

Chapter IR:III

III. Indexing

- ❑ Indexing Basics
- ❑ Inverted Index
- ❑ Query Processing I
- ❑ Query Processing II
- ❑ Index Construction
- ❑ Index Compression

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	\dots
t_1						
t_2						
t_3						
t_4						
t_5						
\vdots						\dots

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	...
t_1						
t_2						
t_3						
t_4						
t_5						
⋮						...

□ 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	...
t_1	1					
t_2	1					
t_3	1					
t_4	0					
t_5	1					
⋮						...

□ 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	...
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	
⋮						...

□ 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	...
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	
⋮						...

□ 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	...
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						...

□ 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	...
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						...

□ 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	\dots
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						\dots

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 only depend on D are done offline.
- ❑ Online, given a query q , the term weights required are looked up to score documents.

Inverted Index

Term-Document Matrix

	d_1	d_2	d_3	d_4	d_5	\dots
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						\dots

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	→	Postings (Posting Lists, Postlists)
t_1	→	$d_1, w_{1,1}$ $d_2, w_{1,2}$
t_2	→	$d_1, w_{2,1}$ $d_2, w_{2,2}$ $d_4, w_{2,4}$
t_3	→	$d_1, w_{3,1}$ $d_2, w_{3,2}$ $d_4, w_{3,4}$ $d_5, w_{3,5}$
t_4	→	$d_2, w_{4,2}$
t_5	→	$d_1, w_{5,1}$
\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 $d_i, [\dots]$, so-called [postings](#)

Inverted Index

Data Structure

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

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

Design choices:

- ❑ Information stored in a posting $d_i, [\dots]$.
- ❑ 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 space at the expense of flexibility.

`<positions>`:

- ❑ Term positions within the document, i.e., 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 $O(\log df(t, D))$.
- ❑ First entry of a posting list, and then at random (or regular) intervals.
An effective amount of skip entries has been found to be $\sqrt{df(t, D)}$.

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 $O(\log df(t, D))$.
- ❑ First entry of a posting list, and then at random (or regular) intervals.
An effective amount of skip entries has been found to be $\sqrt{df(t, D)}$.

Inverted Index

Posting List, Postlist

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

T	Postings														
\vdots															
t_i	<table><tr><td>2, 4</td><td></td><td>4, 9</td><td>8, 2</td><td>16, 1</td><td>19, 7</td><td>23, 5</td><td>28, 6</td><td></td><td>41, 8</td><td>50, 6</td><td>77, 8</td><td></td><td>...</td></tr></table>	2, 4		4, 9	8, 2	16, 1	19, 7	23, 5	28, 6		41, 8	50, 6	77, 8		...
2, 4		4, 9	8, 2	16, 1	19, 7	23, 5	28, 6		41, 8	50, 6	77, 8		...		
t_j	<table><tr><td>1, 1</td><td></td><td>2, 3</td><td>3, 5</td><td>5, 2</td><td>8, 17</td><td>41, 6</td><td>51, 5</td><td></td><td>60, 5</td><td>71, 3</td><td>77, 2</td><td></td><td>...</td></tr></table>	1, 1		2, 3	3, 5	5, 2	8, 17	41, 6	51, 5		60, 5	71, 3	77, 2		...
1, 1		2, 3	3, 5	5, 2	8, 17	41, 6	51, 5		60, 5	71, 3	77, 2		...		
\vdots															

Ordering:

- ❑ by document identifier. Problem: good documents randomly distributed.
- ➔ by document quality. Early termination, but re-assigning IDs necessary.
- ➔ by term weight. Early termination, but renders 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 name “inverted index” is redundant: an index always maps terms to (parts of) documents where they occur. Better suited, but used less often, is “inverted file”, which conveys that a (document) file is inverted to form an index.
- ❑ Some retrieval models do not assign zero weights, but default to non-zero weights instead. Such weights can be omitted from an inverted index as well; they can be stored as a constant and used whenever a term weight for a given term-document pair is required that is not present in the inverted index.
- ❑ There is a tradeoff between the amount of information stored in a posting, and the time it takes to process a posting list in search of a document. The more information is stored in a posting, the more must be decoded or at least loaded into memory during postlist traversal.
- ❑ A skip entry may include more than one pointer, allowing for skip steps of various lengths.
- ❑ Dependent on the search scenario, constructing more than one index with different properties may be beneficial.

Chapter IR:III

III. Indexing

- ☐ Indexing Basics
- ☐ Inverted Index
- ☐ Query Processing I
- ☐ Query Processing II
- ☐ Index Construction
- ☐ Index Compression

Query Processing I

Retrieval Types

Query processing can be done according to two basic retrieval types:

- ❑ **Set retrieval**

A query induces a subset of the indexed documents which is considered relevant.

- ❑ **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 an intrinsically Boolean semantics, either implicitly implied by user behavior and expectations, or explicitly specified.

We distinguish four types:

- ❑ Single-term queries

- ❑ Disjunctive multi-term queries

Only Boolean OR connectives. Example: $\text{Antony} \vee \text{Brutus} \vee \text{Calpurnia}$.

- ❑ Conjunctive multi-term queries

Only Boolean AND connectives. Example: $\text{Antony} \wedge \text{Brutus} \wedge \text{Calpurnia}$.

- + Constraint: Proximity

Example: $\text{Antony} / \epsilon \text{Caesar}$

- + Constraint: Phrase

Example: “Antony and Caesar”

- ❑ “Complex” Boolean multi-term queries











Remainder of Boolean formulas. Example: $(\text{Antony} \vee \text{Caesar}) \wedge \neg \text{Calpurnia}$.

Can be normalized to disjunctive or conjunctive normal form.

Query Processing I

Conjunctive Multi-Term Queries

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

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

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 $[k, tf(t, d_k)]$ and a query $q = t_1 \wedge \dots \wedge t_n$, compute the collection $R \subseteq D$ of documents relevant to q .

T	Postings
\vdots	
t_i	<div>2,4</div> <div>4,9</div> <div>8,2</div> <div>16,1</div> <div>19,7</div> <div>23,5</div> <div>28,6</div> <div>41,8</div> <div>50,6</div> <div>77,8</div> <div>...</div>
t_j	<div>1,1</div> <div>2,3</div> <div>3,5</div> <div>5,2</div> <div>8,17</div> <div>41,6</div> <div>51,5</div> <div>60,5</div> <div>71,3</div> <div>77,2</div> <div>...</div>
\vdots	

- Problem: List Intersection.
- Instance: L_1, \dots, 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: Intersect 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 the x_1 .
5. While the key of x_2 is smaller than that of x_1 advance the x_2 .
6. Return R , once an iterator reaches the end of its list.

Query Processing I

List Intersection

Algorithm: Intersect 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.   $R = list(); x_1 = L_1.head; x_2 = L_2.head$ 
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.       $x_1 = x_1.next; x_2 = x_2.next$ 
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
     $\vdots$  Like lines 7-13 with  $i$  and  $j$  reversed.
21. ENDDO
22. return( $R$ )
```


Query Processing I

List Intersection

Algorithm: Intersect 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.   $R = list()$ ;  $x_1 = L_1.head$ ;  $x_2 = L_2.head$ 
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.       $x_1 = x_1.next$ ;  $x_2 = x_2.next$ 
6.    ENDIF
7.     $\vdots$  Like lines 14-20 with  $i$  and  $j$  reversed.
14.  WHILE  $x_1 \neq NIL$  AND  $x_2 \neq NIL$  AND  $x_2.key < x_1.key$  DO
15.    IF  $canSkip(x_2, x_1.key)$  THEN
16.       $x_2 = skip(x_2, x_1.key)$ 
17.    ELSE
18.       $x_2 = x_2.next$ 
19.    ENDIF
20.  ENDDO
21. ENDDO
22.  $return(R)$ 
```

Query Processing I

List Intersection: Example

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









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

Execute *IntersectTwo*(L_i, L_j).

Query Processing I

List Intersection: Example

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

T	Postings
\vdots	
t_i	<div>2, $\textcolor{violet}{4}$ </div> <div>4, $\textcolor{violet}{9}$</div> <div>8, $\textcolor{violet}{2}$</div> <div>16, $\textcolor{violet}{1}$ </div> <div>19, $\textcolor{violet}{7}$</div> <div>23, $\textcolor{violet}{5}$</div> <div>28, $\textcolor{violet}{6}$ </div> <div>41, $\textcolor{violet}{8}$</div> <div>50, $\textcolor{violet}{6}$</div> <div>77, $\textcolor{violet}{8}$ </div> <div>...</div>
t_j	<div>1, $\textcolor{violet}{1}$ </div> <div>2, $\textcolor{violet}{3}$</div> <div>3, $\textcolor{violet}{5}$</div> <div>5, $\textcolor{violet}{2}$ </div> <div>8, $\textcolor{violet}{17}$</div> <div>41, $\textcolor{violet}{6}$</div> <div>51, $\textcolor{violet}{5}$ </div> <div>60, $\textcolor{violet}{5}$</div> <div>71, $\textcolor{violet}{3}$</div> <div>77, $\textcolor{violet}{2}$ </div> <div>...</div>
\vdots	

Execute $IntersectTwo(L_i, L_j)$.

Result $R = ()$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, tf(t, d_k)}$ and 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	<div>2, 4</div> <div>4, 9</div> <div>8, 2</div> <div>16, 1</div> <div>19, 7</div> <div>23, 5</div> <div>28, 6</div> <div>41, 8</div> <div>50, 6</div> <div>77, 8</div> <div>...</div>
t_j	<div>1, 1</div> <div>2, 3</div> <div>3, 5</div> <div>5, 2</div> <div>8, 17</div> <div>41, 6</div> <div>51, 5</div> <div>60, 5</div> <div>71, 3</div> <div>77, 2</div> <div>...</div>
\vdots	x_j











Execute *IntersectTwo*(L_i, L_j).

Result $R = ()$

Query Processing I

List Intersection: Example

Given an index with postings $\boxed{k, \textcolor{violet}{tf}(t, d_k)}$ and 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	<div>2, $\textcolor{violet}{4}$  </div> <div>4, $\textcolor{violet}{9}$</div> <div>8, $\textcolor{violet}{2}$</div> <div>16, $\textcolor{violet}{1}$ </div> <div>19, $\textcolor{violet}{7}$</div> <div>23, $\textcolor{violet}{5}$</div> <div>28, $\textcolor{violet}{6}$ </div> <div>41, $\textcolor{violet}{8}$</div> <div>50, $\textcolor{violet}{6}$</div> <div>77, $\textcolor{violet}{8}$ </div> <div>...</div>
t_j	<div>1, $\textcolor{violet}{1}$  </div> <div>2, $\textcolor{violet}{3}$</div> <div>3, $\textcolor{violet}{5}$</div> <div>5, $\textcolor{violet}{2}$ </div> <div>8, $\textcolor{violet}{17}$</div> <div>41, $\textcolor{violet}{6}$</div> <div>51, $\textcolor{violet}{5}$ </div> <div>60, $\textcolor{violet}{5}$</div> <div>71, $\textcolor{violet}{3}$</div> <div>77, $\textcolor{violet}{2}$ </div> <div>...</div>
\vdots	x_j

Execute $IntersectTwo(L_i, L_j)$.

Result $R = \boxed{2, \textcolor{violet}{...}}$

Query Processing I

List Intersection: Example

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

Execute $IntersectTwo(L_i, L_j)$.

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

Query Processing I

List Intersection: Example

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

Execute $IntersectTwo(L_i, L_j)$.

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

Query Processing I

List Intersection: Example

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

Execute $IntersectTwo(L_i, L_j)$.

Result $R = \boxed{2, \textcolor{violet}{...}}$

Query Processing I

List Intersection: Example

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

<i>T</i>	Postings										
⋮											
	<i>x_i</i>										
<i>t_i</i>	2, 4	4, 9	8, 2	16, 1	19, 7	23, 5	28, 6	41, 8	50, 6	77, 8	...
<i>t_j</i>	1, 1	2, 3	3, 5	5, 2	8, 17	41, 6	51, 5	60, 5	71, 3	77, 2	...
⋮	<i>x_j</i>										

Execute *IntersectTwo*(*L_i*, *L_j*).

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

Query Processing I

List Intersection: Example

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

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

Execute $IntersectTwo(L_i, L_j)$.

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

Query Processing I

List Intersection: Example

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

<i>T</i>	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 $R = \boxed{2, \dots} \boxed{8, \dots} \boxed{41, \dots}$

Query Processing I

List Intersection: Example

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

<i>T</i>	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 $R = \boxed{2, \dots} \boxed{8, \dots} \boxed{41, \dots} \boxed{77, \dots}$

Query Processing I

List Intersection: Example

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

Execute $IntersectTwo(L_i, L_j)$.

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

Remarks:

- ❑ Postlists are typically too large to fit into main memory so that iterating them has performance benefits.
- ❑ The *key* attribute stores the document identifier stored in a posting.
- ❑ The *merge* function returns a posting merged from the two passed ones. The merge function has to reconcile the potentially stored term weights and other information stored in a posting.
- ❑ The *next* attribute stores the successive posting.
- ❑ The *CanSkip* function checks whether the current posting has skip information, and whether a target with a document identifier smaller than the one passed is available.
- ❑ The *Skip* function returns the posting closest, but smaller to the passed *key* value.

Query Processing I

List Intersection

Algorithm: Intersect Many Lists.

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

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

IntersectMany(L_1, \dots, L_n)

// Sort by list length.

1. $H = \text{BuildMinHeap}(L_1, \dots, L_n);$

2. $R = \text{ExtractMin}(H)$

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

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

5. $R = \text{IntersectTwo}(R, L_{\min})$

6. **ENDDO**

7. *return*(R)

Query Processing I

List Intersection

Algorithm: Intersect Many Lists.

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

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

IntersectMany(L_1, \dots, L_n)

// Sort by list length.

1. $H = \text{BuildMinHeap}(L_1, \dots, L_n);$

2. $R = \text{ExtractMin}(H)$

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

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

5. $R = \text{IntersectTwo}(R, L_{\min})$

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, \dots, 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 $\epsilon \geq 1$.

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 $\epsilon \geq 1$.

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 $\epsilon \geq 1$.

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.”
 1 2 3 4 5 6 7 8 9 10 11 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”, “is”, “a”, “because”}.

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.  $R = \text{map}()$ 
2. FOR  $i = 1$  TO  $A_1.\text{length}$  DO
3.    $R' = \text{list}()$ 
4.   FOR  $j = 1$  TO  $A_2.\text{length}$  DO
5.     IF  $|A_1[i] - A_2[j]| \leq \epsilon$  THEN
6.        $\text{insert}(R', A_2[j])$ 
7.     ELSE IF  $A_2[j] > A_1[i]$  THEN
8.        $\text{break}$ 
9.     ENDIF
10.     $j = j + 1$ 
11.  ENDDO
12.   $\text{insert}(R, A_1[i], R')$ 
13.   $i = i + 1$ 
14. ENDDO
15.  $\text{return}(R)$ 
```

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
Line 4 of *IntersectTwo* is wrapped into an if-statement to check if *IntersectPositions* returns a non-empty result.

Query Processing I

Phrasal Queries

Given a phrasal query $q = "t_1 \dots t_m"$, retrieve documents in which the terms t_1, \dots, t_m occur in the same order as in the query q .

Processing phrasal queries requires term positions in postings.

Example:

<i>T</i>	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

Phrasal Queries

Given a phrasal query $q = "t_1 \dots t_m"$, retrieve documents in which the terms t_1, \dots, t_m occur in the same order as in the query q .

Processing phrasal 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 phrasal queries can be reduced to the list intersection problem.
Algorithms `IntersectMany` and `IntersectTwo` can be adjusted to process phrasal queries.
- ❑ The runtime for query processing is in $O(\sum_{d \in D} |d|)$.

Query Processing I

Phrasal Queries

Given a phrasal query $q = "t_1 \dots t_m"$, retrieve documents in which the terms t_1, \dots, t_m occur in the same order as in the query q .

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

Example:

<i>T</i>	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.

How much faster can phrasal queries be processed?

Query Processing I

Phrasal Queries

Given a phrasal query $q = "t_1 \dots t_m"$, retrieve documents in which the terms t_1, \dots, t_m occur in the same order as in the query q .

To speed up phrasal 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 phrasal 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-phrasal queries as well.

Remarks:

- ❑ The space requirements of a positional index are 2-4 times that of a nonpositional index.
- ❑ Most retrieval models do not encode positional information, so that keyword proximity is used as an additional relevance signal or as prior probability for a document.

Query Processing II

Retrieval Types

Query processing can be done according to two basic retrieval types:

- ❑ **Set retrieval**

A query induces a subset of the indexed documents which is considered relevant.

- ❑ **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

Query Semantics for Ranked Retrieval

Keyword queries have an intrinsically Boolean semantics, either implicitly implied by user behavior and expectations, or explicitly specified.

We distinguish four types:

- ❑ Single-term queries
- ❑ Disjunctive multi-term queries
Only Boolean OR connectives. Example: $\text{Antony} \vee \text{Brutus} \vee \text{Calpurnia}$.
- ❑ Conjunctive multi-term queries
Only Boolean AND connectives. Example: $\text{Antony} \wedge \text{Brutus} \wedge \text{Calpurnia}$.
 - + Constraint: Proximity
Example: $\text{Antony} / \epsilon \text{Caesar}$
 - + Constraint: Phrase
Example: “Antony and Caesar”
- ❑ “Complex” Boolean multi-term queries
Remainder of Boolean formulas. Example: $(\text{Antony} \vee \text{Caesar}) \wedge \neg \text{Calpurnia}$.
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	...
t_j	1, 1	2, 3	3, 5	5, 2	8, 17	41, 6	51, 5	60, 5	71, 3	77, 2	...
\vdots											

In the worst case, the last document of the postlist is the most relevant one. Hence, 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)									
\vdots										
t_i	<div>4, 9<div><div></div><div></div><div></div></div></div> <div>41, 8<div><div></div><div></div><div></div></div></div> <div>77, 8<div><div></div><div></div><div></div></div></div> <div>19, 7<div><div></div><div></div><div></div></div></div> <div>28, 6<div><div></div><div></div><div></div></div></div> <div>50, 6<div><div></div><div></div><div></div></div></div> <div>23, 5<div><div></div><div></div><div></div></div></div> <div>2, 4<div><div></div><div></div><div></div></div></div> <div>8, 2<div><div></div><div></div><div></div></div></div> <div>16, 1<div><div></div><div></div><div></div></div></div> <div>...</div>									
t_j	<div>8, 17<div><div></div><div></div><div></div></div></div> <div>41, 6<div><div></div><div></div><div></div></div></div> <div>3, 5<div><div></div><div></div><div></div></div></div> <div>51, 5<div><div></div><div></div><div></div></div></div> <div>60, 5<div><div></div><div></div><div></div></div></div> <div>2, 3<div><div></div><div></div><div></div></div></div> <div>71, 3<div><div></div><div></div><div></div></div></div> <div>5, 2<div><div></div><div></div><div></div></div></div> <div>77, 2<div><div></div><div></div><div></div></div></div> <div>1, 1<div><div></div><div></div><div></div></div></div> <div>...</div>									
\vdots										

By definition of term weighting schemes, the document to which a term t is most important is the one with the highest term weight.

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

Query Processing II

Document Scoring

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.

The two most salient strategies for index-based document scoring are as follows:

❑ Document-at-a-time scoring

- Precondition: a total order of documents in the index's posting lists is enforced (e.g., ordering criterion document ID or rather document quality (IDs re-assigned)).
- Postlists of a query's terms are traversed in parallel to score one document at a time.
- 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

- Traverse postlists one a time (e.g., term ordering criterion frequency or importance).
- Maintain temporary query postlist, containing candidate documents.
- As each document's score accumulates, 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.

Query Processing II

Document Scoring

Algorithm: Document-at-a-time Scoring.

Input: L_1, \dots, L_m . The postlists of the terms t_1, \dots, 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($L_1, \dots, L_m, \mathbf{q}$)

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

Document Scoring

DAATScoring(L_1, \dots, L_m, q)

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

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 runtime of this approach can be poor (albeit, theoretically, 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

Document Scoring

Algorithm: Term-at-a-time Scoring.

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

Output: A sorted list of relevance scores for each document in D .

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

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

Remarks:

- ❑ TAAT = Term at a time
- ❑ Term-at-a-time scoring has a comparably high main memory load, since $|R| = |D|$. Otherwise, postlists are read consecutively, which suits rotating hard disks. Massive parallelization is possible.
- ❑ The order in which terms are processed (Line 3) 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, and would render skip lists useful. However, to speed up retrieval and allow for (unsafe) early termination, ordering by term weight is required.

Query Processing

Top-k Retrieval

The user of a search engine is only interested in the top-ranked k documents and will only look at those. All other documents retrieved and ranked will likely never be shown to the user.

Ideas for optimizing query processing:

- ❑ **Term score threshold**

Disregard terms from a query that have a significantly lower inverse document frequency than other terms from the same query (e.g., in term-at-a-time scoring); exceptions include stop word-heavy queries (to be or not to be).

- ❑ **Document score threshold**

In document-at-a-time scoring, once k documents have been found, determine which terms co-occur in documents exceeding the k -th most relevant document, and skip documents where the terms in question do not co-occur.

- ❑ **Early termination**

In indexes with postlists ordered by term weight, stop postlist traversal early, disregarding the rest of the postlist.

- ❑ **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

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**

Distributing postlists across local storage devices allows for parallelization, which pertains particularly to spinning hard disks.

- ❑ **Document distribution** (also: sharding)

Dividing the document collection into subsets by some criteria (so-called shards), and indexing each shard on a different index server adds a level of indirection: a query broker dispatches a query to index servers, which process a query as explained above. The broker fuses the results returned by index servers.

- ❑ **Tiered indexes**

Index tiers are distributed across index servers, and optionally across device types within a server: for example, the index of important shards (tier 1) may be kept in RAM at all times, whereas tier 2 shards are kept in flash memory, and tier 3 shards on spinning hard disks.

Query Processing

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:III

III. Indexing

- ☐ Indexing Basics
- ☐ Inverted Index
- ☐ Query Processing I
- ☐ Query Processing II
- ☒ Index Construction
- ☐ Index Compression

Index Construction

In-Memory Index Construction

Algorithm: Index Construction.

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

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

InMemoryIndex(D)

```
1.  $I = \text{map}()$ 
2. FOR  $i \in [1, n]$  DO
3.    $d_i = D[i]$ ;  $T = \text{set}()$ ;  $TF = \text{map}()$ 
4.   FOR  $t \in d_i$  DO
5.      $\text{Insert}(T, t)$ 
6.      $TF[t] = TF[t] + 1$ 
7.   ENDDO
8.   FOR  $t \in T$  DO
9.     IF  $t \notin I$  THEN  $I[t] = \text{list}()$  ENDIF
10.     $L = I[t]$ 
11.     $\text{posting} = \text{record}(i, TF[t])$ 
12.     $\text{InsertEnd}(L, \text{posting})$ 
13.   ENDDO
14. ENDDO
15.  $\text{return}(I)$ 
```

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 *Index* 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				
\vdots					
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					
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					

Index Construction

Index Merging

T	Postings								
\vdots	x_{i_1}								
t_i	<table><tr><td>4, 9</td><td></td><td>19, 7</td><td>23, 5</td><td>28, 6</td><td></td><td>50, 6</td><td>...</td></tr></table>	4, 9		19, 7	23, 5	28, 6		50, 6	...
4, 9		19, 7	23, 5	28, 6		50, 6	...		
t_j	<table><tr><td>1, 1</td><td></td><td>3, 5</td><td>51, 5</td><td>60, 5</td><td></td><td>71, 3</td><td>...</td></tr></table>	1, 1		3, 5	51, 5	60, 5		71, 3	...
1, 1		3, 5	51, 5	60, 5		71, 3	...		
\vdots									

T	Postings								
\vdots	x_{i_2}								
t_i	<table><tr><td>2, 4</td><td></td><td>8, 2</td><td>16, 1</td><td>41, 8</td><td></td><td>77, 8</td><td>...</td></tr></table>	2, 4		8, 2	16, 1	41, 8		77, 8	...
2, 4		8, 2	16, 1	41, 8		77, 8	...		
t_j	<table><tr><td>2, 3</td><td></td><td>5, 2</td><td>8, 17</td><td>41, 6</td><td></td><td>77, 2</td><td>...</td></tr></table>	2, 3		5, 2	8, 17	41, 6		77, 2	...
2, 3		5, 2	8, 17	41, 6		77, 2	...		
\vdots									

T	Postings
\vdots	
t_i	

Index Construction

Index Merging



















T	Postings								
\vdots	x_{i_1}								
t_i	<table><tr><td>4, 9</td><td></td><td>19, 7</td><td>23, 5</td><td>28, 6</td><td></td><td>50, 6</td><td>...</td></tr></table>	4, 9		19, 7	23, 5	28, 6		50, 6	...
4, 9		19, 7	23, 5	28, 6		50, 6	...		
t_j	<table><tr><td>1, 1</td><td></td><td>3, 5</td><td>51, 5</td><td>60, 5</td><td></td><td>71, 3</td><td>...</td></tr></table>	1, 1		3, 5	51, 5	60, 5		71, 3	...
1, 1		3, 5	51, 5	60, 5		71, 3	...		
\vdots									



















T	Postings							
\vdots	x_{i_2}							
t_i	<table><tr><td>2, 4</td><td></td><td>8, 2</td><td>16, 1</td><td>41, 8</td><td>77, 8</td><td>...</td></tr></table>	2, 4		8, 2	16, 1	41, 8	77, 8	...
2, 4		8, 2	16, 1	41, 8	77, 8	...		
t_j	<table><tr><td>2, 3</td><td></td><td>5, 2</td><td>8, 17</td><td>41, 6</td><td>77, 2</td><td>...</td></tr></table>	2, 3		5, 2	8, 17	41, 6	77, 2	...
2, 3		5, 2	8, 17	41, 6	77, 2	...		
\vdots								







T	Postings		
\vdots			
t_i	<table><tr><td>2, 4</td><td></td></tr></table>	2, 4	
2, 4			

Index Construction

Index Merging

T	Postings									
\vdots	x_{i_1}									
t_i	<table><tr><td>4, 9</td><td></td><td></td><td>19, 7</td><td>23, 5</td><td>28, 6</td><td></td><td>50, 6</td><td>...</td></tr></table>	4, 9			19, 7	23, 5	28, 6		50, 6	...
4, 9			19, 7	23, 5	28, 6		50, 6	...		
t_j	<table><tr><td>1, 1</td><td></td><td></td><td>3, 5</td><td>51, 5</td><td>60, 5</td><td></td><td>71, 3</td><td>...</td></tr></table>	1, 1			3, 5	51, 5	60, 5		71, 3	...
1, 1			3, 5	51, 5	60, 5		71, 3	...		
\vdots										

T	Postings									
\vdots	x_{i_2}									
t_i	<table><tr><td>2, 4</td><td></td><td></td><td>8, 2</td><td>16, 1</td><td>41, 8</td><td></td><td>77, 8</td><td>...</td></tr></table>	2, 4			8, 2	16, 1	41, 8		77, 8	...
2, 4			8, 2	16, 1	41, 8		77, 8	...		
t_j	<table><tr><td>2, 3</td><td></td><td></td><td>5, 2</td><td>8, 17</td><td>41, 6</td><td></td><td>77, 2</td><td>...</td></tr></table>	2, 3			5, 2	8, 17	41, 6		77, 2	...
2, 3			5, 2	8, 17	41, 6		77, 2	...		
\vdots										

T	Postings				
\vdots					
t_i	<table><tr><td>2, 4</td><td></td><td></td><td>4, 9</td></tr></table>	2, 4			4, 9
2, 4			4, 9		

Index Construction

Index Merging

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

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

T	Postings		
\vdots			
t_i	2, 4	4, 9	8, 2

Index Construction

Index Merging

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

T	Postings						
\vdots	x_{i_2}						
t_i	<div>2, 4</div>	<div></div>	<div>8, 2</div>	<div>16, 1</div>	<div>41, 8</div>	<div>77, 8</div>	...
t_j	<div>2, 3</div>	<div></div>	<div>5, 2</div>	<div>8, 17</div>	<div>41, 6</div>	<div>77, 2</div>	...
\vdots							

T	Postings				
\vdots					
t_i	<div>2, 4</div>	<div></div>	<div>4, 9</div>	<div>8, 2</div>	<div>16, 1</div>

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

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					
\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 28, 6

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	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
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	28, 6	41, 8	50, 6	

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	28, 6	41, 8	50, 6	77, 8	...

Index Construction

Index Merging

T	Postings					
\vdots						
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						
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	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	...	
\vdots												

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:

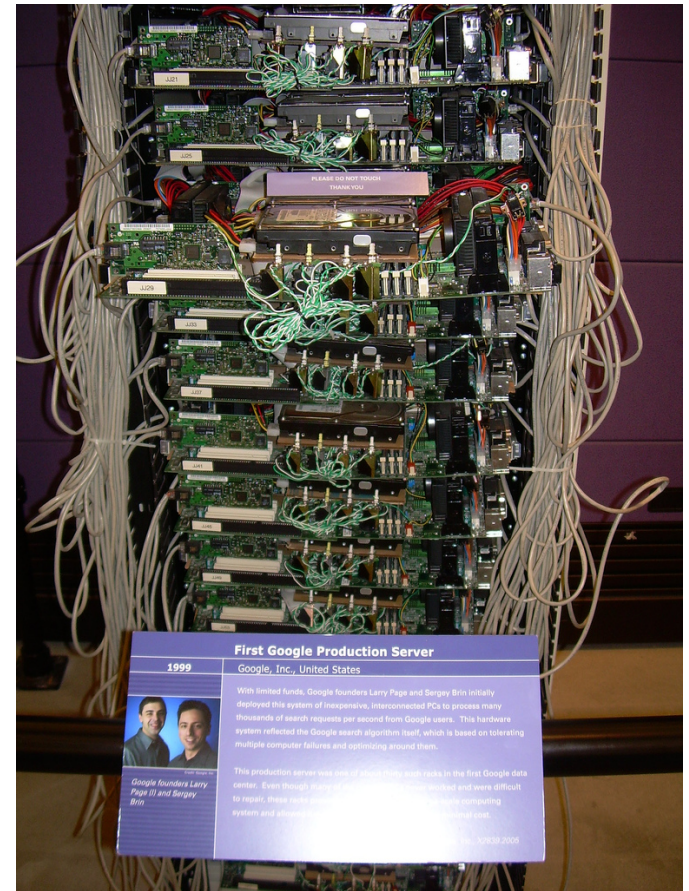
- ❑ Map: Given a key-value pair as input, it outputs a list of key-values pairs.
- ❑ Reduce: Given a key and the list of values output by map under that key, it outputs a key-value pair.

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 $t \in d_i$.
- ❑ **DF**Reducer: Given $(t, [\dots, i, \dots])$ as input, output $(t, |[\dots, i, \dots]|) = (t, \text{df}(t, D))$.

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.

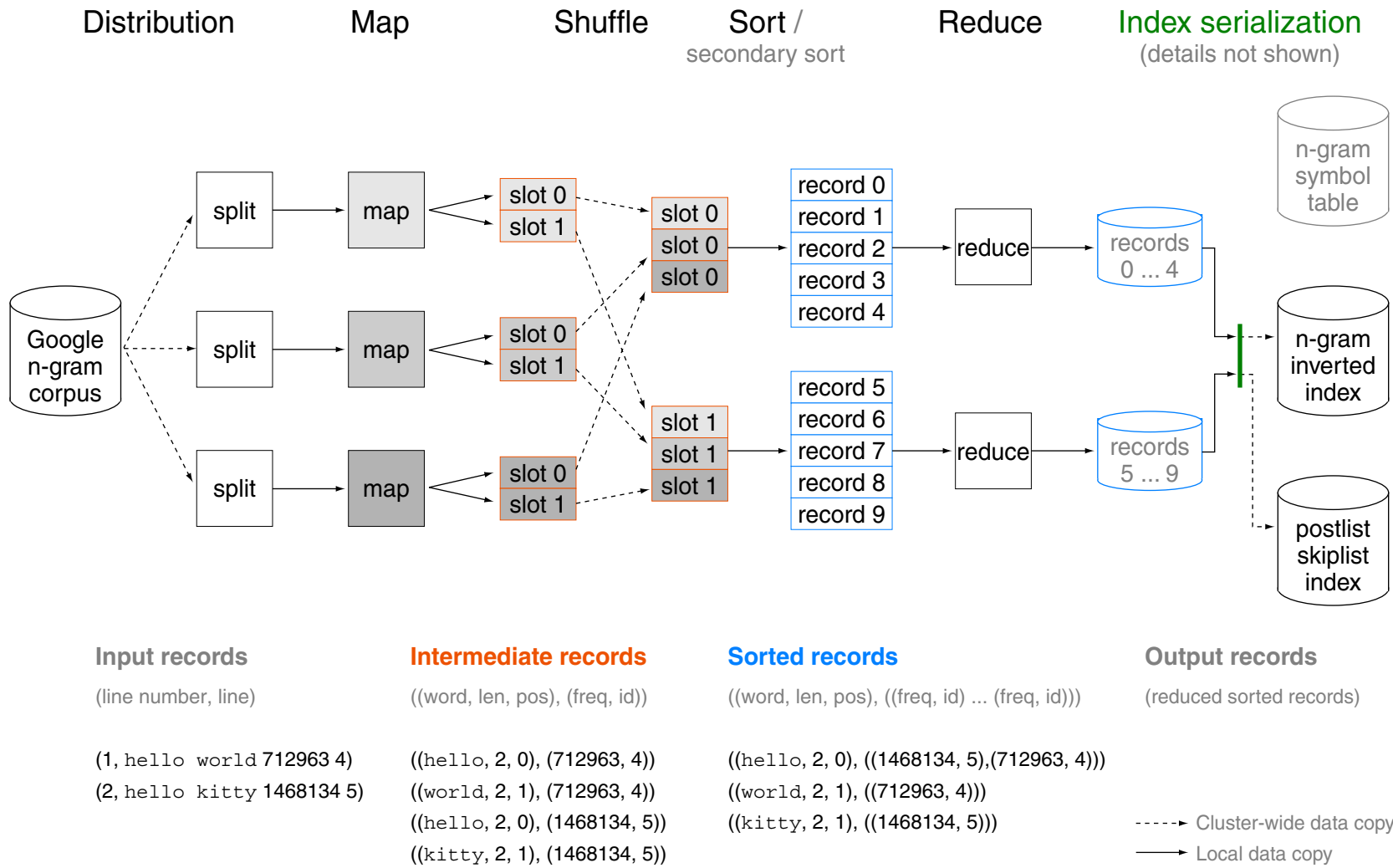
- ❑ **Reduce phase**

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 [\[www.netspeak.org\]](http://www.netspeak.org)



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.