Index construction

CE-324: Modern Information Retrieval

Sharif University of Technology

M. Soleymani

Fall 2018

Most slides have been adapted from: Profs. Manning, Nayak & Raghavan (CS-276, Stanford)

Some slides have been adapted from Mining Massive Datasets Course: Prof. Leskovec (CS-246, Stanford)

Outline

- Scalable index construction
  - BSBI
  - SPIMI
- Distributed indexing
  - MapReduce
- Dynamic indexing

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 memory is ***much*** faster than access to 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 tens of GB of main memory.
- Available disk space is several (2–3) orders of magnitude larger.

Hardware assumptions for this lecture

statistic	**value**
saverage **seek** time	5 ms = 5×10^{-3} s
transfer time per byte	0.02 μs = 2×10^{-8} s
processor's **clock** rate	10^9 per s
low-level **operation** (e.g., compare & swap a word)	0.01 μs = 10^{-8}

2007 Hardware

Recall: index construction

- Docs are parsed to extract words and these are saved with the Doc 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

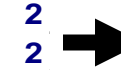
Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Recall: index construction (key step)

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

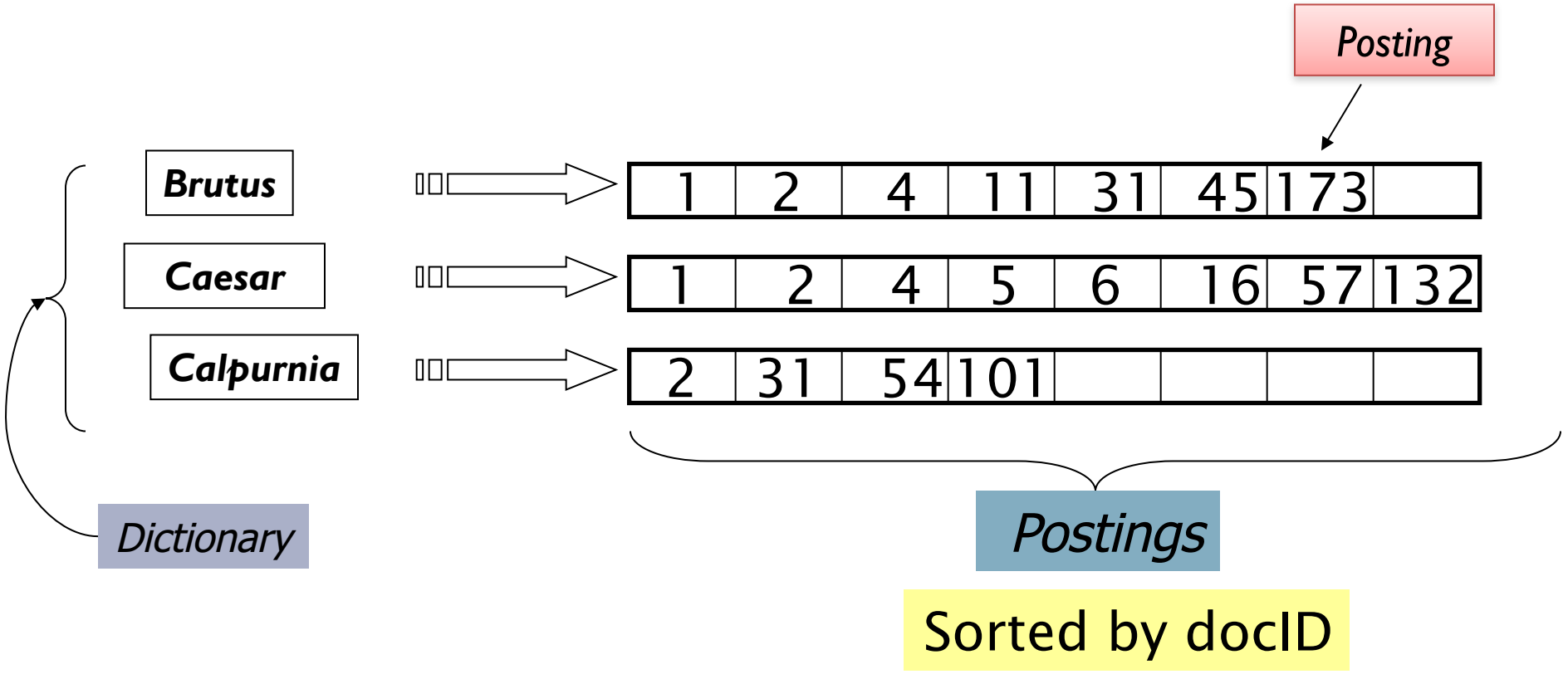
We focus on this sort step.
We have 100M items to sort.

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

Recall: Inverted index



Scaling index construction

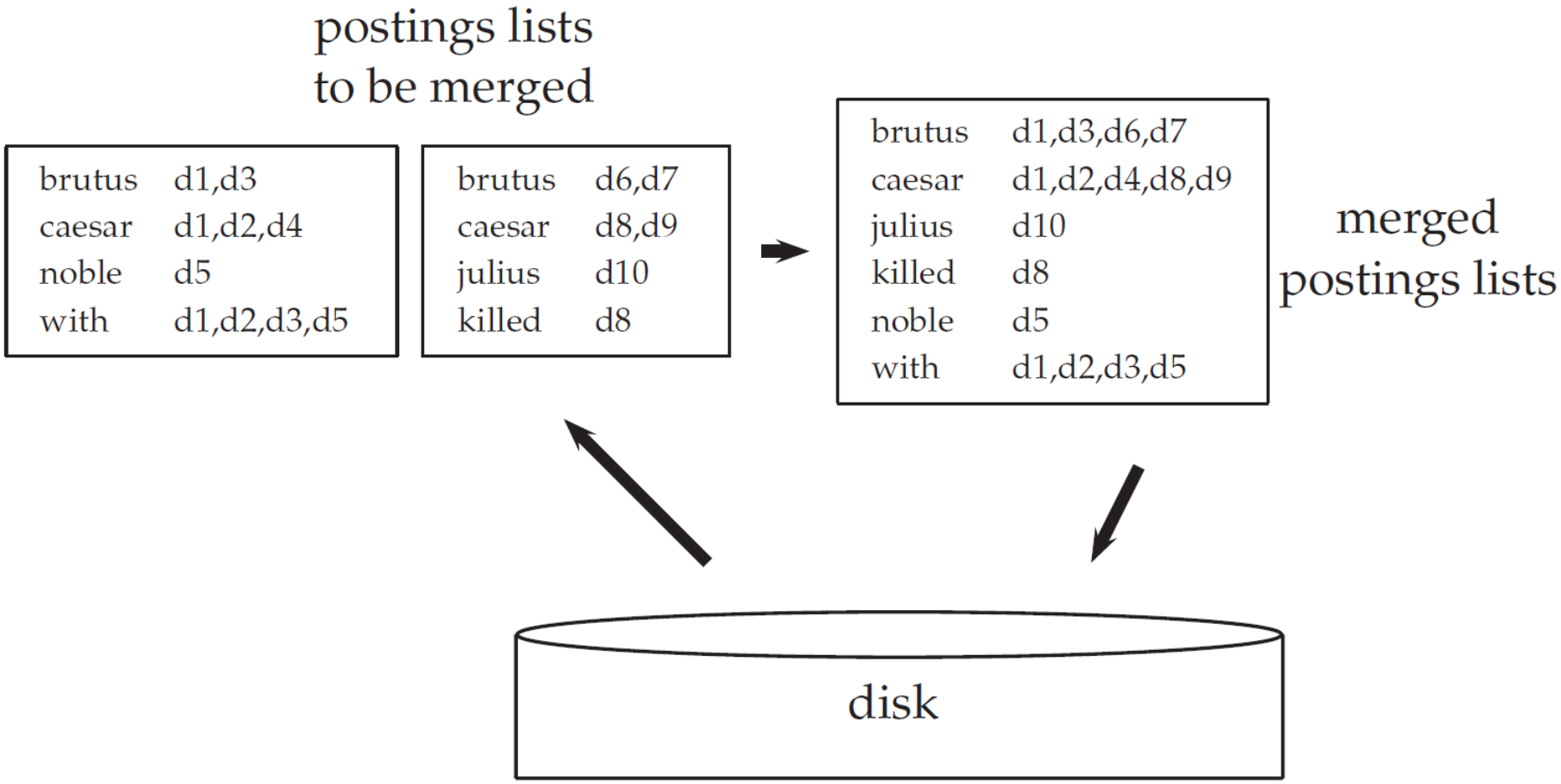
- In-memory index construction does not scale
  - Can't stuff entire collection into memory, sort, then write back
- Indexing for very large collections
- Taking into account the hardware constraints we just learned about . . .
  - We need to store intermediate results on disk.

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.
      - Doing this with random disk seeks would be too slow
      - If every comparison needs two disk seeks, we need $O(T \log T)$ disk seeks
- We need an *external* sorting algorithm.

BSBI: Blocked Sort-Based Indexing (Sorting with fewer disk seeks)

- Basic idea of algorithm:
  - Segments the collection into blocks (parts of nearly equal size)
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.


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
disk

BSBI

- Must now sort T of such records by *term*.
- Define a Block of such records
  - Can easily fit a couple into memory.
- First read each block and sort it and then write it to the disk
- Finally merge the sorted blocks

```
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)
```

BSBI: terms to termIDs

- It is wasteful to use *(term, docID)* pairs
  - Term must be saved for each pair individually
- Instead, it uses *(termID, docID)* and thus needs a data structure for mapping terms to termIDs
  - This data structure must be in the main memory
- *(termID, docID)* are generated as we parse docs.
  - 4+4=8 bytes records

Reuters RCV1 statistics

symbol	statistic	value
N	# documents	800,000
L_{ave}	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

4.5 bytes per word token vs. 7.5 bytes per word type: why?

RCV1 Collection

- This is one year of Reuters newswire (part of 1995 and 1996)

REUTERS

You are here: Home > News > Science > Article

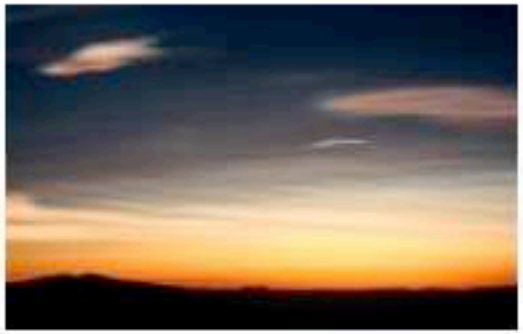
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

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.

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.
- 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 "keeping 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.
  - If we use terms themselves in this method, 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
  - Term is saved one time (in a block) for the whole of its posting list (not one time for each of the docIDs containing it)
- **Key idea 2**: Accumulate (and implicitly sort) 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.
  - Merging of blocks is analogous to BSBI.
  - No need to maintain term-termID mapping across blocks

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
```

SPIMI: $O(T)$

- Sort terms before writing to disk
  - Write posting lists in the lexicographic order to facilitate the final merging step

SPIMI properties

- Scalable: SPIMI can index collection of any size (when having enough disk space)
  - It is more efficient than BSBI since it does not allocate a memory to maintain term-termID mapping

- Some memory is wasted in the posting list (variable size array structure) which counteracts the memory savings from the omission of termIDs.
  - During the index construction, it is not required to store a separate termID for each posting (as opposed to BSBI)

Distributed indexing

- For web-scale indexing must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- Fault tolerance is very expensive
  - It's much cheaper to use many regular machines rather than one fault tolerant machine.
- How do we exploit such a pool of machines?

Google Example

- 20+ billion web pages x 20KB = 400+ TB
  - 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
  - ~1,000 hard drives to store the web
    Takes even more to **do** something useful with the data!

- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
    Commodity network (ethernet) to connect them

Large-scale challenges

- How do you distribute computation?
  - How can we make it easy to write distributed programs?
- Machines fail:
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to loose 1/day
  - People estimated Google had ~1M machines in 2011
    - 1,000 machines fail every day!

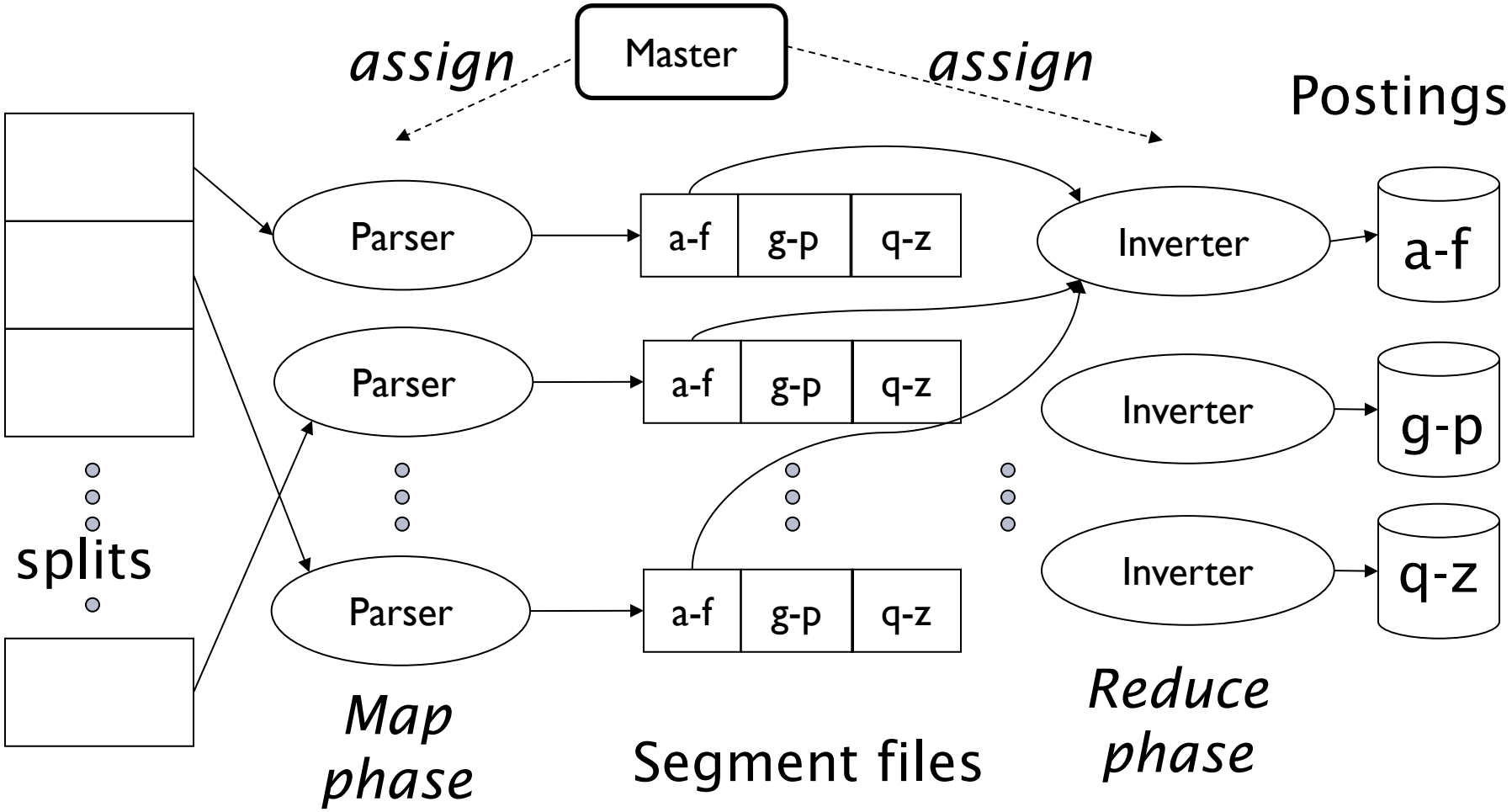
Distributed indexing

- Maintain a *master* machine directing the indexing job – considered "safe".
  - To provide a fault-tolerant system massive data center
  - Stores metadata about where files are stored
  - Might be replicated
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into *splits*
- Each split is a subset of docs (corresponding to blocks in BSBI/SPIMI)

Data flow


assign
Master
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Segment files
Reduce phase

Parsers

- Master assigns a split to an idle parser machine
- Parser reads a doc at a time and emits (term, doc) pairs and writes pairs into *j* partitions
- Each partition is for a range of terms
  - Example: *j* = 3 partitions ***a-f***, ***g-p***, ***q-z*** terms' first letters.

Inverters

- An inverter collects all (term,doc) pairs for one term-partition.
- Sorts and writes to postings lists

Map-reduce

- **Challenges:**
  How to distribute computation?
  Distributed/parallel programming is hard
- **Map-reduce** addresses all of the above
  Google's computational/data manipulation model
  - Elegant way to work with big data

MapReduce

- The index construction algorithm we just described is an instance of *MapReduce*.
- MapReduce (Dean and Ghemawat 2004): a robust and conceptually simple framework for distributed computing
  - without having to write code for the distribution part.
- Google indexing system (ca. 2002): a number of phases, each implemented in MapReduce.

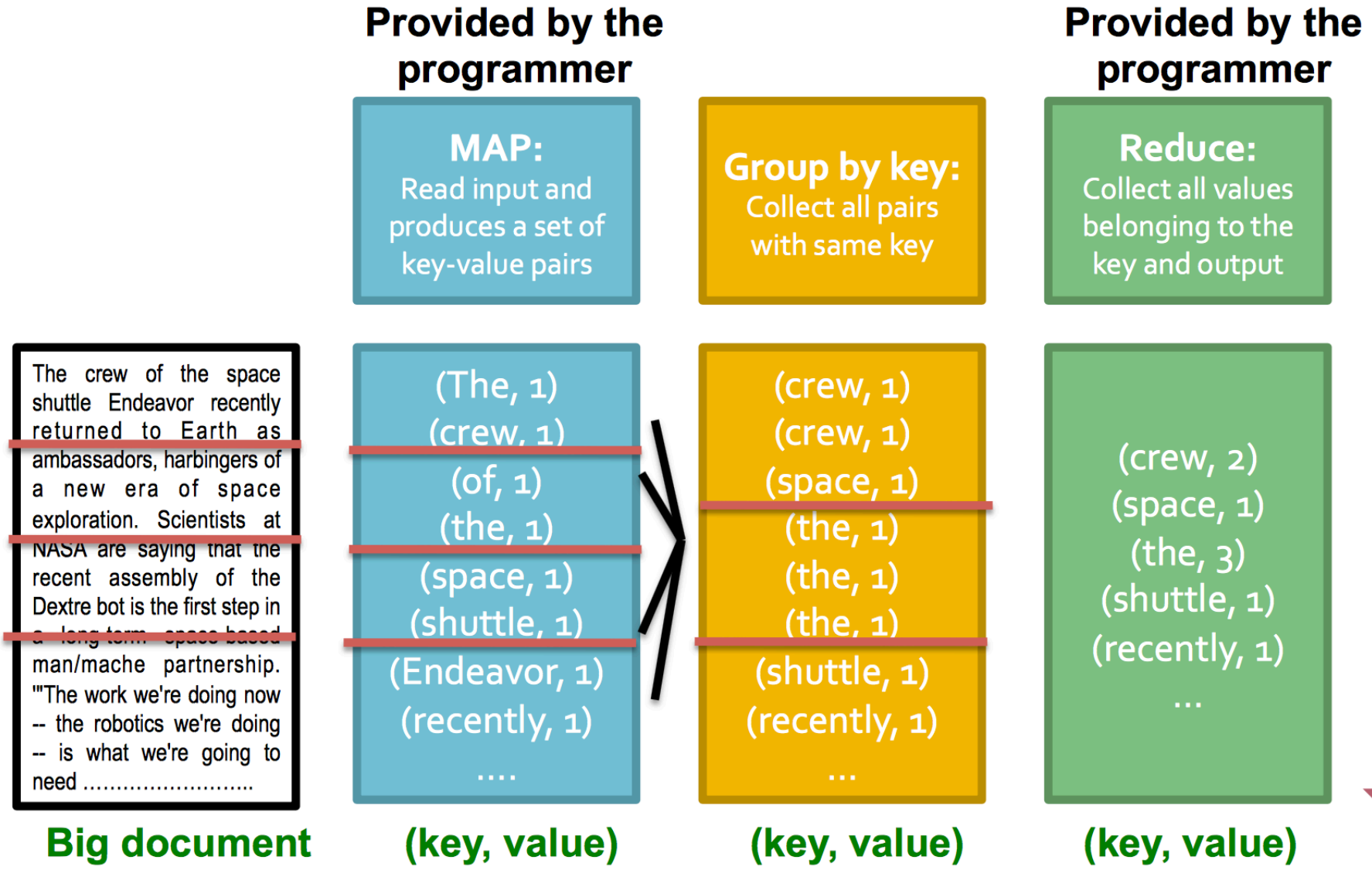
Schema of map and reduce functions

- map: (k,v) → list(key, value)
- reduce: (key,list(value)) → (key2,list(value2))

Word-count example

```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

```
reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```



J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Index construction in MapReduce

- **Schema of map and reduce functions**
  - map: (k,v) → list(key, value)
  - reduce: (key,list(value)) → (key2,list(value2))

- Example:
  - Map:
    - d1 : C came, C eat.
    - d2 : C died.
    - → <C,d1>, <came,d1>, <C,d1>, <eat,d1>, <C,d2>, <died,d2>
  - Reduce:
    - (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <eat,(d1)>) → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <eat,(d1:1)>)

Map-Reduce: Overview

- Sequentially read a lot of data
- **Map:** Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:** Aggregate, summarize, filter or transform
- Write the result

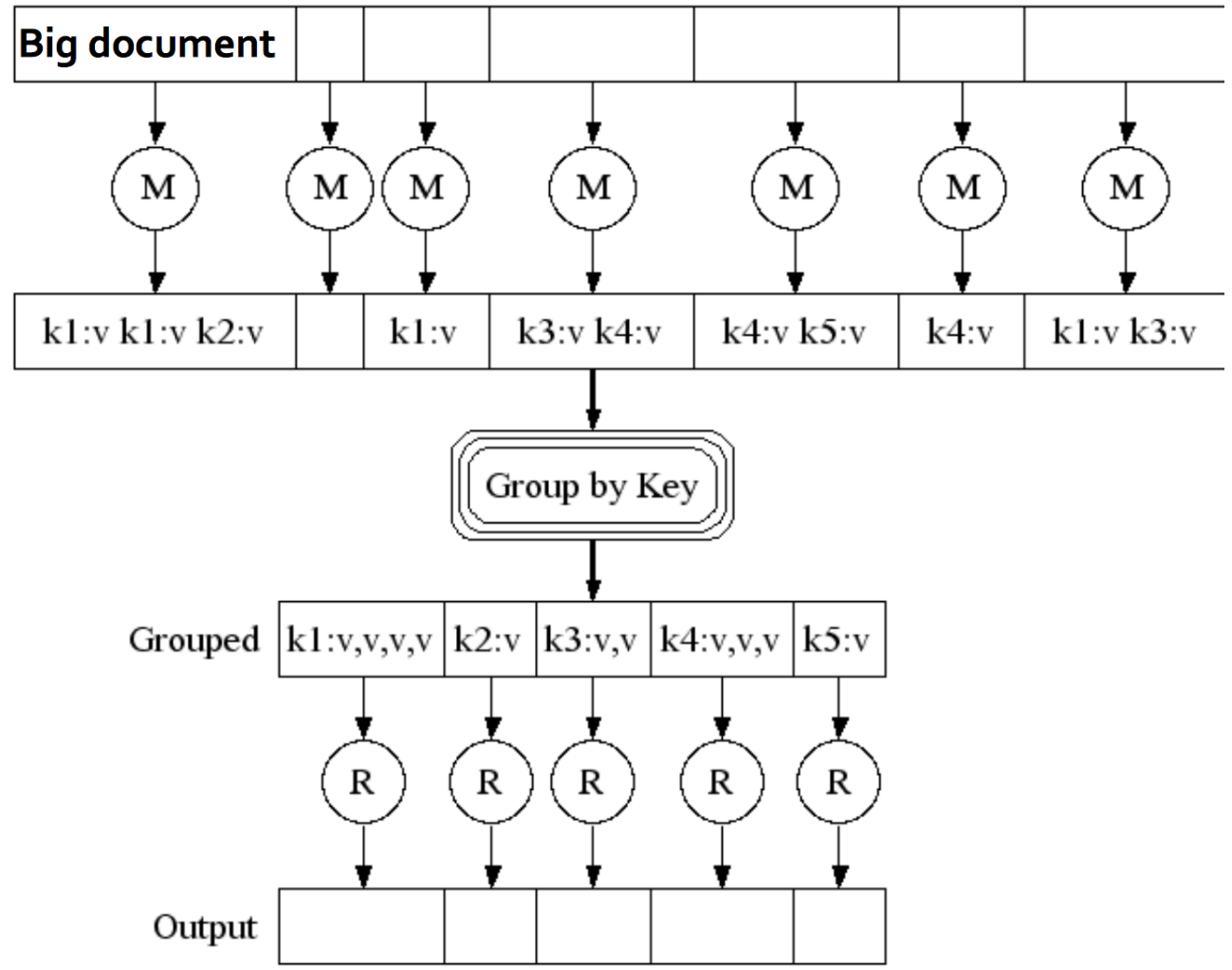
Map-Reduce Environment

- Map-Reduce environment takes care of:
  - Partitioning the input data
  - Scheduling the program's execution across a set of machines Performing the group by key step
  - Handling machine failures
  - Managing required inter-machine communication

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

How to distribute indexing?

- *Term-partitioned:* one machine handles a subrange of terms
- *Document-partitioned:* one machine handles a subrange of docs

Term-partitioned vs. doc-partitioned

	Term-partitioned	Doc partitioned
Load balancing	✗	✓
Scalability	✗	✓
Disk seek	✓	✗
Dynamic	✗	✓

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Docs come in over time and need to be inserted.
  - Docs 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

- Maintain "big" **main index**
- New docs go into "small" **auxiliary index**
- 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

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News, blogs, new topical web pages
- 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 deleted

Resources

- Chapter 4 of IIR
- Mining Massive Datasets, Chapter 2
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)