COP509

Natural Language Processing

Boolean Information Retrieval

Prof. Georgina Cosma

Department of Computer Science
Loughborough University

g.cosma@lboro.ac.uk

January 2026

Learning Outcomes

By the end of this session, you will be able to:

1. Define information retrieval and explain its importance in NLP
2. Explain the purpose and structure of an inverted index
3. Construct an inverted index from a small document collection
4. Process Boolean queries using AND, OR, and NOT operators
5. Apply query optimisation techniques for conjunctive queries
6. Evaluate the advantages and limitations of Boolean retrieval

Outline

1. Introduction to Information Retrieval
2. The Inverted Index
3. Processing Boolean Queries
4. Query Optimisation
5. Advantages and Limitations
6. Summary

Part 1

Introduction to Information Retrieval

Where Does Information Retrieval Fit in NLP?

Natural Language Processing (NLP) is the field of AI concerned with enabling computers to understand, interpret, and generate human language.

Information Retrieval (IR) is a core subfield of NLP that focuses on:

- Finding relevant documents from large collections
- Matching user queries to stored information
- Ranking results by relevance

Key Point

IR provides the foundation for search engines, question answering systems, and many other NLP applications. Understanding IR is essential before studying more advanced NLP techniques.

Definition of Information Retrieval

Information retrieval (IR) is **finding** material (usually documents) of an **unstructured** nature (usually text) that satisfies an **information need** from within **large collections** (usually stored on computers).

Key Terms Explained

- **Unstructured data**: Data without a predefined format (e.g., text documents, web pages) – as opposed to structured data in databases with rows and columns
- **Information need**: What the user is looking for (expressed as a query)
- **Large collections**: Millions or billions of documents (e.g., the web)

Real-World Applications of Information Retrieval

IR powers many systems you use every day:

Web Search

- Google, Bing, DuckDuckGo
- Billions of queries per day

Enterprise Search

- Searching company documents
- Email search (Gmail, Outlook)
- Slack, Microsoft Teams

Specialised Search

- Legal databases (Westlaw, LexisNexis)
- Medical literature (PubMed)
- Academic papers (Google Scholar)

Personal Search

- Desktop search (Windows, Spotlight)
- Photo search by content

Boolean Retrieval: The Simplest IR Model

- The **Boolean model** is the simplest and oldest model for information retrieval
- Queries are **Boolean expressions** using AND, OR, and NOT operators
- Example: `CAESAR AND BRUTUS`
- The search engine returns **all documents** that satisfy the Boolean expression
- Results are **not ranked** – a document either matches or it doesn't

Historical Note

Boolean retrieval was the dominant commercial search model from the 1970s until the mid-1990s. Many professional search systems (legal, medical) still support it today.

Part 2

The Inverted Index

A Motivating Example: Shakespeare's Plays

Imagine you have access to all 37 of Shakespeare's plays and want to search them. How would you find plays containing specific words or combinations of words?

The Naive Approach: Why Not Just Search the Text?

Question: Which plays of Shakespeare contain the words `BRUTUS AND CAESAR`, but not `CALPURNIA`?

Naive approach: Use `grep` to search through all plays for `BRUTUS` and `CAESAR`, then filter out lines containing `CALPURNIA`.

Why is grep not a good solution?

- **Slow** for large collections (must scan every document for every query)
- `grep` is **line-oriented**, but IR is **document-oriented**
- "NOT CALPURNIA" is **non-trivial** to implement
- **Complex queries** (e.g., "ROMANS near COUNTRYMEN") are not feasible

`grep` Overview

`grep` (Global Regular Expression Print) is a Unix command that searches text files line-by-line for patterns matching a regular expression.

Key Terminology: Documents, Terms, and Tokens

Before we proceed, let's clarify some important terminology:

Definitions

- **Document**: A unit of retrieval (e.g., a web page, an email, a book chapter, a play)
- **Collection (Corpus)**: A set of documents to be searched
- **Token**: An instance of a sequence of characters that forms a meaningful unit (e.g., each occurrence of "the" is a token)
- **Term**: A normalised token that is used for indexing (e.g., "running" $\rightarrow$ "run")
- **Type**: A distinct (i.e. unique) term in the vocabulary ignoring repetitions

Example: In "To be or not to be", there are 6 tokens but only 4 types (to, be, or, not).

Term-Document Incidence Matrix

One way to represent which terms appear in which documents is a **term-document matrix**:

	Anthony and Cleopatra	**Julius Caesar**	**The Tempest**	**Hamlet**	**Othello**	**Macbeth**
ANTHONY	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

Entry is **1** if the term occurs in the document, **0** otherwise.
Example: CALPURNIA occurs in *Julius Caesar* but not in *The Tempest*.

Using Incidence Vectors for Boolean Queries

Each term has a corresponding **binary vector** (one bit per document).

To answer `BRUTUS AND CAESAR AND NOT CALPURNIA`:

1. Get the vectors for BRUTUS, CAESAR, and CALPURNIA
2. **Complement** the CALPURNIA vector (flip all bits)
3. Perform **bitwise AND** on all three vectors

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	**1**

Truth Table: AND returns 1 only when **both** inputs are 1

Worked Example: Boolean Query Processing

Query: BRUTUS AND CAESAR AND NOT CALPURNIA

Step 1: Get the original vectors from the matrix

	A&C	**JC**	**TT**	**Ham**	**Oth**	**Mac**
BRUTUS	1	1	0	1	0	0
CAESAR	1	1	0	1	1	1
CALPURNIA	0	1	0	0	0	0

Step 2: Complement CALPURNIA (for NOT operation)

	A&C	**JC**	**TT**	**Ham**	**Oth**	**Mac**
BRUTUS	1	1	0	1	0	0
CAESAR	1	1	0	1	1	1
NOT CALPURNIA	1	0	1	1	1	1

Step 3: Perform bitwise AND on all three vectors

`110100 AND 110111 AND 101111 = 100100`

Result: Documents 1 (Anthony & Cleopatra) and 4 (Hamlet) match the query.

Understanding the NOT Operation

The key insight is how we handle "NOT" in Boolean retrieval:

1. **Original CALPURNIA vector:** 010000
   - 1s indicate documents that **contain** CALPURNIA
   - Only Julius Caesar (position 2) contains CALPURNIA
2. **Complemented vector (NOT CALPURNIA):** 101111
   - 1s now indicate documents that **do NOT contain** CALPURNIA
   - All documents except Julius Caesar
3. **Why AND works:**
   - AND returns 1 only when **all** conditions are true
   - Position gets 1 only if: has BRUTUS **AND** has CAESAR **AND** lacks CALPURNIA

Final calculation: 110100 AND 110111 AND 101111 = **100100**

Verifying Our Answer

Our query returned documents 1 and 4. Let's verify:

Anthony and Cleopatra, Act III, Scene ii:

> *Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,*
> *When Antony found Julius* ***Caesar*** *dead,*
> *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.*

Both plays contain BRUTUS and CAESAR, and neither contains CALPURNIA. ✓

The Problem: Scaling to Real Collections

Consider a realistic document collection:

- $N = 10^6$ documents (1 million), each with $\sim$1000 tokens
- Total: 10^9 tokens (1 billion)
- At $\sim$6 bytes per token: collection size $\approx$ 6 GB
- Assume M = 500,000 distinct terms (vocabulary size)

Size of the term-document matrix:

- $M \times N = 500{,}000 \times 10^6 = 5 \times 10^{11}$ entries
- That's **500 billion** bits = 62.5 GB just for the matrix!

But wait...

- The matrix has at most **1 billion 1s** (one per token occurrence)
- The matrix is **extremely sparse** (99.8% zeros)

The Solution: The Inverted Index

Key insight: **Only record the 1s!**

Instead of storing a huge sparse matrix, we store:

- A **dictionary** of all terms
- For each term, a **postings list** of documents containing that term

Term		Postings List (Document IDs)
BRUTUS	$\longrightarrow$	1, 2, 4, 11, 31, 45, 173, 174
CAESAR	$\longrightarrow$	1, 2, 4, 5, 6, 16, 57, 132, ...
CALPURNIA	$\longrightarrow$	2, 31, 54, 101

Why "Inverted"?

It's called an **inverted** index because it inverts the natural relationship: instead of mapping documents $\rightarrow$ terms, it maps terms $\rightarrow$ documents.

Inverted Indices in Modern Search Engines

The inverted index is **still the fundamental data structure** used by modern search engines:

- **Apache Lucene** – The most widely-used open-source search library
- **Elasticsearch** – Built on Lucene, powers Wikipedia, GitHub, Stack Overflow
- **Apache Solr** – Enterprise search platform built on Lucene
- **Google, Bing** – Use highly optimised variants of inverted indices

Why Still Relevant?

Despite advances in neural search and vector embeddings, inverted indices remain essential because they provide extremely fast exact-match lookups and are highly efficient for Boolean and keyword queries. Modern systems often combine inverted indices with neural approaches.

Building an Inverted Index: Overview

The construction process has four main steps:

1. **Collect** the documents to be indexed

   $\boxed{\text{Friends, Romans, countrymen.}}$ $\boxed{\text{So let it be with Caesar}}$ ...

2. **Tokenise** the text into individual tokens

   $\boxed{\text{Friends}}$ $\boxed{\text{Romans}}$ $\boxed{\text{countrymen}}$ $\boxed{\text{So}}$ ...

3. **Normalise** tokens into indexing terms (linguistic preprocessing)

   $\boxed{\text{friend}}$ $\boxed{\text{roman}}$ $\boxed{\text{countryman}}$ $\boxed{\text{so}}$ ...

4. **Index** by creating the dictionary and postings lists

Note: We will cover tokenisation and normalisation in detail in the next session.

Building an Inverted Index: Detailed Example

Step 1: Start with two documents

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.

Step 2: Tokenise and normalise

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

Building an Inverted Index: Generate Postings

Step 3: Create (term, docID) pairs

From Doc 1:

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

From Doc 2:

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

Note: Terms can appear multiple times (e.g., "caesar" appears 3 times across both documents).

Building an Inverted Index: Sort and Create Postings Lists

Step 4: Sort by term, then merge duplicates

Sorted pairs:

ambitious	2
be	2
brutus	1
brutus	2
caesar	1
caesar	2
caesar	2
capitol	1
did	1
...	...

Final postings lists:

term	freq	postings
ambitious	1	$\rightarrow 2$
be	1	$\rightarrow 2$
brutus	2	$\rightarrow 1 \rightarrow 2$
capitol	1	$\rightarrow 1$
caesar	2	$\rightarrow 1 \rightarrow 2$
did	1	$\rightarrow 1$
hath	1	$\rightarrow 2$
...		

The **document frequency** tells us how many documents contain each term.

The Final Structure: Dictionary + Postings

Dictionary		Postings Lists
$\boxed{\text{BRUTUS}}$	$\longrightarrow$	$1 \rightarrow 2 \rightarrow 4 \rightarrow 11 \rightarrow 31 \rightarrow 45 \rightarrow 173 \rightarrow 174$
$\boxed{\text{CAESAR}}$	$\longrightarrow$	$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 16 \rightarrow 57 \rightarrow \ldots$
$\boxed{\text{CALPURNIA}}$	$\longrightarrow$	$2 \rightarrow 31 \rightarrow 54 \rightarrow 101$

- **Dictionary**: Stored in memory for fast lookup (often as a hash table or tree)
- **Postings**: Stored on disk, sorted by document ID for efficient intersection

Part 3

Processing Boolean Queries

Processing a Simple Conjunctive Query

Query: `BRUTUS AND CALPURNIA`

Algorithm:

1. Locate BRUTUS in the dictionary
2. Retrieve its postings list: [1, 2, 4, 11, 31, 45, 173, 174]
3. Locate CALPURNIA in the dictionary
4. Retrieve its postings list: [2, 31, 54, 101]
5. **Intersect** the two postings lists
6. Return the intersection to the user

BRUTUS	$\longrightarrow$	1, 2, 4, 11, 31, 45, 173, 174
CALPURNIA	$\longrightarrow$	2, 31, 54, 101
Intersection	$\Longrightarrow$	2, 31

The Intersection Algorithm (Merge Algorithm)

Since postings lists are **sorted by document ID**, we can intersect efficiently:

Algorithm 1 INTERSECT(p_1, p_2)

$answer \leftarrow \langle\rangle$ ▷ Empty list
while $p_1 \neq$ NIL **and** $p_2 \neq$ NIL **do**
    if $docID(p_1) = docID(p_2)$ **then**
        ADD($answer$, $docID(p_1)$) ▷ Match found!
        $p_1 \leftarrow next(p_1)$; $p_2 \leftarrow next(p_2)$
    else if $docID(p_1) < docID(p_2)$ **then**
        $p_1 \leftarrow next(p_1)$ ▷ Advance smaller pointer
    else
        $p_2 \leftarrow next(p_2)$
    end if
end while
return *answer*

Time complexity: $O(|p_1| + |p_2|)$ – linear in the total length of both lists.

Intersection: Visual Walkthrough

Intersecting BRUTUS [1, 2, 4, 11, 31] and CALPURNIA [2, 31, 54]:

Step	BRUTUS ptr	CALPURNIA ptr	Action
1	**1**	**2**	$1 < 2$, advance BRUTUS
2	**2**	**2**	Match! Add 2, advance both
3	**4**	**31**	$4 < 31$, advance BRUTUS
4	**11**	**31**	$11 < 31$, advance BRUTUS
5	**31**	**31**	Match! Add 31, advance both
6	–	**54**	BRUTUS exhausted, stop

Result: [2, 31]

Key insight: We never backtrack – each pointer only moves forward, giving us linear time.

Exercise: Process a Complex Boolean Query

Given these postings lists:

FRANCE	$\longrightarrow$	1, 2, 3, 4, 5, 7, 8, 9, 11, 12, 13, 14, 15
PARIS	$\longrightarrow$	2, 6, 10, 12, 14
LEAR	$\longrightarrow$	12, 15

Compute the result for: `(PARIS AND NOT FRANCE) OR LEAR`

Hint: Break it down step by step:

1. First compute PARIS AND NOT FRANCE
2. Then OR the result with LEAR

Exercise Solution

Query: (PARIS AND NOT FRANCE) OR LEAR

Step 1: Compute PARIS AND NOT FRANCE

- PARIS documents: 2, 6, 10, 12, 14
- FRANCE documents: 1, 2, 3, 4, 5, 7, 8, 9, 11, 12, 13, 14, 15
- NOT FRANCE = documents *not* in the FRANCE list
- PARIS AND NOT FRANCE = PARIS documents that are *not* in FRANCE
- Remove 2, 12, 14 from PARIS (they appear in FRANCE)
- Result: **6, 10**

Step 2: Compute (6, 10) OR LEAR (12, 15)

- OR = union of both sets
- Result: **6, 10, 12, 15**

Final answer: Documents 6, 10, 12, 15
These are documents that either (contain PARIS but not FRANCE) or (contain LEAR).

Part 4

Query Optimisation

Query Optimisation for Conjunctive Queries

Question: For a query with multiple AND terms, what order should we process them?

Example: `BRUTUS AND CALPURNIA AND CAESAR`

Term	Frequency	Postings
BRUTUS	8	1, 2, 4, 11, 31, 45, 173, 174
CALPURNIA	4	2, 31, 54, 101
CAESAR	2	5, 31

Optimisation strategy: **Process in order of increasing frequency**

- Start with the **shortest** postings list
- Each intersection can only **reduce** the result size
- Smaller intermediate results = fewer comparisons

Why Processing Order Matters

Query: BRUTUS AND CALPURNIA AND CAESAR

Optimised order: CAESAR (2) $\rightarrow$ CALPURNIA (4) $\rightarrow$ BRUTUS (8)

1. Start with CAESAR: [5, 31] *(2 documents)*
2. Intersect with CALPURNIA [2, 31, 54, 101]: **[31]** *(1 document)*
3. Intersect with BRUTUS [1, 2, 4, 11, 31, ...]: **[31]** *(1 document)*

Total comparisons: $\approx 2 + 4 + 8 = 14$

Unoptimised order: BRUTUS $\rightarrow$ CALPURNIA $\rightarrow$ CAESAR

- Would require more comparisons with larger intermediate results

Key Insight

Processing short lists first minimises the size of intermediate results, reducing overall work.

Optimised Intersection Algorithm

Algorithm 2 INTERSECT($\{t_1, \ldots, t_n\}$)

```
terms ← SORTBYINCREASINGFREQUENCY({t1, ..., tn})
result ← postings(first(terms))
terms ← rest(terms)
while terms ≠ NIL and result ≠ NIL do
    result ← INTERSECT(result, postings(first(terms)))
    terms ← rest(terms)
end while
return result
```

Note: If *result* becomes empty at any point, we can stop early – no documents match!

Part 5

Advantages and Limitations

Advantages of Boolean Retrieval

- **Predictable and transparent:** Results are easy to understand – a document either matches or it doesn't
- **Precise control:** Users know exactly what they will get
- **Efficient:** Can quickly eliminate documents that don't match
- **Flexible operands:** Can combine with metadata (date, document type, author)
- **Professional preference:** Many domain experts (lawyers, medical researchers, patent searchers) prefer Boolean queries for high-stakes searches
- **Reproducible:** Same query always returns the same results

Still in Use Today

Many professional systems (Westlaw, LexisNexis, PubMed) support Boolean queries. About 80% of legal searches on Westlaw use Boolean operators.

Limitations of Boolean Retrieval

- **No ranking:** All matching documents are treated as equally relevant
- **Feast or famine:** Queries often return too many results (OR) or too few results (AND)
- **Difficult to formulate:** Writing effective Boolean queries requires skill and practice
- **All-or-nothing matching:** A document with 99% of query terms is treated the same as one with 0%
- **No partial matches:** Cannot find documents that are "close" to the query
- **Users overestimate recall:** The precision of results can make users think they've found everything relevant

The Alternative: Ranked Retrieval

Modern search engines use **ranked retrieval** models (e.g., TF-IDF, BM25) that score and rank documents by relevance. We will cover these in later sessions.

Boolean vs Ranked Retrieval: A Comparison

Boolean Retrieval	**Ranked Retrieval**
Binary matching (yes/no)	Relevance scores (0.0–1.0)
No ranking of results	Results ranked by relevance
Precise, predictable	Fuzzy, probabilistic
Complex query syntax	Natural language queries
Expert users	General users
High precision possible	High recall easier
Exact match required	Partial matches allowed
Professional/legal search	Web search, general IR

In Practice

Many modern systems combine both approaches: Boolean operators for precision when needed, with ranking within the matched set.

Part 6

Summary

Summary

Key concepts covered today:

1. **Information Retrieval** is finding relevant documents from large collections to satisfy an information need
2. The **term-document matrix** represents document-term relationships but doesn't scale
3. The **inverted index** efficiently stores only the 1s (term $\rightarrow$ document mappings) and is the foundation of modern search engines
4. **Boolean queries** use AND, OR, NOT to combine terms; results are unranked
5. The **merge algorithm** intersects sorted postings lists in linear time
6. **Query optimisation**: process terms in order of increasing document frequency
7. Boolean retrieval offers **precision and control** but lacks **ranking and partial matching**

References and Further Reading

Required reading:

- Manning, C.D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. Chapters 1–2.
  Free online: `https://nlp.stanford.edu/IR-book/`

Additional resources:

- Manning's lecture slides: `https://nlp.stanford.edu/~manning/talks/colm-slides.pdf`
- Elasticsearch documentation on inverted indices: `elastic.co/guide`
- Apache Lucene: `lucene.apache.org`

Historical note:

- The Boolean retrieval model was formalised in the 1970s and dominated commercial IR until the mid-1990s when ranked retrieval (pioneered by web search engines) became prevalent.

Questions?

Prof. Georgina Cosma

g.cosma@lboro.ac.uk