Inverted Index Implementation

Adam Hadraba

hadraba.adam@gmail.com

Table of contents

- **Introduction**
- **Implementation**
- **Construction**
- **Compression**

Introduction

- **Inverted index is structure that provides background for:**
  - Processing large number of queries over massive amount of data each second
  - "Fast" response & dealing w/ hardware limitations
  - Different kinds of queries
  - Data set changes
  - Ranking results

Inverted index

- **Most common indexing method used in IR systems**
- **Way to avoid linearly scanning the texts**
  - **Index in advace**
- **Widely used in search engines**

- **Normally, documents – lists of words**
  - **Inverted index — for each word lists of documents**

Inverted index cont.

Dictionary	
Term	**DocFreq**
able	20
about	5
above	7
...	

Posting lists					
docID	**docID**	**docID**	**docID**	**docID**	**...**
1	4	5	7	12	...
2	4	5	6	10	...
5	8	12	25	100	...
...					

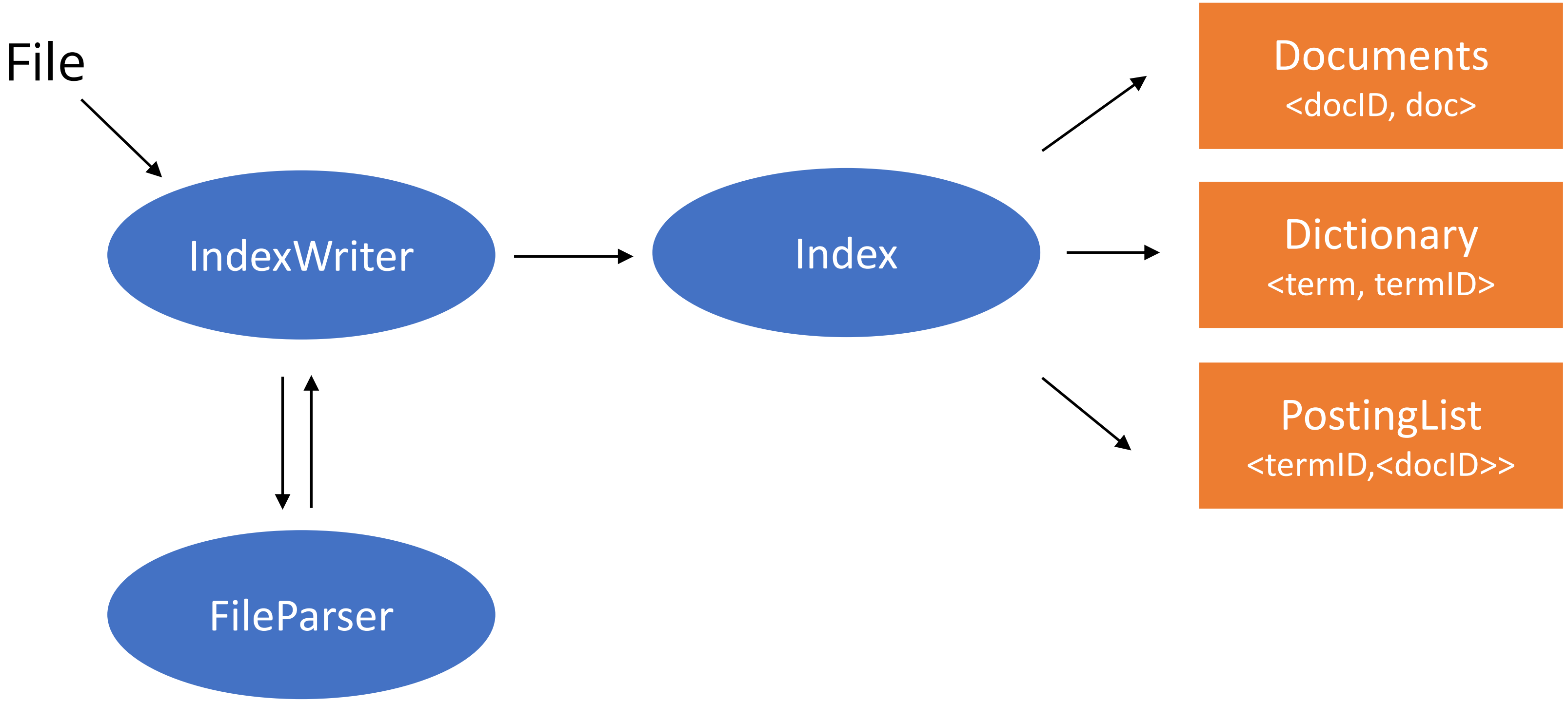
Data

- Profimedia database:
  - *20 000 000 documents*
  - *Each document around 50 annotations in english*
  - *360 000 unique annotations*
  - *Approx. size of 4.5 GB*
  - *Terms occured in 1 to 3 500 000 documents*
- No stop words
- Annotations unique per document (no need for term frequency)
- No position of term in document

Implementation

- IR engine has two main components
  - Indexing documents
  - Query processing
- Requirements
  - Scalability
    - Handling big collection of data
  - Index efficiency
    - Index must be constructed in reasonable amount of time
  - Query efficiency and effectivness
    - Queries must run fast and the result set must be relevant

Indexing documents

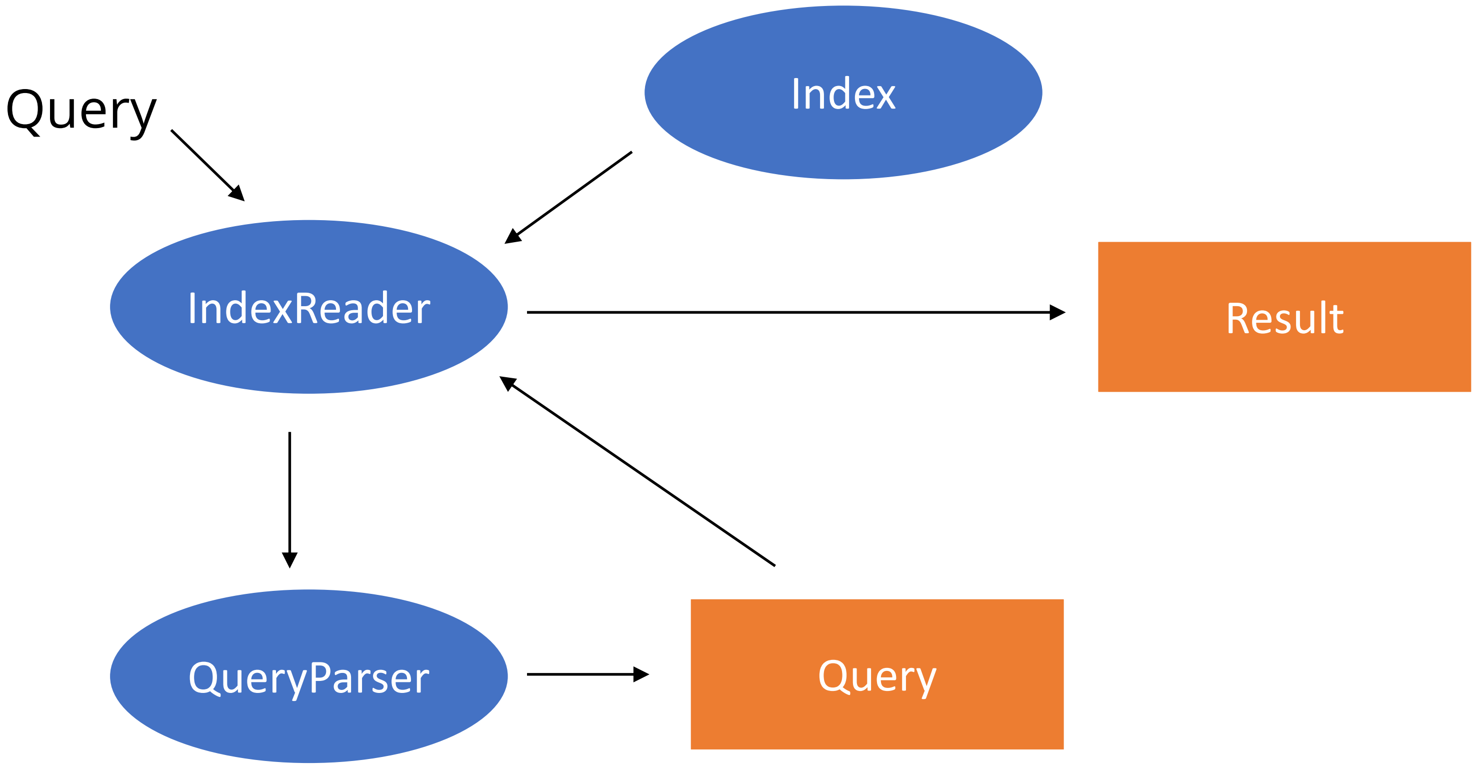


Indexing documents cont.

- Dictionary <term, termID>
  - Stored in memory as a HashMap
  - Serves as lookup structure on top of the posting lists
- PostingList <termID, <docID>>
  - Majority of data are here – stored on disc
  - During query processing are the query term‘s loaded into memory
- Documents <docID, doc>
  - Stored on disc

- Other supporting structures – offsets and skip lists to increase lookup efficiency

Query processing



Index Construction

Possible approaches

- Memory-based
  - For each document indentify distinct terms and update posting list for each term in memory
  - Pro: very fast algorithm, easy to implement
  - Con: Does not work when you run out of memory
- Blocked sort-based (sort-inversion)
  - Sorting
  - Merging (2-way, multi-way)
- Single-pass in-memory indexing
  - When dictionary does not fit into memory
  - Each block has own dictioary

Disc-based index construction

- **Phase I**
  - **Create temp files of pairs <termID, docID>**

Run 1:

termID	docID
1	1
5	4
2	5
1	5
6	5
2	6
2	2

Run 2:

termID	docID
4	5
2	4
1	7
3	1
7	6
3	5
2	8

Disc-based index construction cont.

- **Phase II**
  - Sort the pairs in each run

Run 1:

termID	docID
1	1
1	5
2	2
2	5
2	6
5	4
6	5

Run 2:

termID	docID
1	7
2	4
2	8
3	1
3	5
4	5
7	6

Disc-based index construction cont.

- **Phase III**
  - **Merge sorted temp files (2-way, multi-way)**

Run 1:

termID	docID
1	1
1	5
2	2
…	

Run 2:

termID	docID
1	7
2	4
2	8
…	

Temporary set:

termID	docID
1	1
1	7

→ **Output buffer**

Disc-based index construction cont.

- **Phase III**
  - **Merge sorted temp files (2-way, multi-way)**

Run 1:

termID	docID
1	5
2	2
…	

Temporary set:

termID	docID
1	5
1	7

Output buffer

Run 2:

termID	docID
1	7
2	4
2	8
…	

Disc-based index construction cont.

- **Phase IV**
  - Read all pairs for a given term
  - Construct a posting list (compress it)
  - Write it to file

Output buffer:

termID	docID
1	1
1	5
1	7
…	

→

termID	docFr	docID	docID	docID	docID
1	3	1	5	7	…
…					

Disc-based index construction cont.

- **Pro**
  - **Scalable**
- **Con**
  - **Not fast as memory-based approach**
  - **Requires twice the amount of disk space as the size of original text**

Size of index without compression

- **Dictionary – 5.8 MB**
- **Posting lists – 3.7 GB**
- **Documents – 400 MB**

Approximately 90 % of original size.

Dynamic indexing

- **Untill now, we assumed that collections are static**

- **New documents need to be iserted**
- **Documents are deleted and modified**

► **Postings updates for terms already in dictionary**

► **New terms added to dictionary**

Dynamic indexing

- **“Big” main index**
- **New documents go into „small“ auxiliary index**
- **Search across both, merge results**
- **Deletions**
  - Invalidation bit-vector
- **Periodically, re-index into one main index**

Index Compression

Index Compression

- **Why?**
  - Less disc space consumption
    - Compression ratios of 1:4 are easily achievable
  - Increased use of caching
    - Usually, we are caching frequently used parts of posting lists into the memory
    - With compression we can fit a lot more into memory
  - Faster transfer of data from disc to memory
    - Reduction of I/O
    - It is usually faster to transfer compressed posting list and then to decompress it, rather than transferring uncompressed posting list

Posting list compression

- **DocIDs are ordered in posting list**
  - ► **Replace DocID by the interval difference $DocID_i - DocID_{i-1}$**

- **Then encode interval difference – fewer bits for smaller, common numbers**

...	docID	docID	docID	docID	...
...	256454	256460	256475	256478	...

↓

...	ΔdocID	ΔdocID	ΔdocID	...
...	6	15	3	...

Compression techniques

- **VBytes – Simple and good, but we can do better**
- **Elias' Gamma/Delta Code, Rice Coding, Golomb Coding – good compression for very small numbers, but slow**
- **Simple9 (Anh/Moffat 2001), PFOR-DELTA (Heman 2005) – compression done in chunks – more numbers at a time**

Var-Byte compression

- **Simple byte-oriented method for encoding data**
  - If < 128, use 1 byte (highest bit set to 0)
  - If < 128 × 128 = 16384, use 2 bytes (first highest bit 1, the other 0)
  - If < 128 × 128 × 128, use 3 bytes (first highest bit 1, second 1, last 0)

- **Example:** 14169 = (110 × 128) + 89 = | 11101110 | 01011001 |

We covered

- **What is inverted index**
- **Simple system overview**
- **Index construction**
- **Compression**

Thank You.

Questions?