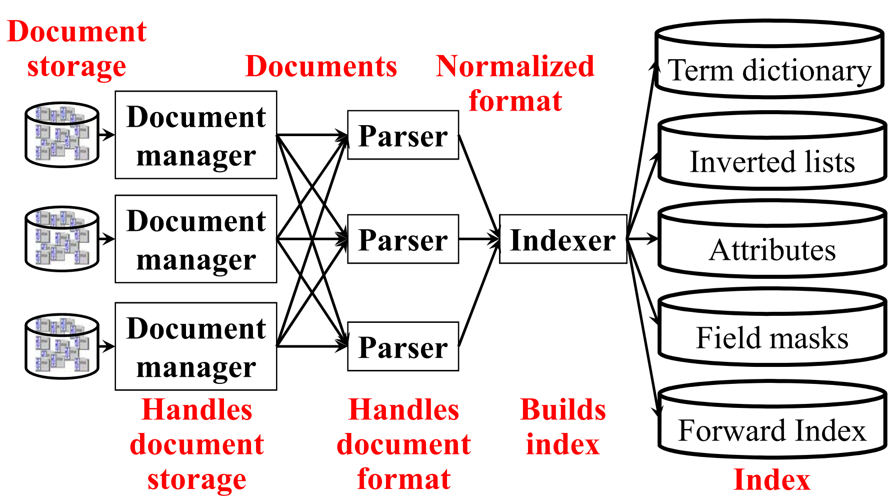
11-442 / 11-642:
Search Engines

Index Creation

Jamie Callan
Carnegie Mellon University
callan@cs.cmu.edu

Overview of Index Construction



Lecture Outline

- **Building inverted lists on a single processor**
- **Inverted lists and inverted files**
  - – Inverted list compression
  - – Inverted list optimizations
- **Forward indexes**
- **Storing document structure**
- **Indri and Lucene indexes**
- **Index updates**

Basic Facts That Affect Indexing

Indexing architectures are designed to be very efficient

- Quickly turn a massive document corpus into an index
- Efficiency is the reason for many design decisions

Integers are usually preferred over strings

- Integers almost always take less space to store
- Integers can be compared more quickly

So…convert strings to integers whenever possible

Basic Facts That Affect Indexing

Usually the corpus is <u>much</u> bigger than the available RAM

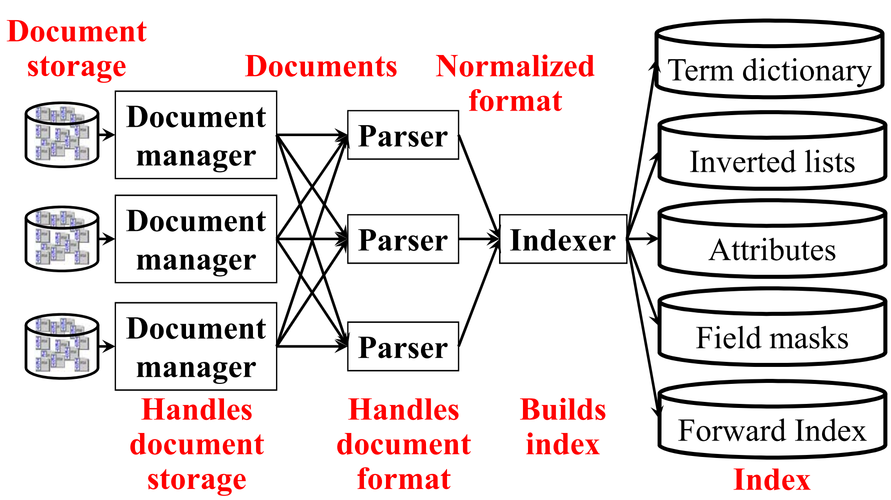
- E.g., 10 million web documents is 250 GB
- You can't do the whole task in memory

Disks are slow compared to processors

- Only write to the disk when absolutely necessary
- Compress data to reduce I/O
- Sequential access is much faster than random access

Most of this is true for all parts of the search engine
…but it is especially important during indexing

Overview of Index Construction



An Example Indexing API

Services that an indexing API might provide

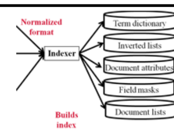
- documentBegin (externalId)
- documentEnd
- fieldBegin (fieldType)
- fieldBegin (fieldType, FieldId, parentFieldId)
- fieldEnd
- addAttribute (attributeType, value)
- addToken (tokenString, location)
- …

An Example Indexing API

Services that an indexing API might provide

- documentBegin (“http://www.cs.cmu.edu/~callan/”)
- documentEnd
- fieldBegin (TITLE)
- fieldBegin (TITLE, “21”, “18”)
- fieldEnd
- addAttribute (DATE, “09/30/2013”)
- addToken (“textbook”, 13)
- …

Indexing Text Tokens



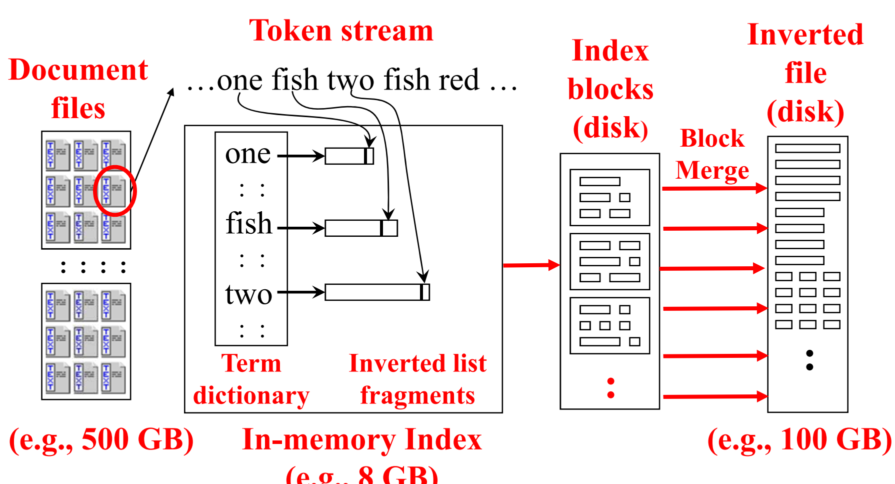
Most text representation tasks are done in the indexer

- Documents from different sources can be treated the same
- The indexer knows what data structures it needs

Typical text representation tasks (earlier lecture)

- Case folding (mixed case → lower case)
- Stopword removal
- Stemming (morphological processing / lemmatization)
- …

How Inverted Files are Built: Single Processor



(Adapted from Zobel & Moffat, 2006)

How Inverted Files are Built: Single Processor

The in-memory index buffers store

- Part of the term dictionary
- Fragments of inverted lists

The in-memory buffers are small compared to the final index

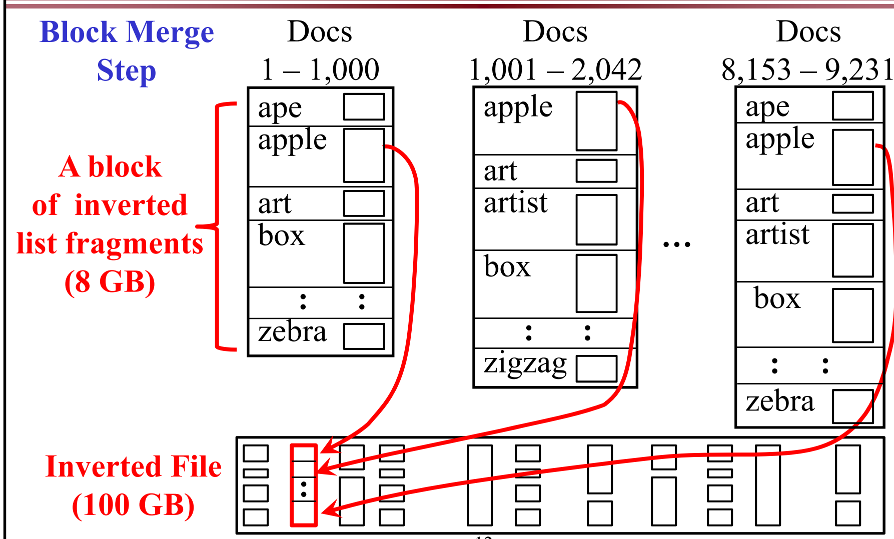
When in-memory buffers are full

- Flush in-memory buffers to disk and reinitialize them
- Continue parsing to refill the in-memory buffers

When all documents are parsed, merge index blocks on disk

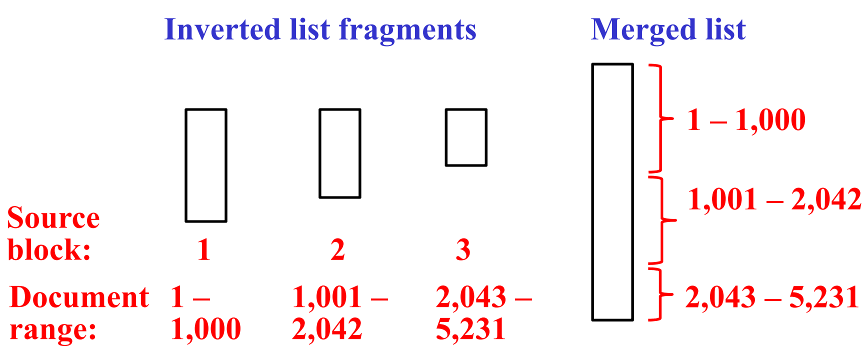
- Very fast – essentially a merge of sorted lists

How Inverted Files are Built: Single Processor



How Inverted Files are Built: Single Processor

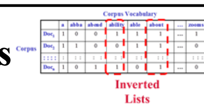
Block merge of the inverted list for 'apple'



Lecture Outline

- Building inverted lists on a single processor
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Storing document structure
- Indri and Lucene indexes
- Index updates

Different Types of Inverted Lists for the Term “Apple”



Binary

df:	4356
docid:	42
docid:	94
	:
	:

Frequency

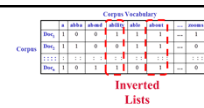
df:	4356
docid:	42
tf:	3
docid:	94
	:
	:

Positional

df:	4356
docid:	42
tf:	3
locs:	14
	83
	157
docid:	94
	:
	:

df: document frequency
tf: term frequency ($tf_{t,d}$)
doclen: document length (in terms)

Inverted Lists



Conceptually an inverted list looks like an object

apple

df:	**4356**
docid:	**42**
tf:	**3**
locs:	**14**
	83
	157
docid:	**94**
	:
	:

Usually it is stored on disk as a sequence of integers

apple

4356
42
3
14
83
157
94
:
:

Inverted List Indexes: Compression

Usually inverted lists are compressed – why?

- Save disk space? Favor aggressive compression algorithms
- Save time? Favor simple compression algorithms
  - I/O savings > CPU time required to uncompress

Today, the most common goal is to reduce query time

Algorithms:

- Gap encoding
- Restricted variable-length (RVL) encoding
- The book also covers slower, more aggressive algorithms

Inverted File Compression: Delta Gap ("DGap") Encoding

Store the differences between numbers ("DGaps")

- Increases probability of smaller numbers
- A more skewed distribution
- Lower entropy

Stemming also increases the probability of smaller numbers

- Why?

Before

Doc ID	121
TF	3
Loc	18
Loc	47
Loc	68
DocID	135
TF	2
Loc	22
Loc	35

After

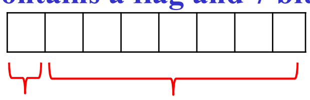
Doc ID	**121**
TF	3
Loc	18
Loc	29
Loc	21
DocID	**14**
TF	2
Loc	22
Loc	13

Inverted File Compression: Variable Byte Encoding

Variable byte encoding stores a number in a sequence of bytes

$byte_1$ **$byte_2$** …. **$byte_n$**

Each byte contains a flag and 7 bits of payload (the number)



The flag indicates whether this is the last byte in the sequence

0: Not the last byte 1: The last byte

Concatenate the payload bits to reconstruct the number

Inverted File Compression: Variable Byte Encoding

Decimal: 5

Binary: 00000000 00000000 00000000 $0\underbrace{0000101}_{\text{7 bits}}$

Compressed: 10000101

Decimal: 57

Binary: 00000000 00000000 00000000 $0\underbrace{0111001}_{\text{7 bits}}$

Compressed: 10111001

The flag identifies the last byte

Inverted File Compression: Variable Byte Encoding

					7 bits
Decimal:	127				
Binary:	00000000	00000000	00000000	01111111	
Compressed:	11111111				
			7 bits	7 bits	
Decimal:	128				
Binary:	00000000	00000000	00000000	10000000	
Compressed:	00000000	10000001			
	Byte$_0$	**Last byte**			
			7 bits	7 bits	
Decimal:	131				
Binary:	00000000	00000000	00000000	10000011	
Compressed:	00000011	10000001			

Inverted File Compression: Variable Byte Encoding

		7 bits	7 bits	7 bits
Decimal:	613,521			
Binary:	00000000	00001001	01011100	10010001
Compressed:	00010001	00111001	10100101	
	Byte$_0$	**Byte$_1$**	**Last byte**	

Inverted File Compression: Variable Byte Decoding

	$Byte_0$	$Byte_1$	$Byte_2$	
Compressed:	00010001	00111001	10100101	

Last? Payload

Initial:	00000000	00000000	00000000	00000000
After $Byte_0$:	00000000	00000000	00000000	00010001
After $Byte_1$:	00000000	00000000	00011100	10010001
After $Byte_2$:	00000000	00001001	01011100	10010001
Decimal:	613,521			

Inverted File Compression: Variable Byte Encoding

$[0 \ldots 2^7-1]$:	1 byte:	1xxxxxxx
$[2^7 \ldots 2^{14}-1]$:	2 bytes:	0xxxxxxx1xxxxxxx
$[2^{14} \ldots 2^{21}-1]$:	3 bytes:	0xxxxxxx0xxxxxxx1xxxxxxx
: :	:	: : :

Can store numbers of arbitrary size when needed

Advantages:

- Encoding and decoding can be done <u>very efficiently</u>
- Can find the n^th number without decoding the previous numbers

Inverted File Compression

There are other inverted list compression algorithms

- E.g., Gamma and Delta codes
  - See the textbook for details
  - **Note:** DGap Encoding ≠ Delta Code

The most effective compression algorithms are …

- About 15-20% smaller than variable byte encoding
- Slower than restricted variable length encoding

Disks are cheap, and speed is important

- So restricted variable length compression is a common solution

Inverted File Compression: Summary

A compressed inverted file, without positional information:

- Less than 10% the size of the original text

A compressed inverted file with positional information:

- 15-20% the size of the original text

Lecture Outline

- Building inverted lists on a single processor
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Storing document structure
- Indri and Lucene indexes
- Index updates

Inverted List Optimizations: Skip Lists

Skip lists are pointers that allow parts of the inverted list to be skipped

One heuristic

- List length: df_t
- A skip pointer every $\sqrt{df_t}$ documents

Purpose

- Avoid computation
- Avoid I/O

Inverted List With Skip Pointers

```
df   25
ctf  37
skip past doc 19
doc  3, tf 3 locs …
doc  7, tf 1, locs …
doc 10, tf 2, locs …
doc 13, tf 1, locs …
doc 19, tf 4, locs …
skip past doc 44
doc 23, tf 1, locs …
doc 27, tf 2, locs …
doc 32, tf 1, locs …
doc 41, tf 1, locs …
doc 44, tf 1, locs …
skip past doc 84
doc 57, tf 5, locs …
 :  :  :  :  :
```

Inverted List Optimizations: Skip Lists

When is a skip list useful? Consider #NEAR/3 (jamie apple)

42 ≠ 43, so advance the pointer with the smaller docid

jamie		apple	
df:	23	df:	1,033,436
→ docid:	42	→ docid:	43
tf:	3	tf:	3
docid:	59,356	docid:	49
tf:	1	tf:	1
:		:	

Document locations are not shown due to lack of space on the slide

29

Inverted List Optimizations: Skip Lists

When is a skip list useful? Consider #NEAR/3 (jamie apple)

59,356 ≠ 43, so advance the pointer with the smaller docid

- But advancing to the next 'apple' document is inefficient

jamie		apple	
df:	23	df:	1,033,436
docid:	42	→ docid:	43
tf:	3	tf:	3
→ docid:	59,356	docid:	49
tf:	1	tf:	1
:		:	

- Better to advance the 'apple' pointer to at least docid 59,356
- **Note:** QryIop.java has docIteratorAdvanceTo (docid)
  – Advance to docid, or beyond if docid isn't in the list
  – It would be easy to add skip lists to the QryEval code

30

Inverted List Optimizations: Skip Lists

Skip lists are useful for any query operator that needs all of its arguments to occur in a document

- #NEAR, #WINDOW, #SYN
- Boolean AND
- Document structure operators, e.g., jamie.title
  - A short jamie inverted list + a long title inverted list

Skipping can also occur when score calculations are complex

- Some query evaluation optimizations stop calculating a document score when it becomes obvious that the score is low
  - We may cover this later in the course, if there is time

Inverted List Optimizations : Top-Docs (Champion) Lists

Main idea

- Some inverted lists are long
- Most queries only need to return ≤ 100 documents
- Why rank all documents if only 100 are needed?

Top-docs lists:

- Truncated inverted lists that contain only the best docs
- Faster
- Lower recall

"apple" Inverted List

Doc 1 tf 2
Doc 2 tf 4
: : :
Doc 258392 tf 3
Doc 258393 tf 5
: : :
Doc 1025429 tf 6
Doc 1025430 tf 4

1,025,430 documents

"apple" Top-Docs List

Doc 1025429 tf 6
Doc 258393 tf 5
: : :
Doc 2 tf 4
Doc 1025430 tf 4

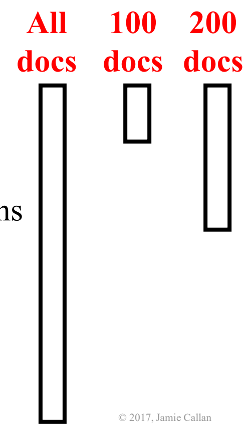
1,000 documents

Inverted List Optimizations : Top-Docs (Champion) Lists

How are top-docs lists constructed?

- **<u>Select</u> documents to go in the list by…**
  - tf
  - PageRank
  - …
- **<u>Order</u> the top-docs list by document id**
  - Faster, if the whole list is read
  - May require multiple lists of different lengths
- **<u>Order</u> the top-docs list by tf**
  - Supports reading variable amount of list
  - Requires just one list
- **…**

Inverted lists for "apple"



33

Inverted List Optimizations : Top-Docs (Champion) Lists

How many terms are frequent enough to have a top-docs list?

- **How big is an average inverted list entry?**
- **Suppose 5 integers, on average**
  - Uncompressed: 16 bytes
  - Compressed: Assume 5 bytes
    - » 30% compression rate × 16 bytes = 5 bytes
- **Linux filesystem page size:** 4096 bytes
- **How many term inverted list entries fit in one page?**
  - 4096 / 5 = 819
  - Assume some overhead … 800 entries

docid
tf
loc
…
loc

34

Inverted List Optimizations : Top-Docs (Champion) Lists

How many terms are frequent enough to have a top-docs list?

- **Zipf's Law:** Rank × Frequency$_c$ = A × N
- **Rank of a word that occurs once (ctf=1):** A × N / 1
  - Also an estimate of the vocabulary size
- **Rank of a word that occurs 800 times:** A × N / 800
- **The <u>percentage of terms</u> with ctf ≥ 800:**

  (A × N / 800) / (A × N) = 1 / 800 = 0.125%
- **So … if the vocabulary is 1,000,000 terms**
  - Probably fewer than 1,250 top-docs lists
  - Each list is perhaps 4-8 KB long
  - So … 5-10 MB

Ranking of terms by ctf

freq=800 →

freq=1 →

Inverted List Optimizations: Multiple Inverted Lists Per Term

Some common query operators do not use locations

- **Operators:** SUM, WEIGHT, AND, OR, ANDNOT, …
- Using inverted lists with locations causes wasted I/O

What are the I/O costs?

- **No location:** 2 integers per document
- **Locations:** Assume 1.5 extra integers per document

docid
tf
loc
…
loc
docid

So, 42% of the I/O is wasted effort

- Compression changes these numbers, but not the main idea

Inverted List Optimizations: Multiple Inverted Lists Per Term

Some common query operators do not use locations

- **Operators:** SUM, WEIGHT, AND, OR, ANDNOT, …
- Using inverted lists with locations causes wasted I/O

Solution: Store two types of inverted list for each term

I/O costs are reduced for queries that don't need locations

- **Cost:** Extra disk space

docid tf loc … loc docid

docid tf docid …

37

Inverted List Optimizations: Multiple Inverted Lists Per Term

Is it worthwhile to store 2 inverted lists for every term?

- **How many terms have a topdocs / champion list?**
  - Only 0.125%
  - All other terms have 'short' inverted lists that fit in one disk page
- **What is the effect of making a 'short' inverted list shorter?**
  - Reduced computational effort
  - Probably no change in I/O
- **So … maybe have 2 inverted lists only for frequent terms**

38

Lecture Outline

- Building inverted lists on a single processor
- Inverted lists and inverted files
  - Inverted list compression
  - Inverted list optimizations
- **Forward indexes**
- Storing document structure
- Indri and Lucene indexes
- Index updates

Forward Indexes

Suppose I want to know which words occur in documents about Microsoft … how would I do it?

- This is a common component of text mining tasks
- E.g., sentiment analysis of documents about Microsoft
- E.g., query expansion, relevance feedback

First step: Use an inverted list to find out which documents contain the word Microsoft

- Easy

Now what?

Forward Indexes

Sometimes your software needs to know what terms are in the document … how does it find out?

Parse the document again?

- A little slow (but not terrible, because indexing is fast)
- Done when storage is expensive
- But software evolves, so it might not be <u>exactly the same</u> parse

Store the parsed document?

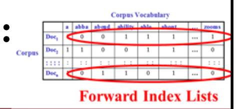
- Fast, and guaranteed to be accurate
- Done when storage is cheap

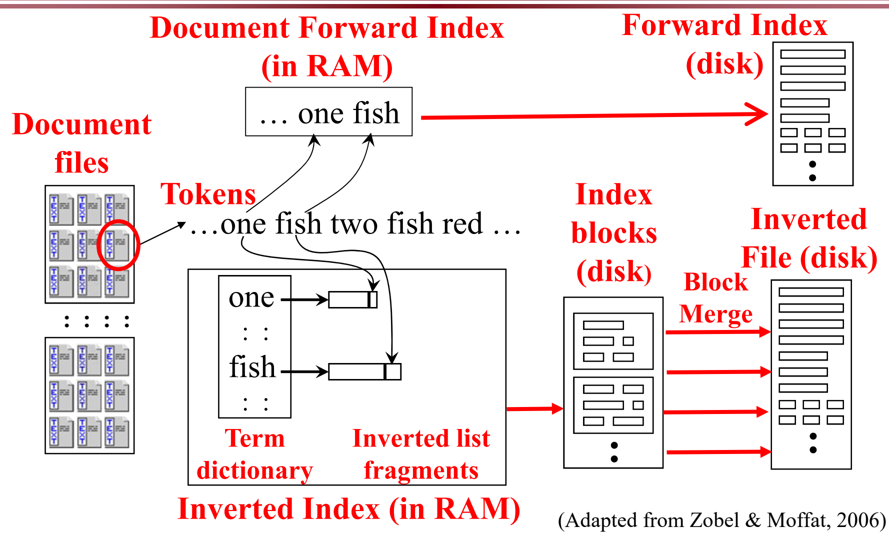
Forward Indexes

Forward indexes store the <u>indexed form of a document</u>

- **The location of every term that made it into the index**
  - Term id, location
  - Information about where stopwords appeared
- **Optionally: Information about document structure**
  - Field names and extents

How Forward Indexes are Built: Single Processor





(Adapted from Zobel & Moffat, 2006)

How Does Indri Do It? Two Different Approaches

Indri provides two classes for accessing forward indexes

- **Via the indri::index class**
  - A somewhat low-level access to the index
  - Very efficient, not always very friendly
- **Via the QueryEnvironment class**
  - Higher-level, more abstract access to the index
  - Somewhat less efficient, somewhat more user friendly

We start with the indri::index class because it exposes the data structures more clearly

How Does Indri Do It?
The indri::index::TermList Class

Text: "OBAMA STATE OF THE UNION SPEECH
President Barack Obama delivered …"

int terms [ ]

obama	41321
state	34127
OOV	0
OOV	0
union	25434
speech	9982
president	98476
barack	12653
obama	41321
deliver	34376
	: :

Term ids in the document.
0: stopword

45

How Does Indri Do It?
The indri::index::TermList Class

Text: "OBAMA STATE OF THE UNION SPEECH
President Barack Obama delivered …"

int terms [ ]

obama	41321
state	34127
OOV	0
OOV	0
union	25434
speech	9982
president	98476
barack	12653
obama	41321
deliver	34376
	: :

Term ids in the document.
0: stopword

fields [ ]

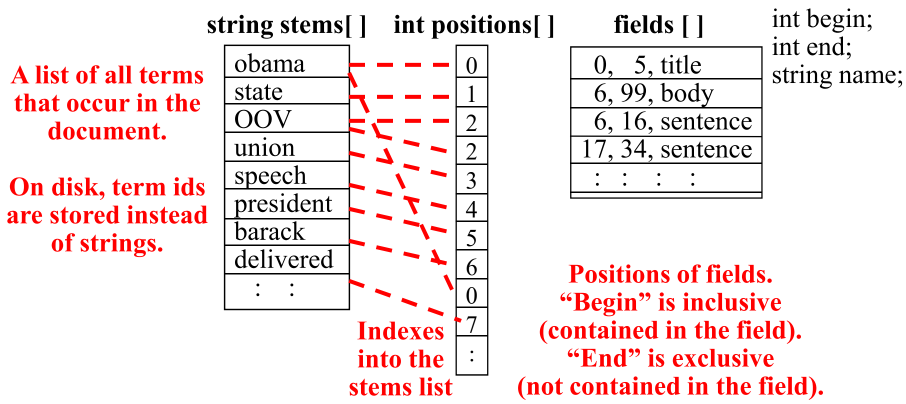
int begin;
int end;
int id;

0, 5, 3	**title**
6, 99, 4	**body**
6, 16, 18	**sentence**
17, 34, 18	**sentence**
: : :	: : :

46

How Does Indri Do It?
The DocVector Class

Text: "OBAMA STATE OF THE UNION SPEECH
President Barack Obama delivered …"



Lecture Outline

- Building inverted lists on a single processor
- Inverted lists and inverted files
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- **Storing document structure**
- Indri and Lucene indexes
- Index updates

For More Information

- I.H. Witten, A. Moffat, and T.C. Bell. "Managing Gigabytes." Morgan Kaufmann. 1999.
- G. Salton. "Automatic Text Processing." Addison-Wesley. 1989.
- J. Zobel and A. Moffat. "Inverted files for text search engines." *ACM Computing Surveys*, 38 (2). 2006.