Information Retrieval

Index Construction

Hamid Beigy

Sharif university of technology

October 6, 2018

Table of contents

Table of contents

Inverted index

1. The goal is constructing inverted index

For each term t, we store a list of all documents that contain t.

dictionary		postings
Brutus	$\longrightarrow$	1, 2, 4, 11, 31, 45, 173, 174
Caesar	$\longrightarrow$	1, 2, 4, 5, 6, 16, 57, 132, . . .
Calpurnia	$\longrightarrow$	2, 31, 54, 101
⋮		

dictionary **postings**

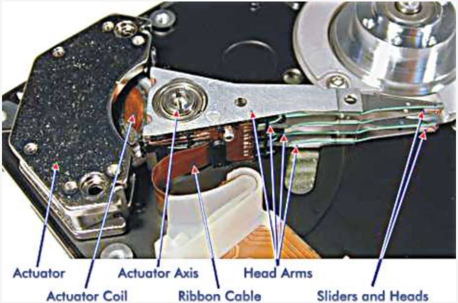
RCV1 collection

1. Shakespeare's collected works are not large enough for demonstrating many of the points in this course.
2. As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
3. English newswire articles sent over the wire in 1995 and 1996 (a year).
4. RCV1 statistics
   - Number of documents (N): 800,000
   - Number of tokens per document (L): 200
   - terms (M) : 400,000
   - Bytes per token (including spaces): 6
   - Bytes per token (without spaces): 4.5
   - Bytes per term: 7.5
5. Why does the algorithm given in previous sections not scale to very large collections?

Hardware Basics

1. Access to data is much faster in memory than on disk. (roughly a factor of 10)
2. Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
3. To optimize transfer time from disk to memory: one large chunk is faster than many small chunks.
4. Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks). Block sizes: 8KB to 256 KB
5. Servers used in IR systems typically have many GBs of main memory and TBs of disk space.
6. Fault tolerance is expensive: Its cheaper to use many regular machines than one fault tolerant machine.

Hard Disk


Actuator
Actuator Axis
Head Arms
Actuator Coil
Ribbon Cable
Sliders and Heads

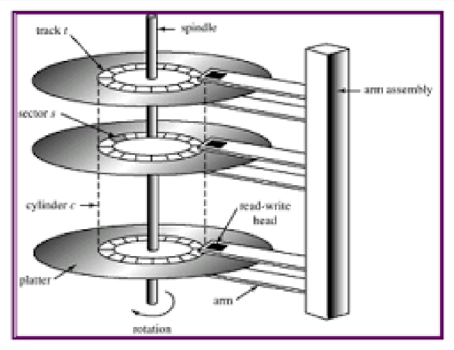

track t
spindle
arm assembly
sector s
cylinder c
read-write head
platter
arm
rotation

Table of contents

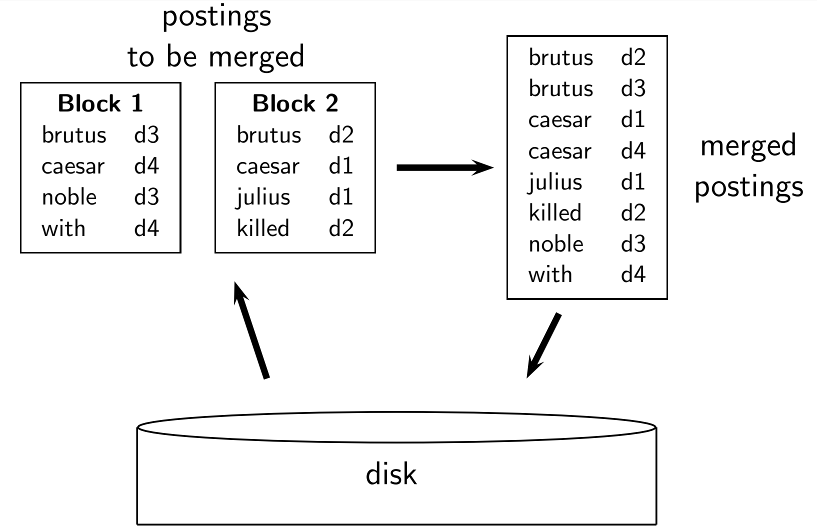
Sort-based index construction

1. As we build index, we parse docs one at a time.
2. The final postings for any term are incomplete until the end.
3. Can we keep all postings in memory and then do the sort in-memory at the end?
4. No, not for large collections
5. Thus: We need to store intermediate results on disk.
6. Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
7. No: Sorting very large sets of records on disk is too slow– too many disk seeks.
8. We need an external sorting algorithm.

External sorting algorithm

1. We must sort $T = 100{,}000{,}000$ non-positional postings.
2. Each posting has size 12 bytes (4+4+4: termID, docID, term frequency).
3. Define a block to consist of 10,000,000 such postings
4. We can easily fit that many postings into memory. We will have 10 such blocks for RCV1.
5. Basic idea of algorithm:
6. For each block do
   - accumulate postings
   - sort in memory
   - write to disk
7. Then merge the blocks into one long sorted order.

Merging two blocks



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)
```

Problem with sort-based algorithm

1. The assumption was: we can keep the dictionary in memory.
2. We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
3. Actually, we could work with term,docID postings instead of termID,docID postings . . .
4. The intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

Table of contents

Single–pass in-memory indexing (SPIMI)

1. Key idea 1: Generate separate dictionaries for each block no need to maintain term-termID mapping across blocks.
2. Key idea 2: Dont sort. Accumulate postings in postings lists as they occur.
3. With these two ideas we can generate a complete inverted index for each block.
4. These separate indexes can then be merged into one big index.

Single–pass in-memory indexing algorithm

```
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.

Single–pass in-memory indexing : compression

1. Compression makes SPIMI even more efficient.
   - Compression of terms
   - Compression of postings

Table of contents

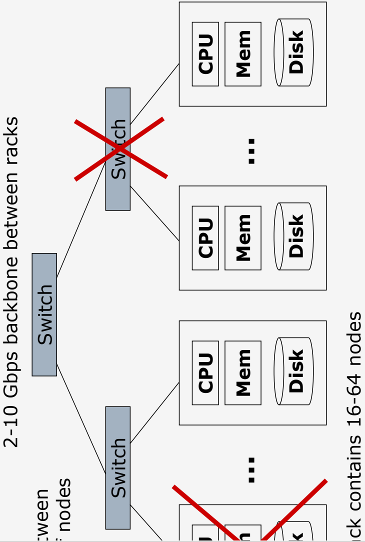
Distributed indexing

1. For web-scale indexing: must use a distributed computer cluster
2. Individual machines are fault-prone.
   Can unpredictably slow down or fail.
3. How do we exploit such a pool of machines?
4. Distributed index is partitioned across several machines - either according to term or according to document.

google data centers (Gartner estimates)

1. Google data centers mainly contain commodity machines. Data centers are distributed all over the world.
2. 1 million servers, 3 million processors/cores
3. Google installs 100,000 servers each quarter.
4. Based on expenditures of 200250 million dollars per year. This would be 10% of the computing capacity of the world!
5. If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system (assuming it does not tolerate failures)?
6. Answer: 37%
7. Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?
8. Answer: Less than two minutes.

Cluster architecture



Distributed indexing

1. Maintain a master machine directing the indexing job – considered "safe"
2. Break up indexing into sets of parallel tasks
3. Master machine assigns each task to an idle machine from a pool.

Parallel tasks

1. We will define two sets of parallel tasks and deploy two types of machines to solve them:
   Parsers and Inverters
2. Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
3. Each split is a subset of documents.

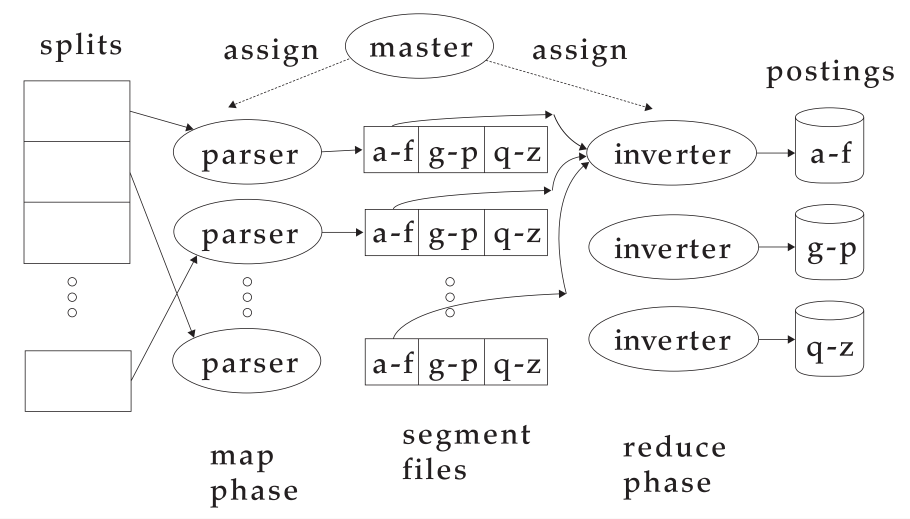
Parsers

1. Master assigns a split to an idle parser machine.
2. Parser reads a document at a time and emits (term,docID)-pairs.
3. Parser writes pairs into j term-partitions. Each for a range of terms first letters
   E.g., a-f, g-p, q-z (here: $j = 3$)

Inverters

1. An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
2. Sorts and writes to postings lists

Data flow


splits
assign
master
assign
postings
parser
a-f g-p q-z
inverter
a-f
parser
a-f g-p q-z
inverter
g-p
parser
a-f g-p q-z
inverter
q-z
map phase
segment files
reduce phase

Mapreduce

1. The index construction algorithm we just described is an instance of MapReduce.
2. MapReduce is a robust and conceptually simple framework for distributed computing . . . without having to write code for the distribution part.
3. The Google indexing system consisted of a number of phases, each implemented in MapReduce.
4. Index construction was just one phase.

MapReduce: word count example

```
map(key, value):
// key: document name; value: text of 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(result)
```

Table of contents

Dynamic indexing

1. Up to now, we have assumed that collections are static.
2. They rarely are: Documents are inserted, deleted and modified.
3. This means that the dictionary and postings lists have to be dynamically modified.

Dynamic indexing: simplest approach

1. Maintain big main index on disk
2. New docs go into small auxiliary index in memory.
3. Search across both, merge results
4. Periodically, merge auxiliary index into big index
5. Deletions:
   - Invalidation bit-vector for deleted docs
   - Filter docs returned by index using this bit-vector

Issues with auxiliary and main index

1. Frequent merges
2. Poor search performance during index merge

Logarithmic merge

1. Logarithmic merging amortizes the cost of merging indexes over time. Users see smaller effect on response times.
2. Maintain a series of indexes, each twice as large as the previous one.
3. Keep smallest (Z0) in memory
4. Larger ones (I0, I1, ...) on disk
5. If Z0 gets too big (¿ n), write to disk as I0
6. . . . or merge with I0 (if I0 already exists) and write merger to I1 etc.

Logarithmic merge

LMergeAddToken($indexes, Z_0, token$)

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

LogarithmicMerge()

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

Reading

Please read chapter 4 of Information Retrieval Book.