2. Boolean Retrieval and Term Indexing

Prof. Dr. Goran Glavaš

(Based on slides from Laura Dietz and Jan Šnajder)

Data and Web Science Group
Fakultät für Wirtschaftsinformatik und Wirtschaftsmathematik
Universität Mannheim

After this lecture, you'll...

- Know what Boolean retrieval is and what Boolean queries look like
- Know what inverted index is and how to use it to answer Boolean queries
- Understand skip lists and how they may make Boolean retrieval more efficient
- Comprehend phrase queries and understand biword indexes
- Understand the structure of the positional index and how it can be used to support phrase queries as well as proximity queries

Outline

- Recap of Lecture #1
- Basic Boolean retrieval
- Inverted index
- Skip lists and faster merges
- Positional index, phrase and proximity queries

Recap of the previous lecture

- Basic notions of information retrieval
  - **Q:** What is information retrieval and what are the elements of the retrieval process?
  - **Q:** What is an information need?
  - **Q:** What is relevance?
- Text representations and preprocessing
  - **Q:** Differences between unstructured and weakly-structured representations
  - **Q:** What are the common text preprocessing steps?
  - **Q:** Explain what tokenization, lemmatization, and stemming are?
  - **Q:** What are stopwords and why do we remove them?
- General information retrieval model
  - **Q:** What are the three components of every information retrieval system?
  - **Q:** What are index terms?

Recap of the previous lecture

- **Index terms** are all terms in the collection (i.e., the vocabulary)
  - Except those we ignore in preprocessing (like stopwords)
  - The set of all index terms: $K = \{k_1, k_2, ..., k_t\}$
  - Each term k_i is, for each document d_j, assigned a weight w_{ij}
  - The weight of the index terms not appearing in the document is 0
- Document d_j is represented by term vector $[w_{1j}, w_{2j}, ..., w_{tj}]$ where t is the number of index terms
- Let g be the function that computes the weights, i.e., $w_{ij} = g(k_i, d_j)$
- Different choices for the weight-computation function g and the ranking function r define different IR models
- **Today we examine what the weighting function g and ranking function r look like for Boolean retrieval**

Outline

- Recap of Lecture #1
- Basic Boolean retrieval
- Inverted index
- Skip lists and faster merges
- Positional index, phrase and proximity queries

Boolean retrieval: case study

- **Case study:** you're an SciFi and Fantasy fan who is studying similarities and differences between the Star Trek, Lord of the Rings, and Harry Potter
- Information need: Contexts in which there are mentions of "aliens" AND "swords" but NOT "wizards"
- **Q:** How would you do this?
- **Attempt #1** (regular Joe's approach): Let's (1) **grep** all documents for "aliens" and "swords" and then (2) take out the lines containing "wizards"
  - **Q:** Why is this not a good approach?
  - **A:** Slow for large corpora
  - **A:** Does not support other types of information needs, e.g., find documents where "aliens" appears **near** "swords"

Boolean retrieval

- Boolean retrieval model is arguably the simplest IR model
- Queries are Boolean expressions
  - E.g., “aliens” AND “swords” AND NOT “wizards”
- The search engine returns all documents from the collection that satisfy the Boolean expression

Boolean retrieval

- Let's analyze Boolean IR model in terms of three common IR components

1. Query representation
   - Query *q* is given as a propositional logic formula over index terms
   - Index terms are connected via Boolean operators (∧, ∨) and can be negated (¬)
   - Being a propositional logic formula, each query can be transformed into disjunctive normal form (DNF)
     - $q = c_1 \lor c_2 \lor \cdots \lor c_n$ – where c_l is the l-th conjunctive component of *q*'s DNF
     - E.g., $c_l = t_{l1} \land \neg t_{l2} \land \ldots \land t_{lk}$

Boolean retrieval

- Let's analyze Boolean IR model in terms of three common IR components

2. Document representation
   - Each document d in the collection is represented as a bag of words
     - Strictly speaking, it's a set of words, not a bag (i.e., not a multiset)
   - The frequency of terms is irrelevant, only whether the term appears in the document
     - Thus, term weights are all binary – $w_{ij} \in \{0, 1\}$
     - $w_{ij} = 1$ if document d_j contains the index term t_i , $w_{ij} = 0$ otherwise

Boolean retrieval

- Let's analyze Boolean IR model in terms of three common IR components

3. Relevance of the document for the query
   - The document is relevant for the query if it satisfies the propositional logic formula of the query
   - As queries are in DNF this means the document must satisfy at least one of the conjunctive components c_l of the query *q*

$$\text{relevance}(d_j, q) = \begin{cases} 1, \text{ if } \exists c_l \mid \forall ti \in \text{terms}(c_l), w_{ij} = 1 \\ 0, \text{ otherwise.} \end{cases}$$

Boolean retrieval

- Does Google use Boolean retrieval?
  - Google's **default interpretation** of the query "Frodo gave Sam the sword" is "Frodo" AND "gave" AND "Sam" AND "sword"
- A retrieved document might not contain some of the query terms if
  - The full Boolean expression generates very few relevant documents
  - The result contains some morphological variation or a synonym of the term
  - If your query is long, Google discards less relevant terms
- **Boolean retrieval and results ranking**
  - Boolean retrieval returns matching documents in no particular order
  - Well-designed search engines need to rank the relevant results

Boolean retrieval – example

- Let us have the following set of index terms
  - K = { “Frodo”, “Sam”, “blue”, “sword”, “orc”, “Mordor”}
- Let us have the following collection of documents
  - d1: „Frodo stabbed the orc with the red sword”
  - d2: „Frodo and Sam used the blue lamp to locate orcs”
  - d3: „Sam killed many orcs in Mordor with the blue sword”
- Which documents are relevant for the following queries?
  - q1: („Frodo” AND „orc” AND „sword”) OR („Frodo” AND „blue”)
    - {d1, d2}
  - q2: („Sam” AND „blue” AND NOT „Frodo”) OR („Sam” AND „orc” AND „Mordor”)
    - {d3}

Boolean retrieval – incidence matrix

- **Attempt #2**: use the incidence matrix to answer queries like
  - „Sam” AND „blue” AND NOT „Frodo”
- Term-document incidence matrix

Term	d1: „Frodo stabbed the orc with the red sword”	d2: Frodo and Sam used the blue lamp to locate orcs	d3: Sam killed many orcs in Mordor with the blue sword
Frodo	True (1)	True (1)	False (0)
Sam	False (0)	True (1)	True (1)
blue	False (0)	True (1)	True (1)
sword	True (1)	False (0)	True (1)
orc	True (1)	True (1)	True (1)
Mordor	False (0)	False (0)	True (1)

Boolean retrieval – incidence matrix

- Query: „Sam” AND „blue” AND NOT „Frodo”

Term	d1: Frodo stabbed the orc with the red sword	d2: Frodo and Sam used the blue lamp to locate orcs	d3: Sam killed many orcs in Mordor with the blue sword
Frodo	True (1)	True (1)	False (0)
Sam	False (0)	True (1)	True (1)
blue	False (0)	True (1)	True (1)

- „Sam”: d1 – False; d2 – True; d3 – True -> [0, 1, 1]
- „blue”: d1 – False; d2 – True; d3 – True -> [0, 1, 1]
- „Frodo”: d1 –True; d2 – True; d3 – False -> [1, 1, 0]

Boolean retrieval – incidence matrix

- Incidence matrix – 0/1 vector for each index term
- To answer the query „Sam” AND „blue” AND NOT „Frodo” we just need to
  1. Take the vectors for terms „Sam” and „blue”

     „Sam” -> [0, 1, 1]; „blue” -> [0, 1, 1]
  2. Invert the vector for the term „Frodo”

     „Frodo” -> [1, 1, 0]; thus NOT „Frodo” -> [0, 0, 1]
  3. Perform a bitwise conjunction (bitwise AND) on these three vectors

     [0, 1, 1] AND [0, 1, 1] AND [0, 0, 1] -> [0, 0, 1]
     - d3 is the only relevant document for the query

Boolean retrieval – incidence matrices

- Incidence matrix is a good solution for small collections
- However, real world collections can be very large
  - E.g., N = 1 million of documents, each around 1000 words
  - E.g., 500 thousand index terms
  - The incidence matrix size is 500K x 1M -> 0.5 **trillion** elements (0's and 1's)
    - But only 1000 x 1M = 1 billion 1's – incidence matrix is **very sparse**

- **Q:** What would be a better solution?
  - A representation that would remedy for the sparseness of incidence matrices
- **A:** We can only store positions of 1's
    - We know that the rest are 0's

Outline

- Recap of Lecture #1
- Basic Boolean retrieval
- Inverted index
- Skip lists and faster merges
- Positional index, phrase and proximity queries

Inverted index

- **Inverted index** is a data structure for computationally efficient retrieval
- Inverted index contains a list of references to documents for all index terms
  - For each term *t* we store the list of all documents that contain *t*
  - Documents are represented with their identifier numbers (ordinal, starting from 1)

„Frodo" -> [1, 2, 7, 19, 174, 210, 331, 2046]
„Sam" -> [2, 3, 4, 7, 11, 94, 210, 1137]
„blue" -> [2, 3, 24, 2001]

- The list of documents that contains a term is called a **posting list** (or just **posting**)
- In memory, postings are implemented as either
  - Linked lists or
  - Variable length arrays (**Q:** why not fixed length arrays?)

- **Q:** Postings are always sorted. Why?

Inverted index – query processing

- The inverted index is an efficient structure for storing term incidences
  - Requires much less storage than full incidence matrix
- But we need to couple the storage with an efficient algorithm(s) for finding relevant documents for queries
- Consider the query „Sam” AND „Frodo”
  1. Retrieve the posting list for the term „Sam”
     „Sam” -> [2, 3, 4, 7, 11, 94, 210, 1137]
  2. Retrieve the posting list for the term „Frodo”
     „Frodo” -> [1, 2, 6, 7, 19, 174, 210, 331, 2046]
  3. Find the intersection between the two posting lists (i.e., „merge” the postings)
     [2, 3, 4, 7, 11, 94, 210, 1137] ∩ [1, 2, 7, 19, 174, 210, 331, 2046] -> [2, 7, 210]

The merge

- If posting lists are sorted, the time of the merge is linear in the total number of posting entries
  - The „merge” is performed by simoultaneously walking through the two postings
  - If the first posting has x elements and second posting has y elements, the (worst-case) time complexity of the merge is O(x+y)
- If the posting lists would not be sorted, the merge complexity would be quadratic
  - For each element in the first list, we must, in the worst case go through the whole second list
  - Complexity is O(xy)

The merge algorithm

- The following is the pseudocode of the algorithm for merging (sorted) postings

```
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
```

The merge algorithm

- The given algorithm works for merges for queries of type t_1 AND t_2
- What about other query constructs?
  - t_1 AND NOT t_2
  - t_1 OR t_2
  - **Q:** Can we perform the merge for these constructs in linear time $O(x+y)$?

- What about an arbitrary Boolean formula?
  - („Sam” OR „Frodo”) AND NOT („orc” OR „Mordor”)
  - **Q:** Can we always merge in linear time?
    - Linear in what?
  - **Q:** Can we maybe do better than linear complexity?

Query optimization

- What is the best order for processing the query?
- Consider a query which is a conjunction of *n* terms
  - We get the postings list for each of the *n* terms and merge them together
  - Merge is a binary operator
    - **Q:** Does it matter in which order we do the merges?

Query: „Frodo” AND „Sam” AND „blue”

Postings:

„Frodo” -> [1, 2, 7, 19, 174, 210, 331, 2046]

„Sam” -> [2, 3, 4, 7, 11, 94, 210, 1137]

„blue” -> [2, 3, 24, 2001]

Query optimization

- The set of binary merges is going to be executed fastest if we start from the shortest postings and perform merges in increasing order of posting length

  Query: „Frodo” AND „Sam” AND „blue”

  Postings:

  „Frodo” -> [1, 2, 7, 19, 174, 210, 2046] (length = 7)

  „Sam” -> [2, 3, 4, 7, 11, 94, 210, 1137] (length = 8)

  „blue” -> [2, 3, 24, 2001] (length = 4)

  Merges (count comparisons):

  („Frodo” AND „Sam”) AND „blue” -> complexity: 11 + 5 = 16 comparisons

  „Frodo” AND („Sam” AND „blue”) -> complexity: 9 + 3 = 12 comparisons

  („Frodo” AND „blue”) AND „Sam” -> complexity: 8 + 1= **9** comparisons

Merge algorithm for conjunctive queries

- The following is the algorithm for efficient merging of conjunctive queries with multiple terms

```
INTERSECT(⟨t_1, ..., t_n⟩)
terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
result ← postings(first(terms))
terms ← rest(terms)
while terms ≠ NIL and result ≠ NIL
do result ← INTERSECT(result, postings(first(terms)))
    terms ← rest(terms)
return result
```

General Boolean query optimization

- Each Boolean query can be transformed to a disjunctive normal form (DNF)

  (t_{11} AND ... AND t_{1n}) OR

  (t_{21} AND ... AND t_{2n}) OR

  ... OR

  (t_{k1} AND ... AND t_{kn})

- Algorithm:
  1. For each conjunction (t_{j1} AND ... AND t_{jn}), perform the optimized conjunction merge algorithm from the previous slide and obtain conjunction postings
  2. For each pair of conjunction postings estimate the cost of performing an OR operation as the sum of lengths of these two conjunction postings
  3. Process in increasing order of OR sizes

General Boolean query optimization

- Query:
  („Frodo” AND „orc” AND „sword”) OR („Frodo” AND „blue”) OR („orc” AND „blue”)
- Term postings:
  „Frodo” -> [1, 2, 7, 19, 174, 210, 331, 2046] (length = 8)
  „orc” -> [2, 3, 7, 11, 94, 210] (length = 6)
  „sword” -> [2, 7, 24, 2001] (length = 4)
  „blue” -> [8, 19, 94] (length = 3)
- Conjunction postings:
  („Frodo” AND „orc” AND „sword”) -> [2,7]
  („Frodo” AND „blue”) -> [19]
  („orc” AND „blue”) -> [94]
- OR merges:
  - First: („Frodo” AND „blue”) OR („orc” AND „blue”) -> [19, 94]
  - Second: result first OR („Frodo” AND „orc” AND „sword”) -> [2,7,19,94]
- **Exercise:** compute the real complexity of processing this query

Outline

- Recap of Lecture #1
- Basic Boolean retrieval
- Inverted index
- Skip pointers and faster merges
- Positional index, phrase and proximity queries

Basic merge and linear complexity

- If posting lists are sorted, the time of the merge is linear in the total number of posting entries
  - The „merge" is performed by simoultaneously walking through the two postings
  - If the first posting has x elements and second posting has y elements, the computational complexity of the merge is $O(x+y)$

- **Q:** Can we do better than linear complexity?
  - Yes, if we are dealing with read-only indexes!
- **Q:** How?
  - By enriching postings with additional pointers
  - These pointers are called **skip pointers**

Merge with skip pointers

- Consider the merge between the following postings:

  „Sam” -> [1, 2, 7, 174, 210, 331, 2046]

  „Frodo” -> [2, 3, 4, 7, 21, 29, 38, 91, 101, 122, 134, 171, 1137]

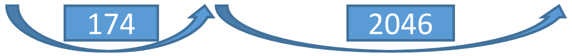
  - With the basic merge algorithm we need to compare all red numbers (21, 29, ..., 171) from the „Frodo” posting list with 174 from the ”Sam” posting list

  - What if we could skip directly from 21 to 171 in the „Frodo” posting list?
    - We would save many comparisons
    - This would not affect the merge result at all

  - The idea is to skip parts of posting lists that lead to empty results
    - Central question: where do we place the skip pointers?

Augmenting postings with skip pointers

„Sam" -> [1, 2, 7, 174, 210, 331, 2046]



„Frodo" -> [2, 3, 4, 7, 21, 29, 38, 91, 101, 122, 134, 171, 1137]

- Suppose we went through the posting lists until we processed 7 in both lists
  - I.e., we found a match at 7 and added it to the merge result
- We then have element 174 in the „Sam" list and 21 in the „Frodo" list
  - Instead of going linearly through the lists, we can try to jump via skip pointers
  - Skip successor of 21 in the „Frodo" list is 134, which is still smaller than the current element 174 in the „Sam" list
  - This means that we can safely, i.e., without missing any matches, skip to 134 in the „Frodo" list
  - **Q:** What if the skip from 21 in „Frodo" list was larger than 174?

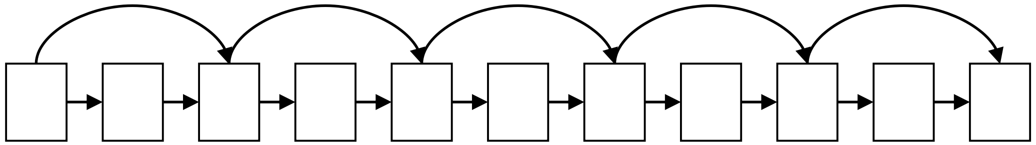
Merge with skip pointers

- The following is the pseudocode of the merge algorithm on lists augmented with skip pointers

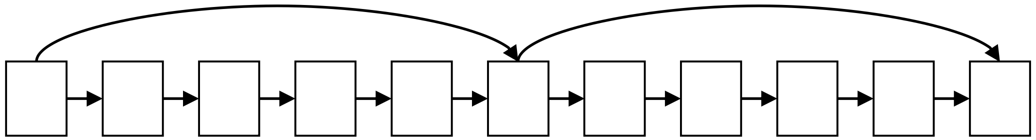
```
INTERSECTWITHSKIPS(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
       then ADD(answer, docID(p1))
            p1 ← next(p1)
            p2 ← next(p2)
       else if docID(p1) < docID(p2)
              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
  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

Merge with skip pointers

- Central questions:
  - How frequent should the skips be?
  - Where to place the skip pointers?
- There is a tradeoff:
  - Option 1: More skips
    - Shorter skip spans
    - More likely to skip
    - But a lot of skip position comparisons (so the complexity isn't very sublinear)
    - More data to store (larger index)



  - Option 2: Fewer skips
    - Longer skip spans
    - Less likely to skip (fewer successful skips)
    - But fewer pointer comparisons
    - Less data to store (smaller index)



Placing skips

- Simple heuristic that works well in practice
  - For postings of length L, use $\sqrt{L}$ evenly-spaced skip pointers
- Easy to implement if the index is read-only
  - The lengths of the posting lists do not change
- Maintenance required when index is updated
  - Not recommended for indexes that are being updated frequently

- This heuristic ignores the distribution of terms over indexed documents
  - **Q:** How to improve the placement of skip pointers taking into account distributions?

Outline

- Recap of Lecture #1
- Basic Boolean retrieval
- Inverted index
- Skip pointers and faster merges
- Positional index, phrase and proximity queries

Phrase queries

- Some queries contain phrases which are not meant to be split into terms, e.g.,
  - Named entities („Berkeley university", „San Francisco")
  - Collocations and idioms („fast food", „hot potato")

- We don't want documents containing these words separately to be considered relevant
  - E.g., „The lady thought the tea was too hot and the potatoes were not well-cooked" should not be relevant for the query „hot potato"

- For handling phrase queries, it is no longer enough to store only posting lists for individual terms

Phrase queries: biword index

- Besides individual terms, we could additionally index pairs of consecutive terms
- For example, the text „Frodo stabbed the orc with the red sword” would generate the following biwords (with stopword removal in place):
  - „Frodo stabbed”, „stabbed orc”, „orc red”, „red sword”
- Each of the biwords becomes an index term
- The query is also transformed into biwords for lookup and merging
  - Query: „Sam stabbed the orc” -> „Sam stabbed”, „stabbed orc”
- The query processing – posting lookup and merging – is then performed in exactly the same way as for single-token terms

Phrase queries: extended biwords

- Indexing all biwords often doesn’t make much sense
  - The vast majority of biwords are not real-world „concepts” users would look for
    - Not named entities, collocations, idioms, etc.
  - The number of biwords in the document collection is much larger than number of terms (combinatorial explosion)
- Extension: Let’s index only biwords that have a higher probability of being concepts that users might search for
  - Biwords that satisfy certain part-of-speech patterns (e.g., noun phrases)
  - E.g., all sequences of POS-tags of the form NX*N where N is a noun and X is a preposition or article
    - E.g., „catcher in the rye” has the POS-signature N X X N
    - Annotate with parts-of-speech and extract extended biwords (NX*N)
      - Look up in the index: „catcher rye”

Issues with biword indexes

1. Indexing biwords can lead to **false positives**
   - Because we index many biwords that are not concepts
   - E.g., document „Pequenos Angeles, United States won the competition" will be relevant for the query „Los Angeles, United States"
     - Both representations will contain the non-conceptual biword „Angeles United"
2. Large index
   - Combinatorial explosion
     - Storing all biwords creates very large index
     - Storing tri-words already infeasible for reasonably large collections

- Alternative approach
  - Index only small subset of most frequent or most relevant biwords (or larger n-words), along with unigram terms

Phrase queries: positional index

- A better alternative to biword (or generally n-word) indexes are **positional indexes**
- Positional index is an extended index
  - For each document that contains the index term ***t*** we store positions of all tokens of term ***t*** in the document
  - Format: <term: number of documents containing the term;

    docID1: position1, position2, ...;

    docID2: position1, position2, ...;

    ...>
  - Example: <„Frodo”: 324;

    2: 3, 99;

    9: 17, 191, 430, 522;

    ...;

    321: 4>

Positional index: query processing

- To support the phrase queries, we need to adapt the merge algorithm to handle phrases and proximity
- Processing a phrase query with positional index
  - (query: „blue Mordor orc”)

1. Fetch (positional) posting lists for each of the terms in the phrase query

   <„blue”: 523; 2: 1, 17, 74; 4: 8, 16, **429**, **563**; 7: 13, 23, 191; ...>

   <„Mordor”: 14; 1: 16, 31; 4: 17, 45, **430**, **564**; 5: 14, 19, 102; ...>

   <„orc”: 341; 3: 19, 321, 512; 4: 121, **431**, **565**; 6: 3, 42; ...>

2. Merge the posting lists by considering not only documents but also term positions (for matching documents)

Proximity queries

- Besides phrase queries, users often pose proximity queries
  - Users define how far apart the query terms may be from each other
  - E.g., „Frodo" /3 „stab" /2 „orc"; where /k means „within k words from"

- Positional index can be leveraged not only for phrase queries but also for proximity queries
  - Because positional postings contain positional information for all term tokens
  - Same merge algorithm can be used both for phrase queries and proximity queries
- Biword indexes lack positional information
  - May support phrase queries,
  - But cannot support proximity queries

Proximity merge algorithm


```
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 – pros and cons

- A positional index substantially expands the postings storage
  - Simple index: For each term and each document in which it appears we stored only (optionally) the frequency with which the terms appears in the document
    - 1 integer per term-document pair
  - Positional index: Store position for each occurrence of the term in the document – *n* occurrences of the term in the document -> *n* integers
    - Positional index size is directly related to the average document size, i.e., the longer the documents, the larger the positional index

- Benefits of a positional index outweight the storage costs
  - Supporting phrase and proximity queries is important

Index sizes – rules of thumb

- A positional index is 2-4 times larger than a corresponding non-positional index
  - **Q:** Why not more than 2-4 times?
- Positional index size is 35-50% of the size of the original text
- Caveat: This approximate size relations hold only for „English-like” languages

Combination schemes

- Combining different indexing strategies may be beneficial

1. Non-positional index to index some frequent phrases (e.g., „Michael Jackson”, „hot potato”)
   - More efficient than merging posting lists
2. Positional indexing as back-off for other phrases (and positional queries)
   - Those that are not directly indexed in the non-positional index

- For more details on indexing combinations, see:
  - H.E. Williams, J. Zobel, and D. Bahle. 2004. Fast Phrase Querying with Combined Indexes, ACM Transactions on Information Systems.

Boolean retrieval – prons and cons

- Advantages
  - Only one: simplicity (i.e., computational efficiency)
  - Popular in early commercial systems (e.g., Westlaw)
- Shortcomings
  - Expressing information needs as Boolean expressions is unintuitive
  - Boolean IR is a pure model
    - No ranking – documents are either relevant or non-relevant
    - Relative importance of indexed terms is ignored
    - Extended Boolean model – a variant of the Boolean model that accounts for the partial fulfillment of the Boolean expression

Boolean retrieval – example

- The Boolean retrieval model was the primary commercial retrieval tool for over 3 decades
- Many search engines we still use daily implement Boolean IR models:
  - Email, library catalog, Mac OS X Spotlight
- Prominent example: Westlaw
  - Largest commercial legal search engine
    - Created in 1975; ranking functionality added only in 1992
    - Tens of terrabytes of data, 700K users
    - Majority of users still use Boolean queries (habit :)
  - Example query:
    - „What is the statute of limitations in cases involving the federal tort claims act?"
    - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM

Now you...

- Know what Boolean retrieval is and what Boolean queries look like
- Know what inverted index is and how to use it to answer Boolean queries
- Understand skip lists and how they may make Boolean retrieval more efficient
- Comprehend phrase queries and understand biword indexes
- Understand the structure of the positional index and how it can be used to support phrase queries as well as proximity queries