

Introduction to
Information Retrieval

Basic inverted index construction

Index construction

- How do we construct an index?
 - We call this process *index construction* or *indexing*
 - the process or machine that performs it the *indexer*
- The design of indexing algorithms is governed by hardware constraints
- What strategies can we use with limited main memory?

Recall 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.

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

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2



We focus on this sort step.

RCV1: Our collection for this lecture

- 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)
- The collection isn't really large enough, but it's publicly available and is a plausible example.

A Reuters RCV1 document



You are here: Home > News > Science > Article

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

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) | [Print This Article](#) | [Reprints](#)



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.

[+] Text [-]

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

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

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

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 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 - When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located. This time is called the *seek time* and it averages 5 ms for typical disks.
- 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

- Data transfers from disk to memory are handled by the system bus, not by the processor. This means that the processor is available to process data during disk I/O. We can exploit this fact to speed up data transfers.
- Servers used in IR systems typically have several gigabytes (GB) of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger

Hardware assumptions

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

Sort-based index construction

- The basic steps in constructing a nonpositional index are
 - We first make a pass through the collection assembling all term–docID pairs.
 - We then sort the pairs with the term as the dominant key and docID as the secondary key.
 - initially, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency.
- For small collections, all this can be done in memory.
- At 8 bytes per *(termID, docID)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
 - So ... we can do this in memory today, 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 for large collections

Term-id

- To make index construction more efficient, we represent terms as termIDs, where each *termID* is a unique serial number.
- We can build the mapping from terms to termIDs on the fly while we are processing the collection; or,
- In a two-pass approach, we compile the vocabulary in the first pass and construct the inverted index in the second pass.
- The index construction algorithms described in this chapter all do a single pass through the data.

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 hardware constraints. . .
 - Memory, disk, speed, etc.

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.
- With main memory insufficient, we need to use an *external sorting algorithm*, that is, one that uses disk.

Exercise

- If we need $T \log_2 T$ comparisons (where T is the number of termID–docID pairs) and two disk seeks for each comparison, how much time would index construction for Reuters-RCV1 take if we used disk instead of memory for storage and an unoptimized sorting algorithm? Use the system parameters in Table 4.1.

Introduction to
Information Retrieval

CS276: Information Retrieval and Web Search

External memory indexing

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

- 8-byte records ($termID, docID$)
- These are generated as we parse docs
- Must now sort 100M such 8-byte records by $termID$
- Define a Block $\sim 10M$ 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

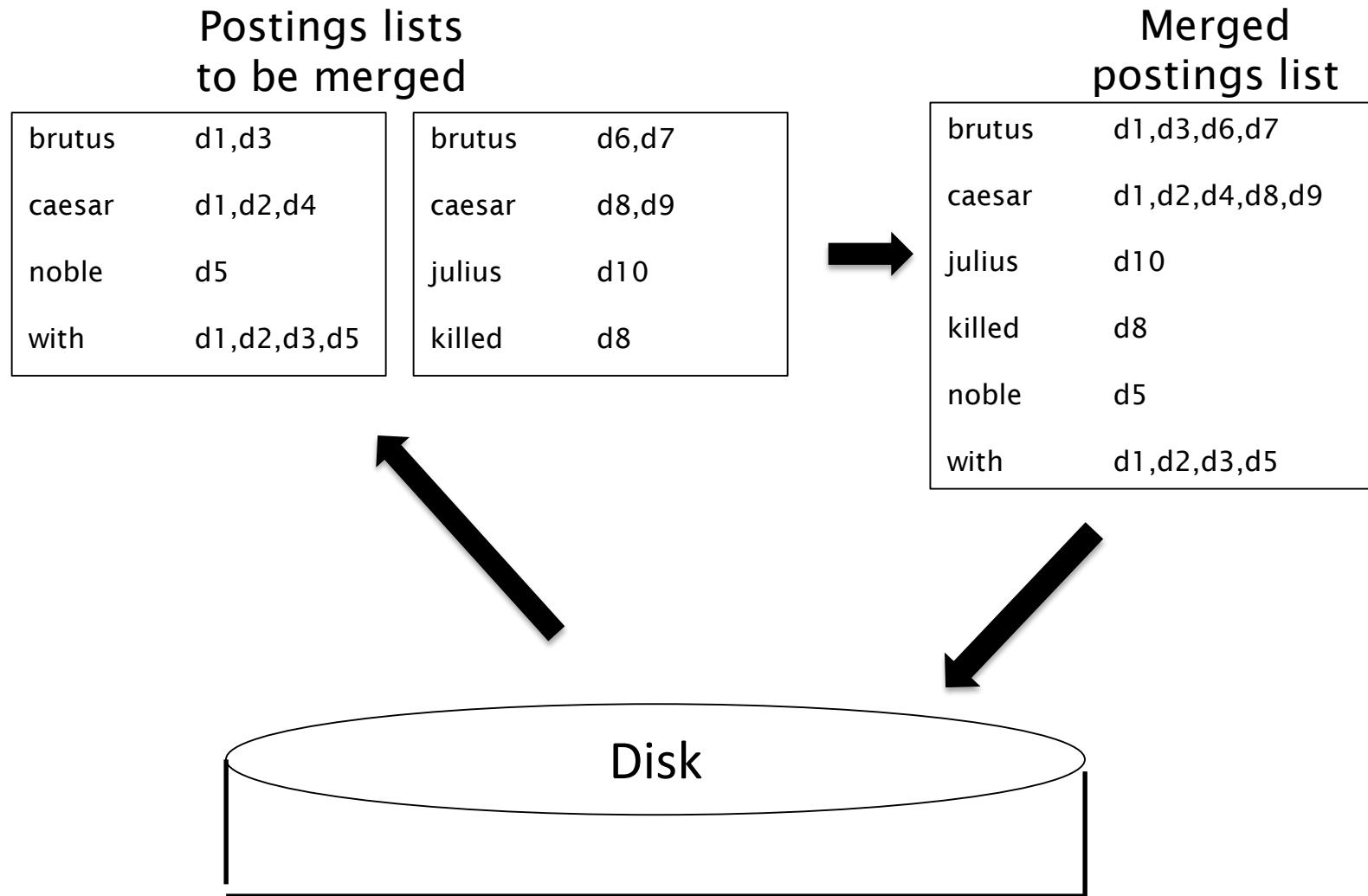
BSBINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4       $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5      BSBI-INVERT( $block$ )
6      WRITEBLOCKTODISK( $block, f_n$ )
7      MERGEBLOCKS( $f_1, \dots, f_n; f_{\text{merged}}$ )
```

One solution is the *blocked sort-based indexing algorithm* or *BSBI*. BSBI (i) segments the collection into parts of equal size, (ii) sorts the termID–docID pairs of each part in memory, (iii) stores intermediate sorted results on disk, and (iv) merges all intermediate results into the final index.

- The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full.
- The block is then inverted
 - First, we sort the termID–docID pairs.
 - Next, we collect all termID–docID pairs with the same termID into a postings list, where a *posting* is simply a docID.
- The block is then written to disk

How to merge the sorted runs?



How to merge the sorted runs?

- It is efficient to do a multi-way merge, where you are reading from all blocks simultaneously
 - Open all block files simultaneously and maintain a read buffer for each one and a write buffer for the output file
 - In each iteration, pick the lowest termID that hasn't been processed using a priority queue
 - Merge all postings lists for that termID and write it out
- 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.
- Blocked sort-based indexing has excellent scaling properties, but it needs a data structure for mapping terms to termIDs.
- For very large collections, this data structure does not fit into memory.

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  $\leftarrow$  next(token_stream)
5    if term(token)  $\notin$  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  $\leftarrow$  SORTTERMS(dictionary)
12       WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13   return output_file
```

- Merging of blocks is analogous to BSBI.
- The part of the algorithm that parses documents and turns them into a stream of term–docID pairs, which we call *tokens* here, has been omitted.

SPIMI in action

Input token

Caesar d1

with d1

Brutus d1

Caesar d2

with d2

Brutus d3

with d3

Caesar d4

noble d5

with d5

Dictionary

brutus d1 d3

with d1 d2 d3 d5

noble d5

caesar d1 d2 d4

Sorted
dictionary

brutus d1 d3

caesar d1 d2 d4

noble d5

with d1 d2 d3 d5

Difference between BSBI and SPIMI

- SPIMI adds a posting directly to its postings list
- Instead of first collecting all termID–docID pairs and then sorting them(as we did in BSBI), each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings.
 - It is faster because there is no sorting required, and
 - It saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.
- In addition to constructing a new dictionary structure for each block and eliminating the expensive sorting step, SPIMI has a third important component: compression. Both the postings and the dictionary terms can be stored compactly on disk if we employ compression.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
 - Compression of terms
 - Compression of postings
- More on this later ...

Original publication on SPIMI: Heinz and Zobel (2003)

Introduction to
Information Retrieval

CS276: Information Retrieval and Web Search

Distributed indexing

Distributed indexing

- For web-scale indexing :
 - 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?

Web search engine data centers

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)
- Web search engines use *distributed indexing* algorithms for index construction.
- The result of the construction process is a distributed index that is partitioned across several machines – either **according to term** or according to document.

Massive data centers

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the entire system?
- Answer: 37% - meaning, 63% of the time one or more servers is down.
- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.

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 – Map Reduce

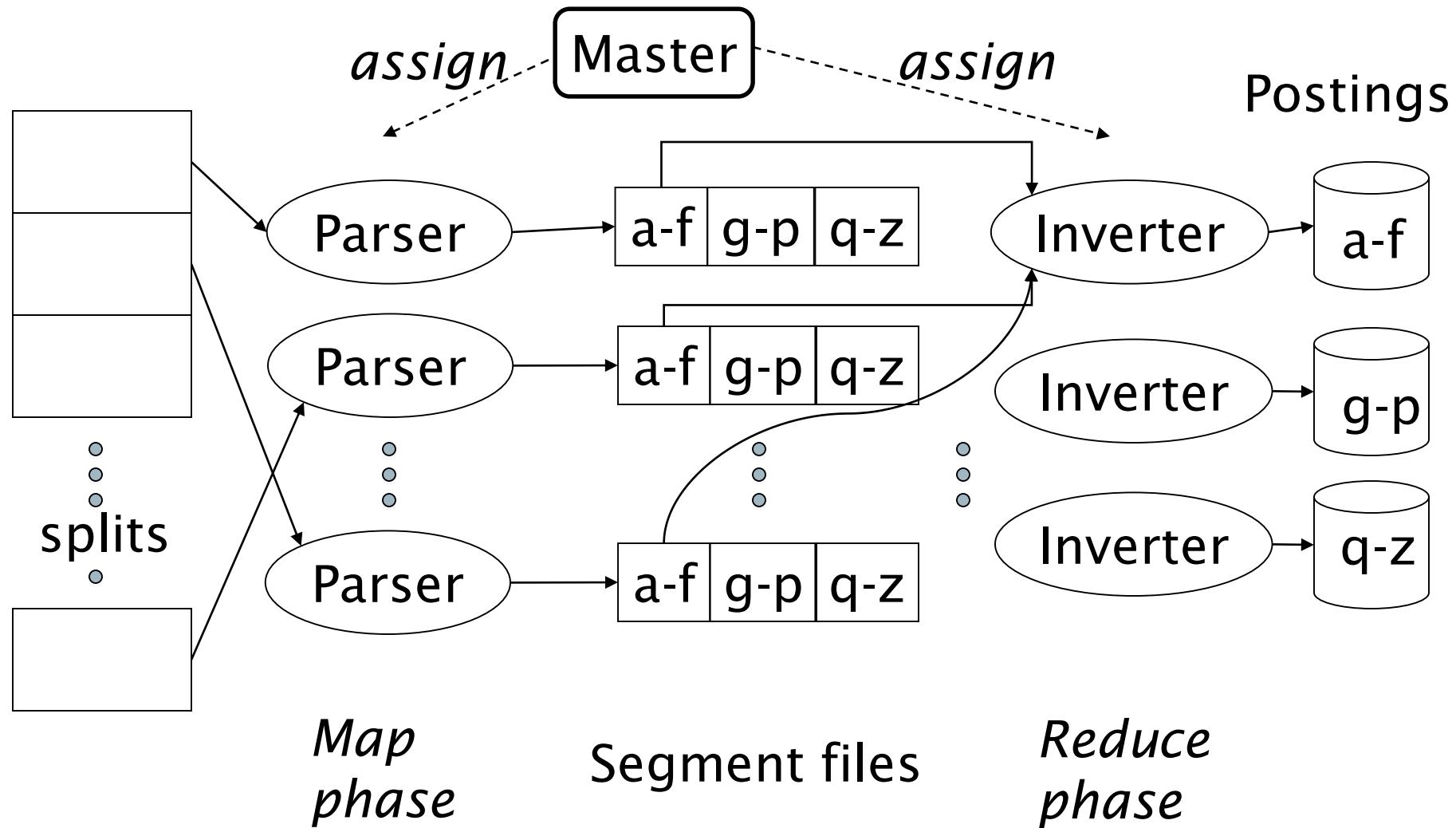
- We will use two sets of parallel tasks
 - Parsers
 - Inverters
- Break the input document collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)
- Splits are not preassigned to machines,
- *MapReduce*, is a general architecture for distributed computing.
 - MapReduce is designed for large computer clusters.
 - Use to solve large computing problems on cheap commodity machines or *nodes* that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware.
 - Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time.
 - One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and – in case of failure – reassign.

-
- A *master node* directs the process of assigning and reassigning tasks to individual worker nodes.
 - As a machine finishes processing one split, it is assigned the next one.
 - If a machine dies or fails due to hardware problems, the split it is working on is simply reassigned to another machine.
 - MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of *key-value pairs*.
 - For indexing, a key-value pair has the form (termID,docID).
 - In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing.
 - A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms.

Parsers

- A *master node* directs the process of assigning and reassigning tasks to individual worker nodes.
- Master assigns a split to an idle parser machine
- As a machine finishes processing one split, it is assigned the next one.
- If a machine dies or fails due to hardware problems, the split it is working on is simply reassigned to another machine.
- The *map phase* of MapReduce consists of mapping splits of the input data to key-value pairs.
- Parser reads a document at a time and generates (term, doc) pairs and writes its output to local intermediate files, the *segment files*
- Parser writes pairs into j partitions
- Example: Each partition is for a range of terms' first letters
 - (e.g., **a-f**, **g-p**, **q-z**) – here $j = 3$.
- Now to complete the index inversion

Data flow



Inverters

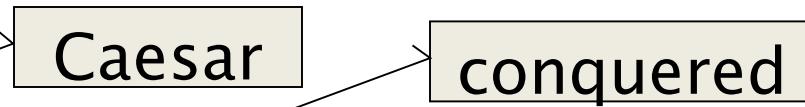
- Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the *inverters* in the reduce phase.
- The master assigns each term partition to a different inverter
- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Each term partition (corresponding to r segment files, one on each parser) is processed by one inverter.
- Finally, the list of values is sorted for each key and written to the final sorted postings list.
- Parsers and inverters are not separate sets of machines.
- The master identifies idle machines and assigns tasks to them.
- The same machine can be a parser in the map phase and an inverter in the reduce phase.
- And there are often other jobs that run in parallel with index construction, so in between being a parser and an inverter a machine might do some crawling or another unrelated task.

Optimization

- To minimize write times before inverters reduce the data, each parser writes its segment files to its *local disk*.
- In the reduce phase, the master communicates to an inverter the locations of the relevant segment files .
- Each segment file only requires one sequential read because all data relevant to a particular inverter were written to a single segment file by the parser.
- This setup minimizes the amount of network traffic needed during indexing.

Example for index construction

Map:



d1 : C came, C c' ed.

d2 : C died.

→

<C,d1>, <came,d1>, <C,d1>, <c' ed, d1>, <C, d2>, <died,d2>

Reduce:

(<C,(d1,d1,d2)>, <died,(d2)>, <came,(d1)>, <c' ed,(d1)>)

→

(<C,(d1:2,d2:1)><died,(d2:1)>, <came,(d1:1)>,<c' ed,(d1:1)>)

Index construction

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
 - *Term-partitioned*: one machine handles a subrange of terms
 - *Document-partitioned*: one machine handles a subrange of documents
- As we'll discuss in the web part of the course, most search engines use a document-partitioned index ... better load balancing, etc.

Schema for index construction in MapReduce

- **Schema of map and reduce functions**
- map: input \rightarrow list(k, v) reduce: (k, list(v)) \rightarrow output
- **Instantiation of the schema for index construction**
- map: collection \rightarrow list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, ...) \rightarrow (postings list1, postings list2, ...)

- MapReduce offers a robust and conceptually simple framework for implementing index construction in a distributed environment.

Introduction to **Information Retrieval**

CS276: Information Retrieval and Web Search
Dynamic indexing

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

- 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
- 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 OS.
- 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.)

Further issues with multiple indexes

- Collection-wide statistics are hard to maintain
- E.g., when we speak of spell-correction: which of several corrected alternatives do we present to the user?
 - We may want to 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
- 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

Earlybird: Real-time search at Twitter

- Requirements for real-time search
 - Low latency, high throughput query evaluation
 - High ingestion rate and immediate data availability
 - Concurrent reads and writes of the index