Information Retrieval

Weike Pan

The slides are **adapted from those provided by Prof. Hinrich Schütze** at University of Munich (http://www.cis.lmu.de/~hs/teach/14s/ir/).

Chapter 4 Index construction

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

Outline

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

4.1 Hardware basics

- Access to data is much faster in memory than on disk (roughly a factor of 10).
- Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
- To optimize the transfer time from disk to memory: one large chunk is faster than 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.
- Servers used in IR systems typically have many GBs of main memory and TBs of disk space.
- Fault tolerance is expensive: It's cheaper to use many regular machines than one fault tolerant machine.

4.1 Hardware basics

Some statistics (ca. 2008)

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

Outline

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

4.2 Blocked sort-based indexing

- Reuters RCV1 statistics (a dataset of text classification)
  - documents: N=800,000
  - tokens per document: L=200
  - terms: M=400,000
  - bytes per token (incl. spaces/punctuations): 6
  - bytes per token (without spaces/punctuations): 4.5
  - bytes per term: 7.5
  - non-positional postings: T=100,000,000
    - A posting: (termID, docID) or (termID, docID, term frequency)

- Average frequency of a term? 4.5 bytes per word token vs. 7.5 bytes per word type (i.e., term): why the difference? How many positional postings?

4.2 Blocked sort-based indexing

Sort-based index construction

- As we build index, **we parse documents one at a time**.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
  - No, not for large collections.
  - Hence, we need to store the intermediate results on disk.

4.2 Blocked sort-based indexing

Same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
  - No, sorting very large sets of records on disk is too slow - too many disk seeks.
  - We need an external sorting algorithm (i.e., sort in the main memory).

4.2 Blocked sort-based indexing

"External" sorting algorithm (using few disk seeks)

- For the RCV1 data, we must sort T = 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, term frequency) or 8 bytes (4+4: termID, docID).

- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into the main memory.
  - We will have 10 such blocks for the RCV1 data.

4.2 Blocked sort-based indexing

- **Basic idea**
  - For each block:

    (i) accumulate postings

    (ii) sort in the main memory w.r.t. termID

    (iii) write to disk

  - Then merge the blocks into one.

4.2 Blocked sort-based indexing

Algorithm: Blocked Sort-Based Indexing

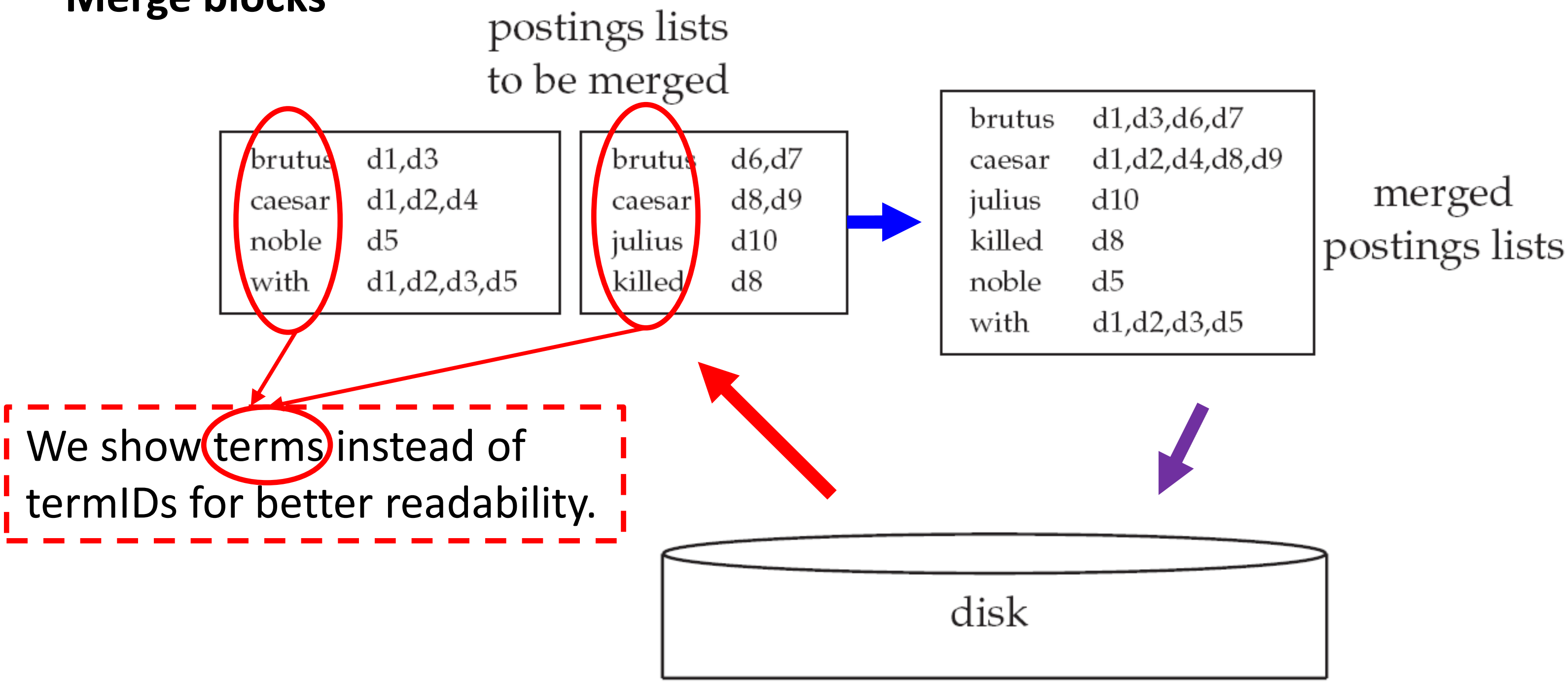
```
BSBIndexConstruction()
  n ← 0
  while (all documents have not been processed)
  do n ← n + 1
       block ← ParseNextBlock()
       BSBI-Invert(block)
       WriteBlockToDisk(block, f_n)
  MergeBlocks(f_1, ..., f_n; f_merged)
```

n: the nth block

- BSBI-Invert(block)
  - **Sort** the (termID, docID) pairs w.r.t. termID
  - Collect all (termID, docID) pairs with the same termID into a postings list

4.2 Blocked sort-based indexing

Merge blocks



Merging in blocked sort-based indexing (BSBI). Two blocks ("postings lists to be merged") are **loaded from disk into memory**, **merged in memory** ("merged postings lists") and **written back to disk**.

Outline

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

4.3 Single-pass in-memory indexing

Problem with sort-based algorithm

- Our assumption: We can keep the dictionary in memory, because 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).

4.3 Single-pass in-memory indexing

Single-pass in-memory indexing (SPIMI)

- Key idea 1(第一个想法): Generate **separate dictionaries** for each block. Then, we do not need to maintain term-termID mapping across blocks.
  Notes: each separate dictionary is **NOT** used to do term-termID mapping.

- Key idea 2(第二个想法): **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.

4.3 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
```

token_stream refers to the (term, docID) pairs

One separate dictionary for each block

sorted_terms instead of **sorted_termIDs**, because there is a separate dictionary for each block.

Notes: Merging of blocks is very similar to that of BSBI.

4.3 Single-pass in-memory indexing

Exercise

- What is the difference between the **sort** in BSBI and the **sort** SPIMI?

4.3 Single-pass in-memory indexing

SPIMI: Compression

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

  - See Chapter 5

Outline

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

4.4 Distributed indexing

- For web-scale indexing: must use a distributed computer cluster
- Individual machines are fault-prone. Can unpredictably slow down or fail.
- How do we exploit such a pool of machines?
  - Maintain a master machine directing the indexing job - considered "safe"
  - Break up indexing into sets of parallel tasks
  - The master machine assigns each task to an idle machine from a pool

4.4 Distributed indexing

Parallel tasks

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

4.4 Distributed indexing

Parsers

- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term, docID) pairs.
- 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)

4.4 Distributed indexing

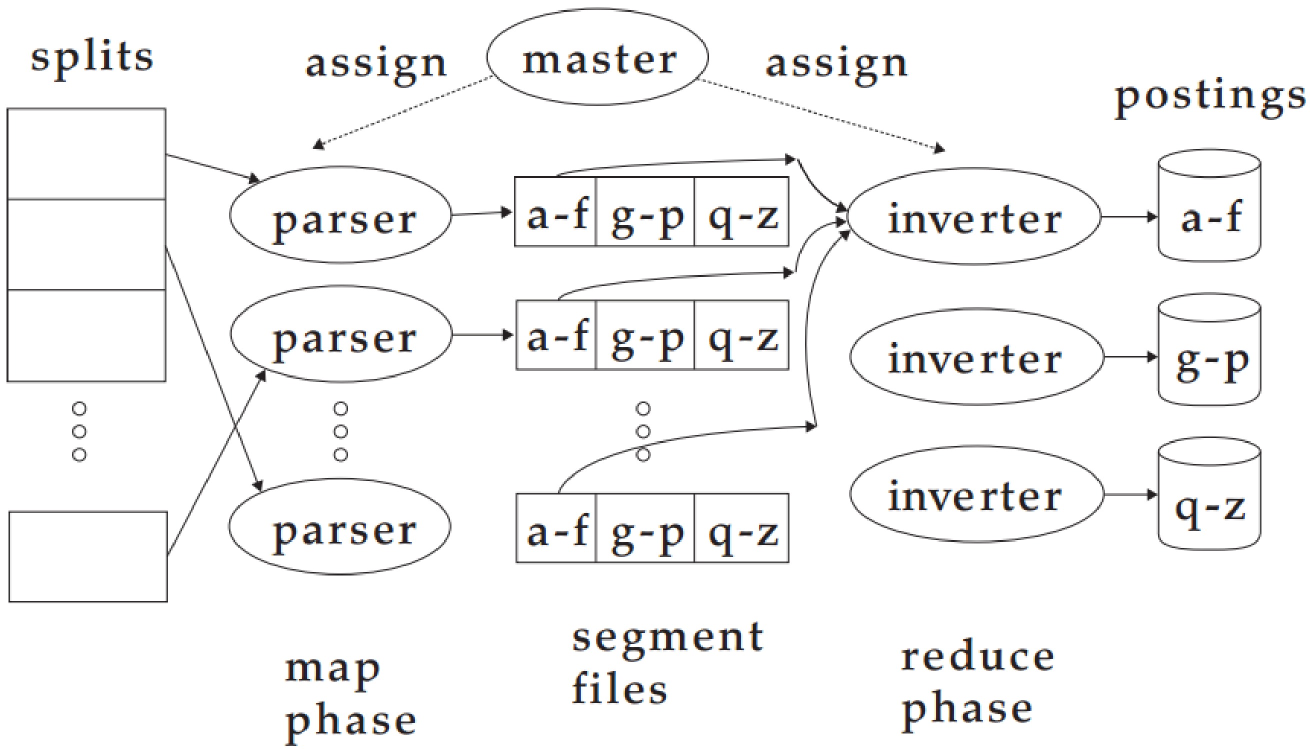
Inverters

- An inverter collects all (term, docID) pairs (i.e., postings) for one term-partition (e.g., for a-f).

- Sorts and writes to postings lists

4.4 Distributed indexing

Data flow



4.4 Distributed indexing

Map/Reduce

- The index construction algorithm we just described is an instance of Map/Reduce.
- Map/Reduce is a robust and conceptually simple framework for distributed computing... without having to write code for the distribution part.
- The Google indexing system consisted of a number of phases, each implemented in Map/Reduce.
  - Index construction was just one phase.
  - Another phase: transform term-partitioned into document-partitioned index.

4.4 Distributed indexing

Index construction in Map/Reduce

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:	$(\langle \text{termID}_1, \text{list(docID)}\rangle, \langle \text{termID}_2, \text{list(docID)}\rangle, \ldots)$	$\rightarrow$ $(\text{postings\_list}_1, \text{postings\_list}_2, \ldots)$

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow (\langle \text{C}, d_2\rangle, \langle \text{DIED}, d_2\rangle, \langle \text{C}, d_1\rangle, \langle \text{CAME}, d_1\rangle, \langle \text{C}, d_1\rangle, \langle \text{C'ED}, d_1\rangle)$
reduce:	$(\langle \text{C},(d_2,d_1,d_1)\rangle,\langle \text{DIED},(d_2)\rangle,\langle \text{CAME},(d_1)\rangle,\langle \text{C'ED},(d_1)\rangle)$	$\rightarrow (\langle \text{C},(d_1{:}2,d_2{:}1)\rangle,\langle \text{DIED},(d_2{:}1)\rangle,\langle \text{CAME},(d_1{:}1)\rangle,\langle \text{C'ED},(d_1{:}1)\rangle)$

Outline

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading

4.5 Dynamic indexing

Dynamic indexing: Simplest approach

- Maintain a big main index on disk
- New docs go into a small auxiliary index in the main memory
- Search across both, merge results
- Periodically, merge auxiliary index into big index

- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

4.5 Dynamic indexing

Issue with auxiliary and main index

- Frequent merges
- Poor search performance during index merge

4.5 Dynamic indexing

Logarithmic merge (1/3)

- Logarithmic merging amortizes the cost of merging indexes over time -> Users see smaller effect on response times.

- Maintain a series of indexes, each twice as large as the previous one.
  - Keep smallest (Z0) in memory
  - Larger ones (I0, I1, ...) on disk
  - If Z0 gets too big (>n), write to disk as I0 ... or merge with I0 (if I0 already exists) and write merger to I1, etc.

4.5 Dynamic indexing

Logarithmic merge (2/3)

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

$Z_0 \leftarrow \textsc{Merge}(Z_0, \{\mathit{token}\})$
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
      do if $I_i \in \mathit{indexes}$
          then $Z_{i+1} \leftarrow \textsc{Merge}(I_i, Z_i)$
              (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.)*
              $\mathit{indexes} \leftarrow \mathit{indexes} \cup \{I_i\}$
              $\textsc{Break}$
      $Z_0 \leftarrow \emptyset$

n is the size of the auxiliary index

$\textsc{LogarithmicMerge}()$

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

4.5 Dynamic indexing

Logarithmic merge (3/3)

- Number of indexes bounded by O(log T/n), where n is the size of the auxiliary index and T is the total number of postings.

- So, query processing requires the merging of O(log T/n) indexes.

- Time complexity of index construction is O(T log T/n) because each of T postings is merged O(log T/n) times.

Summary

- 4.1 Hardware basics
- 4.2 Blocked sort-based indexing
- 4.3 Single-pass in-memory indexing
- 4.4 Distributed indexing
- 4.5 Dynamic indexing
- 4.6 Other types of indexes
- 4.7 References and further reading