Indexing

March 21, 2015

1 Documents and query representation

* term incidence matrix
    * About retrieval Models

1.1 bag of words representation

* TF, DF, DLength, AVG(DLength), V, N, IDF
    * subsection what and how to get from index
    Q: can i write from other places like wiki or something

2 Preprocessing

In Information Retrieval, it is often necessary to interpret natural text where a a large amount of text has to be interpreted, so that it is available as a full text search and is represented efficiently in terms of both space (document storing) and time (retrieval processes) requirements.

It can also be regarded as : process of incorporating a new document into an information retrieval system.

2.1 Tokenization

Tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. Input: "John_DavenPort #person 52 years_old #age"

John | DavenPort | person | 52 | years | old | age

2.2 Stopwords

Stopwords refer to the words that have no meaning for "Retrieval Purposes". E.g.

· **Articles** : a, an, the, etc.

· **Prepositions** : in, on, of, etc.

· **Conjunctions** : and, or, but, if, etc

· **Pronouns** : I, you, them, it, etc

· **Others** : some verbs, nouns, adverbs, adjectives (make, thing, similar, etc.).

Stopwords can be up to 50% of the page content and not contribute to any relevant information w.r.t. retrieval process. Removal of these can improve the size of the index considerably. Sometimes we need to be careful in terms of words in phrases! e.g.: Library of Congress, Smoky the Bear!

Word	Occurrences	Percentage
the	8,543,794	6.8
of	3,893,790	3.1

Q: There's more to these word/occurrences, if the format looks OK, I'll add?

2.3 Stemming

Stemming is commonly used in Information Retrieval to conflate morphological variants. Typical stemmer consists of collection of rules! and/or dictionaries! Similar approach may be used in other languages too!

e.g.: The following stem to the word as shown below:

servomanipulator ← servomanipulators servomanipulator
logic ← logical logic logically logics logicals logicial logicially
login ← login logins
microwire ← microwires microwire
knead ← kneaded kneads knead kneader kneading kneaders

Q : added a section to give a classic stemmer example and example of stemming

2.3.1 Porter Stemmer

Q: should I add Porter stemmer desc?

2.3.2 Stemming Example

Original text	Porter Stemmer
Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales	market strateg carr compan agricultur chemic report predict market share chemic report market statist agrochem pesticid herbicid fungicid insecticid fertil predict sale stimul demand price cut volum sale

2.4 Term Positions

3 Index Construction

A reasonably-sized index of the web contains many billions of documents and has a massive vocabulary. Search engines run roughly 105 queries per second over that collection. We need fine-tuned data structures and algorithms to provide search results in much less than a second per query. O(n) and even O(log n) algorithms are often not nearly fast enough. The solution to this challenge is to run an inverted index on a massive distributed system.

Text search has unique needs compared to, e.g., database queries, and needs its own data structures' primarily, the inverted index.

· **Forward Index** : A forward index is a map from documents to terms (and positions). These are used when you search within a document.

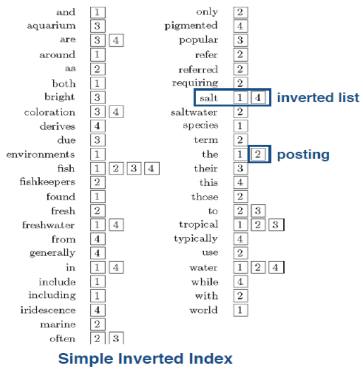
· **Inverted Index** : An inverted index is a map from terms to documents (and positions). These are used when you want to find a term in any document.

3.1 Inverted lists and catalog/offset files

In an inverted index, each term has an associated inverted list. At minimum, this list contains a list of identifiers for documents which contain that term. Usually we have more detailed information for each document as it relates to that term. Each entry in an inverted list is called a posting. Document postings can store any information needed for efficient ranking. For instance, they typically store term counts for each document - tf $_{w,d}$.

Depending on the underlying storage system, it can be expensive to increase the size of a posting. It's important to be able to efficiently scan through an inverted list, and it helps if they're small.

Inverted Indexes are primarily used to allow fast, concurrent query processing. Each term found in any indexed document receives an independent inverted list, which stores the information necessary to process that term when it occurs in a query. Next, we'll see how to process proximity queries, which involve multiple inverted lists.



Simple Inverted Index

3.2 Memory Structure, and limitations

Given a collection of documents, how can we efficiently create an inverted index of its contents? The basic steps are:

1. Tokenize each document, to convert it to a sequence of terms.
2. Add doc to inverted list for each token.

This is simple at small scale and in memory, but grows much more complex to do efficiently as the document collection and vocabulary grow.

The basic indexing algorithm will fail as soon as you run out of memory. To address this, we store a partial inverted list to disk when it grows too large to handle. We reset the in-memory index and start over. When we're finished, we merge all the partial indexes. The partial indexes should be written in a manner that facilitates later merging. For instance, store the terms in some reasonable sorted order. This permits merging with a single linear pass through all partial lists.

There are multiple ways as given below by which a Inverted Index can be built.

3.2.1 Option 1: Multiple Passes

Making multiple passes through the document collection is the key here. In each pass, we create the inverted lists for the next 1,000 terms, each in its own file. At the end of each pass, we concatenate the new inverted lists onto the main index file. This process is easy to concatenate the inverted files, but have to manage the catalog/offsets files.

3.2.2 Option 2: Partial inverted lists

Create partial inverted lists for all terms in a single pass through the collection is what happens in this process. As each partial list is filled, append it to the end of a single large index file. When all documents have been processed, run through the file a term at a time and merge the partial lists for each term. This second step can be greatly accelerated if you keep a list of the positions of all the partial lists for each term in some secondary data structure or file.

3.2.3 Option 3: preallocate the right amount of space

We could have dis-contiguous postings also. Here, we lay our index file as a series of fixed-length records of, say, 4096 bytes each. Each record will contain a portion of the inverted list for a term. A record will consist of a header followed by a series of inverted list entries. The header will specify the term_id, the number of inverted list entries used in the record, and the file offset of the next record for the term. Records are written to the file in a single pass through the document collection, and the records for a given term are not necessarily adjacent within the index.

3.3 Updating an inverted index

If each term's inverted list is stored in a separate file, updating the index is straightforward: we simply merge the postings from the old and new index. However, most filesystems can't handle very large numbers of files, so several inverted lists are generally stored together in larger files. This complicates merging, especially if the index is still being used for query processing. There are ways to update live indexes efficiently, but it's often simpler to simply write a new index, then redirect queries to the new index and delete the old one.

4 Other things to store in the index

5 Proximity Search

virgil

$\sum_{i=1}^{5} a_i$

6 Compression

*probabilities as matching evidence

6.1 Basics of Compression, Entropy

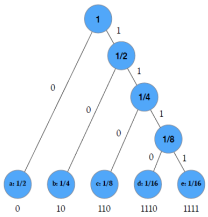
6.2 Restricted Variable-Length Codes

6.3 Huffman codes

In an ideal encoding scheme, a symbol with probability pi of occurring will be assigned a code which takes log(pi) bits. The more probable a symbol is to occur, the smaller its code should be. By this view, UTF-32 assumes a uniform distribution over all unicode symbols; UTF-8 assumes ASCII characters are more common. Huffman Codes achieve the best possible compression ratio when the distribution is known and when no code can stand for multiple symbols.

Symbol	p	Code	E[length]
a	1/2	0	0.5
b	1/4	10	0.5
c	1/8	110	0.375
d	1/16	1110	0.25
e	1/16	1111	0.25

Huffman Codes are built using a binary tree which always joins the least probable remaining nodes.



1. Create a leaf node for each symbol, weighted by its probability.
2. Iteratively join the two least probable nodes without a parent by creating a parent whose weight is the sum of the childrens weights.
3. Assign 0 and 1 to the edges from each parent. The code for a leaf is the sequence of edges on the path from the root.

Huffman codes achieve the theoretical limit for compressibility, assuming that the size of the code table is negligible and that each input symbol must correspond to exactly one output symbol. Other codes, such as Lempel-Ziv encoding, allow variable-length sequences of input symbols to correspond to particular output symbols and do not require transferring an explicit code table. Compression schemes such as gzip are based on Lempel-Ziv encoding. However, for encoding inverted lists it can be beneficial to have a 1:1 correspondence between code words and plaintext characters.

7 Encoding integers

8 Distributed Indexing

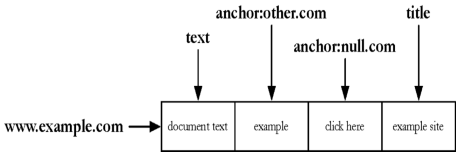
9 Map-Reduce

10 Big Table

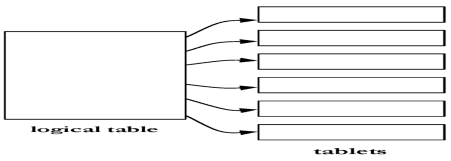
Storage systems such as BigTable are natural fits for distributed algorithm execution. Google invented BigTable to handle its index, document cache, and most of its other massive storage needs. This has produced a whole generation of distributed storage systems, called NoSQL systems. Some examples include MongoDB, Couchbase, etc.

BigTable was developed by Google to manage their storage needs. It is a distributed storage system designed to scale across hundreds of thousands of machines, and to gracefully continue service as machines fail and are replaced. Storage systems such as BigTable are natural fits for processes distributed with MapReduce.

"A Bigtable is a sparse, distributed, persistent multidimensional sorted map." - Chang et al, 2006.



The data in BigTable is logically organized into rows. For instance, the inverted list for a term can be stored in a single row. A single cell is identified by its row key, column, and timestamp. Efficient methods exist for fetching or updating particular groups of cells. Only populated cells consume filesystem space: the storage is inherently sparse.



BigTable rows reside within logical tables, which have pre-defined columns and group records of a particular type. The rows are subdivided into 200MB tablets, which are the fundamental underlying filesystem blocks. Tablets and transaction logs are replicated to several machines in case of failure. If a machine fails, another server can immediately read the tablet data and transaction log with virtually no downtime.

All operations on a BigTable are row-based operations. Most SQL operations are impossible here: no joins or other structured queries. BigTable rows can have massive numbers of columns, and individual cells can contain large amounts of data. For instance, it's no problem to store a translation of a document into many languages, each in its own column of the same row.

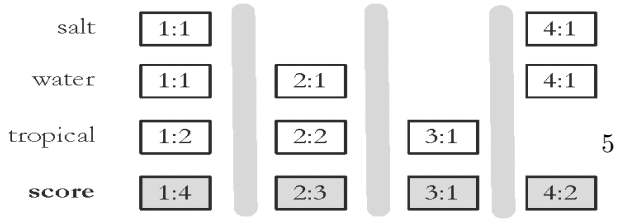
11 Query Processing

There are two main approaches to scoring documents for a query on an inverted index.

· **Document-at-a-time** : It processes all the terms' posting lists in parallel, calculating the score for each document as it's encountered.

· **Term-at-a-time** : Term-at-a-time processes posting lists one at a time, updating the scores for the documents for each new query term. There are optimization strategies for either approach that significantly reduce query processing time document.

11.1 Doc-at-a-Time Processing



We scan through the postings for all terms simultaneously, calculating the score for each document. We remember scores for the top k documents found so far.

Recall that the document score has the form:

$\sum_{w \in q} f(w).g(w)$

for document features f(w) and query features g(w).

This algorithm implements doc-at-atime retrieval. It uses a list L of inverted lists for the query terms, and processes each document in sequence until all have been scored. The documents are placed into the priority queue R so the top k can be returned

11.2 Term-at-a-Time Processing

For term-at-a-time processing, we read one inverted list at a time. We maintain partial scores for the documents we've seen so far, and update them for each term. This may involve remembering more document scores, because we don't necessarily know which documents will be in the top k (but sometimes we can guess).