NYU | TANDON SCHOOL OF ENGINEERING

Indexing and Inverted Index Construction

CS6913
CSE Department
NYU Tandon School of Engineering

Today's Lecture

- **Index Setup**
  - Indexing and Parsing
  - Index Structures and Layout
- **Disks and I/O-Efficient Sorting**
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- **Index Building**
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Index Setup:

- Goal: a data structure that allows us to efficiently:
  - find all documents containing a term
  - and/or find if a given document contains a term
  - should supports multi-term ranked queries
  - may also store additional data such as (a) the number of term occurrences, (b) their positions, (3) context, (4) impact scores ...
- Most popular structure: Inverted Index
  - Alternatives: Bloom filters, bitmaps, signature files
  - Or for positions: n-grams indexes, wavelets, nextword indexes
  - But all current search engines seem to use inverted indexes
  - Sometimes, Bloom filters or bitmaps in addition (e.g., Bing bitfunnel)
  - Future/emerging: nearest-neighbor search in high dimensions?

We Focus on Inverted Indexes

- inverted index consists of inverted lists
- one inverted list per term
- each inverted list consists of index postings
- each index posting usually of form (docID, frequency)
- but there are other options

aalborg	3452, 11437, ...
.	
.	
.	
.	
.	
arm	4, 19, 29, 98, 143, ...
armada	145, 457, 789, ...
armadillo	678, 2134, 3970, ...
armani	90, 256, 372, 511, ...
.	
.	
.	
.	
.	
zebra	602, 1189, 3209, ...

inverted index

Index Setup:

- What is a "term" for the purpose of indexing?
  - Any string between two separating symbols
  - Called "full-text indexing" as opposed to keyword indexing
  - How about "New York City"? Or "roller coaster"
  - How about other languages? Say, Chinese?
- Alternatives: indexing by (important) keywords
  - Only index, say, the most important 20 terms in document
  - Or only words in a dictionary of important words
  - Or human-created or data-mined keywords
  - Or visual features in image search
- Also remember: stemming, stopwords, positions

Alternative #1: Keyword Index

- Index only a few important terms
- E.g., terms supplied by a publishers
- Or terms from a dictionary of important terms
  (e.g., lexicon of medical or legal terms)
- Or terms identified as central to a document
  using data mining/machine learning

Alternative #2: Expansion with other Terms

- Add terms not in the document that are relevant, or that might be used by people searching for it
- Based on synonyms
- Based on language models, e.g., transformer models
- Might result in much larger indexes
- Entity extraction and knowledge bases
- E.g., people, companies, geographic references

Alternative #3: Phrases, n-Grams, Term-Pairs

- Index common phrases such as "New York City" or "good morning"
- Meaning, build inverted lists for them
- Or index all n-grams: "the mouse is in the house"
- Term pair (intersection) for (dog, cat):
  - index all documents containing both dog and cat
  - results in a much larger index with more inverted lists
  - but each one much shorter than the single-term lists
- Or only those documents where dog and cat appear within distance five in document

Alternative #3: Phrases, n-Grams, Term-Pairs

- Index common phrases such as "New York City" or "good morning"
- Meaning, build inverted lists for them
- Or index all n-grams: "the mouse is in the house"
- Term pair (intersection) for (dog, cat):
  - index all documents containing both dog and cat
  - results in a much larger index with more inverted lists
  - but each one much shorter than the single-term lists
- Or only those documents where dog and cat appear within distance five in document

Alternative #3: Phrases, n-Grams, Term-Pairs

- Index common phrases such as “New York City” or “good morning”
- Meaning, build inverted lists for them

3-gram

- Or index all n-grams: “the mouse is in the house”
- Term pair (intersection) for (dog, cat):
  - index all documents containing both dog and cat
  - results in a much larger index with more inverted lists
  - but each one much shorter than the single-term lists
- Or only those documents where dog and cat appear within distance five in document

Alternative #3: Phrases, n-Grams, Term-Pairs

- Index common phrases such as "New York City" or "good morning"
- Meaning, build inverted lists for them
- Or index all n-grams: "the mouse is in the house"
- Term pair (intersection) for (dog, cat):
  - index all documents containing both dog and cat
  - results in a much larger index with more inverted lists
  - but each one much shorter than the single-term lists
- Or only those documents where dog and cat appear within distance five in document

Alternative #3: Phrases, n-Grams, Term-Pairs

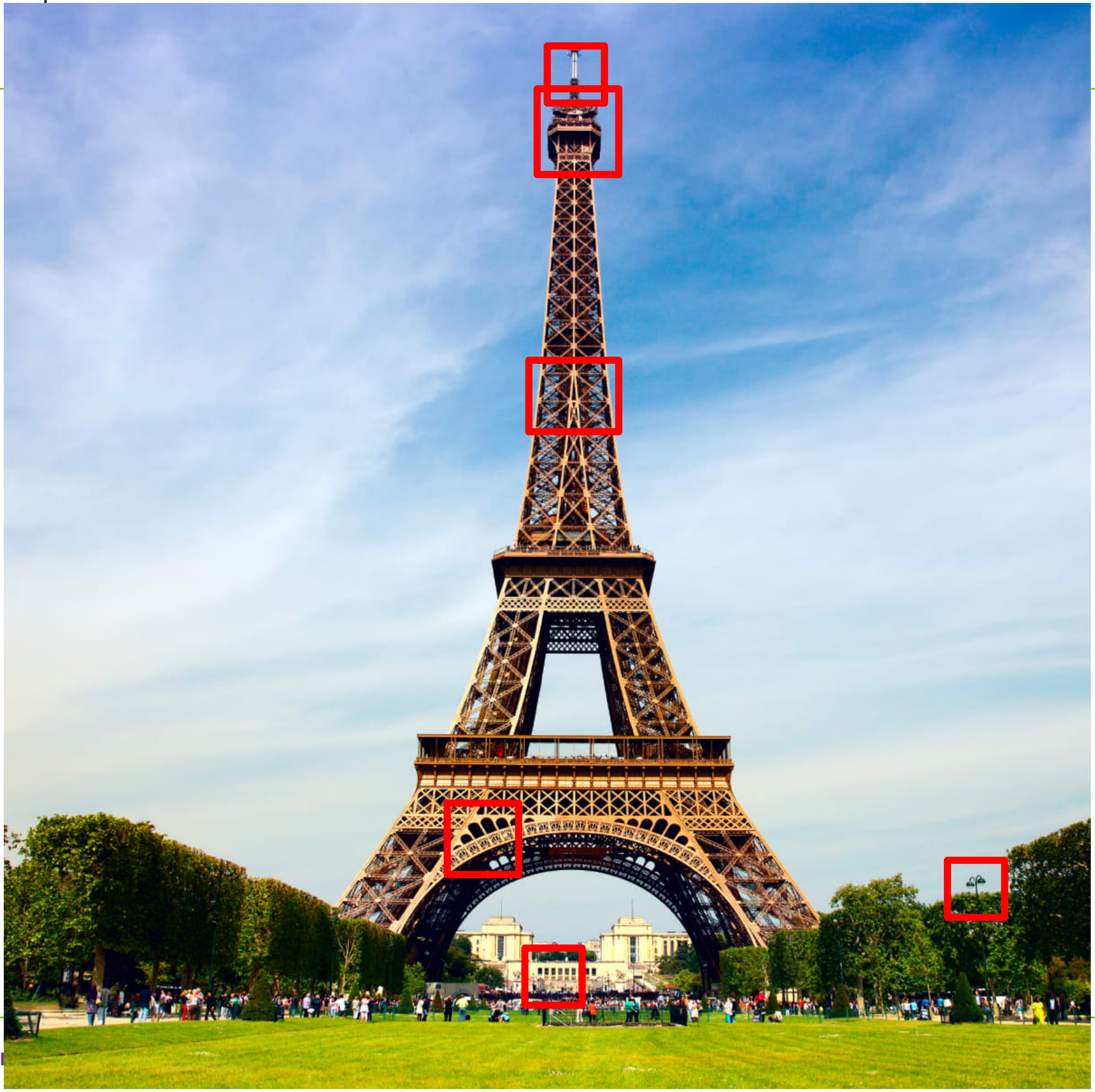
- Index common phrases such as "New York City" or "good morning"
- Meaning, build inverted lists for them
- Or index all n-grams: "the mouse is in the house"
- Term pair (intersection) for (dog, cat):
  - index all documents containing both dog and cat
  - results in a much larger index with more inverted lists
  - but each one much shorter than the single-term lists
- Or only those documents where dog and cat appear within distance five in document

Alternative #4: Visual Features in Image Search

- Object recognition in augmented reality (AR) systems
- Identify object in picture pointed at by phone camera
- E.g., tree, person, famous landmark, product
- Solution: find matching object in a prelabeled database of images ("instance retrieval")
- Common approaches identify and encode characteristic parts of the image, and index them as "visual words"
- Encoding using SIFT or more recently CNNs
- Must be robust against shifts, rotations scaling
- An inverted index is used to index the visual words

  BoVW (bag-of-visual-words) approach
- Search index to find candidates to more closely analyze

NYU TANDON SCHOOL OF ENGINEERING



Context and Fielded Indexes:

- We may want to store in what context word occurs
  - In the title of a document, or in a heading, or in bold
  - In the URL of the document
  - In the anchor text of a hyperlink pointing to the page
  - Or it does not occur, but was data-mined as a suitable keyword
- How to store this information?
  - Attach a context to each posting: format (docID, freq, context)
  - Encode in position field: say, 1-20 means URL, 21-80 anchortext
  - Or fielded index: subdocuments for different fields
  - E.g., each document has 3 subdocuments: url, title, body
- Or: enclude context in pre-computed impact score

Impact Scores:

- Suppose we have a “simple” ranking function
  - E.g., cosine, BM25, or simple language-modeling based (LM)
  - score(q, d) = SUM $_{t \text{ in } q}$ s(t, d)
  - This means we can precompute s(t, d) at indexing time!

Impact Scores:

- **Suppose we have a "simple" ranking function**
  - **E.g., cosine, BM25, or simple language-modeling based (LM)**
  - **score(q, d) = SUM $_{t \text{ in } q}$ s(t, d)**
  - **This means we can precompute s(t, d) at indexing time!**

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

$$K = k_1 \times ((1-b) + b \times \frac{|d|}{|d|_{avg}})$$

Impact Scores:

- **Suppose we have a "simple" ranking function**
  - **E.g., cosine, BM25, or simple language-modeling based (LM)**
  - **score(q, d) = SUM $_{\text{t in q}}$ s(t, d)**
  - **This means we can precompute s(t, d) at indexing time!**

$$BM25(q,d) = \sum_{t \in q} \boxed{\log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}}$$

$$K = k_1 \times \left((1-b) + b \times \frac{|d|}{|d|_{avg}}\right)$$

  - **Part in red box can be precomputed and stored in posting**

Impact Scores:

- Suppose we have a "simple" ranking function
  - E.g., cosine, BM25, or simple language-modeling based (LM)
  - score(q, d) = SUM $_{t \text{ in } q}$ s(t, d)
  - This means we can precompute s(t, d) at indexing time!
  - And then store f(t, d) in quantized form in the index posting
  - Quantized: rounded to one of, say, 256 values (one byte)
  - During query processing, only add scores for query terms
- Index posting formats for full-text:
  - (docID, frequency)
  - (docID, impact score)
  - (docID, freq, pos_1, … , pos_{freq})
  - But note: positions are usually stored in a separate place

Impact Scores:

- Suppose we have a "simple" ranking function
  - E.g., cosine, BM25, or simple language-modeling based (LM)
  - score(q, d) = SUM $_{t \text{ in } q}$ s(t, d)
  - This means we can precompute s(t, d) at indexing time!
  - And then store f(t, d) in quantized form in the index posting
  - Quantized: rounded to one of, say, 256 values (one byte)
  - During query processing, only add scores for query terms
- Index posting formats for full-text:
  - (docID, frequency)
  - (docID, impact score) widely used in production systems!
  - (docID, freq, pos_1, … , pos_{freq})
  - But note: positions are usually stored in a separate place

Summary

- Inverted indexes are very versatile and can be used in many different scenarios
- In the following we focus on full-text indexing
- That is, we index all the words in a document
- But this is not always how they are used

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Index Structures and Layout:

- We actually need to build three structures:
  - The actual inverted index, with one inverted list per term
  - A lexicon structure, storing data about each distinct term:
    - where the inverted list for it starts. How many docs contain it, ...
  - A page table, containing for each page/document:
    - its URL, its length in bytes or words, its pagerank, ...
  - And a storage system to fetch pages by URL or by docID
- How to store these structures?
  - At end of indexing, each structure should be a file(s) on disk
  - Lexicon and page table smaller than inverted index
  - At least part of lexicon and page table in memory during QP?
  - Note: lexicon stores f_t, and page table stores $|d|$
  - May use a tuple store for the pages themselves

Lexicon Structure:

- Contains one element for each distinct term
- Dictionary, hash table, concise DS, or disk-based DS
- Lookup based on term (the term is key)
- Stores start of corresponding inverted list in index
- Say, a file offset for a disk-based index, or a pointer
- Also stores length of list (f_t value), maybe other items
- Can get large in some cases

Page or Document Table:

- Contains one element for each indexed document
- Keeps the mapping between documents and docIDs
- docID: unique integer identifying a document
- Given docID, we need to be able to look up URL
- Also, must be able to retrieve document from store
- Maybe store document size and pagerank

Page or Document Table:

- Simplest approach: records ordered by docID

1	www.x.com/index.html	156	0.00125
2	www.yug.org/pages/h.html	256	0.0145
3	….		

- More space-efficient:
  - Store URLs in alphabetic order, maybe compressed
  - Replace URL in above table by pointer or offset
  - Also allows lookup of docID by URL

Inverted Index Layout:

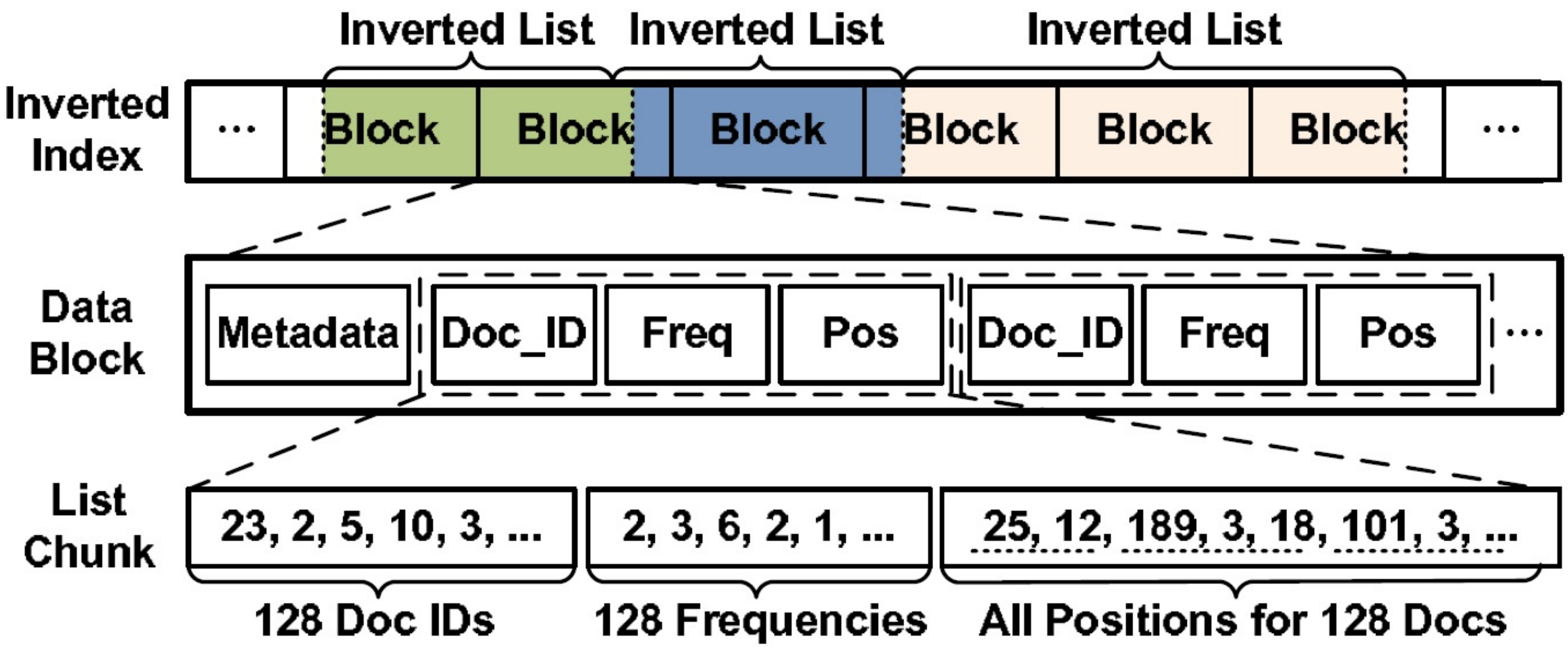
- How do we actually store the inverted index?
- On disk, and in memory
- Usually in blocks of certain number of postings
- In compressed form, even if in memory
- Posting format (docID, frequency) or (d, f)
- DO NOT store docIDs and freqs in interleaved form

Inverted Index Layout:

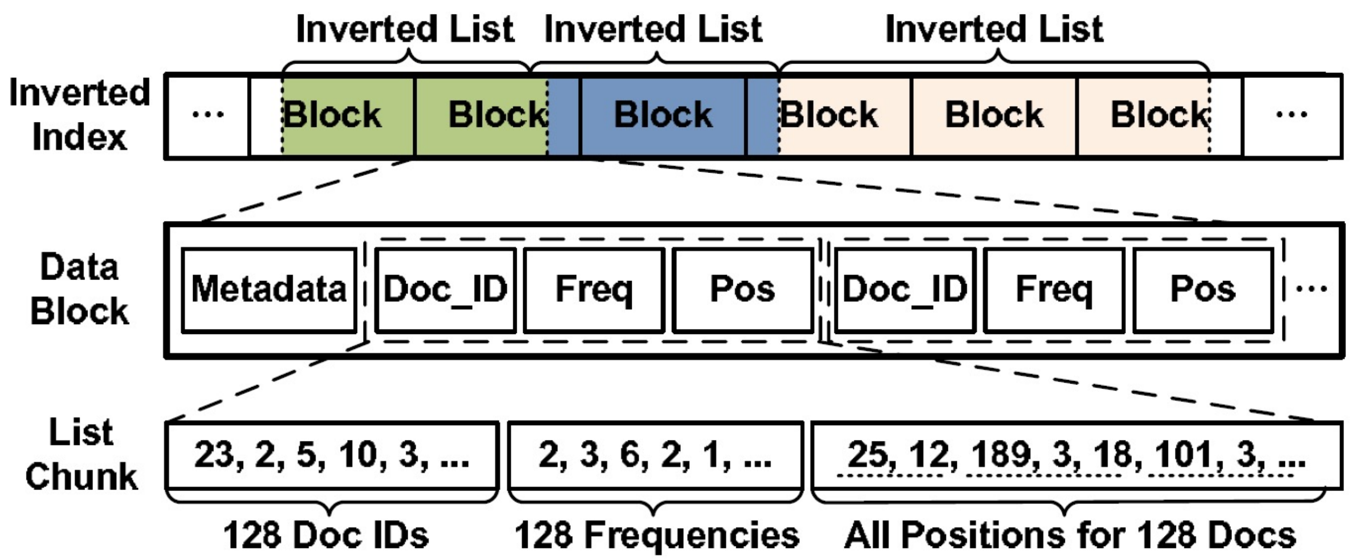
- How do we actually store the inverted index?
- On disk, and in memory
- Usually in blocks of certain number of postings
- In compressed form, even if in memory
- Posting format (docID, frequency) or (d, f)
- DO NOT store docIDs and freqs in interleaved form

- It is d, d, …, d f, f, ..., f
- Not d, f, d, f, d, f, …, d, f

Index Layout Example



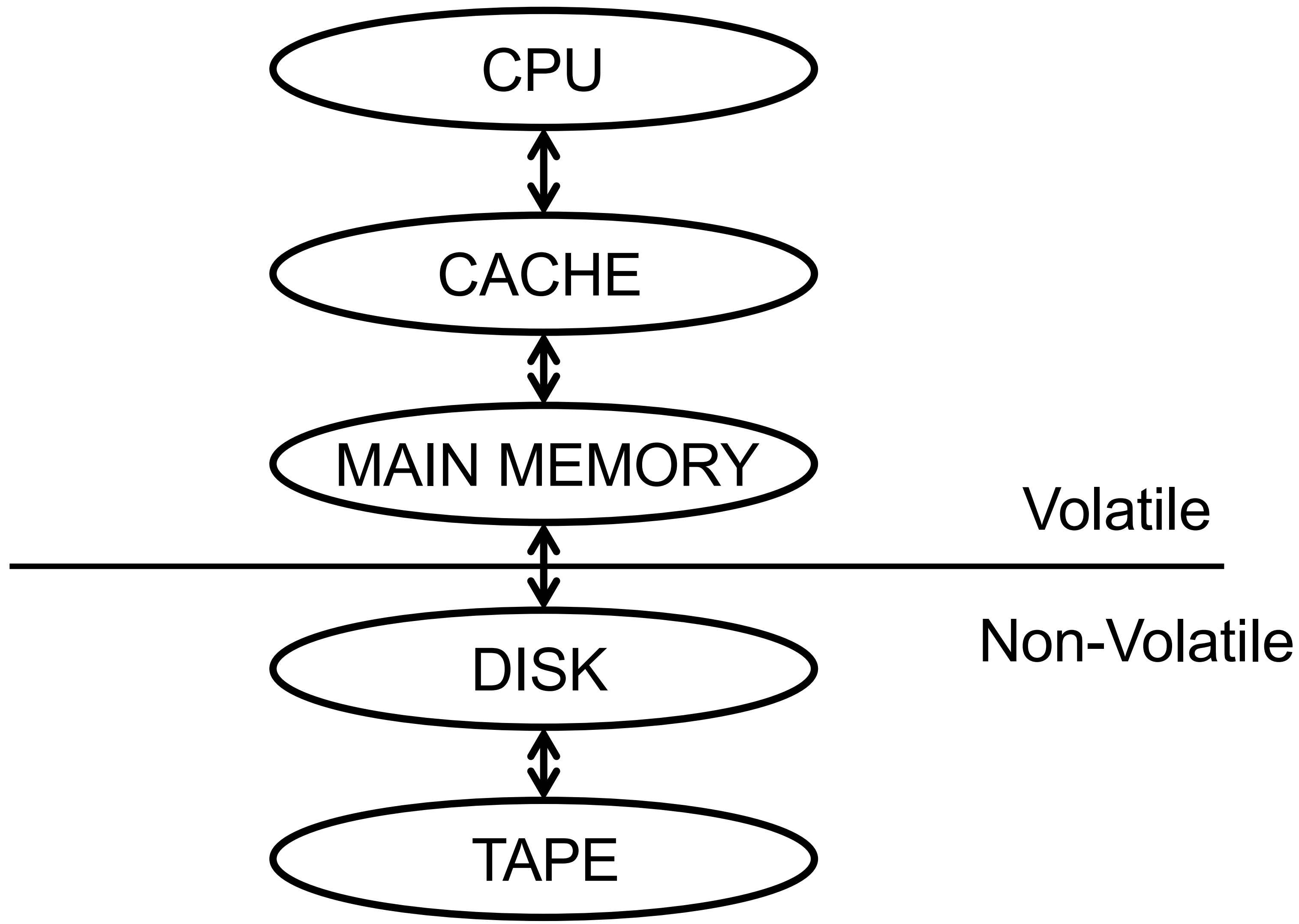
Index Layout Example



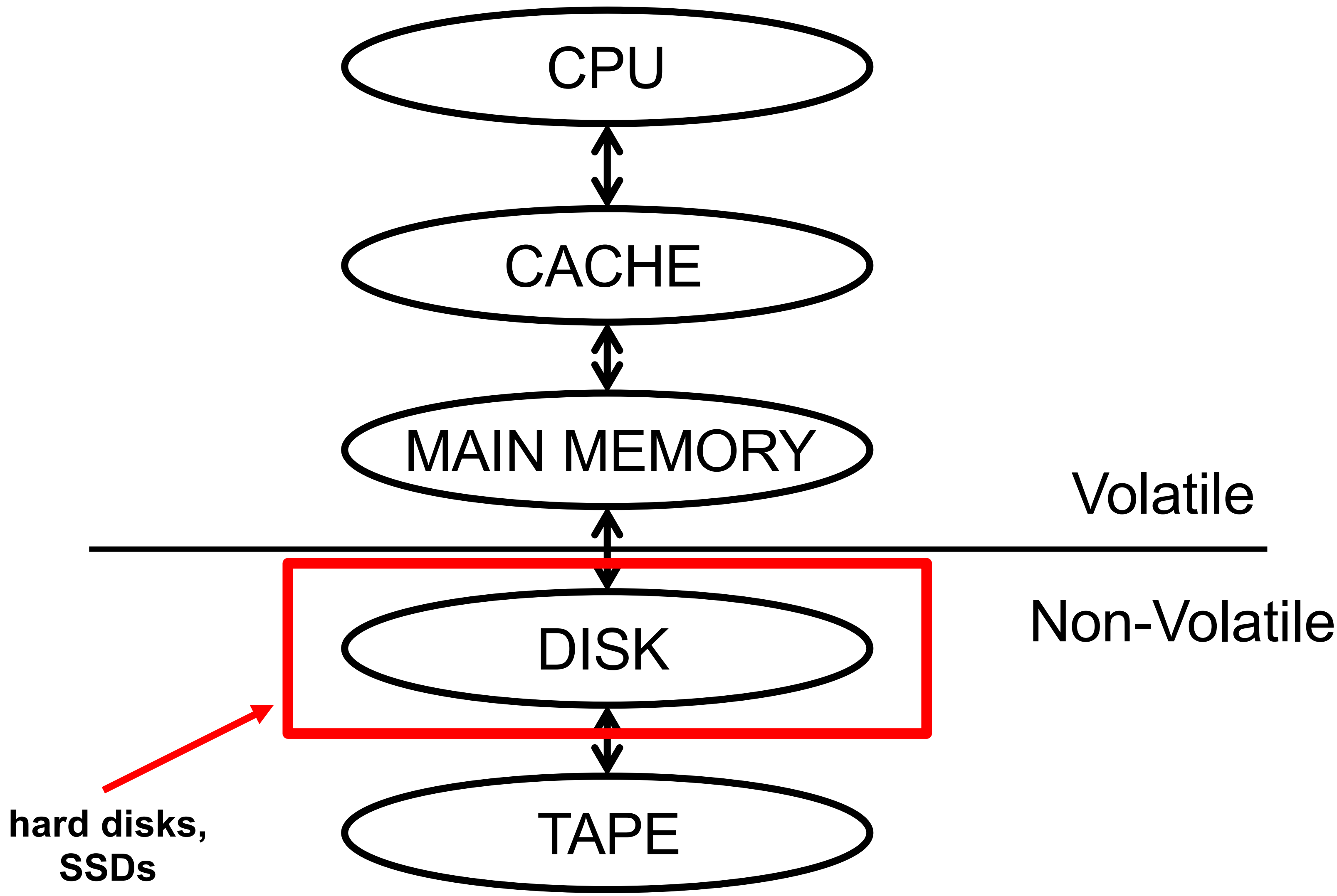
- Inverted index may be partially cached in main memory
- Inverted index file organized into blocks of say 64 KB
- Caching is done on a data block level (or by list, but tricky in dynamic case)
- Lists go across data block boundaries
- Lists are divided into chunks (list blocks) of, say, 128 compressed postings
- Metadata: arrays with last docIDs and sizes of chunks (per block or list)
- Positions often stored in separate structure, or not indexed at all

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

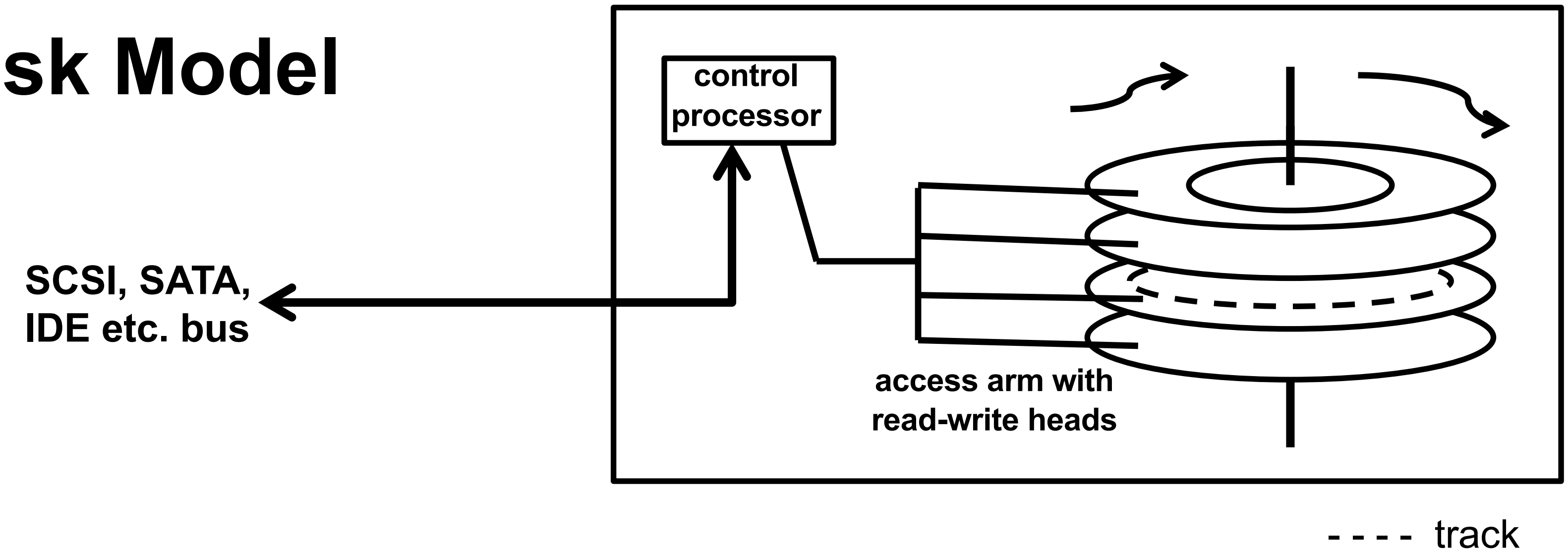


Memory Hierarchy



Memory Hierarchy

Disk Model



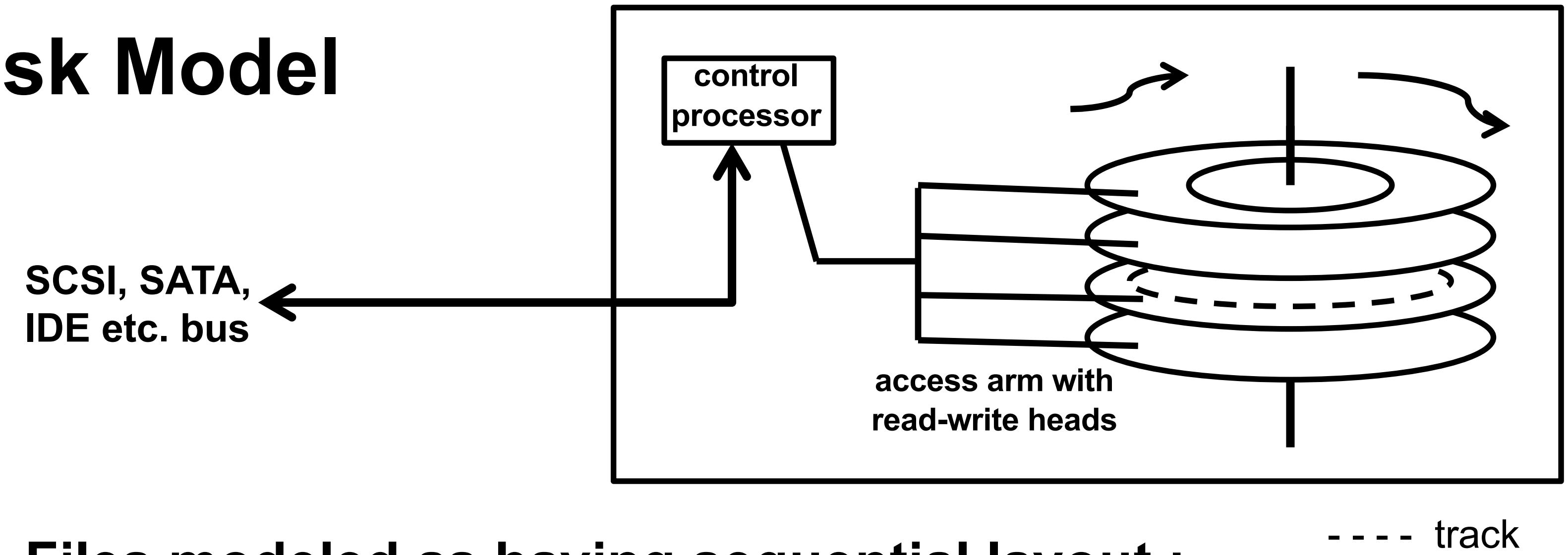
To read data, disk needs to:

- swivel access arm so head is over track holding data
- wait for start of data to rotate under head
- read all the data

Disk access latency: swivel + rotation

Max transfer rate: per rotation, all data that fits on one track

Disk Model



Files modeled as having sequential layout :

COST = (SEEK + ROTATIONAL_LATENCY) + TRANSFER_COST_PER_BYTE

Seek time: related to speed of electric motor in access arm

Rotational latency: depends on rotations per second: 7200 RPM = 120 RPS

Transfer cost depends on how mucb data per track, plus RPS

NYU | TANDON SCHOOL OF ENGINEERING

Disk Model

Basically, we model disk performance by two parameters:

Access time: time to find the start of the data (~5-10ms)

Transfer rate: MB/s of data retrieved afterwards (~50-120MB/s)

SSD: Solid State Drives

- Non-volatile, partially replacing hard disk in servers + laptops
- Still more expensive than hard drives (HDD)
- But getting cheaper: say $100-200 per TB
- Much faster!!
- Transfer rate >500 MB/s, <100 us per random access
- Big impact on large data and I/O-efficient computing
- Used widely in search clusters

SSD: Solid State Drives (ctd)

- However, random access still more expensive than sequential
- Also, SSD blocks only allow limited # of writes before fail
- File system needs to do smart allocation to avoid wearout
- May requires different file system for best performance
- Some vendors combine hard disks and SSDs in one device
- And of course, many servers have both in data centers
- SSDs use less power

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

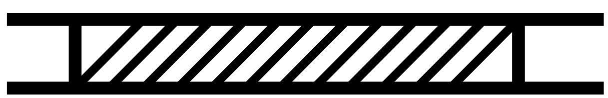
DISK PERFORMANCE MODELING

- Seek Time (5ms)
- Rotational Latency (5ms)

access time 10ms

- Transfer Rate (80MB/s)

- File of Length 400KB



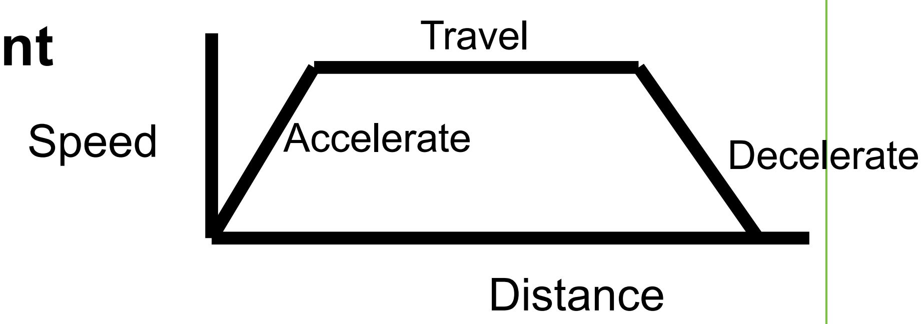
Time to read : t_R = 10ms + 400/80ms = 15ms

- FILE of size 4KB (or 8, 16..)

  t_R = 10ms + 4/80ms = 10.05ms

More Details

Disk arm swivel speed non-constant



Optimized moves to neighboring tracks (1 – 2ms)

Buffering on disk, read–ahead

Bus contention (SCSI, master–slave on IDE, SATA)

Elevator algorithm for disk scheduling

Files may not be fully sequentially (disk fragmentation)

Note: cost of directory lookups for small files

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

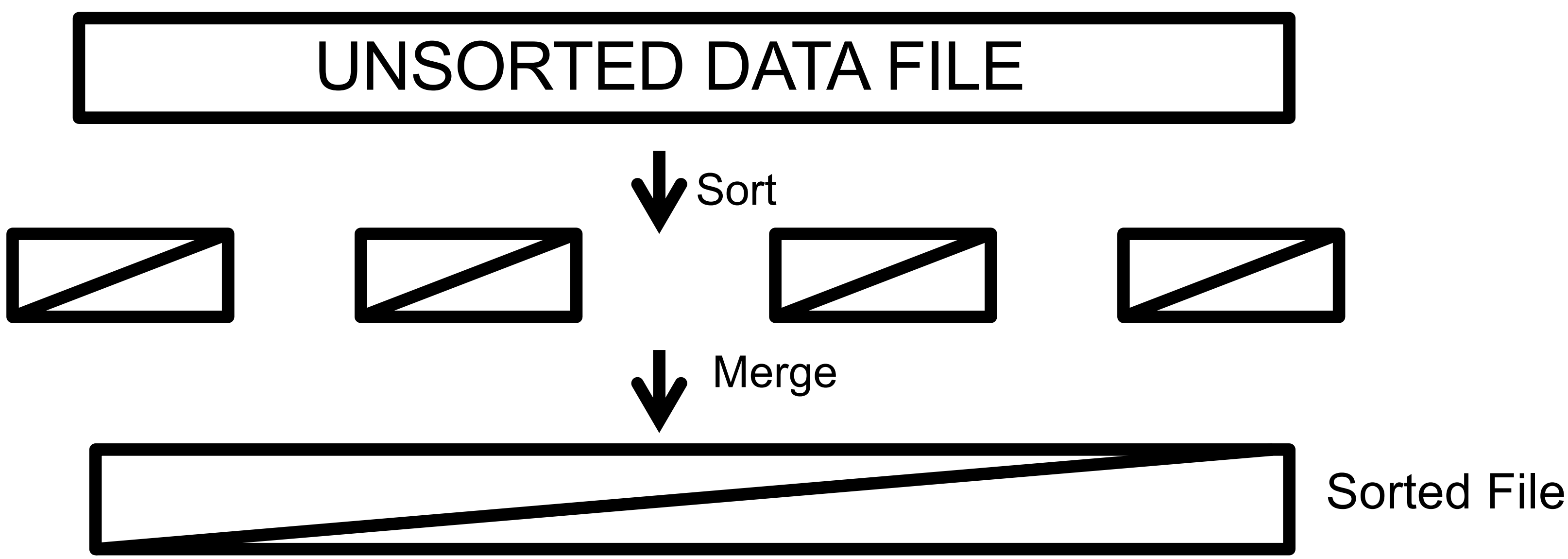
I/O – Efficient Sorting

Sorting needed in many IR and DB scenarios:

- **Inverted index construction**
- **Sort – Based join**
- **Offline B-tree index construction**
- **Duplicate elimination, group-by**

I/O – Efficient Sorting

- Data may not fit in main memory
- Many algorithms will be inefficient if data on disk
- Most popular I/O-efficient method: Merge Sort



MERGE SORT EXAMPLE

25.6 GB of data (256 million records of 100 Bytes)

100 MB of work space in main memory

Phase 1 : Repeat:

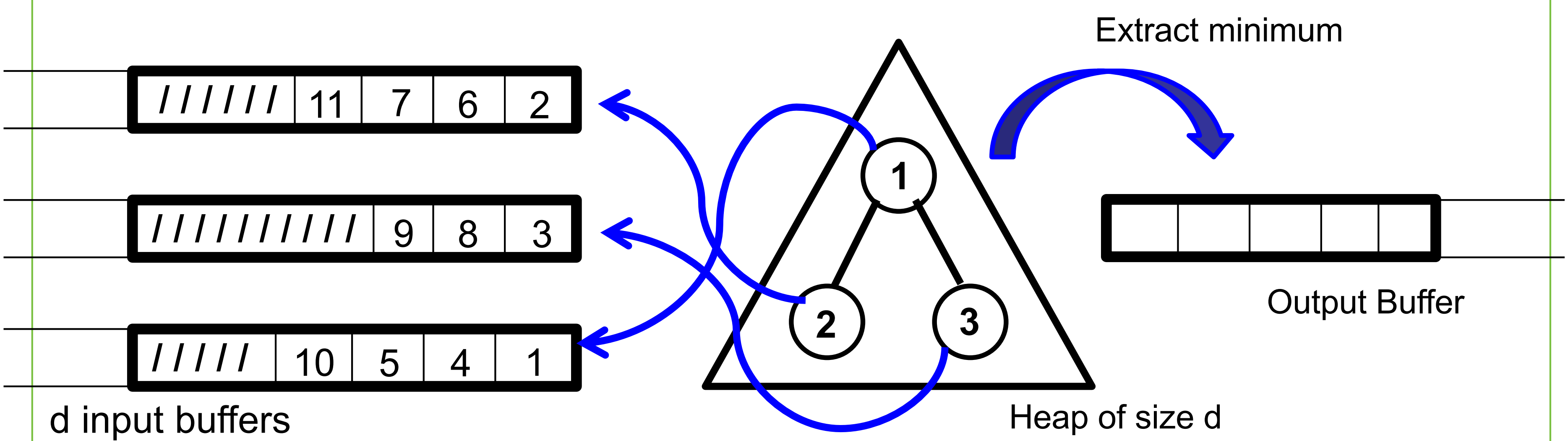
- read 100 MB data
- sort in main memory using any sorting algo
- write into a new file

Until all data read

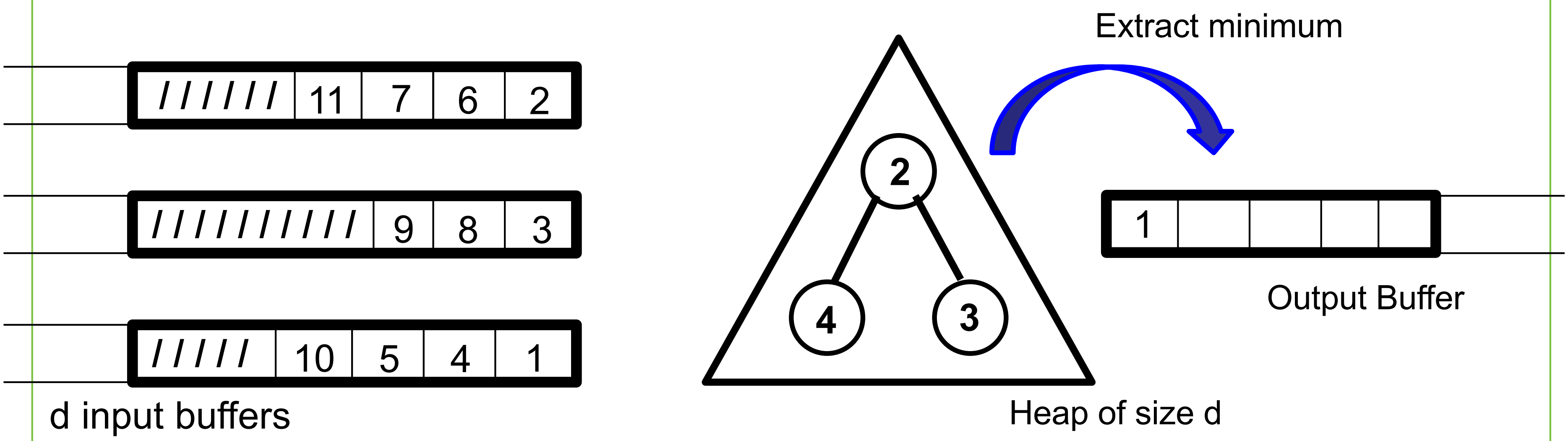
Phase 2: Merge the 256 files created in Phase 1

- in 1 pass: merge 256 files into one
- in 2 passes: merge 256 files into 16, then 16 into 1

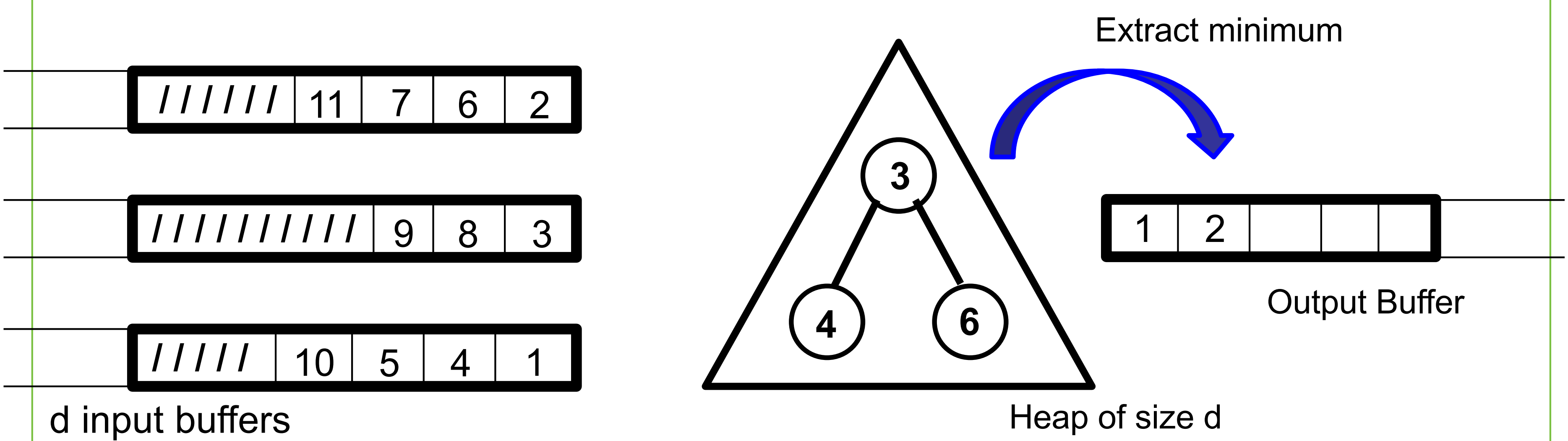
HOW TO MERGE d LISTS



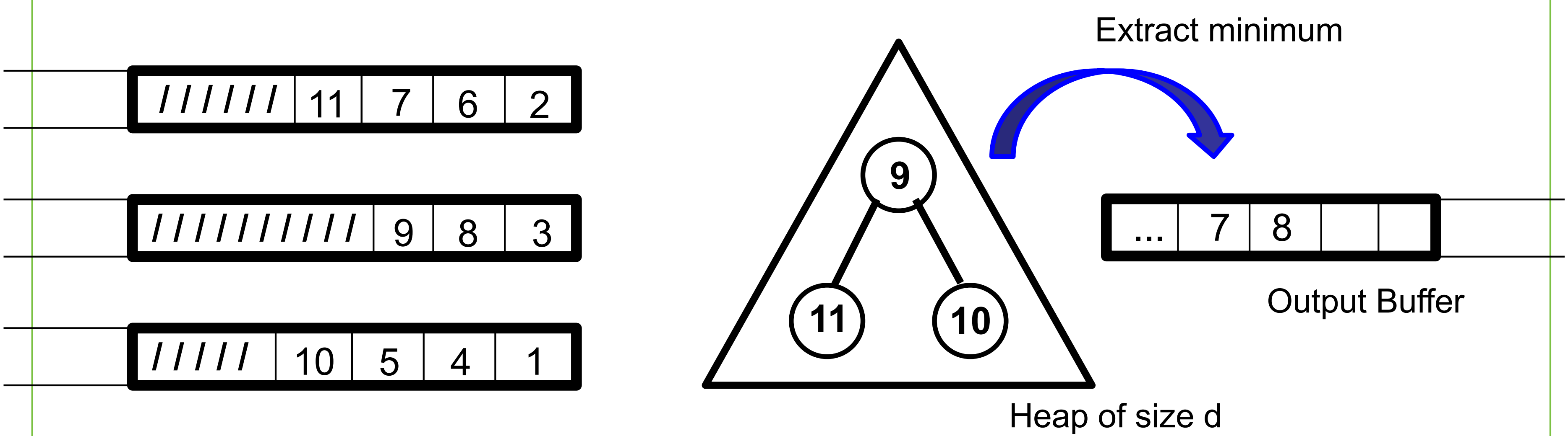
HOW TO MERGE d LISTS



HOW TO MERGE d LISTS



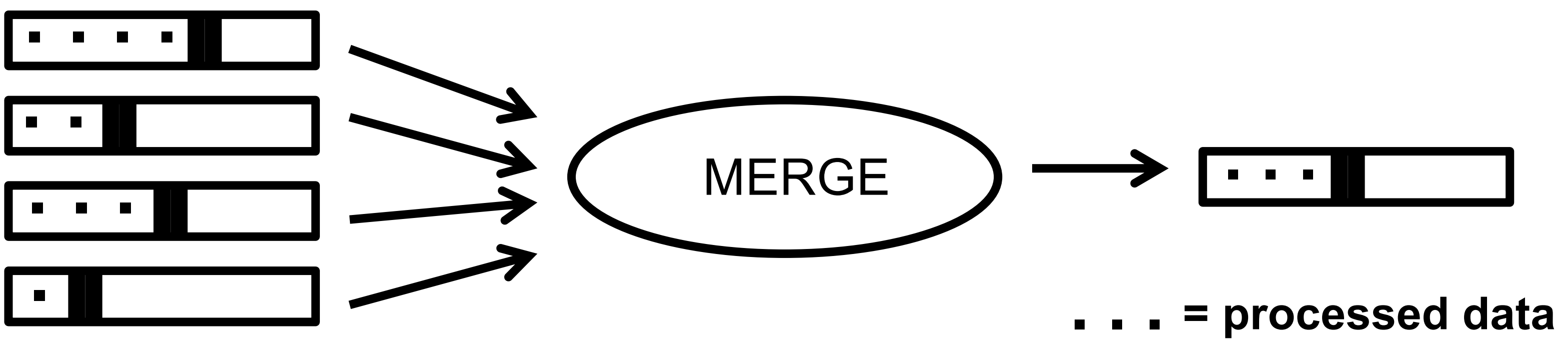
HOW TO MERGE d LISTS INTO ONE



- **Initially insert first (smallest) element from each list into heap**
- **Extract minimum and write out to output buffer**
- **Replace extracted element with the next element from the list where the minimum came from, then heapify again**
- **Repeat steps until heap is empty → all d lists are merged**

DATA ACCESS/MOVEMENT DURING MERGE

d-way merge: need d input buffers and 1 output buffer



- Files now sorted in ascending order (left to right)
- Note: heap-based merge makes pass from left to write
- If output buffer full, write it out: append to output file
- If input buffer empty, read next chunk of data from that file

Larger d: fewer passes but smaller buffers, thus slower disk I/O

Back to our Example:

25.6 GB of data to be sorted

100 MB of main memory available for sorting

$\Rightarrow$ after sort phase, we need to merge 256 sorted files of size 100 MB each

Disk with 10ms access time (seek time plus rotational latency) and 50 MB/s maximum transfer rate

Thus it takes 10 + x/50 ms to read x KB of data

What is the best choice of d?

d = 2 (8 passes since 2^8 = 256, same as standard mergesort)

d = 16 (2 passes since 16^2 = 256)

d = 256 (1 pass)

d = 2:

- 2 input and 1 output buffer of 33.33MB each
- Reading/writing one buffer of data takes :
  10 + 33333/50 = 676.66 ms

⇒ Reading all 25.6 GB in 768 pieces of 33.33 MB takes:
768 * 676.66 ~ 520 s

⇒ Each pass (read in + write out)
1040 s

⇒ All 8 passes
8320 seconds = 2.3 hours

d = 16:

- **16 input and 1 output buffer of 5.88 MB each**
- **Reading/writing one buffer of data takes :**
  10 + 5.888/50 = 127.6 ms

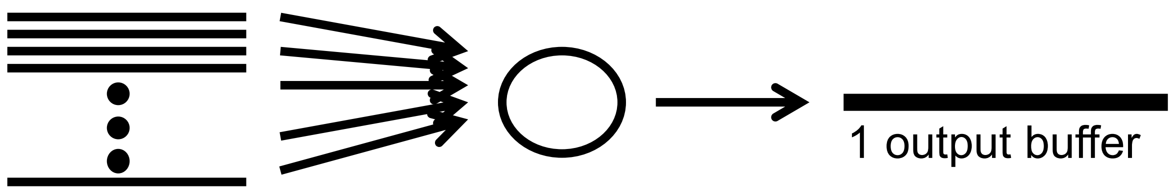
$\Rightarrow$ **Reading all 25.6 GB in 4354 pieces of 5.88 MB takes:**
4354 * 127.6 ms = 555.6 s

$\Rightarrow$ **Each pass (read in + write out)**
1111.2 sec

$\Rightarrow$ **Total time for 2 passes**
2222.4 sec or about 40 minutes

d = 256:

- **257 buffers of 389 KB each**



- **Reading/writing one buffer of data takes:**

  10 + 389/50 ~ 17.78 ms **(more than half of time on seeks)**

$\Rightarrow$ **Reading all 25.6 GB takes:**

1170 s

$\Rightarrow$ **Total (read in + write out)**

2340 seconds, or slightly slower than d=16

$\Rightarrow$ Choosing d for I/O-efficient Mergesort:

- Not too small (we want few passes)
- Not too large (we want fast disk access)
- Typically 1 or 2 passes for current machines and data sizes

Also:

- Use double buffering if CPU time counts
- Make the output buffer larger than the input buffer

E.g.:

- Choose 16 input buffers of 5 MB each
- 1 output buffer of 20 MB

Relation to Unix Sort

- This algorithm is basically the one used in Unix sort
- You may want to use Unix sort in your HW#2!
- Unix sort will create temp files in some directory
- Default is /var/tmp/ but you may change this
- Make sure that dir has enough space for temp files!
- May want to change other params, e.g., avail. mem.
- See also pseudo code on website (on project page)

I/O – Efficient Algorithms

- I/O-Efficient Algorithms: area dealing with theory and practice of designing algorithms for disk-resident data
- Many algorithms for many different problems
- Sorting: merge (mergesort) vs. split (quicksort, postal sort)
  - d-way merge and split, not binary
- Graph algorithms based on repeated sorting of edges
  - E.g., Pagerank algorithm
- Important operations: scan, split, merge, sort over the data
- I/O-efficient data structures: e.g., B+-tree
  - also degree d > 2 (but for slightly different reasons)

Conclusions

- Hard disks and SSDs are much slower than memory
- High cost of random accesses, esp. for HDDs
- When data does not fit in RAM, need to redesign algorithms to avoid random accesses to data (instead, stream/scan data)
- Area of I/O-efficient computing
- Design algorithms by repeatedly scanning, merging, splitting, sorting large data sets
- Reading and writing/appending to new files
- Also relevant to SSDs, and even RAM (a little)
  - E.g., a fully optimized in-memory merge sorts may merge 8 sorted lists at a time

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Inverted Index Creation:

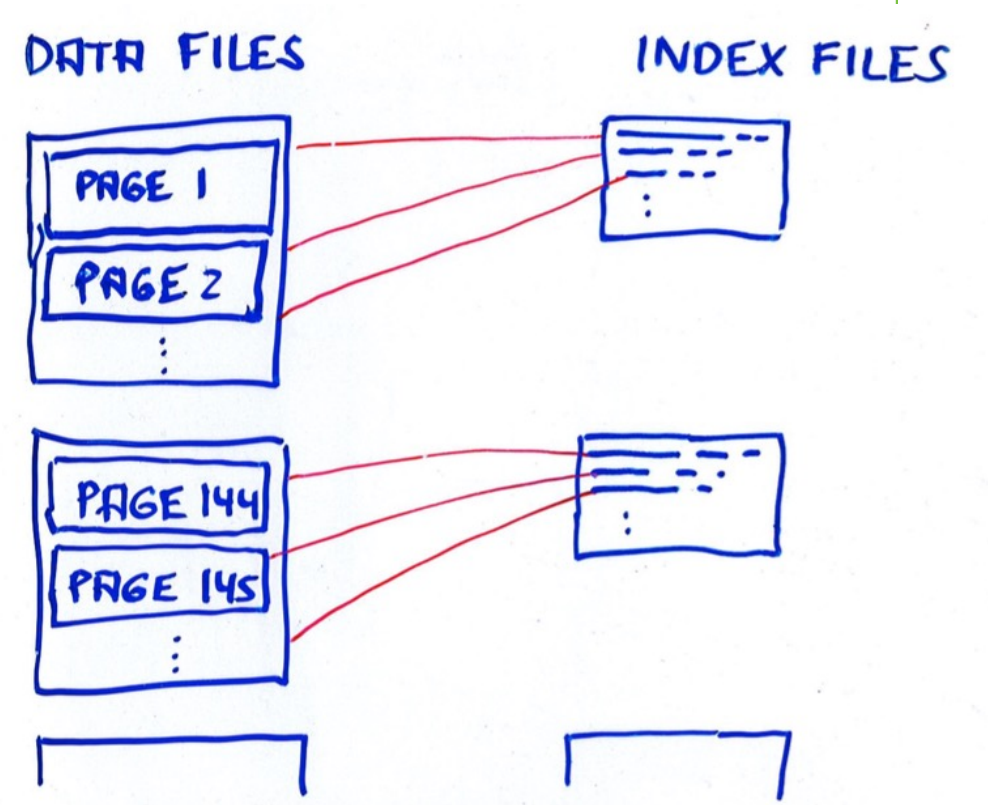
- Given a collection of documents, how do we efficiently create an inverted index?
- Historic problem, related to sorting
- Optimizing time, space (disk and RAM)
- Four basic methods:
  - Based on DIMDS (bad)
  - Merge-Sort
  - Merging Subindexes
  - Lexicon Partitioning (as in Google paper)

Problem Definition: Typical Input

- Each data file has a few hundred pages
- Including HTTP headers
- Files are, e.g., gzipped

- One index file per data file
- One line for each page
- Host name, path, IP, length, ..

- 100s of files per directory
- 100s of subdirs per directory
- And so on ...



Problem Definition: Input

Note: In many industrial setups, input is stored in scalable storage systems such as HDFS or Google's BigTable

Problem Definition: Output

- Need to build three structures
  - Inverted Index
    - with inverted lists containing postings
  - Term Lexicon
    - For each term, info about start in index, term stats
  - Page/URL Table
    - For each document, info about size, url/docID
  - And also store (parsed) pages, say in in tuple store

DIMDS Algorithm:

- DIMDS: Dynamic In-Memory Data Structure
  (search trees, hash tables etc., allowing insertions and maybe deletions)
- Parse the documents to generate intermediate postings
- Assume this posting of form (term, docid, frequency)
- Insert posting into dynamic in-memory data struct (DIMDS)
- For example, a dictionary, or one linked list per inverted list
- At the very end, write index to disk in compressed form

- Very slow when DIMDS grows beyond RAM
- Even in RAM, many cache misses (but maybe sort of OK)
- Also uses a lot of RAM in DS overhead (pointers)

DIMDS algorithm: (ctd.)

- Note: we could use many types of DIMDS
- Simple example: put postings into a binary search tree
- Two problems with DIMDS:
- DIMDS are slower than bulk reorganization
  - E.g., bulk-building a search tree is much faster than inserting elements one by one (where bulk-building uses sorting)
  - But may be acceptable if data fits in memory
- DIMDS have horrible performance if data larger than mem
  - Swapping generates lots of small I/O requests
- Note: DIMDS could still be used e.g. for the lexicon, which might however slow things down.

Merge-Sort Index Building:

- (1) Parse documents, generate postings, and write postings (term, docid, freq) out to new file on disk
- (2) Use I/O-efficient sorting to sort by (term, docid)
- (3) Scan sorted file, convert into compressed index

- Fairly fast, scales to large data sets
- Can overlap (2) with parts of (1) and (3)
- Needs temp storage on disks
- Could use terms or term IDs

What are Term IDs?

- Idea: replace ascii terms, e.g., “mouse” with integer term IDs
- Why? Integers might be smaller, can be stored in a fixed-size binary format (32-bit ints), maybe faster for sorting and I/O
- How? There are many ways:
  - We could assign term ID 0 to the first term inserted into the lexicon, then term ID 1 to the next, etc.
  - We could assign term ID 0 to the most frequent term, etc
  - We could assign term IDs alphabetically
  - The last two choices require two passes over data, or sampling
- Lexicon stores mapping from terms to term IDs
- Requires to keep and access lex in memory during parsing
- We can store parsed documents as sequence of term IDs

Merge-Sort Improved:

- (1) Parse documents, generate postings, and write postings out to a large buffer in RAM
  - Whenever buffer full, sort and write into temp file
- (2) Use I/O-efficient merging to merge temp files
  - During final merge, pipe output buffer into next step
- (3) Scan sorted file, convert into compressed index

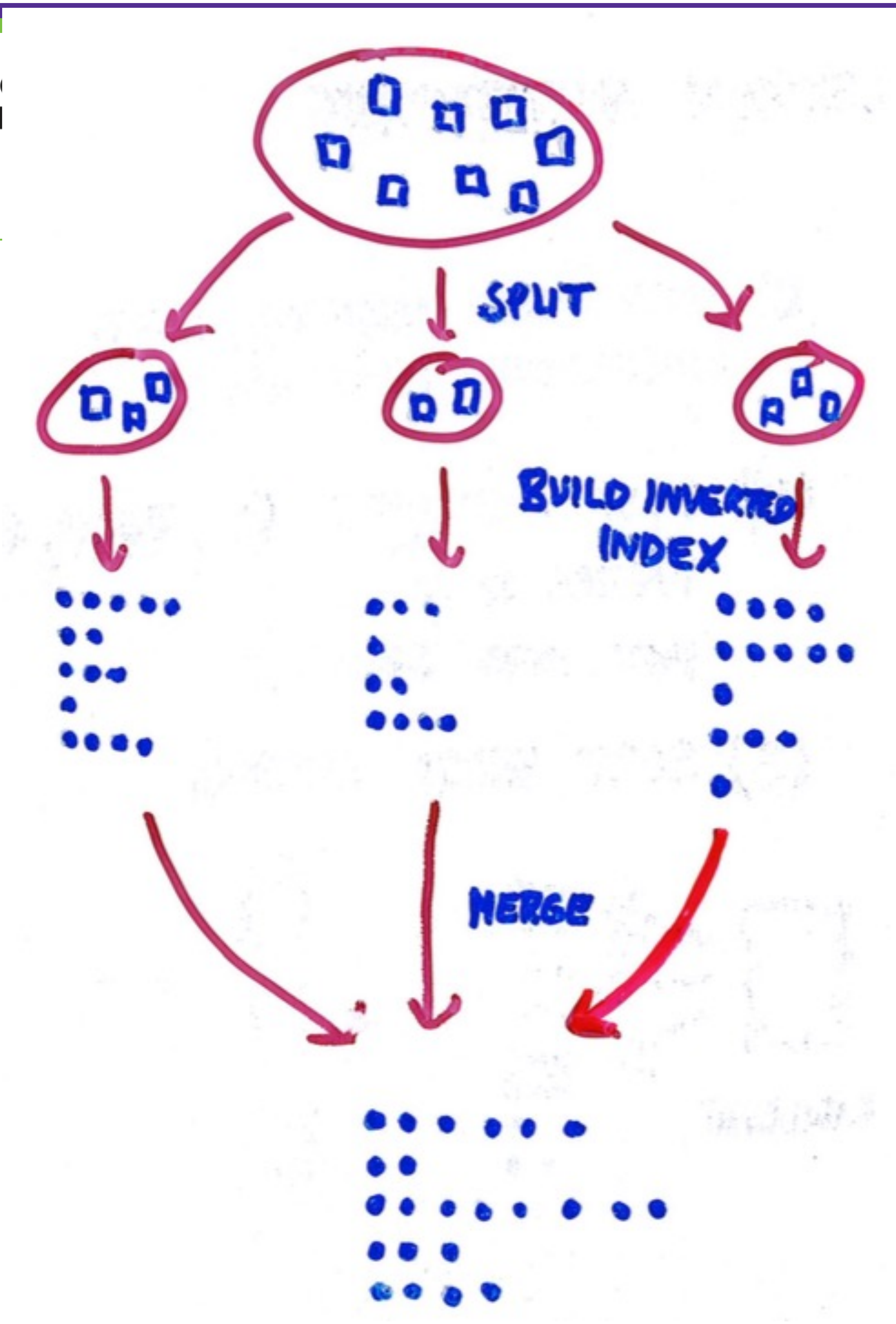
- Cost: scan documents once, then write and read intermediate postings, then write out final index
- Can try to compress postings in all steps

Merging Subindexes Algorithm:

- (1) Partition collection into subsets that fit in RAM
- (2) For each subset, build a separate inverted index, say using the Linked List approach
- (3) Then merge indexes into one index

- Need code to merge complete indexes (including merging of inverted lists, lexicon, and page table)
- Can be done by extending merge sort
- Advantage: makes it easier to keep temp data compressed during index building
- Also, index merge is a useful tool (index updates)

NYU | POLYTE
OF ENGI



Lexicon Partitioning Algorithm:

- (1) Partition lexicon into partitions (called barrels in Google paper) so that each partition receives about the same number of posting
- (2) Now parse and generate postings as before, and assign each posting to its partition
  - One temp output file for each partition
  - Write posting to output buffer of its partition
  - Whenever output buffer full, append to file
- (3) Sort all the partitions
- Note: if partitions do not fit in RAM, may recurse

Lexicon Partitioning Algorithm:

- How to partition the lexicon
  - By first character, into 26 or 256 (ascii) barrels?
    - Bad idea, uneven distribution!
  - By hashing words to [0 .. p-1]? (p # of partitions)
    - Easy and fast, but “apple”, “apples” in different barrels
  - By creating alphabetic ranges?
    - (e.g., barrel #1 from A to CAT, #2 from CAU to EET, ...)
    - Needs initial sampling to estimate distribution
    - Basically an I/O-efficient version of quicksort
    - Multi-way split instead of multi-way merge

Lexicon Partitioning Algorithm:

- Note that this is similar to quicksort algorithm
- But we are splitting into >> 2 subsets
- In fact, optimized quicksort also splits into >2 sets
- Mergesort indexing is bottom-up, this is top-down
- Basic complexity is about the same
- But split instead of merge

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Discussion:

- When optimized, all methods (except DIMDS) have somewhat comparable performance
- Read original data, write & read intermediate posting data, finally write out final index
- Details depend on various choices
  - Write intermediate data in binary or ascii?
  - Keep intermediate postings compressed?
  - Use terms or term IDs?
  - Do we need the terms/lists in index alphabetically sorted?
- Recall: use sequences of scans, merges, splits, sorts to design I/O-efficient methods

Many Design Choices:

- Do lexicon and URL table fit in memory?
  - might have to use compressed/concise data structures
  - how many accesses to these per query?
  - or cache most common query terms in the lexicon?
- Term IDs, or terms in ascii?
  - ascii terms make it easy to keep inverted lists in alphabetic order
  - how to assign term IDs? (in order, by frequency, alphabetic)
  - note: there are no explicit term IDs in final index structures
  - when merging indexes, better to have lists in same order
- How to assign docIDs?
  - in crawl or index order, alphabetically by URL, by quality or PR?
  - docID assignment can have major impact on size and speed

Today's Lecture

- Introduction
  - Indexing and Parsing
  - Index Structures and Layout
- Disks and I/O-Efficiency
  - Hard Disks & SSDs
  - Modeling Disk Performance
  - I/O – Efficient Sorting
- Index Building
  - Four index building algorithms
  - I/O-efficiency of index building
  - Discussion of choices and setups
  - Index Maintenance

Index Maintenance:

- How to update an index as new data is crawled?
- The basic challenge:
  - a new document is crawled and parsed
  - contains, say, 700 words, including 250 distinct words
  - 250 postings need to be inserted in 250 distinct inverted lists
  - new document gets new (highest) docID
  - 250 "random" writes to disk (assuming space at end of list)
- Solution: Lazy algorithms with good amortized performance
  - compare to algorithm for dynamic tables in sec 17 of CLRS book
  - such algorithms are even more useful on disk!
- Nutshell: build small dynamic index in mem, and merge later

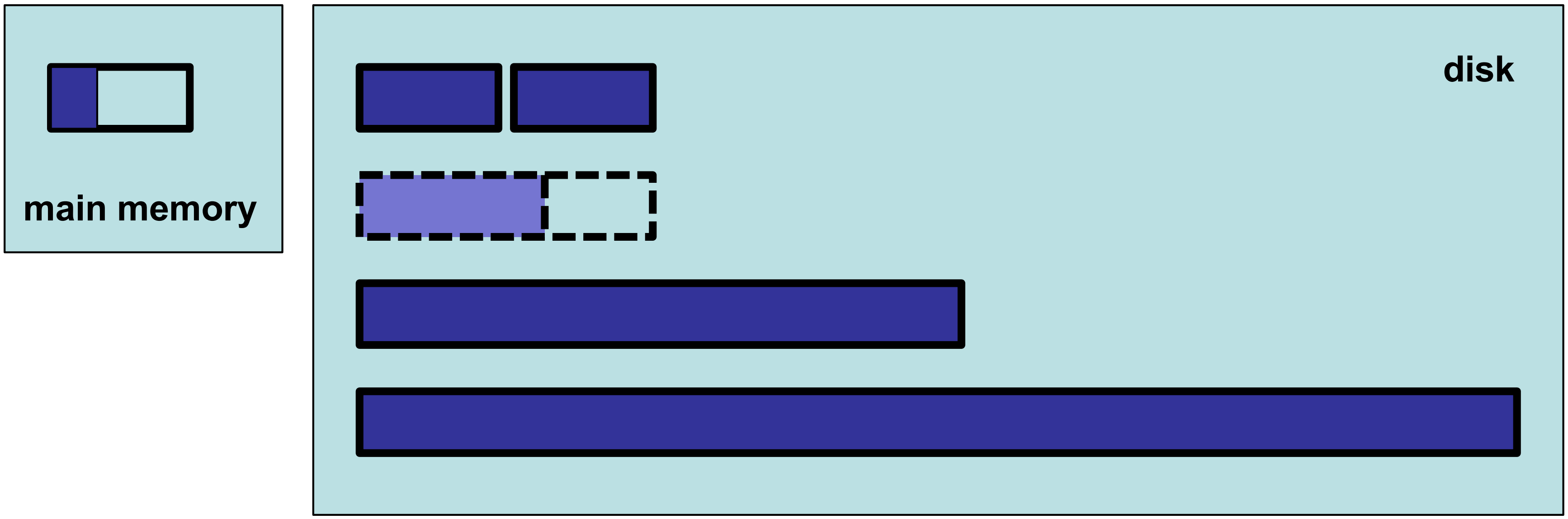
Index Maintenance Setup:

- Three operations that need to be supported
  - *Inserting* a new document and its postings
  - *Deleting* an old document and its postings
  - *Replacing* an indexed document with a new version
- Let us focus first on inserting a new document
- Maintain an additional dynamic index in memory
  - Maybe based on dictionary or other dynamic in-mem DS
  - In addition to “main” index structure on disk
- Evaluate a query on both indexes, combine results
- When in-memory index too large, merge them

Example Algorithm: Logarithmic Merge

- Used in *Lucene* open-source search tool
- When in-memory index too large, write it out to disk as a new index
- Whenever there are two indexes of roughly same size, merge them into one index
- Every query is evaluated on all current indexes
- Merges are scheduled to run in background
  - To make sure merging does not disrupt query processing

Example Algorithm: Logarithmic Merge



- In figure, two small on-disk indexes are currently being merged
- Resulting merged index is partially built, in dashed outlines
- Up to ~ log(total_index_size / mem_index_size) indexes
- But not all of them will exist

Implementing Deletion and Update

- Deletion: put docID on blacklist of deleted docs
  - During query processing, ignore deleted results
  - During merging, remove posts from deleted docs
- Replacing an old with a new version of a document

Implementing Deletion and Update

- Deletion: put docID on blacklist of deleted docs
  - During query processing, ignore deleted results
  - During merging, remove posts from deleted docs
- Replacing an old with a new version of a document
  - Delete document, then insert with new (higher) docID

Implementing Deletion and Update

- Deletion: put docID on blacklist of deleted docs
  - During query processing, ignore deleted results
  - During merging, remove posts from deleted docs
- Replacing an old with a new version of a document
  - Delete document, then insert with new (higher) docID
- But what if new document has only minor changes?
  - Say, term “cat” was replaced with term “dog”
  - Do we really need to reindex as new document?
- Well, now things get interesting ...

Optimizing Document Updates

- Suppose term “cat” was replaced with “dog”
- Assume non-positional index: (docID, freq)

Optimizing Document Updates

- Suppose term "cat" was replaced with "doc"
- Assume non-positional index: (docID, freq)
- Idea: negative postings:
  - Create a posting for "cat" with frequency -1
  - Create a posting for "dog" with frequency +1
  - Document keeps its old docID
- During merges, cancel out negative and positive postings for same term and docID
- During query processing, try to do the same
- Or write new posts for cat and dog with correct freqs
- Positional index: what if all positions shift?

Optimizing Document Updates

- Suppose term “cat” was replaced with “doc”
- Assume non-positional index: (docID, freq)
- Idea: negative postings:
  - Create a posting for “cat” with frequency -1
  - Create a posting for “dog” with frequency +1
  - Document keeps its old docID
- During merges, cancel out negative and positive postings for same term and docID
- During query processing, try to do the same
- Or write new posts for cat and dog with correct freqs
- Positional index: what if all positions shift?

This gets more complicated/interesting!

Summary: Index Maintenance

- Using lazy structures with good amortized cost
- E.g., Apache Lucene, Google Caffeine
- Logarithmic merge and variations
- Actually, logarithmic not always best
- Depends on how often a term is inserted and queried
- Best strategy uses different policies for diff terms
- Efficiently replacing docs with new versions tricky

- Also, this is only about inverted index
- How about Pagerank, spam score, clicks, etc?
- Major challenge for data-mining/indexing pipeline