Chapter IR:II

II. Indexing

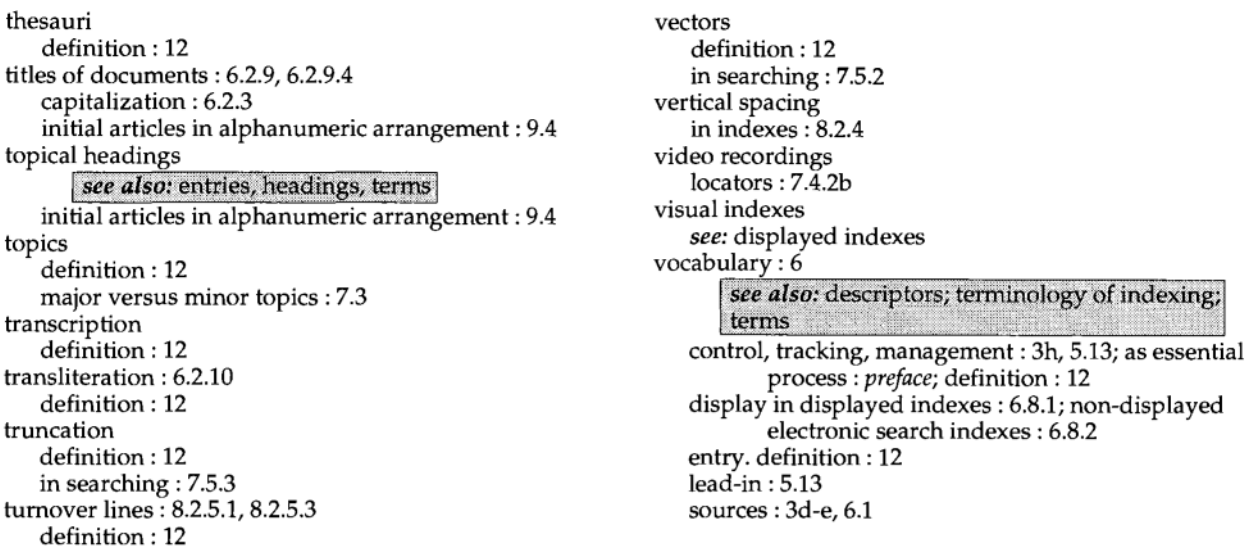
- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

Indexing Basics

Definition 1 (Index [ANSI/NISO 1997])

An index is a systematic guide designed to indicate topics or features of documentary units as index terms in order to facilitate their retrieval.

The function of an index is to provide users with an effective means for locating documentary units relevant to their information needs in answer to queries.



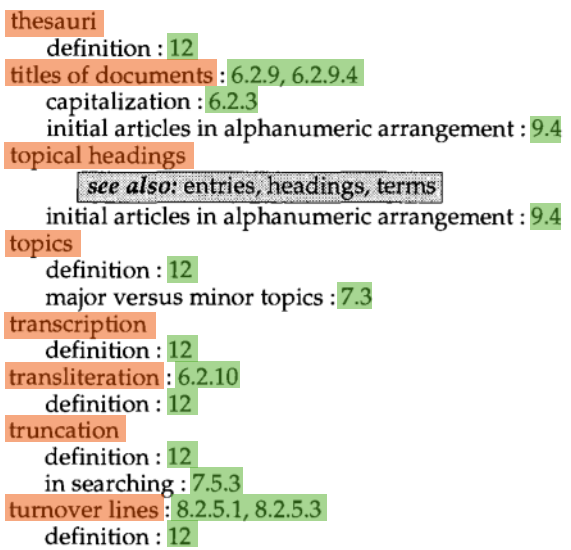
thesauri
definition : 12
titles of documents : 6.2.9, 6.2.9.4
capitalization : 6.2.3
initial articles in alphanumeric arrangement : 9.4
topical headings
see also: entries, headings, terms
initial articles in alphanumeric arrangement : 9.4
topics
definition : 12
major versus minor topics : 7.3
transcription
definition : 12
transliteration : 6.2.10
definition : 12
truncation
definition : 12
in searching : 7.5.3
turnover lines : 8.2.5.1, 8.2.5.3
definition : 12
vectors
definition : 12
in searching : 7.5.2
vertical spacing
in indexes : 8.2.4
video recordings
locators : 7.4.2b
visual indexes
see: displayed indexes
vocabulary : 6
see also: descriptors; terminology of indexing; terms
control, tracking, management : 3h, 5.13; as essential process : *preface*; definition : 12
display in displayed indexes : 6.8.1; non-displayed electronic search indexes : 6.8.2
entry. definition : 12
lead-in : 5.13
sources : 3d-e, 6.1

Indexing Basics

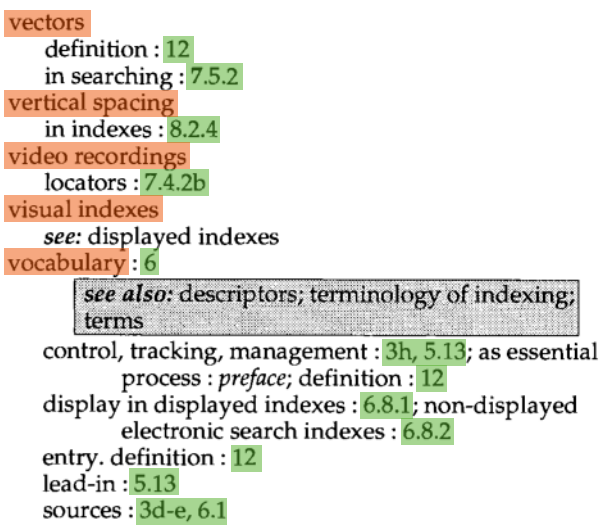
Definition 1 (Index [ANSI/NISO 1997])

An index is a systematic guide designed to indicate topics or features of documentary units as index terms in order to facilitate their retrieval.

The function of an index is to provide users with an effective means for locating documentary units relevant to their information needs in answer to queries.



thesauri
definition : 12
titles of documents : 6.2.9, 6.2.9.4
capitalization : 6.2.3
initial articles in alphanumeric arrangement : 9.4
topical headings
see also: entries, headings, terms
initial articles in alphanumeric arrangement : 9.4
topics
definition : 12
major versus minor topics : 7.3
transcription
definition : 12
transliteration : 6.2.10
definition : 12
truncation
definition : 12
in searching : 7.5.3
turnover lines : 8.2.5.1, 8.2.5.3
definition : 12



vectors
definition : 12
in searching : 7.5.2
vertical spacing
in indexes : 8.2.4
video recordings
locators : 7.4.2b
visual indexes
see: displayed indexes
vocabulary : 6
see also: descriptors; terminology of indexing; terms
control, tracking, management : 3h, 5.13; as essential process : *preface*; definition : 12
display in displayed indexes : 6.8.1; non-displayed electronic search indexes : 6.8.2
entry. definition : 12
lead-in : 5.13
sources : 3d-e, 6.1

Indexing Basics

Querying an Index

Queries are users' formulations of information needs in a search engine's language:

- Keyword queries
- Question queries
- Query by example

Chapter IR:II

II. Indexing

- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

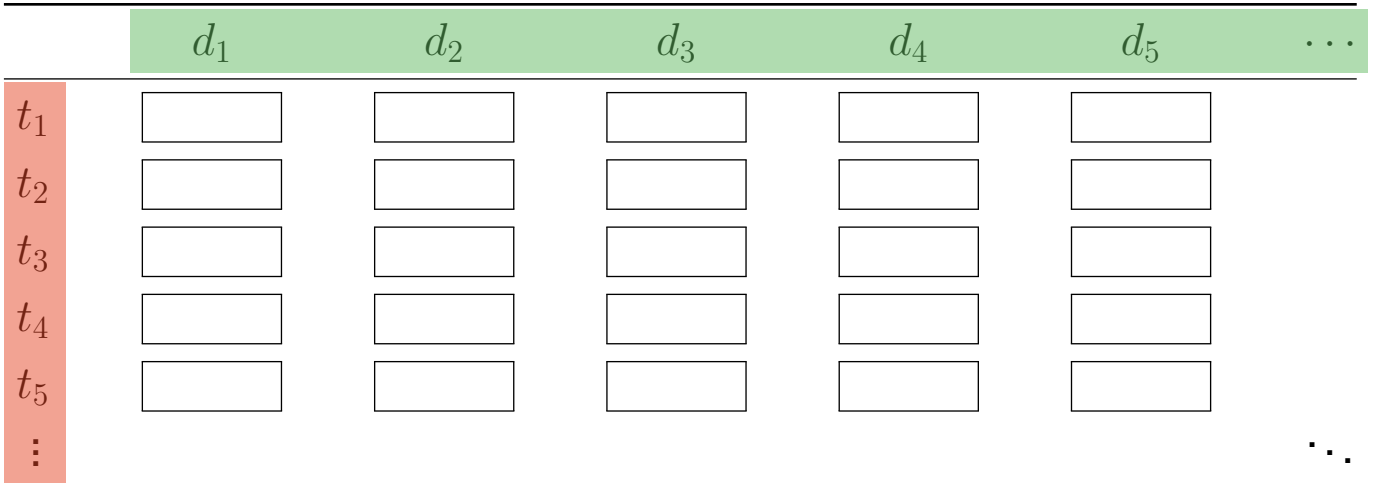
Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1						
t_2						
t_3						
t_4						
t_5						
$\vdots$						$\ddots$

Inverted Index

Term-Document Matrix



- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	1					
t_2	1					
t_3	1					
t_4	0					
t_5	1					
$\vdots$						$\ddots$

- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	1	1	0	0	0	
t_2	1	1	0	1	0	
t_3	1	1	0	1	1	
t_4	0	1	0	0	0	
t_5	1	0	0	0	0	
$\vdots$						$\ddots$

- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	382	128	0	0	0	
t_2	4	379	0	1	0	
t_3	289	272	0	2	1	
t_4	0	16	0	0	0	
t_5	271	0	0	0	0	
$\vdots$						$\ddots$

- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	
t_2	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	
t_3	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	
t_4	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$w_{4,4}$	$w_{4,5}$	
t_5	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$w_{5,4}$	$w_{5,5}$	
$\vdots$						$\ddots$

- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	
t_2	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	
t_3	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	
t_4	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$w_{4,4}$	$w_{4,5}$	
t_5	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$w_{5,4}$	$w_{5,5}$	
$\vdots$						$\ddots$

- Documents D
  - d_1 Antony and Cleopatra
  - d_2 Julius Caesar
  - d_3 The Tempest
  - d_4 Hamlet
  - d_5 Othello
- Index terms T
  - t_1 Antony
  - t_2 Brutus
  - t_3 Caesar
  - t_4 Calpurnia
  - t_5 Cleopatra
- Term Weights
  - Boolean
  - Term frequency
  - ...

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	
t_2	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	
t_3	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	
t_4	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$w_{4,4}$	$w_{4,5}$	
t_5	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$w_{5,4}$	$w_{5,5}$	
$\vdots$						$\ddots$

Observations:

- Most retrieval models induce a term-document matrix by computing term weights $w_{i,j}$ for each pair of term $t_i \in T$ and document $d_j \in D$.
- Query-independent computations that depend only on D are done offline.
- Online, for a query q, the required term weights are looked up to score documents.

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1	$w_{1,1}$	$w_{1,2}$				
t_2	$w_{2,1}$	$w_{2,2}$		$w_{2,4}$		
t_3	$w_{3,1}$	$w_{3,2}$		$w_{3,4}$	$w_{3,5}$	
t_4		$w_{4,2}$				
t_5	$w_{5,1}$					
$\vdots$						$\ddots$

Observations:

- The size of the term-document matrix is $|T| \cdot |D|$.
- The term-document matrix is sparse: the vast majority of term weights are 0.
- Therefore, most of the storage space required for the full matrix is wasted.
- Using a sparse-matrix representation yields significant space savings.

➜ An inverted index efficiently encodes a sparse term-document matrix.

Inverted Index

Data Structure

T	$\rightarrow$	**Postings** (Posting Lists, Postlists)			
t_1	$\rightarrow$	$\boxed{d_1, w_{1,1}}$	$\boxed{d_2, w_{1,2}}$		
t_2	$\rightarrow$	$\boxed{d_1, w_{2,1}}$	$\boxed{d_2, w_{2,2}}$	$\boxed{d_4, w_{2,4}}$	
t_3	$\rightarrow$	$\boxed{d_1, w_{3,1}}$	$\boxed{d_2, w_{3,2}}$	$\boxed{d_4, w_{3,4}}$	$\boxed{d_5, w_{3,5}}$
t_4	$\rightarrow$	$\boxed{d_2, w_{4,2}}$			
t_5	$\rightarrow$	$\boxed{d_1, w_{5,1}}$			
$\vdots$					

An index is implemented as a multimap (i.e., a hash table with multiple values).

Components of an externalized implementation:

- Term vocabulary file
  Lookup table which maps terms $t_i \in T$ to the start of their posting list in the postings file.
- Postings file(s)
  File(s) that store posting lists on disk.
- Index entries $\boxed{d_i, [\ldots]}$, so-called postings

Inverted Index

Data Structure

T	$\rightarrow$	**Postings** (Posting Lists, Postlists)			
t_1	$\rightarrow$	$\boxed{d_1, w_{1,1}}$	$\boxed{d_2, w_{1,2}}$		
t_2	$\rightarrow$	$\boxed{d_1, w_{2,1}}$	$\boxed{d_2, w_{2,2}}$	$\boxed{d_4, w_{2,4}}$	
t_3	$\rightarrow$	$\boxed{d_1, w_{3,1}}$	$\boxed{d_2, w_{3,2}}$	$\boxed{d_4, w_{3,4}}$	$\boxed{d_5, w_{3,5}}$
t_4	$\rightarrow$	$\boxed{d_2, w_{4,2}}$			
t_5	$\rightarrow$	$\boxed{d_1, w_{5,1}}$			
$\vdots$					

An index is implemented as a multimap (i.e., a hash table with multiple values).

Design choices:

- Information stored in a posting $\boxed{d_i, [\ldots]}$.
- Ordering of each term's posting list.
- Encoding and compression techniques for further space savings.
- Physical implementation details, such as external memory and distribution.

Inverted Index

Posting

Given term t and document d, their posting may include the following:

```
<document> [<weights>] [<positions>] ...
```

`<document>`:

- Reference to the document d in which term t occurs (or to which it applies).

`<weights>`:

- Term weight w for term t in document d.
- Often, only basic term weights are stored (e.g., term frequency $tf(t, d)$).
  Storing model-specific weights saves runtime at the expense of flexibility.

`<positions>`:

- Term positions within the document, e.g., term, sentence, page, chapter, etc.
- Field information, e.g., title, author, introduction, etc.

Inverted Index

Posting

Two special-purpose entries are distinguished:

`... [<list length>]`

`... [<skip pointer>]`

`<list length>`:

- Added to the first entry of the posting list of a term t.
- Stores the length of the posting list.
- What does the length of a posting list indicate?

`<skip pointer>`:

- Used to implement a skip list in a term's posting list, when ordered by ID.
- Allows for random access to postings in $O(\log df(t, D))$.
- An effective amount of skip entries has been found to be $\sqrt{df(t, D)}$.
  First entry of a posting list, and then at random (or regular) intervals.

Inverted Index

Posting

Two special-purpose entries are distinguished:

`... [<list length>]` `... [<skip pointer>]`

`<list length>`:

- Added to the first entry of the posting list of a term t.
- Stores the length of the posting list.
- Equals the number of documents containing t (document frequency $df(t, D)$).

`<skip pointer>`:

- Used to implement a skip list in a term's posting list, when ordered by ID.
- Allows for random access to postings in $O(\log df(t, D))$.
- An effective amount of skip entries has been found to be $\sqrt{df(t, D)}$.
  First entry of a posting list, and then at random (or regular) intervals.

Inverted Index

Posting List, Postlist

Example for two posting lists, where for term t_i postings $\boxed{k, \mathit{tf}(t_i, d_k)}$ are stored:

T	**Postings**
⋮	
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	

Ordering:

- by document identifier. Problem: “good” documents randomly distributed.
- by document quality. Problem: index updates more complicated.
- by term weight. Problem: no canonical order across rows; skip lists useless.

Compression:

- The size of an index is in $O(|D|)$, where $|D|$ denotes the disk size of D.
- Posting lists can be effectively compressed with tailored techniques.

Remarks:

- The term “inverted index” is redundant: “index” already denotes the structure in which terms are assigned to the (parts of) documents in which they occur. Better suited, but less frequently used, is “inverted file”, which expresses that a (document) file is “inverted” to form an index. So instead of assigning terms to documents, an index assigns documents to terms.

- A trade-off must be made between the amount of information stored in a posting and the time required to process a post list. The more information stored in a posting, the more has to be loaded into memory and decoded as the posting list is traversed.
- A skip entry can contain more than one pointer, so skip steps of different lengths are possible.
- Depending on the search domain, it may be beneficial to create more than one index with different properties.

Chapter IR:II

II. Indexing

- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

Query Processing I

Retrieval Types

Query processing can be based on two basic approaches:

- Set retrieval
  A query induces a subset of the indexed documents which is considered relevant.
  Important applications: e-discovery, patent search, systematic reviews.

- Ranked retrieval
  A query induces a ranking among all indexed documents in descending order of relevance.

Ranked retrieval is the norm in virtually all modern search engines.

Query Processing I

Query Semantics for Set Retrieval

Keyword queries have Boolean semantics that is either implicitly specified by user behavior and expectations or explicitly specified.

We distinguish four types:

- Single-term queries
- Disjunctive multi-term queries
  Only Boolean OR connectives. Example: Antony $\vee$ Brutus $\vee$ Calpurnia.
- Conjunctive multi-term queries
  Only Boolean AND connectives. Example: Antony $\wedge$ Brutus $\wedge$ Calpurnia.
  - Constraint: Proximity
    Example: Antony /5 Caesar
  - Constraint: Phrase
    Example: “Antony and Caesar”
- “Complex” Boolean multi-term queries
  Remainder of Boolean formulas. Example: (Antony $\vee$ Caesar) $\wedge\neg$ Calpurnia.
  Normalized to disjunctive or conjunctive normal form.

Remarks:

- Which index configuration applies to which type of query?

Query types:

- Single-term queries
- Disjunctive multi-term queries
- Conjunctive multi-term queries
  - Boolean AND queries
  - Proximity queries
  - Phrase queries

Index configurations:

- Postlists ordered by document ID
- Postlists ordered by document quality
- Postlists ordered by term weight
- Positional indexing
  Postings also store term positions.

Remarks:

- Which index configuration applies to which type of query?

  Query types:

  - Single-term queries
  - Disjunctive multi-term queries
  - Conjunctive multi-term queries
    - Boolean AND queries
    - Proximity queries
    - Phrase queries

  Index configurations:

  - Postlists ordered by document ID
  - Postlists ordered by document quality
  - Postlists ordered by term weight
  - Positional indexing
    Postings also store term positions.

- Single-term queries are directly answered with a term weight ordering.
- Disjunctive multi-term queries can be processed with any postlist ordering.
- Conjunctive multi-term queries benefit from a canonical postlist order.
- Proximity and phrase queries require positional indexing.

Query Processing I

Conjunctive Multi-Term Queries

Given an index with postings $k, \mathit{tf}(t, d_k)$ and a query $q = t_1 \wedge \ldots \wedge t_n$, compute the collection $R \subseteq D$ of documents relevant to q.

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

What is the underlying problem to which processing query q can be reduced?

Query Processing I

Conjunctive Multi-Term Queries

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$ and a query $q = t_1 \wedge \ldots \wedge t_n$, compute the collection $R \subseteq D$ of documents relevant to q.

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

Problem: List Intersection.

Instance: $L_1, \ldots, L_n$. $n \geq 2$ skip lists of numbers.

Solution: A sorted list R of numbers, so that each number occurs in all n lists.

Idea: (1) Intersection of the two shortest lists L_i and L_j to obtain $R' \supseteq R$.

(2) Iterative intersection of R' with the remaining lists in ascending order of length.

Query Processing I

List Intersection

Algorithm: Intersection of Two Lists.

Input: L_1, L_2. Skip lists of numbers implemented as singly linked lists.

Output: Sorted list of numbers occurring in both L_1 and L_2.

IntersectTwo(L_1, L_2)

1. Initialization of result list R and one iterator variable x_1 and x_2 per list.
2. While the iterators point to list entries, process them as follows.
3. If the list entries' keys match, append a merged entry to the result list R.
4. While the key of x_1 is smaller than that of x_2 advance x_1.
5. While the key of x_2 is smaller than that of x_1 advance x_2.
6. Return R, once an iterator reaches the end of its list.

Query Processing I

List Intersection

Algorithm: Intersection of Two Lists.

Input: L_1, L_2. Skip lists of numbers implemented as singly linked lists.

Output: Sorted list of numbers occurring in both L_1 and L_2.

IntersectTwo(L_1, L_2)

```
 1.  R = list();  x1 = L1.head;  x2 = L2.head
 2.  WHILE x1 ≠ NIL AND x2 ≠ NIL DO
 3.    IF x1.key == x2.key THEN
 4.      R = Insert(R, merge(x1, x2))
 5.      x1 = x1.next;  x2 = x2.next
 6.    ENDIF
 7.    WHILE x1 ≠ NIL AND x2 ≠ NIL AND x1.key < x2.key DO
 8.      IF CanSkip(x1, x2.key) THEN
 9.        x1 = Skip(x1, x2.key)
10.      ELSE
11.        x1 = x1.next
12.      ENDIF
13.    ENDDO
 ⋮     Like lines 7-13 with x1 and x2 exchanged.
21.  ENDDO
22.  return(R)
```

Query Processing I

List Intersection

Algorithm: Intersection of Two Lists.

Input: L_1, L_2. Skip lists of numbers implemented as singly linked lists.

Output: Sorted list of numbers occurring in both L_1 and L_2.

IntersectTwo(L_1, L_2)

```
 1.  R = list();  x1 = L1.head;  x2 = L2.head
 2.  WHILE x1 ≠ NIL AND x2 ≠ NIL DO
 3.    IF x1.key == x2.key THEN
 4.      R = Insert(R, merge(x1, x2))
 5.      x1 = x1.next;  x2 = x2.next
 6.    ENDIF
  ⋮    Like lines 14-20 with x1 and x2 exchanged.
14.    WHILE x1 ≠ NIL AND x2 ≠ NIL AND x2.key < x1.key DO
15.      IF CanSkip(x2, x1.key) THEN
16.        x2 = Skip(x2, x1.key)
17.      ELSE
18.        x2 = x2.next
19.      ENDIF
20.    ENDDO
21.  ENDDO
22.  return(R)
```

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

Execute *IntersectTwo*(L_i, L_j).

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

Execute *IntersectTwo*(L_i, L_j).

Result $R = ()$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = ()$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$ $\boxed{41, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$ $\boxed{41, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$ $\boxed{41, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**
⋮	x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$ …
⋮	x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$ $\boxed{41, \ldots}$ $\boxed{77, \ldots}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \mathit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**	
⋮		x_i
t_i	$\boxed{2,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ $\boxed{23,5}$ $\boxed{28,6}$ $\boxed{41,8}$ $\boxed{50,6}$ $\boxed{77,8}$	$\ldots$
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{3,5}$ $\boxed{5,2}$ $\boxed{8,17}$ $\boxed{41,6}$ $\boxed{51,5}$ $\boxed{60,5}$ $\boxed{71,3}$ $\boxed{77,2}$	$\ldots$
⋮		x_j

Execute *IntersectTwo*(L_i, L_j).

Result $R = \boxed{2, \ldots}$ $\boxed{8, \ldots}$ $\boxed{41, \ldots}$ $\boxed{77, \ldots}$

Remarks:

- Postlists are usually too large to fit in main memory, so iterating them brings performance benefits.
- The *key* attribute stores the document identifier of a posting.
- The *merge* function returns a posting merged from the two postings passed in. It merges the potentially stored term weights and other information stored in them.
- The *next* attribute stores the successive posting.
- The *CanSkip* function checks whether the current posting contains skip information and whether a target with a document identifier less than or equal to the *key* value passed is available.
- The *Skip* function returns the posting that is closest to but less than or equal to the *key* value passed.

Query Processing I

List Intersection

Algorithm: Intersect Many Lists.

Input: $L_1, \ldots, L_n$. Skip lists of numbers implemented as singly linked lists.

Output: Sorted list of numbers occurring in all $L_1, \ldots, L_n$.

IntersectMany$(L_1, \ldots, L_n)$

```
// Sort by list length.
1. H = BuildMinHeap(L_1, ..., L_n);
2. R = ExtractMin(H)
3. WHILE |H| > 0 DO
4.     L_min = ExtractMin(H)
5.     R = IntersectTwo(R, L_min)
6. ENDDO
7. return(R)
```

Why are lists intersected in ascending order of list length?

Query Processing I

List Intersection

Algorithm: Intersect Many Lists.

Input: $L_1, \ldots, L_n$. Skip lists of numbers implemented as singly linked lists.

Output: Sorted list of numbers occurring in all $L_1, \ldots, L_n$.

IntersectMany$(L_1, \ldots, L_n)$

```
// Sort by list length.
1. H = BuildMinHeap(L_1, ..., L_n);
2. R = ExtractMin(H)
3. WHILE |H| > 0 DO
4.     L_min = ExtractMin(H)
5.     R = IntersectTwo(R, L_min)
6. ENDDO
7. return(R)
```

Observations:

- The amount of memory required to store the result list R is bounded by the shortest list from $L_1, \ldots, L_n$.
- The smaller the result list R, the more effective are the skip pointers.
- Hard disk seeking is minimized since every list is read sequentially.

Query Processing I

Proximity Queries

Given a query $q = t_i /\epsilon\ t_j$, retrieve documents in which t_i and t_j are in close proximity, i.e., within an ϵ-environment of one another, where $\epsilon \geq 1$ terms.

Query Processing I

Proximity Queries

Given a query $q = t_i /\epsilon\ t_j$, retrieve documents in which t_i and t_j are in close proximity, i.e., within an ϵ-environment of one another, where $\epsilon \geq 1$ terms.

Processing proximity queries requires term positions in postings:

```
<document> [<weights>] [<positions>] [...]
```

Query Processing I

Proximity Queries

Given a query $q = t_i /\epsilon\ t_j$, retrieve documents in which t_i and t_j are in close proximity, i.e., within an ϵ-environment of one another, where $\epsilon \geq 1$ terms.

Processing proximity queries requires term positions in postings:

```
<document> [<weights>] [<positions>] [...]
```

Example:

$d =$ “You cannot end a sentence with because because because is a conjunction.”
$\quad\ 1\quad\ 2\quad\ 3\ \ 4\quad\ 5\quad\ 6\quad\ 7\quad\ 8\quad\ 9\quad\ 10\ 11\quad\ 12$

Posting for “because” and d:

```
d, 3, (7,8,9)
```

Posting for “sentence” and d:

```
d, 1, (5)
```

In d, “because” is in a 2-environment of {“sentence”, “with”, “because”, “is”, “a”}.

Query Processing I

Proximity Queries

Algorithm: Position List Intersection.

Input: A_1, A_2. Sorted arrays of positions of two terms t_1, t_2 in a document d. ϵ. Maximal term distance.

Output: For each position in A_1, the positions from A_2 within an ϵ-environment.

IntersectPositions(A_1, A_2, ϵ)

```
1.  R = map()
2.  FOR i = 1 TO A1.length DO
3.    R' = list()
4.    FOR j = 1 TO A2.length DO
5.      IF |A1[i] − A2[j]| ≤ ε THEN
6.        insert(R', A2[j])
7.      ELSE IF A2[j] > A1[i] THEN
8.        break
9.      ENDIF
10.   ENDDO
11.   insert(R, A1[i], R')
12. ENDDO
13. return(R)
```

Remarks:

- Pruning unnecessary comparisons
  Lines 7–9: Stop comparing once the j-th position in A_2 exceeds the i-th position in A_1 by more than ϵ. The difference can never get smaller than ϵ again.
- Integration into postlist intersection
  The if-statement of Line 3 of *IntersectTwo* then additionally checks whether *IntersectPositions* returns a non-empty result.

Query Processing I

Phrase Queries

Given a phrase query $q =$“$t_1 \ \ldots \ t_m$”, retrieve documents in which the terms $t_1, \ldots, t_m$ occur in the same order as in the query q.

Query Processing I

Phrase Queries

Given a phrase query $q =$“$t_1 \ \ldots \ t_m$”, retrieve documents in which the terms $t_1, \ldots, t_m$ occur in the same order as in the query q.

Processing phrase queries requires term positions in postings.

Example:

T	**Postings**
to	... $4, 250, (..., 133, 137, ...)$ ...
be	... $4, 125, (..., 134, 138, ...)$ ...
or	... $4, 40, (..., 135, ...)$ ...
not	... $4, 15, (..., 136, ...)$ ...

What phrase does document 4 contain?

Query Processing I

Phrase Queries

Given a phrase query $q =$“$t_1 \ \ldots \ t_m$”, retrieve documents in which the terms $t_1, \ldots, t_m$ occur in the same order as in the query q.

Processing phrase queries requires term positions in postings.

Example:

T	**Postings**
`to`	$\ldots$ $4, 250, (\ldots, 133, 137, \ldots)$ $\ldots$
`be`	$\ldots$ $4, 125, (\ldots, 134, 138, \ldots)$ $\ldots$
`or`	$\ldots$ $4, 40, (\ldots, 135, \ldots)$ $\ldots$
`not`	$\ldots$ $4, 15, (\ldots, 136, \ldots)$ $\ldots$

Document 4 contains the phrase `to be or not to be` at term positions 133–138.

Observations:

- Processing phrase queries can be reduced to the list intersection problem.
  Algorithms *IntersectMany* and *IntersectTwo* can be adjusted to process phrase queries.
- The additional run time for phrase processing is in $O(\sum_{d \in \textit{IntersectMany}(L_t : t \in q)} |d|)$.

Query Processing I

Phrase Queries

Given a phrase query $q =$“$t_1 \ \ldots \ t_m$”, retrieve documents in which the terms $t_1, \ldots, t_m$ occur in the same order as in the query q.

To speed up phrase search, n-grams can be used as index terms.

Example:

T	**Postings**
`to be`	$\ldots$ $4, 80, (\ldots, 133, 137, \ldots)$ $\ldots$
`be or`	$\ldots$ $4, 55, (\ldots, 134, \ldots)$ $\ldots$
`or not`	$\ldots$ $4, 20, (\ldots, 135, \ldots)$ $\ldots$
`not to`	$\ldots$ $4, 7, (\ldots, 136, \ldots)$ $\ldots$

Document 4 contains the phrase `to be or not to be` at term positions 133–138.

How much faster can phrase queries be processed?

Query Processing I

Phrase Queries

Given a phrase query $q =$“$t_1 \ \ldots \ t_m$”, retrieve documents in which the terms $t_1, \ldots, t_m$ occur in the same order as in the query q.

To speed up phrase search, n-grams can be used as index terms.

Example:

T	**Postings**
`to be`	$\ldots$ $4, 80, (\ldots, 133, 137, \ldots)$ $\ldots$
`be or`	$\ldots$ $4, 55, (\ldots, 134, \ldots)$ $\ldots$
`or not`	$\ldots$ $4, 20, (\ldots, 135, \ldots)$ $\ldots$
`not to`	$\ldots$ $4, 7, (\ldots, 136, \ldots)$ $\ldots$

Document 4 contains the phrase `to be or not to be` at term positions 133–138.

Observations:

- The time to process phrase queries of length at least n is divided by n.
  Only non-overlapping n-grams need to be intersected.
- Maintaining an index with n-grams and/or common phrases as index terms speeds up non-phrase queries as well.

Remarks:

- The space requirements of a positional index are 2–4 times that of a nonpositional index.
- Most basic retrieval models do not directly employ positional information. If keyword proximity is a desired feature in a retrieval system using a basic retrieval model, positional information usually is implemented as an additional relevance signal or as a prior probability for a document.

Chapter IR:II

II. Indexing

- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

Query Processing II

Retrieval Types

Query processing can be based on two basic approaches:

- Set retrieval
  A query induces a subset of the indexed documents which is considered relevant.
  Important applications: e-discovery, patent search, systematic reviews.

- Ranked retrieval
  A query induces a ranking among all indexed documents in descending order of relevance.

Ranked retrieval is the norm in virtually all modern search engines.

Query Processing II

Relevance Scoring (Recap)

Quantification of the relevance of an indexed document d to a query q.

Query Processing II

Relevance Scoring (Recap)

Quantification of the relevance of an indexed document d to a query q.

Let $t \in T$ denote a term t from the terminology T of index terms, and let $\omega_X : T \times X \to \mathbf{R}$ denote a term weighting function, where X may be a set of documents D or a set of queries Q. Then the most basic relevance function ρ is:

$$\rho(q, d) = \sum_{t \in T} \omega_Q(t, q) \cdot \omega_D(t, d),$$

where $\omega_Q(t, q)$ and $\omega_D(t, d)$ are term weights indicating the importance of t for the query $q \in Q$ and the document $d \in D$, respectively.

Query Processing II

Relevance Scoring (Recap)

Quantification of the relevance of an indexed document d to a query q.

Let $t \in T$ denote a term t from the terminology T of index terms, and let $\omega_X : T \times X \to \mathbf{R}$ denote a term weighting function, where X may be a set of documents D or a set of queries Q. Then the most basic relevance function ρ is:

$$\rho(q, d) = \sum_{t \in T} \omega_Q(t, q) \cdot \omega_D(t, d),$$

where $\omega_Q(t, q)$ and $\omega_D(t, d)$ are term weights indicating the importance of t for the query $q \in Q$ and the document $d \in D$, respectively.

Observations:

- A term t may have importance, and hence non-zero weights, for a query q or document d despite not occurring in them. Example: synonyms.
- The majority of terms from T will have insignificant importance to both.
- The term weights $\omega_D(t, d)$ can be pre-computed and indexed.
- The term weights $\omega_Q(t, q)$ must be computed on the fly.

Query Processing II

Query Semantics for Ranked Retrieval

Keyword queries have Boolean semantics that is either implicitly specified by user behavior and expectations or explicitly specified.

We distinguish four types:

- Single-term queries
- Disjunctive multi-term queries
  Only Boolean OR connectives. Example: Antony $\vee$ Brutus $\vee$ Calpurnia.
- Conjunctive multi-term queries
  Only Boolean AND connectives. Example: Antony $\wedge$ Brutus $\wedge$ Calpurnia.
  - + Constraint: Proximity
    Example: Antony $/\epsilon$ Caesar
  - + Constraint: Phrase
    Example: “Antony and Caesar”
- “Complex” Boolean multi-term queries
  Remainder of Boolean formulas. Example: (Antony $\vee$ Caesar) $\wedge\neg$ Calpurnia.
  Can be normalized to disjunctive or conjunctive normal form.

Query Processing II

Single-Term Queries

Given a single-term query $q = t$, the optimal postlist ordering is by term weight.

Example:

T	**Postings** (ordered by document identifier)
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

Worst case: The last document of the postlist is the most relevant one.
The whole postlist must be examined.

Query Processing II

Single-Term Queries

Given a single-term query $q = t$, the optimal postlist ordering is by term weight.

Example:

T	**Postings** (ordered by term weight)
⋮	
t_i	$4, 9$ $41, 8$ $77, 8$ $19, 7$ $28, 6$ $50, 6$ $23, 5$ $2, 4$ $8, 2$ $16, 1$ …
t_j	$8, 17$ $41, 6$ $3, 5$ $51, 5$ $60, 5$ $2, 3$ $71, 3$ $5, 2$ $77, 2$ $1, 1$ …
⋮	

Best case: The document whose content is best represented by the term t is the one with the highest term weight. A partial examination of the postlist suffices.

Including a skip list in a postlist ordered by term weights may not be useful.

Query Processing II

Disjunctive Queries

In general, a query q is processed as a disjunctive query, where each term $t_i \in q$ may or may not occur in a relevant document d, as long as at least one t_i occurs.

Document-at-a-time scoring

- Precondition: a total order of documents in the index's postlists is enforced
  Ordering criterion: document ID or document quality
- Parallel traversal of query term postlists, document ID by document ID.
- Each document's score is instantly complete, but the ranking only at the end.
- Concurrent disk IO overhead increases with query length.

Query Processing II

Disjunctive Queries

In general, a query q is processed as a disjunctive query, where each term $t_i \in q$ may or may not occur in a relevant document d, as long as at least one t_i occurs.

Document-at-a-time scoring

- Precondition: a total order of documents in the index's postlists is enforced
  Ordering criterion: document ID or document quality
- Parallel traversal of query term postlists, document ID by document ID.
- Each document's score is instantly complete, but the ranking only at the end.
- Concurrent disk IO overhead increases with query length.

Term-at-a-time scoring

- Iterative traversal of query term postlists (e.g., in order of term frequency).
- Temporary query postlist contains candidate documents.
- As document scores accumulate, an approximate ranking becomes available.

- More main memory required for maintaining temporary postlist.

Safe and unsafe optimizations exist (e.g., to stop the search early).

Remarks:

- Web search engines often return results without some of a query's terms for very specific queries, indicating a disjunctive interpretation. Nevertheless, many retrieval models assign higher scores to documents matching more of a query's terms, leaning toward a "conjunctive" interpretation at least for the (visible) top results.

Query Processing II

Disjunctive Queries

Algorithm: Document-at-a-time Scoring.

Input: $L_1, \ldots, L_m$. The postlists of the terms $t_1, \ldots, t_m$ of query q.
$\mathbf{q}$. Representation of query q, e.g., as array of m term weights.

Output: A list of documents in D, sorted in descending order of relevance to q.

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

1. Initialization of result list R as priority queue, and postlist iterator variables.
2. While not all postlists have been processed, repeat the following steps.
3. Determine the smallest document identifier d to which the iterators point.
4. Collect all term weights of d in an array $\mathbf{d}$.
5. Calculate the relevance score $\rho(\mathbf{q}, \mathbf{d})$ and insert it in R.
6. Advance all iterators pointing to d.
7. Return the list of scored documents R.

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**
t_i	$\boxed{1,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ …
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{5,5}$ $\boxed{7,2}$ $\boxed{8,8}$ …
t_k	$\boxed{1,2}$ $\boxed{2,4}$ $\boxed{5,1}$ $\boxed{6,3}$ $\boxed{8,5}$ …

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**
⋮	x_i
t_i	$1,4$ $4,9$ $8,2$ $16,1$ $19,7$ …
⋮	x_j
t_j	$1,1$ $2,3$ $5,5$ $7,2$ $8,8$ …
⋮	x_k
t_k	$1,2$ $2,4$ $5,1$ $6,3$ $8,5$ …
⋮	

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_1 = \begin{pmatrix} \vdots \\ 4 \\ \vdots \\ 1 \\ \vdots \\ 2 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	Postings
	x_i
t_i	$1,4$ $4,9$ $8,2$ $16,1$ $19,7$ ...
	x_j
t_j	$1,1$ $2,3$ $5,5$ $7,2$ $8,8$ ...
	x_k
t_k	$1,2$ $2,4$ $5,1$ $6,3$ $8,5$ ...

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_2 = \begin{pmatrix} \vdots \\ 0 \\ \vdots \\ 3 \\ \vdots \\ 4 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head;  ...;  x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	Postings
	x_i
t_i	$1,4$ $4,9$ $8,2$ $16,1$ $19,7$ …
	x_j
t_j	$1,1$ $2,3$ $5,5$ $7,2$ $8,8$ …
	x_k
t_k	$1,2$ $2,4$ $5,1$ $6,3$ $8,5$ …

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_4 = \begin{pmatrix} \vdots \\ 9 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**
⋮	x_i
t_i	$1,4$ $4,9$ $8,2$ $16,1$ $19,7$ …
⋮	x_j
t_j	$1,1$ $2,3$ $5,5$ $7,2$ $8,8$ …
⋮	x_k
t_k	$1,2$ $2,4$ $5,1$ $6,3$ $8,5$ …
⋮	

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_5 = \begin{pmatrix} \vdots \\ 0 \\ \vdots \\ 5 \\ \vdots \\ 1 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	Postings					
			x_i			
t_i	$1,4$	$4,9$	$8,2$	$16,1$	$19,7$	$\ldots$
			x_j			
t_j	$1,1$	$2,3$	$5,5$	$7,2$	$8,8$	$\ldots$
			x_k			
t_k	$1,2$	$2,4$	$5,1$	$6,3$	$8,5$	$\ldots$

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_6 = \begin{pmatrix} \vdots \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 3 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	Postings
	x_i
t_i	$\boxed{1,4}$ $\boxed{4,9}$ $\boxed{8,2}$ $\boxed{16,1}$ $\boxed{19,7}$ …
	x_j
t_j	$\boxed{1,1}$ $\boxed{2,3}$ $\boxed{5,5}$ $\boxed{7,2}$ $\boxed{8,8}$ …
	x_k
t_k	$\boxed{1,2}$ $\boxed{2,4}$ $\boxed{5,1}$ $\boxed{6,3}$ $\boxed{8,5}$ …

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_7 = \begin{pmatrix} \vdots \\ 0 \\ \vdots \\ 2 \\ \vdots \\ 0 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	Postings
	x_i
t_i	$1,4$ $4,9$ $8,2$ $16,1$ $19,7$ …
	x_j
t_j	$1,1$ $2,3$ $5,5$ $7,2$ $8,8$ …
	x_k
t_k	$1,2$ $2,4$ $5,1$ $6,3$ $8,5$ …

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_8 = \begin{pmatrix} \vdots \\ 2 \\ \vdots \\ 8 \\ \vdots \\ 5 \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head;  ...;  x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**					
					x_i	
t_i	$1,4$	$4,9$	$8,2$	$16,1$	$19,7$	$\ldots$
						x_j
t_j	$1,1$	$2,3$	$5,5$	$7,2$	$8,8$	$\ldots$
						x_k
t_k	$1,2$	$2,4$	$5,1$	$6,3$	$8,5$	$\ldots$

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_{16} = \begin{pmatrix} \vdots \\ 1 \\ \vdots \\ w_j \\ \vdots \\ w_k \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**					
					x_i	
t_i	$1,4$	$4,9$	$8,2$	$16,1$	$19,7$	$\ldots$
						x_j
t_j	$1,1$	$2,3$	$5,5$	$7,2$	$8,8$	$\ldots$
						x_k
t_k	$1,2$	$2,4$	$5,1$	$6,3$	$8,5$	$\ldots$

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d}_{19} = \begin{pmatrix} \vdots \\ 7 \\ \vdots \\ w_j \\ \vdots \\ w_k \\ \vdots \end{pmatrix}$$

Query Processing II

Disjunctive Queries

DAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = PriorityQueue()
2.  x_1 = L_1.head; ...; x_m = L_m.head
3.  continue = TRUE
4.  WHILE continue DO
5.    d = min_{i∈[1,m]}(x_i.key)
6.    d = Array(|q|)
7.    FOR i ∈ [1,m] DO
8.      IF x_i ≠ NIL AND x_i.key = d THEN
9.        d[i] = x_i.weight
10.     ENDIF
11.   ENDDO
12.   r = ρ(q, d)
13.   Insert(R, record(d, r))
14.   continue = FALSE
15.   FOR i ∈ [1,m] DO
16.     IF x_i ≠ NIL AND x_i.key = d THEN
17.       x_i = x_i.next
18.     ENDIF
19.     IF x_i ≠ NIL THEN
20.       continue = TRUE
21.     ENDIF
22.   ENDDO
23. ENDDO
24. return(R)
```

Example:

T	**Postings**					
⋮						x_i
t_i	$1,4$	$4,9$	$8,2$	$16,1$	$19,7$	…
⋮						x_j
t_j	$1,1$	$2,3$	$5,5$	$7,2$	$8,8$	…
⋮						x_k
t_k	$1,2$	$2,4$	$5,1$	$6,3$	$8,5$	…
⋮						

$$\mathbf{q} = \begin{pmatrix} \vdots \\ 5 \\ \vdots \\ 8 \\ \vdots \\ 3 \\ \vdots \end{pmatrix} \qquad \mathbf{d} = \begin{pmatrix} \vdots \\ w_i \\ \vdots \\ w_j \\ \vdots \\ w_k \\ \vdots \end{pmatrix}$$

Remarks:

- DAAT = Document at a time
- We distinguish between a real-world query q and its computer representation $\mathbf{q}$. Likewise, document (identifier) d's representation is $\mathbf{d}$. More complex representations can be imagined than the array-of-weights representations exemplified.
- Relevance function $\rho(\mathbf{q}, \mathbf{d})$ maps pairs of document and query representations to a real-valued score indicating document d's relevance to query q.
- Document-at-a-time scoring makes heavy use of disk seeks. With increasing query length $|q|$, dependent on the type of disks used, and the distribution of the index across disks, the practical run time of this approach can be poor (albeit, theoretically, exactly the same postings are processed as for term-at-a-time scoring).
- Document-at-a-time scoring has a rather small memory footprint on the order of the number of documents to return. This footprint can easily be bounded within top-k retrieval by limiting the size of the results priority queue to the k entries with the currently highest scores.
- Document-at-a-time scoring presumes a global postlist ordering by document identifier or document quality.

Query Processing II

Disjunctive Queries

Algorithm: Term-at-a-time Scoring.

Input: $L_1, \ldots, L_m$. The postlists of the terms $t_1, \ldots, t_m$ of query q.
$\mathbf{q}$. Representation of query q, e.g., as array of m term weights.

Output: A list of documents in D, sorted in descending order of relevance to q.

TAATScoring$(L_1, \ldots, L_m, \mathbf{q})$

```
1.  R = map()
2.  FOR i ∈ [1, m] DO
3.    x_i = L_i.head
4.    WHILE x_i ≠ NIL DO
5.      d = x_i.key
6.      w = x_i.weight
7.      R[d] = R[d] + q[i] · w
8.      x_i = x_i.next
9.    ENDDO
10. ENDDO
11. return(PriorityQueue(R))
```

1. Initialization of result list R as map.
2. Process postlists interatively.
3. Initialization of postlist iterator for the i-th postlist.
4. For each document d's posting in the postlist:
5. Get d's ID.
6. Get t's term weight for d.
7. Update d's partial document score.
8. Advance the iterator.
11. Return the result list, ordered by document scores.

Remarks:

- TAAT = Term at a time
- Term-at-a-time scoring has a comparably high main memory load, since the last "intermediate" $|R| = |\bigcup_{i=1}^{m} L_i|$ before an actual ordering is performed. Otherwise, postlists are read consecutively, which suits rotating hard disks. Massive parallelization is possible.
- The order in which terms are processed (Line 2) affects how quick the intermediate scores in R approach the final document scores.
- The relevance function ρ must be additive (Line 7), or otherwise incrementally computable.
- Term-at-a-time scoring makes no a priori assumptions about postlist ordering; in case of conjunctive interpretation some ordering by document identifier is still very helpful since then skip lists can be exploited. However, to speed up retrieval and allow for (unsafe) early termination, ordering by term weight is required.

Query Processing II

Top-k Retrieval

Search engine users are often interested only in the top ranked k documents. Lower-ranked documents will likely never be viewed.

Query processing optimization approaches:

- Term weight threshold
  TAAT-scoring: skip query terms whose inverse document frequency is lower than that of other query terms. Exception: stop word-heavy queries (e.g., `to be or not to be`).
- Relevance score threshold
  DAAT-scoring: once $>k$ documents have been found, determine co-occurring query terms in the top k ones; skip remaining documents not containing co-occurring query terms.
- Early termination
  Postlists ordered by term weight: stop postlist traversal early, disregarding the rest of the postlist that cannot contribute enough to a document's relevance score.
- Tiered indexes
  Divide documents into index tiers by quality or term frequency. If an insufficient amount of documents is found in the top tier, resort to the next one.

Query Processing II

Index Distribution

The larger the size of the document collection D to be indexed, the more query processing time can be improved by scaling up and scaling out.

Term distribution

- Distribution of postlists across local disks.
- Speeds up processing on spinning hard drives.

Document distribution (also: sharding)

- Random division of the document collection into subsets (so-called shards) and indexing of each shard on a different server for parallel query processing.

- Benefit: Smaller indexes return (more) results faster due to shorter postlists.
- Overhead: Query broker to dispatch queries and fuse each server's results.

Tiered indexes

- Sharding of the document collection into tiers (e.g., by document importance)
- For instance: Tier 1 shards are kept in RAM, Tier 2 shards are kept in flash

memory, and Tier 3 shards on spinning hard disks.

Query Processing II

Caching

Queries obey Zipf's law: roughly half the queries a day are unique on that day. Moreover, about 15% of the queries per day have never occurred before [Gomes 2017].

Consequently, the majority of queries have been seen before, enabling the use of caching to speed up query processing.

Caching can be applied at various points:

- Result caching
- Caching of postlist intersections
- Postlist caching

Individual cache refresh strategies must be employed to avoid stale data. Cache hierarchies of hardware and operating system should be exploited.

Chapter IR:II

II. Indexing

- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

Compression

Size Issues

Inverted lists can become very large.

- Rule of thumb: 25–50% of document collection.
- 2–4 times higher if n-grams are indexed.

Compression of indexes saves disk and/or memory space.

- Best techniques have good compression ratios, easy to decompress.
- Reduces seek time on disk.
- Disadvantage: Decompression time.

We need lossless compression ➜ no information lost

- Lossy compression for images, audio, video with very high compression ratios

As we iterate the posting list, read a stream of bits to decode postings.

- Postings must be decoded while reading.

Compression

Basic Idea

Common elements use short codes, uncommon elements use long codes.

- Posting lists are just lists of numbers.

Naïve number coding:

- Number sequence: 0, 1, 0, 2, 0, 3, 0
- Possible encoding (2 bits): 00 01 00 10 00 11 00
- Encode 0 using a single 0: 0 01 0 10 0 11 0 ➜ 0 is common element
- Only 10 bits, but looks like: 0 01 01 0 0 11 0
- Which encodes: 0, 1, 1, 0, 0, 3, 0
  - Oops!

Compression

Basic Idea

Common elements use short codes, uncommon elements use long codes.

- Posting lists are just lists of numbers.

Naïve number coding:

- Number sequence: 0, 1, 0, 2, 0, 3, 0
- Possible encoding (2 bits): 00 01 00 10 00 11 00
- Encode 0 using a single 0: 0 01 0 10 0 11 0 ➜ 0 is common element
- Only 10 bits, but looks like: 0 01 01 0 0 11 0
- Which encodes: 0, 1, 1, 0, 0, 3, 0
  - Oops!

Unambiguous coding:

- $0 \rightarrow 0$, $1 \rightarrow 101$, $2 \rightarrow 110$, $3 \rightarrow 111$ ➜ add a 1 before each number
- Yields 0 101 0 110 0 111 0 (13 bits)
- 2-bit encoding was also unambiguous (14 bits)

Compression

Unambiguous codes

Goal: Small numbers receive small code values ➜ Unary code.

- Encode k by k 1s followed by 0 (0 at end makes code unambiguous).
- $0 \to 0, \quad 1 \to 10, \quad 2 \to 110, \quad 3 \to 1110, \ldots$

Unary: efficient for small numbers such as 0 and 1, but quickly becomes expensive.

- 1023 can be represented in 10 binary bits, but requires 1024 bits in unary.

Binary: efficient for large numbers, but it may be ambiguous (not byte aligned).

- Not so useful on its own for compression.

Compression

Elias-γ Code

Let's use advantages from unary and binary encoding schemes.

To encode a number k, compute $k_d = \lfloor \log_2 k \rfloor$ and $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

- k_d is least amount of binary digits needed (highest power of 2).
- k_r is k after removing the leftmost 1 of its binary encoding ($k > 0$).

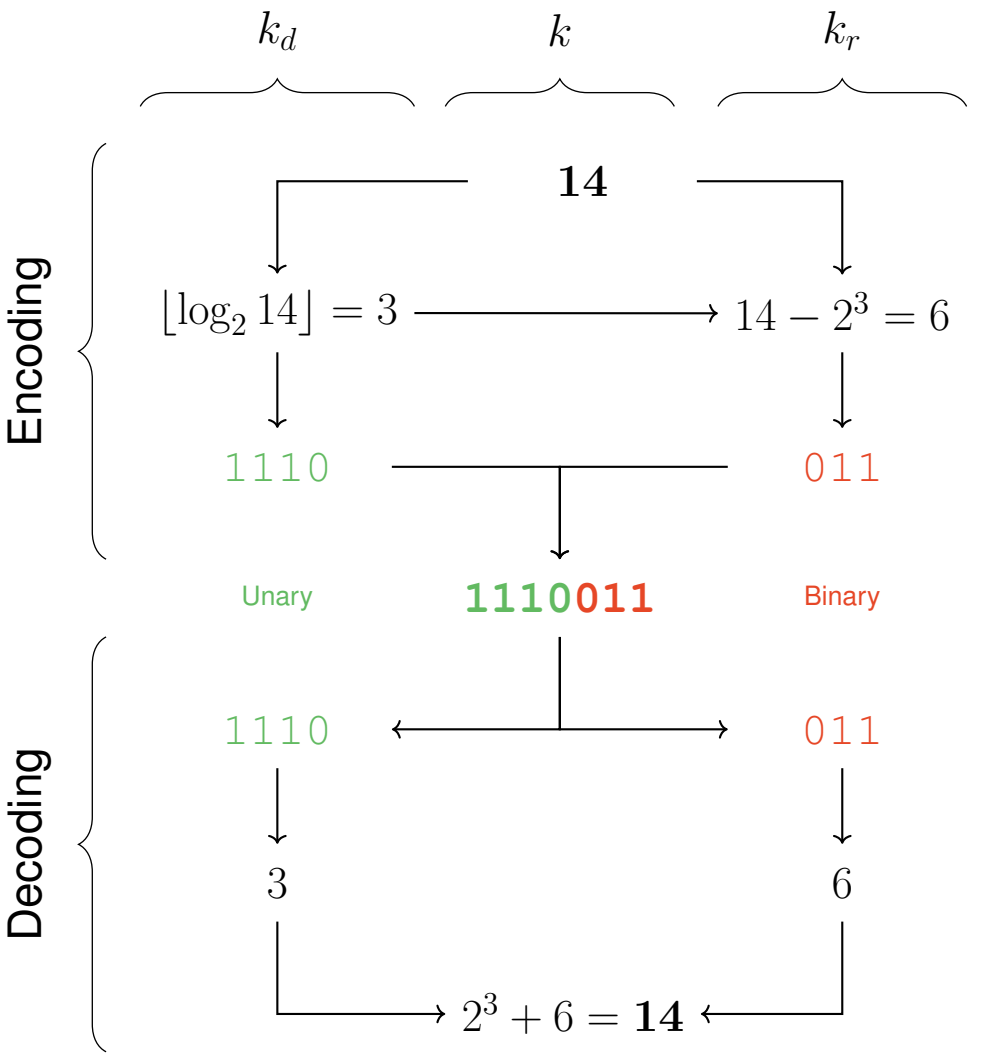
Encode: k_d as unary (followed by 0) and k_r as binary (in k_d binary digits).

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Decode: Let N be k_d without the final 0, then $2^N + k_r$.

Compression

Elias-γ Code



Compression

Elias-γ Code

Elias-γ code never uses more bits than unary, many fewer for $k > 2$

- 1023 takes just 19 bits instead of 1024 bits using unary

In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits

- $\lfloor \log_2 k \rfloor + 1$ for unary part
- $\lfloor \log_2 k \rfloor$ for binary part

In binary, can encode k in $\lfloor \log_2 k \rfloor$ bits.

- Elias-γ needs twice as much as binary to make it unambiguous.

Compression

Elias-γ Code

Elias-γ code never uses more bits than unary, many fewer for $k > 2$

- 1023 takes just 19 bits instead of 1024 bits using unary

In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits

- $\lfloor \log_2 k \rfloor + 1$ for unary part
- $\lfloor \log_2 k \rfloor$ for binary part

In binary, can encode k in $\lfloor \log_2 k \rfloor$ bits.

- Elias-γ needs twice as much as binary to make it unambiguous.

We have an unambiguous code.

What can we encode such that we have few large numbers?

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

Document identifiers.

- Longer documents occur more often in an index.

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

Document identifiers.

- Longer documents occur more often in an index.

Differences between document identifiers in a posting list.

- Document identifiers grow but distances between are on average the same.
- Differences in document identifiers are mostly small numbers.
- Delta encoding: Encode differences between document numbers (d-gaps)

Compression

Delta Encoding

Posting list of document ids.

- 1, 5, 9, 18, 23, 24, 30, 44, 45, 48

Differences between adjacent numbers (d-gaps).

- 1, 4, 4, 9, 5, 1, 6, 14, 1, 3

Ordered list of (large) numbers turns into ordered list of small numbers.

- We can still do better than Elias-γ when we have large gaps.

To improve coding of large numbers, use Elias-δ code.

- Instead of encoding k_d in unary, we encode $k_d + 1$ using Elias-γ
- Takes approximately $2 \log_2 k + \log_2 k$ bits (as opposed to $2\lfloor \log_2 k \rfloor + 1$).

Compression

Elias-δ Code

Split k_d into: $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$ and $k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$

- Encode: k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

- Decode: Count the L ones until first zero (k_{dd}); read another L bits after the zero (k_{dr}); decode all using Elias-γ decoding (N). Decode remaining bits as binary (k_r). Final decoded value is $(N - 1) + k_r$.
- Produces longer encodings of small numbers than Elias-γ (<16, same space between 16 and 32)
- Produces shorter encodings of large numbers than Elias-γ (>32)

Compression

Elias-δ Code