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

	d_1	d_2	d_3	d_4	d_5	$\cdots$
t_1						
t_2						
t_3						
t_4						
t_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	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 $O(\log \mathit{df}(t, D))$.
- An effective amount of skip entries has been found to be $\sqrt{\mathit{df}(t, D)}$.
  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 $O(\log \mathit{df}(t, D))$.
- An effective amount of skip entries has been found to be $\sqrt{\mathit{df}(t, D)}$.
  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**
$\vdots$	
t_i	$2, 4$ · $4, 9$ · $8, 2$ · $16, 1$ · $19, 7$ · $23, 5$ · $28, 6$ · $41, 8$ · $50, 6$ · $77, 8$ · $\ldots$
t_j	$1, 1$ · $2, 3$ · $3, 5$ · $5, 2$ · $8, 17$ · $41, 6$ · $51, 5$ · $60, 5$ · $71, 3$ · $77, 2$ · $\ldots$
$\vdots$	

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 $\boxed{k, \textit{tf}(t, d_k)}$ 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮											

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

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

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.

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. $R = list();\quad x_1 = L_1.head;\quad x_2 = L_2.head$
2. **WHILE** $x_1 \neq NIL$ **AND** $x_2 \neq NIL$ **DO**
3.   **IF** $x_1.key == x_2.key$ **THEN**
4. $R = Insert(R, merge(x_1, x_2))$
5. $x_1 = x_1.next;\quad x_2 = x_2.next$
6.   **ENDIF**
7.   **WHILE** $x_1 \neq NIL$ **AND** $x_2 \neq NIL$ **AND** $x_1.key < x_2.key$ **DO**
8.   **IF** $CanSkip(x_1, x_2.key)$ **THEN**
9. $x_1 = Skip(x_1, x_2.key)$
10.   **ELSE**
11. $x_1 = x_1.next$
12.   **ENDIF**
13.   **ENDDO**

⋮   Like lines 7-13 with x_1 and x_2 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. $R = \mathit{list}();\quad x_1 = L_1.\mathit{head};\quad x_2 = L_2.\mathit{head}$
2. **WHILE** $x_1 \neq \mathit{NIL}$ **AND** $x_2 \neq \mathit{NIL}$ **DO**
3.   **IF** $x_1.\mathit{key} == x_2.\mathit{key}$ **THEN**
4. $R = \mathit{Insert}(R, \mathit{merge}(x_1, x_2))$
5. $x_1 = x_1.\mathit{next};\quad x_2 = x_2.\mathit{next}$
6.   **ENDIF**

⋮   Like lines 14-20 with x_1 and x_2 exchanged.

14.   **WHILE** $x_1 \neq \mathit{NIL}$ **AND** $x_2 \neq \mathit{NIL}$ **AND** $x_2.\mathit{key} < x_1.\mathit{key}$ **DO**
15.   **IF** $\mathit{CanSkip}(x_2, x_1.\mathit{key})$ **THEN**
16. $x_2 = \mathit{Skip}(x_2, x_1.\mathit{key})$
17.   **ELSE**
18. $x_2 = x_2.\mathit{next}$
19.   **ENDIF**
20.   **ENDDO**
21. **ENDDO**
22. *return*(R)

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textit{tf}(t, d_k)}$, 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	$\boxed{2, 4}$ $\boxed{4, 9}$ $\boxed{8, 2}$ $\boxed{16, 1}$ $\boxed{19, 7}$ $\boxed{23, 5}$ $\boxed{28, 6}$ $\boxed{41, 8}$ $\boxed{50, 6}$ $\boxed{77, 8}$ $\ldots$
t_j	$\boxed{1, 1}$ $\boxed{2, 3}$ $\boxed{3, 5}$ $\boxed{5, 2}$ $\boxed{8, 17}$ $\boxed{41, 6}$ $\boxed{51, 5}$ $\boxed{60, 5}$ $\boxed{71, 3}$ $\boxed{77, 2}$ $\ldots$
⋮	

Execute *IntersectTwo*(L_i, L_j).

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textbf{\textit{tf}}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮											

Execute *IntersectTwo*(L_i, L_j).

Result $R = ()$

Query Processing I

List Intersection: Example

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

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

Execute *IntersectTwo*(L_i, L_j).

Result $R = ()$

Query Processing I

List Intersection: Example

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

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

Execute *IntersectTwo*(L_i, L_j).

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

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textit{tf}(t, d_k)}$, 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 $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮				x_j							

Execute *IntersectTwo*(L_i, L_j).

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

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮					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 $\boxed{k, \textit{tf}(t, d_k)}$, two postlists L_i, L_j for terms t_i, t_j, and the query $q = t_i \wedge t_j$:

T	**Postings**										
$\vdots$				x_i							
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
$\vdots$						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 $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\dots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\dots$
⋮						x_j					

Execute *IntersectTwo*(L_i, L_j).

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

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮						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 $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮							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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮							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	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 $R = \boxed{2, ...}\ \boxed{8, ...}\ \boxed{41, ...}\ \boxed{77, ...}$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textit{tf}(t, d_k)}$, 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	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$
t_j	$1, 1$	$2, 3$	$3, 5$	$5, 2$	$8, 17$	$41, 6$	$51, 5$	$60, 5$	$71, 3$	$77, 2$	$\ldots$
⋮											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.

$H =$ *BuildMinHeap*$(L_1, \ldots, L_n)$;

$R =$ *ExtractMin*(H)

WHILE $|H| > 0$ **DO**

  $L_{\text{min}} =$ *ExtractMin*(H)

  $R =$ *IntersectTwo*(R, L_{min})

ENDDO

return(R)

Why are lists intersected in ascending order of list length?

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.

$H =$ *BuildMinHeap*$(L_1, \ldots, L_n)$;

$R =$ *ExtractMin*(H)

WHILE $|H| > 0$ **DO**

  $L_{\text{min}} =$ *ExtractMin*(H)

  $R =$ *IntersectTwo*(R, L_{min})

ENDDO

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$_1$ cannot$_2$ end$_3$ a$_4$ sentence$_5$ with$_6$ because$_7$ because$_8$ because$_9$ is$_{10}$ a$_{11}$ conjunction$_{12}$.”

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”}.

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 A_1.length DO
3.    R' = list()
4.    FOR j = 1 TO A_2.length DO
5.      IF |A_1[i] - A_2[j]| <= ε THEN
6.        insert(R', A_2[j])
7.      ELSE IF A_2[j] > A_1[i] THEN
8.        break
9.      ENDIF
10.   ENDDO
11.   insert(R, A_1[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`	$\ldots$ $\boxed{4, 250, (..., 133, 137, ...)}$ $\ldots$
`be`	$\ldots$ $\boxed{4, 125, (..., 134, 138, ...)}$ $\ldots$
`or`	$\ldots$ $\boxed{4, 40, (..., 135, ...)}$ $\ldots$
`not`	$\ldots$ $\boxed{4, 15, (..., 136, ...)}$ $\ldots$

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`	...	$4, 250, (..., 133, 137, ...)$	...
`be`	...	$4, 125, (..., 134, 138, ...)$	...
`or`	...	$4, 40, (..., 135, ...)$	...
`not`	...	$4, 15, (..., 136, ...)$	...

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 $O(\sum_{d \in \mathit{IntersectMany}(L_t : t \in q)} |d|)$.

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`	$\ldots$ $\boxed{4, 80, (..., 133, 137, ...)}$ $\ldots$
`be or`	$\ldots$ $\boxed{4, 55, (..., 134, ...)}$ $\ldots$
`or not`	$\ldots$ $\boxed{4, 20, (..., 135, ...)}$ $\ldots$
`not to`	$\ldots$ $\boxed{4, 7, (..., 136, ...)}$ $\ldots$

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`	...	4, 80, (..., 133, 137, ...)	...
`be or`	...	4, 55, (..., 134, ...)	...
`or not`	...	4, 20, (..., 135, ...)	...
`not to`	...	4, 7, (..., 136, ...)	...

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

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.

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 $\omega_X : T \times X \rightarrow \mathbf{R}$ 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.

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 $\omega_X : T \times X \rightarrow \mathbf{R}$ 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)
$\vdots$	
t_i	$2, 4$ · $4, 9$ · $8, 2$ · $16, 1$ · $19, 7$ · $23, 5$ · $28, 6$ · $41, 8$ · $50, 6$ · $77, 8$ · $\ldots$
t_j	$1, 1$ · $2, 3$ · $3, 5$ · $5, 2$ · $8, 17$ · $41, 6$ · $51, 5$ · $60, 5$ · $71, 3$ · $77, 2$ · $\ldots$
$\vdots$	

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	$4, 9$	$41, 8$	$77, 8$	$19, 7$	$28, 6$	$50, 6$	$23, 5$	$2, 4$	$8, 2$	$16, 1$	$\ldots$
t_j	$8, 17$	$41, 6$	$3, 5$	$51, 5$	$60, 5$	$2, 3$	$71, 3$	$5, 2$	$77, 2$	$1, 1$	$\ldots$
⋮											

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
  d = min_{i∈[1,m]}(x_i.key)
  d = Array(|q|)
  FOR i ∈ [1,m] DO
    IF x_i ≠ NIL AND x_i.key = d THEN
      d[i] = x_i.weight
    ENDIF
  ENDDO
  r = ρ(q, d)
  Insert(R, record(d, r))
  continue = FALSE
  FOR i ∈ [1,m] DO
    IF x_i ≠ NIL AND x_i.key = d THEN
      x_i = x_i.next
    ENDIF
    IF x_i ≠ NIL THEN
      continue = TRUE
    ENDIF
  ENDDO
ENDDO
return(R)
```

Example:

T	**Postings**					
$\vdots$						
t_i	1, 4	4, 9	8, 2	16, 1	19, 7	...
$\vdots$						
t_j	1, 1	2, 3	5, 5	7, 2	8, 8	...
$\vdots$						
t_k	1, 2	2, 4	5, 1	6, 3	8, 5	...
$\vdots$						

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

$R = \textit{PriorityQueue}()$
$x_1 = L_1.\textit{head}; \;\ldots;\; x_m = L_m.\textit{head}$
$\textit{continue} = \textit{TRUE}$
WHILE $\textit{continue}$ **DO**
  $d = \min_{i \in [1,m]}(x_i.\textit{key})$
  $\mathbf{d} = \textit{Array}(|q|)$
  FOR $i \in [1, m]$ **DO**
    IF $x_i \neq \textit{NIL}$ **AND** $x_i.\textit{key} = d$ **THEN**
      $\mathbf{d}[i] = x_i.\textit{weight}$
    ENDIF
  ENDDO
  $r = \rho(\mathbf{q}, \mathbf{d})$
  $\textit{Insert}(R, \textit{record}(d, r))$
  $\textit{continue} = \textit{FALSE}$
  FOR $i \in [1, m]$ **DO**
    IF $x_i \neq \textit{NIL}$ **AND** $x_i.\textit{key} = d$ **THEN**
      $x_i = x_i.\textit{next}$
    ENDIF
    IF $x_i \neq \textit{NIL}$ **THEN**
      $\textit{continue} = \textit{TRUE}$
    ENDIF
  ENDDO
ENDDO
$\textit{return}(R)$

Example:

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

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1, m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1, m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

$R = \textit{PriorityQueue}()$
$x_1 = L_1.\textit{head}; \quad \ldots; \quad x_m = L_m.\textit{head}$
$\textit{continue} = \textit{TRUE}$
WHILE $\textit{continue}$ **DO**
  $d = \min_{i \in [1,m]}(x_i.\textit{key})$
  $\mathbf{d} = \textit{Array}(|q|)$
  FOR $i \in [1, m]$ **DO**
    IF $x_i \neq \textit{NIL}$ **AND** $x_i.\textit{key} = d$ **THEN**
      $\mathbf{d}[i] = x_i.\textit{weight}$
    ENDIF
  ENDDO
  $r = \rho(\mathbf{q}, \mathbf{d})$
  $\textit{Insert}(R, \textit{record}(d, r))$
  $\textit{continue} = \textit{FALSE}$
  FOR $i \in [1, m]$ **DO**
    IF $x_i \neq \textit{NIL}$ **AND** $x_i.\textit{key} = d$ **THEN**
      $x_i = x_i.\textit{next}$
    ENDIF
    IF $x_i \neq \textit{NIL}$ **THEN**
      $\textit{continue} = \textit{TRUE}$
    ENDIF
  ENDDO
ENDDO
$\textit{return}(R)$

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	…
⋮						

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   d = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         d[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(q, d)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

Query Processing II

Disjunctive Queries

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

$R = \mathit{PriorityQueue}()$
$x_1 = L_1.\mathit{head}; \quad \ldots; \quad x_m = L_m.\mathit{head}$
$\mathit{continue} = \mathit{TRUE}$
WHILE $\mathit{continue}$ **DO**
    $d = \min_{i \in [1,m]}(x_i.\mathit{key})$
    $\mathbf{d} = \mathit{Array}(|q|)$
    FOR $i \in [1, m]$ **DO**
        IF $x_i \neq \mathit{NIL}$ **AND** $x_i.\mathit{key} = d$ **THEN**
            $\mathbf{d}[i] = x_i.\mathit{weight}$
        ENDIF
    ENDDO
    $r = \rho(\mathbf{q}, \mathbf{d})$
    $\mathit{Insert}(R, \mathit{record}(d, r))$
    $\mathit{continue} = \mathit{FALSE}$
    FOR $i \in [1, m]$ **DO**
        IF $x_i \neq \mathit{NIL}$ **AND** $x_i.\mathit{key} = d$ **THEN**
            $x_i = x_i.\mathit{next}$
        ENDIF
        IF $x_i \neq \mathit{NIL}$ **THEN**
            $\mathit{continue} = \mathit{TRUE}$
        ENDIF
    ENDDO
ENDDO
$\mathit{return}(R)$

Example:

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

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

Query Processing II

Disjunctive Queries

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

```
R = PriorityQueue()
x_1 = L_1.head;  ...;  x_m = L_m.head
continue = TRUE
WHILE continue DO
   d = min_{i∈[1,m]}(x_i.key)
   𝐝 = Array(|q|)
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         𝐝[i] = x_i.weight
      ENDIF
   ENDDO
   r = ρ(𝐪, 𝐝)
   Insert(R, record(d, r))
   continue = FALSE
   FOR i ∈ [1,m] DO
      IF x_i ≠ NIL AND x_i.key = d THEN
         x_i = x_i.next
      ENDIF
      IF x_i ≠ NIL THEN
         continue = TRUE
      ENDIF
   ENDDO
ENDDO
return(R)
```

Example:

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

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

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.

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. $R = \textit{map}()$
2. **FOR** $i \in [1, m]$ **DO**
3. $\quad x_i = L_i.\textit{head}$
4. $\quad$ **WHILE** $x_i \neq \textit{NIL}$ **DO**
5. $\quad\quad d = x_i.\textit{key}$
6. $\quad\quad w = x_i.\textit{weight}$
7. $\quad\quad R[d] = R[d] + \mathbf{q}[i] \cdot w$
8. $\quad\quad x_i = x_i.\textit{next}$
9. $\quad$ **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.

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

Index Construction

In-Memory Index Construction

Algorithm: Index Construction.

Input: $D = \{d_1, \ldots, d_n\}$. Set of documents $d_i = (t_1, \ldots, t_m)$ as lists of terms.

Output: Inverted index of D; postlist of term t_j contains postings $\boxed{i,\ \mathit{tf}(t_j, d_i)}$.

InMemoryIndex(D)

```
I = map()
FOR i ∈ [1, n] DO
  d_i = D[i];  T = set();  TF = map()
  FOR t ∈ d_i DO
    Insert(T, t)
    TF[t] = TF[t] + 1
  ENDDO
  FOR t ∈ T DO
    IF t ∉ I THEN I[t] = list() ENDIF
    L = I[t]
    posting = record(i, TF[t])
    InsertEnd(L, posting)
  ENDDO
ENDDO
return(I)
```

1. Initialization of an empty map as index I.
2. For each document d in D:
3. Collect d's terminology in T.
4. Accumulate d's term frequencies.
5. For each term t in d's terminology T:
6. Insert new posting $\boxed{i,\ \mathit{tf}(t_j, d_i)}$ for d in I.
7. Return index of D.

Index Construction

Index Merging

If the document collection D does not fit into main memory, indexing is done iteratively, sharding the document collection and merging the shard indexes similar to an external merge sort:

1. The *InMemoryIndex* procedure runs until main memory is full.
2. The postlists are written to disk in alphabetical order of terms.
3. Steps 1 and 2 are repeated until D is processed.
4. All k postlist files created are read concurrently, performing a k-way merge.

Index Construction

Index Merging

T	**Postings**					
⋮						
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
⋮						

T	**Postings**					
⋮						
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
⋮						

Index Construction

Index Merging

T	**Postings**					
$\vdots$	x_{i_1}					
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$	x_{i_2}					
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**
$\vdots$	
t_i	

Index Construction

Index Merging

T	**Postings**					
$\vdots$	x_{i_1}					
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$		x_{i_2}				
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**
$\vdots$	
t_i	$2, 4$

Index Construction

Index Merging

T	**Postings**					
$\vdots$		x_{i_1}				
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$		x_{i_2}				
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**	
$\vdots$		
t_i	$2, 4$	$4, 9$

Index Construction

Index Merging

T	**Postings**					
⋮		x_{i_1}				
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	...
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	...
⋮						

T	**Postings**					
⋮			x_{i_2}			
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	...
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	...
⋮						

T	**Postings**		
⋮			
t_i	$2, 4$	$4, 9$	$8, 2$

Index Construction

Index Merging

T	**Postings**					
$\vdots$						
		x_{i_1}				
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$						
				x_{i_2}		
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**			
$\vdots$				
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$

Index Construction

Index Merging

T	**Postings**					
$\vdots$			x_{i_1}			
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$				x_{i_2}		
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**				
$\vdots$					
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$

Index Construction

Index Merging

T	**Postings**					
$\vdots$						
				x_{i_1}		
t_i	4, 9	19, 7	23, 5	28, 6	50, 6	...
t_j	1, 1	3, 5	51, 5	60, 5	71, 3	...
$\vdots$						

T	**Postings**					
$\vdots$						
				x_{i_2}		
t_i	2, 4	8, 2	16, 1	41, 8	77, 8	...
t_j	2, 3	5, 2	8, 17	41, 6	77, 2	...
$\vdots$						

T	**Postings**					
$\vdots$						
t_i	2, 4	4, 9	8, 2	16, 1	19, 7	23, 5

Index Construction

Index Merging

T	Postings					
⋮						
					x_{i_1}	
t_i	4, 9	19, 7	23, 5	28, 6	50, 6	…
t_j	1, 1	3, 5	51, 5	60, 5	71, 3	…
⋮						

T	Postings					
⋮						
				x_{i_2}		
t_i	2, 4	8, 2	16, 1	41, 8	77, 8	…
t_j	2, 3	5, 2	8, 17	41, 6	77, 2	…
⋮						

T	Postings						
⋮							
t_i	2, 4	4, 9	8, 2	16, 1	19, 7	23, 5	28, 6

Index Construction

Index Merging

T	**Postings**					
$\vdots$						
					x_{i_1}	
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$						
					x_{i_2}	
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**							
$\vdots$								
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$

Index Construction

Index Merging

T	**Postings**					
$\vdots$						x_{i_1}
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
$\vdots$						

T	**Postings**					
$\vdots$					x_{i_2}	
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
$\vdots$						

T	**Postings**								
$\vdots$									
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$

Index Construction

Index Merging

T	**Postings**					
⋮						x_{i_1}
t_i	$4, 9$	$19, 7$	$23, 5$	$28, 6$	$50, 6$	$\ldots$
t_j	$1, 1$	$3, 5$	$51, 5$	$60, 5$	$71, 3$	$\ldots$
⋮						

T	**Postings**					
⋮						x_{i_2}
t_i	$2, 4$	$8, 2$	$16, 1$	$41, 8$	$77, 8$	$\ldots$
t_j	$2, 3$	$5, 2$	$8, 17$	$41, 6$	$77, 2$	$\ldots$
⋮						

T	**Postings**										
⋮											
t_i	$2, 4$	$4, 9$	$8, 2$	$16, 1$	$19, 7$	$23, 5$	$28, 6$	$41, 8$	$50, 6$	$77, 8$	$\ldots$

Index Construction

Index Merging

T	**Postings**					
⋮						
t_i	4, 9	19, 7	23, 5	28, 6	50, 6	...
t_j	1, 1	3, 5	51, 5	60, 5	71, 3	...
⋮						

T	**Postings**					
⋮						
t_i	2, 4	8, 2	16, 1	41, 8	77, 8	...
t_j	2, 3	5, 2	8, 17	41, 6	77, 2	...
⋮						

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	...
⋮											

Remarks:

- Alphabetical ordering of intermediary postlist files ensures that the index can be read sequentially, albeit concurrently, during merging. Compare to document-at-a-time scoring.
- If a term appears in only one of the indexes, its postlist is directly added to the merged index.
- Postings with skip pointers can be pre-determined before merging a postlist so that appropriate space can be allocated immediately, but the actual skip pointers need to be recomputed after the postlist is merged.
- The number k of intermediary postlist files that can be read concurrently without causing too much seeking overhead depends on the underlying hardware (e.g., k is smaller for spinning hard disks than for solid state disks). In case k is too large, the intermediary postlist files are merged in multiple passes, $k' < k$ at a time, until all are merged.

Index Construction

Distributed Indexing

If neither the document collection D, nor its index can be stored on a single machine, indexing must be performed distributed across a computer cluster.

Many cluster computing frameworks exist nowadays; the NoSQL movement, and ultimately the Big Data hype, was kicked off by Google's MapReduce [Dean 2004].

Index Construction

Distributed Indexing

If neither the document collection D, nor its index can be stored on a single machine, indexing must be performed distributed across a computer cluster.

Many cluster computing frameworks exist nowadays; the NoSQL movement, and ultimately the Big Data hype, was kicked off by Google's MapReduce [Dean 2004].

From a developer perspective, data processing with MapReduce boils down to implementing two procedures:

- Map: Given a key-value pair as input, it outputs a list of key-values pairs.
- Reduce: Given a key and the list of values output by map under that key, it outputs a key-value pair.

Index Construction

Distributed Indexing

If neither the document collection D, nor its index can be stored on a single machine, indexing must be performed distributed across a computer cluster.

Many cluster computing frameworks exist nowadays; the NoSQL movement, and ultimately the Big Data hype, was kicked off by Google's MapReduce [Dean 2004].

From a developer perspective, data processing with MapReduce boils down to implementing two procedures:

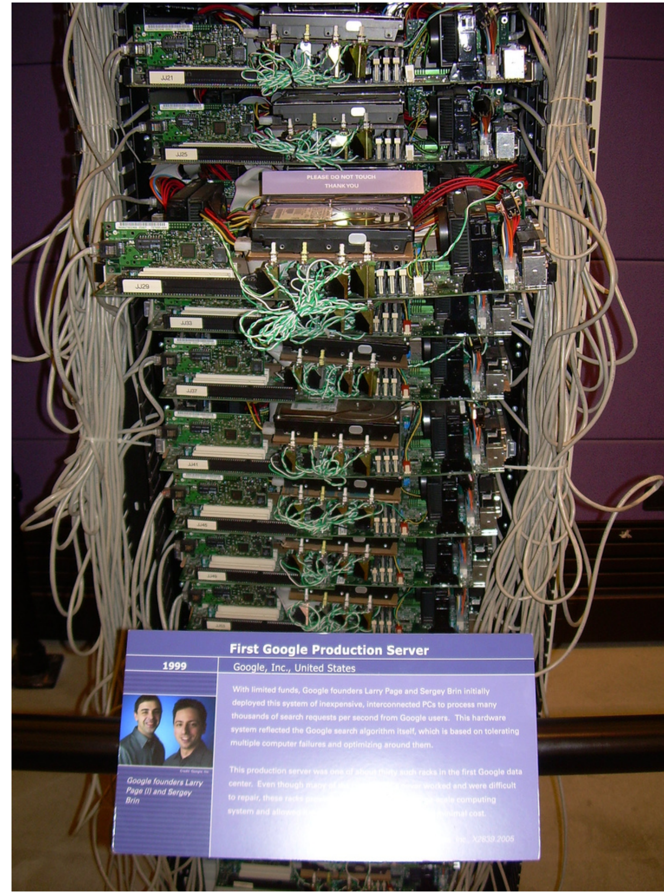
- Map: Given a key-value pair as input, it outputs a list of key-values pairs.
- Reduce: Given a key and the list of values output by map under that key, it outputs a key-value pair.

Example:

- IndexingMapper: Given a pair (i, d_i), where i is the document identifier of document d_i as input, output a pair (t, i) for every unique $t \in d_i$.
- DFReducer: Given $(t, [\ldots, i, \ldots])$ as input, output $(t, |[\ldots, i, \ldots]|) = (t, \mathit{df}(t, D))$.

Remarks:

- Computer clusters are often built from inexpensive commodity hardware. In the early days, desktop computers were used as Beowulf clusters, or dismantled and stacked. Google 1997 and 1999:



- The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine [Wikipedia].

- This framework is best-suited for problems that are embarrassingly parallel.

- The most widespread open source implementation is found in Apache Hadoop.

Index Construction

Distributed Indexing

Presuming the document collection is stored in a distributed document storage across the cluster, the execution of a MapReduce job divides into three basic phases:

- Map phase

  The map function is called in parallel on all cluster nodes and fed chunks of the data. Its output is recorded locally on each cluster node.

- Shuffle phase

  The output is transferred to a random cluster node chosen using a hash function, so that the same key is always transferred to the same cluster node. Once all data belonging to a key are on the same node, the values are sorted.

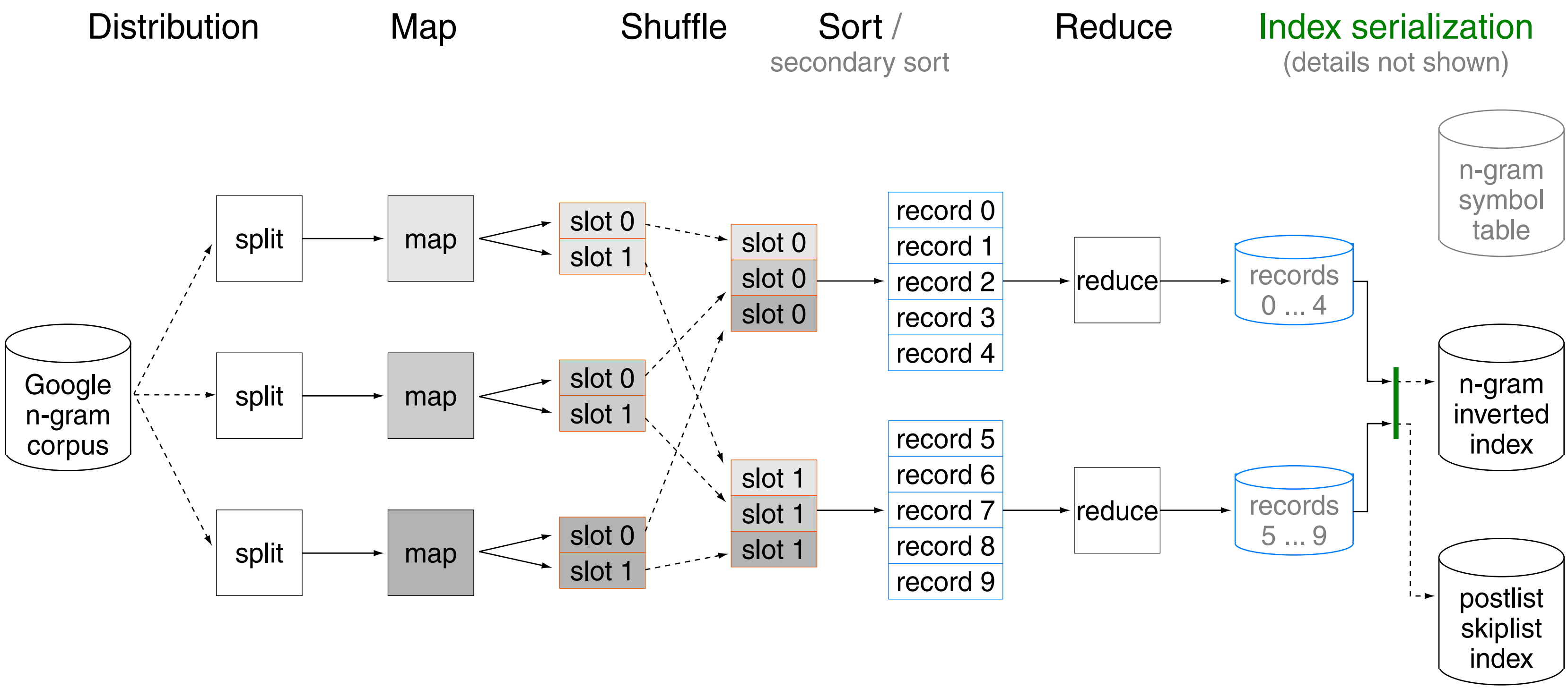
- Reduce phase

  The reduce function is called in parallel on all cluster nodes and fed the sorted lists, recording their output.

The map and reduce functions are idempotent: they are reexecuted in case of failures. To make optimal use of available resources, the framework may execute the same task more than once on different machines, retaining the first output that emerges (so-called speculative execution).

Index Construction

Distributed Indexing: Example Netspeak [www.netspeak.org]



Input records

(line number, line)

(1, `hello world` 712963 4)
(2, `hello kitty` 1468134 5)

Intermediate records

((word, len, pos), (freq, id))

((`hello`, 2, 0), (712963, 4))
((`world`, 2, 1), (712963, 4))
((`hello`, 2, 0), (1468134, 5))
((`kitty`, 2, 1), (1468134, 5))

Sorted records

((word, len, pos), ((freq, id) ... (freq, id)))

((`hello`, 2, 0), ((1468134, 5),(712963, 4)))
((`world`, 2, 1), ((712963, 4)))
((`kitty`, 2, 1), ((1468134, 5)))

Output records

(reduced sorted records)

Cluster-wide data copy
Local data copy

Index Construction

Index Updates

Document collections grow and change. Therefore, the index must be updated. The following strategies are applied:

- Index merging

  When new documents arrive in large numbers at a time, they are indexed and then the existing index is merged with the new one.

- Result merging

  When new documents arrive in small numbers at a time, a separate, small index is maintained and updated. Queries are processed against both the existing index and the small one containing the new arrivals, fusing the results.

- Deletions list

  Deletions are recorded in a deletions list, and deleted documents are removed from search results before results are shown.

Modifications are done by inserting a new document, and deleting the previous version.

Chapter IR:II

II. Indexing

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

Size Estimation

Query Result Set Size

tropical fish aquarium | Search

Web results Page 1 of 3,880,000 results

The total number of results is estimated, since web search engines typically do not explore the entire indexed document collection to compute the first page of results returned, but only a subset.

Approaches:

- Joint probability estimation
- Conditional probability estimation
- Initial result set-based estimation

Remarks:

- Example data from the GOV2 collection (collection size $|D|$ is 25,205,179):

Query	Document frequency
`aquarium`	26,480
`breeding`	81,885
`fish`	1,131,855
`lincoln`	771,326
`tropical`	120,990
`aquarium breeding`	1,848
`fish aquarium`	9,722
`fish breeding`	36,427
`tropical aquarium`	1,921
`tropical breeding`	5,510
`tropical fish`	18,472
`tropical fish aquarium`	1,529
`tropical fish breeding`	3,629

Size Estimation

Query Result Set Size: Joint Probability

Let $P_{df}(t)$ denote the probability of t occurring at least once in a document:

$$P_{df}(t) \;=\; \frac{df(t)}{|D|},$$

where D denotes the document collection of size $|D|$ and $df(t)$ the number of documents in D containing t, called document frequency.

Size Estimation

Query Result Set Size: Joint Probability

Let $P_{\mathit{df}}(t)$ denote the probability of t occurring at least once in a document:

$$P_{\mathit{df}}(t) \;=\; \frac{\mathit{df}(t)}{|D|},$$

where D denotes the document collection of size $|D|$ and $\mathit{df}(t)$ the number of documents in D containing t, called document frequency.

The result set size $\mathit{df}(q)$ of a query q of length $|q|$ terms can be estimated with

$$\mathit{df}(q) \;=\; |D| \cdot \prod_{i=1}^{|q|} P_{\mathit{df}}(t_i) \;=\; \frac{\prod_{i=1}^{|q|} \mathit{df}(t_i)}{|D|^{|q|-1}},$$

where t_i denotes the i-th term in q. This estimation presumes term independence.

Size Estimation

Query Result Set Size: Joint Probability

Let $P_{df}(t)$ denote the probability of t occurring at least once in a document:

$$P_{df}(t) \;=\; \frac{df(t)}{|D|},$$

where D denotes the document collection of size $|D|$ and $df(t)$ the number of documents in D containing t, called document frequency.

The result set size $df(q)$ of a query q of length $|q|$ terms can be estimated with

$$df(q) \;=\; |D| \cdot \prod_{i=1}^{|q|} P_{df}(t_i) \;=\; \frac{\prod_{i=1}^{|q|} df(t_i)}{|D|^{|q|-1}},$$

where t_i denotes the i-th term in q. This estimation presumes term independence.

Examples:

- *df*(`tropical fish aquarium`) $= 5.71$ actual: 1,529 documents
- *df*(`tropical fish breeding`) $= 17.65$ actual: 3,629 documents

Size Estimation

Query Result Set Size: Conditional Probability

By exploiting term co-occurrence information, we can obtain better estimates with

$$P_{\mathit{df}}(q) \;=\; P_{\mathit{df}}(t_1, t_2, t_3) \;=\; P_{\mathit{df}}(t_1, t_2) \cdot P_{\mathit{df}}(t_3 \mid t_1, t_2),$$

where $P_{\mathit{df}}(t_3 \mid t_1, t_2) \approx P_{\mathit{df}}(t_3 \mid t_x) = \max\{P_{\mathit{df}}(t_3 \mid t_1), P_{\mathit{df}}(t_3 \mid t_2)\}$ and $|q| = 3$. Recall that $P(A \mid B) = P(A, B)/P(B)$. Hence

$$\mathit{df}(q) \;=\; |D| \cdot P_{\mathit{df}}(q) \;=\; \frac{\mathit{df}(t_1, t_2) \cdot \mathit{df}(t_x, t_3)}{\mathit{df}(t_x)}.$$

Size Estimation

Query Result Set Size: Conditional Probability

By exploiting term co-occurrence information, we can obtain better estimates with

$$P_{\mathit{df}}(q) \;=\; P_{\mathit{df}}(t_1, t_2, t_3) \;=\; P_{\mathit{df}}(t_1, t_2) \cdot P_{\mathit{df}}(t_3 \mid t_1, t_2),$$

where $P_{\mathit{df}}(t_3 \mid t_1, t_2) \approx P_{\mathit{df}}(t_3 \mid t_x) = \max\{P_{\mathit{df}}(t_3 \mid t_1), P_{\mathit{df}}(t_3 \mid t_2)\}$ and $|q| = 3$. Recall that $P(A \mid B) = P(A, B)/P(B)$. Hence

$$\mathit{df}(q) \;=\; |D| \cdot P_{\mathit{df}}(q) \;=\; \frac{\mathit{df}(t_1, t_2) \cdot \mathit{df}(t_x, t_3)}{\mathit{df}(t_x)}.$$

- Queries of length $|q| = 2$ need not be estimated, anymore.
- Queries of length $|q| = 3$ are typically underestimated.
- Queries of length $|q| > 3$ still require estimations based on term independence, or storing higher-order co-occurrence information.

Examples:

- *df*($\texttt{tropical fish aquarium}$) $= 293$ actual: 1,529 documents
- *df*($\texttt{tropical fish breeding}$) $= 841$ actual: 3,629 documents

Size Estimation

Query Result Set Size: Initial Result Set-based Estimation

Let $D' \subset D$ denote the documents initially scored for a query q. Then the size of the total result set in D can be estimated with

$$\textit{df}(q) \;=\; |D_t| \cdot \frac{|D'_q|}{|D'|} \;=\; |D_t| \cdot \frac{|\{d \mid d \in D' \wedge q \in d\}|}{|D'|},$$

where t is the query term with the smallest subset $D_t \subset D$ of documents that contain t, and $D'_q \subset D'$ is the subset of the initially scored documents D' that contain all terms of q.

Size Estimation

Query Result Set Size: Initial Result Set-based Estimation

Let $D' \subset D$ denote the documents initially scored for a query q. Then the size of the total result set in D can be estimated with

$$\textit{df}(q) \;=\; |D_t| \cdot \frac{|D'_q|}{|D'|} \;=\; |D_t| \cdot \frac{|\{d \mid d \in D' \wedge q \in d\}|}{|D'|},$$

where t is the query term with the smallest subset $D_t \subset D$ of documents that contain t, and $D'_q \subset D'$ is the subset of the initially scored documents D' that contain all terms of q.

This estimation presumes relevant documents are uniformly distributed across all documents in D_t. Why does it usually overestimate the result set size?

Examples:

- With $D_{\texttt{aquarium}} = 26,480$, let $|D'| = 3,000$, and $|D'_q| = 258$:
- $\textit{df}(\texttt{tropical fish aquarium}) = 2,277$ actual: 1,529 documents
- With $D_{\texttt{breeding}} = 81,885$, let $|D'| = 3,000$, and $|D'_q| = 150$:
- $\textit{df}(\texttt{tropical fish breeding}) = 4,094$ actual: 3,629 documents

Size Estimation

Query Result Set Size: Initial Result Set-based Estimation

Let $D' \subset D$ denote the documents initially scored for a query q. Then the size of the total result set in D can be estimated with

$$\mathit{df}(q) \;=\; |D_t| \cdot \frac{|D'_q|}{|D'|} \;=\; |D_t| \cdot \frac{|\{d \mid d \in D' \wedge q \in d\}|}{|D'|},$$

where t is the query term with the smallest subset $D_t \subset D$ of documents that contain t, and $D'_q \subset D'$ is the subset of the initially scored documents D' that contain all terms of q.

This estimation presumes relevant documents are uniformly distributed across all documents in D_t. Overestimations result from D' containing the most "important" documents indexed. As $|D'|$ approaches $|D_w|$, estimations approach the true value.

Examples:

- With $D_{\texttt{aquarium}} = 26,480$, let $|D'| = 6,000$, and $|D'_q| = 402$:
- $\mathit{df}(\texttt{tropical fish aquarium}) = 1,774$ actual: 1,529 documents
- With $D_{\texttt{breeding}} = 81,885$, let $|D'| = 6,000$, and $|D'_q| = 276$:
- $\mathit{df}(\texttt{tropical fish breeding}) = 3,767$ actual: 3,629 documents

Size Estimation

Indexed Collection Size: Joint Probability-based

Most search engines are black boxes to outsiders, and many do not share the size of the document collection they index.

Size Estimation

Indexed Collection Size: Joint Probability-based

Most search engines are black boxes to outsiders, and many do not share the size of the document collection they index.

Given a web search engine, the size $|D|$ of the document collection D indexed can be estimated using two independently occurring terms t_1 and t_2:

$$P_{\mathit{df}}(t_1, t_2) = P_{\mathit{df}}(t_1) \cdot P_{\mathit{df}}(t_2) \quad \rightsquigarrow \quad |D| = \frac{\mathit{df}(t_1) \cdot \mathit{df}(t_2)}{\mathit{df}(t_1, t_2)}.$$

Averaging over many term pairs improves the estimate.

Example for GOV2:

- $\mathit{df}(\texttt{tropical}) = 120,990$,
  $\mathit{df}(\texttt{lincoln}) = 771,326$, and
  $\mathit{df}(\texttt{tropical, lincoln}) = 3018$.
- Then $|D| = 30,922,045$ actual: 25,205,179 documents

Size Estimation

Indexed Collection Size: Proportionality [van den Bosch 2016] [worldwidewebsize.com]

Given a web search engine, the size $|D|$ of the document collection D indexed can be estimated when presuming proportionality to a different reference collection D':

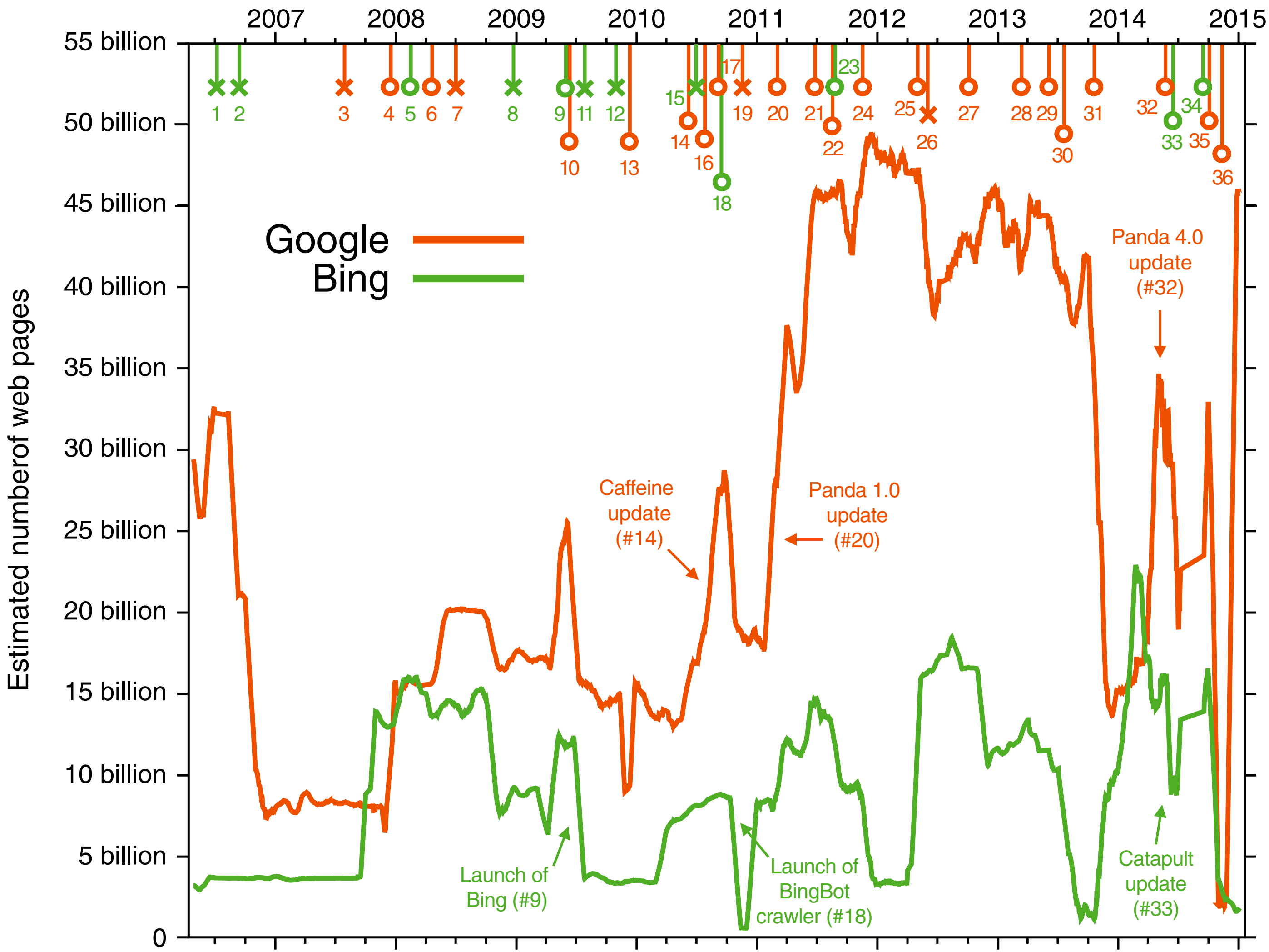
$$P_{\textit{df}}^{(D)}(t) = P_{\textit{df}}^{(D')}(t) \quad \leadsto \quad |D| = \frac{\textit{df}_D(t) \cdot |D'|}{\textit{df}_{D'}(t)},$$

where $\textit{df}_D$ computes the document frequency for D.

Averaging over terms of varying frequencies improves the estimate.

Size Estimation

Indexed Collection Size: Proportionality [van den Bosch 2016] [worldwidewebsize.com]



Remarks:

1. [2006-07-04] MSN Search outage
2. [2006-09-11] Launch of (improvements to) Live Search
3. [2007-07-31] Update to supplemental results indexing
4. [2007-12-18] No more supplemental index; whole index is searched for every query
5. [2008-02-12] Crawler improvements for Live Search
6. [2008-04-11] Improved crawling of HTML forms
7. [2008-06-30] Improved Flash indexing
8. [2008-12-11] First experiments with MSNBot 2.0
9. [2009-05-28] Launch of Bing
10. [2009-06-18] Improved Flash indexing
11. [2009-07-31] Bing and Yahoo! team up on search
12. [2009-11-04] MSNBot 2.0
13. [2009-12-07] Updates to real-time search
14. [2010-06-08] Launch of new web indexing system Caffeine
15. [2010-06-28] Experiments with BingBot crawler
16. [2010-07-29] Improved Flash & AJAX indexing
17. [2010-08-31] Google indexes SVG
18. [2010-09-03] Launch of BingBot crawler
19. [2010-11-11] Improved Flash indexing
20. [2011-02-24] Panda Refresh (update to promote (English) high-quality sites more)

Remarks: (continued)

21. [2011-06-21] Panda 2.2
22. [2011-08-12] Panda (rolled out to all languages)
23. [2011-08-15] Gradual roll-out of Tiger indexing architecture
24. [2011-11-03] Panda (update, affects 35% of queries)
25. [2012-04-24] Penguin update (targeting Web spam, impacting around 3.1% of queries)
26. [2012-05-26] Penguin 2 update (impacting less than 0.1% of queries)
27. [2012-10-05] Penguin 3 update (impacting around 0.3% of queries)
28. [2013-03-12] Panda update
29. [2013-05-22] Penguin 4 (v2.0, impacting 2.3% of queries)
30. [2013-07-18] Panda update
31. [2013-10-04] Penguin 5 (v2.1, impacting around 1% of queries)
32. [2014-05-21] Panda 4.0
33. [2014-06-18] Launch of Bing Catapult
34. [2014-09-09] Improved spam filtering
35. [2014-09-26] Panda 4.1 (3-5% of queries affected)
36. [2014-10-17] Penguin 6 (v3.0, impacting less than 1% English queries)