POLITECNICO DI MILANO

Inverted Indexing

Luca Bondi

Inverted Indexing
Data Structures for IR

- Efficient data structures needed to process large document collections quickly

- How do we store documents in order to maximize retrieval performance?

  - We must avoid linear scans of text (e.g. `grep` commend) at query time

  - We must index documents in advance

Inverted Indexing
Term-document incidence matrix

- Naïve data structure: term-document incidence matrix

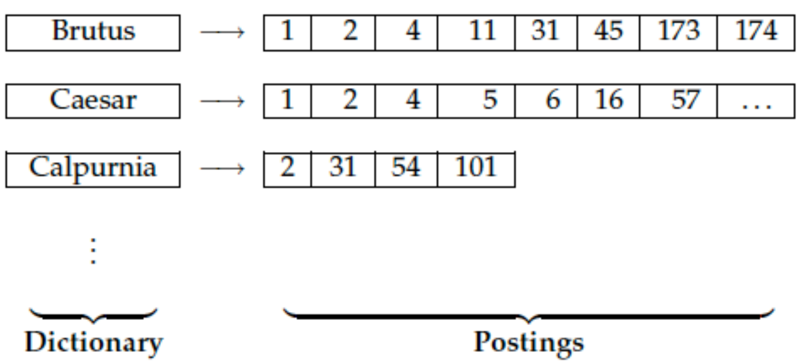
	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

- Is it feasible for large document collections?
  - Consider $N = 10^6$ documents, each with about $1K$ terms
  - Avg. 6 bytes/term including spaces/punctuation
  - $6GB$ of data in the documents
  - Suppose there are $M = 500K$ distinct terms among the documents

Inverted Indexing
Term-document incidence matrix

- $500K \times 1M$ matrix has half-a-trillion 0's and 1's
  - But it has no more than one billion 1's
  - Matrix is extremely sparse

- What's a better representation?
  - We only record the 1's positions → inverted index



Inverted Indexing
Inverted index

- For each term, we have a list that records in which documents the term occurs
  - Each term in the list is conventionally called a posting
  - A posting is a tuple of the form (t_i, d_j), where t_i is a term identifier and d_j is a document identifier
  - The list is called a posting list (or inverted list)
- All the postings list taken together are referred to as the postings

Inverted index

- Inverted indexes are independent from the adopted IR model (Boolean model, vector space model, etc.)

- Each posting usually contains:
  - The identifier of the linked document
  - The frequency of appearance of the term in the document
  - The position of the term for each document (optional)
    - Expressed as number of words from the begin of the document, number of bytes, etc.
    - A.k.a. positional posting

- For each term is also usually stored the frequency of appearance of the term in the dictionary

Building an inverted index

- To gain the speed benefits of the indexing at retrieval time, we have to build the index in advance:
  1. Collect the documents to be indexed
     - “Friends, Romans, countrymen…”
     - “So let it be with Caesar…”
     - …
  2. Tokenize the text, turning each document into a list of tokens
     - |Friends|Romans|countrymen|So|…
  3. Linguistic pre-processing. Each document is represented as a list of normalized tokens
     - |friends|romans|countrymen|so|…
  4. Index the documents that each token occurs in by creating an inverted index, consisting of a dictionary and postings

Building an inverted index

- Within a document collection, we assume that each document has a unique serial number, the document identifier (docID)

- Indexing process:
  1. Input: list of normalized tokens for each document
  2. Sort the terms alphabetically
  3. Merge multiple occurrences of the same term
  4. Record the frequency of occurrence of the term in the document
     - not needed by Boolean models
     - used by vector space models
  5. Group instances of the same term and split dictionary and postings

Inverted Indexing
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:"

1. Sequence of (token, Document ID) pairs.

term	docID
I	1
did	1
enact	1
julius	1
casear	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

Inverted Indexing
Example

term	docID
I	1
did	1
enact	1
julius	1
casear	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

2. Sort by term

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

3. Merge multiple term entries

4. Add frequency information

term	docID	term freq.
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Inverted Indexing
Example

term	docID	freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

5. Split results in dictionary and posting

term	coll. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	3	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	2	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	2	→	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

Each posting may store other information such as the term frequency in each document and the positions of the term in the document

Inverted Indexing
Processing Boolean queries

- Consider processing the Boolean query: $q = t_a \wedge t_b$
  - Locate t_a in the dictionary
  - Retrieve its postings
  - Locate t_b in the dictionary
  - Retrieve its postings
  - Intersect the two postings

t_a: 2 → 4 → 8 → 16 → 32 → 64 → 128

t_b: 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Inverted Indexing
Processing Boolean queries

- The intersection operation is the crucial one: we need to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms

- If the list lengths are n_1 and n_2, then it takes $O(n_1 + n_2)$ operations.
- Assumption: postings sorted by docID

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
        then ADD(answer, docID(p1))
             p1 ← next(p1)
             p2 ← next(p2)
        else if docID(p1) < docID(p2)
               then p1 ← next(p1)
               else p2 ← next(p2)
  return answer
```

Processing Boolean queries

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

t_a: 2 → 4 → 8 → 16 → 32 → 64 → 128

t_b: 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

⇒ 2 → 8

- We would like to be able to extend the intersection operation to process more complicated queries like

$$t_a \wedge (t_b \vee \neg t_c)$$

- This is accomplished by means of query optimization
  - how to organize the work of answering a query so that the least amount of work needs to be done by the system

Inverted Indexing
Storing an inverted index

- Terms generally occurs in a number of documents
  - inverted indexes reduce the storage requirements of the index
  - provide the basis for efficient retrieval
  - this inverted index structure is essentially without rivals as the most efficient structure for supporting text search
- Linked lists generally preferred to arrays
  - Dynamic space allocation
  - Insertion of terms into documents is easy
  - Space overhead of pointers
- Space requirements
  - The size of the dictionary grows according to Heap's law $O(n^{\beta})$
    - $n \rightarrow$ space occupied by the document collection
    - $n = 1GB \rightarrow$ dictionary $\approx 5MB$
    - Usually kept in memory
  - Postings are usually larger $O(n)$
    - Normally kept on disk

Dictionary compression

- Dictionary size < postings size
  - Why compressing the dictionary?
  - Main determinant of IR system's response time is the number of disk seeks required to answer a query
    - If part of the dictionary is on the disk, then more disk access are required for query evaluation

- The dictionary is compressed in order to fit into the main memory
  - Large enterprise systems with terabytes of documents in many different languages
  - Search in low-end devices (e.g. smartphones)
  - Fast start-up time
  - …

Dictionary compression

- Dictionary as a string
  1. Simplest data structure: store the lexicographically ordered list of all terms in an array of fixed-width entries
     - Assuming UNICODE → $2 \cdot 20$ bytes for each term (20 character words), 4 bytes for its frequency, 4 bytes for the pointer to the posting file (4GB address space)
     - Dictionary size can grow in the order of dozen of MB
     - Fixed-width entries are wasteful
       – Average length of a word in English is 8 characters

  2. Dictionary terms are stored as a long string of characters
     - We add term pointers to locate terms in the string (~ 3 bytes)
     - Dictionary size can be reduced to half

Dictionary compression

3. Blocked storage
   - Grouping terms in the string into blocks of k terms and keeping a pointer only for the first term of the block
   - The length of the term is stored into the string as additional byte
   - We eliminate $k-1$ term pointers, but we need additional k bytes for storing the length of each term
   - Bigger block size → better compression but slower performances

Inverted Indexing
Posting compression

- Empirical observation: posting for frequent terms are close together

	encoding	posting list								
the	docIDs	...	283042		283043		283044		283045	...
	gaps			1		1		1		...
computer	docIDs	...	283047		283154		283159		283202	...
	gaps			107		5		43		...
arachnocentric	docIDs	252000	500100							
	gaps	252000	248100							

- Idea: gaps between documents are short, requiring a lot less space to store
  - And gaps for rare terms have the same magnitude as docIDs

- For an economical representation of the distribution of gaps we need variable encoding methods

- Additional information [Manning et al., 2008, Chapter 5]

Searching in an inverted index structure

1. Access the dictionary file and search for the query terms

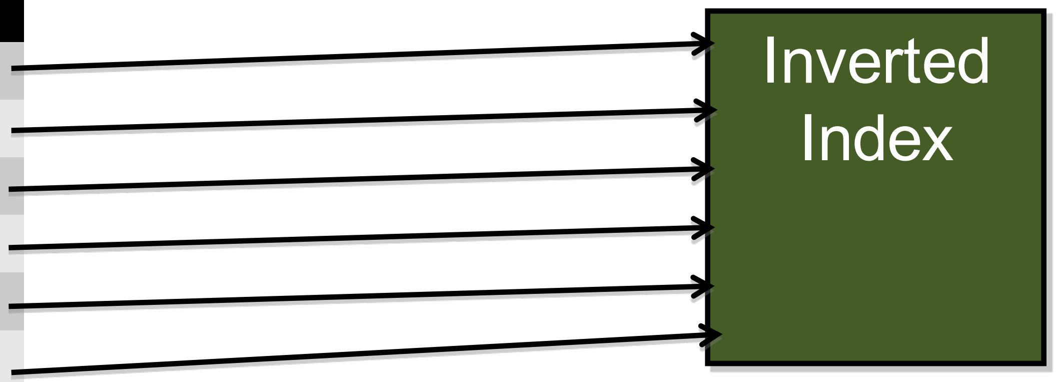
2. Retrieve the posting files for each term

3. Filter results: if the query is composed by several terms (possibly connected by logical operators), partial result lists must be fused in a final list

4. If proximity operators are specified in the query (and the inverted index adopts a blocked storage), posting files must be looked up to get the offset of terms

Searching the dictionary: Linear index

TERM	POINTER TO POSTING FILE
ant	
bee	
cat	
dog	
elk	
fox	
gnu	
…	



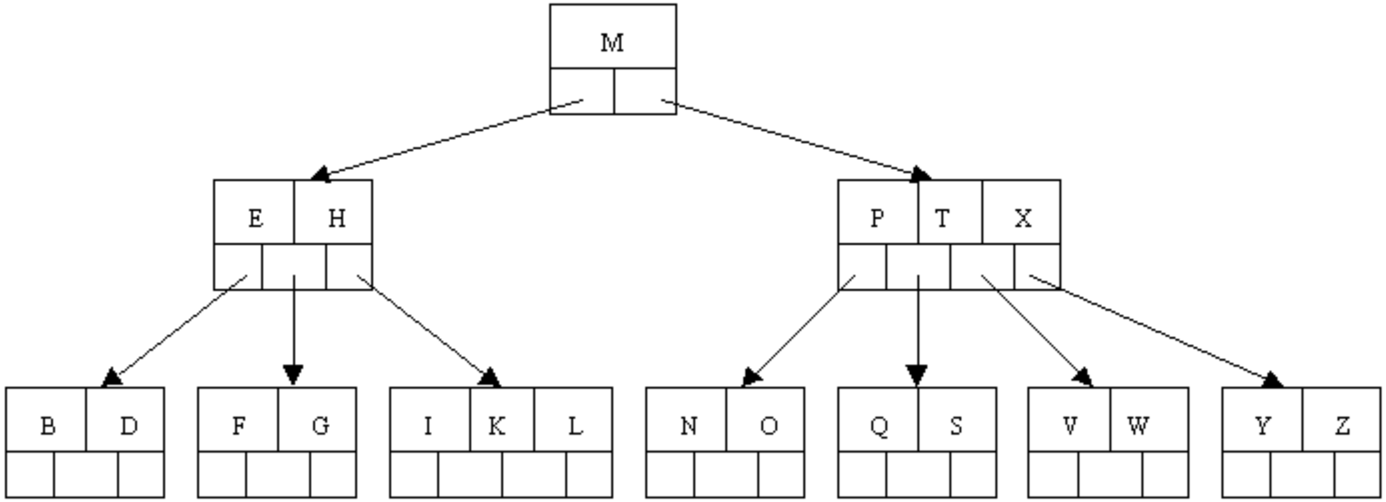
- Pros
  - Low access time – e.g., binary search $O(\log n)$
  - Effective for sequential evaluation
  - Low space occupation
- Cons
  - Indexes must be re-built at every insertion of new documents

Searching the dictionary: B-tree index

- B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:
  - Every node has at most m children
  - Every node (except root and leaves) has at least $m/2$ children
  - The root has at least two children
  - All leaves appear in the same level, and carry information
  - A non-leaf node with k children contains $k-1$ keys

Searching the dictionary: B-tree index

- Each internal node's elements act as separation values which divide its sub-trees
  - if an internal node has 3 child nodes (or sub-trees) then it must have 2 separation values of elements a_1 and a_2
  - All values in the leftmost sub-tree will be less than a_1
  - All values in the middle sub-tree will be between a_1 and a_2
  - All values in the rightmost sub-tree will be greater than a_2



Searching the dictionary: B-tree index

- Pros
  - Really low access time
    - The maximum number of access for a B-tree of order m is $O(\log_m n)$, where $\log_m n$ is the height of the B-tree
  - Effective for updates and insertion of new terms
  - low space occupation

- Cons
  - Poor performances in sequential search (B+ trees might help)
  - It gets unbalanced after many insertions (rebalancing procedures are required)

References

- [Manning et al., 2008] C.D. Manning, P. Raghavan and H. Schütze, “Introduction to Information Retrieval”, Cambridge University Press, 2008 (http://nlp.stanford.edu/IR-book/)