Information Retrieval

BITS Pilani
Pilani Campus

Abhishek
January 2020

CS F469, Information Retrieval
Lecture topic: Index Construction

BITS Pilani
Pilani Campus

Most of these slides are based on:

https://web.stanford.edu/class/cs276/
https://www.inf.unibz.it/~ricci/ISR/
https://www.cis.uni-muenchen.de/~hs/teach/14s/ir/

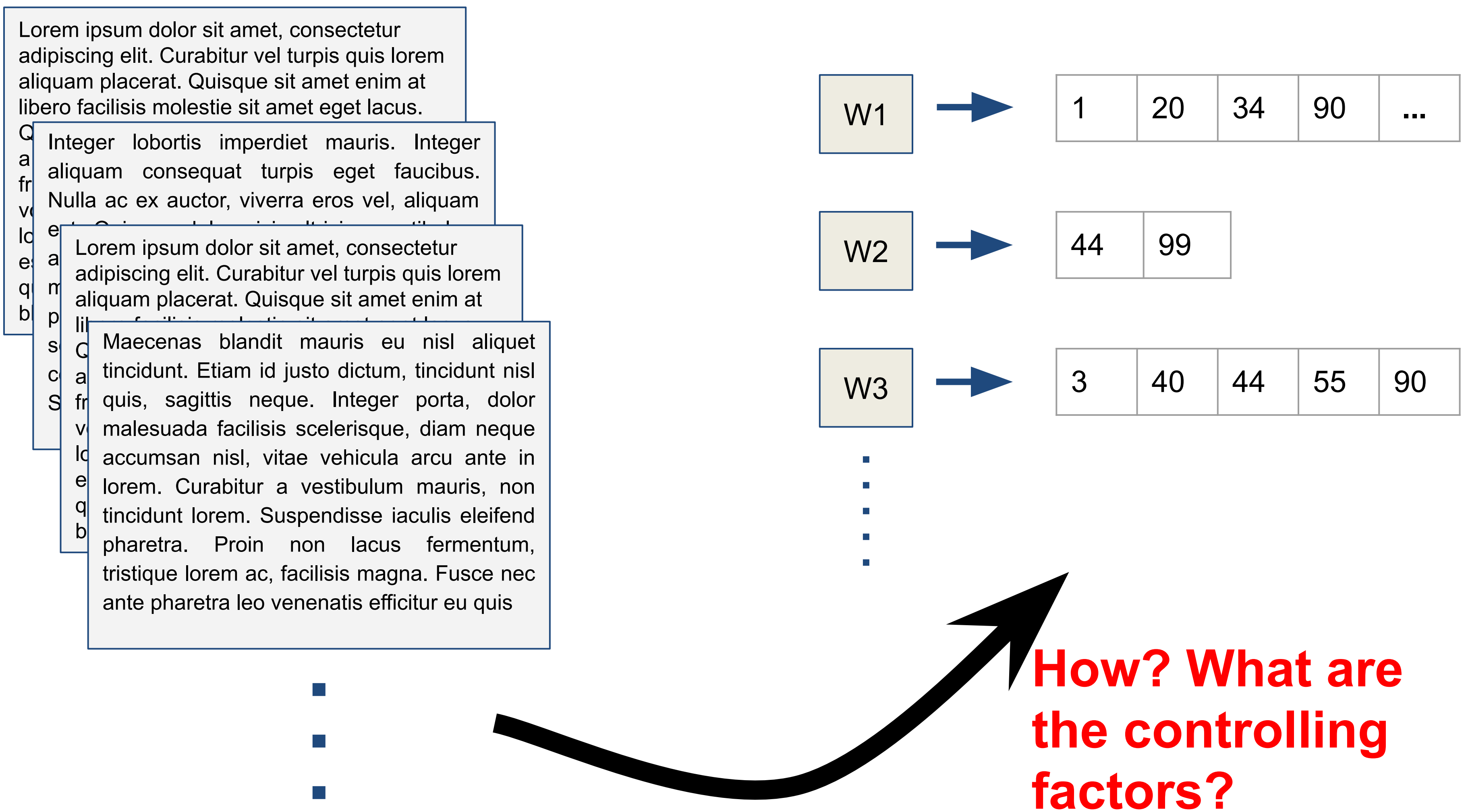
Recap of Previous Lecture

- Tolerant Retrieval
  - Wildcard queries
  - Spelling correction

This Lecture

- Index Construction
  - Blocked sort-based indexing
  - Single pass in-memory indexing
  - Distributed Indexing
  - Dynamic Indexing

Documents → Index



How? What are the controlling factors?

Recap: Building an Inverted Index (Lecture 2)

Step 1: Collect the documents to be indexed.

Example with two documents.

Doc 1:

A quick brown fox jumps over a lazy dog.

Doc 2:

The quick sly fox jumped over the lazy brown dog.

Recap: Building an Inverted Index (Lecture 2)

Step 2: Tokenize the documents into list of tokens.

Doc 1:

A	quick	brown	fox	jumps	over	a	lazy	dog	.

Doc 2:

The	quick	sly	fox	jumped	over	the	lazy	brown	dog	.

Recap: Building an Inverted Index (Lecture 2)

Step 3: Do some linguistic preprocessing, eg. lowercase

Doc 1:

a	quick	brown	fox	jumps	over	a	lazy	dog

Doc 2:

the	quick	sly	fox	jumped	over	the	lazy	brown	dog

Recap: Building an Inverted Index (Lecture 2)

Step 4: Build the inverted index considering the tokens as terms.

a	1	the	2
quick	1	quick	2
brown	1	sly	2
fox	1	fox	2
jumps	1	jumped	2
over	1	over	2
a	1	the	2
lazy	1	lazy	2
dog	1	brown	2
		dog	2

Recap: Building an Inverted Index (Lecture 2)

Step 4: Build the inverted index considering the tokens as terms.

a	1
brown	1, 2
dog	1, 2
fox	1, 2
jumped	2
jumps	1
lazy	1, 2
over	1, 2
quick	1, 2
sly	2
the	2

Sort Based Index Construction (Summary)

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

Sort Based Index Construction (Summary)

1. Parse each documents **once** (tokenization, normalization etc.) and generate (term, doc_ID) ordered pairs.
2. Sort (term, doc_ID) ordered pairs.
3. Merge sorted ordered pairs.

Limitations of Sort Based Index Construction

Assumes all of the data (used for sorting) can fit in memory.

Data and Scale of IR systems

Data: Two Extreme Cases

~1k to 10k documents

Use Cases:

- Personal PC
- Single websites
- Small organization

Over Billion documents

Use Cases:

- Internet search
- Very large organizations

Scale of IR Systems

1. The unstructured corpus contains thousands of documents that **can fit into a RAM of a single system**.
2. The unstructured corpus contains millions of documents that can not fit into RAM of a single system but **can fit into DISK of a single system**.
3. The unstructured corpus contains billions of documents that **can not fit into a DISK of a single system**.

Algorithms for millions of documents

Sort using Disk as Memory?

- We can use the same algorithm for sorting for large collection, but by using disks as memory?

Sort using Disk as Memory?

- We can use the same algorithm for sorting for large collection, but by using disks as memory?

- NO: Disk and RAM have different access time. Disk is much slower than RAM.

Sort using Disk as Memory?

- We can use the same algorithm for sorting for large collection, but by using disks as memory?
- NO: Disk and RAM have different access time. Disk is much slower than RAM.
- We need an external sorting algorithm.
- Let's look at some hardware basics.

Hardware Basics

- Assess to data is much faster on memory than disk.
- Disks have mechanical (moving) components, whereas RAMs are purely semiconductor devices.
- **Disk Seek:** No data is transferred, while the disk head is being positioned.
- Disk IO is block based: 8KB to 512KB.

How hard drive work:
https://www.youtube.com/watch?v=wteUW2sL7bc

Hardware Assumptions

Average seek time (disk)	5 milli sec
Transfer time per byte (disk)	0.02 micro sec (2×10^{-8} s/B)
processor's clock rate	1 nano sec

Example: Reading 1GB from disk

- If stored in contiguous blocks: 2×10^{-8} s/B $\times 10^{9}$ B = 20s
- If stored in 1M chunks of 1KB: 20s + $10^{6} \times 5 \times 10^{-3}$s = 5020 s = 1.4 h

BSBI: Blocked sort-based Indexing

BSBI: Blocked sort-based Indexing

- Assumption: The {term → termID} dictionary is available in memory.

BSBI: Blocked sort-based Indexing

- Assumption: The {term → termID} dictionary is available in memory.

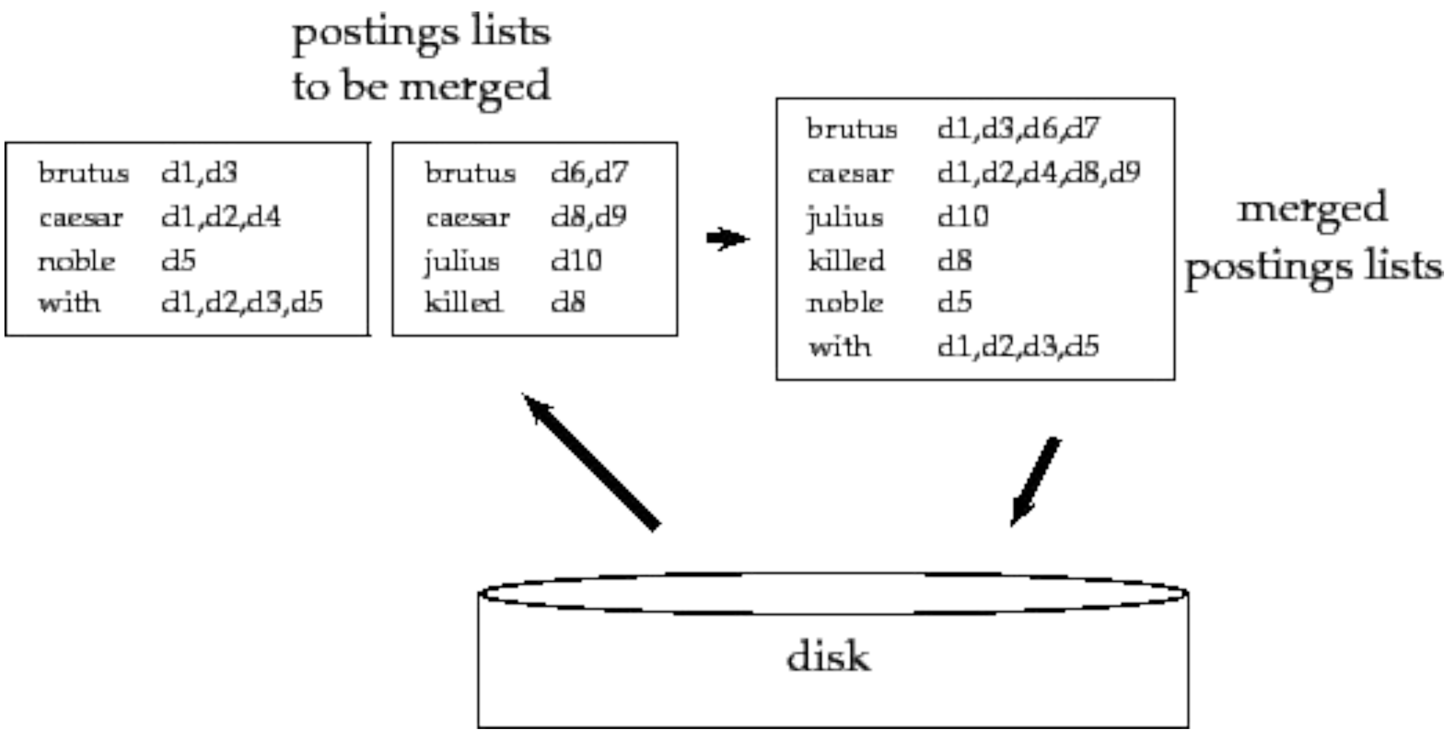
1. Generate ordered pairs (termID, docID) as we parse the documents.
2. Divide the list of ordered pairs into blocks.
3. Sort block in memory, create a index, write it on the disk.
4. Merge several sorted blocks to construct the final index.

BSBI: 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 4.1** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

BSBI: Blocked sort-based Indexing (merging)



BSBI: Limitations

- Assumption: The {term $\rightarrow$ termID} dictionary is available in memory.
- We can construct the dictionary dynamically.
- Actually, we can also work without the dictionary, …
- … but the intermediate files would become larger, and the algorithm will be slower.

Single-pass in-memory Indexing

Single-pass in-memory Indexing

- Generate separate dictionaries for each block, no need for a term - termID mapping.
- Don't sort the posting, accumulate the posting as they occurs.
  - In the end, before writing the block, sort the terms.

Single-pass in-memory Indexing

```
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
```

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

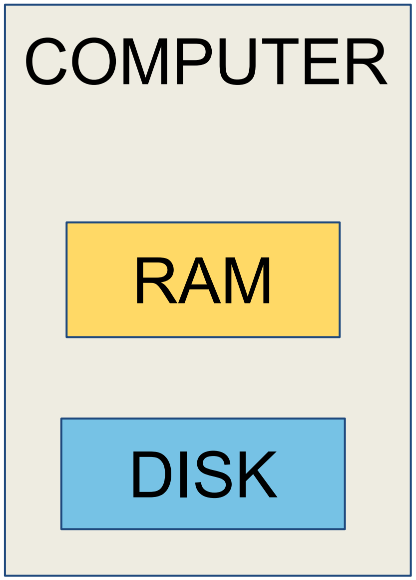
Till now

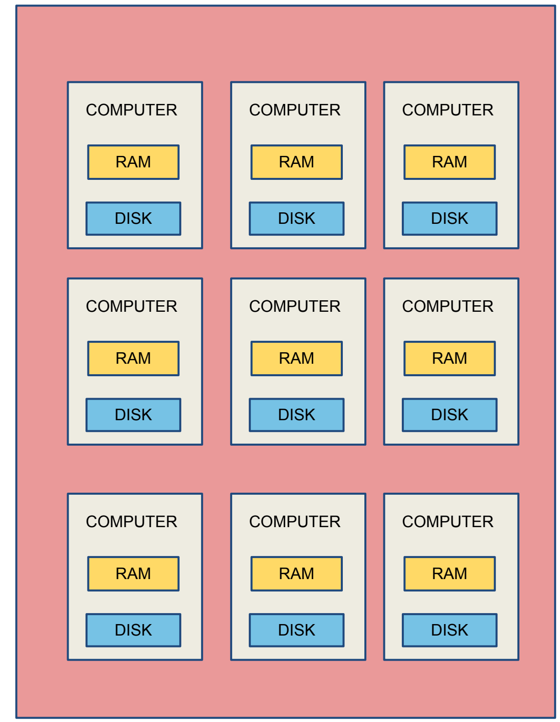
- Document collection was static.
- Data can fit into a single machine (Disk or memory).

Algorithms for billions of documents

- Distributed Indexing

Hardware: Single vs Multiple Systems





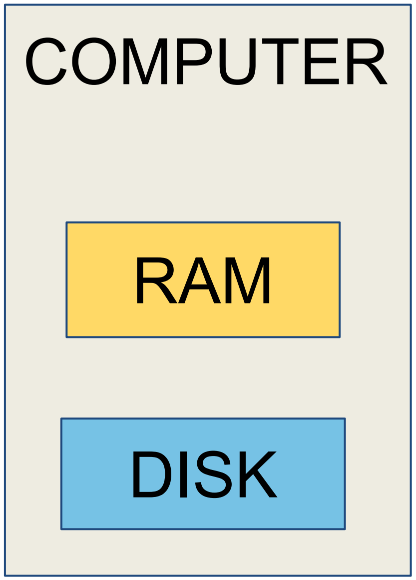
SIngle Machine Max Limit:
RAM: ~ **6TB**
Disk: ~ **360TB**

Very expensive: approx 200K USD

Average Machine:
RAM: 64GB
Disk: 12TB

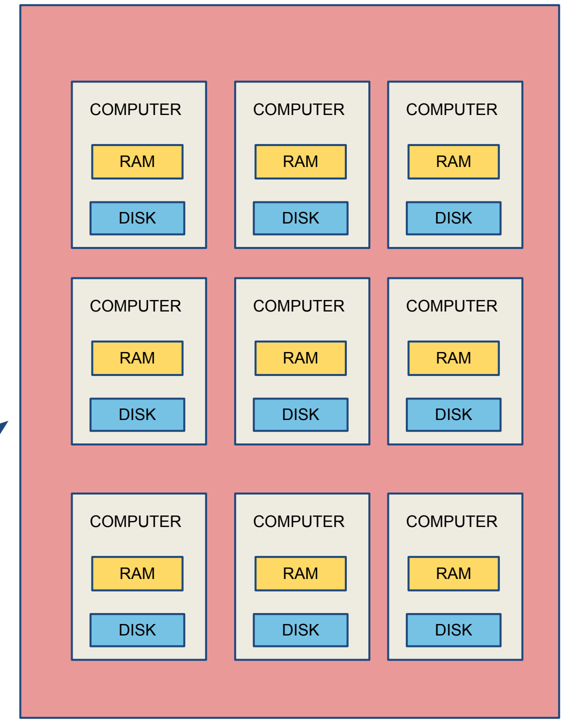
Relatively cheap: approx 2K USD

Hardware: Single vs Multiple Systems



Single point of failure

Each node with an uptime of 99.9%. What would be the uptime of the entire system?



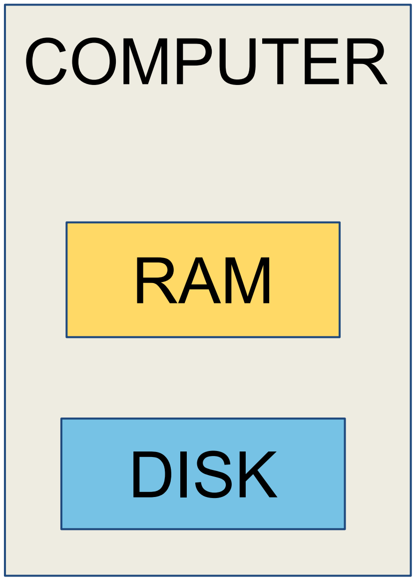
SIngle Machine Max Limit:
RAM: ~ **6TB**
Disk: ~ **360TB**

Very expensive: approx 200K USD

Average Machine:
RAM: 64GB
Disk: 12TB

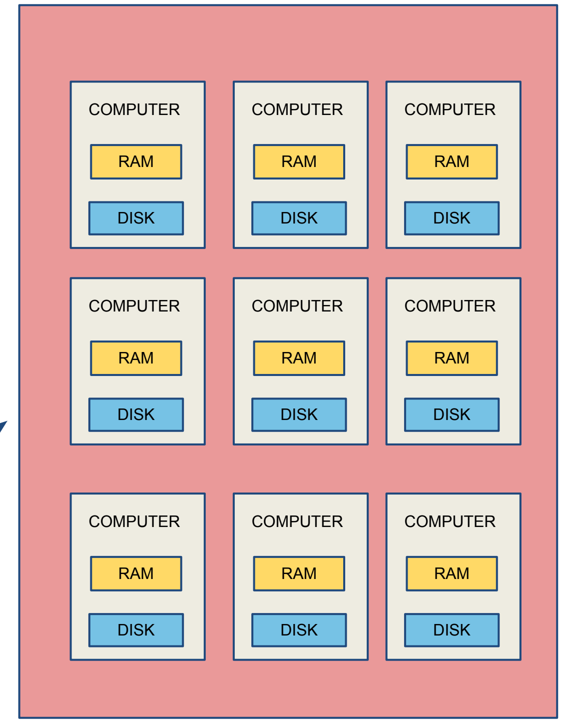
Relatively cheap: approx 2K USD

Hardware: Single vs Multiple Systems



Single point of failure

Each node with an uptime of 99.9%. What would be the uptime of the entire system?



SIngle Machine Max Limit:
RAM: ~ **6TB**
Disk: ~ **360TB**

Very expensive: approx 200K USD

(0.999)^100 = 90%
(0.999)^150 = 86%

Average Machine:
RAM: 64GB
Disk: 12TB

Relatively cheap: approx 2K USD

Google Data centers

- Mainly consists of commodity machines.
- Data centers are distributed around the world.
- An estimate of about 2.5 millions servers.[1]
- Suppose a server fails in 3 years. How many Google servers will fail in a day?

Video: [Inside a Google data center]

[1] https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq

Mapreduce Basics

Mapreduce key Terminologies

- **Master Node (Name Node)**
- **Slave/worker Node (Data Node)**
- **Map Process**
- **Reduce Process**

The data is not stored on the master node. The master nodes controls and assign tasks to slave/worker nodes.

Mapreduce paradigm

- **Data parallelism:**
  - Lots of data → Break into chunks
  - Process individual chunks of data in parallel
  - Combine the results from individual chunks
- **Example:**
  - Number of Documents = 20 billion
  - Divide these documents into say 10 million documents.
  - Process every 10 million documents parallely on different machines.
  - Combine the result obtained from individual machines.

Mapreduce Workflow

(Use board)

Mapreduce Example: Count Word Frequency

(Use board)

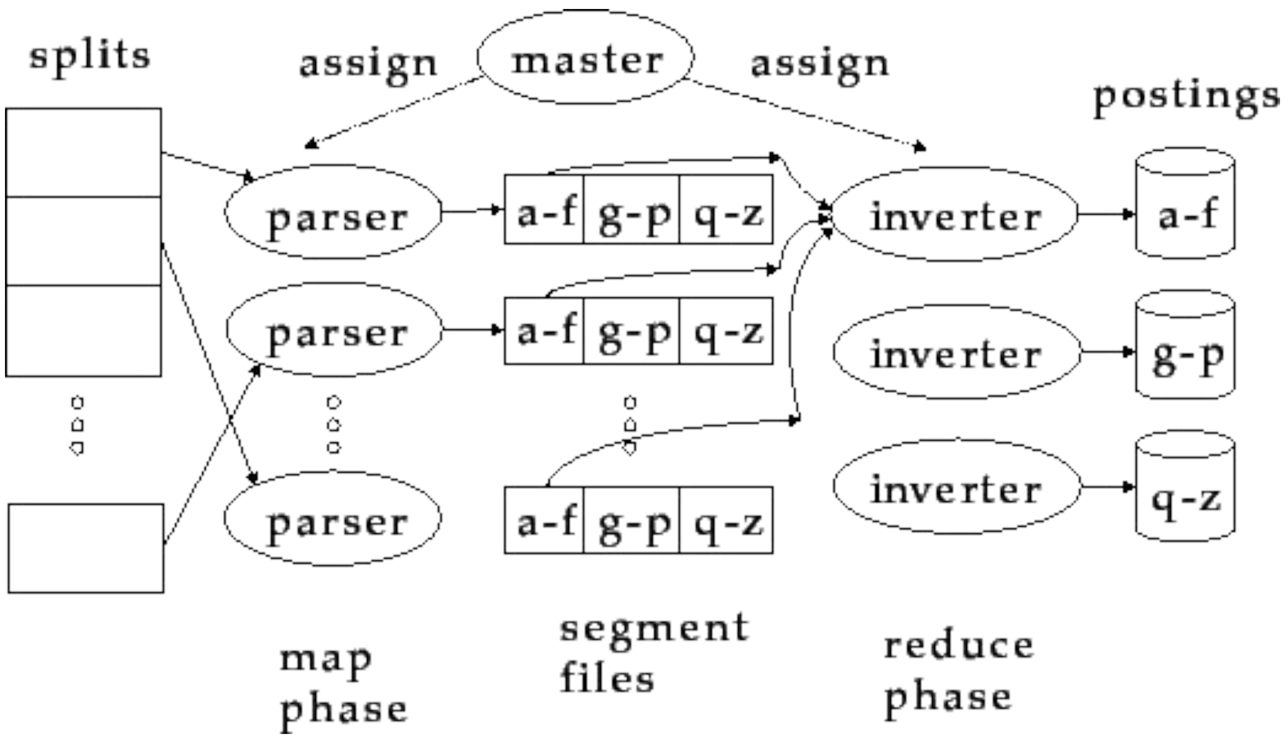
Mapreduce: Key points

- Each MAP processes only one data record at a time.
- A MAP process can generate None, one or more key values pair.
- The key for REDUCE is same as the key for MAP output.
- The REDUCE function must be order agnostic.

Mapreduce: Inverted index example

(Use board)

Mapreduce: Inverted index example (from textbook)



Mapreduce: Inverted index example (from textbook)

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)$\rangle$)

Other Applications

Is the task really data parallel?

- Recursive task (Binary search)
- Highly dependency task (fibonacci series)

There are several problems where mapreduce can be used. Example:

- Parallelizing K-means clustering.
- FInding all Maximal clique in a graph.
- Similarity between all pair of documents.

Mapreduce Implementations

- **Apache Hadoop:** https://hadoop.apache.org/
- Apache CouchDB: https://couchdb.apache.org/
- DISCO: http://discoproject.org/
- Infinispan: https://infinispan.org/

Dynamic Indexing

Dynamic Indexing

- Till now we assumed that document collection is static, i.e., they are never updated.
- However, it rarely happens. Documents are inserted, deleted and modified frequently.
- We need a way to update the index, based on the changes happened in the documents.
  - Updates in posting list.
  - Updates in the vocabulary.

Dynamic Indexing: Simple approach

- Maintain a big main index on disk.
- New documents go to a small auxiliary index in the main memory.
- For every query, search both index and aggregate results.
- Deletion:
  - Using bit vectors, one can filter outs deleted documents.
- Periodically merge auxiliary index to main index.

Dynamic Indexing: Simple approach: Issues

- Frequent merges
- Poor performance during merge.
- Merging:
  - If each posting list is stored in a single file, then merge is easier.
  - However, it can lead to too many files; inefficient for OS.

- Assumption: The index is one big file.
- In reality: the index in distributed across several files, although not one posting per file.

Dynamic Indexing: Logarithmic Merges

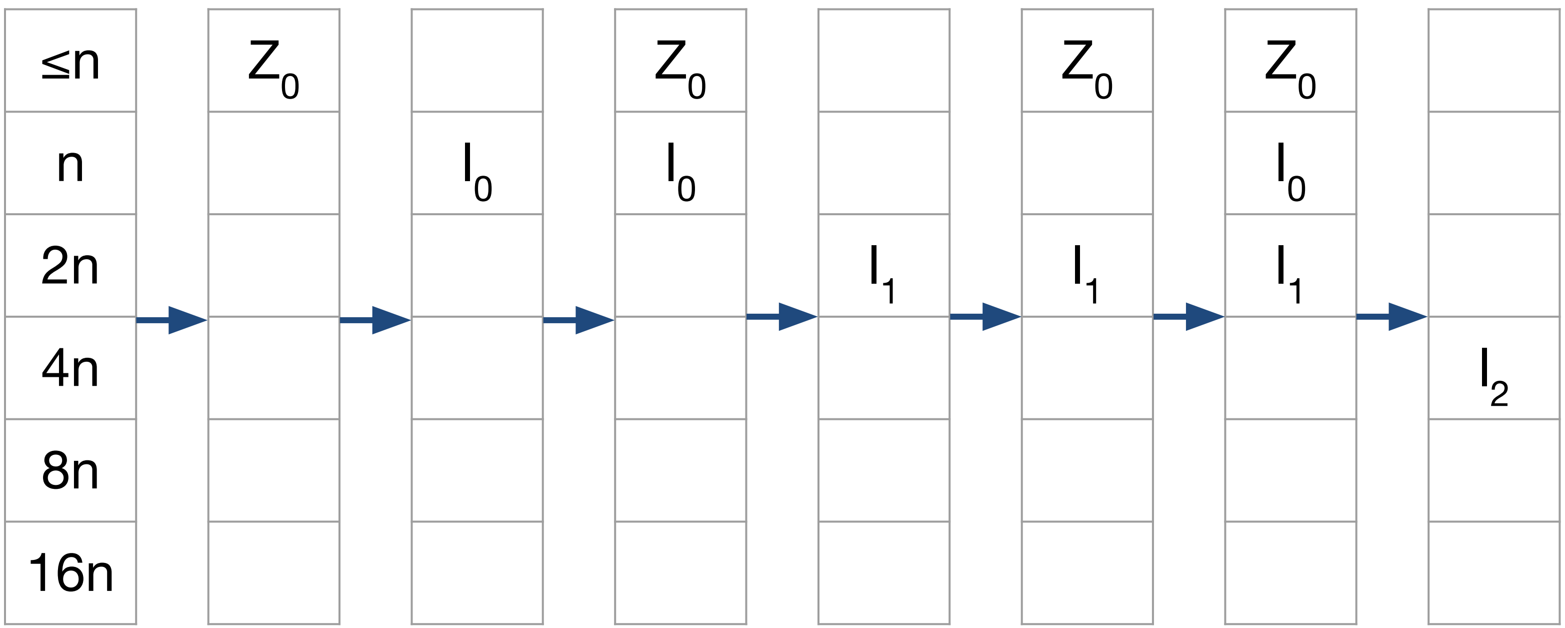
- Logarithmic merges amortize the cost of merging indexes over time.
- Maintain a series of indexes, each twice as larger than the previous index.
- Maintain a smallest index (Z_0) in memory.
- Maintain larger indexes I_0, I_1, I_2, … on disk.
- If $Z_0 > n$, either write to disk as I_0
- or merge z_0 with I_0 (is I_0 exists) and write to disk as I_1

Dynamic Indexing: Logarithmic Merges

```
LMERGEADDTOKEN(indexes, Z_0, token)
  Z_0 ← MERGE(Z_0, {token})
  if |Z_0| = n
    then for i ← 0 to ∞
           do if I_i ∈ indexes
                then Z_{i+1} ← MERGE(I_i, Z_i)
                     (Z_{i+1} is a temporary index on disk.)
                     indexes ← indexes − {I_i}
                else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
                     indexes ← indexes ∪ {I_i}
                     BREAK
         Z_0 ← ∅

LOGARITHMICMERGE()
  Z_0 ← ∅    (Z_0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

Dynamic Indexing: Logarithmic Merges: Intuition



Dynamic Indexing: Logarithmic Merges: Intuition

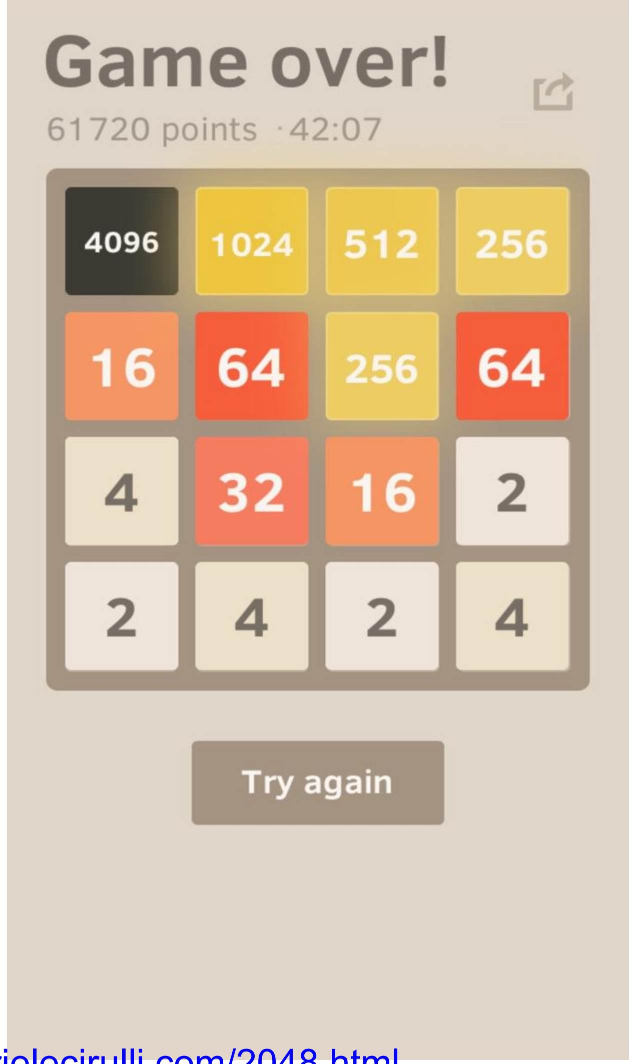


Image source: https://gabrielecirulli.com/2048.html

Dynamic Indexing: Logarithmic Merges

- Number of indexes bounded by O(log(T)), where T is total number of postings.
- So query requires O(log(T)) merges.
- Time complexity of index construction is O(T log T).
  - Because each of T postings is merged O(log T) times.

- Normal merges requires $O(T^2)$ time complexity.
  - Suppose auxiliary index has size **a**.
  - $a + 2a + 3a + 4a + \ldots + na = (an(n+1)) / 2 = O(n^2)$

Dynamic Indexing at search engines

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

- Occasionally they also reconstruct the index from scratch. However, its becomes harder, as the data size is ever increasing.
- Query processing is then switched to the new index, and the old index is deleted.

Thank You!