11-442 / 11-642 / 11-742: Search Engines

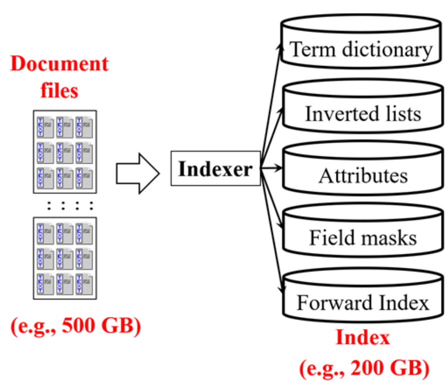
Index Creation

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

1

Lecture Outline

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



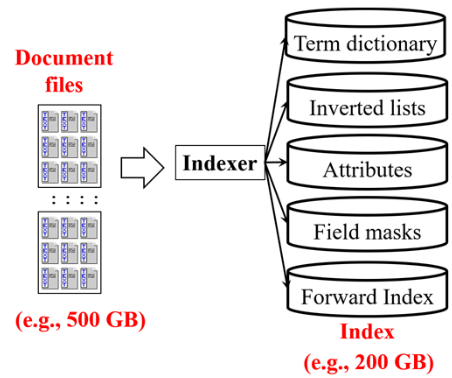
2

Basic Facts That Affect Indexing

Usually the corpus is much bigger than the available RAM
- E.g., 20 million web documents is 500 GB
- You can't do the whole task in memory

Disks are slow compared to processors
- Only use 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

The parser uses an API to communicate with the indexer
- There is no standard API
- We will take a quick look at what Lucene does

Lucene's Indexing API:
Configure the Inverted Index

Configure the lexical analyzer

```
EnglishAnalyzer analyzer = new EnglishAnalyzer ();
analyzer.setLowercase (true);
analyzer.setStopwordRemoval (true);
analyzer.setStemmer (EnglishAnalyzer.StemmerType.KSTEM);
```

Configure the inverted index

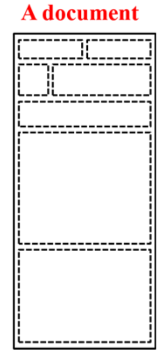
```
IndexWriterConfig iwc = new IndexWriterConfig(analyzer);
iwc.setOpenMode(OpenMode.CREATE);
IndexWriter writer = new IndexWriter(path, iwc);
```


Lucene's Indexing API:
Define Field Types

Define field types to control how different parts of the document are indexed

- Example: A field type for full-text fields (e.g., body, title)
  - Tokenize the content and store it in the inverted index and the forward index
  - Inverted lists contains docids, tf, positions
  - TermVectors (the forward index) contain positions

A document



```
FieldType fullText = new FieldType();
fullText.setTokenized(true);
fullText.setIndexOptions(
    IndexOptions.DOCS_AND_FREQS_AND_POSITIONS);
fullText.setStoreTermVectors(true);
fullText.setStoreTermVectorPositions(true);
```

Lucene's Indexing API: Document Objects

Start a loop over all available documents

Create document object

```
Document doc = new Document();
```


Lucene's Indexing API: Adding Fields to a Document Object

Create metadata field objects

f1= new StringField (“externalId”, “GX016-79-782”, Field.Store.YES);

f2 = new StringField (“PageRank”, “4.33”, Field.Store.YES);

Create content field objects

f3 = new Field (“title”, “Juice Lyrics”, fullText);

f4 = new Field (“body”, “It ain't my fault that I’m out here gettin' loose…”, fullText);

Add field objects to the document object

- doc.add (f1);
- doc.add (f2);
- …

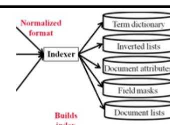
Lucene's Indexing API:
Indexing a Document Object

Add the document to the index

- IndexWriter.addDocument (doc);

End the loop over all available documents

Indexing Text Tokens



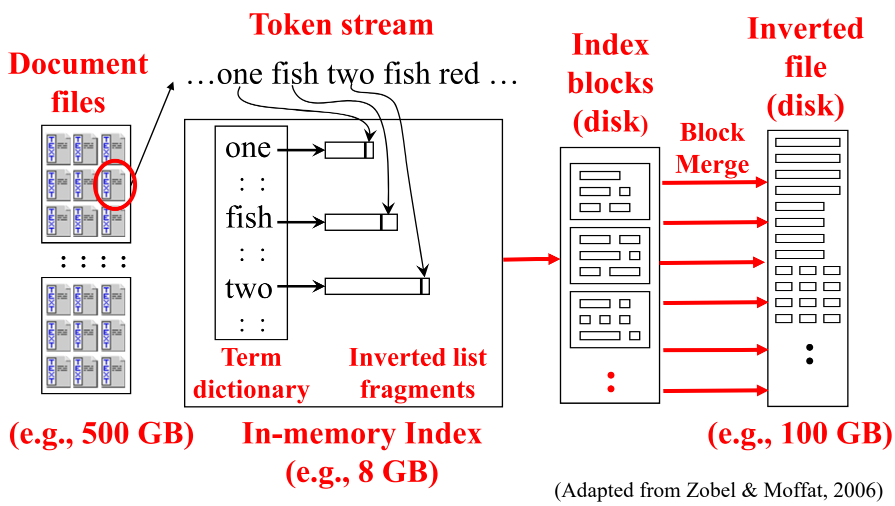
Most text representation tasks are done in the indexer

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

Typical text representation tasks (earlier lecture)

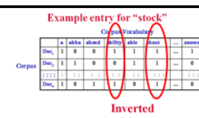
- Case folding (mixed case → lower case)
- Stopword removal
- Stemming (morphological processing / lemmatization)
- …

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

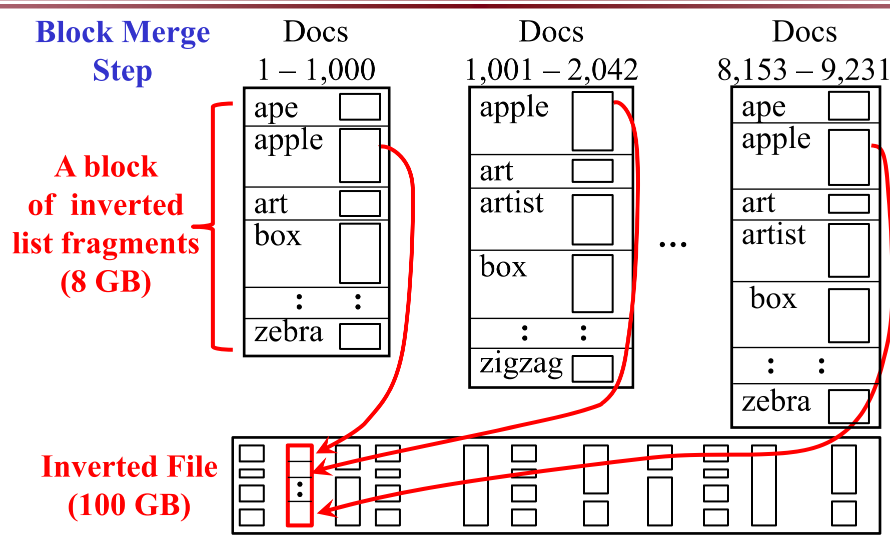
When in-memory buffers are full

- 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

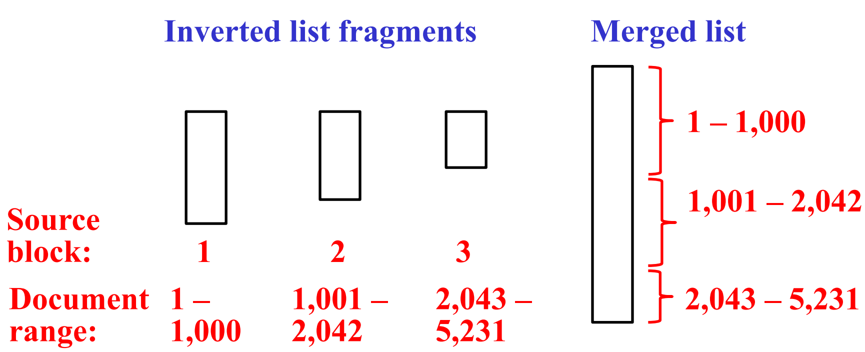
- Very fast – essentially a merge of sorted lists

How Inverted Files are Built: Single Processor



How Inverted Files are Built: Single Processor

Block merge of the inverted list for 'apple'



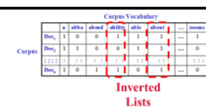
Lecture Outline

- Building inverted lists on a single processor
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Index updates

15

15

Inverted List Indexes



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
:
:

16

16

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

17

17

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*

18

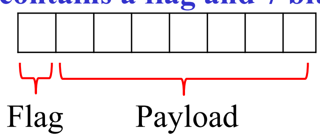
18

Inverted File Compression: Variable Byte Encoding

Variable byte encoding stores a number in a sequence of bytes

$\boxed{\text{byte}_n}$ .... $\boxed{\text{byte}_2}$ $\boxed{\text{byte}_1}$

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:			10000001		00000000
			Last byte		Byte_0
			7 bits		7 bits
Decimal:	131				
Binary:	00000000	00000000	00000000		10000011
Compressed:			10000001		00000011

21

21

Inverted File Compression: Variable Byte Encoding

		7 bits	7 bits	7 bits
Decimal:	613,521			
Binary:	00000000	00001001	01011100	10010001
Compressed:		10100101	00111001	00010001
		Last byte	Byte_1	Byte_0

22

22

Inverted File Compression: Variable Byte Decoding

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

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 … 2^7-1]:	**1 byte:**	**1xxxxxxx**
[2^7…2^{14}-1]:	**2 bytes:**	**1xxxxxxx0xxxxxxx**
[2^{14}…2^{21}-1]:	**3 bytes:**	**1xxxxxxx0xxxxxxx0xxxxxxx**
: :	: :	: :

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

25

25

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

26

26

Lecture Outline

- Building inverted lists on a single processor
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Index updates

27

27

Inverted List Optimizations: Multiple Inverted Lists Per Term

Storing several types of inverted list per term improves efficiency

- Binary: For unranked Boolean
- Frequency: For ranking with scores
  - ~2✕ longer than binary lists
- Positional: For #NEAR, #WINDOW
  - ~2✕ longer than frequency lists

Use as little data as possible for each task

- Reduced I/O and reduced computation

Cost: Extra disk space

Binary

df:	4356
docid:	42
docid:	94
:	
:	

Frequency

df:	4356
docid:	42
tf:	2
docid:	94
:	
:	

Positional

df:	4356
docid:	42
tf:	2
locs:	14
	83
docid:	94
:	
:	

28

28

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

- Reduced computation
- Reduced I/O
  – If inverted lists are read in blocks

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 …
: : : : :

29

29

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

df:	23
→ docid:	42
tf:	3
docid:	59,356
tf:	1
:	

apple

df:	1,033,436
→ docid:	43
tf:	3
docid:	49
tf:	1
:	

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

30

30

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
- Better to advance the ‘apple’ pointer to at least docid 59,356

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
:			:	

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

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

31

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
- More advanced query operators that we haven’t covered

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

32

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
- Reduced I/O and reduced computation
- 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

“apple” Top-Docs List
Doc 1025429 tf 6
Doc 258393 tf 5
: : :
Doc 2 tf 4
Doc 1025430 tf 4

1,000 documents

1,025,430 documents

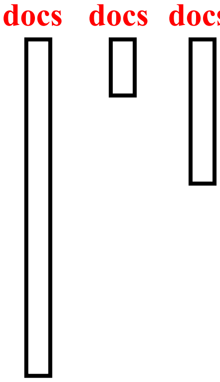
Inverted List Optimizations: Top-Docs (Champion) Lists

How are top-docs lists constructed?

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

Inverted lists for “apple”

All docs **100 docs** **200 docs**



Inverted List Optimizations: Top-Docs (Champion) Lists

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

- **Linux filesystem page size:** 4096 bytes
  - – A page is the minimum unit of I/O
  - – Top-docs lists for lists of less than 4096 bytes don't save I/O
- **How many terms have (compressed) inverted lists longer than 4096 bytes?**
  - – Terms with ctf $\geq$ 1,000
    - » Probably higher than 1,000, but a careful answer requires corpus analysis

35

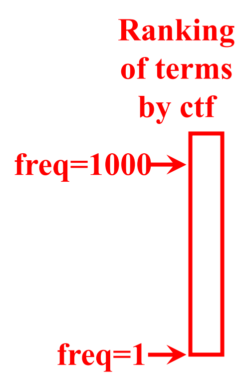
35

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 1,000 times:** A × N / 1000
- **The percentage of terms with ctf $\geq$ 1000:**

  (A × N / 1000) / (A × N) = 1 / 1000 = 0.1%
- **So … if the vocabulary is 1,000,000 terms**
  - – There are fewer than 1,000 top-docs lists
  - – Each list is perhaps 4-8 KB long
  - – So … 4-8 MB



36

36

Inverted List Optimizations

A high-level view of inverted list optimizations

	Reduces I/O	Reduces Computation	Reduces Accuracy
Compression	++	–	
Multiple inverted lists per term	++	+	
Skip lists	Maybe	++	
Top docs lists	+++	++++	Sometimes

37

37

Lecture Outline

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

38

38

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?

39

39

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 / small

Store the parsed document?

- Fast
- Done when storage is cheap

40

40

Forward Indexes

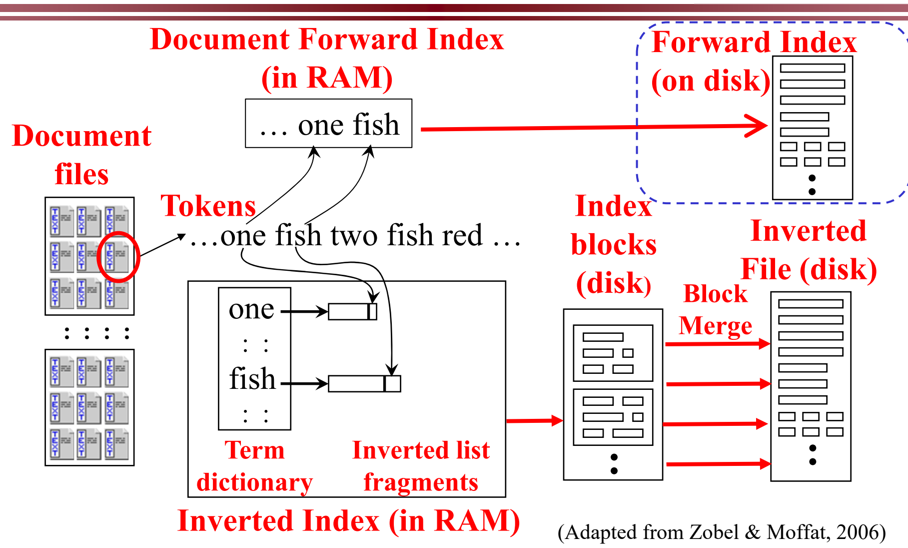
Forward indexes store the <u>indexed form of a document</u>

- **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

41 © 2021, Jamie Callan

41

How Forward Indexes are Built: Single Processor



(Adapted from Zobel & Moffat, 2006)

42 © 2021, Jamie Callan

42

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

43 © 2021, Jamie Callan

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

44 © 2021, Jamie Callan

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**
: : :	: : :

45 © 2021, Jamie Callan

45

How Does Indri Do It?
The DocumentVector Class

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

A list of all terms that occur in the document.

On disk, term ids are stored instead of strings.

string stems[ ]
OOV
obama
state
union
speech
president
barack
delivered
: :

int positions[ ]
1
2
0
0
3
4
5
6
1
7
:

Indexes into the stems list

fields [ ]
0, 6, title
6, 99, body
6, 16, sentence
16, 34, sentence
: : : :

int begin;
int end;
string name;

Positions of fields.
"Begin" is inclusive (contained in the field).
"End" is exclusive (not contained in the field).

46 © 2021, Jamie Callan

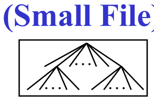
46

Lecture Outline

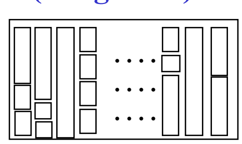
- Building inverted lists on a single processor
- Inverted lists and inverted files
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Index updates

Inverted File Management: Static File (No Updates)

Access Information (Small File)



Inverted Lists (Large File)



- Create files when inverted list fragments are merged
- There is no empty space between inverted lists
- Lists are stored in canonical order (e.g., alphabetic)
- Easy to create, very space efficient
- Very difficult to update; easier to rebuild
  - Update by merging fragments with file to create new file

Updating Indexes

Indexes are expensive to update

- Suppose a new document contains 100 unique terms
- Adding that document means updating 100 inverted lists
  - Acquire lock, read list, write list, release lock
  - A lot of complexity, a lot of I/O
- Adding one document is tolerable, adding several is expensive

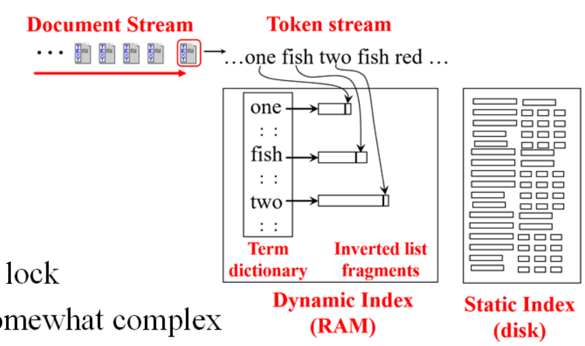
Updates are often done in batches

- Update every day, or after N documents arrive, or …
- Parse documents to generate index modifications
- Update each inverted list for <u>all</u> documents in the batch

Updating Indexes

Sometimes dynamic updates are unavoidable

- E.g., news, Twitter, …



Split index into dynamic and static parts

- The dynamic index is small
- The static index is big
- Make updates to the dynamic index
  - Acquire lock, read list, update list, write lock
  - Faster because lists are small, but still somewhat complex
- Search both static (big) and dynamic (small) components
- Periodically merge dynamic into static

Deleting Documents

Deleting a document is an expensive operation

- If the document contains N terms, must update N inverted lists
- A major problem in a system that is being used dynamically

Delete lists are a less expensive option

- When a document is deleted, add its id to a delete list
  – Don't actually delete it from the index
- When doing a search
  – Evaluate the query to produce a ranked list
  – Scan the list, removing any documents on the delete list
- When the delete list becomes large
  – Garbage collect the inverted lists, or rebuild the index

Lecture Outline

- **Building inverted lists on a single processor**
- **Inverted lists and inverted files**
  – Inverted list compression
  – Inverted list optimizations
- **Forward 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.