Lecture 3: Phrasal queries and wildcards

Trevor Cohn (tcohn@unimelb.edu.au)

COMP90042, 2015, Semester 1
What we’ll learn today

- Building on the boolean index and query mechanism to support multi-word queries
- Support for adjacent and ‘near to’ operators
- Concept of a positional index
- Handling wildcards in queries
Phrase queries

- Some search needs are tricky to express as a simple boolean query
- E.g., the Geelong Cats football team
- Searching for Geelong AND Cats is likely to match some documents about pets, not football
- Inverted index storing (term, documents) from last lecture won't suffice

Need special support for phrase queries
Biword indices

Simplest solution:

- treat every pair of words in the document as a term
- I have a dream becomes I–have have–a a–dream
- biwords now form our dictionary items
- can easily query for two word phrases
Biword indices

How to query for longer phrases?

- express as conjunction over biwords
  
  North Melbourne Kangaroos →
  
  North Melbourne AND Melbourne Kangaroos

- can process this query in the usual way
- finally need to filter results to remove spurious matches

Generally an inefficient and inflexible solution

- index gets large, and doesn’t scale beyond bi-words
- problem of filtering out false positives
- typically biword indices are not used on their own
Positional indices

Don’t just index the document for each term, but also the position in the document.

1: Imagine no possessions
2: I wonder if you can
3: No need for greed or hunger
4: A brotherhood of man
5: Imagine all the people
6: Sharing all the world

Each entry includes term and document frequency and

- linked list with nodes containing document id and
- a list of offsets into the document
Querying a positional index

Take the phrase query **imagine all**. Matching documents must satisfy the condition

- there exists a location \( l \), such that \( w_l = \text{imagine} \land w_{l+1} = \text{all} \)
- i.e., **imagine** precedes **all** by two words, somewhere in the document

Using the index, we can

- find candidate documents as before
- followed by checking for correctly offset location pairs

\[
\text{imagine} \rightarrow \text{df}=45; \quad \langle 1:3 \rangle, \langle 5:18 \rangle, \langle 22:3,18,53 \rangle, \langle 34:1,5 \rangle, \ldots
\]
\[
\text{all} \rightarrow \text{df}=61; \quad \langle 3:1 \rangle, \langle 22:1,13,54 \rangle, \langle 34:10 \rangle, \ldots
\]
Merging positional posting lists

Last step requires checking for correctly offset location pairs

- for each candidate document containing **imagine** AND **all**
- traverse the *sorted* list of locations, \( l_1, l_2 \)
- maintain two pointers \( p_1, p_2 \) into the two lists
- increment \( p_1 \) when \( l_1[p_1] < l_2[p_2] − k \), where \( k \) is the offset (\( k = 1 \) here)
- increment \( p_2 \) when \( l_1[p_1] > l_2[p_2] − k \)
- otherwise if both equal, record a match and increment \( p_1, p_2 \)

**imagine all example**

22: 3, 18, 53
22: 1, 13, 54
Merging positional posting lists

Last step requires checking for correctly offset location pairs

- for each candidate document containing **imagine** AND **all**
- traverse the sorted list of locations, $l_1, l_2$
- maintain two pointers $p_1, p_2$ into the two lists
- increment $p_1$ when $l_1[p_1] < l_2[p_2] - k$, where $k$ is the offset ($k = 1$ here)
- increment $p_2$ when $l_1[p_1] > l_2[p_2] - k$
- otherwise if both equal, record a match and increment $p_1, p_2$

**imagine all example**

22: 3, 18, 53
22: 1, 13, 54
Merging positional posting lists

Last step requires checking for correctly offset location pairs
  ▶ for each candidate document containing imagine AND all
  ▶ traverse the sorted list of locations, \( l_1, l_2 \)
  ▶ maintain two pointers \( p_1, p_2 \) into the two lists
  ▶ increment \( p_1 \) when \( l_1[p_1] < l_2[p_2] - k \), where \( k \) is the offset (\( k = 1 \) here)
  ▶ increment \( p_2 \) when \( l_1[p_1] > l_2[p_2] - k \)
  ▶ otherwise if both equal, record a match and increment \( p_1, p_2 \)

**imagine all example**

22: 3, 18, 53
22: 1, 13, 54
Merging positional posting lists

Last step requires checking for correctly offset location pairs

- for each candidate document containing **imagine** AND **all**
- traverse the sorted list of locations, $l_1, l_2$
- maintain two pointers $p_1, p_2$ into the two lists
- increment $p_1$ when $l_1[p_1] < l_2[p_2] - k$, where $k$ is the offset ($k = 1$ here)
- increment $p_2$ when $l_1[p_1] > l_2[p_2] - k$
- otherwise if both equal, record a match and increment $p_1, p_2$

**imagine all** example

22: 3, 18, 53
22: 1, 13, 54
Merging positional posting lists

Last step requires checking for correctly offset location pairs

- for each candidate document containing **imagine** AND **all**
- traverse the *sorted* list of locations, $l_1, l_2$
- maintain two pointers $p_1, p_2$ into the two lists
- increment $p_1$ when $l_1[p_1] < l_2[p_2] - k$, where $k$ is the offset ($k = 1$ here)
- increment $p_2$ when $l_1[p_1] > l_2[p_2] - k$
- otherwise if both equal, record a match and increment $p_1, p_2$

**imagine all** example

- 22: 3, 18, 53
- 22: 1, 13, 54

$53 + 1 = 54 \rightarrow$ record a match for doc 22
Longer phrases

How to cope with phrases of length 3 or greater?
- will have posting lists for each term in the query
- and can easily compute offsets between any pair of terms
- as before, work from smallest \( df \) to largest (why?)
- when combining posting lists, consider appropriate offset based on order in query

Can also easily support “NEAR-TO” queries, i.e., terms in the query might need to appear \( \leq k \) tokens apart in a document.
Complexity

Positional indices store a record of every token
- rather than every type occurring in a document
- long documents may see many tokens repeated
- time complexity $O(|W|)$, word tokens, not $O(D)$, documents
- consider looking up “The who”

Mixed biword and positional indices
Index the pathological cases using explicit biword index, and easier cases in positional index. Get the benefits of both techniques, without the nasty time and space complexity problems.
Proximal queries

Legal world full of carefully designed complex queries, e.g.,

*What is the statute of limitations in cases involving the federal tort claims act?*

*LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM*

Westlaw example, IIR Chapter 1.

- support for range operators straightforward with positional indices
- but not possible with biword indexes
- storage cost still high, but can be compressed well (gaps are small between repeats of a word in a document)
What if we don’t know how to spell a crucial search term?

Some words have many accepted spellings (particularly foreign names)
Gaddafi, Kadafi, Qaddafi, Gathafi, Kadafi or Gadafy

Or just are a nightmare to spell, e.g., Eyjafjallajökull, the volcano which erupted in Iceland disrupting air travel

Might want to query on eyja*, e*kull or E*yall*kull

To support these kinds of queries, we need to be able to search over sub-word units
Wildcard queries

Generally useful tool to allow incomplete query terms

- leading wildcard: *addafi
- trailing wildcard: Gadd*
- internal wildcard: Gad*fi
- several wildcards: *add*fi

Must map these patterns to term/s in our vocabulary, then we can query the index using the vocabulary terms.
Tree based approach

Store tree, e.g., B+ tree, over the vocabulary

- using a sort order over the vocabulary entries
- can find all vocabulary items matching trailing wildcards
  all child nodes will share the common prefix
  e.g., **Gadd*** → **Gadd**

- what about leading wildcards?
  easy, just use a tree in reverse order, to allow suffix matching
  e.g., **addafi** → **ifadda**

General queries

Not quite so easy

- match prefix and suffix and intersect to obtain candidates
- filter these to find the entries which match the pattern
Permuterm index

More elegant method is to store an index over the vocabulary items

- append a sentinel symbol, $, to each term
- **rotate** the term by a character at a time
- store in a sorted tree the mappings between the rotated versions and the term

**Example:** ‘rotate’

rotate$  otate$r  tate$ro  ate$rot  →  rotate
te$rota  e$rotat  $rotate

Consider searching for **ro***te**:

- add the sentinel, **ro***te$**
- rotate the wildcard to the end, **te$ro***
- now lookup in permutum index, **te$ro**
Permuterm index lookup

Matches for \texttt{te$ro} prefix in permuterm index may uncover

- \texttt{te$rota} $\rightarrow$ rotate
- \texttt{te$ro} $\rightarrow$ rote
- \texttt{te$rou} $\rightarrow$ route
- \texttt{te$roulet} $\rightarrow$ roulette

Permuterm index supports

- single leading, trailing or internal wildcard
- but to use several wildcards still requires final filtering step
- once we have found the terms, using inverted index to find matching documents and combine the results
Complexity

Permuterm is a very convenient datastructure

- but has significant space requirement
- especially for vocabularies with long terms
- expands dictionary storage by around 10×
- simple forward and reverse trees are more compact
K-gram index

Break terms into *character sequences*
- index each sequence of $k$ characters
  - *rote* $\rightarrow$ $\text{ro}$, $\text{rot}$, $\text{ote}$, $\text{te}$ (where $k = 3$)
- store a mapping between each $k$-gram and the term

Querying follows similar pattern
- lookup partial fragments of the query term
  - *ro*te $\rightarrow$ $\text{ro}$, $\text{te}$
- merge the list of results both each fragment
- filter the candidates to remove false matches
Looking back and forward

Back

- Phrases as query terms
- Indexing and querying with biword and positional indices
- Dealing with wildcard query terms: using trees, a permuterm index or a k-gram index
Looking back and forward

Forward

- In the next lecture, we will look at the *vector space model* of information retrieval
- Introduce real valued weighting of terms, and ranked result sets
Further reading

- Section 2.4, “Positional postings and phrase queries” of Manning, Raghavan, and Schutze, *Introduction to Information Retrieval*
- Chapter 3 (§3.1, §3.2), “Dictionaries and tolerant retrieval” of Manning, Raghavan, and Schutze, *Introduction to Information Retrieval*