CS4051

Information Retrieval

Week 04

Muhammad Rafi
February 15, 2024

Agenda

- Hardware Basics
- Blocked Sort-Based Indexing
- Single-Pass In-Memory Indexing
- Distributed indexing
- Dynamic indexing
- Conclusion

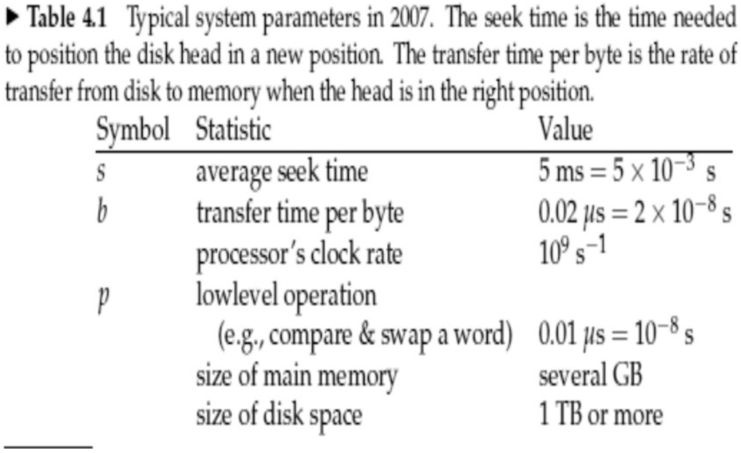
Hardware Basics

- Many design decisions in information retrieval are based on the characteristics of hardware
- Most search systems have their own hardware accessing mechanism and standards
- 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 (2007)



▶ **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	0.02 μs = 2×10^{-8} s
	processor's clock rate	10^9 s^{-1}
p	lowlevel 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

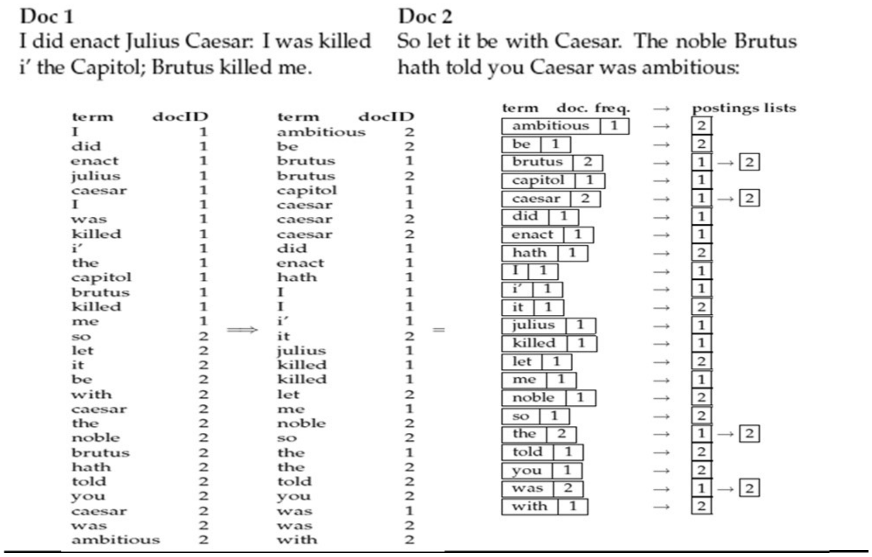
Hardware Basics

- 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.

Basic Inverted Index

- The basic steps in constructing a non-positional index as discussed in chapter 1.
- 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.
- Finally, 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.
- For large collections (most now in IR)

Example



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	docID
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

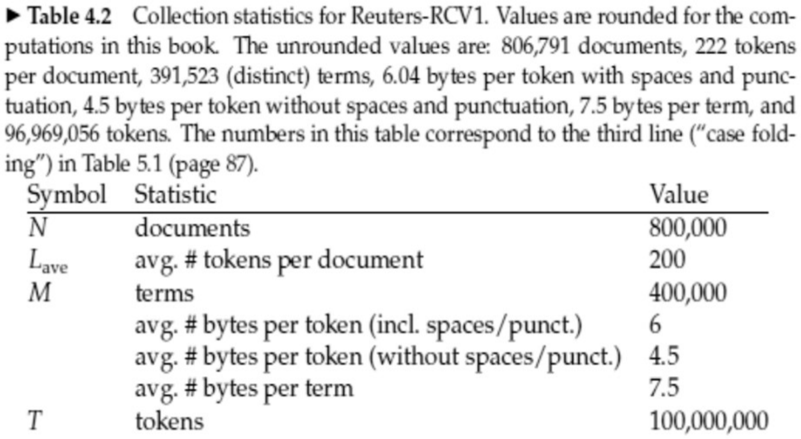
⟹

term	docID
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

Reuters- RCV1 Collection



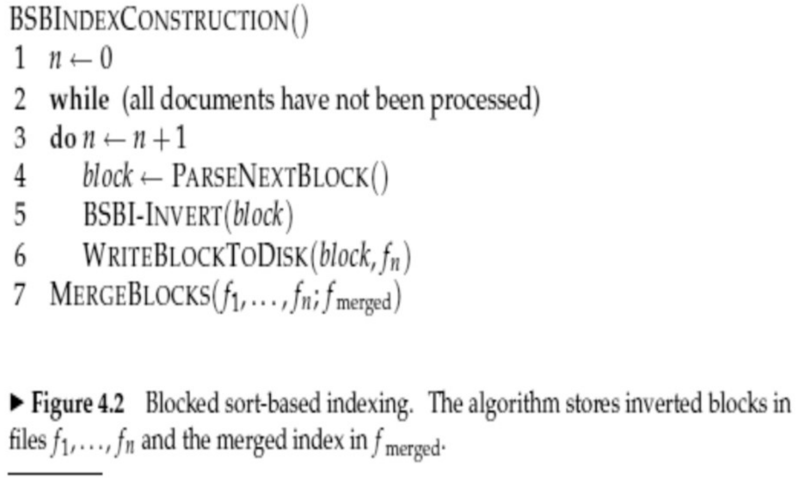
▶ **Table 4.2** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this book. The unrounded values are: 806,791 documents, 222 tokens per document, 391,523 (distinct) terms, 6.04 bytes per token with spaces and punctuation, 4.5 bytes per token without spaces and punctuation, 7.5 bytes per term, and 96,969,056 tokens. The numbers in this table correspond to the third line ("case folding") in Table 5.1 (page 87).

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	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
T	tokens	100,000,000

Blocked Sort-Based Indexing

- Blocked Sort-Based Indexing (BSBI)
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.
- Algorithm (BSBI)
  - Segments the collection into parts of equal size,
  - Sorts the termID–docID pairs of each part in memory,
  - Stores intermediate sorted results on disk, and
  - Merges all intermediate results into the final index.

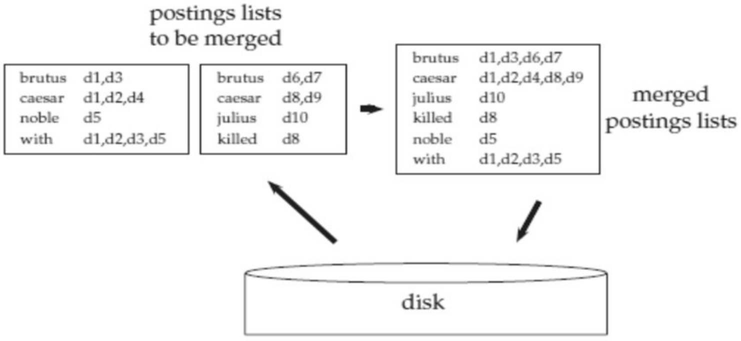
Blocked Sort-Based Indexing



```
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)
```

► **Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

Blocked Sort-Based Indexing



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

► **Figure 4.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

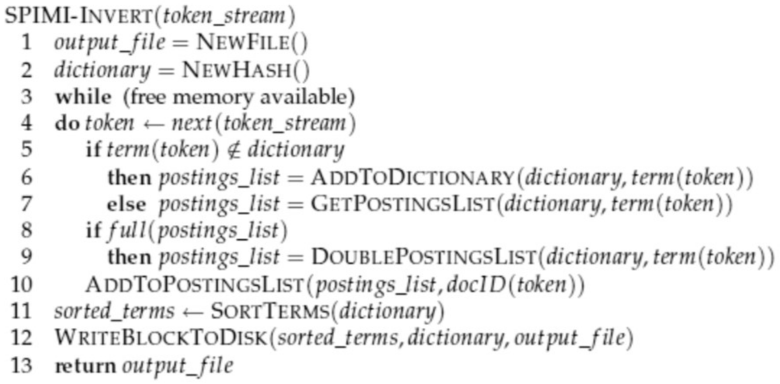
Blocked Sort-Based Indexing

- BSBI Complexity
  - How expensive is BSBI?
  - Its time complexity is Q(T log T) because the step with the highest time complexity is sorting and T is an upper bound for the number of items we must sort (i.e., the number of termID–docID pairs).
  - The actual indexing time is usually dominated by the time it takes to parse the documents (PARSENEXTBLOCK) and to do the final merge (MERGEBLOCKS).

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.

Single-Pass In-Memory Indexing



```
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
```

► **Figure 4.4** Inversion of a block in single-pass in-memory indexing

BSBI vs. SPIMI

- A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list (line 10). 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.
  - This has two advantages:
    - It is faster because there is no sorting required
    - it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored. As a result, the blocks that individual calls of SPIMI-INVERT can process are much larger and the index construction process as a whole is more efficient.

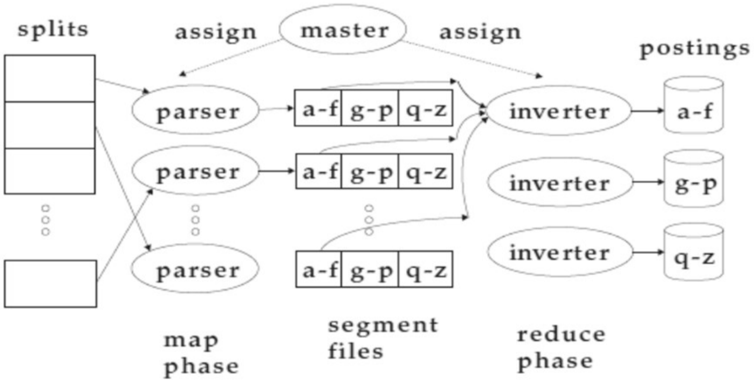
Distributed Indexing

- Web scale search engine cannot build index on single server. Single machine is fault-prone?
- Distributed Systems is the key
  - 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.
- 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)

Distributed Indexing

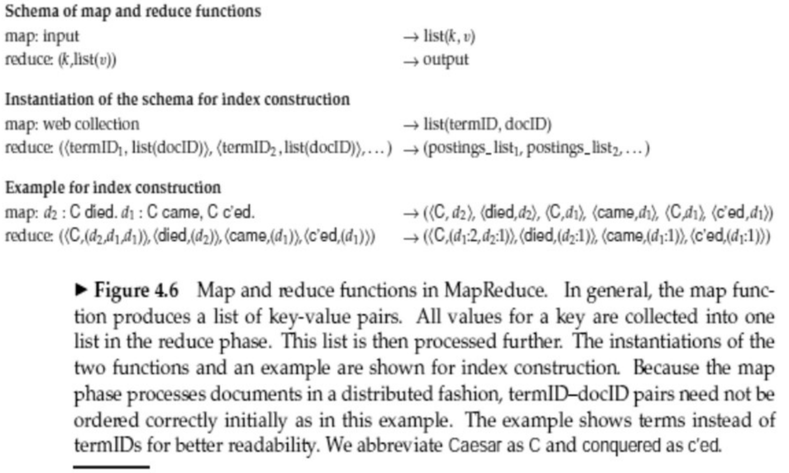
- Distributed Indexing
  - Parallel Tasks
  - Parser
    - Master assigns a split to an idle parser machine
    - Parser reads a document at a time and emits (term, doc) pairs
    - Parser writes pairs into *j* partitions
  - Indexer
    - An inverter collects all (term,doc) pairs (= postings) for one term-partition.
    - Sorts and writes to postings lists

Distributed Indexing



► **Figure 4.5** An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

Map-Reduce Based Indexing



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: web collection	$\rightarrow$ list(termID, docID)
reduce: ($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (postings_list$_1$, postings_list$_2$, ...)

Example for index construction

map: d_2 : C died. d_1 : C came, C c'ed.	$\rightarrow$ ($\langle$C, $d_2\rangle$, $\langle$died,$d_2\rangle$, $\langle$C,$d_1\rangle$, $\langle$came,$d_1\rangle$, $\langle$C,$d_1\rangle$, $\langle$c'ed,$d_1\rangle$)
reduce: ($\langle$C,(d_2,d_1,d_1)$\rangle$, $\langle$died,(d_2)$\rangle$, $\langle$came,(d_1)$\rangle$, $\langle$c'ed,(d_1)$\rangle$)	$\rightarrow$ ($\langle$C,(d_1:2,d_2:1)$\rangle$, $\langle$died,(d_2:1)$\rangle$, $\langle$came,(d_1:1)$\rangle$, $\langle$c'ed,(d_1:1)$\rangle$)

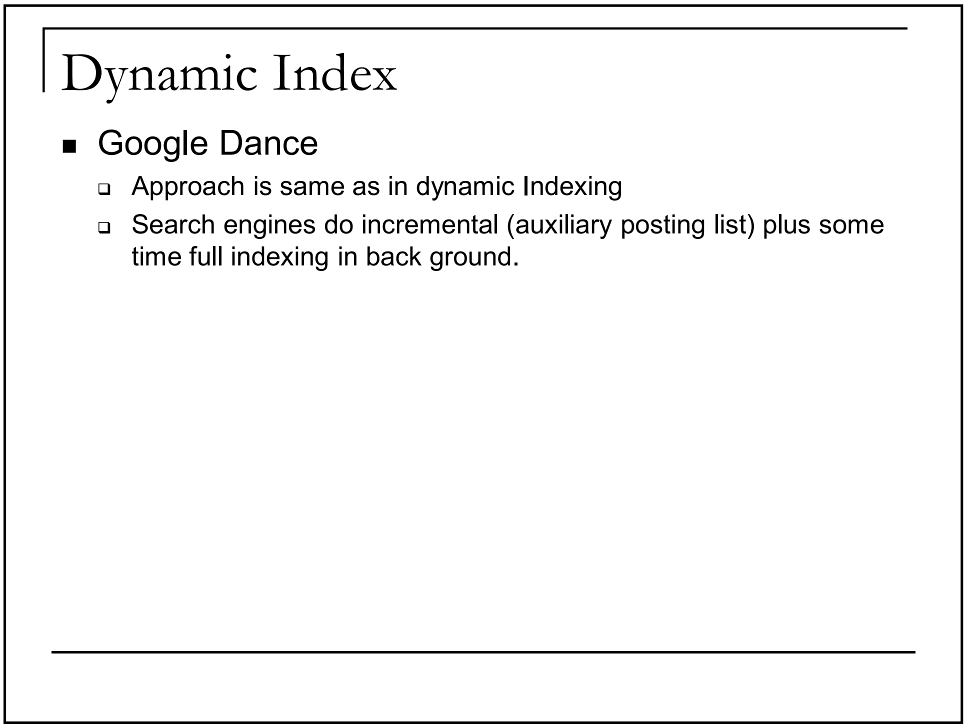
► **Figure 4.6** Map and reduce functions in MapReduce. In general, the map function produces a list of key-value pairs. All values for a key are collected into one list in the reduce phase. This list is then processed further. The instantiations of the two functions and an example are shown for index construction. Because the map phase processes documents in a distributed fashion, termID–docID pairs need not be ordered correctly initially as in this example. The example shows terms instead of termIDs for better readability. We abbreviate Caesar as C and conquered as c'ed.

Dynamic Index

- 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

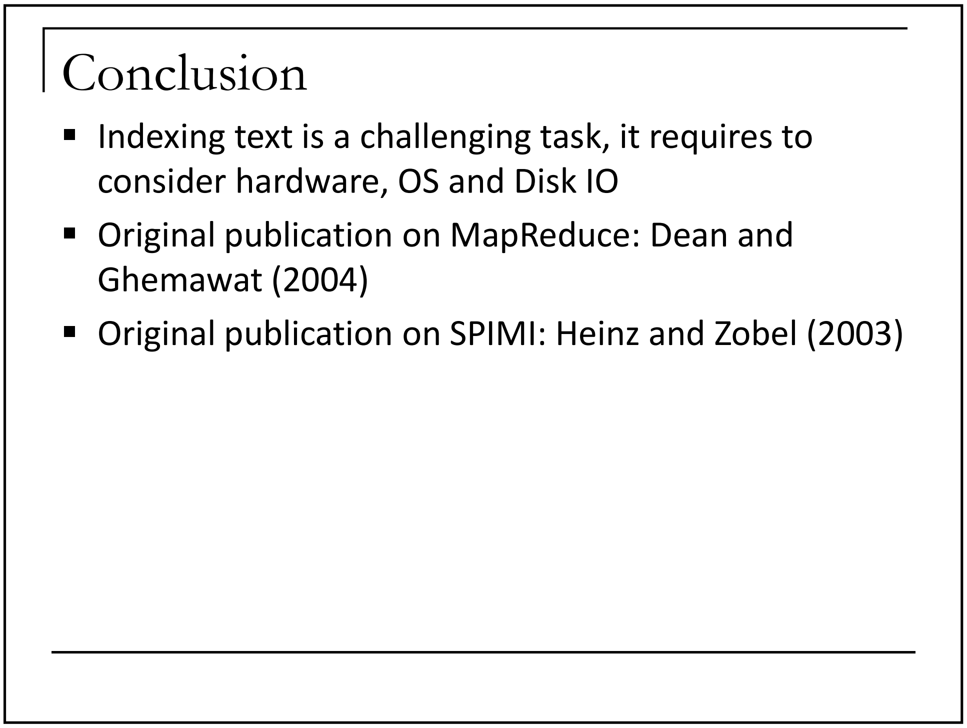
Dynamic Index

- Simple 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



Dynamic Index

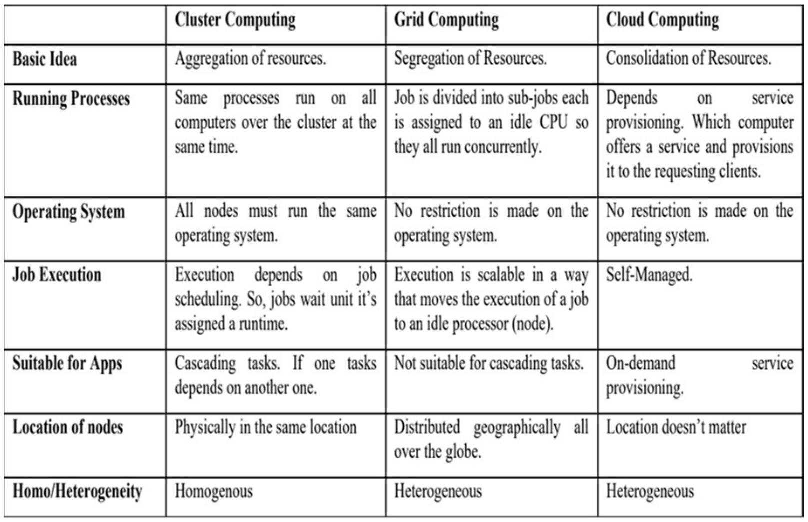
- Google Dance
  - Approach is same as in dynamic Indexing
  - Search engines do incremental (auxiliary posting list) plus some time full indexing in back ground.



Conclusion

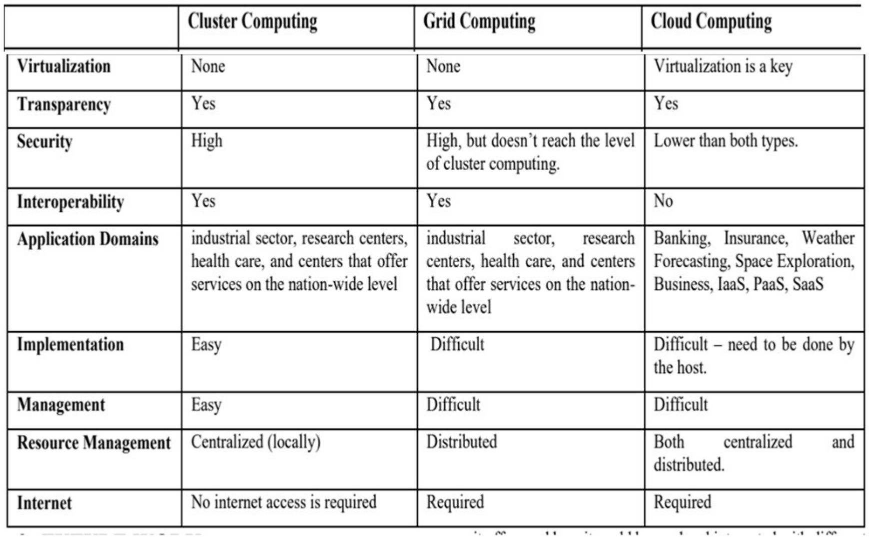
- Indexing text is a challenging task, it requires to consider hardware, OS and Disk IO
- Original publication on MapReduce: Dean and Ghemawat (2004)
- Original publication on SPIMI: Heinz and Zobel (2003)

Cluster vs. Grid vs. Cloud



	Cluster Computing	Grid Computing	Cloud Computing
Basic Idea	Aggregation of resources.	Segregation of Resources.	Consolidation of Resources.
Running Processes	Same processes run on all computers over the cluster at the same time.	Job is divided into sub-jobs each is assigned to an idle CPU so they all run concurrently.	Depends on service provisioning. Which computer offers a service and provisions it to the requesting clients.
Operating System	All nodes must run the same operating system.	No restriction is made on the operating system.	No restriction is made on the operating system.
Job Execution	Execution depends on job scheduling. So, jobs wait unit it's assigned a runtime.	Execution is scalable in a way that moves the execution of a job to an idle processor (node).	Self-Managed.
Suitable for Apps	Cascading tasks. If one tasks depends on another one.	Not suitable for cascading tasks.	On-demand service provisioning.
Location of nodes	Physically in the same location	Distributed geographically all over the globe.	Location doesn't matter
Homo/Heterogeneity	Homogenous	Heterogeneous	Heterogeneous

Cluster vs. Grid vs. Cloud



	Cluster Computing	Grid Computing	Cloud Computing
Virtualization	None	None	Virtualization is a key
Transparency	Yes	Yes	Yes
Security	High	High, but doesn't reach the level of cluster computing.	Lower than both types.
Interoperability	Yes	Yes	No
Application Domains	industrial sector, research centers, health care, and centers that offer services on the nation-wide level	industrial sector, research centers, health care, and centers that offer services on the nation-wide level	Banking, Insurance, Weather Forecasting, Space Exploration, Business, IaaS, PaaS, SaaS
Implementation	Easy	Difficult	Difficult – need to be done by the host.
Management	Easy	Difficult	Difficult
Resource Management	Centralized (locally)	Distributed	Both centralized and distributed.
Internet	No internet access is required	Required	Required