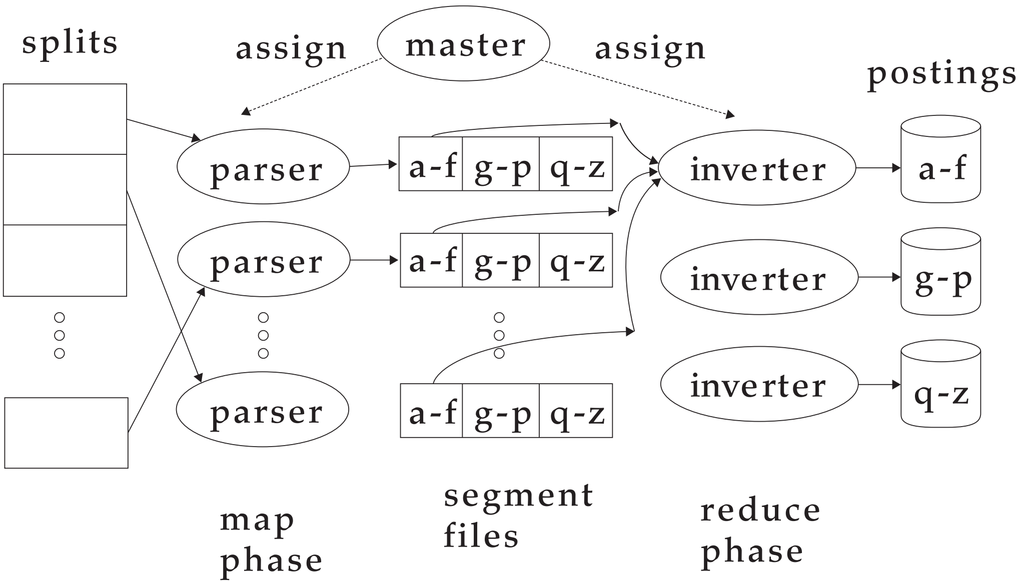


We will define two sets of parallel tasks and deploy two types of machines to solve them: Parsers and Inverters

1. **Parsers**
   -
   - Master assigns a split to an idle parser machine.
   - Parser reads a document at a time and emits (term,docID)-pairs.
   - Parser writes pairs into j term-partitions. Each for a range of terms' first letters

     E.g., a-f, g-p, q-z (here: $j = 3$)
2. **Inverters**
   - An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
   - Sorts and writes to postings lists

Map-Reduce



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

Dynamic indexing

Dynamic indexing

1. Up to now, we have assumed that collections are static.
2. They rarely are: Documents are inserted, deleted and modified.
3. This means that the dictionary and postings lists have to be dynamically modified.

Dynamic indexing: simplest approach

1. Maintain big main index on disk
2. New docs go into small auxiliary index in memory.
3. Search across both, merge results
4. Periodically, merge auxiliary index into big index
5. Deletions:
   - Invalidation bit-vector for deleted docs
   - Filter docs returned by index using this bit-vector
6. **Issues with auxiliary and main index**
   - Frequent merges
   - Poor search performance during index merge

Logarithmic merge

1. Logarithmic merging amortizes the cost of merging indexes over time.

   Users see smaller effect on response times.

2. Maintain a series of indexes, each twice as large as the previous one.
3. Keep smallest (Z_0) in memory
4. Larger ones ($I_0, I_1, \ldots$) on disk
5. If Z_0 gets too big ($> n$), write to disk as I_0 or merge with I_0 (if I_0 already exists) and write merger to I_1 etc.

Logarithmic merge

$\text{LMERGEADDTOKEN}(\mathit{indexes}, Z_0, \mathit{token})$

$Z_0 \leftarrow \text{MERGE}(Z_0, \{\mathit{token}\})$
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
    do if $I_i \in \mathit{indexes}$
      then $Z_{i+1} \leftarrow \text{MERGE}(I_i, Z_i)$
        (Z_{i+1} is a temporary index on disk.)
        $\mathit{indexes} \leftarrow \mathit{indexes} - \{I_i\}$
      else $I_i \leftarrow Z_i$ *(Z_i becomes the permanent index I_i.)*
        $\mathit{indexes} \leftarrow \mathit{indexes} \cup \{I_i\}$
        BREAK
    $Z_0 \leftarrow \emptyset$

$\text{LOGARITHMICMERGE}()$

$Z_0 \leftarrow \emptyset$ *(Z_0 is the in-memory index.)*
$\mathit{indexes} \leftarrow \emptyset$
while true
do $\text{LMERGEADDTOKEN}(\mathit{indexes}, Z_0, \text{GETNEXTTOKEN}())$

References

Reading

1. Chapters 4 of Information Retrieval Book[1]
2. Section 5.6 of Search Engines - Information Retrieval in Practice Book[2]

[1] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze (2008). *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press.

[2] W. Bruce Croft, Donald Metzler, and Trevor Strohman (2009). *Search Engines - Information Retrieval in Practice*. Pearson Education.

References

Croft, W. Bruce, Donald Metzler, and Trevor Strohman (2009). *Search Engines - Information Retrieval in Practice*. Pearson Education.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008). *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press.

Questions?