Implementation Issues and IR Systems

This lecture is about how to implement an information retrieval system or a search engine. In general, an IR system consists of three components: (1) Indexer: this is the module that processes documents and index them with appropriate data structures. An Indexer can be run offline. The main challenge are to index large amounts of documents quickly with limited amount of memory. Other challenges include supporting addition and deletion of documents. (2) Scorer/Ranker: this is the module that takes a query and returns a ranked list of documents. Here the challenge is to implement a retrieval model efficiently so that we can score documents efficiently. (3) Feedback/Learner: this is the module that is responsible for relevance feedback or pseudo feedback. When there is a lot of implicit feedback information such as user clickthroughs available (as in a modern Web search engine), this learning module can be fairly sophisticated. The techniques for implementing a learner, however, highly depend on the learning approaches and applications. For (1) and (2), there are fairly standard techniques that are essentially used in all current search engines.

Inverted index is the main data structure used in a search engine. It allows for quick lookup of documents that contain any given term. The relevant data structures include (1) term lexicon (mapping between a string form of a term and an integer ID); (2) document id lexicon (mapping between a string form of a document ID (e.g., URL) and an integer ID); (3) inverted index (mapping from any term ingeger ID to a list of document IDs and frequency/position information of the term in those documents).

Indexing is the process of creating these data structures based on a set of documents. A popular approach for indexing is the following sorting-based approach:

- Scan the document stream sequentially. Store each document ID in the doc id lexicon. Parse each document to obtain terms and store any new term in the term lexicon.
- While scanning documents, we can collect term counts for each term-document pair and build an inverted index for a subset of documents in memory. When we reach the limit of memory, we write the incomplete inverted index into the disk.
- Continue this process to generate many incomplete inverted indices (called "runs") all written on disk.
- Marge all these runs in a pair-wise manner to produce a single sorted file.

Read this paper for more detailed discussion of a new efficient method for constructing an inverted index.

Once an inverted index is built, scoring a query can be done efficiently using the following procedure:

- Iterate over all terms in the query.
- For each term, fetch the corresponding entries in the inverted index.
- Create document score accumulators as needed.
- Scan the inverted index entries for the current term and for each entry (corresponding to a document containing the term), update its score accumulator based on some term weighting method.
- As we finish processing all the query terms, the score accumulators should have the final scores for all the documents that contain at least one query term. (Note that we don't need to create a score accumulator if the document doesn't match any query term.)

The basic information stored in an inverted index is the term frequency. (IDF can be stored together with the term lexicon.) In order to support "proximity heuristics" (rewarding matching terms that are together), it is also common to store the position of each term occurrence. Such position information can be used to check whether all the query terms are matched within a certain window of text. In an extreme case, it can be used to check whether a phrase is matched.

Another technical component in a retrieval system is integer compression, which is generally applied to compress the inverted index, which is often very large. A compressed index is not only smaller, but also faster when it's loaded into the memory. The general idea of compressing integers (and compression in general) is to exploit the non-uniform distribution of values. Intuitively, we will assign a short code to values that are frequent at the price of using longer codes for rare values. The optimal compression rate is related to the entropy of the random variable taking the values that we consider -- skewed distributions would have lower entropy and are thus easier to compress. Some commonly used variable-length coding methods include unary coding, gamma-coding, and delta-coding. Read the slides to understand how each works. Since inverted index entries are often accessed sequentially, we may exploit this property to compress document ids based on their gaps. The document IDs would otherwise be distributed relatively uniformly, but the distribution of their gaps would be skewed since when a term is frequent, its inverted list would have many document IDs, leading to many small gaps.

There are many existing IR toolkits that we can use to build a search engine. Thus in practice, we rarely need to write our own code to build an index. Sometimes, however, if you want to implement a new retrieval function that needs access to some special statistics that are not available in a regular inverted index, you may need to extend an existing method to store the additional information.

The following book is the best book about the implementation of a retrieval system: Managing Gigabytes: Compressing and Indexing Documents and Images (The Morgan Kaufmann Series in Multimedia and Information Systems) (Hardcover) by Ian H. Witten (Author), Alistair Moffat (Author), Timothy C. Bell (Author) .