Lesson 4

Inverted Indexing for Text Retrieval

Introduction

Web search is the quintessential large-data problem. Given a short query consisting of a few terms, the system's task is to retrieve relevant web objects (web pages, PDF documents, PowerPoint slides, etc.) and present them to the user. How large is the web? It is difficult to compute exactly, but even a conservative estimate would place the size at several tens of billions of pages, totaling hundreds of terabytes (considering text alone). In real-world applications, users demand results quickly - query latencies longer than a few hundred milliseconds will try a user's patience.

Introduction

Nearly all retrieval engines for full-text search today rely on a data structure called an inverted index, which given a term provides access to the list of documents that contain the term. In information retrieval parlance, objects to be retrieved are generically called “documents" even though in actuality they may be web pages, PDFs, Java code, …. Given a user query, the retrieval engine uses the inverted index to score documents that contain the query terms with respect to some ranking model, taking into account features such as term matches, term proximity, attributes of the terms in the document (e.g., bold, appears in title, etc.), as well as the hyperlink structure of the documents (e.g., PageRank [117], which we'll discuss in Chapter 5, or related metrics such as HITS [84] and SALSA [88]).

Introduction

The web search problem decomposes into three components: gathering web content (crawling), construction of the inverted index (indexing) and ranking documents given a query (retrieval). Crawling and indexing share similar characteristics and requirements, but these are very different from retrieval. Gathering web content and building inverted indexes are for the most part offline problems. Both need to be scalable and efficient, but they do not need to operate in real time. Indexing is usually a batch process that runs periodically: the frequency of refreshes and updates is usually dependent on the design of the crawler. Some sites (e.g., news organizations) update their content quite frequently and need to be visited often; other sites (e.g., government regulations) are relatively static.

Introduction

However, even for rapidly changing sites, it is usually tolerable to have a delay of a few minutes until content is searchable. Furthermore, since the amount of content that changes rapidly is relatively small, running smaller-scale index updates at greater frequencies is usually an adequate solution. Retrieval, on the other hand, is an online problem that demands sub-second response time. Individual users expect low query latencies, but query throughput is equally important since a retrieval engine must usually serve many users concurrently. Furthermore, query loads are highly variable, depending on the time of day, and can exhibit “spikey" behavior due to special circumstances (e.g., a breaking news). On the other hand, resource consumption for the indexing problem is more predictable

WEB CRAWLING

Before building inverted indexes, we must first acquire the document collection over which these indexes are to be built. In academia and for research purposes, this can be relatively straightforward. Standard collections for information retrieval research are widely available for a variety of genres ranging from blogs to newswire text. For researchers who wish to explore web-scale retrieval, there is the ClueWeb09 collection that contains one billion web pages in ten languages (totaling 25 terabytes) crawled by Carnegie Mellon University in early 2009. Obtaining access to these standard collections is usually as simple as signing an appropriate data license from the distributor of the collection, paying a reasonable fee, and arranging for receipt of the data.

WEB CRAWLING

For real-world web search, however, one cannot simply assume that the collection is already available. Acquiring web content requires crawling, which is the process of traversing the web by repeatedly following hyperlinks and storing downloaded pages for subsequent processing. Conceptually, the process is quite simple to understand: we start by populating a queue with a “seed" list of pages. The crawler downloads pages in the queue, extracts links from those pages to add to the queue, stores the pages for further processing, and repeats. In fact, rudimentary web crawlers can be written in a few hundred lines of code.

WEB CRAWLING

However, effective and efficient web crawling is far more complex. The following lists a number of issues that real-world crawlers must contend with:

- A web crawler must practice good “etiquette" and not overload web servers. For example, it is common practice to wait a fixed amount of time before repeated requests to the same server. In order to respect these constraints while maintaining good throughput, a crawler typically keeps many execution threads running in parallel and maintains many TCP connections (perhaps hundreds) open at the same time.

WEB CRAWLING

- Since a crawler has finite bandwidth and resources, it must prioritize the order in which unvisited pages are downloaded. Such decisions must be made online and in an adversarial environment, in the sense that spammers actively create “link farms" and “spider traps" full of spam pages to trick a crawler into over representing content from a particular site.

WEB CRAWLING

- Most real-world web crawlers are distributed systems that run on clusters of machines, often geographically distributed. To avoid downloading a page multiple times and to ensure data consistency, the crawler as a whole needs mechanisms for coordination and load-balancing. It also needs to be robust with respect to machine failures, network outages, and errors of various types.

WEB CRAWLING

- Web content changes, but with different frequency depending on both the site and the nature of the content. A web crawler needs to learn these update patterns to ensure that content is reasonably current. Getting the right recrawl frequency is tricky: too frequent means wasted resources, but not frequent enough leads to stale content.

WEB CRAWLING

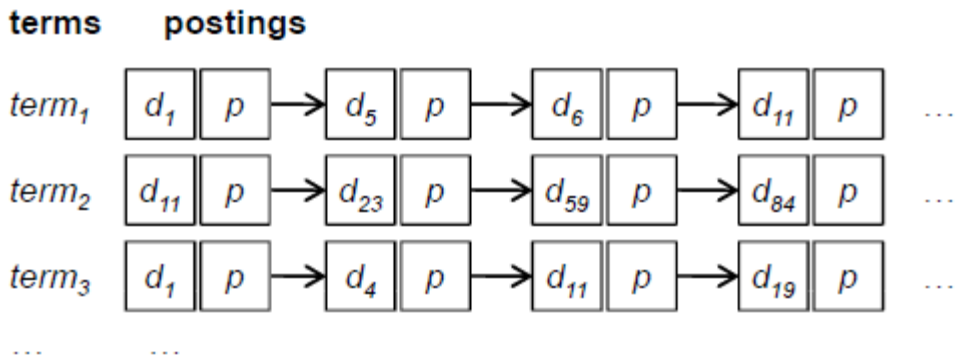
- The web is full of duplicate content. Examples include multiple copies of a popular conference paper, mirrors of frequently-accessed sites such as Wikipedia, and newswire content that is often duplicated. The problem is compounded by the fact that most repetitious pages are not exact duplicates but near duplicates (that is, basically the same page but with different ads, navigation bars, etc.) It is desirable during the crawling process to identify near duplicates and select the best exemplar to index.

WEB CRAWLING

- The web is multilingual. There is no guarantee that pages in one language only link to pages in the same language. For example, a professor in Asia may maintain her website in the local language, but contain links to publications in English. Furthermore, many pages contain a mix of text in different languages. Since document processing techniques (e.g., tokenization, stemming) differ by language, it is important to identify the (dominant) language on a page.

INVERTED INDEXES

An inverted index consists of postings lists, one associated with each term that appears in the collection.



A postings list is comprised of individual postings, each of which consists of a document id and a payload - information about occurrences of the term in the document. The simplest payload is “nothing”!

INVERTED INDEXES

For simple boolean retrieval, no additional information is needed in the posting other than the document id; the existence of the posting itself indicates that presence of the term in the document. The most common payload, however, is term frequency (tf), or the number of times the term occurs in the document. More complex payloads include positions of every occurrence of the term in the document (to support phrase queries and document scoring based on term proximity),

INVERTED INDEXES

Properties of the term (such as if it occurred in the page title or not, to support document ranking based on notions of importance), or even the results of additional linguistic processing (for example, indicating that the term is part of a place name, to support address searches). In the web context, anchor text information (text associated with hyperlinks from other pages to the page in question) is useful in enriching the representation of document content); this information is often stored in the index as well.

INVERTED INDEXES

In the example shown in Figure 4.1, we see that term1 occurs in d1, d5, d6, d11,…, term2 occurs in d11, d23, d59, d84, … , and term3 occurs in d1, d4, d11, d19, … . In an actual implementation, we assume that documents can be identified by a unique integer ranging from 1 to n, where n is the total number of documents.

INVERTED INDEXES

Generally, postings are sorted by document id, although other sort orders are possible as well. The document ids have no inherent semantic meaning, although assignment of numeric ids to documents need not be arbitrary. For example, pages from the same domain may be consecutively numbered. Or, alternatively, pages that are higher in quality (based, for example, on PageRank values) might be assigned smaller numeric values so that they appear toward the front of a postings list. Either way, an auxiliary data structure is necessary to maintain the mapping from integer document ids to some other more meaningful handle, such as a URL.

INVERTED INDEXES

Given a query, retrieval involves fetching postings lists associated with query terms and traversing the postings to compute the result set. In the simplest case, boolean retrieval involves set operations (union for boolean OR and intersection for boolean

AND) on postings lists, which can be accomplished very effciently since the postings are sorted by document id. In the general case, however, query-document scores must be computed. Partial document scores are stored in structures called accumulators. At the end (i.e., once all postings have been processed), the top k documents are then extracted to yield a ranked list of results for the user. Of course, there are many optimization strategies for query evaluation (both approximate and exact) that reduce the number of postings a retrieval engine must examine.

INVERTED INDEXES

Input to the mapper consists of document ids (keys) paired with the actual content (values). Individual documents are processed in parallel by the mappers. First, each document is analyzed and broken down into its component terms. The processing pipeline differs depending on the application and type of document, but for web pages typically involves stripping out HTML tags and other elements such as JavaScript code, tokenizing, case folding, removing stopwords (common words such as `the', `a', `of', etc.), and stemming (removing affixes from words so that `dogs' becomes `dog'). Once the document has been analyzed, term frequencies are computed by iterating over all the terms and keeping track of counts. Lines 4 and 5 in the pseudo-code reflect the process of computing term frequencies, but hides the details of document processing.

INVERTED INDEXES

After this histogram has been built, the mapper then iterates over all terms. For each term, a pair consisting of the document id and the term frequency is created. Each pair, denoted by < n;H{t} > in the pseudo-code, represents an individual posting. The mapper then emits an intermediate key-value pair with the term as the key and the posting as the value, in line 7 of the mapper pseudo-code. Although as presented here only the term frequency is stored in the posting, this algorithm can be easily augmented to store additional information (e.g., term positions) in the payload.

INVERTED INDEXING: BASELINE IMPLEMENTATION

```
class Mapper
    procedure Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t in doc d do
            H{t} ← H{t} + 1
        for all term t in H do
            Emit(term t; posting < n, H{t} >)
```