Information Retrieval
CS 6900

Lecture 02

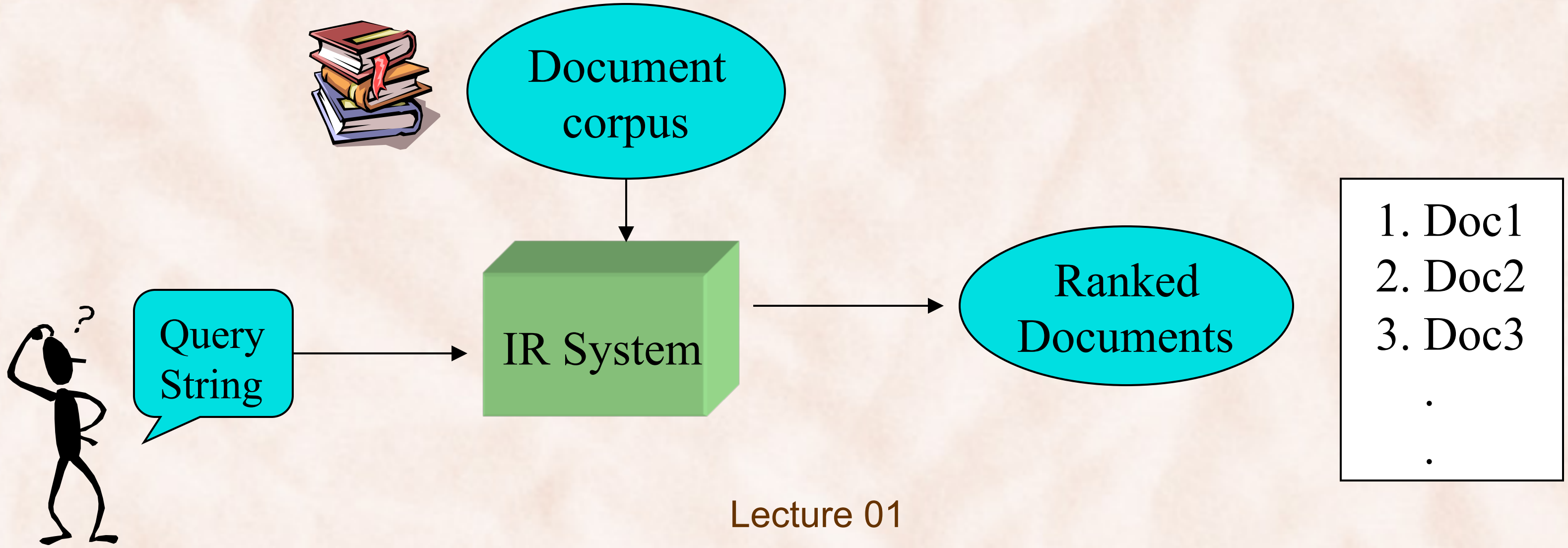
Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

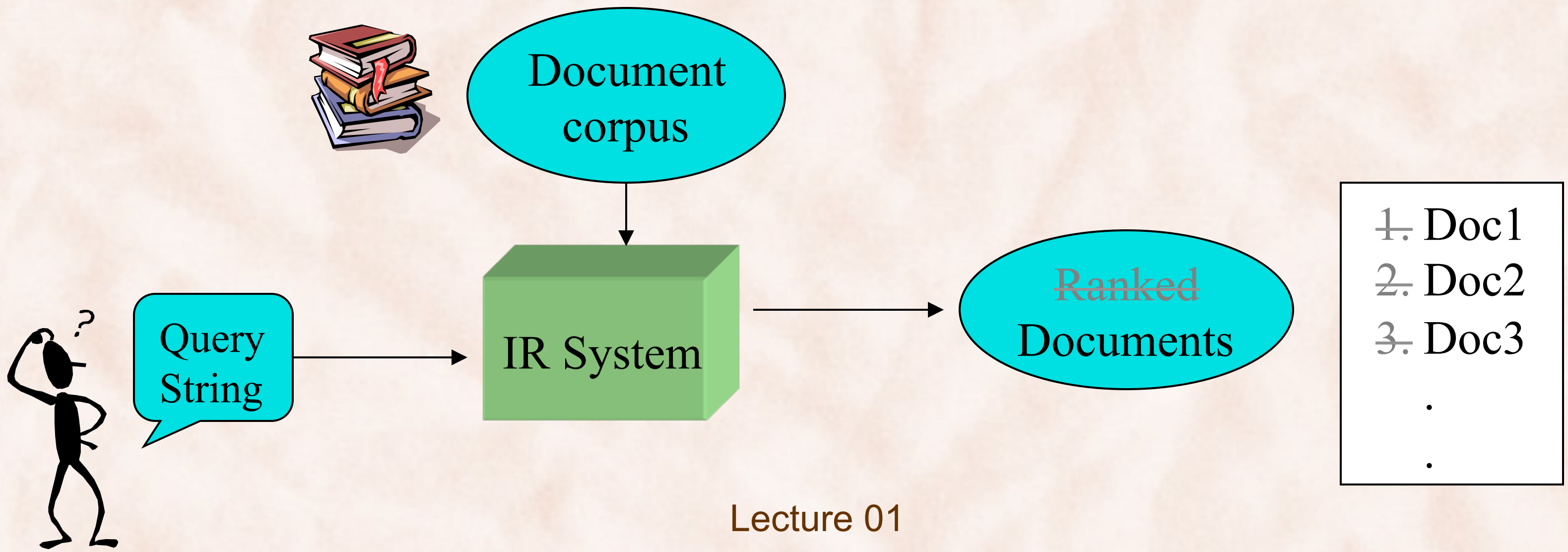
Typical IR task

- Input:
  - A large collection of unstructured text documents.
  - A user query expressed as text.
- Output:
  - A ranked list of documents that are relevant to the query.



Boolean ~~Typical~~ IR task

- Input:
  - A large collection of unstructured text documents.
  - A user query expressed as text.
- Output:
  - A ~~ranked~~ list of documents that are relevant to the query.



Boolean Retrieval

- Information Need: *Which plays by Shakespeare mention Brutus and Caesar, but not Calpurnia?*
- Boolean Query: Brutus AND Caesar AND NOT Calpurnia
- Possible search procedure:
  - Linear scan through all documents (Shakespeare's collected works).
  - Compile list of documents that contain Brutus and Caesar, but not Calpurnia.
  - Advantage: simple, it works for moderately sized corpora.
  - Disadvantage: **need to do linear scan for every query ⇒ slow for large corpora.**

Term-document incidence matrices

- Precompute a data structure that makes search fast for every query.

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if document contains word, 0 otherwise

Term-document incidence matrix M

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Query = **Brutus** AND **Caesar** AND NOT **Calpurnia**

Answer = M(Brutus) ∧ M(Caesar) ∧ ¬M(Calpurnia)
= 1 1 0 1 0 0 ∧ 1 1 0 1 1 1 ∧ 1 0 1 1 1 1
= **1 0 0 1 0 0**
⇒ Anthony and Cleopatra, Hamlet

110100 ∧
110111 ∧
101111

100100

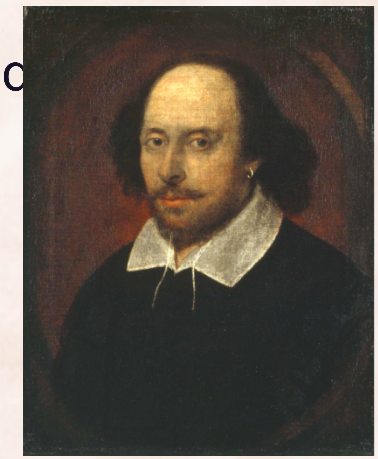
Answers to Query

- **Antony and Cleopatra**, Act III, Scene ii

  Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,

  When Antony found Julius ***Caesar***

  c



He cried almost to roaring; and he wept

When at Philippi he found ***Brutus*** slain.

- **Hamlet**, Act III, Scene ii

  Lord Polonius: I did enact Julius ***Caesar*** I was killed i' the Capitol; ***Brutus*** killed me.

Scalability: Dense Format

- Assume:
  - Corpus has 1 million documents.
  - Each document is about 1,000 words long.
  - Each word takes 6 bytes, on average.
  - Of the 1 billion word tokens 500,000 are unique.

- Then:
  - Corpus storage takes:
    - 1M * 1, 000 * 6 = 6GB ✓
  - Term-Document incidence matrix would take:
    - 500,000 * 1,000,000 = $0.5 * 10^{12}$ bits ✗

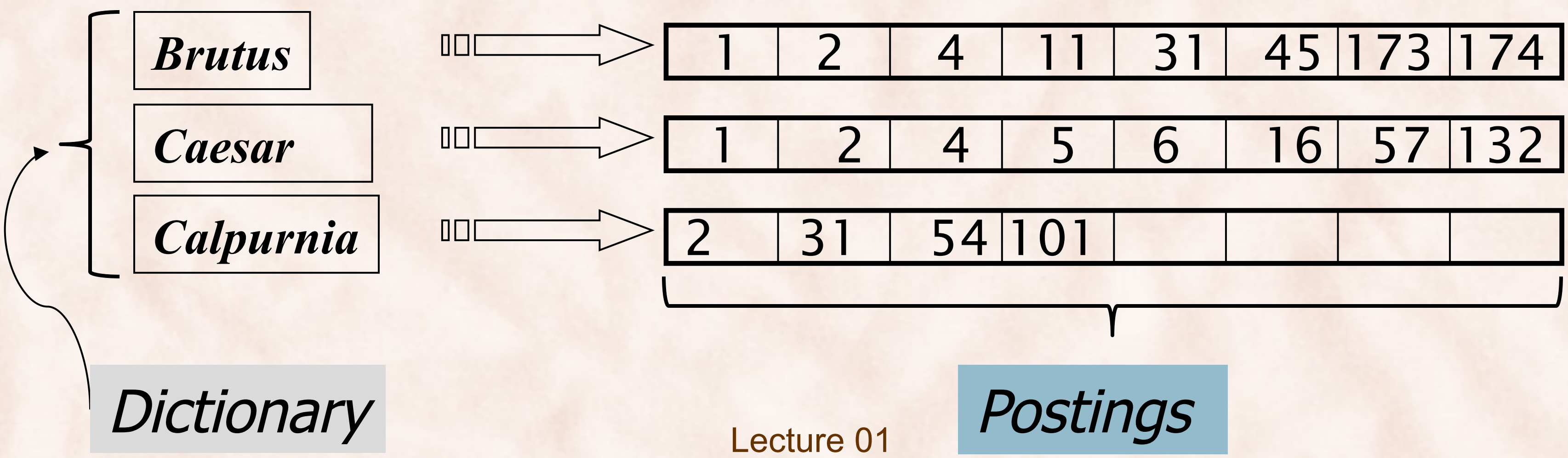
Scalability: Sparse Format

- Of the 500 billion entries, at most 1 billion are non-zero.
  - ⇒ at least 99.8% of the entries are zero.
  - ⇒ use a sparse representation to reduce storage size!

- Store only non-zero entries ⇒ **Inverted Index**.

Inverted Index for Boolean Retrieval

- **Map** each term to a **posting list** of documents containing it:
  - Identify each document by a numerical **docID**.
  - **Dictionary** of **terms** usually in memory.
  - Posting list:
    - linked lists of variable-sized array, if in memory.
    - contiguous run of postings, if on disk.



Inverted Index: Step 1

- Assemble sequence of ⟨token, docID⟩ pairs.
  - assume text has been tokenized (next lecture).

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Inverted Index: Step 2

- Sort by terms, then by docIDs.

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Inverted Index: Step 3

- Merge multiple term entries per document.
- Split into **dictionary** and **posting lists**.
  - keep posting lists sorted, for efficient query processing.
- Add document frequency information:
  - useful for efficient query processing.
  - also useful later in document ranking.

Inverted Index: Step 3

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Query Processing: AND

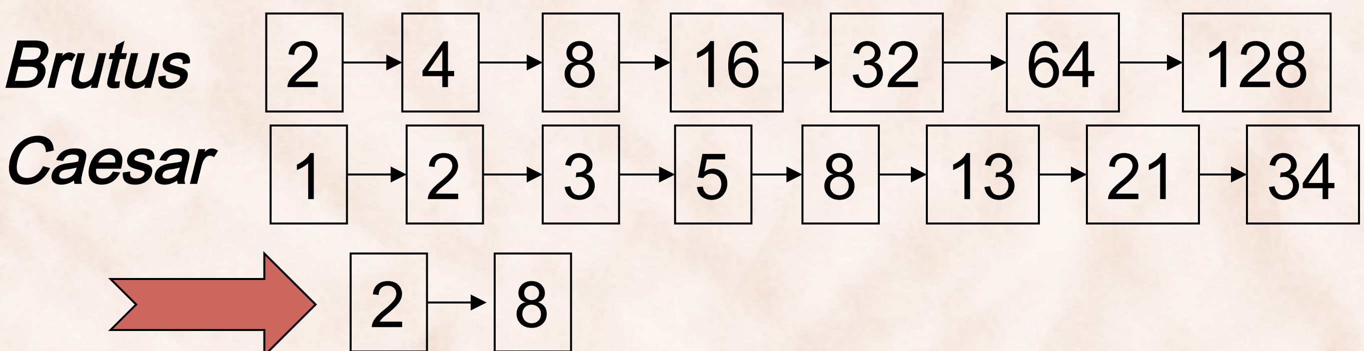
- Consider processing the query:

  Brutus** AND **Caesar

  – Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.

  – Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.

  – "Merge" the two postings (intersect the document sets):



Query Processing: AND

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
        then ADD(answer, docID(p1))
             p1 ← next(p1)
             p2 ← next(p2)
        else if docID(p1) < docID(p2)
                then p1 ← next(p1)
                else p2 ← next(p2)
  return answer
```

Query Processing: OR

```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(
      then ADD(answer,
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
             then p1 ← next(p1)
             else p2 ← next(p2)
return answer
```

Add(*answer*, *docID*(p_1))

Add(*answer*, *docID*(p_2))

Query Processing: NOT

- Exercise: Adapt the merge for the queries:

  Brutus** AND NOT **Caesar

  Brutus** OR NOT **Caesar

- Can we still run through the merge in time O(x+y)?

- Exercise: What about an arbitrary Boolean formula?

  ***(Brutus** OR **Caesar)** AND NOT*

  (Antony** OR **Cleopatra)

Query Optimization: What is the best order for query processing?

- Consider a query that is an *AND* of *n* terms.

Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Brutus 2 → 4 → 8 → 16 → 32 → 64 → 128

Caesar 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Calpurnia 13 → 16

- For each of the *n* terms, get its postings, then *AND* them together.
- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*
  - use document frequencies stored in the dictionary.

⇒ execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***

Query Optimization

- Exercise: recommend a query processing order for:
  - ***(tangerine** OR **trees)** AND*
    ***(marmalade** OR **skies)** AND*
    (kaleidoscope** OR **eyes)
  - which two terms should we process first?

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

  - Get document frequencies (DF) for all terms.
  - Estimate the size of each OR by the sum of its DF's.
  - Process in order of increasing OR sizes
    - *start with smallest set, then keep cutting further.*
    - use document frequencies stored in the dictionary.

Extended Boolean Model

- **Phrase Queries**:
  - Want to answer query "Ohio University", as a phrase.
  - The concept of phrase queries is one of the few "advanced search" ideas has proven easily understood by users.
    - about 10% of web queries are phrase queries.
    - many more are *implicit phrase queries* (e.g. person names).
- **Proximity Queries**:
  - Altavista: Python NEAR language
  - WestLaw: limit! \3 statute \3 federal \2 tort
  - Google: Python * language
  - many search engines use keyword proximity *implicitly*.

Solution 1 for Phrase Queries: Biword Indexes

- Index every two consecutive tokens in the text.
  - Treat each biword as a vocabulary term.
  - The text "*modern information retrieval*" generates **biwords**:
    - *modern information*
    - *information retrieval*
  - Bigram phrase querry processing is now straightforward.
  - Longer phrase queries?
    - Heuristic solution: break them into conjunction of biwords.
      - Query "electrical engineering and computer science":
        - » "electrical engineering" AND "engineering and" AND "and computer" AND "computer science"
    - Without verifying the retrieved docs, can have false positives!

Biword Indexes

- Can have false positives:
  - Unless retrieved docs are verified $\Rightarrow$ increased time complexity.
- Larger dictionary leads to index blowup:
  - clearly unfeasible for ngrams larger than bigrams.

$\Rightarrow$ not a standard solution for phrase queries:

- but useful in compound strategies.

Solution 2 for Phrase Queries: Positional Indexes

- In the postings list:
  - for each token *tok*:
    - for each document *docID*:
      - store the positions in which *tok* appears in *docID*.
        - » < ***be***: 993427;

          1: 7, 18, 33, 72, 86, 231;

          2: 3, 149;

          4: 17, 191, 291, 430, 434;

          5: 363, 367, … >
        - » which documents might contain "to be or not to be"?

Positional Indexes: Query Processing

- Use a merge algorithm at two levels:
  1. Postings level, to find matchings docIDs for query tokens.
  2. Document level, to find consecutive positions for query tokens.

  – Extract index entries for each distinct term: ***to, be, or, not.***
  – Merge their *doc:position* lists to enumerate all positions with "***to be or not to be***".
    - ***to****:* *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
    - ***be****:* *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...

- Same general method for proximity searches.

Proximity Queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  - Again, here, /*k* means “within *k* words of”.

- Cannot use biword indexes.

- Can use positional indexes:
  - Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of *k*?
  - This is a little tricky to do correctly and efficiently.
  - Algorithm in Figure 2.12 of IIR.

```
PositionalIntersect(p1, p2, k)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then l ← ⟨ ⟩
           pp1 ← positions(p1)
           pp2 ← positions(p2)
           while pp1 ≠ NIL
           do while pp2 ≠ NIL
                 do if |pos(pp1) − pos(pp2)| ≤ k
                       then Add(l, pos(pp2))
                       else if pos(pp2) > pos(pp1)
                               then break
                    pp2 ← next(pp2)
                 while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
                 do Delete(l[0])
                 for each ps ∈ l
                 do Add(answer, ⟨docID(p1), pos(pp1), ps⟩)
                 pp1 ← next(pp1)
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

Positional Index: Size

- Need an entry for each occurrence, not just for each document.
- Index size depends on average document size:
  - Average web page has less than 1000 terms.
  - SEC filings, books, even some epic poems … easily 100,000 terms.
    - large documents cause an increase of 2 orders of magnitude.
  - Consider a term with frequency 0.1%:

Document size	Expected postings	Expected entries in positional posting
1000	1	1
100,000	1	100

Positional Index

- A positional index expands postings storage *substantially*.
  - 2 to 4 times as large as a non-positional index
  - compressed, it is between a third and a half of uncompressed raw text.

- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries:
  - whether used explicitly or implicitly in a ranking retrieval system.

Combined Strategy

- Biword and positional indexes can be fruitfully combined:
  - For particular phrases (***"Michael Jackson", "Britney Spears"***) it is inefficient to keep on merging positional postings lists
    - Even more so for phrases like ***"The Who"***. Why?

1. Use a phrase index, or a biword index, for certain queries:
   - Queries known to be common based on recent querying behavior.
   - Queries where the individual words are common but the desired phrase is comparatively rare.
2. Use a positional index for remaining phrase queries.

Boolean Retrieval vs. Ranked Retrieval

- Many users (professionals) prefer Boolean query models:
  - Boolean queries are precise: a document either matches the query or it does not.
    - Greater control and transparency over what is retrieved.
  - Some domains allow an effective ranking criterion:
    - Westlaw returns documents in reverse chronological order.

- Hard to tune precision vs. recall:
  - AND operator tends to produce high precision but low recall.
  - OR operator gives low precision but high recall.
  - Difficult/impossible to find satisfactory middle ground.

Boolean Retrieval vs. Ranked Retrieval

- Need an effective method to rank the matched documents.
  - Give more weight to documents that mention a token several times vs. documents that mention it only once.
    - record **term frequency** in the postings list.

- Web search engines implement ranked retrieval models:
  - Most include at least partial implementations of Boolean models:
    - Boolean operators.
    - Phrase search.
  - Still, improvements are generally focused on free text queries.