Lecture 6
part 2
Index Construction and Search

Index Implementation

- Bag of words
- Inverted files
- Signature files
- Hashing
- …

Inverted Files

- Each document is assigned a list of keywords or attributes.
- Each keyword (attribute) is associated with operational relevance weights.
- An inverted file is the sorted list of keywords (attributes), with each keyword having links to the documents containing that keyword.

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.

Text

Inverted Index

Vocabulary	Occurrences
letters	60 …
made	50 …
many	28 …
text	11, 19 …
words	30, 40 …

addressing granularity:

(1) inverted list –
    word positions
    character positions

(2) inverted file –
    document

Vocabulary space: the vocabulary grows as $O(n^{\beta})$, β: 0.4~0.6

Vocabulary for 1GB of TREC-2 collection: 5MB

(before stemming and normalization)

Occurrences: the extra space $O(n)$

30% ~ 40% of the text size

Block Addressing

- Full inverted indices
  - Point to exact occurrences
- Blocking addressing
  - Point to the blocks where the word appears
  - Pointers are smaller
  - 5% overhead over the text size

block:
fixed size blocks,
files, documents,
Web pages, …

Block1	Block2	Block3	Block 4
This is a text.	A text has many	words. Words are	made from letters.

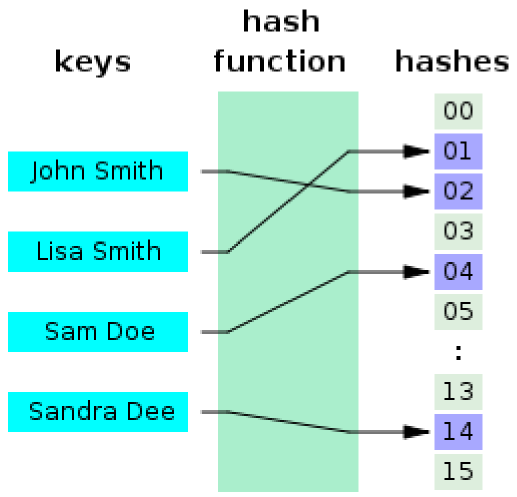
Text

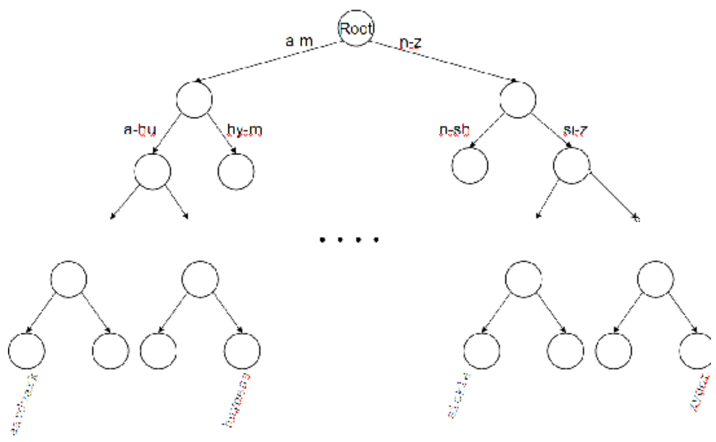
Vocabulary	Occurrences
letters	4 …
made	4 …
many	2 …
text	1, 2 …
words	3 …

Inverted index

Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees

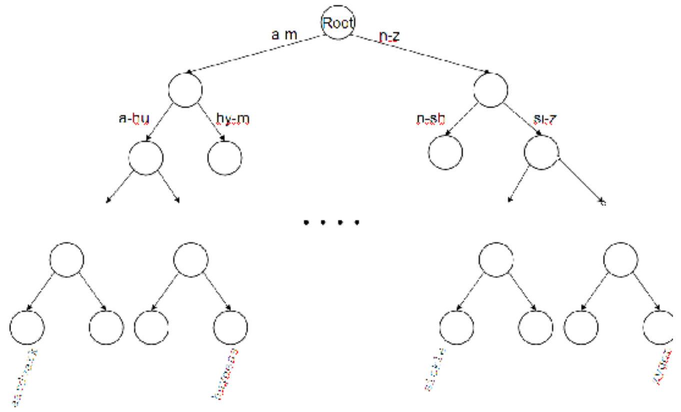




Hashes

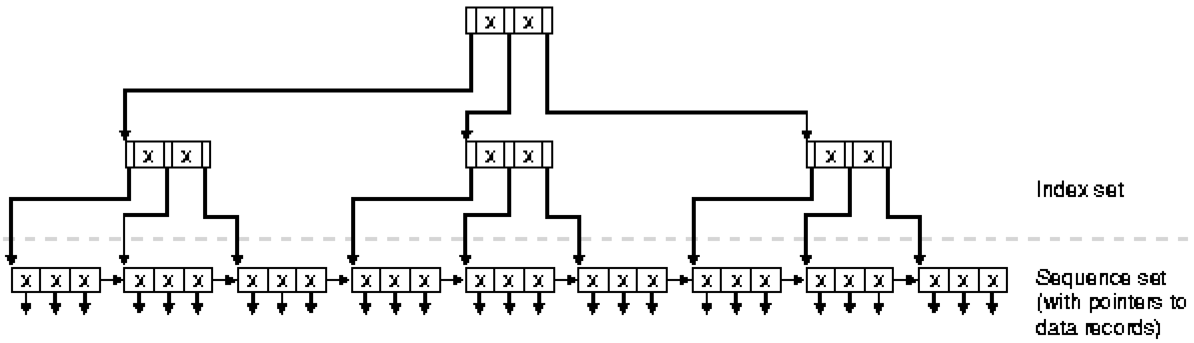
- Each vocabulary term is hashed to an integer
- Pros:
  - Lookup is faster O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps going, need to occasionally do the expensive operation of rehashing *everything*

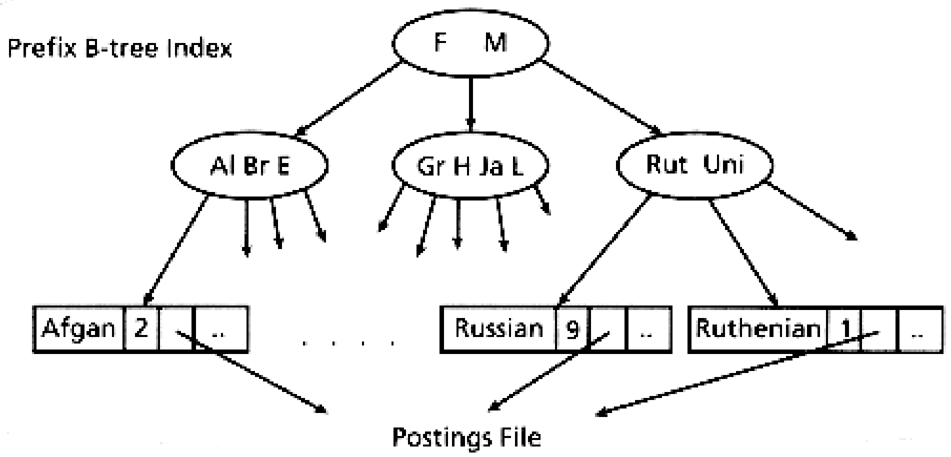
Trees

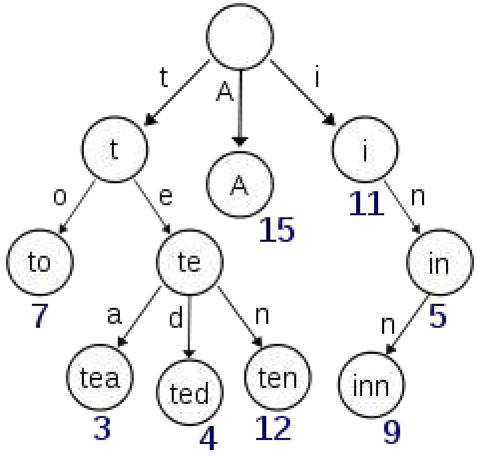


- Simplest: Binary tree
- More usual: B-trees
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

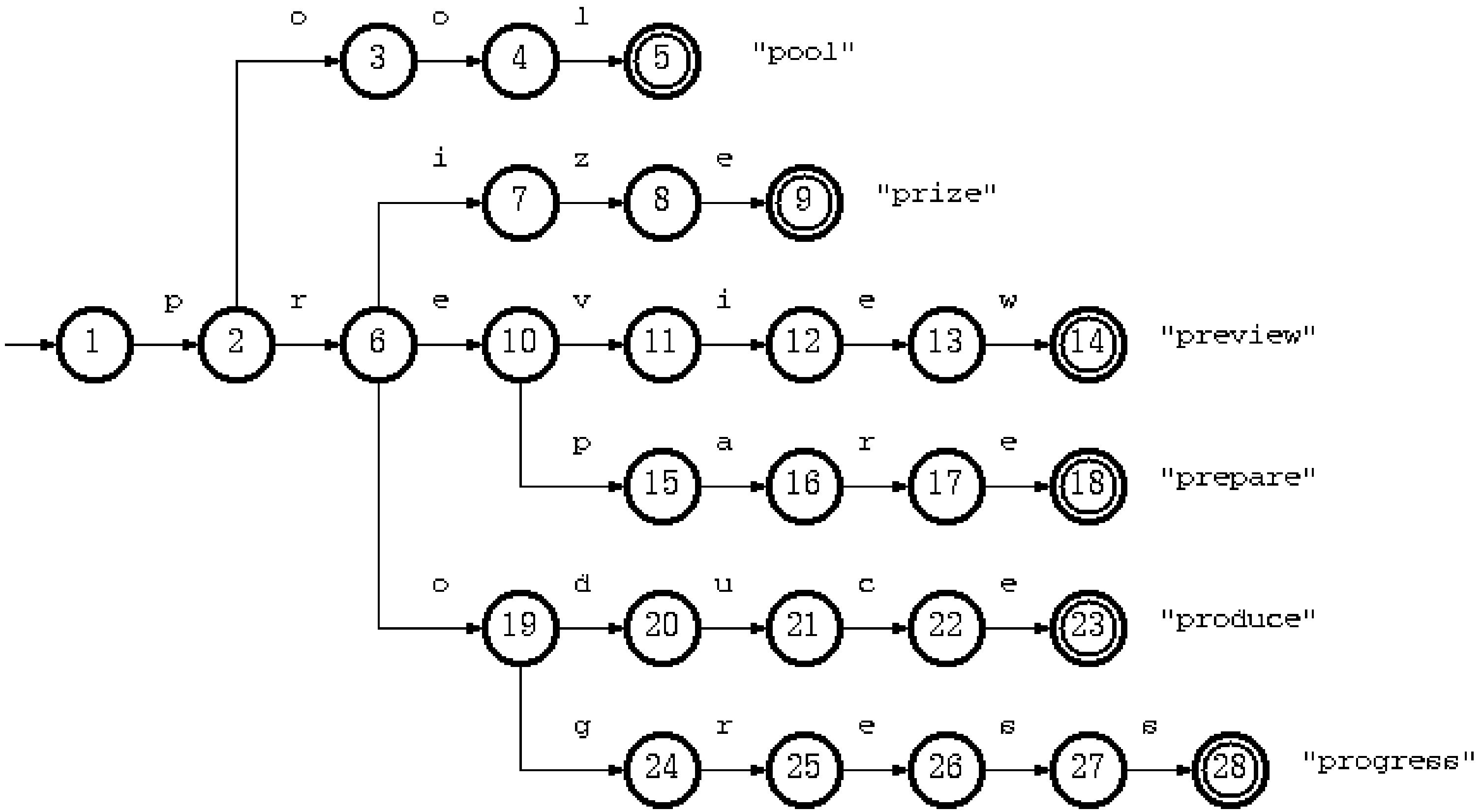
Tree: Dictionary







Tree: Dictionary



Searching

- Vocabulary search
  - Identify the words and patterns in the query
  - Search them in the vocabulary
- Retrieval of occurrences
  - Retrieve the lists of occurrences of all the words
- Manipulation of occurrences
  - Solve phrases, proximity, or Boolean operations
  - Find the exact word positions when block addressing is used

Suffix Trees and Suffix Arrays

- A text is regarded as a long string.

- Each position corresponds to a semi-infinite string.

- Suffix: a string that goes from a text position to the end of the text

- Each suffix is uniquely identified by its position
no structures and no keywords

Text

This is a text. A text has many words. Words are made from letters.

text. A text has many words. Words are made from letters.

text has many words. Words are made from letters.

many words. Words are made from letters.

Words are made from letters.

Suffixes

made from letters.

letters.

Index points are selected from the text, which point to the beginning of the text positions which are retrievable.

Reuters RCV1(Reuters Corpus Volume 1) documents

REUTERS

Reuters-21578

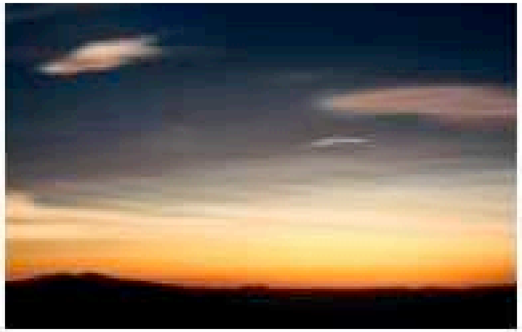
You are here: Home > News > Science > **Article**

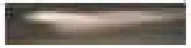
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

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

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?

Recall IIR1 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 #		Term	Doc #
I	1		ambitious	2
did	1		be	2
enact	1		brutus	1
julius	1		brutus	2
caesar	1		capitol	1
I	1		caesar	1
was	1		caesar	2
killed	1		caesar	2
i'	1		did	1
the	1		enact	1
capitol	1		hath	1
brutus	1		I	1
killed	1		I	1
me	1		i'	1
so	2		it	2
let	2	→	julius	1
it	2		killed	1
be	2		killed	1
with	2		let	2
caesar	2		me	1
the	2		noble	2
noble	2		so	2
brutus	2		the	1
hath	2		the	2
told	2		told	2
you	2		you	2
caesar	2		was	1
was	2		was	2
ambitious	2		with	2

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 postings entry, 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 2008, but typical collections are much larger. E.g. *New York Times* provides 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?

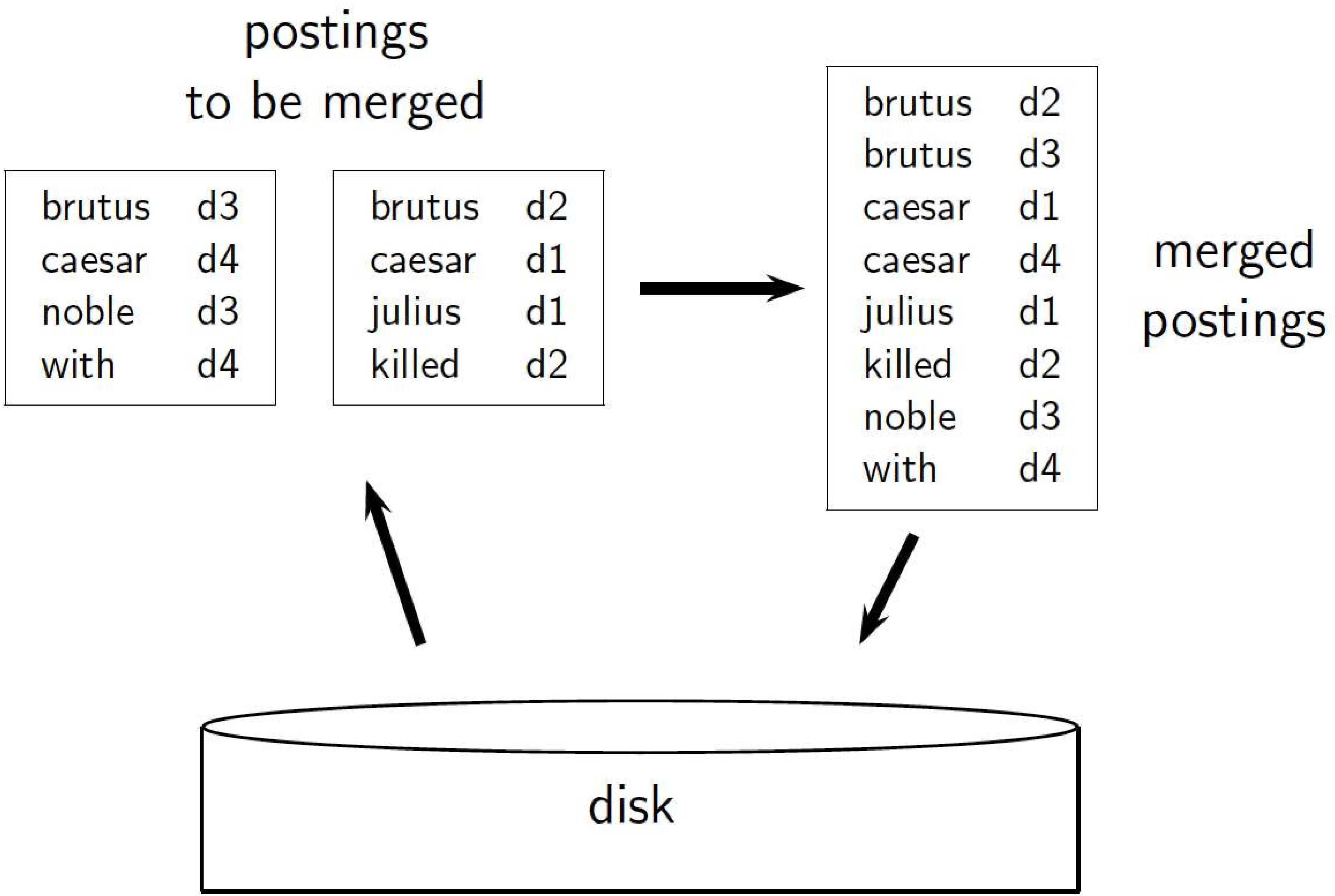
BSBI

Reuters collection example (approximate #'s)

- 100,000,000 records
- Nlog2(N) is = 2,657,542,475.91 comparisons
- 2 disk seeks per comparison = 13,287,712.38 seconds x 2
- = 26,575,424.76 seconds
- = 442,923.75 minutes
- = 7,382.06 hours
- = 307.59 days

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

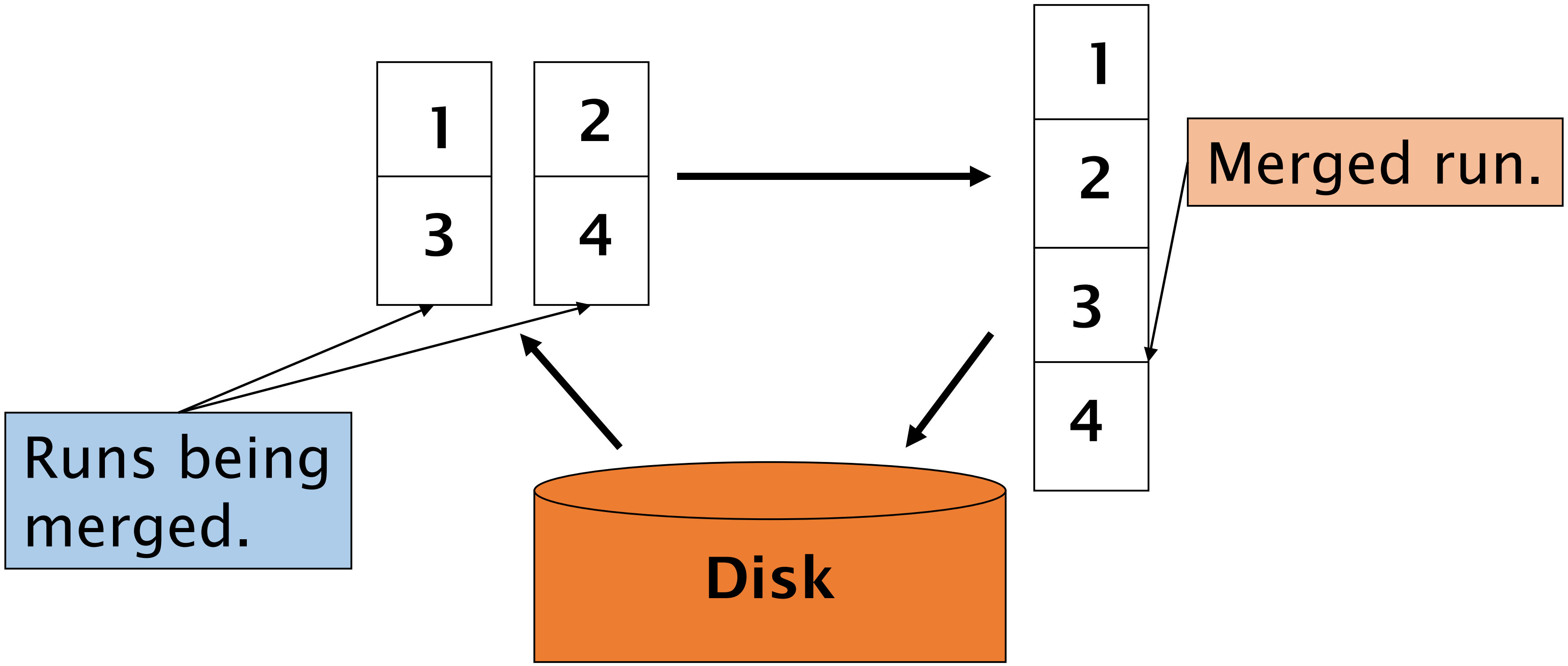
Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes $2N \ln N$ expected steps
  - In our case 2 x (10M ln 10M) steps
- *Exercise: estimate total time to read each block from disk 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()
n ← 0
while (all documents have not been processed)
do n ← n + 1
    block ← ParseNextBlock()
    BSBI-Invert(block)
    WriteBlockToDisk(block, f_n)
MergeBlocks(f_1, ..., f_n; f_merged)
```

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.



How to merge the sorted runs?

- But it is more efficient to do a *n*-way merge, where you are reading from all blocks simultaneously
- Providing you read decent-sized chunks of each block into memory, you're not killed by disk seeks

BSBI - Block sort-based indexing

Analysis of BSBI

- 12-byte records (term, doc, meta-data)
- Need to sort T= 100,000,000 such 12-byte records by term
- Define a block to have 1,600,000 such records
  - can easily fit a couple blocks in memory
  - we will be working with 64 such blocks
- 64 blocks * 1,600,000 records * 12 bytes = 1,228,800,000 bytes
- Nlog2N comparisons is 5,584,577,250.93
- 2 touches per comparison at memory speeds (10e-6 sec) =
  - 55,845.77 seconds = 930.76 min = 15.5 hours

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)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

- Merging of blocks is analogous to BSBI.

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?

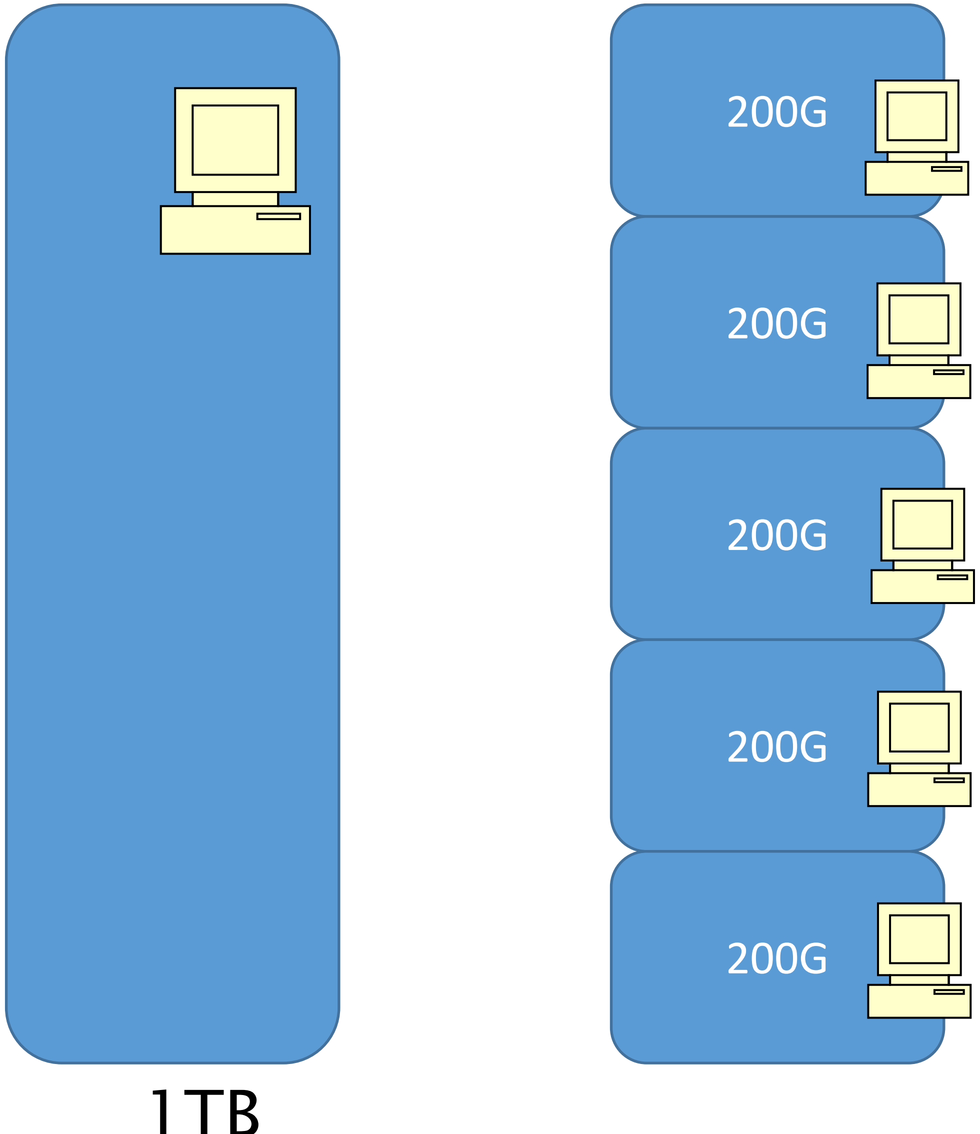
Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world.
- 「海王星計畫」in Taiwan (Changhua County)
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world!?!

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.

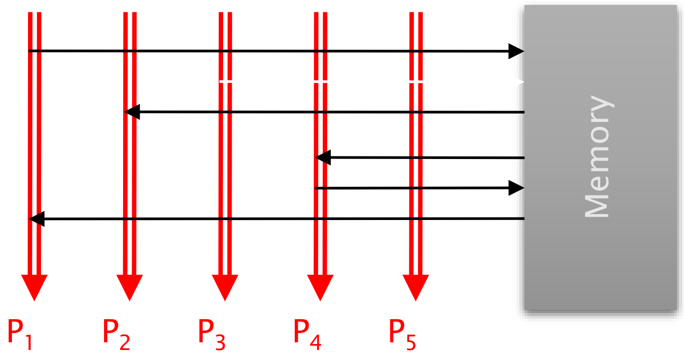
Divide and Conquer



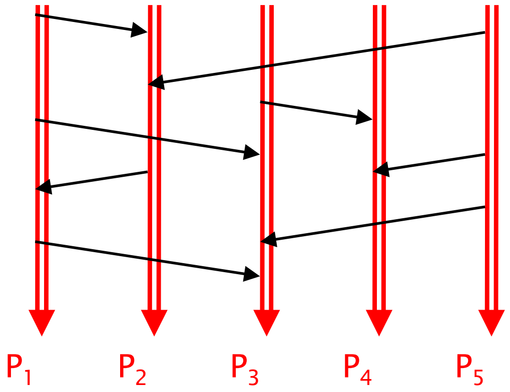
Current Tools

- Parallel Programming models
  - Shared memory (Java Threads)
  - Message passing interface (MPI)

Shared Memory



Message Passing



Managing Multiple Workers

- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
- Still, lots of problems:
  - Deadlock, livelock, …

Source: Ricardo Guimarães

Parallel tasks

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

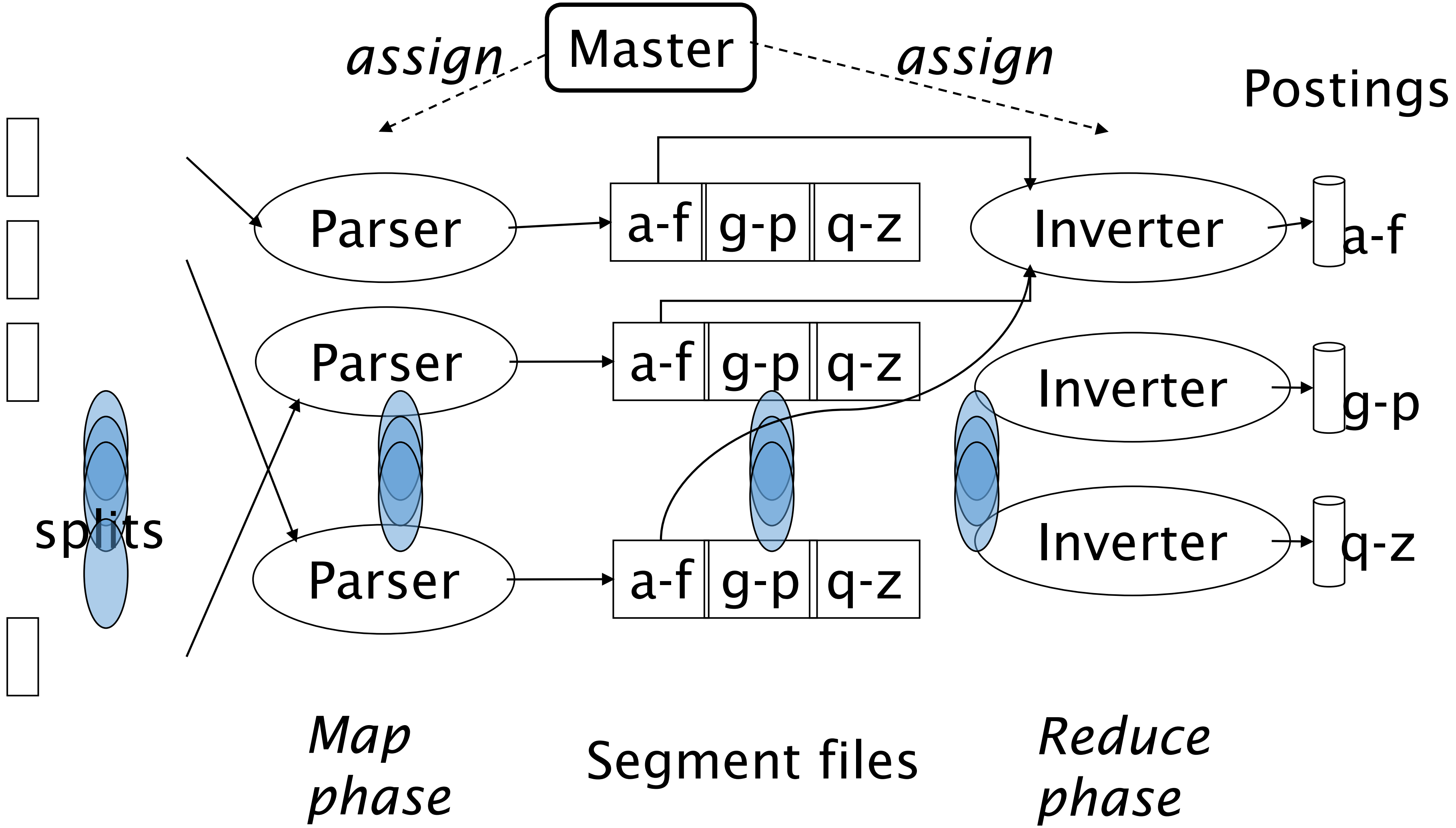
Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into *j* partitions
- 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

Inverters

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

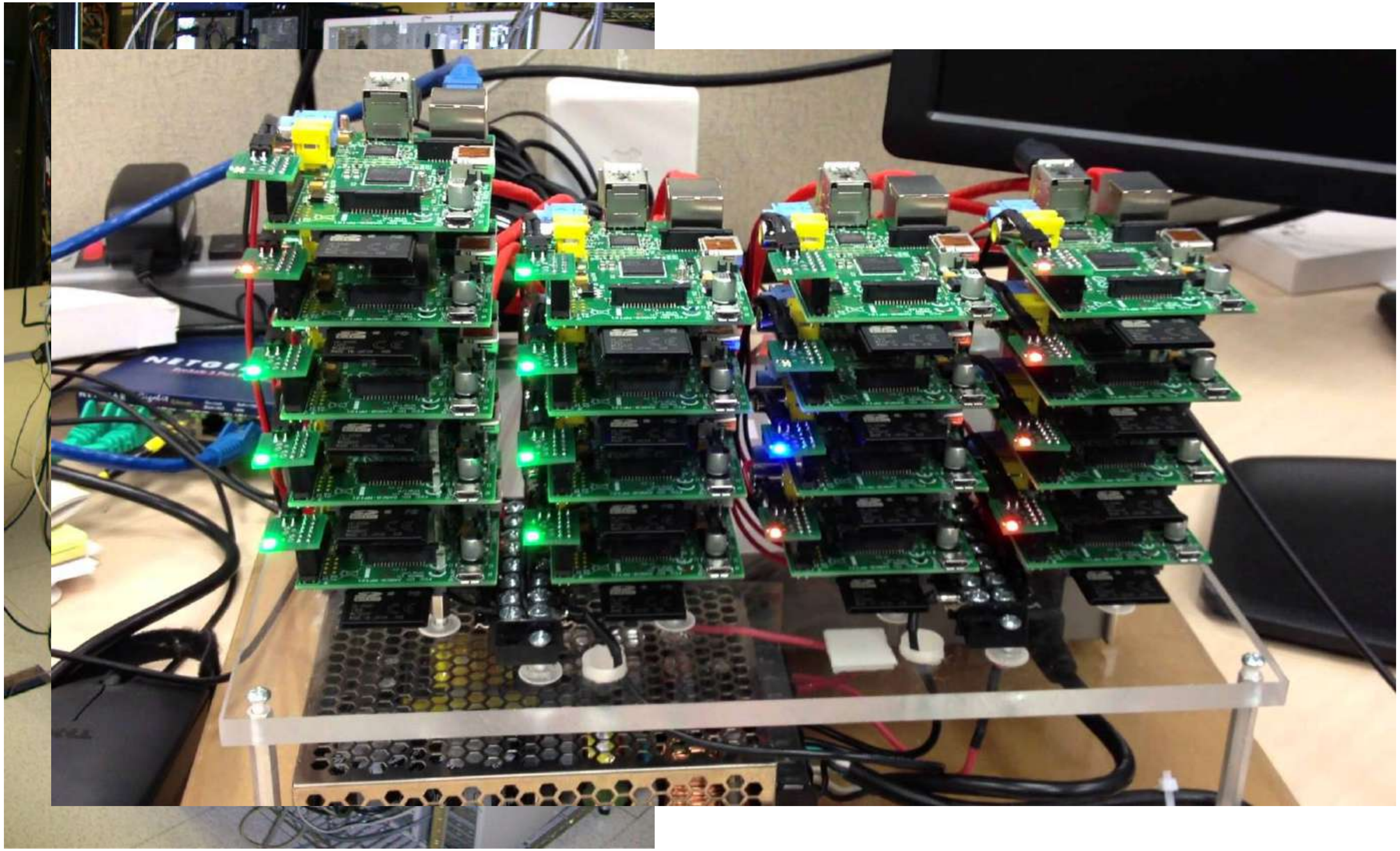
Data flow



Master
assign
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

What’s the point?

- Hide system-level details from the developers
- Using Commodity Machines to Scale up the power !



What is MapReduce?

- Parallel programming model for clusters of commodity machines
- Platform for reliable, scalable parallel computing
- Abstracts issues of parallel environment from programmer.

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem

Introduction to Hadoop

- Apache Hadoop
  - Open Source – Apache Foundation project
    - Apache Platinum Sponsor
- History
  - Started in 2005 by Doug Cutting
  - Yahoo! became the primary contributor in 2006
- Portable
  - Written in Java
  - Runs on commodity hardware
  - Linux, Mac OS/X, Windows, and Solaris

Hadoop 的擴展性以及容錯機制

- 擴展性：Hadoop 可以通過增加附加節點輕易的擴展儲存能力或處理效能，且不需要修改到程式邏輯
- 高容錯：Hadoop 可以設定 data replication，將切成小 block 的檔案複製成多份，分別放到不同的 Data Node 中，並且由 Name Node 控管儲存位置。所以如果某天運行時，其中一個 Data Node 失效、毀損造成資料遺失，還可以從其他台 Data Node 可以取得該檔案的副本資料

Growing Hadoop Ecosystem

- Hadoop Core
  - Distributed File System
  - MapReduce Framework
- Pig (initiated by Yahoo!)
  - Parallel Programming Language and Runtim
- Hbase (initiated by Powerset)
  - Table storage for semi-structured data
- Zookeeper (initiated by Yahoo!)
  - Coordinating distributed systems
- Hive (initiated by Facebook)
  - SQL-like query language and metastore

Hadoop Workflow

You

Hadoop Cluster

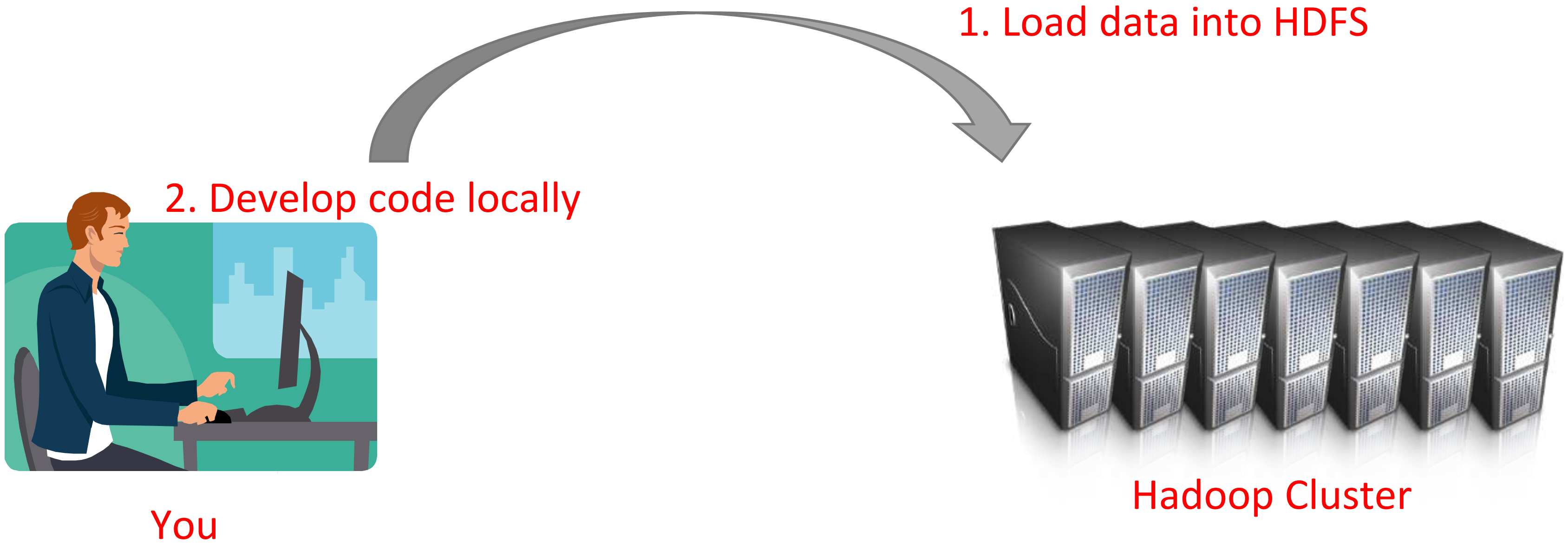
Hadoop Workflow

1. Load data into HDFS

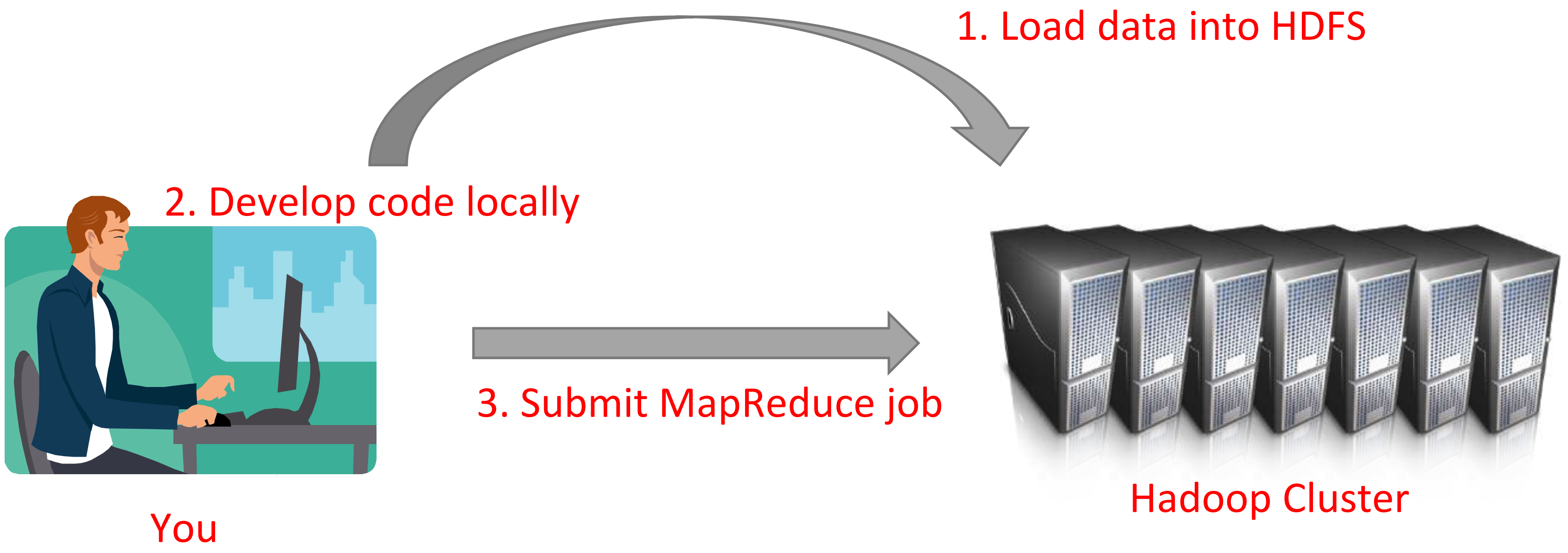
You

Hadoop Cluster

Hadoop Workflow



Hadoop Workflow



Hadoop Workflow

MapReduce

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

MapReduce

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into document-partitioned index.
  - *Term-partitioned:* one machine handles a subrange of terms
  - *Document-partitioned:* one machine handles a subrange of documents
- 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 → list(k, v)   reduce: (k,list(v)) → output

- **Instantiation of the schema for index construction**

  map: web collection → list(termID, docID)

  reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

- **Example for index construction**

  map: d2 : C died. d1 : C came, C c’ed. → (<C, d2>, <died,d2>, <C,d1>, <came,d1>, <C,d1>, <c’ed, d1>

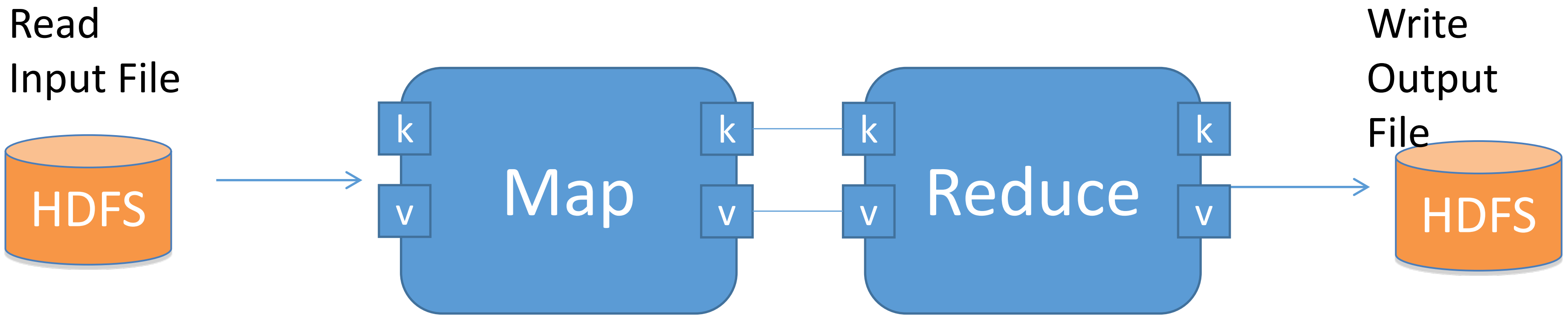
  reduce: (<C,(d2,d1,d1)>, <died,(d2)>, <came,(d1)>, <c’ed,(d1)>) → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c’ed,(d1:1)>)

Challenge

- Sorting?

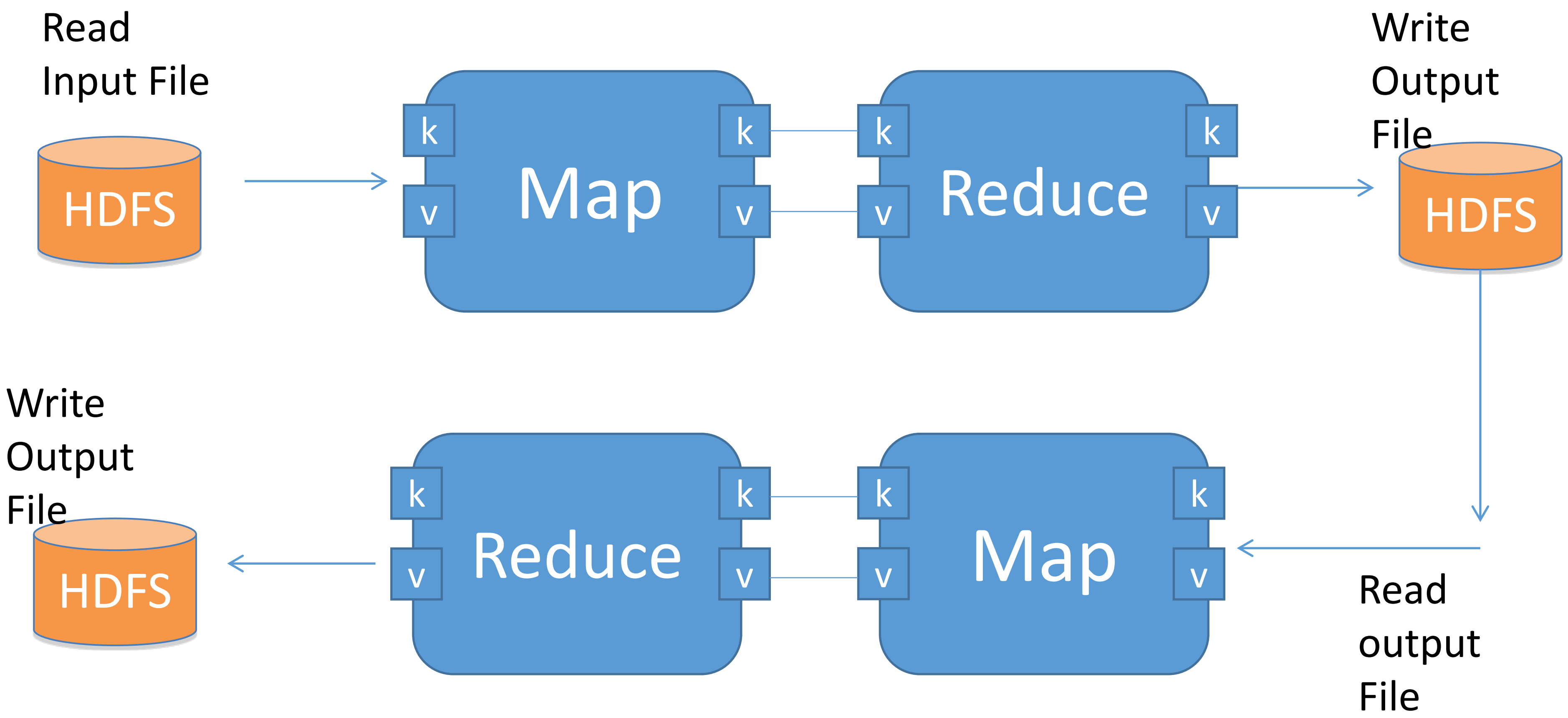
- How to perform sorting using MapReduce?

```java
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setJarByClass(WordCount.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);  //工作執行！
}
```



Parallel Programming
What is your name?
What is your name?
My name is Mickey
Hello Mickey
Hello Mickey
Hello Hello Hello

1 Parallel
1 Programming
3 Name
5 Hello
:
:
:



Parallel Programming
What is your name?
What is your name?
My name is Mickey
Hello Mickey
Hello Mickey
Hello Hello Hello

Parallel 1
Programming 1
Name 3
Hello 5
Mickey 3
:
:

1 Parallel
1 Programming
3 Name
5 Hello
:
:
:

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 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.)

Logarithmic merge

- 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.

Loga

$\text{LMergeAddToken}(\mathit{indexes}, Z_0, \mathit{token})$

$Z_0 \leftarrow \text{Merge}(Z_0, \{\mathit{token}\})$
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
    do if $I_i \in \mathit{indexes}$
      then $Z_{i+1} \leftarrow \text{Merge}(I_i, Z_i)$
        (Z_{i+1} is a temporary index on disk.)
        $\mathit{indexes} \leftarrow \mathit{indexes} - \{I_i\}$
      else $I_i \leftarrow Z_i$ *(Z_i becomes the permanent index I_i.)*
        $\mathit{indexes} \leftarrow \mathit{indexes} \cup \{I_i\}$
        Break
    $Z_0 \leftarrow \emptyset$

$\text{LogarithmicMerge}()$

$Z_0 \leftarrow \emptyset$ *(Z_0 is the in-memory index.)*
$\mathit{indexes} \leftarrow \emptyset$
while true
do $\text{LMergeAddToken}(\mathit{indexes}, Z_0, \text{getNextToken}())$

Logarithmic merge

- Auxiliary and main index: index construction time is $O(T^2)$ as each posting is touched in each merge.
- Logarithmic merge: Each posting is merged $O(\log T)$ times, so complexity is $O(T \log T)$
- 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

- Corpus-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, 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

Search Engine Land
searchengineland.com
Search Engine Land
SEO
SEM
MOBILE
LOCAL
RETAIL
GOOGLE
BING
SOCIAL
MORE
SUBSCRIBE
FOLLOW US
SMX WEST
SAN JOSE
When is SMX West 2017?
March 21-23, 2017 I San Jose, CA
SMX is coming…
Mark your calendars now!
Google begins mobile-first indexing, using mobile content for all
Carousels of Google Shopping ads spotted on YouTube
Study shows Google's Possum update changed 64% of local SERPs
Google to add presidential, senatorial & congressional election results directly in
Meet a Landy Award winner: Noble Studios drives traffic to Tahoe South to win Best SEO Initiative for Small
searchengineland.com/google-shopping-ads-carousel-youtube-262662
上午 07:57
2016/11/8

Other sorts of indexes

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

Why?

- 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