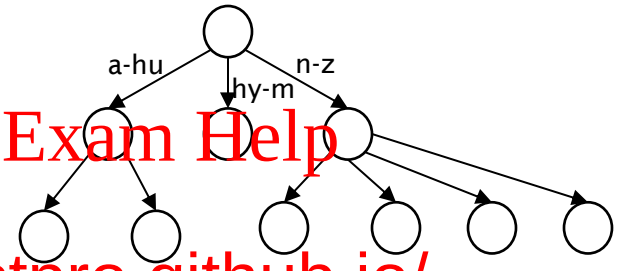
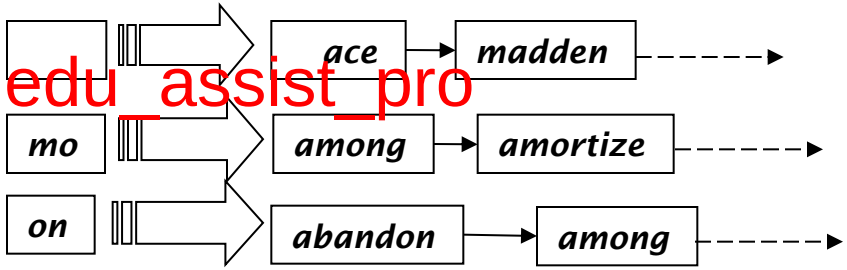
Introduction to Informa I

Lecture 4: Index on

Plan

- Last lecture:
  - Dictionary data structures
  - Tolerant retrieval
    - Wildcards
    - Spell correc
    - 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 b
- Therefore: Transferring on nk of data from disk to memory is faster th rring 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 lar
- Fault tolerance is very expe much cheaper to use many regular machin an one fault tolerant machine.

Hardware assumpt

symbol	statistic	value
s	average seek time	5 ms = 5 x 10^{-3} s
b	transf	2 µs = 2 x 10^{-8} s
	proces	s^{-1}
p	low-level operation (e.g., compare & swap a word)	µs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

RCV1: Our collecti is lecture

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.

- The collection arge enough either, but it' 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 do

Reuters RCV1 stati ounded)

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	ter	400,000
	avg. # bytes per to (incl. spaces/punct.)	
	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
We have 100M items to sort.

	Doc #
	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
	1
	1
	1
	1
	2
	2
	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; cr necessary
- Generally, fas me
- 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 const

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

Sort-based index c tion

- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks

- The final postin plete until the end.
- At 12 bytes per try *(term, doc, freq)*, demands a lot of space fo ctions.

- T = 100,000,000 in the case of R
  - 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 algo r disk?

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

- No: Sorting T on disk is too slow – too ma

- We need an external sorting algo

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 wit[illegible] could be too slow – must sort *T*=100M record

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 few RAM keys)

- 12-byte (4+4+4) records *(term, doc, freq).*
- These are generated as we parse docs.
- Must now sor ecords by *term*.
- Define a Bloc
  - Can easily fit a couple into m
  - Will have 10 such blocks to s
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.

Sorting 10 blocks o ecords

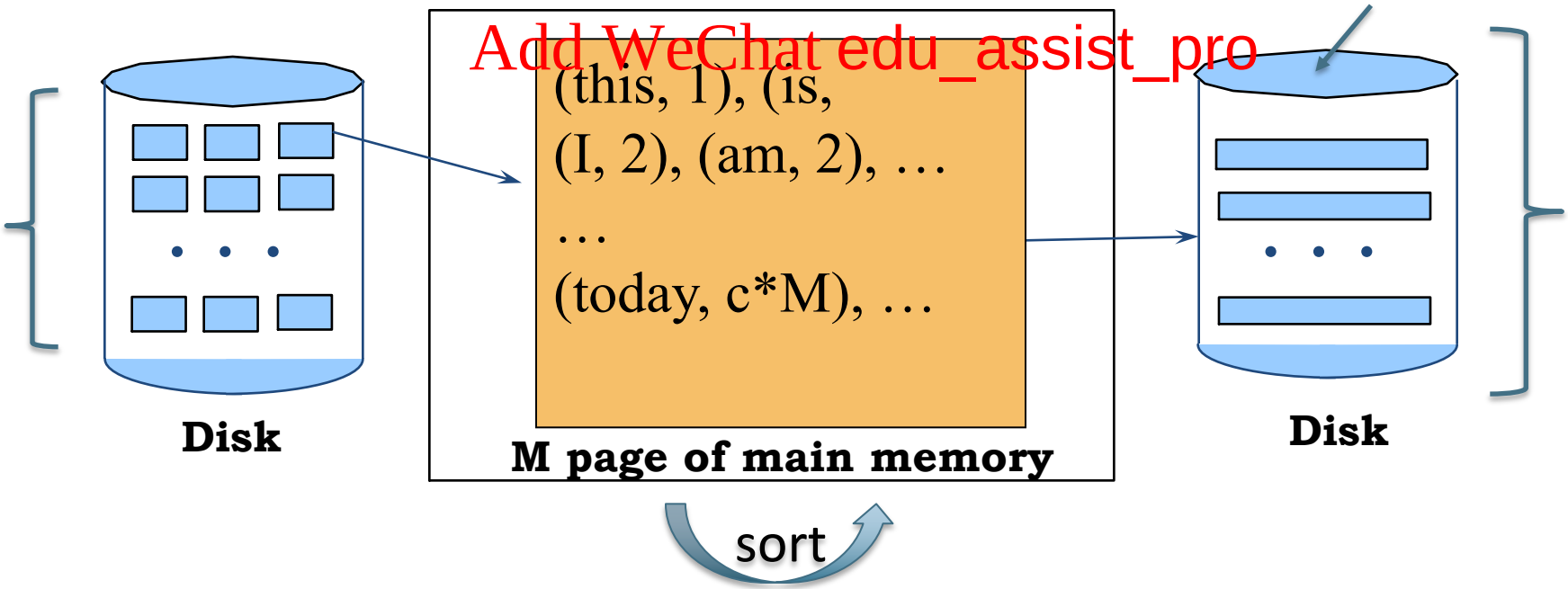
- First, read each block and sort within:
  - Quicksort takes 2*N* ln *N* expected steps
  - In our case 2 × (10M ln 10M) steps
- *Exercise: esti each block from disk and and*
- 10 times this estimate – giv rted *runs* of 10M records each.
- Done straightforwardly, need 2 copies of data on disk
  - But can optimize this

Example

- Settings
  - **B:** Block/page size
  - **M:** Size of m ., = 10 blocks)
  - **N:** Number 00)
  - **R:** Size of the (term, docID) cument 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

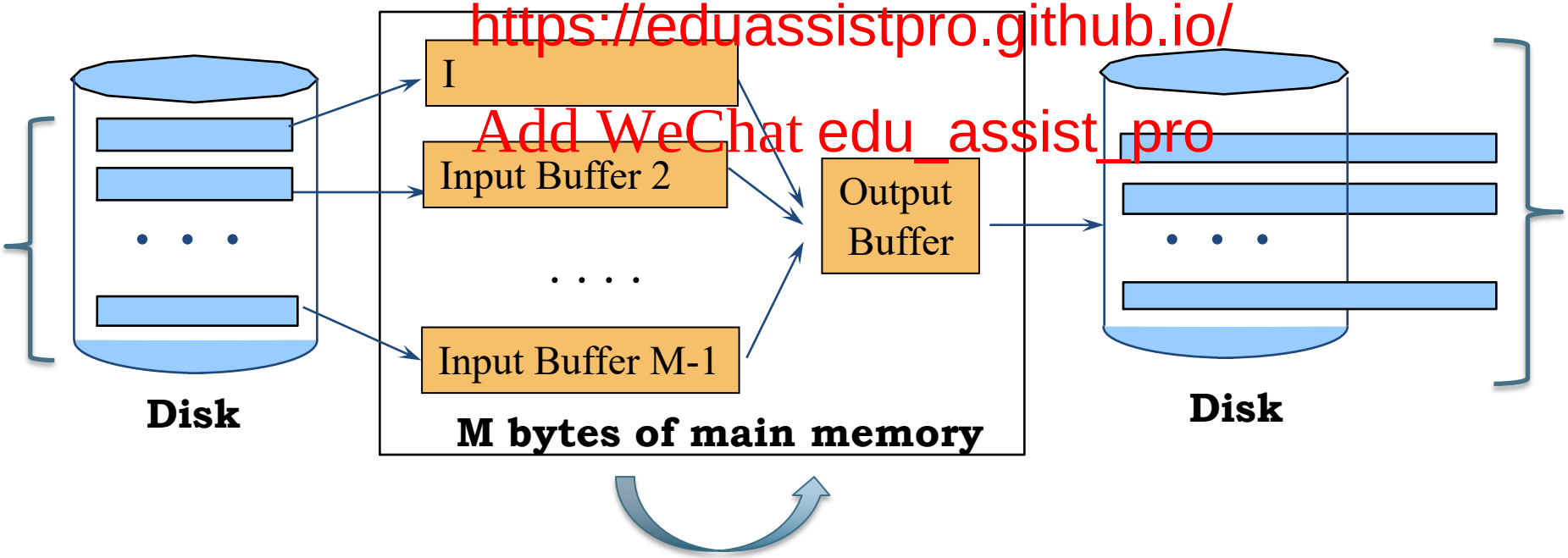
External Merge-Sort

- Phase I: 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 = 2



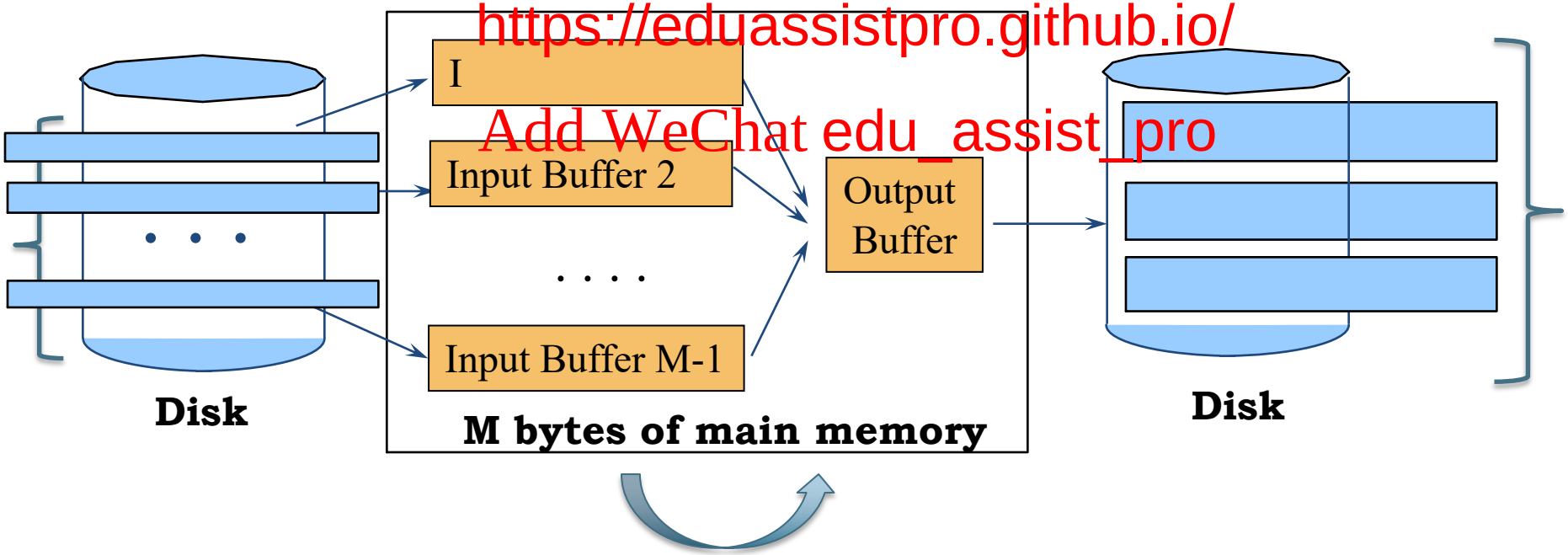
Phase II /1

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



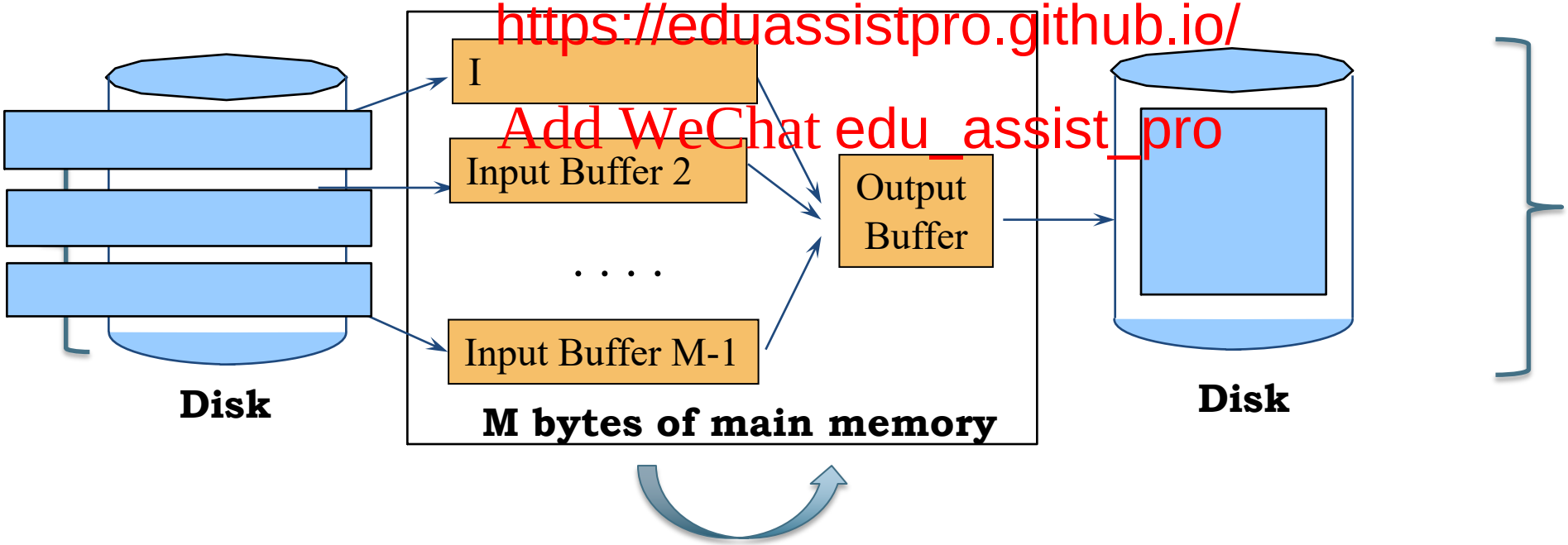
Phase II /2

- **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



(M-1)-way Merge

Cost of External M rt

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

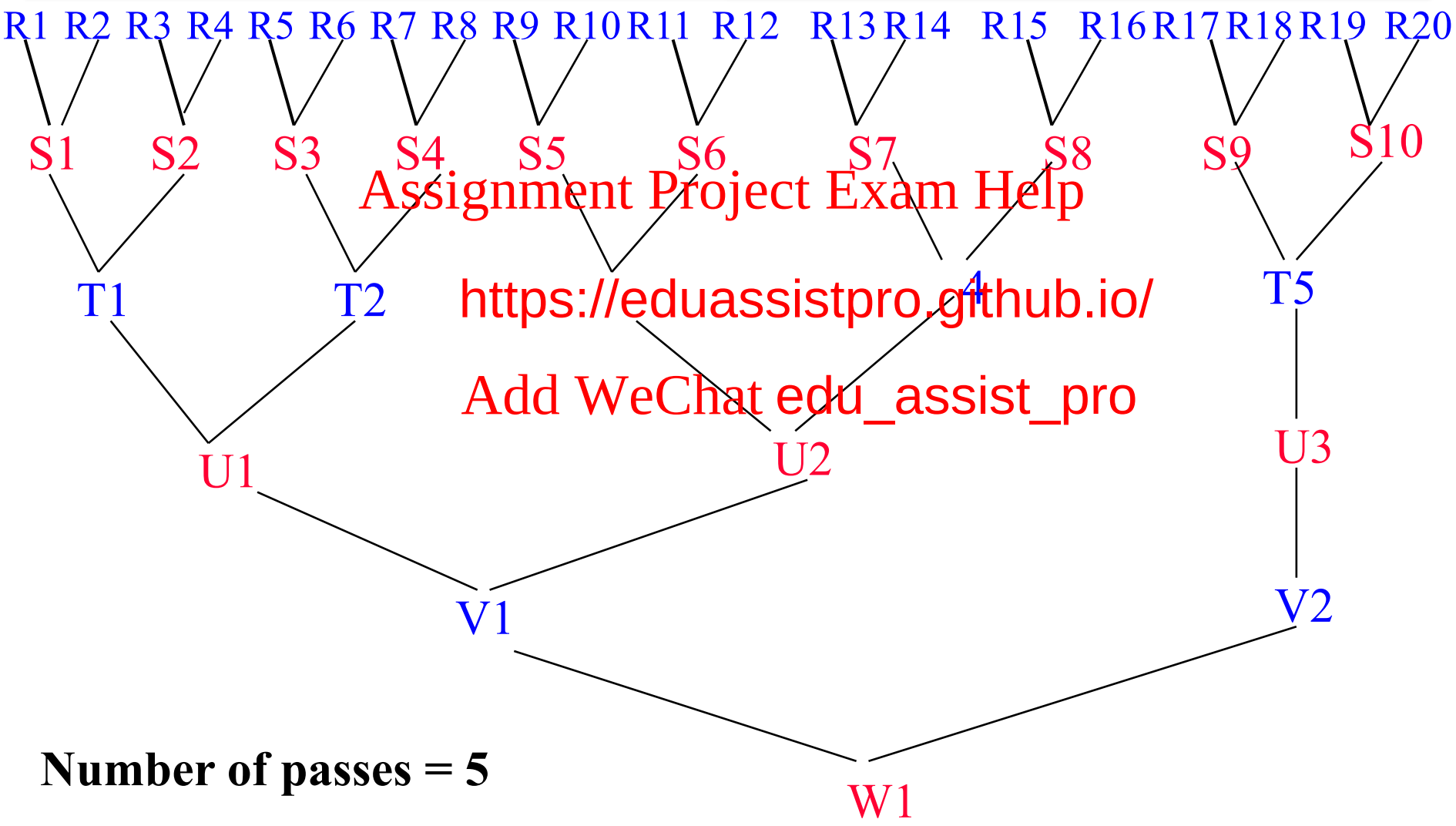
Total I/O cost: $2 \left(\frac{NR}{B}\right) \cdot \left(1 + \left\lceil \log_{M-1} \left\lceil \frac{NR}{MB} \right\rceil \right\rceil\right)$ blocks/pages

- How much data can we sort B 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 !

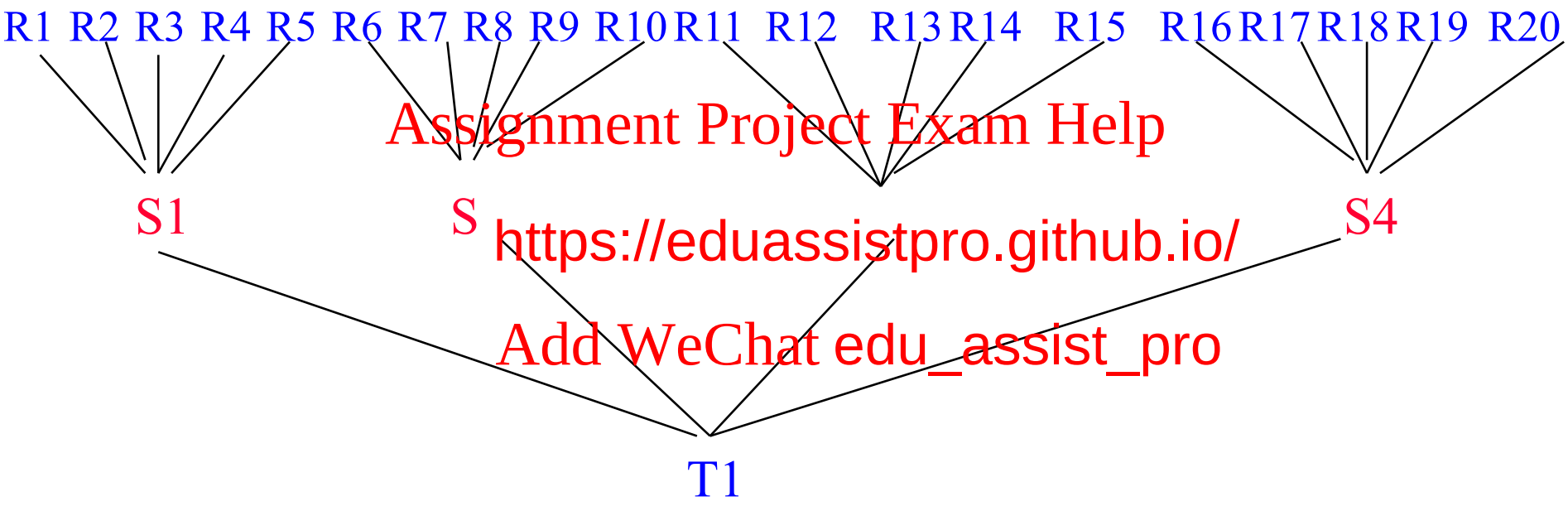
Another Example

Example: 2-Way M r 20 Runs



Number of passes = 5

Example: 5-Way Merge of 20 Runs



Number of passes = 2

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

K-way Merge: using a

99

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 impl D mapping.
- Actually, we could work wi cID postings instead of termID,docID po
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

SPIMI: Single-pass in-memo ng

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.

- Key idea 2: D ostings in postings lists

- With these two ideas we ca a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

- Merging of blocks is analogous to BSBI.

SPIMI: Compressio

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

Distributed indexin

- For web-scale indexing (don't try this at home!):

  must use a distributed computing cluster

- Individual machines are fault-prone
  - Can unpredic

- How do we exploit such a p hines?

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 c d to be inserted.

  - Documents are deleted and

- This means that the diction stings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

Simplest approach – date 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
- 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 a lary indexes

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the dex is efficient if we keep a separat
  - Merge is the same as a simple app
  - But then we would need a lot of fil t 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 – 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 Ω(|C|/M) seeks to fetch the inverted list for a term

Compromise – Log c_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 mu generation = g gives a new index with generation
- 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

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.

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 merged
  O(log C/M) ti (C log (C/M))
- So logarithmic merge is mu fficient 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

Lucene

- `Mergefactor` does not have to be 2
  - http://lucene.sourceforge.net/talks/inktomi/
- In animations:
  - http://blog.m 02/visualizing-lucenes-seg

Further issues with le 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 t
- How do we maintain the to h 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 in 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 inde

- Positional indexes
  - Same sort of sorting problem … just larger
- Building character n-gram indexes:
  - As text is par
  - For each *n*-gram, need point tionary 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

Why?

Resources for today's lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publi Dean and Ghemawat (2
- Original publication on SPI nd Zobel (2003)