CS3245

Information Retrieval

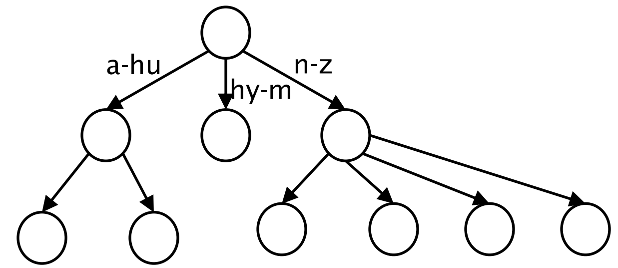
Lecture 5: Index Construction

5

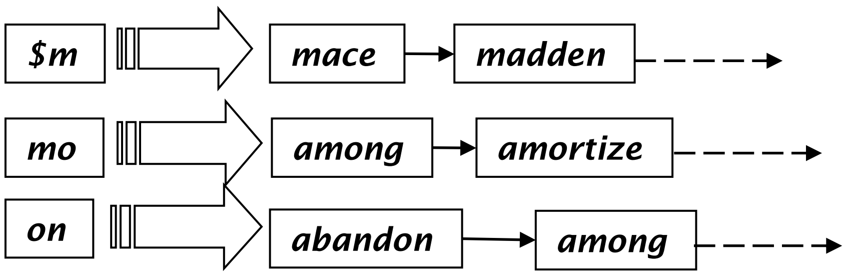
Live Q&A
https://pollev.com/jin

Last Time

- Dictionary data structures



- Tolerant retrieval
  - Wildcards
  - Spelling correction
  - Soundex



Today: Index construction

- How to make index construction scalable?
    1. BSBI (simple method)
    2. SPIMI (more realistic)
    3. Distributed Indexing

- How to handle changes to the index?
    1. Dynamic Indexing

Hardware basics

Many design decisions in information retrieval are based on the characteristics of hardware

Especially with respect to the bottleneck:

Hard Drive Storage

- Seek Time – time to move to a random location
- Transfer Time – time to transfer a data block

Hardware basics

- 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.
- Memory is ***much*** faster but **limited in quantity**.
  - Servers used in IR systems now typically have hundreds of GB of main memory.
  - Available disk space is several (2–3) orders of magnitude larger.

Hardware assumptions

symbol	statistic	value
s	average seek time	8 ms = 8×10^{-3} s
b	transfer time per byte	0.006 μs = 6×10^{-9} s
	processor’s clock rate	34^{9} s^{-1} (Intel i7 6th gen)
p	low-level operation (e.g., compare & swap a word)	0.01 μs = 10^{-8} s
	size of main memory	8 GB or more
	size of disk space	1 TB or more

Stats from a 2016 HP Z Z240 3.4GHz Black SFF i7-6700

Hardware assumptions (Flash SSDs)

symbol	statistic	value
s	average seek time	.1 ms = 1×10^{-4} s
b	transfer time per byte	0.002 μs = 2×10^{-9} s

100x faster seek,
3x faster transfer time.
(But price 8x more per GB of storage)

WD 4 TB Black
S$ 311 (circa Jan 2016)

Samsung 850 Evo (1 TB)
S$ 630 (circa Jan 2016)

Seek and transfer time combined in another industry metric: [IOPS]

RCV1: Our collection for this lecture

- The successor to the Reuters-21578, which you used for your homework assignment. Larger by 35 times.
  - Not really large, but publicly available and a more plausible example.

- One year of Reuters newswire (part of 1995 and 1996)

Reuters RCV1 statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= vocabulary size)	400,000
	avg. # bytes per term	$\boxed{7.5}$
T	term-docID pairs (= tokens)	$\boxed{100{,}000{,}000}$

Key Step in Index Construction

- Sort by terms
  - And then docID

We focus on this sort step.
We have 100M pairs to sort.

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

→

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Scaling index construction

- At **~11.5** bytes per pair: ~7.5 bytes for term + 4 bytes for docID

- T = 100M in the case of RCV1: ~1.1GB
  - So … we can do this easily in memory nowaday, but typical collections are much larger. E.g. the *New York Times* provides an index of >150 years of newswire

- Thus, we need to store intermediate results **on disk**.

BSBI: Blocked sort-based Indexing

- Map **terms** to **termIDs** of 4 bytes with an **in-memory dictionary**.
- 8-byte (4+4) records *(termID, docID)*
- Must now sort 100M 8-byte records (~0.8 GB) by *termID.*

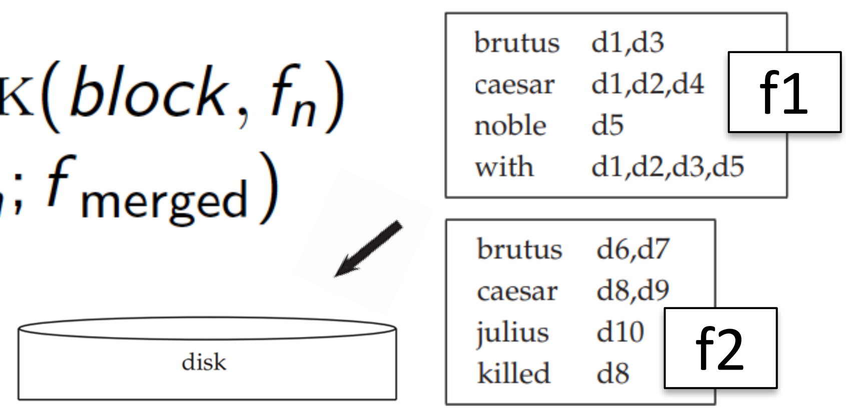
BSBI: Blocked sort-based Indexing

- Define a <u>Block</u> as ~ **10M** such records
  - Can easily fit a couple into memory.
  - Will have **10** such blocks for our collection.

- Basic idea of algorithm:
  - Accumulate records for each block, sort, create the posting lists, write to disk.
  - Then merge the blocks into one long sorted order.

```
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)
```

(The actual terms are shown for clarity.)



Example of Merging in BSBI

postings lists
to be merged

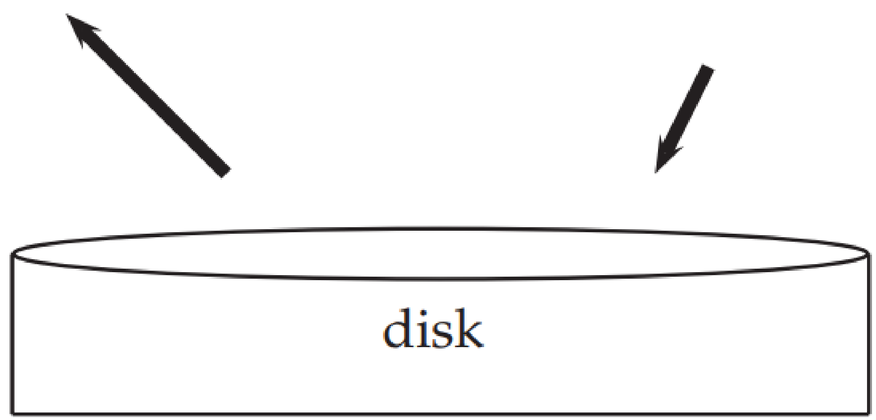
brutus	d1,d3
caesar	d1,d2,d4
noble	d5
with	d1,d2,d3,d5

brutus	d6,d7
caesar	d8,d9
julius	d10
killed	d8

→

brutus	d1,d3,d6,d7
caesar	d1,d2,d4,d8,d9
julius	d10
killed	d8
noble	d5
with	d1,d2,d3,d5

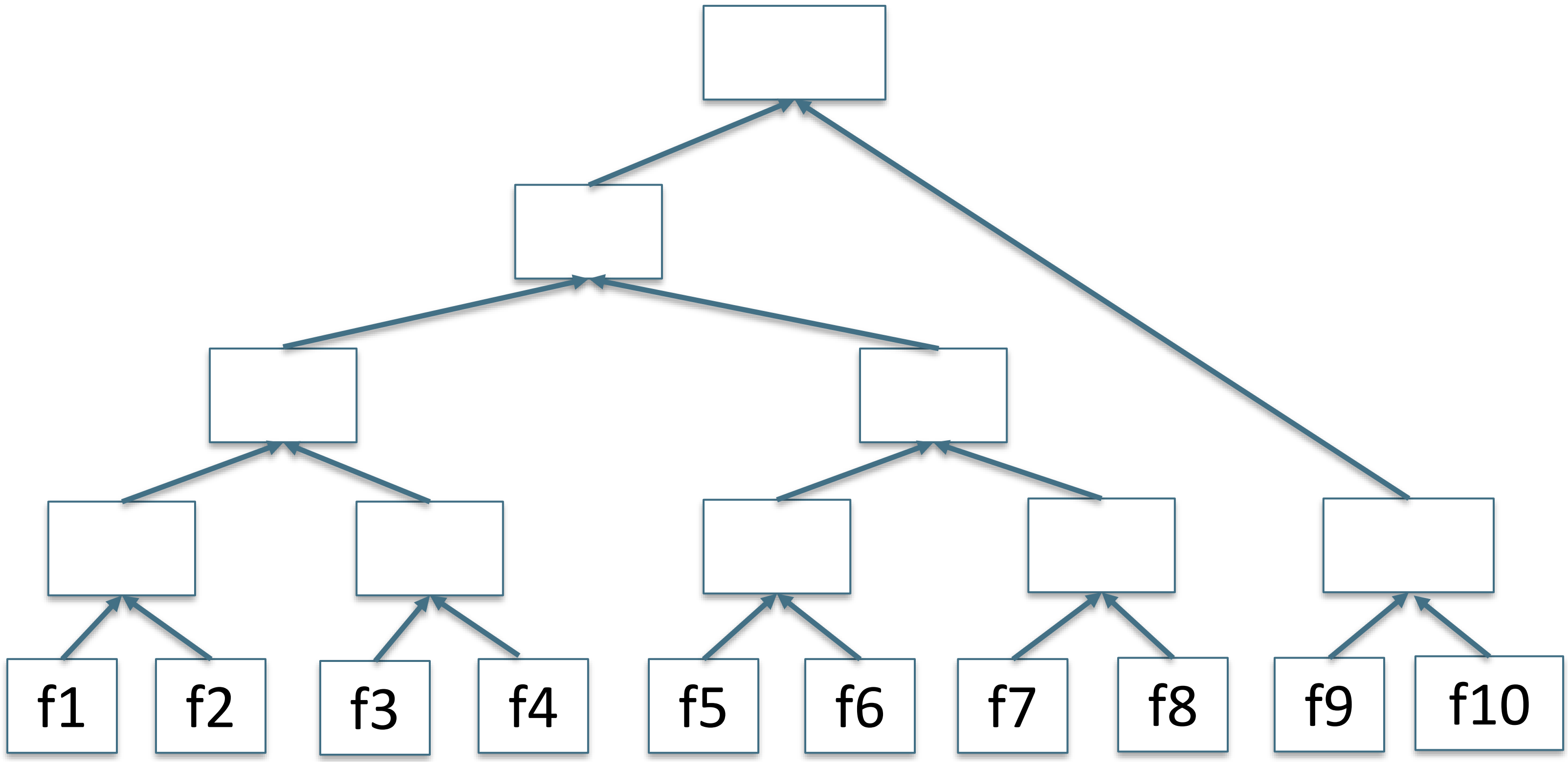
merged
postings lists



(The actual terms are shown for clarity.)

How to merge the sorted runs?

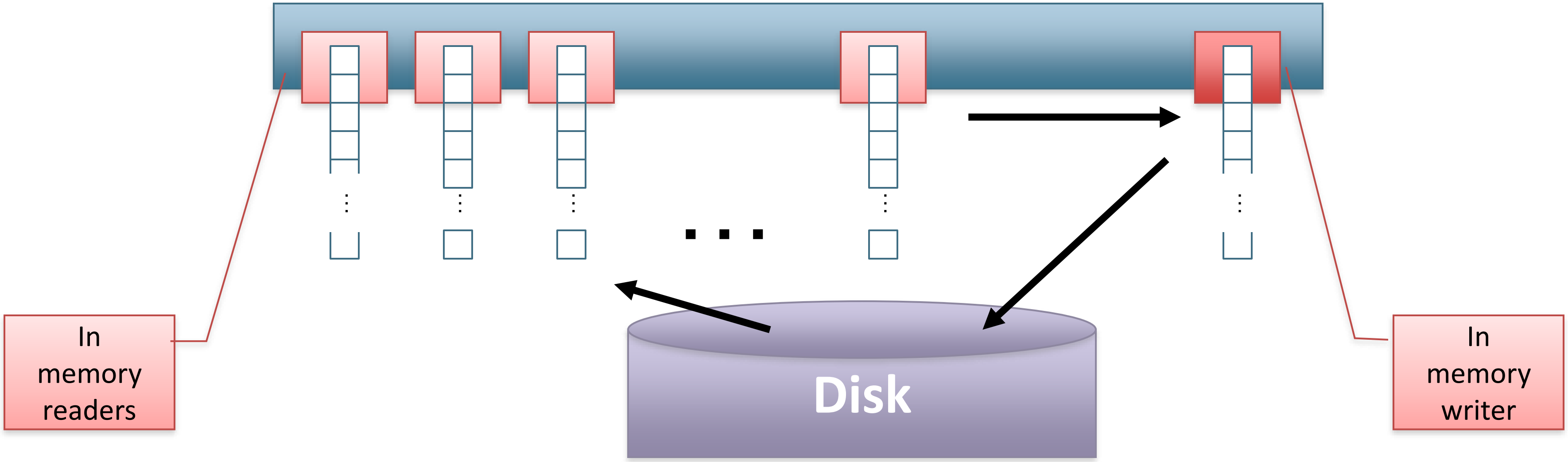
- 2-way Merge: Merge tree of $\log_2 10$ ~= 4 layers.



How to merge the sorted runs?

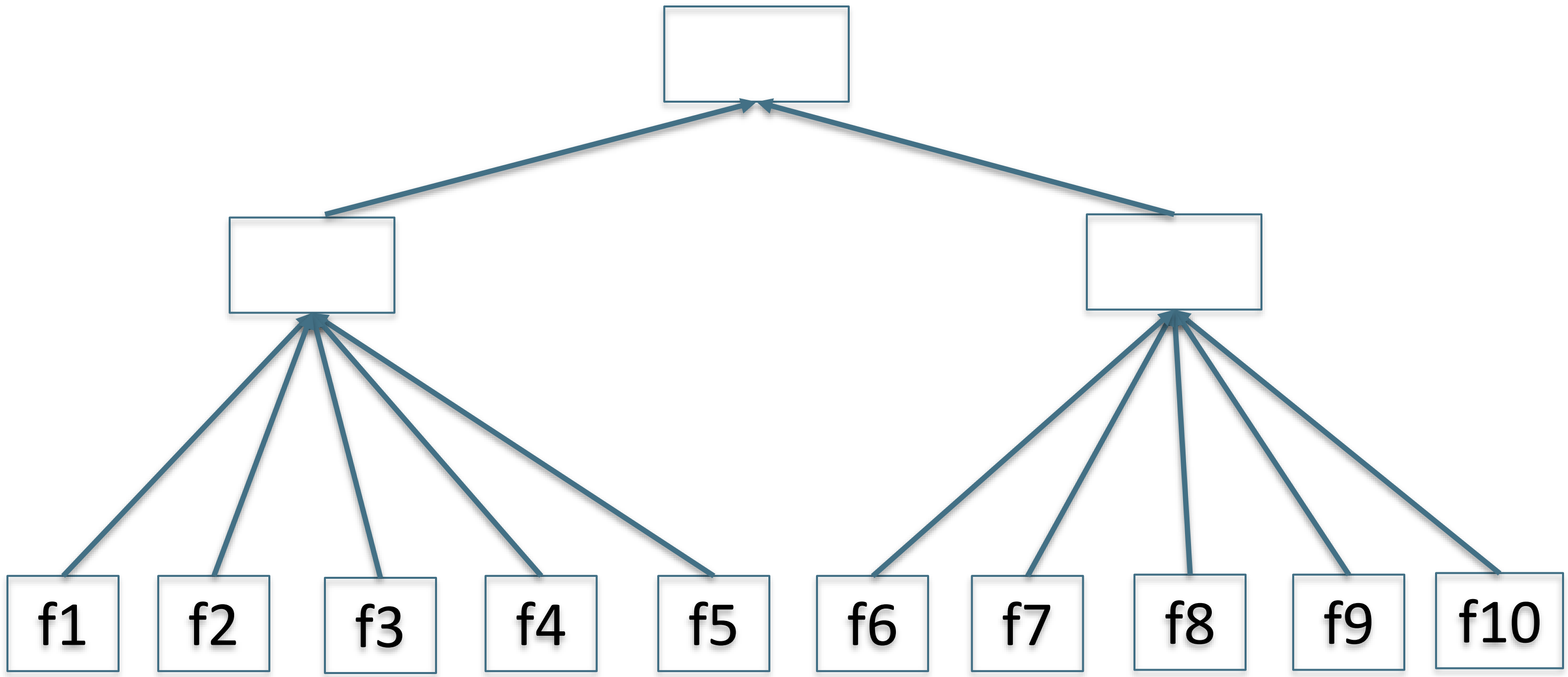
2-way vs N-way merge

- More efficient to do a *n*-way merge by reading from all blocks simultaneously
- Need to read and write in **decent-sized chunks** to fit data into memory yet minimize disk seeks



How to merge the sorted runs?

- 5-way Merge: Merge tree of $\log_5 10$ ~= 2 layers.



Remaining problems with BSBI

- The dictionary must fit into memory
  - Hard to guarantee since it grows dynamically
  - May end up crashing if the dictionary is too big

- A fixed block size must be decided in advance
  - Too small: could be slow since more blocks need to be processed.
  - Too big: may end up crashing if too much memory is used by other applications.

SPIMI: Single-pass in-memory indexing

- **Key idea 1**: Generate an index (i.e., a **real** dictionary + postings lists) as the pairs are processed
- **Key idea 2**: Go as far as memory allows, write out the index and then merge later
- Advantages:
  - No need to keep a single dictionary in memory
  - No need to wait for a fixed-size block to be filled up
  - Able to adapt to the availability of memory

SPIMI: Single-pass in-memory indexing

Hash the pairs into a table and consolidate the postings.

Create a sorted list of terms and write out the table in sorted order.

Merge with others later.

enact	1
julius	1
caesar	1
killed	1
let	2
it	2
be	2
with	2
caesar	2
killed	2
…	…

be	2
with	2
caesar	1,2
it	2
enact	1
julius	1
killed	1,2
let	2

be	2
caesar	1,2
enact	1
it	2
julius	1
killed	1,2
let	2
with	2

SPIMI-Invert

```
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.

SPIMI: Efficiency

- Faster than BSBI
  - No sorting of pairs
  - Only sorting of dictionary terms
- Even faster with compression
  - Compression of terms
  - Compression of postings

More about this in W6.

DISTRIBUTED INDEXING

Distributed indexing

- For web-scale indexing (don’t try this at home!):
  - must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail

How do we exploit such a pool of machines?

Google Data Centers

- Google data centers mainly contain commodity machines, and are distributed worldwide.
- One here in Jurong West (~200K servers back in 2011)
- Must be fault tolerant. Even with 99.9+% uptime, there often will be one or more machines down in a data center.
- As of 2001, they have fit their entire web index in-memory (RAM; of course, spread over many machines)

https://youtu.be/BRH3ST4yK10

http://www.google.com/about/datacenters/inside/streetview/

http://www.straitstimes.com/business/10-things-you-should-know-about-google-data-centre-in-jurong

Architecture of distributed indexing

- Maintain
  - a *master* machine directing the indexing job – considered "safe" (but also "replaceable")
  - a pool of *worker* machines – considered "easily replaceable"
- Break down indexing into (sets of) parallel tasks.
- *Master* machine assigns each task to an idle *worker* machine.

Parallel tasks

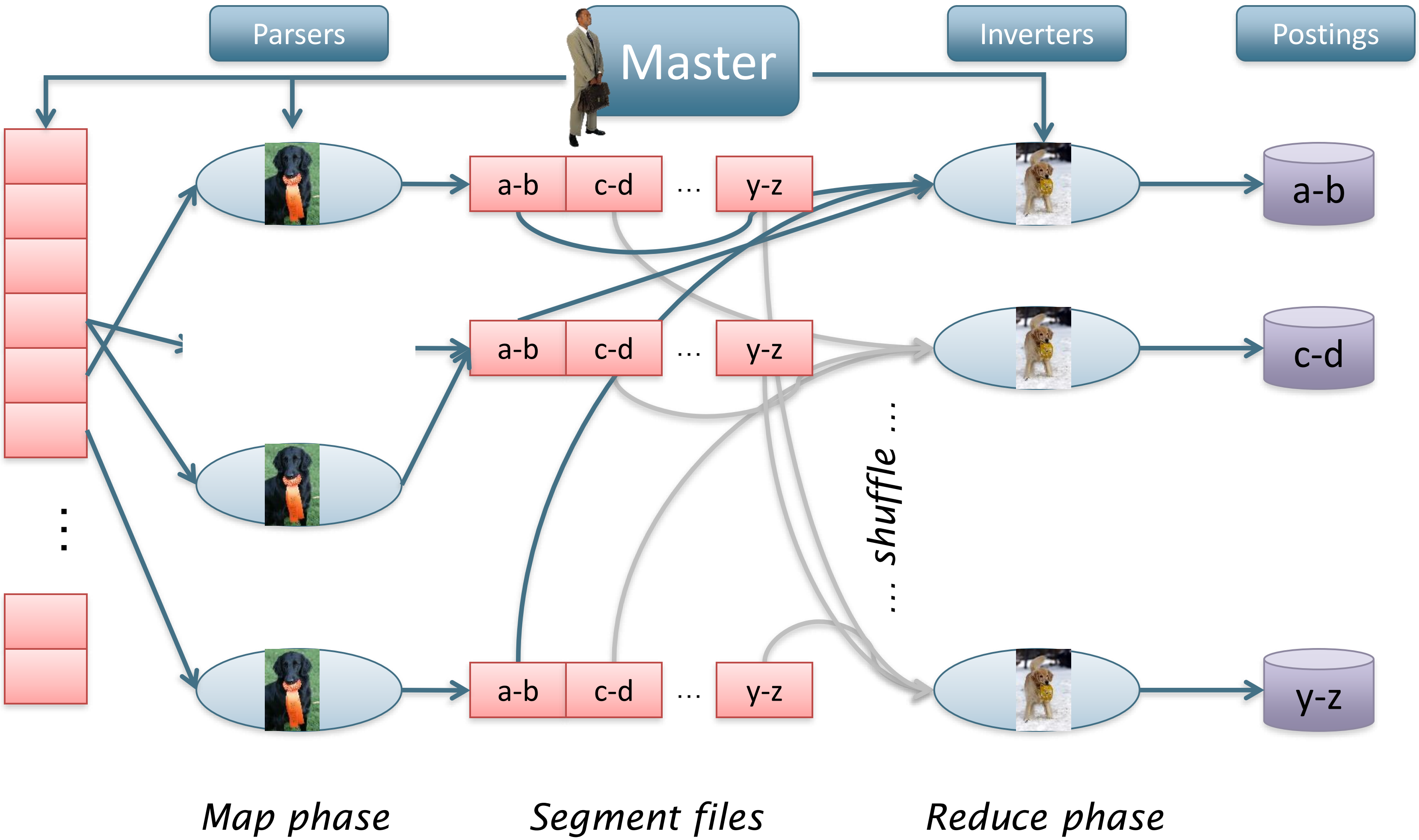
- We will use two sets of parallel tasks
  - Parsing – handled by Parsers 
  - Inversion – handled by Inverters 

- Preprocessing
  - Break the input document collection into subsets of documents called *splits*.

Parallel tasks

- Parsing 
  - The manager assigns a split to a parser.
  - Parser reads the documents from the split and emits (term, doc) pairs.
  - Parser writes pairs into its own *j* partitions based on the first letter of the terms, e.g., ***a-b, c-d, …, y-z*** → *j* = 13.
- Inversion 
  - Manager assign a range to an inverter.
  - An inverter collects all (term, doc) pairs for partitions for the specified range.
  - Inverter sorts and writes the pairs into postings lists.

Data flow


Parsers
Master
Inverters
Postings
a-b
c-d
…
y-z
… shuffle …
Map phase
Segment files
Reduce phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
  - Robust and conceptually simple framework for distributed computing.
  - Can be easily implemented using Apache Hadoop.
  - Widely used in the Google indexing system in the past.

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**: web collection → list(term, docID)
- **reduce**: (<term1, list(docID)>, <term2, list(docID)>, …) →
  (postings list1, postings list2, …)

MapReduce

- **map**
  - d1 : Caesar came, Caesar conquered. d2 : Caesar died →
  - <caesar, d2>, <died,d2>, <caesar, d1>, <came, d1>, <caesar, d1>, <conquered, d1>
- **Reduce**
  - <caesar, (d2, d1, d1)>, <died, (d2)>, <came, (d1)>, <conquered, (d1)> →
  - <caesar, (d1, d2)>, <came, (d1)>, <conquered, (d1)>, <died, (d2)>

DYNAMIC INDEXING

Dynamic indexing

- In practice, collections are rarely static!
  - 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 (yet impractical) approach: re-index every time

2nd simplest approach

- Two indexes
  - One "big" main index (let say **I**)
  - One "small" (in memory) auxiliary index (let say **Z**)
- Mechanism
  - Add: new docs goes to the auxiliary index
  - Delete: maintain a list of deleted docs
  - Update: delete + add
  - Search: search both, merge results and omit deleted docs
- Need to perform **linear merge** when auxiliary index is too large.

Linear Merge

- Let say…
  - The capacity of the auxiliary index **Z** is **n** pairs of (term, docID)
  - The main index **I** can be arbitrarily large
  - Initially both are empty
- The algorithm
  - Once Z is full, write out **Z** and merge with **I**

Linear Merge

- Example:
  - The 1st set of **n** pairs, write out **Z** (**n** items) and merge with **I** (**0** items) → merge **n + 0 = n** items into **I**
  - The 2nd set of **n** pairs, write out **Z** (**n** items) and merge with **I** (**n** items) → merge **n + n** = **2*n** items into **I**
  - The 3rd set of **n** pairs, write out **Z** (**n** items) and merge with **I** (**2*n** items) → merge **n + 2*n = 3*n** items into **I**
  - The 4th set of **n** pairs, write out **Z** (**n** items) and merge with **I** (**3*n** items) → merge **n + 3*n = 4*n** items into **I**
  - …

Linear Merge

- Let say there are a total **T** pairs for which require **k** merges (i.e., k = T / n)

- Cost of merging
  - $n + 2 * n + 3 * n + 4 * n \ldots + k * n$
    $= (k * (k+1) / 2) * n$
    $\sim= nk^2$
    $\sim= \mathbf{O(T^2)}$

Logarithmic merge

- Idea: maintain a series of indexes
  - Z_0: In memory, with the same capacity as I_0 (= n)
  - I_0, I_1, …: on disk, each twice as large as the previous one.

Loop for log levels

  - If Z_0 gets too big (= n), write to disk as I_0, or merge with I_0 (if I_0 already exists) as Z_1
  - Either write Z_1 to disk as I_1 (if no I_1), or merge with I_1 to form Z_2

  … etc.

Logarithmic merge

- Example:
  - The 1st set of **n** pairs, write out Z_0 (**n** items) as $\mathbf{I_0}$
  - The 2nd set of **n** pairs, write out Z_0 (**n** items) but $\mathbf{I_0}$ already exists → merge **n + n** = **2*n** items into $\mathbf{I_1}$ (and remove $\mathbf{I_0}$)
  - The 3rd set of **n** pairs, write out Z_0 (**n** items) as $\mathbf{I_0}$
  - …

	I_0	I_1	I_2
0	0	0	0
n	1	0	0
2*n	0	1	0
3*n	1	1	0
4*n	0	0	1

The presence (1) or absence (0) of the indexes on disk

Logarithmic merge

- Example:
  - …
  - The 4th set of **n** pairs, write out $\mathbf{Z}_0$ (**n** items) but $\mathbf{I_0}$ already exists → merge **n + n** = **2*n** items into a new index $\mathbf{I_1}$ but $\mathbf{I_1}$ already exists → merge **2*n + 2*n = 4*n** items into a new index $\mathbf{I_2}$ (and remove $\mathbf{I_0}$ and $\mathbf{I_1}$)

	I_0	I_1	I_2
0	0	0	0
n	1	0	0
2*n	0	1	0
3*n	1	1	0
4*n	0	0	1

The presence (1) or absence (0) of the indexes on disk

LMergeAddToken(*indexes*, Z_0, *token*)

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

LogarithmicMerge()

$Z_0 \leftarrow \emptyset$ (Z_0 *is the in-memory index.*)
$indexes \leftarrow \emptyset$
while true
do LMergeAddToken(*indexes*, Z_0, getNextToken())

Logarithmic merge

- Cost of merging
  - Each posting is touched *O(log* T) times, so complexity is *O(T log T)*
  - E.g., let n = 4, T = 32, the first pair is touched 4 times (as compared to 8 times in linear merge)
- So logarithmic merge is much more efficient for indexing
- But query processing now is slower
  - Merging results from *O(log T)* indexes (as compared to 2)

Summary

- Indexing
  - Both basic as well as important variants
    - BSBI – sort key values to merge, needs dictionary
    - SPIMI – build mini indexes and merge them, no dictionary
  - Distributed
    - Described MapReduce architecture – a good illustration of distributed computing
  - Dynamic
    - Tradeoff between querying and indexing complexity

Resources for today's lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)