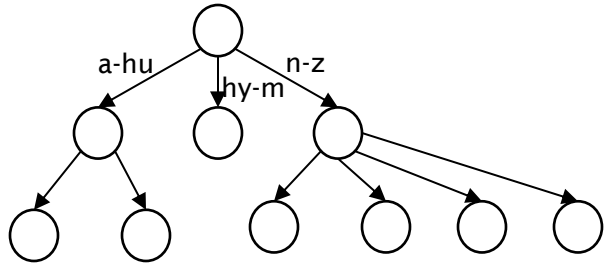
Introduction to

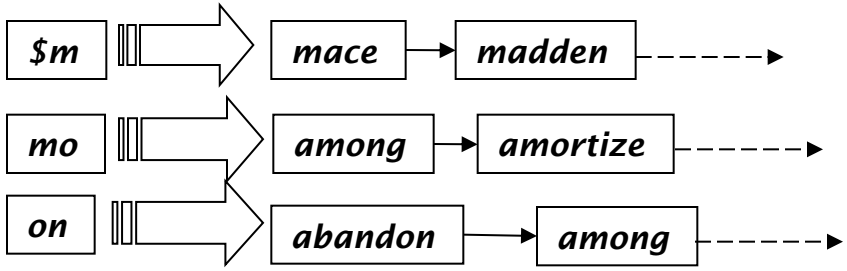
Information Retrieval

Lecture 4: Index Construction

Plan

- Last lecture:
  - Dictionary data structures
  - Tolerant retrieval
    - Wildcards
    - Spell correction
    - Soundex
- This time:
  - 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.

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.

Hardware assumptions

symbol	statistic	value
s	average seek time	5 ms = 5 x 10^{-3} s
b	transfer time per byte	0.02 µs = 2 x 10^{-8} s
	processor's clock rate	10^{9} s^{-1}
p	low-level 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

RCV1: Our collection for this lecture

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.
- The collection we'll use isn't really large enough either, but it's publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Reuters RCV1 statistics (Rounded)

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

Recall IIR 1 index construction

- Documents are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

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

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

Hash based in-memory index construction

- Another in-memory index construction method is to use hash-tables
  - Append (docid, pos) to the existing (partial) postings list of the token; create a new postings list if necessary
- Generally, faster than sorting-based method
- Further optimizations
  - Dealing with collision: insert-at-back and move-to-front heuristics
  - Saving space and time: use ArrayList to implement postings lists

Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.

Sort-based index construction

- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks (you can, but much more complex)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry *(term, doc, freq)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
  - So … we can do this in memory in 2009, 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.

Use the same algorithm for disk?

- 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.
- We need an external sorting algorithm.

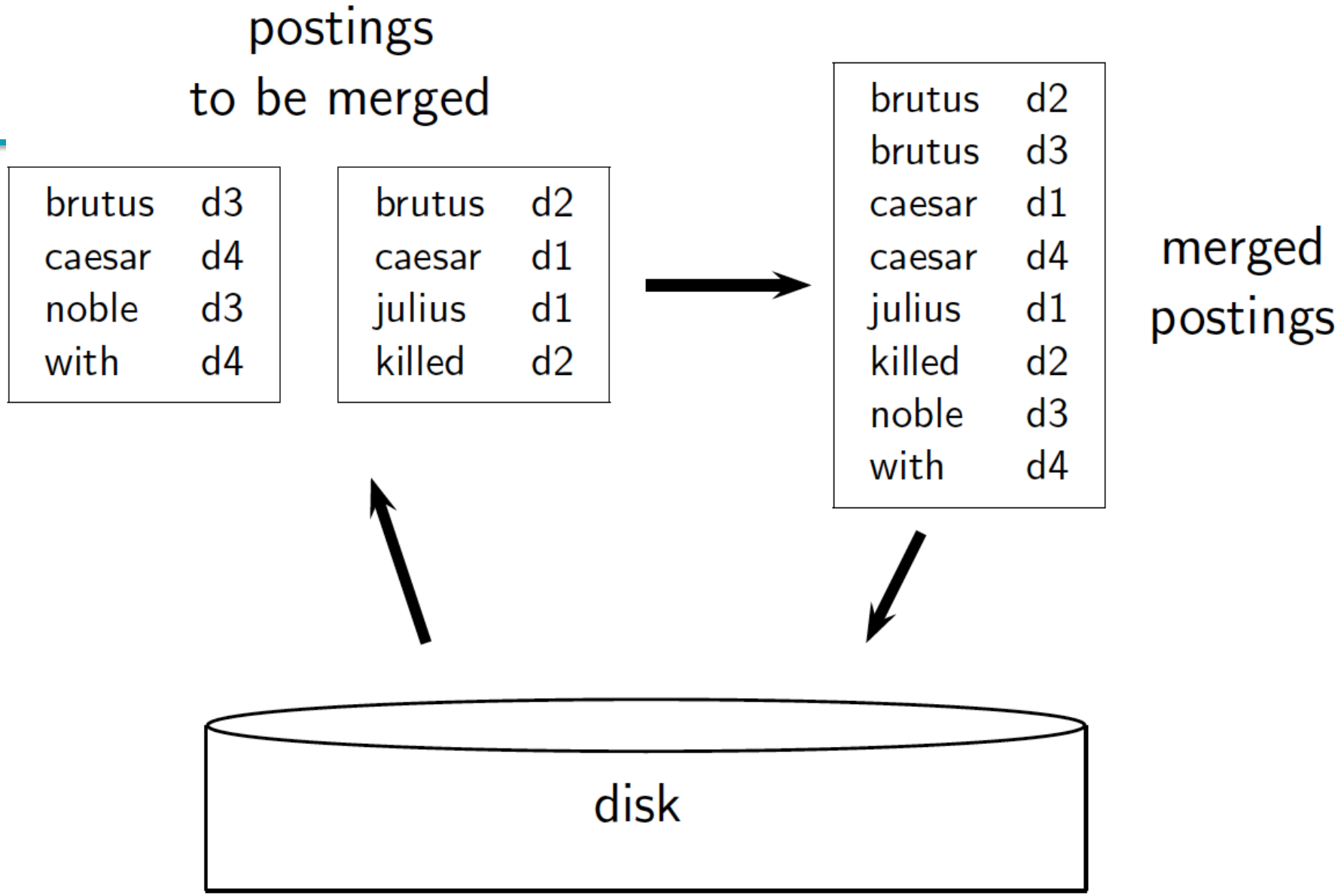
Bottleneck

- 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

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

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

- 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 Block ~ 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
to be merged
brutus d3
caesar d4
noble d3
with d4
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

Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes 2*N* ln *N* expected steps
  - In our case 2 x (10M ln 10M) steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
- 10 times this estimate – gives us 10 sorted *runs* of 10M records each.
- Done straightforwardly, need 2 copies of data on disk
  - But can optimize this

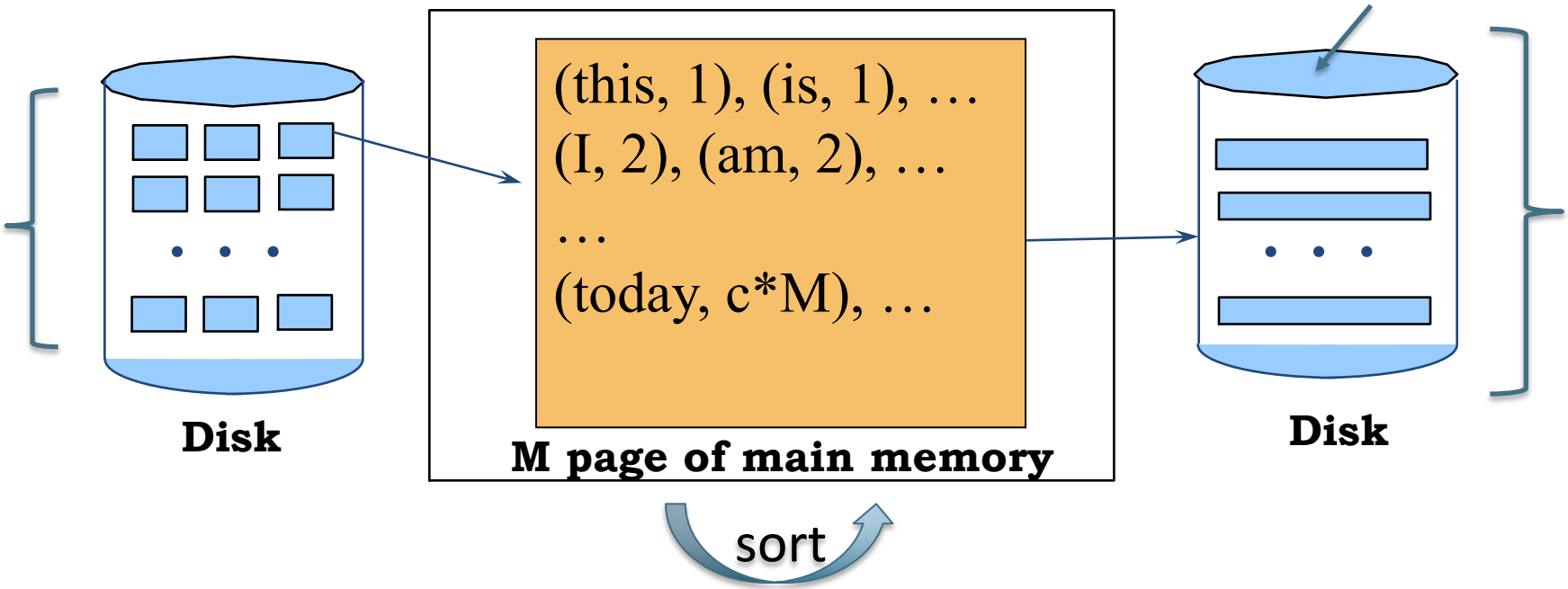
```
BSBIndexConstruction()
 1  n ← 0
 2  while (all documents have not been processed)
 3  do n ← n + 1
 4     block ← ParseNextBlock()
 5     BSBI-Invert(block)
 6     WriteBlockToDisk(block, f_n)
 7  MergeBlocks(f_1, ..., f_n; f_merged)
```

Example

- Settings
  - **B:** Block/page size
  - **M:** Size of main memory in pages (e.g., = 10 blocks)
  - **N:** Number of documents (e.g., = 10000)
  - **R:** Size of the (term, docID) pairs one document emits.
- Simplifying assumptions:
  - **R:** the same for all documents
  - **B** = c***R**, for some integer c (e.g., c = 5)
  - All I/Os have the same cost

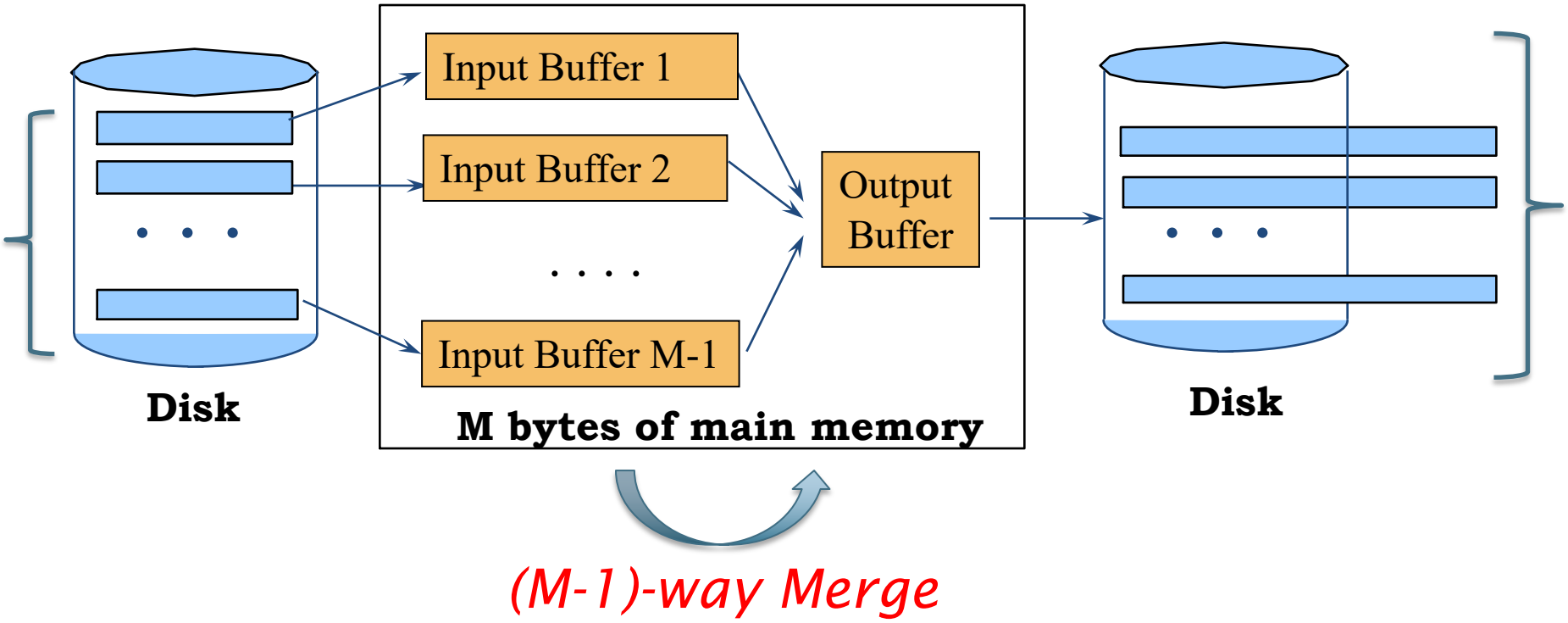
External Merge-Sort: Phase I

- Phase I: load the (term, docID) pairs from (M*B)/R documents (*at a time*) into M buffer pages; sort
  - Result: (initial) runs of length M pages
  - # of runs = 200



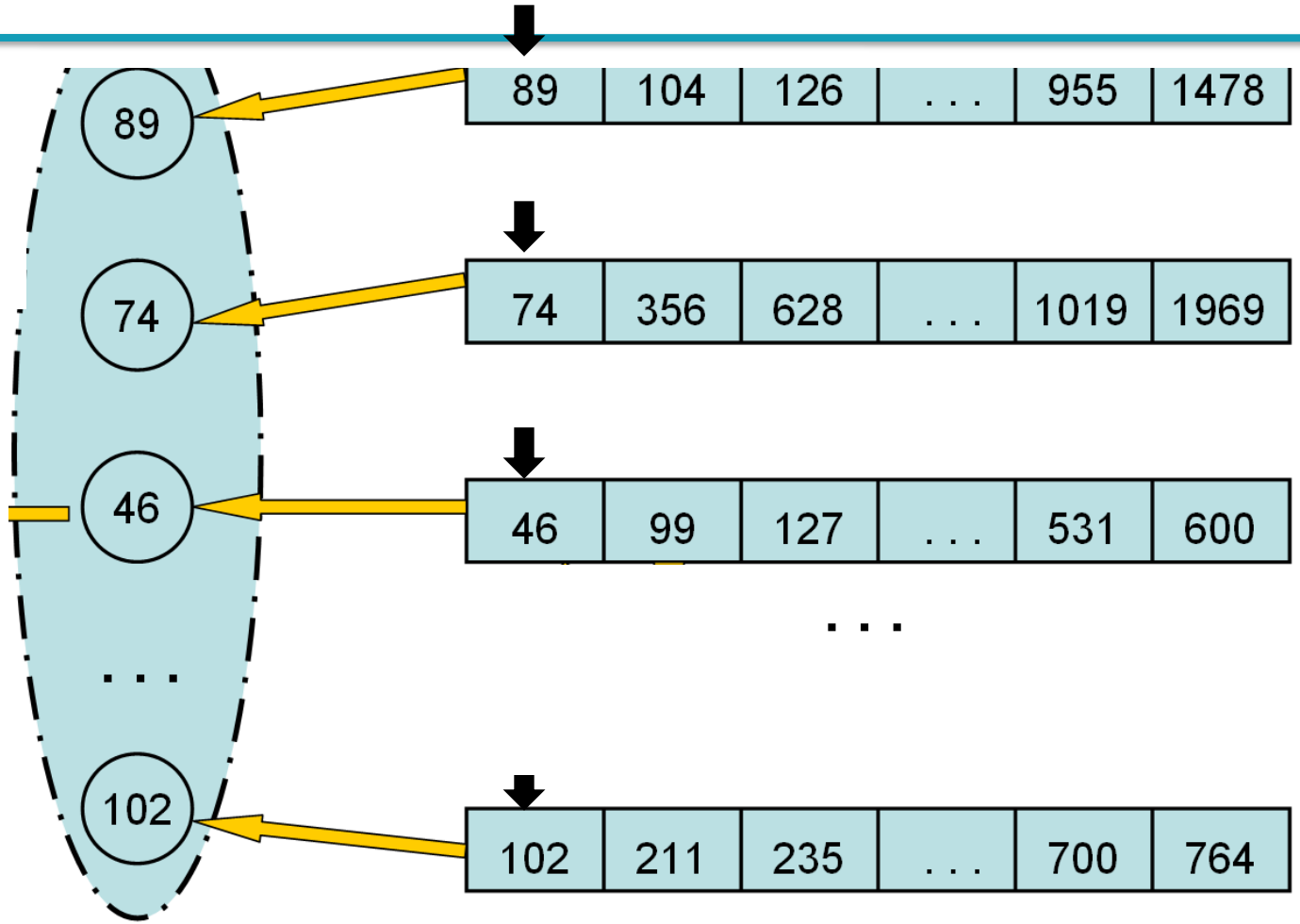
Phase II /1

- **Recursively** merge (up to) M **– 1** runs into a new run
- Result: runs of length M (M – 1) pages



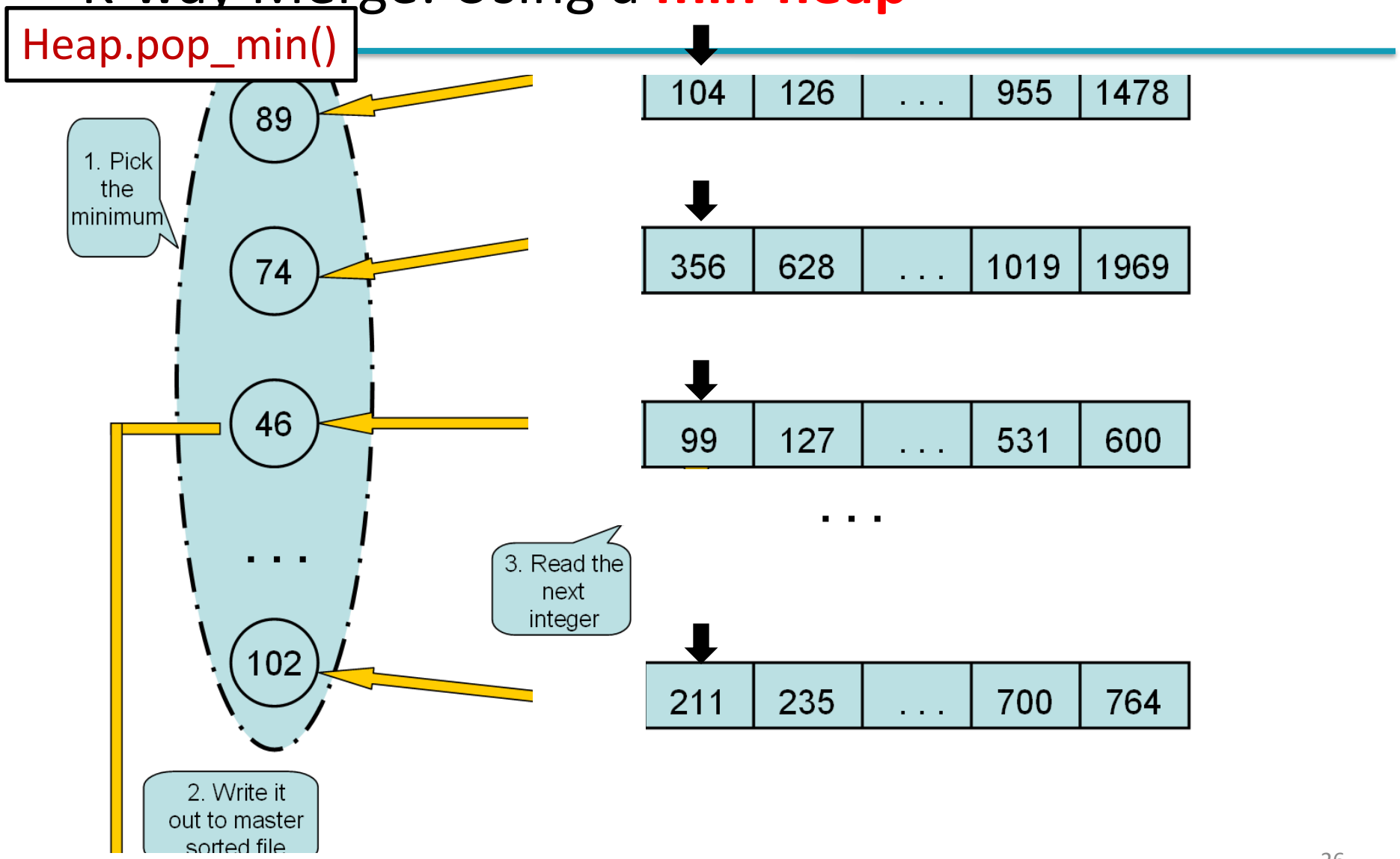
adapted from https://karticks.wordpress.com/2009/07/29/the-mapreduce-design-pattern-demystified/

K-way Merge: Using a **min-heap**



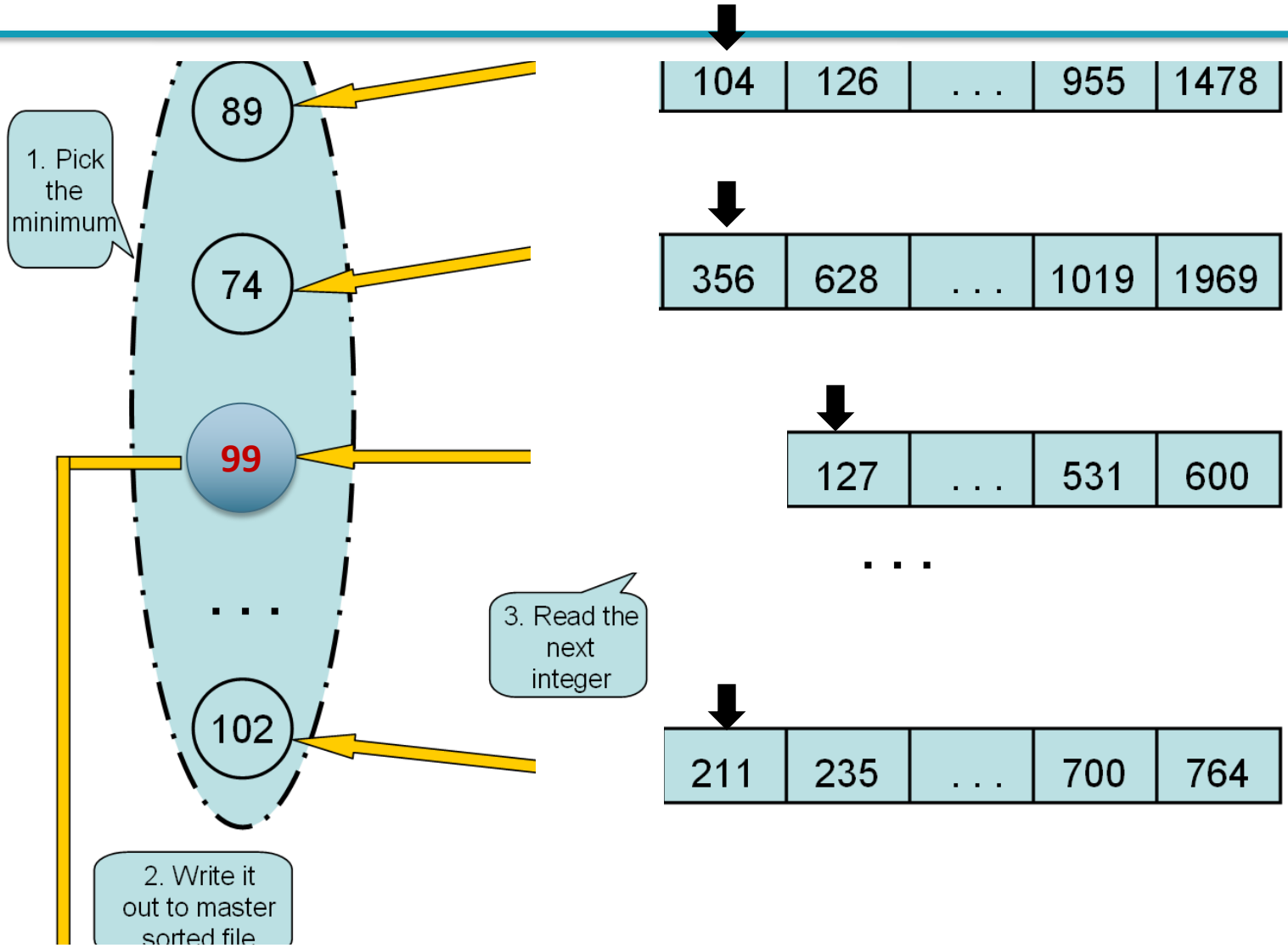
adapted from https://karticks.wordpress.com/2009/07/29/the-mapreduce-design-pattern-demystified/

K-way Merge: Using a **min-heap**



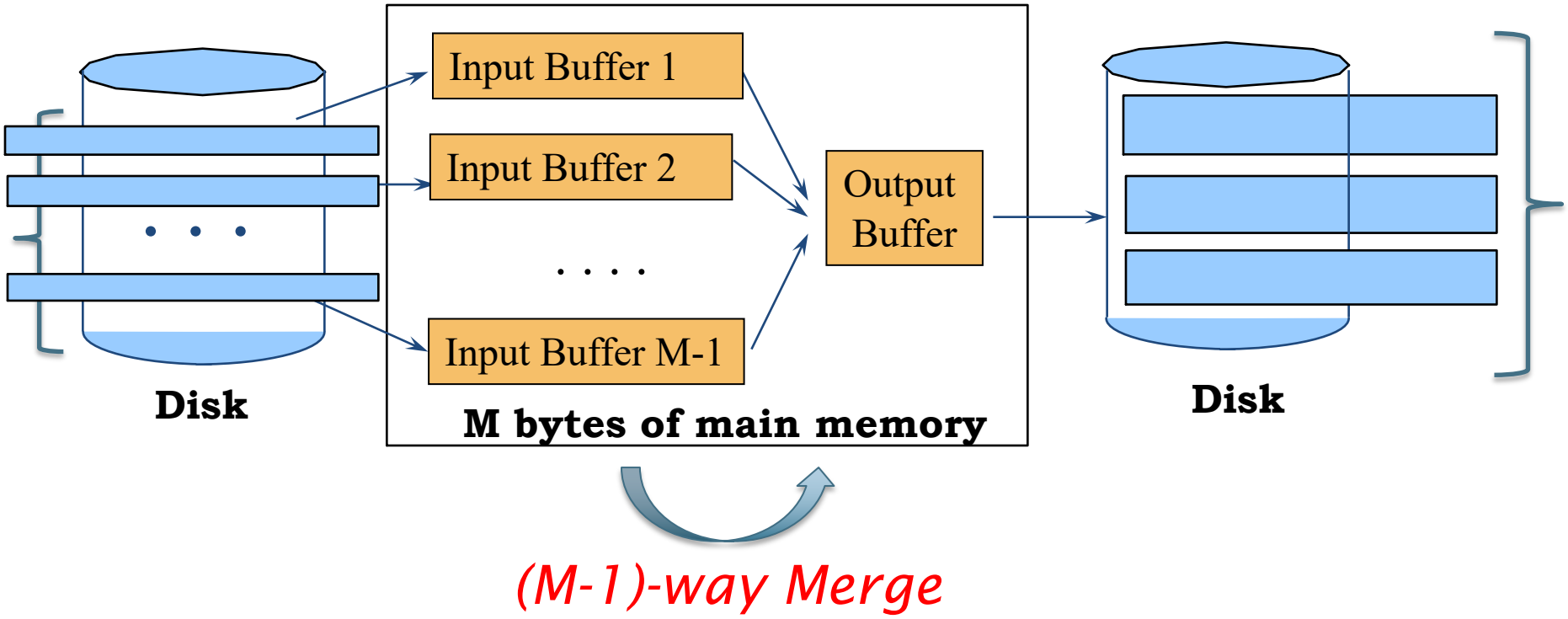
adapted from https://karticks.wordpress.com/2009/07/29/the-mapreduce-design-pattern-demystified/

K-way Merge: Using a **min-heap**



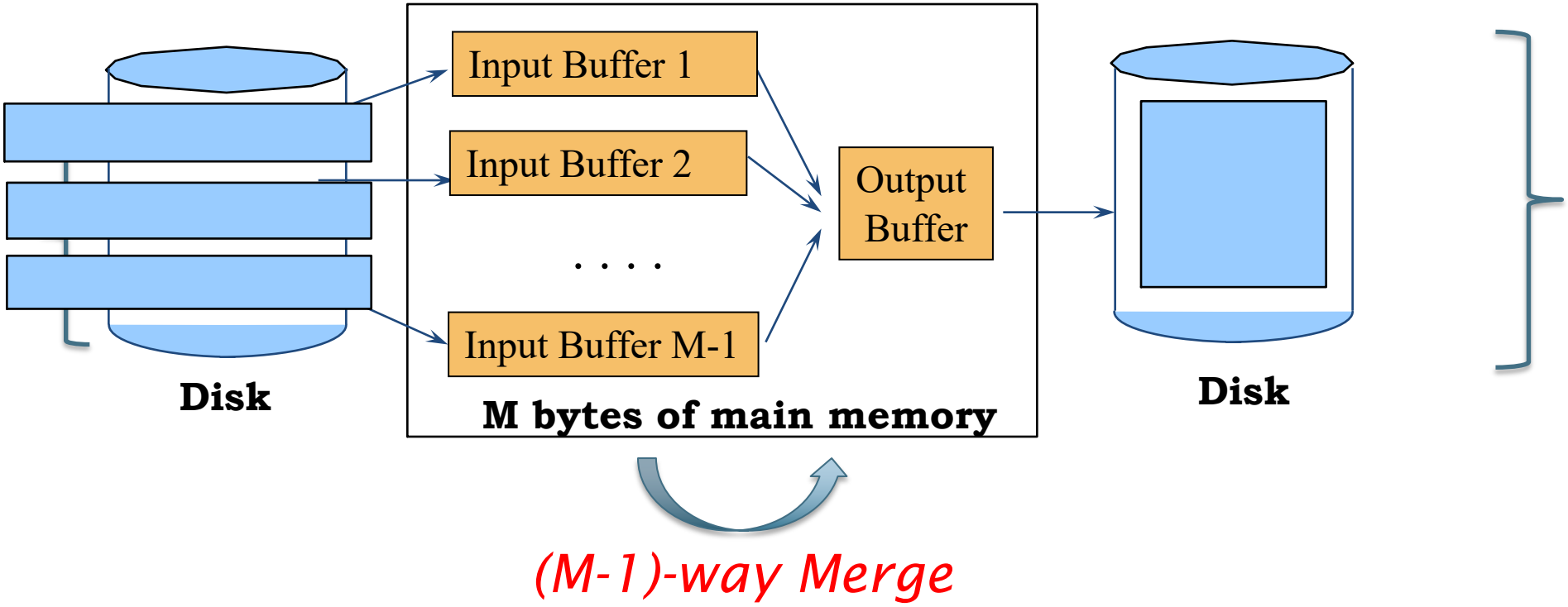
Phase II /2

- **Recursively** merge (up to) M **– 1** runs into a new run
- Result: runs of length M $(M - 1)^2$ pages



Phase II /3

- **Recursively** merge (up to) M **– 1** runs into a new run
- Result: a single run



Cost of External Merge Sort

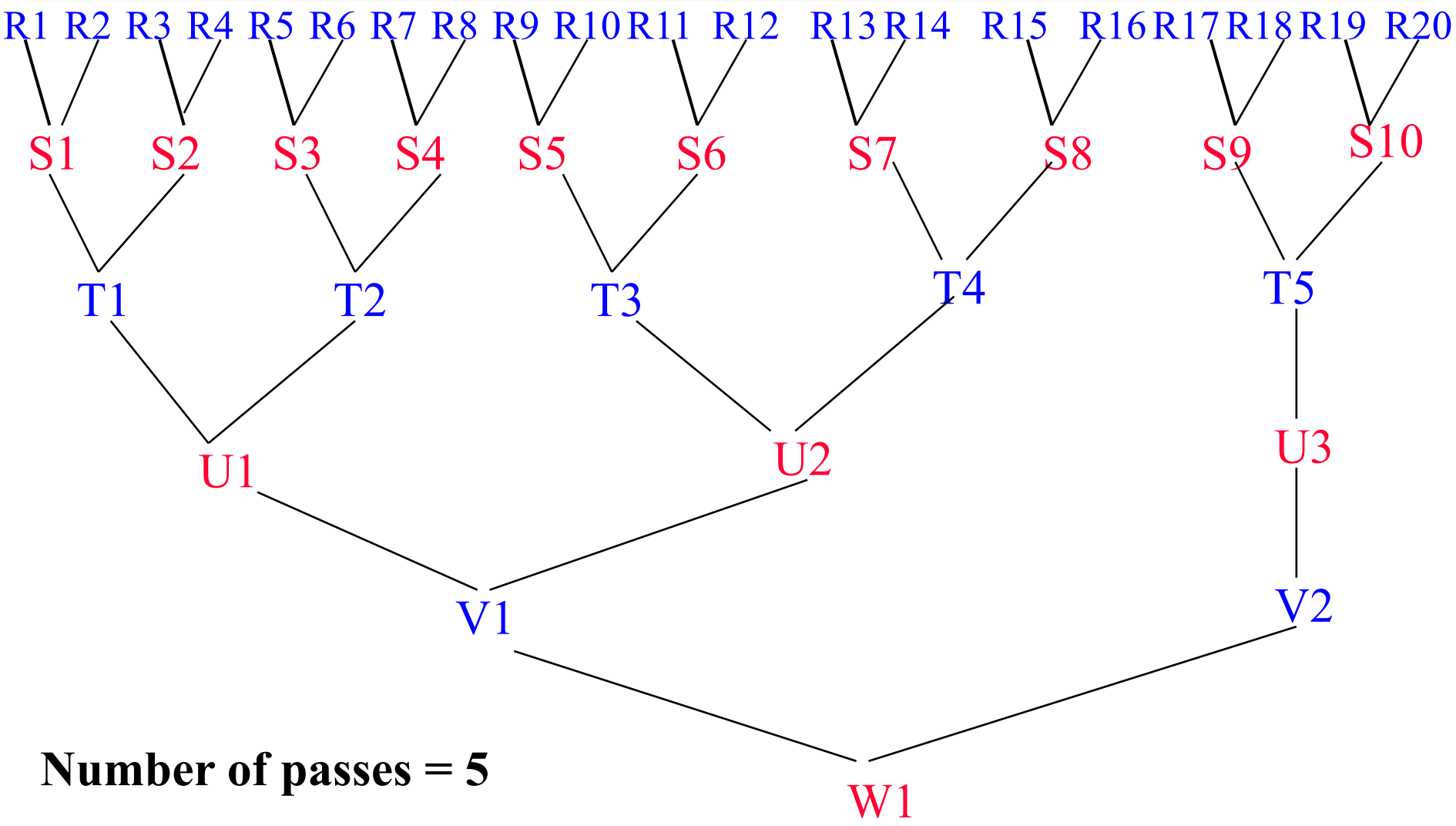
- Number of passes: $1 + \left\lceil \log_{M-1} \left\lceil \frac{NR}{MB} \right\rceil \right\rceil$

  Total I/O cost: $2 \cdot \left(\frac{NR}{B}\right) \cdot \left(1 + \left\lceil \log_{M-1} \left\lceil \frac{NR}{MB} \right\rceil \right\rceil\right)$ blocks/pages

Another Example

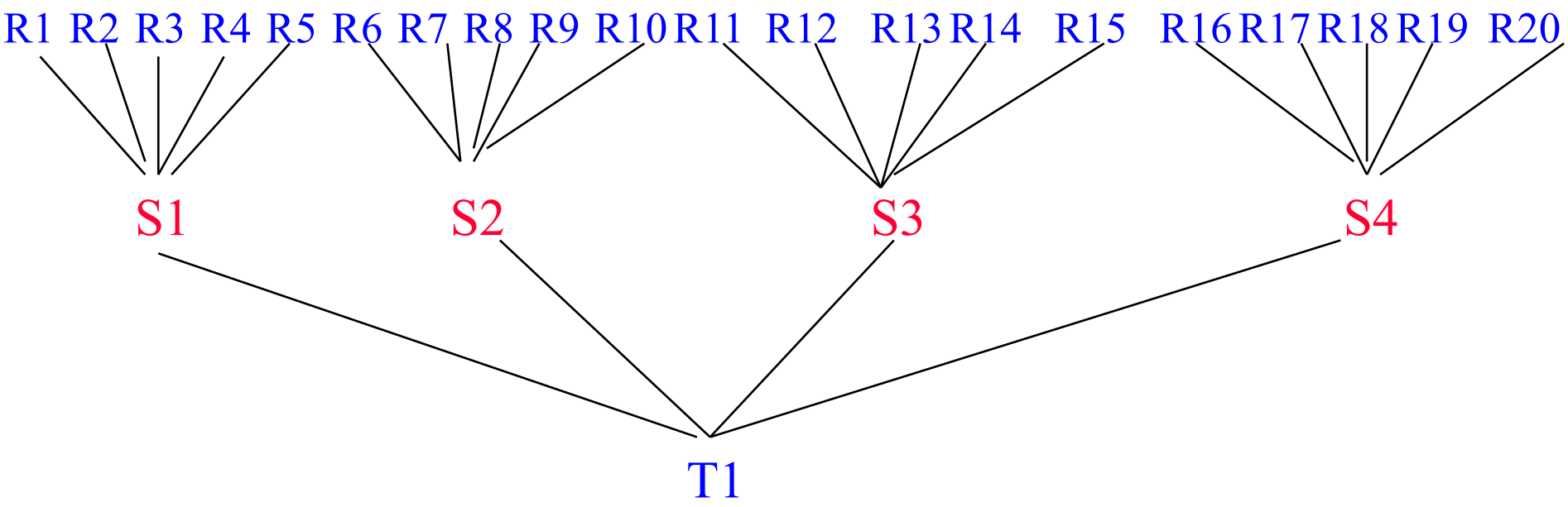
- How much data can we sort with 10MB RAM?
  - Assume B = 4KB
  - 1 pass ➔ 10MB "data"
  - 2 passes ➔ ≈25GB "data" (M-1 = 2559)
  - 3 passes ?
- Can sort most reasonable inputs in 2 or 3 passes !

Example: 2-Way Merge for 20 Runs



Number of passes = 5

Example: 5-Way Merge for 20 Runs



Number of passes = 2

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.

SPIMI-Invert

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

- Merging of blocks is analogous to BSBI.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
- See next lecture

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?

For those interested in the topic, read the textbook for distributed indexing using the Map-Reduce paradigm.
Also check out:
http://terrier.org/docs/v3.5/hadoop_indexing.html

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 – Immediate Merge

- Maintain “big” main index
- New docs go into “small” auxiliary index
  - Merge immediately with the big main index when memory is full
- 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.)

Another Extreme – No Merge

- Whenever memory is full, write the sub-index to the disk
  - Never merge sub-indexes
- Pros:
  - High indexing performance
- Cons:
  - Slow query performance
    - Require $\Omega(|C|/M)$ seeks to fetch the inverted list for a term

Compromise – Logarithmic merge

- Comprise of the previous two extremes
- Generation of a sub-index
  - The one directly created from in-memory index has generation = 0
  - Merge of multiple sub-index with max generation = g gives a new index with generation = g+1
- Invariant: no two sub-indexes can have the same generation
- When memory is full, create I_0
  - If we have two sub-indexes of generation g, merge them to form a single sub-index of generation g+1

Illustration

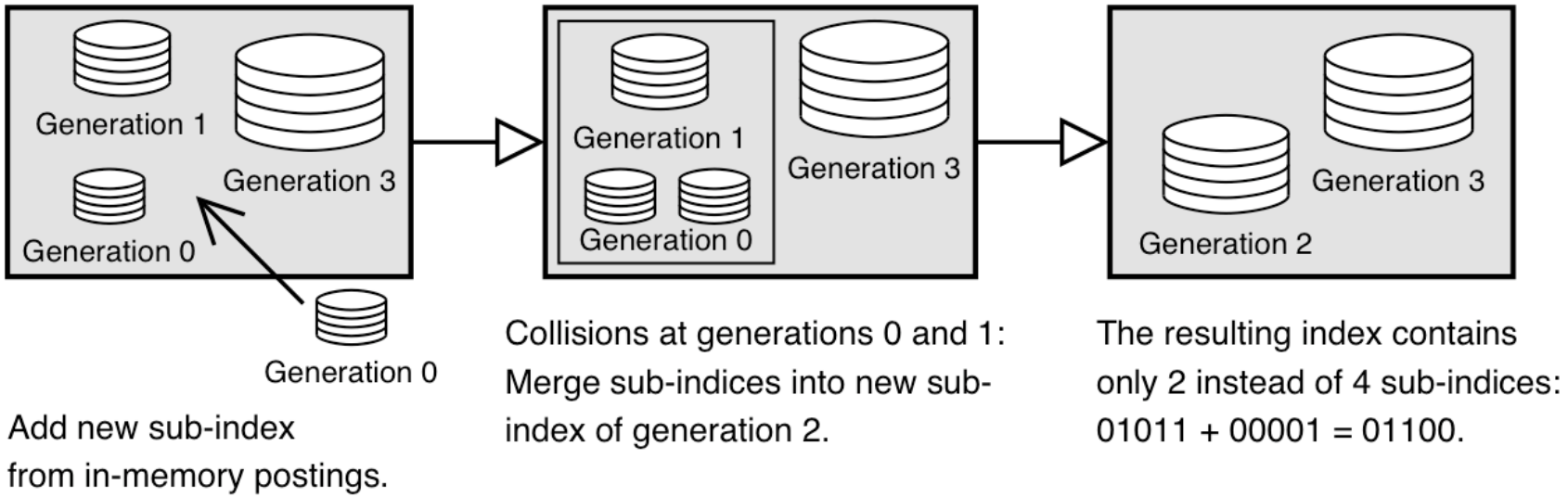


Figure 3: *Logarithmic Merge* by example. A new sub-index is added from in-memory postings. Sub-indices of the same generation n are merged into into a new sub-index of generation $n+1$.

Algorithm

- 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) 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.

```
LMERGEADDTOKEN(indexes, Z0, token)
 1  Z0 ← MERGE(Z0, {token})
 2  if |Z0| = n
 3     then for i ← 0 to ∞
 4          do if Ii ∈ indexes
 5               then Zi+1 ← MERGE(Ii, Zi)
 6                    (Zi+1 is a temporary index on disk.)
 7                    indexes ← indexes − {Ii}
 8               else Ii ← Zi     (Zi becomes the permanent index Ii.)
 9                    indexes ← indexes ∪ {Ii}
10                    BREAK
11          Z0 ← ∅
```

```
LOGARITHMICMERGE()
1  Z0 ← ∅     (Z0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z0, GETNEXTTOKEN())
```

Logarithmic merge

- Auxiliary and main index: index construction time is $O(C^2/M)$ as each posting is touched in each merge.
  - C = Collection Size, and M = memory size
- Logarithmic merge: Each posting is merged $O(\log C/M)$ times, so complexity is $O(C \log (C/M))$
- 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

- Collection-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

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted

Other sorts of indexes

- Positional indexes
  - Same sort of sorting problem … just larger

- Building character n-gram indexes:
  - As text is parsed, enumerate *n*-grams.
  - For each *n*-gram, need pointers to all dictionary terms containing it – the "postings".
  - Note that the same "postings entry" will arise repeatedly in parsing the docs – need efficient hashing to keep track of this.
    - E.g., that the trigram <u>*uou*</u> occurs in the term ***deciduous*** will be discovered on each text occurrence of ***deciduous***
    - Only need to process each term once

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)