Day 4 & 5
Term Weighting & Filtering
The initial m doc set has turned into k terms

- Each term memorizes the frequency with which it appeared in each document
- k*m matrix where row = doc, col = term

Document-Term Matrix (DTM)

	T1	T2	T3
D1	5 (term T1 appeared in D1 5 times)	0	2
D2	0	3	1
D3	2	0	0
D4	0	2	1

Inverted (index) File
Transposing the DTM results in a Term-Document Matrix (TDM) which functions as an index

- TDM is usually called the “inverted file” in principle

In practice simple TDM decreases utility

- The matrix is very sparse (most elements = 0)
- It takes a long time to find the query term in the index

(Transpose of DTM above)	D1	D2	D3	D4
T1	5 (frequency of the term in the doc)	0	2	0
T2	0	3	0	2
T3	2	1	0	1

T1 -> D1 -> D3
T2 -> D2 -> D4
T3 -> D1 -> D2 -> D4
Xxx???? list solves the sparsity problem

- Google index based on everything - ngram, word, phrase ….
- Ngram variation - skip-gram
- We have billions of terms & docs, in practice to use a matrix

Searching Algorithm & Data Structures: Preliminary
L[n] = L[0], L[1], …, L[n-1]: a list of integers

- In our case, L is a list of index terms
- Any data can be asked if the order of any possible pairs of instances is defined

Task: Given w as a query, return such x that satisfies L[x] (index)= w (term) (finding position of w in list L)

- We want to minimize of comparison for 2 values
- Time complexity f(n) = O(g(n)); n -> ∞

Linear Search:

- w is compared to L[x] 1 by 1 while scanning L -> r
- 9 11 19 6 13 15 17 12 9

Binary Search:

- W is compared to the middle of remaining list and moves to the left (w < L[x]) or right (w > L[x]), otherwise found w = L[x]
- Requires a sorted list, so we preprocess the list by sorting it.
- 1(left) 3 4 6 **9(w = 4, start)** 11 12 15 17 19(right)
- w(=4) < 9, move to the left
- w(=4) < 6, move to the left
- w(=4) =4 , found
- L[2] = w

Linear search only excludes 1 element in each iteration
Binary search can halve halves the search space in each iteration

- Disadvantage of binary search: you have to re-sort the list every time the data is updated
- How to overcome this weakness?
  - Use a BST

Logical Operation

- Logical operator (n=#operands)
  - Unary (n=1) NOT
  - Binary (n=2) OR, AND
- Order of precedence: NOT > AND > OR
- Truth table

A	NOT A
0	1
1	0

A	B	A∪B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A∩B
0	0	0
0	1	0

1	0	0
1	1	1

Fundamental Retrieval Models

- Exact match: Boolean
- Best match (or Ranking)
    - Vector space
    - Probabilistic
    - Language
    - Learning to rank (L2R)
    - (Neural network)

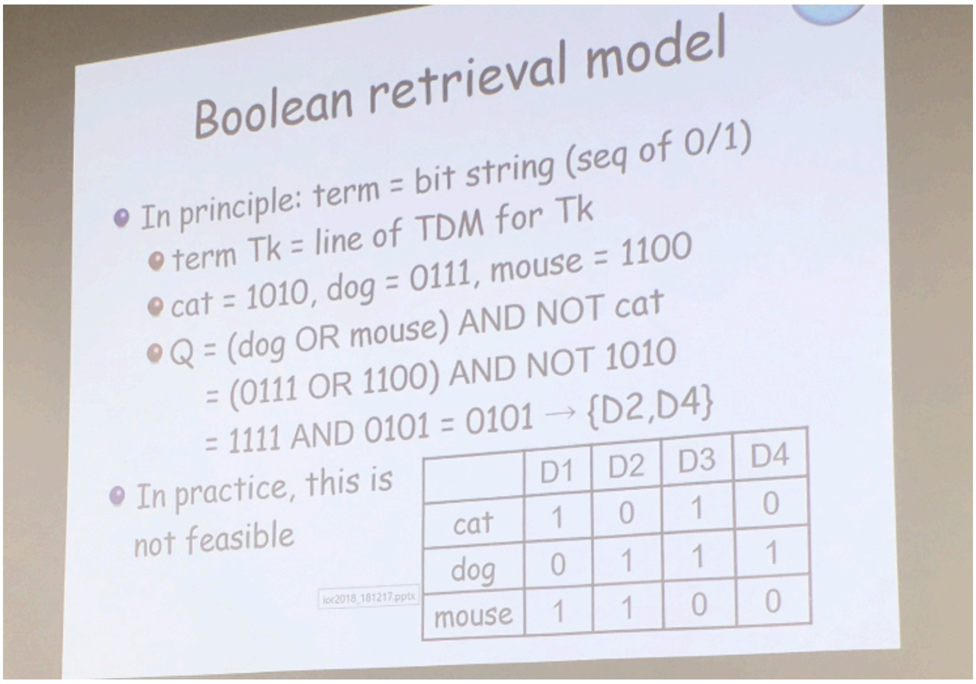
Boolean retrieval model

- In principle: term = bit string (seq of 0/1)
    - Term Tk = line of TDM (term dead mining) for Tk
- Query Q = a logical formula of Q1 … Qi …
    - Retrieved docs R = solution for Q
    - Di is retrieved if Di >0

  Q = (dog OR mouse) AND NOT cat

  R = {D2, D4}
- But, in practice it is prohibitive to compute l-o-n-g bit strings

	D1	D2	D3	D4
Cat	1	0	1	0
Dog	0	1	1	1
Mouse	1	1	0	0



Improving Boolean IR

List of docID containing the head term in ascending order

Dictionary		Postings
Bison	->	1 -> 22 -> 53 -> 145 -> #
Cat	->	1 -> 53 -> 80 -> 172 -> -> 256 -> 404 -> #
Dog	->	9 -> 44 -> 53 -> # sentinel (the end of list)

Efficiently identify and obtain documents whose bit element is 1

Example: Q = bison OR cat (resembles MERGE)

Any D(A) or D(B) can be added to the end of R: D(x) is docID where cursor x is pointing to

docID cannot be duplicated in R

If A or B meets sentinel (#), residue is added to R

 if D(A) = # then D(C) <- L(B) (the rest of list B after A #)

 if D(A) < D(B) then D(C) <- D(A), A++, C++ (if docID from two cursors different, pick the smaller one because the larger ID might appear in D(A) later but docID cannot be duplicated in R)

 if D(A) = D(B) then D(C) <- D(A), A++, B++, C++

Bison	->	1 -> 22 (cursor A) -> 53 -> 145 -> **#**
Cat	->	1 -> 53 (cursor B) -> 80 -> **172** -> **256** -> **404** -> **#**
R	->	1 -> 22 (cursor C) -> … **#**

***目的：比较A&B两个list的每一个docID，按照上面的规则copy到R