Basic Tokenizing, Indexing, and Implementation of Vector-Space Retrieval

Simple Tokenizing

- Analyze text into a sequence of discrete tokens (words).
- Sometimes punctuation (e-mail), numbers (1999), and case (Republican vs. republican) can be a meaningful part of a token.
- However, usually they are not.
- Simplest approach is to ignore all numbers and punctuation and use only case-insensitive unbroken strings of alphabetic characters as tokens.

Tokenizing HTML

- Should text in HTML commands not typically seen by the user be included as tokens?
  - Words appearing in URLs.
  - Words appearing in “meta text” of images.
- Simplest approach used in VSR is to exclude all HTML tag information from tokenization.
  - Parses HTML using utilities in Java Swing package, and collects all raw text.

Stopwords

- It is typical to *exclude* high-frequency words (e.g. function words: “a”, “the”, “in”, “to”; pronouns: “I”, “he”, “she”, “it”).
- Stopwords are language dependent. VSR uses a standard set of about 500 for English.
- For efficiency, store strings for stopwords in a hashtable to recognize them in constant time.

Stemming

- Reduce tokens to "root" form of words to recognize morphological variation.
  - "computer", "computational", "computation" all reduced to same token "compute"
- Correct morphological analysis is language specific and can be complex.
- Stemming "blindly" strips off known affixes (prefixes and suffixes) in an iterative fashion.

Porter Stemmer

- Simple procedure for removing known affixes in English without using a dictionary.
- Can produce unusual stems that are not English words:
  - "computer", "computational", "computation" all reduced to same token "comput"
- May conflate (reduce to the same token) words that are actually distinct.
- Not recognize all morphological derivations.

Porter Stemmer Errors

- Errors of "comission":
  - organization, organ → organ
  - police, policy → polic
  - arm, army → arm
- Errors of "omission":
  - cylinder, cylindrical
  - create, creation
  - Europe, European

Sparse Vectors

- Vocabulary and therefore dimensionality of vectors can be very large, ~10^4 .
- However, most documents and queries do not contain most words, so vectors are sparse (i.e. most entries are 0).
- Need efficient methods for storing and computing with sparse vectors.

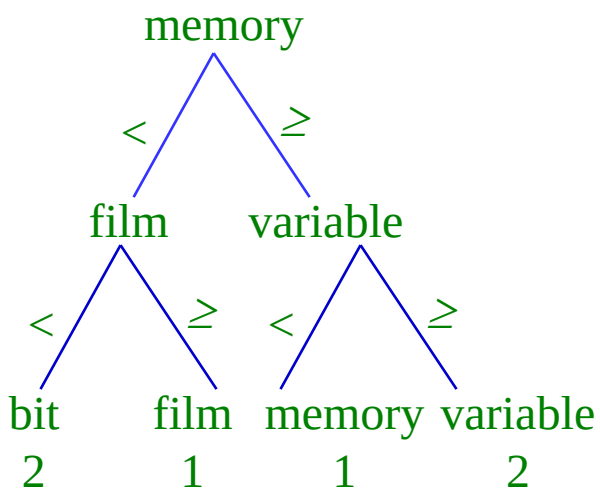
Sparse Vectors as Lists

- Store vectors as linked lists of non-zero-weight tokens paired with a weight.
  - Space proportional to number of unique tokens (n) in document.
  - Requires linear search of the list to find (or change) the weight of a specific token.
  - Requires quadratic time in worst case to compute vector for a document:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Sparse Vectors as Trees

- Index tokens in a document in a balanced binary tree or trie with weights stored with tokens at the leaves.



Balanced Binary Tree

Sparse Vectors as Trees (cont.)

- Space overhead for tree structure: ~2n nodes.
- O(log n) time to find or update weight of a specific token.
- O(n log n) time to construct vector.
- Need software package to support such data structures.

Sparse Vectors as HashTables

- Store tokens in hashtable, with token string as key and weight as value.
  - Storage overhead for hashtable ~1.5*n*.
  - Table must fit in main memory.
  - Constant time to find or update weight of a specific token (ignoring collisions).
  - O(*n*) time to construct vector (ignoring collisions).

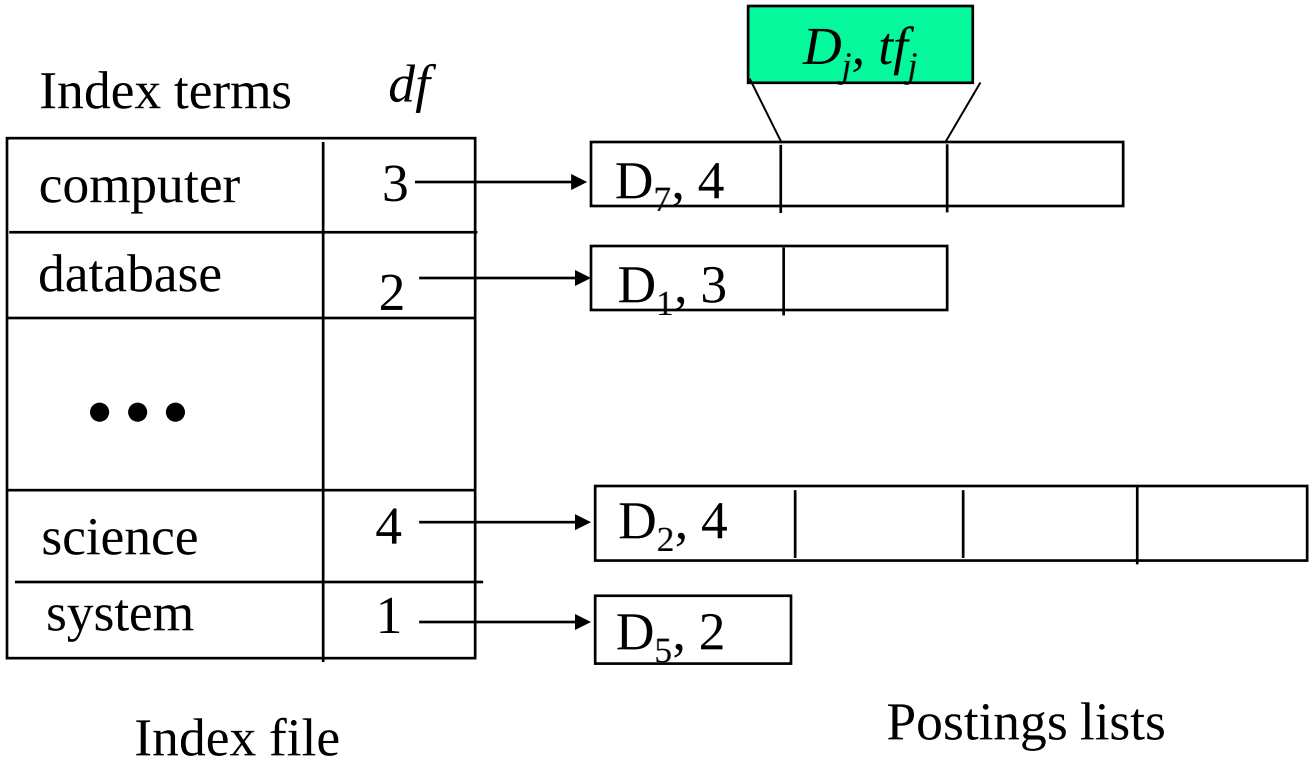
Sparse Vectors in VSR

- Uses the hashtable approach called a HashMapVector.
- The hashMapVector() method of a Document computes and returns a HashMapVector for the document.
- hashMapVector() only works once after initial Document creation (i.e. Document object does **not** store it internally for later reuse).

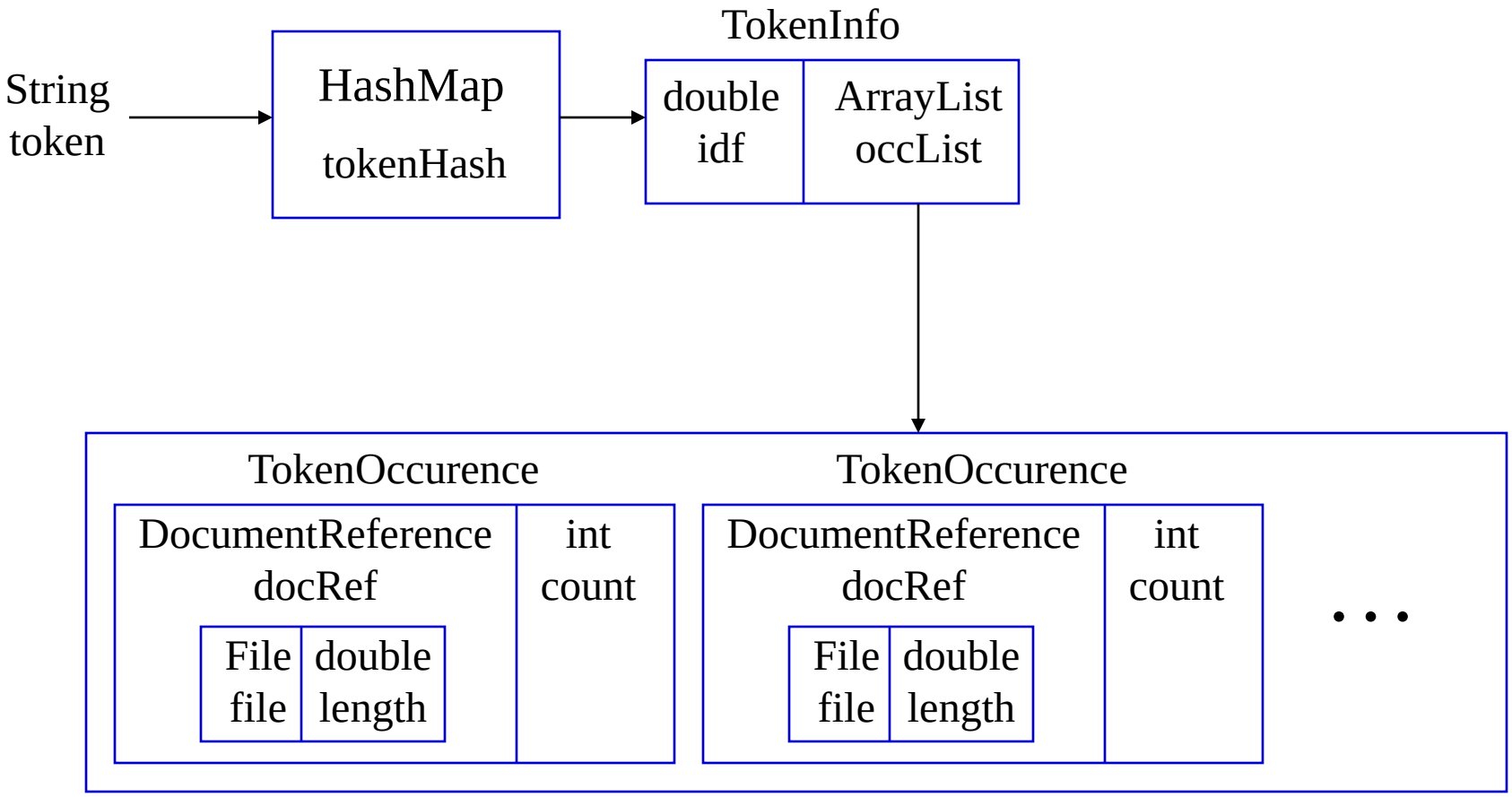
Implementation Based on Inverted Files

- In practice, document vectors are not stored directly; an inverted organization provides much better efficiency.
- The keyword-to-document index can be implemented as a hash table, a sorted array, or a tree-based data structure (trie, B-tree).
- Critical issue is logarithmic or constant-time access to token information.

Inverted Index



VSR Inverted Index



Creating an Inverted Index

Create an empty HashMap, H;
For each document, D, (i.e. file in an input directory):
    Create a HashMapVector,V, for D;
    For each (non-zero) token, T, in V:
        If T is not already in H, create an empty
            TokenInfo for T and insert it into H;
        Create a TokenOccurence for T in D and
            add it to the occList in the TokenInfo for T;
Compute IDF for all tokens in H;
Compute vector lengths for all documents in H;

Computing IDF

Let N be the total number of Documents;

For each token, T, in H:

    Determine the total number of documents, M,

        in which T occurs (the length of T's occList);

    Set the IDF for T to log(N/M);

Note this requires a second pass through all the tokens after all documents have been indexed.

Document Vector Length

- Remember that the length of a document vector is the square-root of sum of the squares of the weights of its tokens.
- Remember the weight of a token is:

  TF * IDF
- Therefore, must wait until IDF's are known (and therefore until all documents are indexed) before document lengths can be determined.

Computing Document Lengths

Assume the length of all document vectors (stored in the DocumentReference) are initialized to 0.0;

For each token T in H:

Let, I, be the IDF weight of T;

For each TokenOccurence of T in document D

Let, C, be the count of T in D;

Increment the length of D by $(I*C)^2$;

For each document D in H:

Set the length of D to be the square-root of the current stored length;

Minimizing Iterations Through Tokens

- To avoid iterating though all tokens twice (after all documents are already indexed), computing IDF's and vector lengths are combined in one iteration in VSR.

Time Complexity of Indexing

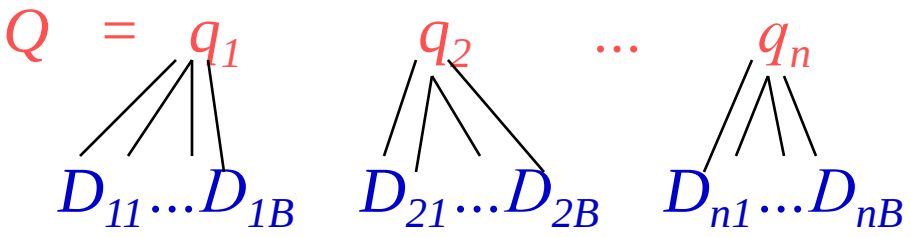
- Complexity of creating vector and indexing a document of n tokens is $O(n)$.
- So indexing m such documents is $O(m\ n)$.
- Computing token IDFs for a vocabularly V is $O(|V|)$.
- Computing vector lengths is also $O(m\ n)$.
- Since $|V| \leq m\ n$, complete process is $O(m\ n)$, which is also the complexity of just reading in the corpus.

Retrieval with an Inverted Index

- Tokens that are not in both the query and the document do not effect cosine similarity.
  - Product of token weights is zero and does not contribute to the dot product.
- Usually the query is fairly short, and therefore its vector is *extremely* sparse.
- Use inverted index to find the limited set of documents that contain at least one of the query words.

Inverted Query Retrieval Efficiency

- Assume that, on average, a query word appears in B documents:



- Then retrieval time is O(|Q| B), which is typically, **much** better than naïve retrieval that examines all N documents, O(|V| N), because |Q| << |V| and B << N.

Processing the Query

- Incrementally compute cosine similarity of each indexed document as query words are processed one by one.
- To accumulate a total score for each retrieved document, store retrieved documents in a hashtable, where DocumentReference is the key and the partial accumulated score is the value.

Inverted-Index Retrieval Algorithm

Create a HashMapVector, Q, for the query.
Create empty HashMap, R, to store retrieved documents with scores.
For each token, T, in Q:
    Let I be the IDF of T, and K be the count of T in Q;
    Set the weight of T in Q: W = K * I;
    Let L be the list of TokenOccurences of T from H;
    For each TokenOccurence, O, in L:
        Let D be the document of O, and C be the count of O (tf of T in D);
        If D is not already in R (D was not previously retrieved)
            Then add D to R and initialize score to 0.0;
        Increment D's score by W * I * C; (product of T-weight in Q and D)

Retrieval Algorithm (cont)

Compute the length, L, of the vector Q (square-root of the sum of the squares of its weights).

For each retrieved document D in R:

Let S be the current accumulated score of D;

(S is the dot-product of D and Q)

Let Y be the length of D as stored in its DocumentReference;

Normalize D's final score to S/(L * Y);

Sort retrieved documents in R by final score and return results in an array.

Efficiency Note

- To save computation and an extra iteration through the tokens in the query, in VSR, the computation of the length of the query vector is integrated with the processing of query tokens during retrieval.

User Interface

Until user terminates with an empty query:

Prompt user to type a query, Q.

Compute the ranked array of retrievals R for Q;

Print the name of top N documents in R;

Until user terminates with an empty command:

Prompt user for a command for this query result:

1) Show next N retrievals;

2) Show the Mth retrieved document;

(document shown in Firefox window)