Web information retreival summary

Eran Hashmonay

July 23, 2021

1 Storage

Collection: A set of documents

Token / Term: "word" appearing in at least one document.

Vocabulary size: Number of **different** tokens appearing in the collection. Cannot assume the vocabulary is finite (grows with collection size).

Collection size: Number of tokens appearing in the collection.

Heap's law:

- Heap's law is not provable, it is a common characteristic observed in many data bases but it is not necessarily always true.
- M: size of the vocabulary
- T: Number of tokens in the collection
- k and b are parameters related to the actual collection (not the size). Typically: $30 \leq k \leq 100, b \approx 0.5$
- $M = kT^b$

Zipf's law: The i-th most frequent term has frequency proportional to $\frac{1}{i}$

- cf_i: number of occurrances of the i-th most frequent token
- K: a normalizing constant
- $cf_i = \frac{K}{i}$

1.1 Boolean retrieval:

Simplifying assamption 1: The user accurately translates his task into a query (boolean combination of keywords)

Simplifying assamption 2: A document is relevant to the user's task iff it satisfies the boolean combination of keywords.

1.2 Dictionary:

- When remembering a word in the dictionary we need to remember:
  - The word itself
  - The pointer the the inverted index (list of files that contain the word)
  - The frequency of the word in the files. Important to understand the length of the file list containing the word and for grading.
- log access
- small, fit in main memory

- Table size must be constant to allow random access which is a requirement for binary search.

space complexity calculation:

- M: number of terms.
- w: average word length in bytes.
- x: length of the longest word in bytes
- tf: term frequency byte size
- p: index pointer byte size
- ps: term pointer byte size, $ps = \left\lceil \frac{\lceil \log_2 \text{string length} \rceil}{8} \right\rceil$

1.2.1 Concatenated string:

Solves the "constant entry size" problem.

Store dictionary as a string of characters.

Tokens are accessed with a table that saves the token's index informtion and a pointer to it's location on the string.

We save a small constant of bytes per entry for the term pointer instead of an estimate of the actuall term string (that would be not correct and wastefull in space).

- The terms sorted alphabetically before concatenation.
- We can tell where the next term starts by looking at the pointer of the next table entry.
- term pointer requires $\log_2$ [string_length] bits.
- We store a table for each word with it's pointer to inverted index and it's frequency. In the table, we also store a pointer to the concatenated string.
- Table entry size is constant and sorted according to the concatenated string so we can run binary search on it.

Space complexity:

string length $= M * w$

table size $= M * (tf + p + ps)$

dictionary size $=$ string length $+$ table size $= M * w + M * (tf + p + ps)$

1.2.2 Blocking:

Instead of storing a pointer for each term, we store a pointer to every k-th term.

In order to know where words end, we also store term lengths.

- Don't need to save the length of the last term in each block because we know the next pointer.

Space complexity:

string length $= M * w$

table size $= M * (tf + p) + \frac{M}{k} * ps + M * \frac{k-1}{k} * \left\lceil \frac{\log_2 x}{8} \right\rceil$

dictionary size $=$ string length $+$ table size $= M * w + M * (tf + p) + \frac{M}{k} * ps + M * \frac{k-1}{k} * \left\lceil \frac{\log_2 x}{8} \right\rceil$

1.2.3 Front coding:

Because the terms are sorted alphabetically, it is common that adjacent words have common prefixes.

Instead of a full concatenated string, remove the common prefix of each word and concatenate the remaining postfix.

To find the term, store the common prefix size and a term pointer in the table.

Because we can't find the missing prefix, we **can't use log search** with this method.

Space complexity:

wpre: average prefix size bytes

string length $= M * (w - wpre)$

table size $= M * (tf + p + ps + \log_2 wpre)$

dictionary size $=$ string length $+$ table size $= M * (w - wpre) + M * (tf + p + ps + \log_2 wpre)$

1.2.4 (Best method learned) k-1 in k front coding:

- Dictionary is stored as a string of characters where the words are sorted alphabetically and concatenated together.
- The string is divided into blocks of k words (k is constant).
- We store a pointer to the beginning of each block. Instead of pointers to beginning of words.
- For each word (except for the last in each block), we store it's length to know where the next start. If the word is the last in It's block, we know it's length because we have a pointer to the beginning of the next block.
- In each block, the first word is completely given and the rest are given in postfixes where the prefix is the same as the first word. We store the prefix size to know the word and where it begins.
- When searching for a word, we can find the right block in log time and then iterate on the block.

Space complexity:

wpre: average prefix size bytes

string length $= \frac{M}{k} * w + (M * \frac{k-1}{k}) * (w - wpre)$

$ps = \left\lceil \frac{\lceil \log_2 \text{string length} \rceil}{8} \right\rceil$

table size $= M * (tf + p) + \frac{M}{k} * ps + (M * \frac{k-1}{k}) * \lceil \log_2 wpre + \log_2 x \rceil$

dictionary size $=$ string length $+$ table size $=$

$\frac{M}{k} * w + (M * \frac{k-1}{k}) * (w - wpre) + M * (tf + p) + \frac{M}{k} * ps + (M * \frac{k-1}{k}) * \lceil \log_2 wpre + \log_2 x \rceil =$

$M * (tf + p) + \frac{M}{k} * (w + ps) + (M * \frac{k-1}{k}) * (w - wpre + \lceil \log_2 wpre \rceil + \lceil \log_2 x \rceil)$

1.3 Inverted index:

- The inverted index is a set of posting lists, one for each term in the lexicon.
- In general, for a given term, we want to save it's docID and it's positions in the document. To save the positions we would also need to save the number of appearances in each document. The template for the posting list is (docID, num_of_appearances, position1, position2, ...).
- Example: if the term "DO" appears once in doc1 in position 2 and twice in doc 3 in positions 10, 20 then DO's posting list will be $(1, 1, 2), (3, 2, 10, 20)$
- Actually the numbers are stored without commas and parenthisis, like so: 112321020
- The inverted index is large so we want to compress it:
  - 0/1 bitmap for very common words.
  - save difference (gaps) between docIDs instead of the IDs themselves. The first docID is saved as it is and the next ID is saved as the difference from the last ID to the actual current one. Good for uncommon words.
  - Code each number to the number of bits required to write it (logn) this is done with variable bit/byte.
  - Compression can be categorized in 2 ways: variable bit/byte and parameterized/non parameterized.

1.3.1 Variable bit compression:

1. **Unary code:** Represent the number n as (n-1) 1's and a single 0 to tell it's the end of the number.
2. **Gamma code:** Represent a gap, G as a pair (length, offset):
   - Offset: is G in binary with leading bit cut off. offset of 5 (101) is 01
   - Length: is the length of binary code. We encode length with unary code. length of 5 (101) is 110
   - Example: Gamma code of 5 (101) is 11001
   - G is encoded using $2 * \lfloor \log(G) \rfloor + 1$ bits.
   - All gamma codes have an odd number of bits.
   - Uniquely prefix-decodable.
   - Parmeter free.
3. **Delta code:** Same as gamma code but the length itself is encoded in gamma code.

Disadvantages of variable bit codes:

- Machines have word bounderies and operations that cross word bounderies are slow. Thus compressing and manipulating at the granularity of bits can be slow. Variable byte encoding is aligned and thus potentially more efficient.
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost.

1.3.2 Variable byte compression:

- numbers are in byte units.
- given a gap value G we want to use the fewest bytes needed to hold log G.

1. **Varint:** Begin with 1 byte to store G and dedicate 1 bit in it to be a continuation bit c.
   - If $G \leq 127$ binary encode it in the available 7 bits.
   - Else, encode G's higher order 7 bits end then use additional bytes to encode the next 7 higher order bits using the same algorithm.
   - At the end, set the continuation bit of the last byte to 1 and the other continuation bits to 0.
2. **Length precoded varint:** Use the first 2 bits of the first byte to indicate the number of bytes used.
3. **Group varint encoding:** Use first byte to encode the length of 4 int variables.

1.3.3 Parametrized variable bit compression:

In parametrized encoding we assume there is a collection of symbols that we want to encode and the probability of each symbol to appear.

A parameterized encoding gets the probability distribution of the input symbols, and creates encodings accordingly.

Given:

S: a set of symbols.

P(s): each symbol has an associated probability P(s).

- Shannon's lower bound on the average number of bits needed per symbol, $s \in S$ is: $\sum_{s \in S} -P(s) \log P(s)$
- Roughly speaking, each symbol s with probability P(s) needs at least $-\log(P(s))$ bits to represent.
- Ideally, we aim to find a compression method that reaches Shannon's bound.

Huffman codes:

- Can be used to compress the dictionary string.
- Optimal for encodings in which no codeword is a prefix of another.
- Input: A set of symbols, along with a probability of each symbol.
- Intuition: common symbols get short encodings, and rare symbols get long encodings.

Creating a huffman code (tree):

- Create a node for each symbol and assign it with the probability of the symbol.
- While there is more than 1 node without a parent:
  - choose 2 nodes with the lowest probabilities and create a new node with both nodes as children.
  - Assign the new node the sum of probabilities of children.
- The tree derived gives the code for each symbol (leftwards is 0, and rightwards is 1).

Canonical huffman code:

- Each symbol is coded separately.
- Each symbol uses a whole number of bits (?).
- Can be very inefficient when there are extremely likely/unlikely values.

We only need

1. The list of symbols, ordered by their codeword length
2. the number of codewords for each length.

Properties:

1. Codewords of given length are consecutive binary numbers.
2. Given 2 symbols s, s' with codewords of the same length then: $cw(s) < cw(s') \iff s < s'$ (second priority of sort is alphabetic).
3. The first shortest codeword is a string of 0's.
4. The last longest codeword is a string of 1's.
5. $c = 2^{j-i} * (d+1)$ where:
   (a) d: last codeword of length i.
   (b) j: the next length of codewords appearing in the code.
   (c) c: the first codeword of length j.

- To find canonical code, create a regular huffman tree and choose codeword lengths according to the tree. Then we can delete the tree.
  - Note that we create the tree only to determine the code lengths for each symbol, not the codes themselves.
- The final code is represented by a list of the numbers of symbols in each length followed by the symbols as they are sorted.

Example: 0232BEAFGCD: There are zero symbols of length 1, two of length 2, three of length 3 and two of length 4. Using the properties above, we can encode into bits.

Pseudo code for decoding:

```
• i = 0
• repeat until $d \le c_i + |a_i| - 1$ :
    – i = i + 1
    – let d be the word derived by reading i symbols.
• return the $d - c_i + 1$ th symbol of length i
```

Arithmetic coding:

- Input: set of symbols, S with corresponding probabilities and an input text to compress.
- Output: n, length of the input text and k, a single number (written in binary) in the range $[0, 1)$.
- Size:
  - To store a number in an interval of size high-low, we need log(high - low) bits.
  - The size of the final interval is $\prod_{i=1}^{n} P(s_i)$, and needs $-\log\left(\prod_{i=1}^{n} P(s_i)\right) = \sum_{i=1}^{n} -\log\left(P(s_i)\right)$ bit.
  - In practice, a little more since we can only write an integral number of bits.
  - In order to decode, the probabilities of each symbol must be known. This must be stored, which adds to overhead.

Encoding pseudo code (S):

```
• low = 0
• high = 1
• for i = 1 to n:
    – (low, high) = Restrict(low, high, $s_i$)
• return any number in range $[low, high)$
```

Restrict $(low, high, s_i)$:

```
• low_bound = $sum\left\{P(s) \mid s \in S \wedge s <_{alphabetic} s_i\right\}$
• high_bound = low_bound + $P(s_i)$
• range = high - low
• new_low = low + range * low_bound
• new_high = low + range * high_bound
• return (new_low, new_high)
```

Decoding pseudo code(n, k):

```
• [low, high) = $[0, 1)$
• for i = 1 to n:
    – for each $s \in S$:
        ∗ [new_low, new_high) = Restrict(low, high, s)
        ∗ if k in [new_low, new_high) then:
            · output "s"
            · [low, high) = [new_low, new_high)
            · break
```

Adaptive arithmetic coding:

- The probabilities may change over the course of the text .
- Compute probabilities from frequency of occurrences.
- Intuitively:
  - Start by assuming equal probabilities, i.e., that each symbol appears once.
  - At each occurence of a symbol in the text, increase its frequency, and adjust all probabilities.
  - Besides, the adjusting of the probabilities, encoding (decoding) is as described before.

2 Tolerant retreival

- The user has a task which he formulates as a query.
- Note: Relevance is subjective and can only be determined by the user.
- Goals:
  - Return all documents relevant to user task.
  - Return no non-relevant documents.
  - Return relevant documents "earlier".
- Suppose that a search engine runs a query Q and returns the result set R. Then, a document falls in one of the 4 categories of the 2 binary variables: retrieved and relevant.

2.1 Quality measures:

- Precision $= \frac{|\text{ relevant } \wedge \text{ retreived}|}{|\text{retreived}|}$
- Recall $= \frac{|\text{ relevant } \wedge \text{ retreived}|}{|\text{relevant}|}$
- F-measure $= 2 * \left(\frac{\text{precision } * \text{ recall}}{\text{precision } + \text{ recall}}\right)$
- Precision at k: $\frac{|\{d_i : i \leq k \wedge d_i \text{ is relevant}\}|}{k}$

2.2 Normalizing data:

2.2.1 Case folding:

- Case folding: convert all words to lower-case before storing in the lexicon.
- Users query is converted to lower-case before looking for the term index.

2.2.2 Stemming:

Porter Stemmer:

A multi-step, longest-match stemmer.

Cannot lower recall.

Notation:

- V: vowel(s) (AEIOU)
- C: consonant(s)
- $(VC)_m$: vowel(s) followed by consonant(s) repeated m times.

Any word can be written as $[C]\,(VC)_m\,[V]$

- Brackets are optional
- m is called the measure of the word.

Porter Stemmer rules (some of them):

step 1a:

Suffix	Replacement	Examples
sses	ss	caresses $\Longrightarrow$ caress
$(m > 0)$ ies	i	ponies $\Longrightarrow$ poni
ss	ss	caress $\Longrightarrow$ caress
s	null	cats $\Longrightarrow$ cat

step 1b:

Conditions	Suffix	Replacement	Examples
$m > 0$	eed	ee	feed $\Longrightarrow$ feed, agreed $\Longrightarrow$ agree
$(*V*)$	ed	null	plastered $\Longrightarrow$ plaster
$(*V*)$	ing	null	motoring $\Longrightarrow$ motor

V the prefix contains a vowel

2.2.3 Stop words:

- Very common words that are generaly not important, e.g.: “the”, “a”, “to”.
- These words usually don’t improve the results.
- Some search engins remove these words completely.

2.3 Wild card queries:

wild card example: “su*s”: should match all words starting with “su” and ending with “s”.

2.3.1 n-grams:

Sequences of n letters in a word.

- bigram := 2-gram

Use “$” to mark beginning and end of word.

- 2-gram of “labor” is: l, la, ab, bo, or, r

Given a word with k letters (not including surrounding $ symbols), the word will have:

- k+1 bigrams
- $k + 2 - (n - 1)$ n-grams

n-grams index:

- To store n-grams, we create an index that maps each n-gram to a list of pointers to the terms in the lexicon that contain that n-gram.
- n-gram index is stored in alphabetic order for easy search.

- Assume terms in the dictionary are sorted in some order (alphabetic probably).

pace complexity:

- M: terms in dictionary
- w: average term length
- k: number of possible bigrams
- 1 byte per character for storage
- 4 bytes for pointer
- There will be k entries in the bigram index
- Total space cost: $2k + 4k + \left(\frac{\log M}{8} * M * (w+1)\right)$

Rotated terms:

If wildcard queries are common, we can save on time at the cost of more space, using a rotated lexicon.

- Rotation is a cyclic shift of the characters.
- i-rotation is shifting the original term i times. 1-rotation of "big" is "big".
- rot(t): Given a term t of length w, we use rot(t) to denote the set derived by padding t with a startig symbol ($) and taking all i-rotations of t where $1 \leq i \leq w+1$.
  In rot(t) there will be w+1 words.
- A rotated lexicon contains, for each term t and for each rotation t' in rot(t), an entry of the form (k, i) where:
  - k: the term number of t in the (regular) lexicon.
  - i: t' is the i-rotation of t.
- (k, i) uniquely identifies a rotated term. We do not actually store the rotated string in the lexicon, the pair of numbers is enough for binary search.
- These entries are stored in alphabetical order of the rotated terms

Space complexity:

- M: terms in dictionary
- w: average term length
- x: longest word
- There will be $M * (w+1)$ entries in the rotated lexicon.
- Search time: logarithmic in $M * (w+1)$
- Space: $M * (w+1) * \left(\frac{\lceil \log M \rceil}{8} + \frac{\lceil \log(x+1) \rceil}{8}\right)$

2.4 Spelling correction:

- The subject is split into 2 categories: detection and correction
- Error: any word not in the dictionary.

2.4.1 Measuring word similarity:

Edit distance:

Given 2 strings, s and t, the edit distance of s and t is the minimum number of basic operations to convert one to the other.

Maximal edit distance: $\max\left(len\left(s\right), len\left(t\right)\right)$. Can be found in polynomial time with dynamic programming.

Basic opertaions:

- insert a character
- delete a character
- replace a character with a different character.
- transpose adjacent characters.

<u>Computing edit distance through dynamic programming:</u>

Intuition:

- Create a matrix d with $|s|$ columns and $|t|$ rows
- The entry $d\left[i, j\right]$ is the edit distance between the words s_i and t_j (i.e. between the prefix of s of size i and the prefix of t of size j).

Algorithm:

```
• n = len(s)
• m = len(t)
• if n = 0 : return m and exit
• if m = 0 : return n and exit
• d = matrix(n, m)
• for i = 1 to n:
    – $d\left[i, 0\right] = i$
• for j = 1 to m:
    – $d\left[0, j\right] = j$
• for i = 1 to n:
    – for j = 1 to m:
        ∗ if $s\left[i\right] = t\left[j\right]$: cost = 0
        ∗ else: cost = 1
        ∗ $d\left[i, j\right] = \min\left\{d\left[i-1, j\right] + 1, d\left[i, j-1\right] + 1, d\left[i-1, j-1\right] + cost\right\}$
• return $d\left[n, m\right]$
```

Weighted edit distance:

- Give different wights to different operations.
- Can consider likely keyboard errors.
- Can consider common spelling mistakes.
- Require symmetric weight matrix as input. Symmetric means replacing 'k' with 'e' has the same weight as replacing 'e' with 'k'.

Algorithm (after initializing the matrix):

```
for i = 1 to n:
    for j = 1 to m:
        if $s[i] = t[j]$: cost = 0
        else: cost = $weight[i,j]$
        $d[i,j] = \min\{d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+cost\}$
return $d[n,m]$
```

2.4.2 Using edit distance to find similar words:

Use n-gram overlap to find words that are likely to have low edit distance.

Given the misspelled word t:

- find all n-grams of t (for a given n)
- using an n-gram index, find all words in the lexicon that have n-grams in common with t.
- Consider as candidates words from the lexicon that are **sufficiently similar** in their n-grams to t.
  - word similarity can be measured using Jaccard coefficient
- For these candidates, compute the actual edit distance.

Jaccard coefficient: Let X and Y be two sets (e.g., the set of n-grams of two words), the Jaccard Coefficient of X and Y is: $\frac{X \cap Y}{X \cup Y}$

- Always between 0 and 1
- Equals 1 when X = Y and 0 when $X \cap Y = \emptyset$

2.4.3 Noisy channel model:

Given a misspelled word, x, find the most likely to be the correct word $\hat{W}$.

V: Set of possible words that the user meant.

$\hat{W} = argmax_{w \in V} P(\text{real w} \mid \text{wrote x}) =_{Bayes} \frac{P(\text{wrote x} \mid \text{real w}) * P(\text{real w})}{P(\text{wrote x})}$

We assume that $P(\text{wrote x})$ is the same for any w, so we can ignore the denominator:

$P(\text{wrote x} \mid \text{real w}) * P(\text{real w})$

How to choose V (candidate generation): Idealy we want the best spell corrections. But it's hard, so instead we find a subset of pretty good corrections and find the best amongst them.

We can look for words with similar spelling:

- Small edit distance to error. Consider words with low Damerau-Levenshtein edit distance (insert, delete, replace and transposition).
- 80% of errors are within edit distance 1.
- Almost all errors within edit distance 2.

We can look for words with similar pronunciation:

- Small distance of pronunciation to error.

How to compute P(wrote x | real w) (Edit model probability):

Intuitively: How likely is it that the user typed x, when he meant to type w?

Misspelled word $x = x_1, \ldots, x_m$

Correct word $w = w_1, \ldots, w_n$

We can estimate the probabilities using a large set of misspellings:

Confusion matrix:

- $del\,[w_{i-1}, w_i]$: how common it is to write w_{i-1} instead of $w_{i-1}w_i$
- $ins\,[w_{i-1}, w_i]$: how common it is to write $w_{i-1}w_i$ instead of w_{i-1}
- $sub\,[x_i, w_i]$: how common it is to write x_i instead of w_i
- $trans\,[w_i, w_{i+1}]$: how common it is to write $w_{i+1}w_i$ instead of w_iw_{i+1}

channel model (single edit error):

$count\,[w_i]$: how common it is for a word to contain the letter w_i .

$count\,[w_{i-1}w_i]$: how common it is for a word to contain the sequence of letters w_iw_{i+1} .

$$P\left(\text{wrote x}|\text{ real w}\right) = \begin{cases} \frac{del[w_{i-1},w_i]}{count[w_{i-1}w_i]} & \text{if error is deletion} \\ \frac{ins[w_{i-1},x_i]}{count[w_{i-1}]} & \text{if error is insertion} \\ \frac{sub[x_i,w_i]}{count[w_i]} & \text{if error is substitution} \\ \frac{trans[w_i,w_{i+1}]}{count[w_iw_{i+1}]} & \text{if error is transposition} \end{cases}$$

Smoothing probabilities:

For errors we have not seen, the confusion matrix will give 0-probability, that seems too harsh.

A simple solution is add-1 smoothing: add 1 to all counts and then if there are n characters in the alphabet, normalize appropriately

e.g. for substitution: $\frac{sub[x_i,w_i]}{count[w_i]} \Longrightarrow \frac{sub[x_i,w_i]+1}{count[w_i]+n}$

How to compute P(real w) (language model) :

Basic Idea: A language model associates a probability with a sequence of words.

- We will only focus on two types of language models: unigram and bigram.
- t_i are words

Unigram Language Model: The likelihood of a word is independent of previous words in the sequence, i.e. ,

- $p\left(\text{real } t_{n+1}|\text{ real prev } t_1, \ldots, t_n\right) = p\left(\text{real } t_{n+1}\right)$
- $p\left(\text{real } t_1, \ldots t_n\right) = p\left(\text{real } t_1\right) \times p\left(\text{real } t_2\right) \times \ldots \times p\left(\text{real } t_n\right)$

A simple way to define a unigram language model is to take a big supply of words (your document collection or query log collection with T tokens):

V: vector of possible corrections and their probabilities.

$C\left(w\right)$: number of occurrences of w in the collection

T: number of tokens

$p\left(\text{real w}\right) = \frac{C(w)}{T}$

return $\arg\max_{w \in V} p\left(\text{wrote x } | \text{ real w}\right) * p\left(\text{real w}\right)$

Bigram Language Model: The likelihood of a word depends only on the previous word in the sequence, i.e.,

- $p\left(\text{real } t_{n+1} \mid \text{real prev } t_1, \ldots, t_n\right) = p\left(\text{real } t_{n+1} \mid \text{real prev } t_n\right)$
- $p\left(\text{real } t_1, \ldots t_n\right) = p\left(\text{real } t_1 \mid \text{real prev none}\right) \times p\left(\text{real } t_2 \mid \text{real prev } t_1\right) \times \ldots \times p\left(\text{real } t_n \mid \text{real prev } t_{n-1}\right)$

Defining a bigram model (naive):

- Start again with a large collection.
- Let w1, w2 be words
- Let $C(w_1, w_2)$ be the number of times that the sequence w_1,w_2 appears.
- Let C(w) be the number of times that w appears.
- $p\left(w_2 \middle| w_1\right) = \frac{C(w_1,w_2)}{C(w_1)}$

Defining a bigram model (smoothed):

- For unigram counts, P(real w) is always non-zero if our dictionary is derived from the document collection. But this won't be true of $p\left(\text{real } w_k \mid \text{real prev } w_{k-1}\right)$. We need to smooth.
  - option 1: add-1 smoothing
  - option 2: interpolation, $p\left(\text{real } w_k \mid \text{real prev } w_{k-1}\right) = \lambda p_{uni}\left(\text{real } w_k\right) + (1-\lambda)\, p_{bi}\left(\text{real } w_k \mid \text{real prev } w_{k-1}\right)$

3 Index construction:

- Complexity is IO complexity, i.e number of read/write operations.
- Disk reads/writes are often the most expensive part in an algorithm.
- Page: A unit that fits in a single IO operation.
- Spetial locality: When using a word from a block, we can use all words in the same block.
- Temporal locality: When using a block, reuse it soon afterwards (before evicting from cache)
- External memory algorithm: algorithm designed to process data that is to large to fit into the computer's main memory at one time.

3.1 Sort-Based construction:

Step 1: Read documents and create dictionary in main memory.

- Associate each term with a termID.

Step 2: Write to disk all pairs (termID, docID) by appending to end of file.
Improved: Read documents. While there is enough memory, store pairs of (termId, docId) in memory. When memory runs out, sort pairs and write to disk. Continue reading files (sorting pairs here while writing them saves I/O complexity)

Step 3: Sort the file we created according to termID with a secondery sort by docID.
Improved: only merge sorted files.

Step 4: Read sorted file consecutively and write posting lists while updating freq and posting pointer in dictionary.

- Each time a posting list is created, update pointer in dictionary.
- perform compression of posting lists now.

Algorithm:

F: file to sort.

B: blocks in file.

M: blocks of internal memory available.

n: number of values in each block.

N: number of values in total.

3.1.1 Special case: $\lceil \frac{B}{M} \rceil < M$:

Algorithm (step1): After step 1 we will have $\lceil \frac{B}{M} \rceil$ sorted sequences.

```
• left = 0
• while (left < B):
    – right = min(left+M, B) - 1
    – read blocks $b_{left}, \ldots, b_{right}$ to main memory
    – sort $b_{left}, \ldots, b_{right}$ in place
    – write sorted sequence to disk
    – left = left + M
```

Step 2:

```
• for i = 1 to $\lceil \frac{B}{M} \rceil$:
    – read first block of the i-th sorted sequence into memory
    – set $p_i$ to point to the first value in this block
• k = 0
• while k < N:
    – let $p_i$ be the pointer to the minimal value among all pointers
    – copy the value of $p_i$ to the output block
    – increment k
    – if k mod n = 0:
        * flush the output block to disk
    – if $p_i$ points to the last value of the block
        * read the next block of the sequence and set $p_i$ to its first value
    – else: increment $p_i$
```

3.1.2 If $\lceil \frac{B}{M} \rceil \geq M$:

We can no longer merge in one phase

instead, merge M-1 sorted sequences gain and again until completely sorted.

- IO cost: $2B * \left(\log_{M-1} \lceil \frac{B}{M} \rceil\right)$ (number of passes is $\left(\log_{M-1} \lceil \frac{B}{M} \rceil\right)$)

3.2 Merge based construction:

- Reading the raw input file is expensive, we should not read it more than once.
- Generate seperate dictionaries for each chunk of data - no need to maintain term-termID mapping accross blocks and merge them later.
- Don't sort, Accumulate postings in postings list as they occur

Algorithm:

```
• repeat until all documents have been completely parsed.
    – while there is available memory, read next token
    – look up this token in an in-memory hash table.
```

  - If it appears:
    * get the in-memory posting list from the hash table and add the new entry to the posting list.
  - else:
    * add it to the hash table.
    * create new posting list.
    * add the new entry to the posting list.
  - When memory runs out:
    * write dictionary, sorted, to disk.
    * write posting lists, sorted, to disk.

- Merge dictionaries
- Merge posting lists (while compressing)

N: number of sub-indexes

S: size of each sub-index

M: number of available pages in memory

- IO cost: $2SN\left(1+\lceil\log_{M-1}N\rceil\right)$

3.3 Dynamic indexing:

- Documents come in over time and need to be inserted
- Documents are deleted and modified
- Postings update for terms already in dictionary.
- New terms added to dictionary.

Simplest approach:

Hold 2 index structures: one main index and whenever a new document arrives we save it in a smaller auxilery index.

- Maintain "big" main index.
- New docs go into "small" auxiliary indexes.
- When searching, we search both index structures and merge the results.
- Maintain a bit vecotr that flags which documents still exists and which are deleted. When searching, skip the indexes of deleted documets.
- Periodically, re-index into one main index.

Issues with this approach:

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge

<u>The merge problem:</u>

Suppose you have a main index I for your data and when changes occur you create an auxilliary index I_0.

n: maximum size of I_0 before merging with I.

T: data size

Will result in $\frac{T}{N}-1$ merges

Total time complexity: $\sum_{i=1}^{\frac{T}{n}}\left(n+(i-1)\,n\right)=O\left(\frac{T^2}{n}\right)$

Logarithmic merge (Extra merging algorithm):

n: maximum in-memory index size

T: data size

- Maintain a series of indexes, each twice as large as the previous one.
- Keep the smallest index, Z_0 in memory and larger ones, $I_0, I_1 \ldots$ on disk.
- If Z_0 gets bigger than n
  - if I_0 doesn't exist: write to disk as I_0.
  - else if I_1 doesn't exist: merge Z_0 with I_0 as I_1
    - else: ...

Pseudo code:

LMergeAddToken(indexes, Z_0, token)

```
• $Z_0 \leftarrow \text{Merge}(Z_0, \{\text{token}\})$
• if $|Z_0| = n$
    – for $i = 0$ to $\infty$:
        ∗ if $I_i \in$ indexes
            · $Z_{i+1} \leftarrow \text{Merge}(I_i, Z_i)$ // ($Z_{i+1}$ is a temporary index on disk)
            · indexes $\leftarrow$ indexes $- \{I_i\}$
        ∗ else:
            · $I_i \leftarrow Z_i$ // ($Z_i$ becomes the permenant index $I_i$)
            · indexes $\leftarrow$ indexes $\cup \{I_i\}$
            · Break
        ∗ $Z_0 \leftarrow \emptyset$
```

LogarithmicMerge():

```
• $Z_0 \leftarrow \emptyset$ // ($Z_0$ is the in memory index)
• indexes $\leftarrow \emptyset$
• while true:
  LMergeAddToken(indexes, $Z_0$, nextToken).
```

Time complexity: Each posting is merged O(log T) times

so complexity is $O(T \log T)$

But query processing now requires the merging of $O(\log T)$ indexes.

4 Text based ranking:

4.1 Naive method:

Refer to a document as a set of terms.

Refer to a query as a set of terms.

Jaccard coefficient: A calculation of similarity between 2 sets.

- Given 2 sets A and B, their Jaccard coefficient is: $\frac{|A \cap B|}{|A \cup B|}$

4.2 Vector space model:

4.2.1 Vector space model:

Each term in the corpus is refered to as a dimention in a vector space. Documents and queries are vectors in this space.

- Value of vector in dimention indicates it's importance.
- We attempt to value rare words over common ones (rare in the entire corpus). Rarity should be relative to corpus size and should have good balance over different document sizes.

tf (Term frequency): value given to a term's frequency throughout the entire corpus.

$tf_{t,d}$: Number of times term t appears in document d.

- Natural (n) term frequency is taking the actual value of $tf_{t,d}$.
- Boolean (b) term frequency:

$$\begin{aligned} tf &= 1 \text{ if } tf_{t,d} > 0 \\ &= 0 \text{ otherwise} \end{aligned}$$

- log-based (l) term frequency:

$$\begin{aligned} tf &= 1 + \log\left(tf_{t,d}\right) \text{ if } tf_{t,d} > 0 \\ &= 0 \text{ otherwise} \end{aligned}$$

- Augmented (a) term frequency: (denominator is the frequency of the term that is most frequent in the document, d)

$$\begin{aligned} tf = \alpha + \frac{(1-\alpha)}{max_s\left\{tf_{s,d}\right\}} &\text{ if } tf_{t,d} > 0 \\ &= 0 \text{ otherwise} \end{aligned}$$

4.2.2 Document frequency:

idf (Inverse document frequency): The importance of a term in a document. monotonically non-increasing as the number of documents containing the term increases.

df_t : Number of documents containing the term t.

N : Number of documents in the corpus.

- no document frequency (n): $idf_t = 1$
- Standard idf (t):

$$\begin{aligned} idf_t = \log\left(\frac{N}{df_t}\right) &\text{if } df_t > 0 \\ &= 0 \text{ otherwise} \end{aligned}$$

- Smoothed idf (s):

$$idf_t = \log\left(\frac{N}{1 + df_t} + 1\right)$$

For each document d, we define vector v_d where there is a coordinte for each term in the corpus. We calculate the value of term t in v_d by: $tf * idf_t$.

For a given query, q, We define a vector v_q where there is a coordinte for each term in the corpus. We calculate the value of term t in v_q by: $tf * idf_t$ where tf_t is the boolean tf (gives 1 for terms that exist in q).

A document's compatibility to a query is determind by the dot product $v_d \cdot v_q$

4.2.3 Normalization:

We don't want to allways give better ranks to larger documents.

given a vector $v = (v_1, \ldots v_n)$

No normalization (n): v

Cosine normalization (c): $v * \frac{1}{\sqrt{\sum (v_i)^2}}$

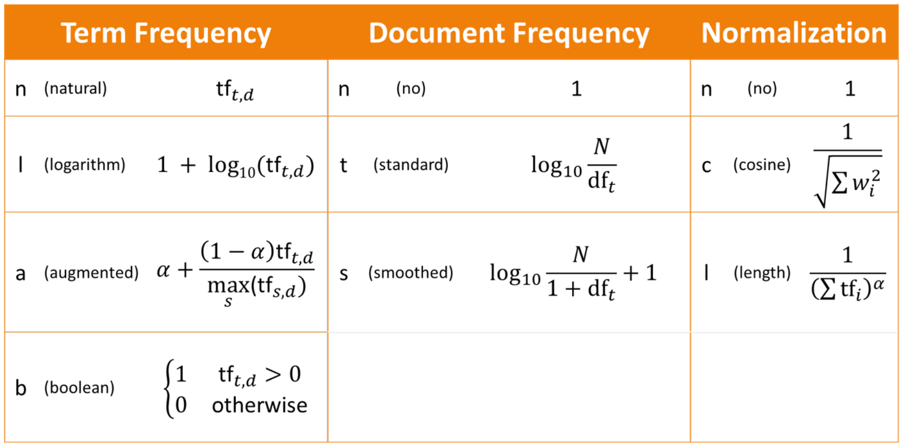
Length normalization (l): $v * \frac{1}{(\sum tf_i)^\alpha}$ for some $\alpha < 1$

- Reduces the weigh for longer documents.

Smart notation: assign 3 letters for the document and 3 letters for the query. Looks like this: ddd.qqq

- first letter: tf
- second letter: idf
- third letter: normalization

Together:



Term Frequency	Document Frequency	Normalization
n (natural) $\text{tf}_{t,d}$	n (no) 1	n (no) 1
l (logarithm) $1 + \log_{10}(\text{tf}_{t,d})$	t (standard) $\log_{10}\frac{N}{\text{df}_t}$	c (cosine) $\frac{1}{\sqrt{\sum w_i^2}}$
a (augmented) $\alpha + \frac{(1-\alpha)\text{tf}_{t,d}}{\max_s(\text{tf}_{s,d})}$	s (smoothed) $\log_{10}\frac{N}{1+\text{df}_t} + 1$	l (length) $\frac{1}{(\sum \text{tf}_i)^\alpha}$
b (boolean) $\begin{cases} 1 & \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$		

4.2.4 Ranking web documents:

Web documents are typically written in HTML, and have markup indicating different types of document elements, e.g., title, headings, bold, links.

Ways to rank such files (examples):

- consider only words in prominent places.
- Give bigger weights according to position of term in file

4.3 Language model:

4.3.1 Unigram model:

A unigram model provids a probability for each term such that $\sum_t P(t) = 1$ as well as probability for stopping.

- We also define a probability to stop the term. But we can ignore it because it doesn't make a difference.
- Each document, d is treated as the basis of a language model.

- Given a query, q rank documents based on bayes formula: $P\left(d|q\right) = \frac{P(q|d) \times P(d)}{P(q)}$. But $P\left(q\right)$ and $P\left(d\right)$ are the same for all documents so they don't matter.
  We are left with: $P\left(d|q\right) = P\left(q|d\right)$. Intutition: A user who wants to find document, d, will write the query q.
- M_d : language model of d.
- Maximum likelihood: $P\left(t|M_d\right) = \frac{tf_{t,d}}{|d|}$
- $P\left(q|M_d\right) = \prod_{t \in q} P\left(t|M_d\right)$
- M_c : language model of the corpus.
- cf_t : number of occurances of t in the corpus.
- T : Total number of terms in the corpus.
- $P\left(t|M_c\right) = \frac{cf_t}{T}$

4.3.2 Mixture model:

$P\left(t|d\right) = \lambda P\left(t|M_d\right) + \left(1-\lambda\right) P\left(t|M_c\right)$

$P\left(q|d\right) = \prod_{t \in q} \left(\lambda * P\left(t|M_d\right) + \left(1-\lambda\right) * P\left(t|M_c\right)\right)$

5 Ranking links:

query independent score: Can be pre-computed.

5.1 Markov chain with teleportation:

N: number of states

α: parameter that determine the probability of teleportation

Markov can be described with a NxN matrix.

1. Create adjacency matrix, which is a matrix that has 1 on [i,j] if state i has a link to state j and 0 otherwise.
2. Find all rows with all 0's and assign $\frac{1}{N}$ to it's values instead.
3. For each row, divide all 1's by the number of 1's in that row.
4. Multiply the resulting matrix by $1 - \alpha$.
5. Add $\frac{\alpha}{N}$ to each cell in the matrix.

Ergodic markov chain: A markov chain is ergodic if there exists a positive integer T_0 such that for all pairs of states, i,j in the chain: If it is started at time 0 in state i, than for all $t > T_0$ the probability of being at state j at time t is greater than 0.

- The markov chain for teleporting surfer is ergodic

Page rank formula:

$P_1, \ldots, P_n$: pages that link to P.

O_i: number of outgoing links in page P_i

If P_i is a sink $O_i = N$

$PR\left(P\right) = \frac{\alpha}{N} + \left(1-\alpha\right)\left(\frac{PR(P_1)}{O_1} + \ldots, + \frac{PR(P_n)}{O_n}\right)$

5.2 HUBs and Authorities:

Hub: A good hub page points to many good authority pages.

Authority: A good authority page is pointed to by many good hub pages.

$h(v)$: hub value of page v.

$a(v)$: authority value of page v.

Root set: Given a query q find all pages containing the words in q. This is the root set.

k: Constant - number of iterations.

Base set contains:

1. all pages in the root set.
2. all pages pointing to a page in the root set.
3. all pages pointed to by a page in the root set.

Computing $h(v), a(v)$:

- Initialize $h(v) = a(v) = 1$ for each page v.
- for i = 1 to k:
  - In each iteration:

$$h(v) = \sum_{y \in out(v)} a(y)$$
$$a(v) = \sum_{y \in in(v)} h(v)$$

  - Normalize $w \in$ BaseSet:

$$h(v) = \frac{h(v)}{\sum_w h(w)}$$
$$a(v) = \frac{a(v)}{\sum_w a(v)}$$

  - If converged: break

Notes:

- In reality ~ 5 iterations get close to stability.
- This method is query independent.
- Iterative computation after text index retrieval - significant overhead.

Issues (dealing with spams and alike):

- Topic drift: Because we add pages, we might return an authority that does not refer to the initial query. (but some related topic or “super topic”)
- Mutually Reinforcing Affiliates: Two or more pages that have many links between one another can boost each other’s scores.

5.3 WebGraph:

Successor: Outgoung link.

$S(x)$: list of successors of node x.

WebGraph: Graph of the page links in the web.

Connectivity server: server that answers queries about the structure of the web graph.

- Which URLs point to a given URL?
- Which URLs does a given URL point to?
- Crawl control
- Web graph analysis
- link analysis

5.3.1 How to compress the webgraph:

- Locality: most links lead the user to a page on the same host. So, if URLs are stored lexicographically, the index and source and target are close.
- Similarity: URLs that are lexicographically close have many common successors.
- Consecutivity: Often many successors of a page have consecutive URLs.

Gap compression:

For each node x, save the successor list in gap compression.

a_0 : the first entry

$$a_0 = \begin{cases} 2*(s_1 - x) & s_1 - x \geq 0 \\ 2*|s_1 - x| - 1 & \text{else} \end{cases}$$

- Successor list of x: $S(x) = [a_0, s_2 - s_1 - 1, \ldots, s_k - s_k - 1]$
  - Notice that, except for the first element, all gaps are reduced by 1.

Reference compression:

For a node, x, code $S(x)$ as a modified version of $S(y)$ for some node y that appear before x.

reference number: $x - y$.

copy list: A bit sequence the length of $S(y)$ which indicates which of the values in $S(y)$ are also in $S(x)$. This is represented using bitmap (a word of 1-s and 0-s such that the i'th digit is 1 if the i'th link in $S(y)$ is also in $S(x)$ and 0 otherwise)

list of extra nodes: $S(x)/S(y)$ (all elements in S(x) that are not in S(y))

sliding window: Usually, we only consider references that are within a sliding window.

Differential compression:

Do not store a list that is the length of $S(y)$.

Look at the copy list as an alteranting sequence of 1 and 0 blocks.

store the number of blocks.

store the length of the first block.

store the lengths -1 of all other blocks besides the last block. (We know that the lengths are at least 1 for all blocks except for the first one)

Always assume that the first block is a string of 1's (could be of length 0).

Compressing extra nodes list:

L_{min} some system parameter

We find subsequences containing consecutive numbers of length $\geq L_{min}$

For each interval, we store:

- Left extremes are compressed using differences between extremes - 2. (We know that the last block of the previous sequence is trimmed so there is a difference of 2 between consecutive sequences) (I think)
- Interval lengths are decremented by L_{min}.

Store list of residuals compressed using gaps and variable encoding.

6 Query processing:

A conjunctive query: boolean query of the form: $t_1 AND\, t_2 AND\, \dots AND\, t_3$

To answer a conjunctive query we must search for all terms in the dictionary and find the intersection of the posting lists.

6.1 TAAT (Term At A Time):

For each term in the query, read each posting list entirely and intersect with what we got until current list.

Considering term frequency:

- Start with lowest frequency term since the result size is bound by this length. As a consequence, intermediate results will not take too much space. This excludes many items of the following lists.
- Continue processing in frequency order to save time on lookup
- Most costly action is the intersection.

Algorithm:

- For each query term t:
  - Preprocess t (stem, normalize case, etc)
  - Search the lexicon for t
  - Record f_t (frequency) and address l_t
- Identify term t with smallest f_t
- Define C := posting list at l_t
- For each remaining term t' in increasing order of $f_{t'}$: (iterate query terms according to order of their frequency)
  - Define C' := posting list at $l_{t'}$
  - For each d in C:
    - if d not in C': remove d from C
- For each d in C:
  - Look up the address of d and return document to the user

6.2 DAAT (Document At A Time):

Iterate over all posting lists at the same time, and deal with the first shared document before moving to the next document.

Algorithm:

- For each query term t:
  - Preprocess t (stem, normalize case, etc)
  - Search the lexicon for t
  - Record f_t (frequency) and address l_t
- Deifine C := ∅
- While not end of any posting list:
  - Read all posting lists simultaneously, increasing the pointer of the list with lowest docID at each step.
  - If all pointers point to the same docID d, add d to C.
- For each d in C:
  - Look up the address of d and return document to the user.

6.3 Intersection methods:

6.3.1 Intersection using lookup:

- For each d in C:
  - if d not in C': remove d from C.
- $|C| \leq |C'|$

Time complexity:

- Decode entire list: $|C'|$
- Lookup each docID: $|C| * \log(|C'|)$

6.3.2 Intersection using merge:

We are given sorted lists $C_1, \ldots C_n$ and assume:

- C_i has length l_i
- $l_1 + \ldots l_n = N$

Time complexity: Intersection can be computed in time $O(N \log(n))$

No need to pre-read entire lists to memory.

Skip pointers:

If one list's next value is much larger than those of the other lists, other lists will use skip pointers to find it fast.

- Skip pointers are implemented on disk with synchronyzation points.
- docID: current ID in the list.
- bitAddress: points to the next synchronization point.
- Synchronyzation point: a pair (docID, bitAddress).

- Synchronization points can be gap encoded. doc lists are gap encoded and synchronyzation points are also gap encoded but seperately. We assume we only get to a synchronization point with a skip pointer so the encodings don't clash.
- Choosing skip pointers:
  - x: size of skip
  - k: length of list
  - We choose: $x = \sqrt{k}$

6.4 Query ranking

Given a query Q, we want to find the top k documents satisfying Q using some ranking function r.

- Must have low latency (e.g 250ms).

6.4.1 Safe ranking:

guarantees that the k documents returned are indeed the highest scoring.

Wand scoring:

Algorithm that computes top-K while pruning extensively.

Assume several assumptions on the index and on the ranking functions:

1. Posting lists are sorted by document ID.
2. There exists an iterator iter(t,X) on the posting lists. "go to the first document $id \geq X$ of term t"
3. Each query term t contributes a value r(t,d) to the score of the document d.
4. Total score: $score(d) = \sum_t r(t,d)$
5. There are no other factors in the document score.
6. There is a threshold θ such that every document in top -k has score at least θ. Start with $\theta = 0$ and increase as we compute scores.
7. For each term t there is an upper bound value UB_t such that always $r(t,d) \leq UB_t$.

Algorithm:

```
• for i = 1 to n:
    – define $p_i$ = pointer to first item in list for $t_i$
• define result = {}
• $\theta = 0$
• while not all pointers at end of lists:
    – $p'_1, \ldots, p'_n$ = sort of $p_1, \ldots p_n$ by increasing docId
    – pivot = $\min_j \left\{ \sum_{i \leq j} UB_{t'_i} > \theta | 1 \leq j \leq n \right\}$
    – docId = pivot → value (value pointed at by $p_{pivot}$)
    – for i = 1 to pivot -1:
        * $p'_i$ = iter($t'_i$, docId)
    – if $\sum_{p'_i \rightarrow value = docId} UB_{t'_i} > \theta$ :
        * if score(docId) > $\theta$:
            · update result = $\{p'_i \rightarrow value\}$
```

    · update $\theta = \sum_{p'_i \to value = docId} UB_{t'_i}$

  – increment all pointers pointing to docId

Sidenote:

Sometimes it is more convenient to order each posting list according to the score that the word gives to each document:

- Higher quality matches come first.
- Cannot use gap encoding.
- Cannot use WAND (or even the linear merge).

No Random Access Threshold algorithm:

Slide description: Access all lists sequentially in parallel until there are k objects for which the lower bound is higher than the upper bound of all other objects.

Assumptions:

1. For each term, posting list is sorted according to rank. (from best to worst).
2. We can only read posting lists in order (no random access).

Algorithm:

1. Look at first row in all postings. the postings are sorted from best to worst so we know the lower bound for each document we see is at least the score we see.
2. Set LB (lower bound) for each document on first row to be the sum of scores of that document we see on first row.
3. Set UB (upper bound) for each document we see to be the sum of all scores on first row.
4. Go to next row, update LB of each file according to it's first appearances in some postings so far. Update UB of each file according to LB and lack of appearances in the postings so far.

6.4.2 Unsafe (vector space) ranking:

May return a document not in top k, inheretly, may miss some of the true top k documents.

Find a set P of contenders, with $k < |p| << N$

Return top k results in p.

Heuristics for choosing p:

Choosing p according to high idf query terms only: Remove common query words (stop words and sometimes other words).

Choosing p according to docs containing many query terms: include in p only documents with several terms from the query.

Choosing p according to champions list: pre-compute the champions list for each term and include in p only documents in the champions list for at least one query term.

- Can choose champions list according to TF, or by PageRank or anything else.

Choosing p according to tiered list: break postings into hierarchy of tiers and use only top tier documents. And, if needed, pick documents from lower tiers according to user demands.

Combining query and document scores: Order docIds according to reversed pageRank (best comes first).

7 Map Reduce:

Hadoop: An open source version of map-reduce

Nodes: computation units (CPUs or computers)

7.1 The problem:

Proccess very large amounts of data will take too long with 1 CPU. So we need to somehow synchronize multiple CPUs.

- Node failure:
  - How not to lose data when nodes can fail?
  - How to deal with node failure during a long-running computation?
- Network bandwidth is limited (~1Gbps)
- Distributed programming is hard.
  - How do we assign work units to workers?
  - What if we have more work units than workers?
  - What if workers need to share partial results?
  - How do we aggregate partial results?
  - How do we know all the workers have finished?
  - What if workers die?

7.2 Distributed file system:

HDFS: Hadoop Distributed File System

- designed to store very large files with streaming access patterns.
- running on clusters of commodity hardware.
- Streaming access patterns = Write once, read (entirely) many times

Block size:

Disk block size - usually around 512 bytes.

HDFS block size usually around 64 MB.

- Large, so that disk seek time is dominated by transfer rate.
- Not larger, so as to still have enough Map tasks.
- Blocks are replicated 2-3 times, typically.

Data kept in blocks spread across machines.

Each block is replicated on different machines to recover from disk or machine failure.

- Block servers also serve as compute servers.

Name nodes:

Name node: (master node) manages the file system namespace.

- Persistently stores file system tree + metadata.
- Keeps lists of data nodes which have copies of each block.
- Failure of this node makes the system unusable! Hadoop provides failure mechanisms.

Data nodes:

Can also be called block server or worker node.

- Store and retrieve blocks.
- periodically report to name-node with list of blocks stored.

Client library (Client Application):

- Contacts the Master to find data-nodes.
- Contacts data-nodes directly to access data.

7.3 Underlying framework:

Data flow:

Input and final output are stored on HDFS:

Scheduler tries to schedule map tasks "close" to physical storage location of input data.

Output is often input to another MapReduce task.

Intermediate results are stored on local FS of Map and Reduce workers

Coordination:

Master:

- Task status: (idle, in-progress, completed).
- When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer and Master pushes this information to reducers.
- Master pings workers periodically to detect failures.

Mapper:

- will typically be a data-node containing the block.

Reducer:

- Reducer for a key will typically be a data-node chosen by hashing the key value to a number between 1 and R.

Slow workers: Nodes that happen to be slower for any reason.

- These need to be identified because they shouldn't be given long tasks. To do so, near end of phase, spawn backup copies of tasks (workers work on the same task redundantly) .Whichever one finishes a task first "wins" and the other copies of the same task are discarded.
- This dramatically shortens job completion time.

How many map and reduce jobs:

N: number of data nodes.

B: data blocks.

M: number of map tasks.

R: number of reduce tasks.

- $M >> N$ Usually $M \approx B$
  - Improves dynamic load balancing and speeds up recovery from failures.
- $M >> R$
  - Output is spread over R files.

7.4 Algorithms:

The example for the entire section: Given a HUGE set of text files (or one HUGE file), count how many times each word appears in the corpus.

General overview:

This outline stays the same, Map and Reduce change to fit the problem.

- Sequentially read a lot of data.
- $Map\,(k, v) \rightarrow (k', v')$: Extract something you care about.
  - Takes a key-value pair and outputs a set of key-value pairs (k',v').
  - There is one Map call for every (k, v) pair.
- Group by key: Sort and Shuffle.
- $Reduce\,(k', v') \rightarrow (k'', v'')$: Aggregate, summarize, filter or transform.
  - All values v' with same key k' are reduced together and processed in v' order.
  - There is one reduce function call per unique key k'.
  - Reduce jobs don't start before the mappers have finished the entire corpus.
  - Each reducer receives the keys sorted.
- Write the result.

Combiner functions (optional in the algorithm):

Mini-reducers that run in memory after the map phase.

- Runs on the same node as the mapper immediately after the mapping.
- Must have precisely the same input as the reducer and the same output as the mapper
  $Combine\,(k', v') \rightarrow (k', v')$
- Used as an optimization to reduce network traffic.
- It's faster to combine while mapping. But, explicit memory management is required and has higher bug potential.

Partition functions (optional in the algorithm):

Function that receive as input a key and the number of reducers and returns a partition id for k'. (chooses which reducer works on which keys)

- $partition\,(k', \text{num\_of\_partitions}) \rightarrow$ partition number of k'.
- Overrides the framework's implementation to the same functionality.

Sort functions (optional in the algorithm):

Sometimes we would like to sort the (k', v') pairs before the reduce.

Local aggregation:

Aggregate the data at the mapper or at the reducer to improve network communication overhead.

- One of the main costs of a program is in its network communication.
- Often, intermediate results are written to disk before communication, making the cost even higher.

7.5 Text retreival:

Build an index using map-reduce:

Pseudo code:

```
map(docId, text):
    • H = new hashTable
    • for each word w in text:
        – H{w} = H{w} + 1
    • for each word w in H:
        – emit((w, docId), H{w})
partition(key):
// key: (word,docId) tuple
    • return hash(key.word)
sort(key1, key2):
// key: (word,docId) tuple
    • if (key1.word != key2.word):
        – return key1.word < key2.word
    • return key1.docId < key2.docId
Reducer: initialize()
    • l_word = null
    • l_P = new List
reduce(key, values):
// key: (word,docId), value: frequency
    • if word != l_word and l_word != null:
        – emit(l_word, P)
        – l_word = word
        – l_P = new List
    • l_P.append(docId, value[1])
Reducer: close()
    • emit(l_word, P)
```

Graphs and map-reduce:

Graph algorithms typically involve:

- Performing computations at each node: based on node features, edge features, and local link structure.
- Propagating computations: “traversing” the graph.

Generic recipe to use map-reduce:

- Perform local computations in mapper.
- Pass along partial results via outlinks, to destination node.
- Perform aggregation in reducer on inlinks to a node.
- Iterate until convergence: controlled by external “driver”.
- Don’t forget to pass the graph structure between iterations.

8 Crawling:

Basic crawler operations:

- begin with seed URLs
- fetch seed, parse, insert URLs into queue
- While queue is not empty, repeat

Complications:

- Not feasable with one machine because the bandwidth is limited. Must be distributed.
- Some pages try to catch crawlers and disrupt the protocol (spider traps).
- Must consider: latency, bandwidth of servers, politeness and crawling directive.
- Legality is debatable and case dependent.

Crawler attributes (a crawler must be):

- Polite
- Robust
- Distributed
- Scalable
- Efficient
- Prioritizable
- Continuous
- Extendable

DNS (Domain Name Server):

- Contains lookup tables to search the web.
- Latency can be high and we need to be efficient:
  - Pre fetching and caching: As soon as we know we will look for a page in the future, we send a DNS request and save the answer in our cache for the expected use later.
  - We can batch several URL queries in a single DNS request to save on time.

8.1 Robot.exe:

Website uses this file to announce which files can / should not be crawled.

- robots.txt file is placed at the root directory of the server (must be called robots.txt so crawlers can easily find it)
- does not prevent robots from crawling entire site.
- can include a sitemap to help the crawler find the site pages.
- syntax:
  - user-agent: [robot_name] (can use * as a wildcard for all robots)
    - disallow: / (don't allow crawling on entire server)
    - disallow: /dir/ (don't allow crawling on directory named "dir")

    * leaving “disallow” field empty means everything is allowed

- If not specified:
- Meta tag: A Web-page author can also publish directions for crawlers. These are expressed by the meta tag with name robots, inside the HTML file.
  The options are:
  - index or noindex: index or do not index this file.
  - follow or nofollow: follow or do not follow the links of this file.

8.2 URL queue structure:

Politeness: Do not hit a web server too frequently.

Freshness: Keep indexed pages up-to-date.

Priority: Crawl more important pages more often.

K: number of different priority values.

B: number of back queues. $B = 3 * \text{number\_of\_crawler\_threads}$

1. First, the URLs flow into a prioritizer that decides which url goes into which queue. Each front queue holds all urls from the same corresponding priority.
   - There are K front queues for K priorities, each is a FIFO.
2. The biasd front queue selector chooses from which queue to extract according to prioritization.
3. From the front queue selector, the urls flow to the back queues.
   - All pages from the same back queue belong to the same host.
   - For each host, we remember when was the last time we extracted a page from that host. That is saved in a heap data structure so we can tell which queue has worked the most since the last time we read from the host.
   - Back queues are all always non-empty during a crawl.

Overview:

- Crawler thread requests a URL from back queue and is given a URL from a host least recently accessed.
- If queue is now empty, a URL is pulled from the front queues.
- If there is already a back queue for the URL’s host, add there and continue pulling from front queue. Otherwise create entry in empty queue.

8.3 Near duplicate detection:

We want to avoid crawling on duplicates of the same page (this is easy with hashing).

We also want to avoid near-duplicate pages which is harder.

To do so:

- Create a sketch for each page that is much smaller than the page.
  - Extract a set of features for each page.
  - Compare pages by comparing sets of features.

The two methods we will learn:

- Minhash is less complex and easier to tune.
- Simhash is typically much faster and requires less storage.

8.3.1 Minhash

w-shingle: A w-shingle of a document D is a sequence of w adjacent tokens in D.

S(D, w): The set of all w-shingles of D.

w-resemblance (r_w) of documents D1 and D2: The Jaccard Coefficient of the sets S(D1, w) and S(D2, w).

- $r_w(D_1, D_2) = \frac{|S(D_1,w) \cap S(D_2,w)|}{|S(D_1,w) \cup S(D_2,w)|}$
- Always between 0 and 1.
- A large w means higher resmblance that include less documents. A small w is more inclusive.

Set of all shingles is often larger than the original document. Instead, sample shingles from documents, and compute Jaccard coefficient of sample sets.

Minimal shingle sample:

Define a total ordering over S(D,w) and choose the minimal shingle.

If the sketch is the single minimal shingle, then: $D_1 = D_2 \Longrightarrow r_w(D_1, D_2) = 1$

Need a large set of shingles and a small sketch size.

Need shingle index.

Algorithm:

Let h be a hash function (e.g., SHA-256).

Let $t_1, \ldots, t_{200}$ be random strings.

Note: "+" sign between 2 strings means concatenation.

GetSketch(D, w):

```
• S = {}
• for i = 1 to 200:
    – m = MAX_INT
    – for s in S(D, w):
        ∗ if $h(s + t_i) < m$ :
            · $m = h(s + t_i)$
    – add m to S
• return S
```

This defines 200 random total orderings of the shingles. And for each of these orderings, take the minimal shingle for S.

8.3.2 Simhash:

Simhash: A simhash of a document is a hash word, such that if D1 and D2 are similar, their simhashes have low Hamming distance.

Simhash is a sequence of bits (usually 64) generated from a set of features, such that:

- changing a small fraction of the features will make little change to the sequence.
- changing a large fraction of the features will cause the sequence to look very different.

Document features can be words, shingles, n-grams or other things.

Generating a simhash: for each of the 64 bits, the corresponding simhash bit will be the average of the corresponding bit over all features. Round average results to 0 / 1, rounding up upon 0.5 (basically means the magority wins).

Finding simhashes with lower hamming distance:

Hamming Distance: The Hamming Distance between 2 bit sequences is the number of places in which they differ.

Explenation by example:

- Given a simhash F, suppose you want to find another simhash G within Hamming distance 3.
- Divide 64 bits into 6 blocks. Differing bits can appear in at most 3 blocks.
- There is some permutation of the blocks, such that all 3 differing blocks are at the lower order bits.
- Instead of storing a single simhash per document, store $\binom{6}{3} = 20$ versions of the simhash, per document, such that every 3 blocks is located at the least significant bits in some simhash.
- If simhashes are stored in sorted order, similar simhashes will be close to one another, allowing for efficient binary search.
- Once we have found simhashes with first 32 bits in common, computing Hamming distance.

9 Data streams:

Tuples (refered in the lectures as rows): Data comming from the stream.

We have a fixed working memory size.

We Have a fixed (but larger) archival storage.

We may be willing to get an approximate answer, instead of an exact answer, to save time and space.

Problem 1:

Search engine stream of tuples (user, query, time).

We have room to store 1/10 of the stream.

What fraction of the typical user's queries were repeated over the past month.

Solution:

- Pick 1/10 of users, and take all their searches in sample.
- Use a hash function that hashes the user name (or user id) into 10 values.
  - Store data if user name is hashed into value 0.

Problem 2:

Store a random sample of fixed size (e.g., at any time k, we should have a random sample of s elements).

Each of the k elements seen so far have the same probability s/k to be in the sample.

Solution:

- First, store all the first s elements of stream to S.
- Suppose we have seen k-1 elements and now the k'th arrives (k > s).
  - With probability s/k, keep the k'th element, else discard.
  - If we keep the k'th element, randomly choose one of the elements already in S to discard.

Claim: After k elements, the sample contains each element seen so far with probability s/k.

9.1 Bloom filter:

$S \subseteq Universe$: A set of elements.

m: $|S|$

Problem: Given an element, $x \in U$ we need to quickly decide wether $x \in S$.

- To take as little space as possible ,we allow false positives.
- if $x \in S$ we must answer "yes".

An array A[n] of n bits (space) , and k independent random hash functions.

- Main challenge is: given m and n, find k that minimizes the error.

Can be made to support deletetion but with more memory overhead.

Initialization:

- Set the array to 0s
- for each $s \in S$:
  - for $1 \leq i \leq k$:
    - $A\left[h_i\left(s\right)\right] \leftarrow 1$

For a given x:

- If $\exists 1 \leq i \leq k : A\left[h_i\left(s\right)\right] = 0$:
  - then $x \notin S$
- else:
  - $x \in S$

Chance for false positive:

m: $|S|$

k: number of hash functions

n: number of bits

Chance that some hash value will not be equal to b: $\frac{n-1}{n}$

Probability that none of the functions will equal to b: $\left(\frac{n-1}{n}\right)^{km} = \left(1-\frac{1}{n}\right)^{km} \rightarrow_{n\rightarrow\infty} e^{-\frac{km}{n}}$

Probability b is lit: $1 - e^{-\frac{km}{n}}$

Probability of a false positive: $\left(1 - e^{-\frac{km}{n}}\right)^k$

9.2 Counting distinct elements:

The challenge: a data stream consists of elements chosen from a set of size n. Maintain a count of the number of distinct elements seen so far.

- Use a small amount of memory, and estimate the correct value.
- Limit probability that the error is large.

FM (Flajolet-Martin):

Pick a hash function h that maps each of the n elements to at least $\log_2 n$ bits.

a: stream element.

r(a) (tail length): the number of trailing 0's in h(a).

R: The maximum r(a) seen for any a in the stream.

m: number of unique items.

The estimated number of distinct values in the stream according to h is 2^R.

Calculation:

- h(a) hashes elements uniformly at random.
- Probability that a random number ends in at least R 0-s is 2^{-R}
- The probability of not seeing a tail of length R among m elements: $(1-2^{-R})^m \approx e^{-m2^{-R}}$
- if $m << 2^R$, the probability of finding a tail of length r tends to 0.
- if $m >> 2^R$, the probability of finding a tail of length R tends to 1.

So 2^R will be around m.

Side note: this is not very precise. There is a workaround that involves using many hash functions and getting many samples from the stream.

9.3 Counting ones, DGIM (Datar-Gionis-Indyk-Motwani Algorithm):

Given a stream of 0's and 1's, Answer questions of the form "how many 1's in the last k bits?", where $k \leq N$.

Use O($\log^2$N) bits and get an estimate that has no more than 50% error

Represent the last N bits as a set of exponentially growing non-overlapping buckets.

Assume: each bit has a timestamp (i.e., position in which it arrives). Represent timestamp modulo N.

Each bucket has:

- timestamp of the most recent end (logN bits)
- size: number of 1-s in the bucket log(log(N)) bits
  - We know that size i is some 2^j, so only store j. j is at most log(N) and needs log(log(N)) bits.

Total number of buckets: $O\log(N)$

Total space for buckets: $O\log^2(N)$

Rules for representing stream by bucket:

- Right end of a bucket always has a 1.
- Every position with 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left.

Query answering:

- Find bucket b with earliest timestamp that includes at least some of the last k bits.
- Estimate number of 1-s as the sum of sizes of buckets to the right of b plus half of the size of b.
- Suppose that the leftmost bucket b included has size 2^j. The maximum error is half the size, i.e., 2^{j-1}.

Algorithm for maintaining the conditions:

Suppose we have a window of length N represented by buckets satisfying DGIM conditions.

When a new bit comes in:

- If its timestamp is not currentTimestamp – N: remove.
- If new bit is 0: do nothing
- If new bit is 1: create a new bucket of size 1.
- If there are now only 2 buckets of size 1: stop
- else: merge previous buckets of size 1 into bucket of size 2.
- If there are now only 2 buckets of size 2, stop.
- else: merge previous buckets of size 2 into buckets of size 4.
- etc ...