Introduction to

Information Retrieval

Resource Person: Asma Naseer

Lecture 4: Index Construction

Index construction

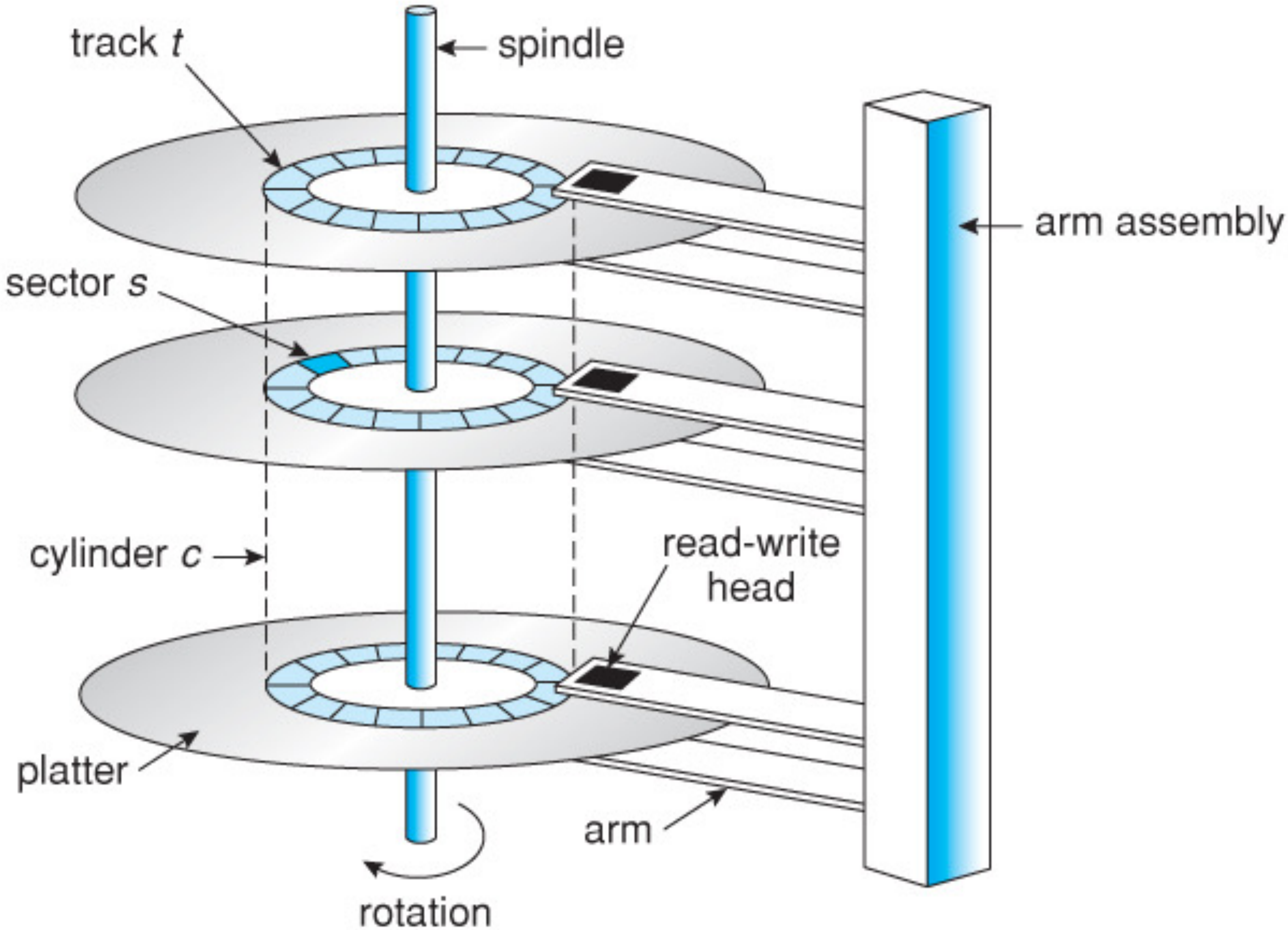
- How do we construct an index?
- What strategies can we use with limited main memory?

Hardware basics

- Many design decisions in information retrieval are based on the characteristics of hardware
- We begin by reviewing hardware basics

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.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.


track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

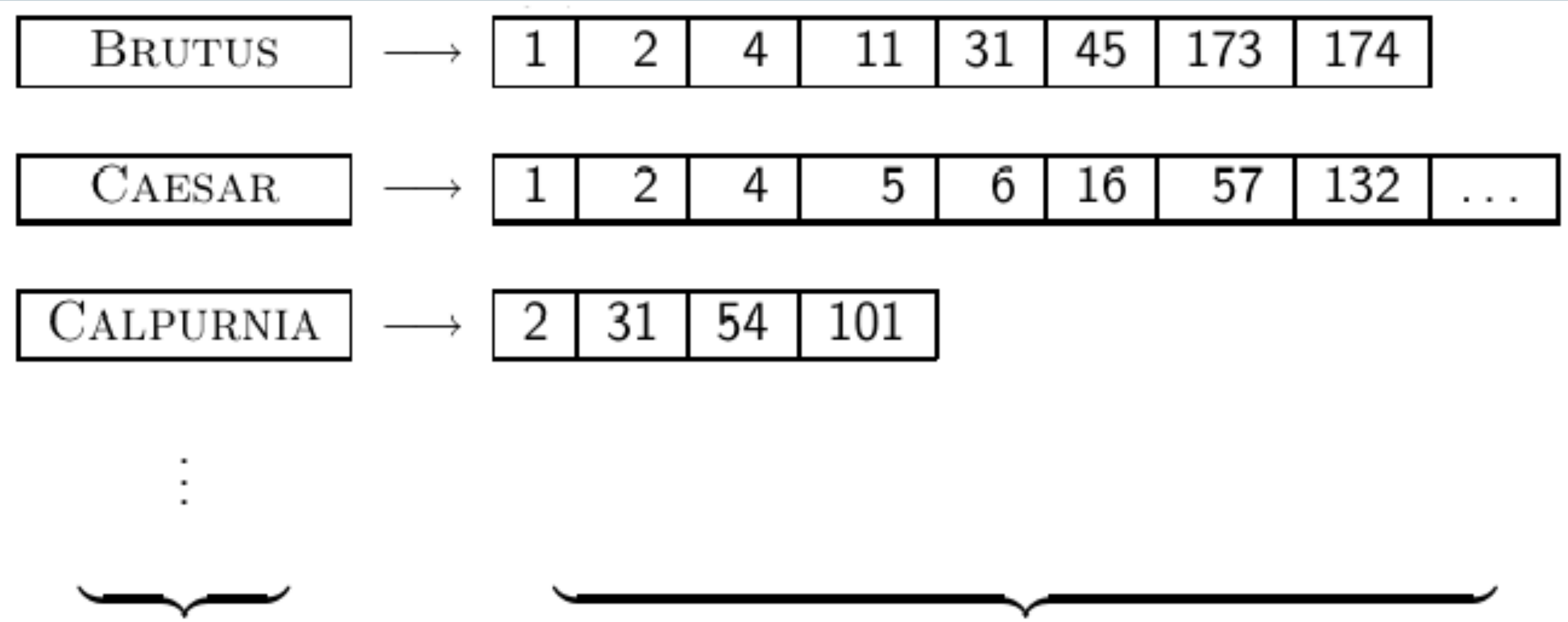
Hardware basics

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It’s much cheaper to use many regular machines rather than one fault tolerant machine.

Some stats

symbol	statistic	value
s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	0.02 µs = 2×10^{-8} s
P	processor's clock rate	10^9 s^{-1}
	lowlevel operation (e.g., compare & swap a word)	0.01 µs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

Goal: Construct Inverted Index



dictionary postings

Key step

- After all documents have been parsed, the inverted file is sorted by terms.

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

Term	Doc #
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	Doc #
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

- In-memory index construction does not scale
  - Can't stuff entire collection into memory, sort, then write back
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.

Block Sort-based Index Construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- $T = 100{,}000{,}000$ in the case of RCV1: we can do this in memory on a typical machine in 2010.
- But in-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.

Sort using disk as "memory"?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?

- No: Sorting T = 100,000,000 records on disk is too slow – too many disk seeks.

Bottleneck

- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort T=100M records

○ *Considering disk seek time = 5 ms*

If every comparison took 2 disk seeks, and N items could be sorted with $N \log_2 N$ comparisons, how long would this take?

Bottleneck

- Q: If every comparison took 2 disk seeks, and *N* items could be sorted with $N \log_2 N$ comparisons, how long would this take?

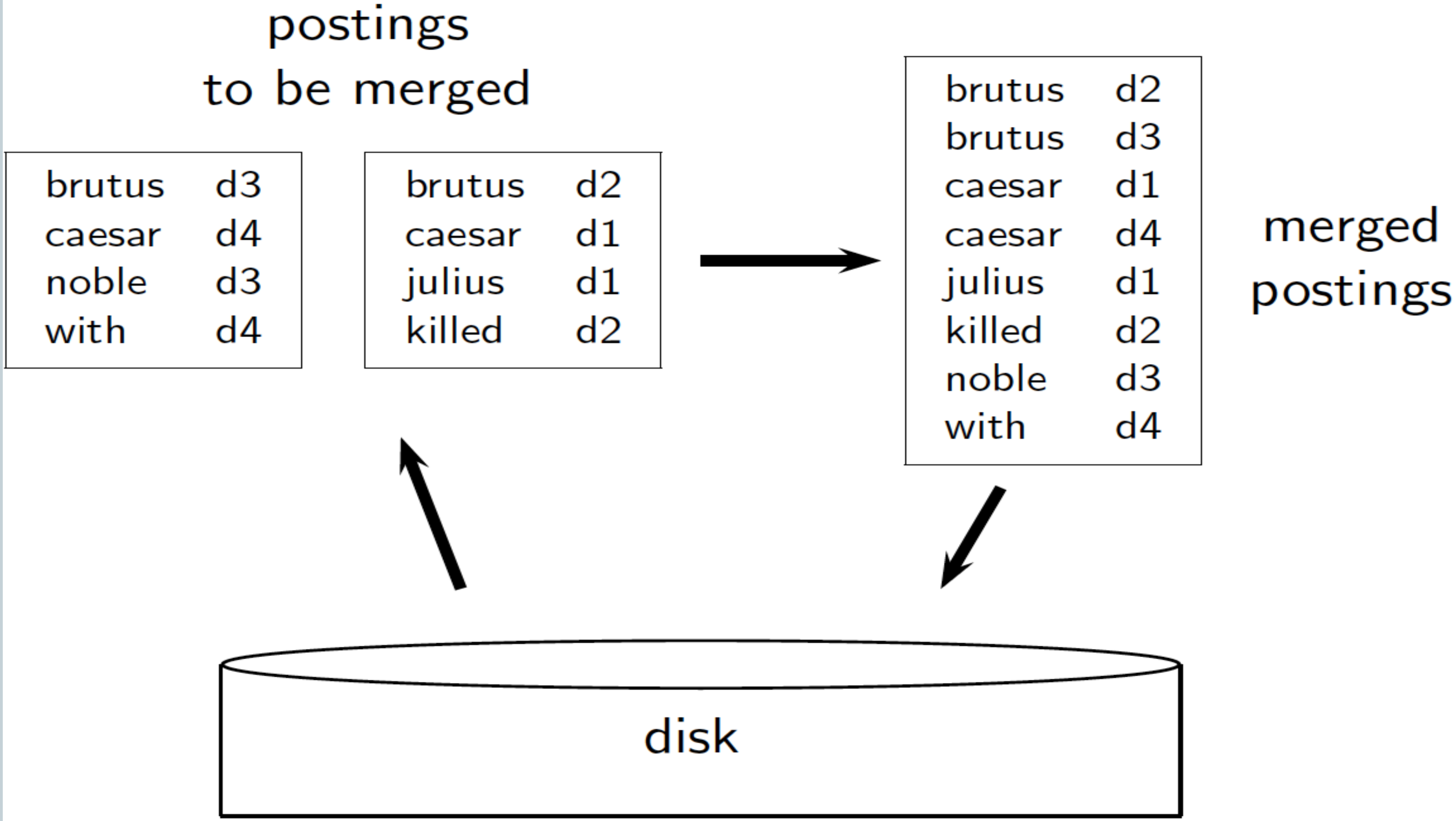
  - *Ans: disk seek time = 5 ms*
  - *N = 100 million*
  - $N \log_2 N$ = *100 million* * $\log_2$*100 million = 2700 million*
  - *Total time = 2700 million * 5 ms * 2 = 27000 million ms*
  - *= 3125 Days = 8.5 years*

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

8

- ~~12~~-byte (4+4+~~4~~) records *(term, doc, freq)*.
- These are generated as we parse docs.
- Must now sort 100M such 12-byte records by *term*.
- Define a <u>Block</u> ~ 10M such records
  - Can easily fit a couple into memory.
  - Will have 10 such blocks to start with.
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.

Postings Merge



Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes $2N \ln N$ expected steps
  - In our case 2 x (10M ln 10M) steps
- *Exercise: estimate total time to read each block from disk and Quicksort it.*
- 10 times this estimate – gives us 10 sorted *runs* of 10M records each.

Sorting 10 blocks of 10M records

- *Exercise: estimate total time to read each block from disk and Quicksort it.*

  - *Solution: 1 block size = N = 10 million*
  - *Processor time for one compare and swap operation* = 10^{-8} s
  - 2*N ln N comparisons = 2* 10 million * 16 = 320 million
  - Total time = 320 million * 10^{-8} s = 320 * 10^{6} * 10^{-8} s
  - = 3.2 seconds

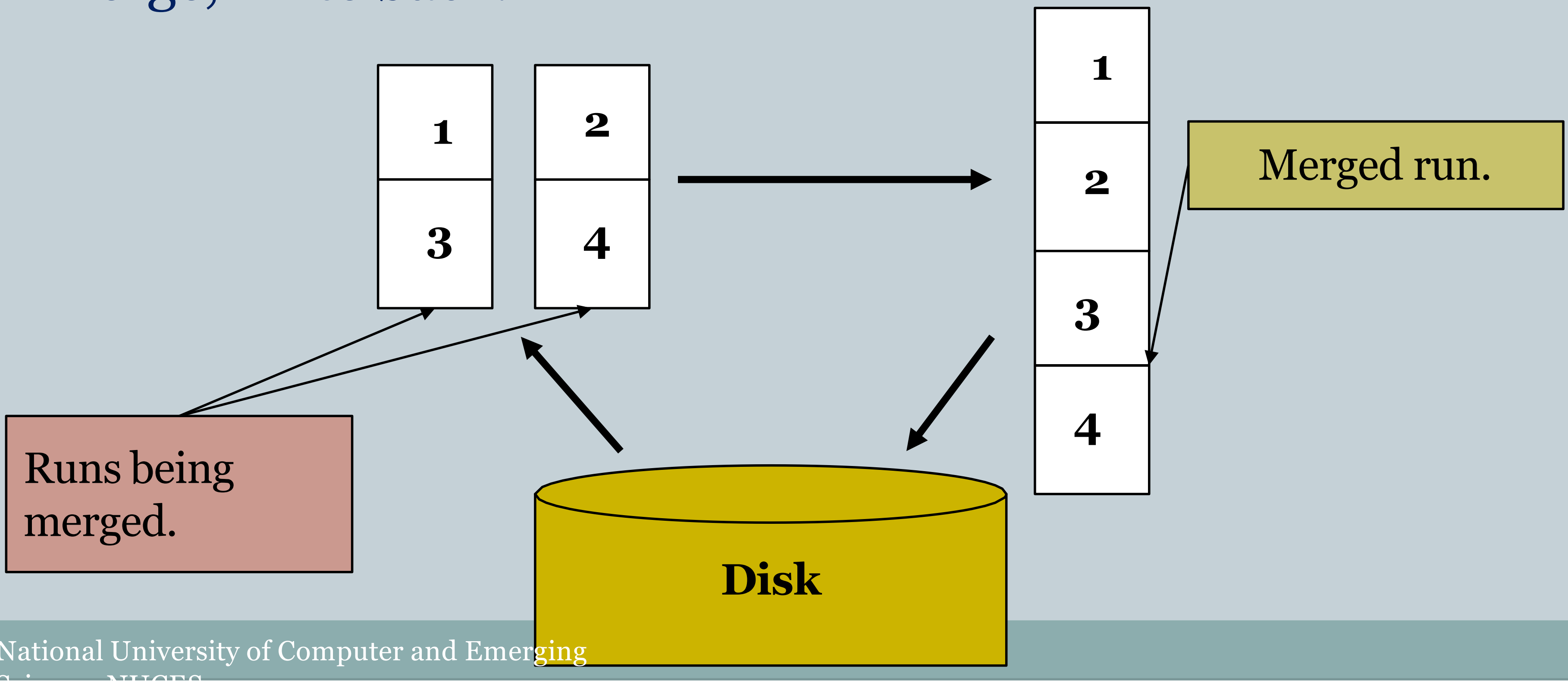
- *Time to sort 10 blocks = 3.2 * 10 = 32 seconds*

BSBIndex Construction


```
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)
```

How to merge the sorted runs?

- Can do binary merges, with a merge tree of $\log_2 10 = 4$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.



How to merge the sorted runs?

- But it is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously

- Providing you **read decent-sized chunks** of each block into memory and then write out **a decent-sized output** chunk, then you're **not killed by disk seeks**

Remaining problem with sort-based algorithm

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

SPIMI: Single-pass in-memory indexing

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

Remaining problem with sort-based algorithm

Some Important Points:

1. Blocked sort -based indexing has excellent scaling properties, but it needs data structure for mapping terms to term-ID. For very large collections, this data structure does not fit into memory.
2. A SPIMI uses term instead of termID's, writes each block's dictionary to disk and then starts a new dictionary for next block.
3. A difference between BSBI and SPIMI is that, SPIMI adds a posting directly to its posting list. Instead of first collecting all Term ID- docID pairs and then sorting them

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: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
- Compression discussed in future lectures

Collection statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
V	unique terms	400,000
	non-positional postings	100,000,000

Distributed indexing

- For web-scale indexing (don’t try this at home!): must use a distributed computer cluster
- Individual machines are fault-prone.
  - Can unpredictably slow down or fail.
- How do we exploit such a pool of machines?

Google data centers (2007 estimates; Gartner)

- Google data centers mainly contain commodity machines.
- Data centers are distributed all over the world.
- 1 million servers, 3 million processors/cores
- Google installs 100,000 servers each quarter.
- Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world!

- 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)? What about 1M nodes?
- Answer: ~~63~~% **It’s 100-63**
- Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?
- Answer: less than two minutes

Distributed indexing

- Maintain a master machine directing the indexing job – considered "safe"
- Break up indexing into sets of parallel tasks
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers
  - Inverters
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Each split is a subset of documents.

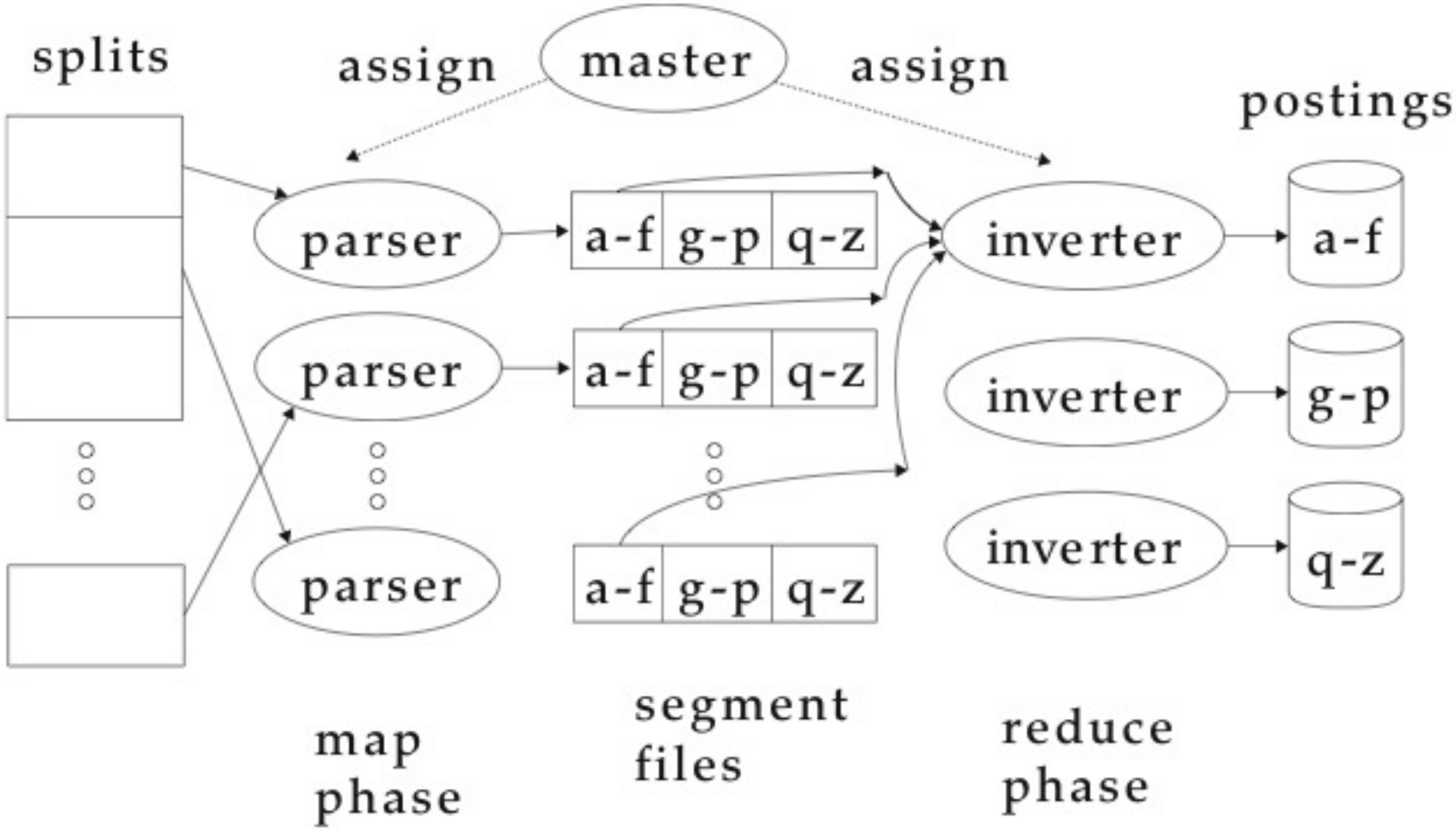
Parsers

- 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: $j = 3$)

Inverters

- An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
- Sorts and writes to postings lists

Data flow


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

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

Dynamic indexing: Simplest approach

- Maintain big main index on disk
- New docs go into small auxiliary index in memory.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

Issue with auxiliary and main index

- Frequent merges
- Poor search performance during index merge
- Actually:
  - Merging of the auxiliary index into the main index is not that costly 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.
- 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 into several files, collect small postings lists in one file etc.)

Logarithmic merge

- Logarithmic merging amortizes the cost of merging indexes over time.
  - → Users see smaller effect on response times.
- 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 I_0 (if I_0 already exists) and write merger to I_1 etc.

```
LMergeAddToken(indexes, Z0, token)
Z0 ← Merge(Z0, {token})
if |Z0| = n
   then for i ← 0 to ∞
           do if Ii ∈ indexes
                 then Zi+1 ← Merge(Ii, Zi)
                        (Zi+1 is a temporary index on disk.)
                      indexes ← indexes − {Ii}
                 else Ii ← Zi     (Zi becomes the permanent index Ii.)
                      indexes ← indexes ∪ {Ii}
                      Break
           Z0 ← ∅

LogarithmicMerge()
Z0 ← ∅     (Z0 is the in-memory index.)
indexes ← ∅
while true
do LMergeAddToken(indexes, Z0, getNextToken())
```

Binary numbers: $I_3I_2I_1I_0 = 2^3 2^2 2^1 2^0$

- 0001
- 0010
- 0011
- 0100
- 0101
- 0110
- 0111
- 1000
- 1001
- 1010
- 1011
- 1100

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number of postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction is $O(T \log T)$.
- . . . because each of T postings is merged $O(\log T)$ times.
- Auxiliary index: index construction time is $O(T^2)$ as each posting is touched in each merge.
  - Suppose auxiliary index has size a
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$
- So logarithmic merging is an order of magnitude more efficient.

Dynamic indexing at large search engines

- Often a combination
  - Frequent incremental changes
  - Rotation of large parts of the index that can then be swapped in
  - Occasional complete rebuild (becomes harder with increasing size – not clear if Google can do a complete rebuild)

Building positional indexes

- Basically the same problem except that the intermediate data structures are large.

Resources

- Chapter 4 of IIR
- Resources at http://ifnlp.org/ir
  - Original publication on MapReduce by Dean and Ghemawat (2004)
  - Original publication on SPIMI by Heinz and Zobel (2003)
  - YouTube video: Google data centers