CS3245 — INFORMATION RETRIEVAL

DR. ZHAO JIN

arsatis[*]

CONTENTS

1 Language Models 3
- 1.1 Unigram Models . . . 3
- 1.2 N-gram Models . . . 3
- 1.3 Smoothing . . . 4

2 Information Retrieval 4
- 2.1 Indexer . . . 6
- 2.2 Boolean Retrieval . . . 6
- 2.3 Skip pointers . . . 8
  - 2.3.1 Placement of Skips . . . 8

3 Phrase Queries 9
- 3.1 Text Extraction . . . 10

4 Tolerant Retrieval 10
- 4.1 Dictionaries . . . 10
- 4.2 Wildcard Queries . . . 11
  - 4.2.1 Permuterm index . . . 11
  - 4.2.2 k-gram index . . . 12
- 4.3 Spelling Correction . . . 12
  - 4.3.1 Soundex . . . 13

[*]author: https://github.com/arsatis

5 Index Construction 14
5.1 Basic Variants . . . 15
5.1.1 Blocked Sort-Based Indexing (BSBI) . . . 15
5.1.2 Single-Pass In-Memory Indexing (SPIMI) . . . 15
5.2 Distributed Indexing . . . 16
5.3 Dynamic Indexing . . . 17

6 Index Compression 17
6.1 Dictionary Compression . . . 18
6.2 Postings Compression . . . 19

7 Vector Space Model 21
7.1 Scoring . . . 21
7.2 Vector Representation of Documents . . . 22
7.3 Efficient Ranking . . . 23
7.3.1 Heuristics for Pruning . . . 23
7.3.2 Incorporating Additional Information . . . 25

8 Evaluation 26
8.1 Metrics . . . 26
8.2 Test Collections . . . 28
8.3 A/B Testing . . . 28

9 Query Refinement 28
9.1 Relevance Feedback . . . 28
9.2 Query Expansion . . . 30

10 XML Retrieval 30
10.1 Vector Space Model for XML IR . . . 31
10.2 XML IR Evaluation . . . 32

11 Probabilistic Information Retrieval 33
11.1 Classic Probabilistic Retrieval Models . . . 33
11.1.1 Binary Independence Model (BIM) . . . 34
11.1.2 Nonbinary Models . . . 36
11.2 Language Models for IR . . . 37

12 Crawling and Link Analysis 38
12.1 Crawling . . . 38
12.2 Link Analysis . . . 39
12.2.1 PageRank . . . 40
12.2.2 Ergodic Markov Chains . . . 40

1 LANGUAGE MODELS

Lecture 1
14th January 2022

A **language model** is a *probabilistic model of language* that is simple without the use of a grammar. It is created based on a collection of text, and used to assign a probability to a sequence of words. Some application of language modules include:

- Deciding between alternatives
- Speech recognition
- Spelling correction
- Plagiarism detection
- Prediction of consumer preferences
- Typeahead prediction (on mobile devices)

1.1 *Unigram Models*

The **unigram model** (a.k.a. bag-of-words) is one of the simplest language models, which views language as an unordered collection of tokens. Each of the n tokens contribute one count to the model, and the model outputs a count (or probability) of an input based on its individual tokens.

However, unigram language models however do not model word order. In order to model word order, additional context has to be introduced to the model.

1.2 N-*gram Models*

The **Markov assumption** is the presumption that the future behaviour of a dynamical system only depends on its *recent* history. In particular, in a k^{th}-order Markov model, the next state only depends on the k most recent states.

More generally, an $n$gram language model is a model that remembers sequences of n tokens. In other words, an $n$gram model is able to predict a given word from the $n-1$ previous context words, i.e., it is an $(n-1)^{\text{th}}$-order Markov model with $\mathrm{P}(x_n|x_1,\ldots,x_{n-1})$.

Sometimes, special **START** and **END** symbols are used for encoding beyond the text boundary in $n$gram models.

Longer $n$gram models are generally more accurate, but exponentially more costly to construct. Consider the case where the size of the vocabulary used in a language is represented by $|\mathrm{V}|$:

- for a unigram model, we will need to store counts/probabilities for all |V| words.
- for a bigram (2-gram) model, we will need to store counts/probabilities for all $|V| \times |V|$ ordered length 2 phrases.
- by induction, for an $n$gram model, we will need to store counts/probabilities for up to $|V|^n$ ordered length n phrases.

1.3 *Smoothing*

When we collect a sample of data from a population, the distribution of our data may not always be great for computation in probability-based language models. More specifically, our data may suggest that a particular word (e.g., "don't") does not exist in the population, when there it could be due to a sampling error/mishap. This could cause probability-based models to incorrectly predict that a certain phrase (e.g., "I don't want") cannot exist in a given vocabulary, since:

$$P(\text{“I”}) \times P(\text{“}don't\text{”}) \times P(\text{“}want\text{”}) = P(\text{“I”}) \times 0 \times P(\text{“}want\text{”}) = 0$$

To tackle such issues, **smoothing** techniques can be used. Some common smoothing techniques include:

- **Add-one/Laplace smoothing**:

$$P(x_n|x_1, ..., x_{n-1}) = \frac{C(x_1, ..., x_n) + k}{\sum_x (C(x_1, ..., x_{n-1}, x) + k)} = \frac{C(x_1, ..., x_n) + k}{C(x_1, ..., x_{n-1}) + kV}$$

- **Add-delta smoothing***
- **Witten-Bell smoothing***
- **Kneser-Ney smoothing***

2 INFORMATION RETRIEVAL

Lecture 2
21st January 2022

Information retrieval (IR) is finding material of an *unstructured* nature (usually text) that satisfies an *information need* from within *large* collections. An illustration of the classic search model used in IR is shown in Figure 1.

It is important to ensure that the returned documents are **relevant**, i.e., they help to satisfy the information need. Some metrics that are used to evaluate relevance include **precision** and **recall**.

precision: fraction of retrieved documents that are *relevant* to the user's information need

recall: fraction of *relevant* documents in the collection that are retrieved

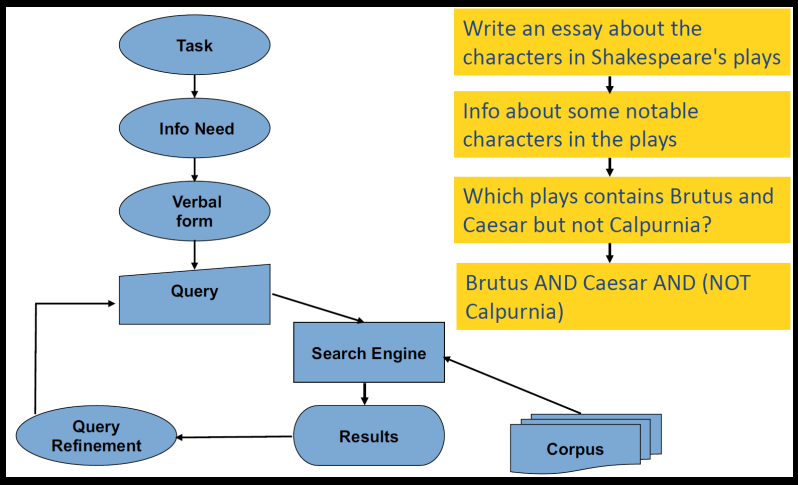

Figure 1: Example of the classic search model. The search engine indexes the corpus and processes the query to return some results.

Since the collection/corpus could be very large (i.e., on the scale of millions of documents), it will be very computationally expensive to build a *term-document incidence matrix* (i.e., a matrix of 0s and 1s indicating whether a document contains a term). Instead, we can use an **inverted index**, which is an adjacency list such that for each term t, we will store a list of all documents that contain t. Each document is identified by a unique serial number, called the **docID**.

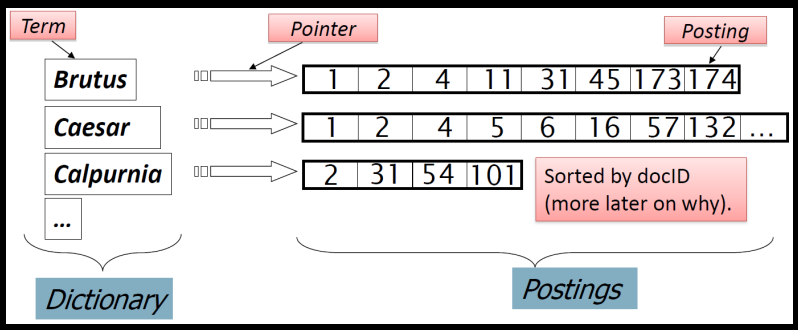

Figure 2: Example of an inverted index.

In memory, this can be implemented by using linked lists or variable length arrays; on the disk, a continuous run of postings would be most optimal. The inverted index can be constructed using the following steps:

fixed-size arrays are not recommended because (1) there will be space wastage, and (2) there may be insufficient space for some terms

1. Pass the documents to be indexed into a tokenizer.
2. The tokens are then passed into linguistic modules to be normalized.
3. Normalized tokens are indexed using an indexer.

2.1 *Indexer*

Inside the indexer, the following steps are taken to output an inverted index.

1. A sequence of (term, docID) pairs (i.e., term entries) are generated.
2. The pairs are then first sorted by term, and then by docID (i.e., the core indexing step).
3. Identical pairs in a single document are merged, and then split into the *dictionary* and *postings*. Information regarding **document frequency** is also stored.

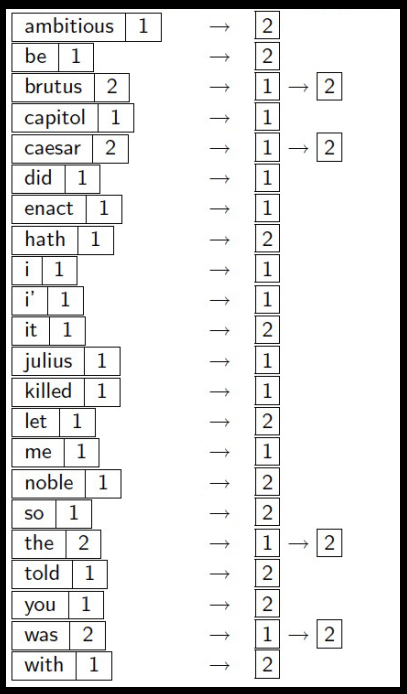

Figure 3: Example of the dictionary-posting split. The dictionary (along with the document frequency of each term) is shown on the left, whereas the postings are shown on the right.

2.2 *Boolean Retrieval*

The **Boolean retrieval model** is a model that processes Boolean queries. Boolean queries are queries that use the operators `AND`, `OR`, and `NOT` to join query terms; they view each document as a *set* of words, and checks whether each document matches the given condition.

We will explore some basic optimization techniques on Boolean queries, which will help to speed up queries on large collections.

- `X AND Y`
  1. Locate `X` in the dictionary and retrieve its postings.
  2. Locate `Y` in the dictionary and retrieve its postings.
  3. Intersect/"Merge" the two postings, keeping only the common entries.
     - the merge step involves walking through the two postings *simultaneously* using two indices, in the same manner as the merge step in *merge sort*.
     - the algorithm stops immediately when we encounter the end of one of the postings (only for the `AND` query) .
     - if the list lengths are n_x and n_y, the merge step takes $O(n_x + n_y)$ operations.
- `X OR Y`
  1. Locate `X` in the dictionary and retrieve its postings.
  2. Locate `Y` in the dictionary and retrieve its postings.
  3. Union/"Merge" the two postings, keeping all entries that appear in any of the two postings.
- `NOT X`
  1. Retrieve the full list of documents.
  2. Locate `X` in the dictionary and retrieve its postings.
  3. "Merge" the full list and the postings, keeping all entries that appear in the full list but *not* in the postings.

When the query involves a conjunction of n terms, some optimization heuristics include:

- for $\underset{i}{\texttt{AND}}\left(\texttt{X}_i\right)$ and $\underset{i}{\texttt{OR}}\left(\texttt{X}_i\right)$:
  - we can process *in the order of increasing frequency*, i.e., start with the smallest set.
- for `X AND NOT Y`:
  - we can perform the "merge" step similar to that in the processing of the query `NOT X`, but with reference to the postings of `X` instead of the full list of documents. The worst case runtime is $O(n_x + n_y)$.
- for `X OR NOT Y`:
  - `NOT Y` will have to be processed first (or in parallel with `X OR ...`), since there could be documents which do not appear in either of the postings of `X` or `Y`. The worst case runtime will thus be $O(|documents|)$.

Lecture 3
28th January 2022

2.3 *Skip pointers*

Another way to speed up the **AND** merging/intersection of postings involves the use of **skip pointers**. Skip pointers are setup at indexing time, and they can be utilized to skip postings that will not contribute to the search results.

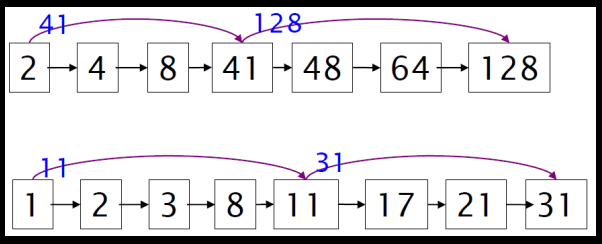

Figure 4: Example of some postings with skip pointers implemented.

For example, suppose we have two lists of postings $\mathrm{P}_1 = p_{1,1} \rightarrow p_{1,2} \rightarrow \ldots \rightarrow p_{1,m}$ and $\mathrm{P}_2 = p_{2,1} \rightarrow p_{2,2} \rightarrow \ldots \rightarrow p_{2,n}$, and we are currently at some posting $p_{1,i} \in \mathrm{P}_1$ and $p_{2,i'} \in \mathrm{P}_2$. Suppose further that $p_{1,i}$ has a skip pointer pointing to $p_{1,i+j}$.

- If $p_{1,i+j} \leq p_{2,i'}$, then we would be able to use the skip pointer to skip from $p_{1,i}$ through $p_{1,i+j}$.
- However, if $p_{1,i+j} > p_{2,i'} > p_{1,i}$, then we would not be able to utilize the skip pointer, and would have to check the posting $p_{1,i+1}$ instead.

However, there are additional costs of using skip pointers, such as the space overhead to store skip pointers, as well as the additional time required to make an additional comparison (with the posting that the skip pointer is pointing to).

2.3.1 Placement of Skips

If more skips are placed, the skip spans would be shorter. This implies a higher likelihood of skipping, but also more comparisons to skip pointers made.

On the other hand, if fewer skips are placed, the skip spans would be longer, which implies a lower likelihood of skipping and fewer comparisons to skip pointers.

A heuristic that can be utilized is to implement $\sqrt{\mathrm{L}}$ evenly-spaced skip pointers for postings of length L.

3 PHRASE QUERIES

Phrase queries help us to answer multi-word queries, such as "Tom and Jerry", as a phrase. Since storing individual terms with their respective docIDs would no longer help with resolving phrase queries, we would need to turn towards other solutions, including:

- **Biword indexes**: i.e., we index every consecutive pair of terms in the text. Then, we can process the long phrase queries as a Boolean query on biwords.
  - For example, for "Tom and Jerry", we could index the biwords "Tom and" and "and Jerry".
  - We can also process a query "Tom and Jerry" as *Tom and* **AND** *and Jerry*.
  - However, this could lead to false positives, since sentences can contain each of the bigrams but not match the desired phrase query, e.g.: "Dad **and Jerry** ate **Tom and** Bob".
- **Extended biword indexes**: i.e., we index all extended biwords in the form $\mathrm{NX} * \mathrm{N}$, where N = noun, X = articles/prepositions. Then, we can process phrase queries by extracting and looking up the extended biwords.
  - For example, for the phrase "Tom and Jerry", we would index the extended biword "Tom Jerry" instead.
  - However, this method fails to handle complex phrases that consist of fully nouns, e.g., "National University of Singapore" cannot be index into an extended biword.
- **Positional indexes**: i.e., in the postings, we store for each term the position(s) in which tokens of it appear, e.g.:

  <**term**, document frequency;

  doc_1 : $\mathrm{position}_1, \mathrm{position}_2, \ldots, \mathrm{position}_{n_1}$;

  doc_2 : $\mathrm{position}_1, \mathrm{position}_2, \ldots, \mathrm{position}_{n_2}$;

  $\vdots$

  >

  We can process a phrase query by first extracting inverted index entries for each distinct term, and then merging their *doc:position* lists to enumerate all positions with the desired phrase.
  - Note that positional indices expands the storage substantially, since the index size depends on the average document size.

  - However, *index compression* methods can be used to compress position values/offsets.
  - We can also combine positional indices with biword indices, in order to retrieve biword postings without merging (since merging is relatively slow for positional indices). However, this would require some additional storage.

3.1 *Text Extraction*

This section is hypercompressed due to time constraints and similarities in content with CS4248.

There are various methods which could be applied to speed up/enhance phrase queries. Some text extraction and preprocessing techniques involve:

- **tokenization**
- removal/separate storage of numbers, dates, and passwords
- **stop word** removal: i.e., using a stop list, we exclude the most common words from the dictionary (e.g., "the", "a", "to", which have little semantic content.
  - However, stop word removal is rarely used nowadays because stop words may be useful for phrase queries and relational queries, and compression and query optimization techniques are relatively good nowadays.
- normalizing tokens to terms, e.g.:
  - dropping punctuation (e.g., within-term periods and hyphens)
  - removing accents and diacritics
- **case folding**: i.e., reduce all letters to lowercase
- **lemmatization** and **stemming**
- other techniques such as fixing spelling variations (e.g., "color" vs "colour"), synonyms, or transliteration variations (e.g., "Beijing" vs "Peking")

4 TOLERANT RETRIEVAL

Lecture 4
1st May 2022

4.1 *Dictionaries*

There are several ways of implementing dictionaries, each with their strengths and weaknesses.

- **Naïve dictionaries**: terms are stored using fixed-length arrays.
  - Cons:

    - Space wastage for shorter words, and insufficient space for longer words.
    - Slow access (i.e., linear scan needed for lookup operations).
- **Hash tables**: each vocabulary term is hashed to an integer.
  - Pros:
    - Lookup is faster (i.e., O(1)).
  - Cons: not very tolerant.
    - No easy way to find minor variants, e.g.: "judgment" vs "judgement".
    - No prefix search.
    - If vocabulary keeps growing, need to occasionally rehash everything (which is an expensive operation).
- **B-trees**: each vocabulary term is stored at the leaf nodes of the tree, sorted according to lexicographical ordering; each internal node of the tree has a number of children in the interval [B/2, B].
  - Pros:
    - Allows for prefix search.
  - Cons:
    - Slower lookup (i.e., O(log |V|)).
    - Rebalancing of trees is expensive.

4.2 *Wildcard Queries*

The wildcard * matches any sequences of letters. By using wildcards, we can perform prefix and suffix queries.

- Prefix queries, e.g.: XY*, can be attended to by maintaining a B-tree structured dictionary. The dictionary will return all words w in the range XY $\leq w <$ XZ.
- Suffix queries, e.g.: *XY, can be attended to by maintaining an additional B-tree for reversed terms. The dictionary will return all words w in the range YX $\leq w <$ YY.
- For general wildcard queries in the form X*Y, we will retrieve all possible words for X* and *Y from the trees, and perform an intersection/"merge" operation.

4.2.1 Permuterm index

The **Permuterm index** technique adds an end marker $ to a word and indexes all rotations of the word in a corpus. For example, for the word "hi", the rotations "hi$", "i$h", and "$hi" will be indexed.

Subsequently, for a wildcard query, the Permuterm index will add an end marker $ to the query, and look up using the rotation with the * at the end. For example, for the query *X*, the rotation X** ≡ X* will be looked up.

However, the use of Permuterm indices causes the lexicon size to blow up proportional to the average word length.

4.2.2 *k*-gram index

An alternative solution is to enumerate all *k*-grams occurring in any term, and maintain an inverted index *from the k-grams to dictionary terms* that match each *k*-gram.

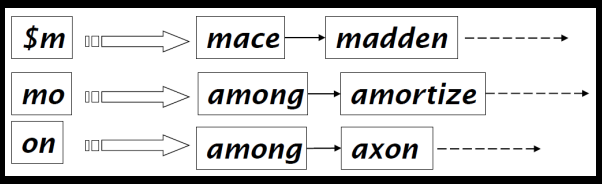

Figure 5: Example of an inverted index for a k-gram index.

Then, a query (e.g., "mon*") can be run as an AND query on the *k*-grams, i.e., "$m" AND "mo" AND "on". However, there is a chance of encountering false positives, thus we will need to perform post-query filtering against the original query, and surviving enumerated terms would then be looked up in the *term-document inverted index*.

k-gram indices are generally faster and more space efficient as compared to Permuterm indices.

4.3 *Spelling Correction*

Sometimes, we will need to correct user queries in order to retrieve the correct answers. In such circumstances, we can either:

- return several suggested alternative queries with the corrected query, or
- immediately retrieve documents indexed by the corrected query.

There are two main flavours of spelling corrections, namely:

1. **isolated word**: checking each word individually for any misspellings.
    - We can use either:
        - a **standard lexicon**, e.g., an English dictionary (may be outdated), or

  - a **lexicon of the indexed corpus**, e.g. all words on a web page (more updated, but may be unreliable, i.e., contain misspellings).
- We also need to decide how to define which word is closest to a misspelling. Some methods that can be used include:
  (a) **minimum edit distance** (which could be expensive and slow)
  (b) **ngram overlap** (faster, but may favor longer terms)

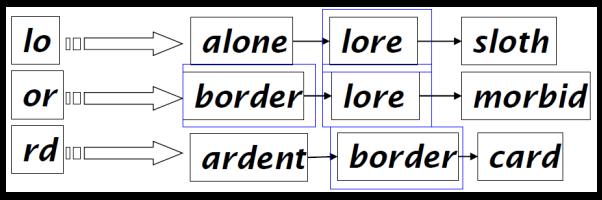

Figure 6: Example of using bigram overlap to identify correct spellings. Here, words with ≥ 2 overlaps are identified by merging.

  (c) **Jaccard coefficient**: let X and Y be two sets of $n$grams, then,

$$\mathrm{JC}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

  JC = 1 when X and Y have the same elements, and JC = 0 when they are disjoint. We would need to decide on a threshold which determines a match (e.g., if JC ≥ 0.5 $\implies$ match).

2. **context-sensitive**: taking the word context (i.e., surrounding words) into account.
   - We can apply correction according to the following steps:
     (a) Dictionary terms close to each query term are retrieved.
     (b) All possible resulting phrases with one word "corrected" at a time are enumerated.
     (c) One (or more) phrases will then be decided based on heuristics, e.g.:
       - **hit-based spelling correction**: i.e., return the correction with the most hits/popularity.

There are some general issues in spelling correction, such its high computational cost. On top of that, we may also have to decide between confirming the corrected phrase with the user, versus immediately searching based on the corrected query.

4.3.1 Soundex

Soundex comprises of a class of heuristics to expand a query into *phonetic equivalents*, e.g., "chebyshev" → "tchebycheff"". The general idea involves

converting every token into a 4-character reduced form, and then building and searching an index on the reduced forms. More specifically, the algorithm involves the following steps:

1. Retain the first letter of the word.
2. Change all occurrences of the following letters: {A, E, I, O, U, H, W, Y} to 0.
3. Change letters to digits as follows:
   - {B, F, P, V} → 1
   - {C, G, J, K, Q, S, X, Z} → 2
   - {D, T} → 3
   - {L} → 4
   - {M, N} → 5
   - {R} → 6
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all 0s from the resulting string.
6. Pad the resulting string with trailing 0s and return the first 4 positions (i.e., of the form "[A – Z]{1}\d{3}").

5 INDEX CONSTRUCTION

Lecture 5
11th February 2022

In order to make index construction scalable, we need to implement solutions based on the characteristics of the underlying hardware, especially with respect to the bottleneck (i.e., hard drive storage). There are two main limitations:

- No data is transferred from disk to memory while the disk head is being positioned (i.e., disk seeking).
  - This implies that transferring one large chunk of data from disk is faster than transferring multiple small chunks.
- Memory is much faster, but limited in quantity.
  - For larger collections/corpora, we would need to store intermediate results on disk.
  - Implementing index construction using disk space would also be inefficient, since there would be too many random seeks involved.

Based on these limitations, some algorithms were proposed to speed up index construction.

5.1 *Basic Variants*

5.1.1 Blocked Sort-Based Indexing (BSBI)

The general idea of **BSBI** consists of the following:

- As documents are parsed, terms (i.e., strings) are mapped to termIDs, and the dictionary storing the mappings would be kept in memory.
- We define blocks of about 10M such records.
- Subsequently, each block have their records:
  - accumulated,
  - sorted,
  - have posting lists created for them, and
  - written to disk.
- Then, the blocks are merged into one long sorted order.
  - We can do **binary (i.e., 2-way) merges**.
  - We can also do *n*-**way merges**, which are more efficient if *decent-sized chunks* of each block are read into memory and written out to disk.

because numbers require less space to store, and are more efficient to work with as compared to strings

However, there are some problems with BSBI, namely:

- The dictionary must fit into memory (which could be hard to guarantee, since it grows dynamically).
- The block size is fixed, and must be decided in advance.
  - Too small → slower, since more blocks need to be processed.
  - Too big → may end of crashing if too much memory is used by other applications.

5.1.2 Single-Pass In-Memory Indexing (SPIMI)

The key ideas of **SPIMI** consists of the following:

- Generate an index (i.e., a dictionary + postings list) using hashing as the pairs are processed.
- Going as far as memory allows, sort the terms lexicographically, write out the index, and merge afterwards.
  - The sorting step is to allow for part of a block to be read into memory (rather than having to read in the entire block) for the merging step.

SPIMI has several advantages over BSBI, including:

- No need to keep dictionary in memory.
- No need to wait for fixed-size blocks to be filled up.
- Adapts to the availability of memory.
- Faster than BSBI due to only sorting dictionary terms (instead of pairs).

5.2 *Distributed Indexing*

For web-scale indexing, a *distributed computing cluster* has to be used, since individual machines tend to be fault-prone. The underlying architecture of distributed indexing consists of the following:

- a **master** node/machine directing the indexing job, i.e.:
  - breaking up input document into *splits* (i.e., subset of documents), and
  - assigning each split to an *idle* parser machine from the pool.
- **parsers** which read documents one at a time, and outputs (term, doc) pairs.
  - the pairs are written into j partitions.
- **inverters** which collect all (term, doc) pairs for some term-partition (e.g., partitions of "a"-"b") from parsers.
  - the pairs are then sorted and written to posting lists.

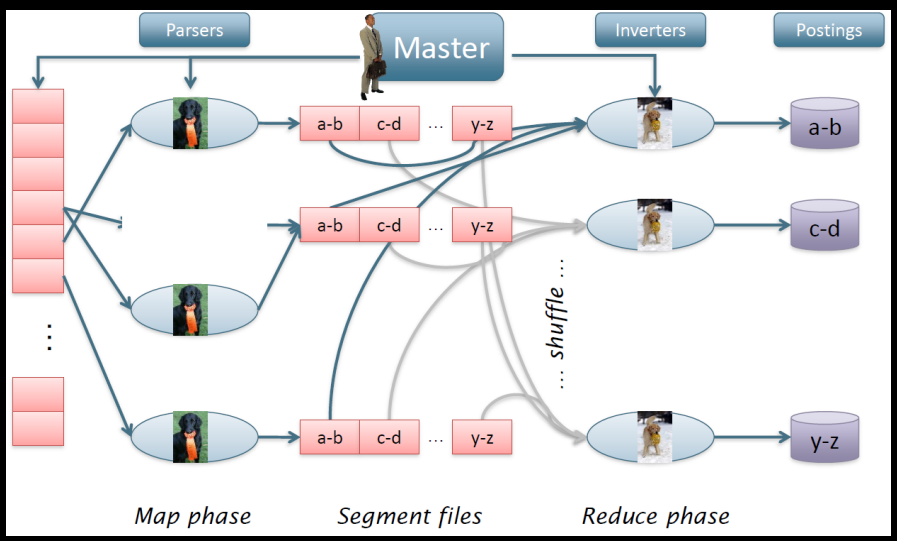

Figure 7: Data flow in distributed indexing.

This index construction algorithm is an instance of the **MapReduce** algorithm.

5.3 *Dynamic Indexing*

Dynamic indexing is implemented for collections which are dynamic (i.e., new documents are added over time → the dictionary and posting lists have to be modified). Some basic approaches to handle such collections include:

- Re-indexing every time (impractical).
- Maintaining two indices (i.e., "big" main index and "small" auxiliary index), where a **linear merge** is performed when the auxiliary index gets too large. Some supported mechanisms include:
  - *add*: new documents go to the auxiliary index.
  - *delete*: a list of deleted documents is maintained.
  - *update*: delete + add.
  - *search*: search both indices, merge results, and omit deleted documents.

  Since linear merge is performed, the cost of merging is rather high (i.e., $O(k^2)$, where k represents the total number of merges performed).
- Maintaining a series of indices:
  - Z_0: in memory, with same capacity as I_0 (similar to the auxiliary index).
  - $I_0, I_1, \ldots, I_n$: on disk, each twice as large as the previous index.

  where a disk write to I_0 is performed when Z_0 gets too large. If I_0 already exists, a **logarithmic merge** is performed with I_0 to give Z_1, which is then written to I_1 (if it does not exist; else repeat until some empty I_i is encountered).

  Since each posting is touched $O(\log k)$ times, complexity is $O(k \log k)$. Even though indexing is more efficient, query processing is slower (since we need to merge results from $O(\log k)$ indices.

6 INDEX COMPRESSION

Lecture 6
18th February 2022

The purpose of compression is to:

- use less disk space,
- keep more data in memory, and
- increase speed of data transfer from disk to memory.

To formalize the discussion on index compression, we will introduce two empirical laws:

- **Heaps' Law**: $M = kT^b$, where:
  - M is the size of the vocabulary,
  - T is the number of tokens/pairs in the collection, and
  - k, b are free parameters to be determined empirically, with typical values of $30 \leq k \leq 100$ and $0.4 \leq b \leq 0.6$.

  Intuitively, this law states that as the size of the vocabulary increases, there will be diminishing returns in terms of discovery of the full vocabulary from which the distinct terms are drawn.
- **Zipf's Law**: $cf_i = \frac{K}{i}$, where:
  - cf_i is the collection frequency of the i^{th} most frequent term,
    - **collection frequency**: the number of occurrences of a term in the collection.
  - i is the rank of the term in the collection's frequency table, and
  - K is a normalizing constant: $K = cf_1$.

  Intuitively, this law states that given some collection/corpus, the frequency of any word in the collection/corpus is inversely proportional to its rank in the frequency table.

6.1 *Dictionary Compression*

Compressing the dictionary provides several benefits, including:

- being able to keep it in memory,
- lower memory competition with other applications, and
- fast search startup time (if the dictionary isn't stored in memory).

Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

Figure 8: An example of storing a dictionary.

Instead of using fixed-size entries for storing individual terms (which is inefficient for small/long terms), we can do:

1. **dictionary-as-a-string**: store the dictionary as a string of characters + replace the term attribute for an attribute for term pointers.

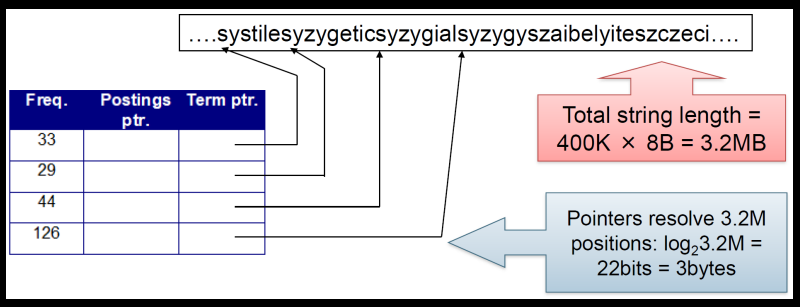

Figure 9: Figure illustrating compression with dictionary-as-a-string.

2. **blocking**: store term lengths within the string + pointers to every k^{th} term on the string.
   - k is typically an intermediate number (e.g., 4), since larger values of k could lead to a drop in efficiency (since linear search is required for searching terms within each block).

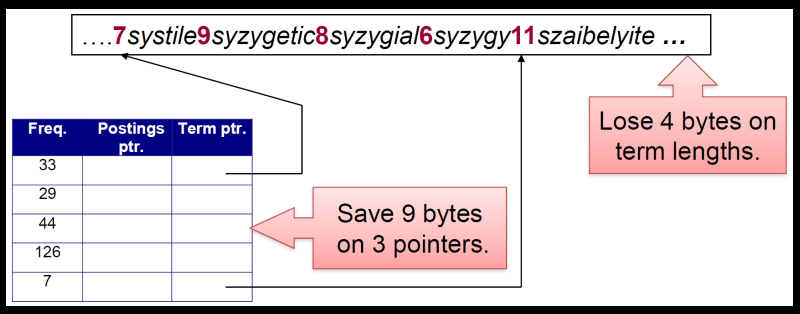

Figure 10: Figure illustrating compression with blocking.

3. **front coding**: store only the differences between sorted words.

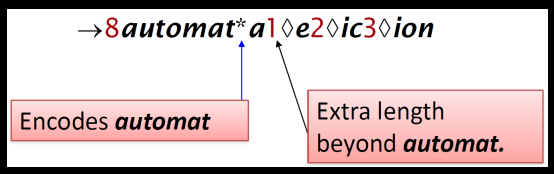

Figure 11: Figure illustrating compression with front coding.

6.2 *Postings Compression*

Aside from the dictionary, we could also compress the postings list, using the following methods:

1. **gap encoding**: store just the *gaps* between adjacent docIDs (since we are storing the list of documents in *increasing order* of docID).

- this is useful for terms with higher collection frequency, since the gaps between their postings would be small (and thus less space required to store the gaps than the postings themselves)

	Encoding	Postings List					
the	docIDs	...	283042	283043	283044	283045	...
	gaps		...	1	1	1	
computer	docIDs	...	2803047	283154	283159	283202	...
	gaps		...	107	5	43	
arachno-centric	docIDs	252000	500100				
	gaps	...	248100				

Figure 12: Figure illustrating compression with gap encoding.

2. **variable byte encoding**: use the fewest bytes needed to store a gap (instead of a fixed number of bits to store every number).

   - We begin with one *byte* to store a gap G. and dedicate 1 bit in it to be a *continuation bit* c (where $c = 0$ indicates that the byte is continuing, and $c = 1$ indicates that the byte is ending).
   - If G < 127, binary-encode it in the 7 available bits, and set $c = 1$.
   - Else, encode G's lower-order 7 bits, and then use additional bytes to encode the higher-order bits using the same algorithm. At the end, set the continuation bit for the last byte to 1 and for the other bytes to 0.
   - Instead of bytes, we can also use other units of alignment, e.g. 32 bits (words), 16 bits, or 4 bits, etc.

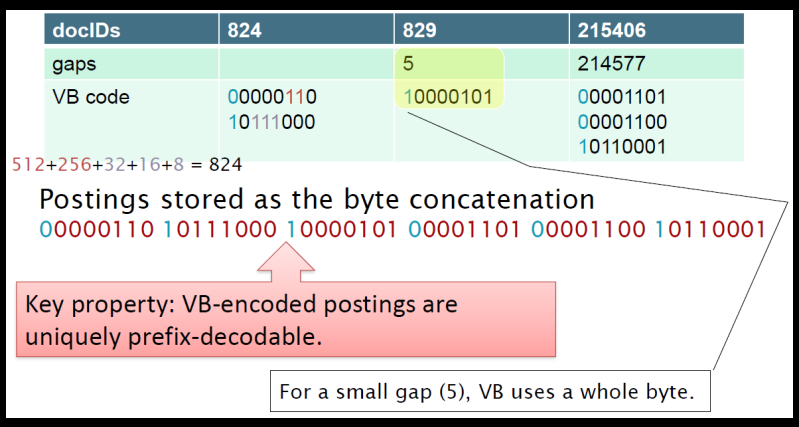

Figure 13: Figure illustrating compression with variable byte encoding.

7 VECTOR SPACE MODEL

Lecture 7
4th March 2022

Since Boolean queries are binary (i.e., they return either "match" or "not a match"), they can result in either too few or too many results. While having many results (e.g., on the scale of thousands) could be useful for users with precise understanding of their needs and of the collection, it would not be good for the typical layman user. Additionally, most layman users are likely to have difficulty in writing Boolean queries.

Thus, rather than returning a set of documents satisfying a query expression, we can adopt **ranked retrieval models**, which return an ordering over the top documents in the collection with respect to a **free text query**.

7.1 *Scoring*

How do we rank the documents in a collection with respect to a query? The typical steps would include:

1. Assigning a score to each document, e.g.:
   - **Jaccard coefficient**: $\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$
     - (+) A and B do not have to be the same size.
     - (+) the score is fixed within the range $[0, 1]$.
     - (−) term and document frequencies are not considered → less important (in terms of idf) or less relevant (in terms of tf) documents may have similar scores to their counterparts.
   - **TF-IDF**: $\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tfidf}_{t,d}$, where

term frequency ($tf_{t,d}$): the number of times that t occurs in d

document frequency (df_t): the number of documents that contain t

$$\text{tfidf}_{t,d} = w_{t,d} = (1 + \log tf_{t,d}) \times \log\left(\frac{N}{df_t}\right) \tag{1}$$

Note that aside from the equation shown in 1, there are other possible ways to compute the tf-idf score, which is briefly summarized in table 14.

Term frequency		Document frequency		Normalization	
n (natural)	$\text{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\text{tf}_{t,d})$	t (idf)	$\log \frac{N}{\text{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \text{tf}_{t,d}}{\max_t(\text{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - \text{df}_t}{\text{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(\text{tf}_{t,d})}{1 + \log(\text{ave}_{t \in d}(\text{tf}_{t,d}))}$				

Figure 14: Variants of tf-idf weighting.

  - Log term frequency, $1 + \log tf_{t,d}$, is used instead because *relevance does not increase proportionally with raw term frequency*.
  - Inverse document frequency, $\log\left(\frac{N}{df_t}\right)$, is used instead because *rare terms are more informative than frequent terms*.
    * We use $\log\left(\frac{N}{df_t}\right)$ instead of $\frac{1}{df_t}$ to keep the value non-negative, and to dampen the effect of idf.
    * Note that idf only affects the ranking of documents for queries with ≥ 2 *terms*.
2. Sorting the documents based on the scores.

7.2 *Vector Representation of Documents*

To improve the scoring metric, we could instead represent documents (and queries) as sparse vectors in a |V|-dimensional vector space, where the axes represent terms in the vocabulary. One key limitation of this bag-of-words representation is that it *does not consider the relative ordering of the words* in a document.

Subsequently, we could rank documents according to their **proximity to the query** in this vector space, using the **cosine similarity score**:

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}} \tag{2}$$

where q_i is the tf-idf weight of term i in the query, and d_i is the tf-idf weight of term i in the document. Note that length normalization is implicitly performed during the computation of cosine similarity.

utilizing the Euclidean distance between $\vec{q}$ and $\vec{d}$ as a measure of proximity is a bad idea because it is unable to capture similarity in term distribution (e.g., dist("hi", "hi hi hi") $\neq 0$)

More specifically, the vector space ranking algorithm can be described as follows:

1. Represent the query as a weighted tf-idf vector.
2. Represent each document as a weighted tf-idf vector.
3. Compute the cosine similarity score for the query vector, and for each document vector.
4. Rank the documents w.r.t. the query by score.
5. Return the top k results to the user.

Some search engines allow for different weightings for queries versus documents. For example, we could adopt ***lnc.ltc*** instead of the standard ***ltc.ltc***

weighting scheme, which reduces the cost of updating the collection (e.g., adding a document to the collection would require updating all the values in the collection).

lnc.ltc is an example of a SMART notation, which denotes that combination used in the weighting scheme, where each letter corresponds to the acronyms from table 14.

7.3 *Efficient Ranking*

Lecture 8
11th March 2022

Instead of returning the true/actual "top" documents, we could speed up the processing using a probabilistic method. Some tweaks which could be performed to speed up the computation of the cosine similarity include:

- assuming each query term has a tfidf weight of 1 when computing $\cos(\vec{q}, \vec{d})$.
  - This removes the need to compute $w_{t,q}$, as well as the multiplication/dot product between $w_{t,d}$ and $w_{t,q}$.

Let J represent the number of documents remaining in the collection, after removing documents which do not contain any query terms (i.e., which have 0 cosine values). Typically, the selection of the top k documents would be preceded by the sorting of the J documents, thus the runtime is $O(J \log J + k)$. Is there any way to speed up this process?

- One possible way is to utilize heaps to select the top k elements instead.
  - The heap takes O(J) operations to construct, and each of the k top documents can be read off in O(log J) steps.
  - Thus, the overall time complexity can be reduced to $O(J + k \log J)$, which is significantly faster since typically $k << J$.
- Another way relies on heuristics to prune less relevant documents from the collection, such that |A| documents (a.k.a. **contenders**) are left in the collection, where $k < |A| << J$.
  - A may not necessarily contain the top k documents, but it suffices if a sizeable portion of its documents are among the top k.

7.3.1 Heuristics for Pruning

Some of the heuristics which could be used for pruning include:

1. **Index elimination**: we could ignore part of the index (i.e., posting lists) based on the query.
   (a) *High idf query terms only*: given a query, we only accumulate scores from high idf terms (e.g., non-stopwords, or rare words such as "retrieval").

- Since the postings of the low idf terms are likely to contain many documents, we could eliminate more documents from the set A (i.e., contenders).
- Similar to stopword removal.

(b) *Documents containing many query terms*: for multi-term queries, we only compute scores for documents containing several of the query terms (e.g., p out of $|q|$ terms).

- The choice of the value p could be optimized via hyperparameter tuning.

2. **Tiered indexes**: we could break the postings up into a hierarchy of lists, ranging from most to least important.

(a) *Champion lists*: we pre-compute the r docs of highest weight in each dictionary term t's postings. Then, at query time, we only compute scores for docs in the champion list of some query term.

- Since r is chosen during the indexing stage, it could be possible that $r < k$.

(b) *High and low lists*: for each term, we maintain two posting lists instead of just one, called *high* and *low* respectively. When traversing postings on a query, we traverse the *high* list first before proceeding to get docs from the *low* list.

- This can help to resolve the issue whereby $k > r$, since we could utilize the *low* list if all the docs in the *high* list has been returned.

(c) *Tiered indexes*: we generalize high-low lists into *tiers*. We will traverse the lower tiers only if the upper tiers are insufficient to get k docs,

3. **Impact-ordered postings**: we sort each postings list by $w_{t,d}$, and only compute scores for docs which $w_{t,d}$ is *high enough*.

(a) *Early termination*: when traversing term t's postings, we stop early after either:

- a fixed number of r docs, or
- $w_{t,d}$ drops below some threshold.

Then, we will take the union of the resulting sets of docs, and compute only the scores for docs in this union.

(b) *idf-ordered query terms*: we consider the postings of query terms in order of decreasing idf.

- We will skip low-idf query terms completely, and move on to the next query term once the score contribution $w_{t,d} \times w_{t,q}$ is low (e.g., ≤ 0.5).

4. **Cluster pruning**: we pick $\sqrt{\mathrm{N}}$ docs at random (a.k.a. **leaders**). For the other docs, we will pre-compute their nearest leader.

Subsequently, when given a query q, we will first find its nearest leader L, and then seek the k nearest docs from L's followers (including L itself).

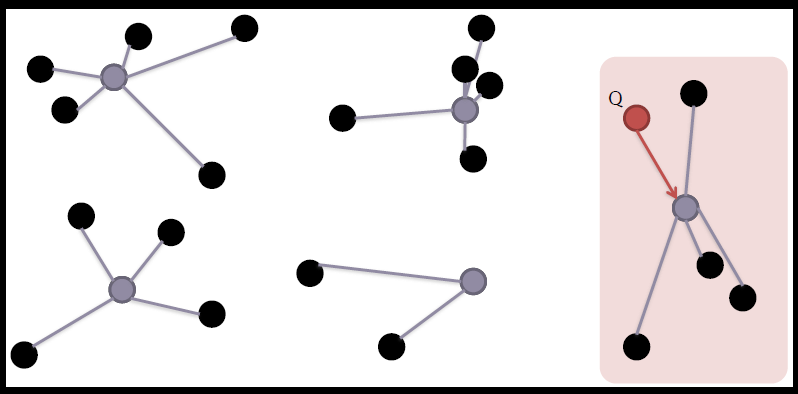

Figure 15: Visualization of cluster pruning.

- One variant involves allowing each follower to be attached to b_1 nearest leaders, and finding the b_2 nearest leaders from the query at query time.
  - b_1 affects the preprocessing step at indexing time.
  - b_2 affects the query processing step at runtime.
  - These parameters would make the algorithm more costly, but note that the preprocessing step will only take place once.

7.3.2 Incorporating Additional Information

Aside from applying heuristics, we could also incorporate additional information to improve the performance of ranking.

1. **Static quality scores**: we might want our top-ranking documents to be both **relevant** and **authoritative**.
   - **Relevance** is modelled by *cosine scores*.
   - **Authority** is typically a query-independent property of a document.

   To keep track of authority, we could assign a *quality score* $g(d) \in [0, 1]$ to each document, and consider a simple net score which combines cosine relevance and authority:

$$\text{net-score}(q, d) = \alpha g(d) + (1 - \alpha)\cos(q, d)$$

   where α is an arbitrary hyperparameter to balance the weighting of the terms. Then, we could seek the top k docs using the net score.

2. **Query term proximity**: users are likely to prefer docs where query terms occur close to each other. To improve the quality of the documents returned, we could collect candidates by running one or more queries on the indexes, before ranking.

   Specifically, we could apply the following steps given a query $q = q_1 q_2 q_3$:

   (a) Run q as a phrase query (e.g., using a positional index).
   (b) If $< k$ docs contain q, run the two phrase queries $q_1 q_2$ and $q_2 q_3$.
   (c) If we still have $< k$ docs, run the vector space query $q_1 q_2 q_3$.
   (d) Rank matching docs by vector space scoring, combining all information (possibly including a proximity score w).

3. **Parametric and zone indexes**: we may sometimes wish to search by **metadata** (e.g., author, title, date, etc.).

   - For **fields**, or regions with a *finite set* of possible values, we could utilize *range trees* (e.g., B-trees) for their queries.
   - For **zones**, or regions which can contain an *arbitrary amount of text*, we could build inverted indexes in order to permit their querying.

8 EVALUATION

Lecture 9
18th March 2022

Aside from the speed of indexing and searching, as well as the expressiveness of the query language, one of the key measures for evaluating a search engine is the **relevance** of the search results. There are 3 key elements for measuring relevance, namely:

1. a fixed document collection.
2. a fixed suite of queries.
3. a (binary) assessment of either relevant/non-relevant for each query and each document.

Note that relevance is assessed relative to the **information need**, rather than to the query.

8.1 *Metrics*

Some common metrics that are used to evaluate retrieved docs are:

- **Precision**:

$$\mathrm{P} = \frac{\text{number of relevant docs retrieved}}{\text{total number of docs retrieved}}$$

- **Recall**:

$$\mathrm{R} = \frac{\text{number of relevant docs retrieved}}{\text{total number of relevant docs}}$$

- F_β measure:

$$\mathrm{F} = \frac{1}{\alpha\frac{1}{\mathrm{P}} + (1-\alpha)\frac{1}{\mathrm{R}}} = \frac{(1+\beta^2)\mathrm{PR}}{\beta^2\mathrm{P} + \mathrm{R}} \tag{3}$$

  - Typically, a balanced F_1 measure (i.e., $\beta = 1$) is used.
  - When β increases, more emphasis is placed on recall; as β decreases, more emphasis is placed on precision.

Typically, precision decreases as either the number of docs retrieved, or the recall increases.

A **precision-recall curve** can be drawn by computing precision at different recall levels. Sometimes, we would use *interpolated precision* (i.e., taking the max of precision to the right of the value) instead of actual precision, since precision could increase with recall locally.

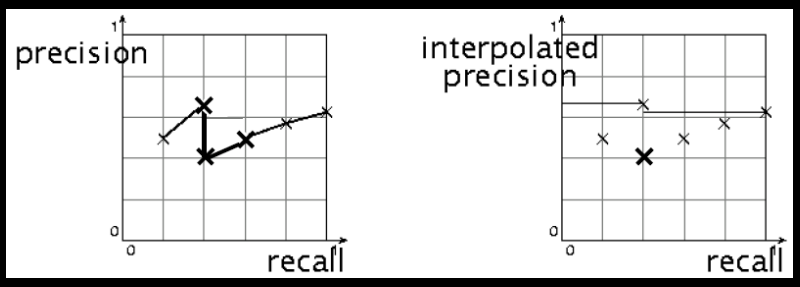

Figure 16: Illustration of interpolated precision.

Some other useful evaluation metrics include:

- Precision at *fixed retrieval level* (i.e., precision-at-k): precision of the top k results.
  - This is based on the understanding that most people want good matches on the first 1/2 result pages.
- 11-point interpolated average precision: taking the precision at 11 levels of recall (i.e., varying from 0 to 1 by tenths) of the documents, using interpolation, and averaging them.
- **Mean average precision (MAP)**: average of the precision value obtained for the top k documents, each time a relevant doc is retrieved.
- **R-precision**

8.2 *Test Collections*

Aside from evaluation metrics, we would also require **test queries** and **relevance assessments**. One useful measure for assessing inter-judge agreement is the **Kappa (κ) measure**:

Sometimes, a document can be redundant even if it is highly relevant (e.g., contains many duplicates). In such cases, **marginal relevance** could be a better measure of utility for the user.

$$\kappa = \frac{P(A) - P(E)}{1 - P(E)}$$

where

- $P(A)$ represents the proportion of docs which the judges agree (i.e., have the same opinion), and
- $P(E) = P(\text{non-relevant})^2 + P(\text{relevant})^2$ represents the proportion of time that the agreement occurs by chance.

The Kappa measure returns 0 for chance agreement, and 1 for total agreement. For > 2 judges, we could utilize average pairwise Kappas instead, or an ANOVA.

8.3 *A/B Testing*

The purpose of **A/B testing** is to test a single innovation (i.e., change) in a search engine that has already been published.

In such cases, we could *divert a small proportion of traffic* (e.g., 1%) to the new system with the innovation, and evaluate the innovation with an automatic **Overall Evaluation Criterion (OEC)** such as *clickthrough on first result* (i.e., number of clicks on the first result).

9 QUERY REFINEMENT

Lecture 10
25th March 2022

9.1 *Relevance Feedback*

To refine a query, we could attempt to gather relevance feedback (RF) with regards to the relevance of our results. This includes:

- Explicit feedback from users.
- Implicit feedback (e.g., clickstream mining).
- Blind/Pseudo relevance feedback.
  - **Blind feedback** automates the "manual" part of true RF, by assuming that *the top k docs are actually relevant*.

    * This works well on average, but can go wrong for some queries.
    * Additionally, employing several iterations of blind feedback could result in *query drift* (i.e., the modified query becomes very different from the original query).

Ideally, after updating the query based on RF, our results should return documents explicitly marked as relevant within the top k results. One way to perform this update utilizes the **Rocchio algorithm**:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|\mathrm{D}_r|}\sum_{\vec{d}_j\in\mathrm{D}_r}\vec{d}_j - \gamma\frac{1}{|\mathrm{D}_{nr}|}\sum_{\vec{d}_j\in\mathrm{D}_{nr}}\vec{d}_j \tag{4}$$

where:

- $\vec{q}_m$ represents the modified query.
- $\vec{q}_0$ represents the initial query.
- D_r represents the (small) set of *known relevant* doc vectors.
- D_{nr} represents the (small) set of *known non-relevant* doc vectors.
- $\vec{\mu}(\mathrm{D}) = \frac{1}{|\mathrm{D}|}\sum_{d\in\mathrm{D}}\vec{d}$ is the **centroid** (i.e., the center of mass) of a set of documents.
- $\{\alpha, \beta, \gamma\}$ are artificial weights, either hand-chosen or empirically determined.
    - Typically, we would want α to be relatively big (i.e., around 0.7), since we would still want some emphasis to be placed on the original query. α could be larger if we do not have sufficient information w.r.t. what is ir/relevant.
    - Also, typically $\beta > \gamma$, since the relevant docs gives us more useful information w.r.t. what we are looking for. There are usually too many non-relevant docs, most of which do not provide useful information.

Since the term weights in the query vector could potentially become negative via this update, we could set those weights to 0 to ignore those terms.

Subsequently, we could either employ a train/test split, or simply utilize the documents in the *residual collection* (i.e., set of documents minus those assessed as relevant) as the test set to evaluate the output based on the modified queries.

- Typically, a single round of query refinement based on RF is often very useful. Subsequent rounds of refinement is only marginally useful.
- Query refinement based on RF works best when the following assumptions hold:

1. User's initial query at least partially works.
2. The set of (non-)relevant documents are similar, or the term distribution in non-relevant documents is sufficiently distinct from that of relevant documents.
   - The effectiveness of RF will be limited if the relevant/non-relevant docs are internally grouped as clusters.

9.2 *Query Expansion*

The underlying idea of **query expansion** is to expand each query term t with the related words of t from a thesaurus (which could be manually compiled and/or automatically generated). Automatic thesaurus generation could be done based on the concept of **distributional similarity**: i.e., two words are deemed to be similar if they **co-occur/share same grammatical relations** with similar words.

Some examples of manually compiled thesauri include MeSH and WordNet.

Query expansion generally increases *recall*, but may decrease *precision* when the terms are ambiguous (e.g., "interest rate" → "interest rate fascinate evaluate").

10 XML RETRIEVAL

As compared to unstructured retrieval as discussed in previous sections, XML retrieval involves the retrieval of information from more *structured data*.

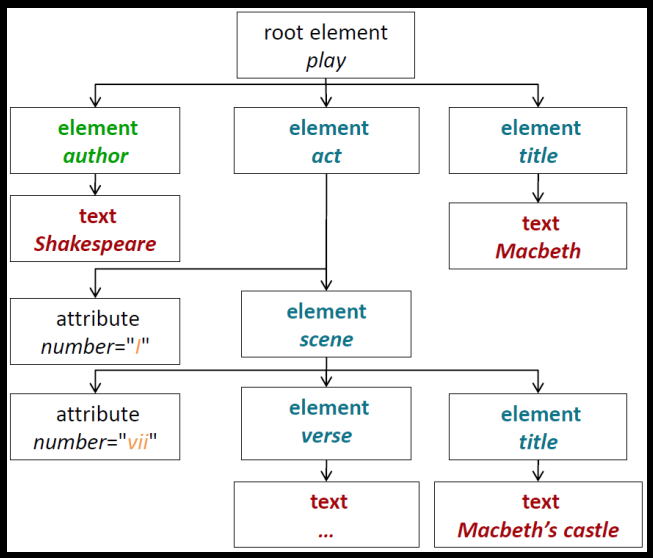

Figure 17: A sample XML document.

An example of an XML document is shown in figure 17. In this document,

- internal nodes encode **document structure** or **metadata**.
- leaf nodes consist of **text**.
- additionally, an element can have one or more **attributes** and *sub-elements*.

10.1 *Vector Space Model for XML IR*

One way of performing XML retrieval is akin to retrieval on unstructured data, i.e., to utilize a vector space model. However, instead of representing each word as a dimension, each dimension in the vector space would now encode a word + its position within the XML tree.

More specifically, the construction of the vector space model is described as follows:

1. Take each text node (i.e., leaf) and break it into multiple nodes, one for each word.
   - e.g., book/name#"tom jerry" → book/name#"tom" + book/name#"jerry"
2. Extract the **lexicalized subtrees** (i.e., subtrees that contain ≥ 1 vocabulary term).
   - e.g., extract book/name#"tom" & book/name#"jerry", ignore the subtree book/name (i.e., without the text)
3. Index all paths that *end in a single vocabulary term* (i.e., all XML-context-term pairs).
   - Each pair is called a **structural term**, denoted by $< c, t >$.
   - e.g., index book/name#"tom" & book/name#"jerry"

How do we define similarity between a query and a structural term in XML? One simple measure is **context resemblance** (C_R), defined by

$$C_R(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases} \tag{5}$$

where:

- c_q represents a structural term in a query.
- c_d represents a structural term in a document.
- $|c_q|$ and $|c_d|$ represents the number of nodes in the terms (i.e., length of context + word), respectively.
- we say c_q **matches** c_d iff we can transform c_q into c_d by *inserting additional nodes*.

"no merge" because each context is calculated separately.

The final score for a document is then computed as a variant of the cosine measure, called **SimNoMerge**.

$$\text{SimNoMerge}(q, d) = \sum_{c_k \in B} \sum_{c_l \in B} C_R(c_k, c_l) \sum_{t \in V} w(q, t, c_k) \frac{w(d, t, c_l)}{\sqrt{\sum_{\substack{c \in B \\ t \in V}} w(d, t, c)^2}} \tag{6}$$

where:

- V represents the vocabulary of non-structural terms.
- B represents the set of all XML contexts.
- $w(q, t, c)$ and $w(d, t, c)$ are the weights of term t in XML context c in query q and document d respectively.

Note that SimNoMerge is not a true cosine measure, since its value can be larger than 1.

10.2 *XML IR Evaluation*

component coverage: evaluates whether the element retrieved is "structurally" correct, i.e., neither too low nor too high in the tree

To evaluate the results of retrieval in XML, we could employ the **INEX** (i.e., INitiative for the Evaluation of Xml retrieval), which defines **component coverage** as an additional dimension of relevance.

Specifically, there are four cases of component coverage:

- **(E) Exact coverage**: the information sought is the main topic of the component, and the component is a meaningful unit of information.
- **(S) Too small**: the information sought is the main topic of the component, but the component is not a meaningful (self-contained) unit of information.
- **(L) Too large**: the information sought is present in the component, but is not the main topic.
- **(N) No coverage**: the information sought is not a topic of the component.

and four levels of topical relevance:

- **(3) Highly relevant**
- **(2) Fairly relevant**
- **(1) Marginally relevant**
- **(0) Non-relevant**

The *relevance-coverage combinations* are then quantized as follows:

$$\mathrm{Q}(\mathbf{rel}, \mathbf{cov}) = \begin{cases} 1.00 & \text{if } (\mathrm{rel}, \mathrm{cov}) = 3\mathrm{E} \\ 0.75 & \text{if } (\mathrm{rel}, \mathrm{cov}) \in \{2\mathrm{E}, 3\mathrm{L}, 3\mathrm{S}\} \\ 0.50 & \text{if } (\mathrm{rel}, \mathrm{cov}) \in \{1\mathrm{E}, 2\mathrm{L}, 2\mathrm{S}\} \\ 0.25 & \text{if } (\mathrm{rel}, \mathrm{cov}) \in \{1\mathrm{S}, 1\mathrm{L}\} \\ 0.00 & \text{if } (\mathrm{rel}, \mathrm{cov}) = 0\mathrm{N} \end{cases}$$

The number of relevant components in a retrieved set A of components can then be computed as

$$\#(\text{relevant items retrieved}) = \sum_{c \in \mathrm{A}} \mathrm{Q}\Big(\mathrm{rel}(c), \mathrm{cov}(c)\Big)$$

which could be in turn used for precision/recall/F1 computations (e.g., in equation 3).

11 PROBABILISTIC INFORMATION RETRIEVAL

Lecture 11
1st April 2022

An IR system has an uncertain understanding of a user's information need, and must make an uncertain guess of whether a document satisfies the query. To reason under uncertainty, probabilistic models (employing probability theory) can be applied to estimate the relevance of a document to a query.

Some of the key concepts in probability theory include:

- **Chain rule**: $\mathrm{P}(\mathrm{A}, \mathrm{B}) = \mathrm{P}(\mathrm{A}|\mathrm{B})\mathrm{P}(\mathrm{B}) = \mathrm{P}(\mathrm{B}|\mathrm{A})\mathrm{P}(\mathrm{A})$
- **Partition rule**: $\mathrm{P}(\mathrm{B}) = \sum_i \mathrm{P}(\mathrm{A}_i, \mathrm{B})$
- **Bayes' rule**: $\mathrm{P}(\mathrm{A}|\mathrm{B}) = \frac{\mathrm{P}(\mathrm{B}|\mathrm{A})\mathrm{P}(\mathrm{A})}{\mathrm{P}(\mathrm{B})}$
- **Odds**: $\mathrm{O}(\mathrm{A}) = \frac{\mathrm{P}(\mathrm{A})}{\mathrm{P}(\overline{\mathrm{A}})}$

11.1 *Classic Probabilistic Retrieval Models*

In classic models, we assume a **binary** notion of relevance ($\mathrm{R}_{d,q}$), where

$$\mathrm{R}_{d,q} = \begin{cases} 1 & \text{if } d \text{ is relevant to } q \\ 0 & \text{else} \end{cases}$$

In probabilistic ranking, the **probability ranking principle (PRP)** is adhered to, i.e., documents are ordered *decreasingly* by their estimated probability of

relevance to the query.

11.1.1 Binary Independence Model (BIM)

One model which is typically used with the PRP is the **BIM**, which assumes:

- **Binary** term incidence vector representation for documents and queries: each document d is represented by a vector $\vec{x} = (x_1, \ldots, x_m)$ where $x_t = 1$ if term t occurs in d, and 0 otherwise.
- **Independence**: no association between terms.

In the model, $\mathrm{P}(\mathrm{R}_{d,q}) = \mathrm{P}(\mathrm{R}|d, q)$ is modelled using term incidence vectors as $\mathrm{P}(\mathrm{R}|\vec{x}, \vec{q})$. Thus, we have

$$\mathrm{P}(\mathrm{R}|d, q) = \mathrm{P}(\mathrm{R}|\vec{x}, \vec{q}) = \frac{\mathrm{P}(\vec{x}|\mathrm{R}, \vec{q})\mathrm{P}(\mathrm{R}|\vec{q})}{\mathrm{P}(\vec{x}|\vec{q})} \tag{7}$$

where:

- $\mathrm{P}(\vec{x}|\mathrm{R}, \vec{q})$: probability that if a relevant document is retrieved, that document's representation is $\vec{x}$.
- $\mathrm{P}(\mathrm{R}|\vec{q})$: *prior* probability of retrieving a relevant document for query q.
- $\mathrm{P}(\vec{x}|\vec{q})$: probability that given a query q, there exists a document whose representation is $\vec{x}$.

To make the ranking process more efficient, we can rank documents by their odds of relevance instead (and ignore the common denominator during computation):

$$
\begin{aligned}
\mathrm{O}(\mathrm{R}|\vec{x},\vec{q}) &= \frac{\mathrm{P}(\mathrm{R}=1|\vec{x},\vec{q})}{\mathrm{P}(\mathrm{R}=0|\vec{x},\vec{q})} \\
&= \frac{\mathrm{P}(\mathrm{R}=1|\vec{q})}{\mathrm{P}(\mathrm{R}=0|\vec{q})} \times \frac{\mathrm{P}(\vec{x}|\mathrm{R}=1,\vec{q})}{\mathrm{P}(\vec{x}|\mathrm{R}=0,\vec{q})} \\
&= \frac{\mathrm{P}(\vec{x}|\mathrm{R}=1,\vec{q})}{\mathrm{P}(\vec{x}|\mathrm{R}=0,\vec{q})} && \text{dropping of constant term} \\
&= \prod_{t=1}^{\mathrm{M}} \frac{\mathrm{P}(x_t|\mathrm{R}=1,\vec{q})}{\mathrm{P}(x_t|\mathrm{R}=0,\vec{q})} && \text{conditional independence assumption} \\
&= \prod_{t:x_t=1} \frac{\mathrm{P}(x_t=1|\mathrm{R}=1,\vec{q})}{\mathrm{P}(x_t=1|\mathrm{R}=0,\vec{q})} \prod_{t:x_t=0} \frac{\mathrm{P}(x_t=0|\mathrm{R}=1,\vec{q})}{\mathrm{P}(x_t=0|\mathrm{R}=0,\vec{q})} && \text{separation of terms} \\
&= \prod_{t:x_t=1} \frac{p_t}{u_t} \prod_{t:x_t=0} \frac{1-p_t}{1-u_t} && \text{replace terms with } p_t \text{ and } u_t \\
&= \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{p_t}{u_t} \prod_{\substack{t:x_t=0 \\ q_t=1}} \frac{1-p_t}{1-u_t} && \text{only consider terms appearing in } q \\
&= \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{p_t}{u_t} \left(\prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{1-u_t}{1-p_t} \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{1-p_t}{1-u_t} \right) \prod_{\substack{t:x_t=0 \\ q_t=1}} \frac{1-p_t}{1-u_t} \\
&= \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{p_t(1-u_t)}{u_t(1-p_t)} \prod_{\substack{t \\ q_t=1}} \frac{1-p_t}{1-u_t} \\
&= \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{p_t(1-u_t)}{u_t(1-p_t)} && \text{dropping of constant term}
\end{aligned}
$$

where:

- $p_t = \mathrm{P}(x_t = 1|\mathrm{R} = 1, \vec{q})$: probability of a term appearing in a relevant document.
- $u_t = \mathrm{P}(x_t = 1|\mathrm{R} = 0, \vec{q})$: probability of a term appearing in a non-relevant document.

Subsequently, the **Retrieval Status Value (RSV)** of a document d is defined by taking the log of the product:

$$\mathrm{RSV}_d = \log \prod_{\substack{t:x_t=1 \\ q_t=1}} \frac{p_t(1-u_t)}{u_t(1-p_t)} = \sum_{\substack{t:x_t=1 \\ q_t=1}} \log \frac{p_t(1-u_t)}{u_t(1-p_t)} = \sum_{\substack{t:x_t=1 \\ q_t=1}} c_t \tag{8}$$

- In theory, we have $c_t = \log \frac{p_t(1-u_t)}{u_t(1-p_t)} = \log \frac{s/(\mathrm{S}-s)}{(\mathrm{df}_t-s)/((\mathrm{N}-\mathrm{df}_t)-(\mathrm{S}-s))}$ where
  - s is the number of relevant documents containing each term t in a query.
  - S is the total number of relevant documents.
  - df_t is the document frequency of t.
  - N is the collection size.

	documents	relevant	nonrelevant	Total
Term present	$x_t = 1$	s	$\mathrm{df}_t - s$	df_t
Term absent	$x_t = 0$	$S - s$	$(N - \mathrm{df}_t) - (S - s)$	$N - \mathrm{df}_t$
	Total	S	$N - S$	N

Figure 18: Illustration of the meaning of each term.

  However, it is unlikely that we would be able to get the values for all these variables in reality.
- Hence in practice, we could assume that the set of relevant documents is a very small percentage of the collection, i.e., $u_t = \frac{\mathrm{df}_t - s}{\mathrm{N}-\mathrm{S}} \approx \frac{\mathrm{df}_t}{\mathrm{N}}$. Hence, we have $\log \frac{1-u_t}{u_t} = \log \frac{1-\mathrm{df}_t/\mathrm{N}}{\mathrm{df}_t/\mathrm{N}} = \log \frac{\mathrm{N}-\mathrm{df}_t}{\mathrm{df}_t} \approx \log \frac{\mathrm{N}}{\mathrm{df}_t} = \mathrm{idf}_t$.

  On the other hand, the statistics of the relevant documents (p_t) can be estimated either:
  - using the frequency of term occurrences in known relevant documents.
  - by setting it as a constant (the RSV would just be the idf in this case).

11.1.2 Nonbinary Models

The **Okapi BM25** utilizes the second approach for estimating p_t, along with the factoring in of the term frequency and document length:

$$\mathrm{RSV}_d = \sum_{t \in q} \log \left(\frac{\mathrm{N}}{\mathrm{df}_t} \times \frac{(k_1+1)\mathrm{tf}_{t,d}}{k_1\left((1-b) + b(\mathrm{L}_d/\mathrm{L}_{\mathrm{ave}})\right) + \mathrm{tf}_{t,d}} \right)$$

where:

- $\mathrm{tf}_{t,d}$: term frequency of term t in document d.

- L_d: length of document d.
- L_{ave}: average document length of all documents in the collection.
- k_1: tuning parameter controlling document term frequency scaling.
- b: tuning parameter controlling the scaling by document length.

If the query is long, we might also utilize similar weighting for query terms:

$$RSV_d = \sum_{t \in q} \log \left(\frac{N}{df_t} \times \frac{(k_1 + 1)tf_{t,d}}{k_1 ((1 - b) + b(L_d/L_{ave})) + tf_{t,d}} \times \frac{(k_3 + 1)tf_{t,q}}{k_3 + tf_{t,q}} \right) \quad (9)$$

where k_3 is a tuning parameter controlling the query term frequency scaling.

In practice, the difference between vector space models and probabilistic IR models is not very significant. The main difference is that the scoring/ranking of queries is not based on cosine similarity and tf-idf values, but via a formula motivated by probability theory.

11.2 *Language Models for IR*

Language models can also assist in the ranking of documents based on $P(d|q)$, i.e., the probability of d being relevant given q. As seen previously, this probability can be computed via Bayes' rule:

$$\begin{aligned} P(d|q) &= \frac{P(q|d)P(d)}{P(q)} && \text{Bayes' rule} \\ &= P(d) \prod_{\text{distinct } t \in q} P(t|d) && \text{drop constant term, apply CI assumption} \\ &= P(d) \prod_{\text{distinct } t \in q} \frac{tf_{t,d}}{|d|} && \text{apply MLE} \end{aligned}$$

Recall that some terms could have term frequencies of 0. To address this, we could apply a **mixture model** to smooth the estimate to avoid zeroes. Let M_d represent the document model (i.e., basically an equivalent alternative way of writing d) and M_c represent the collection model used for smoothing, where $P(t|M_c) = \frac{cf_t}{\sum_{t'} cf_{t'}}$. Hence, $P(t|d) = \lambda P(t|M_d) + (1 - \lambda)P(t|M_c)$, and $P(d|q)$ can be computed as

$$P(d|q) = P(d) \prod_{\text{distinct } t \in q} \Big(\lambda P(t|M_d) + (1 - \lambda)P(t|M_c) \Big)$$

12 CRAWLING AND LINK ANALYSIS

Lecture 12
8th April 2022

12.1 *Crawling*

Crawling enables web search engines to retrieve the content of web pages. As compared to other IR systems, crawling is significantly more complex (e.g., we need to take into account the bandwidth and latency of both the server and the crawler) and large scale.

Assuming that the web is well-linked, the basic crawler operation can be described as follows:

1. Initialize queue with URLs of known seed pages.
2. Repeat:
   (a) Take URL from queue.
   (b) Fetch and parse page.
   (c) Extract URLs from page.
   (d) Add URLs to queue.
   (e) Call the indexer to index the page.

However, it is also important for crawlers to be able to address these issues:

- **Politeness**: the crawler should only crawl the allowed pages of a website. This can be done via these aspects of a crawler:
  - URL filter: filters out pages which should not be crawled.
    * Typically, each website would have a robots exclusion protocol/standard, i.e., `robots.txt`, which specifies the limited access for crawlers to their pages.
    * Crawlers *should* (i.e., malicious ones could still ignore) the specified restrictions as stated on `robots.txt`.
    * To avoid repeated checking of which websites are allowed/forbidden, crawlers could caches the `robots.txt` file of each site.
  - URL frontier: a queue (more complex than simple FIFO) which holds and manages URLS which were encountered but not yet crawled.
    * Ideally, this data structure would ensure that multiple pages from the same host would not be fetched at the same time/in quick succession, to prevent overloading the server.
- **Duplicate detection**: crawlers must be able to avoid duplicated contents and URLs.
  - Contents: for each page, we could check whether the content is already in the index, by using document fingerprints or shingles (i.e., quick summaries of the document).

  - URLs:
    * Duplicate elimination: we could ignore the URLs which have been seen before.
    * Normalization: we need to normalize (i.e., expand) all relative URLs, to prevent duplication.
    * Freshness: some pages may be updated more often than others (e.g., news sites), thus it is ideal for these pages to be re-crawled more frequently.

- **Scalability**: to make the crawling process scalable, we would typically run multiple crawl threads in parallel, and/or use different geographically distributed nodes.

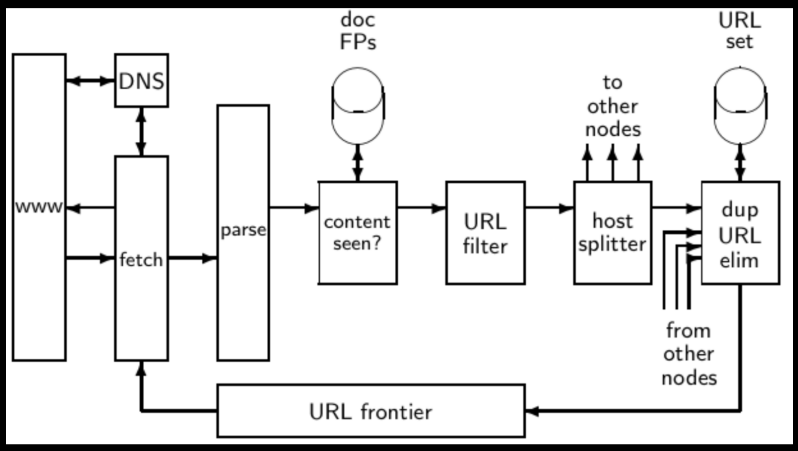

Figure 19: Architecture for distributed crawling.

- **Spider traps**: set of webpages which cause the crawler to go into an infinite loop (or crash). Some examples include:
  - P_1 contains a single link to P_2, which contains a single link back to P_1.
  - Dynamic links (e.g., calendars with dynamic links to the next month).

12.2 *Link Analysis*

Anchor text is loosely defined as the text surrounding a hyperlink.

- Typically, we can assume that the anchor text of a link to a page p_i describes the content of p_i.
- In fact, empirical evidence has shown that anchor text is often a better description of a page's content than the page itself (e.g., of malicious/clickbait pages). Therefore, they can be weighted more highly

than the text within the document when scoring the relevance of a page to a given query.

12.2.1 PageRank

Aside from anchor text, another natural (and generally valid) assumption is that a hyperlink to a page p_i is a quality signal (i.e., a hyperlink $p_0 \rightarrow p_i$ indicates that the author of p_0 deems p_i high-quality and relevant). To elaborate further:

- it is better to have more endorsements (i.e., if $\mathrm{P}_1 = \{p_{a_0}, \ldots, p_{a_m}\}$, $\mathrm{P}_2 = \{p_{b_0}, \ldots, p_{b_n}\}$ with $|\mathrm{P}_1| > |\mathrm{P}_2|$, then if $\mathrm{P}_1 \rightarrow p_i$ and $\mathrm{P}_2 \rightarrow p_j$, we say that p_i is better than p_j).
- it is better for the endorsement to come from an important source (i.e., if P_1 contains more important/credible sources than P_2, then p_i is better than p_j).
- it is better for the endorsement to be exclusive (i.e., if the pages in P_1 only contain a link to p_i, whereas the pages in P_2 contain links to multiple other pages, then p_i is better than p_j).

Based on these assumptions, the idea behind PageRank (which models the behaviour of a random surfer) can be described as follows:

1. Start at a random page.
2. At each step, follow one of the n links on that page (i.e., each with probability $\frac{1}{n}$).
    - When a node has no outlinks, we will teleport to a random page (with equal probability).
    - Else, with probability α (e.g., 10%), we will teleport to a random page, and follow a random link on the page instead with probability $(1 - \alpha)$.
3. Do this repeatedly, and use the *long-term visit rate* as the page's score.

This random walk process can be described as a first-order Markov chain, consisting of n states and an $n \times n$ transition probability matrix A (where A_{ik} is the probability of going from state i to state k).

12.2.2 Ergodic Markov Chains

How can we be sure that the long-term visit rate for each state will be a fixed number? Based on the ergodic theorem, when a Markov chain is ergodic,

the long-term visit rate of each state on the chain would be a stable value. Specifically, a Markov chain is **ergodic** if:

- there is a path from any state to any other, and
- we can be in any state at every time step with non-zero probability.

With the property of teleportation, our Markov chain is ergodic, and thus the ergodic theorem holds for our scoring scheme. In other words, over a long period of time, we will visit each state/page in proportion to a unique/fixed rate (known as the *long-term visit rate*) regardless of our initial/starting position.

Let a represent the steady state which would be eventually reached by our algorithm. The PageRank algorithm can thus be described as follows:

1. Start with *any* probability distribution (e.g., $x = (10\ldots0)$).
2. Multiply x by increasing powers of A (i.e., the transition matrix) until the product $x\mathrm{A}^k$ looks stable.
3. Eventually, we have $x\mathrm{A}^k \approx a$, and we obtain the long-term visit rates of each state.

However, in reality, there are several components that are just as important as PageRank in ranking webpages (e.g., anchor text, proximity, tiered indexes, etc.). In fact, PageRank is typically used together with cosine similarity scores. Therefore, PageRank in its original form often has a negligible impact on ranking.

* * *