11-741:
Information Retrieval

Data Structures, Algorithms and Implementation Issues

Jamie Callan

Carnegie Mellon University

callan@cs.cmu.edu

Overview

- **Document Parsing**
- **Document Indexing**
  - What to index
  - Inverted lists
- **Inverted Files**
  - Access
    - » Permuterm indexes
  - Building
  - Compression
- **Query Evaluation**
  - Ranking documents
  - Optimizations

Basic Facts That Affect Search Engine Design

- **Computational efficiency is <u>really</u> important**
  - Speed <u>and</u> space
  - A <u>research</u> search engine on a <u>research</u> corpus typically processes several hundred million words
    - » Small inefficiencies × 200,000,000 = slow/bloated program
- **Integers are often preferred over strings**
  - Integers almost always take less space
  - Integers can be compared more quickly

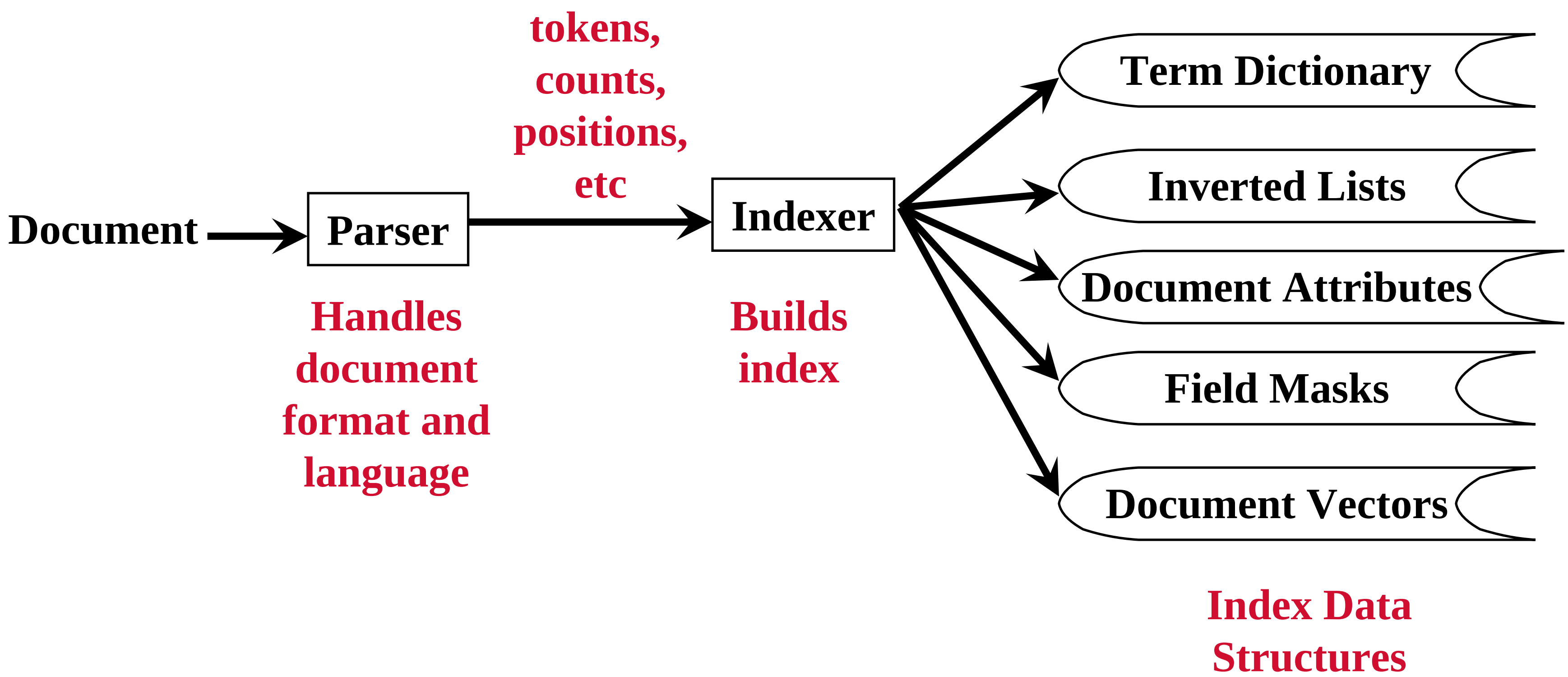
  So…convert strings to integers whenever possible
- **<u>You</u> may not care about efficiency…**

  but you need to understand how it affects search engine design

Parsing and Indexing: Overview

- **Tasks:**
  - Build a set of indexes
    - » Inverted list, document ID, document location, fields, ….
  - Possibly compress the documents
- **Speed:**
  - 2-3 gigabyte per processor hour is respectable (but not fast)
  - "Fast" in 1998 was 4.7 GB per processor hour, on a PC
  - "Fast" in 2004 was 33 GB per processing hour, on a PC
- **Memory:**
  - 1-5% the size of the total uncompressed documents
    - » E.g., 128 MB RAM for 2 GB text, 1 GB RAM for 100 GB text

Parsing and Indexing: Overview



Basic Components of a Search Engine Index

- **Term dictionary**
  - Information about every term in the corpus
- **Inverted lists**
  - Information about where each term occurs in the corpus
- **Document vector**
  - Information about every term in a document
- **Field mask (maybe)**
  - Information about each field in each document
- **Document attributes**
  - Information associated with a document
- **And possibly much more…**

Document Parsing: Sample Document

<DOC>
<DOCNO> AP890101-0001 **</DOCNO>** **markup**
<FILEID> AP-NR-01-01-89 2358EST **</FILEID>**
<FIRST> r a PM-APArts:60sMovies 01-01 1073 **</FIRST>**
<SECOND> PM-AP Arts: 60s Movies,1100 **</SECOND>** **title**
<HEAD> You Don't Need a Weatherman To Know '60s Films Are Here **</HEAD>**
<HEAD> Eds: Also in Monday AMs report. **</HEAD>** **source**
<BYLINE> By HILLEL ITALIE **</BYLINE>**
<BYLINE> Associated Press Writer **</BYLINE>**
<DATELINE> NEW YORK (AP) **</DATELINE>** **Body text**
<TEXT>
The celluloid torch has been passed to a new generation: filmmakers who grew up in the 1960s.
``Platoon,'' ``Running on Empty,'' ``1969'' and ``Mississippi Burning'' are among the movies released in the past two years from writers and ……
</TEXT>
</DOC>

Document Parsing:
Important Parts of the Indexing Process

Task: Convert the document into a set of index elements

- Determine document location
- Determine field/structure boundaries within document
  - E.g., Document ID, Title, Author, Date, Text
- Segment the text in each field into tokens
  - E.g., Apple, AT&T, drive-in, 527-4701, $1,110,427, …
- Case conversion (e.g., “Table” → “table”, “Apple” → “apple”)
- Discard stopwords
- Stem tokens
- Count token occurrences (calculate term frequency)
- Update indexes

Document Parsing: Document Attributes

- **Document attributes are associated with the document text**
  - Could be part of the document markup
  - Not associated with any particular location in the document text
- **Examples:**
  - External Id:
    - » A unique identifier that is independent of the index
    - » Examples: AP890101-0001, http://www.cs.cmu.edu/~callan/
  - Internal Id:
    - » A unique, index-specific identifier
    - » Usually an integer
  - Location in local storage: File-ID, offset, length
  - Page Rank
  - Length

Document Parsing: Storing Document Location

Example:

- Many documents stored in a single file
  - **Why?**

Doc #1 → Once upon a time there was an Engineer. Choo Choo Charlie was his name we hear.

Doc #2 → He had an engine, and he sure had fun. He used Good and Plenty to make his train run.

Doc #3 → Charlie says "Love my Good and Plenty." Charlie says "Really rings a bell." Charlie says "Don't know any other candy that I love so well."

Doc #4 → Bosco puts hustle in your muscle. Bosco puts spree in your knee. Hey, Bosco, wait for me!

Offset	Length
0	2042
2042	2532
4574	3583
8157	2893

Store offset and length
- Good for direct access
- Wasteful

Offset
0
2042
4574
8157
11050

Used in memory

Don't store lengths
- Good for direct access
- Doesn't compress well

Offset
0
2042
2532
3583
2893

Used on disk

Delta encode offsets
- Poor for direct access
- Easy to compress

Document Parsing: Lexical Scanning

- **The lexical scanner identifies the document tokens**
  - Markup (e.g., <DOC>, <TITLE>, <TEXT>)
  - Terms (e.g., “The” “President” “announced” “yesterday”)
- **Lexical scanners need to be very fast**
  - Don’t waste time doing the simple stuff
  - Usually implemented as finite state automata
- **lex: A lexical scanner generator (and its descendent, flex)**
  - You write grammar rules that identify tokens
  - You write C code that does something when a token is found
  - lex generates the finite state automata
  - Few people can write a scanner that is faster than a lex scanner

Document Parsing: Document Grammar

- **Finite state automata are not powerful enough to handle complex document formats**
  - E.g., Nested fields
- **Parsers based on context-free grammars are usually sufficient**
- **Parsers need to be very fast**
  - Don't waste time doing the simple stuff
- **yacc: A parser generator (and its descendent, bison)**
  - You write grammar rules that define the document format
  - You write C code that does something when a rule is matched
  - yacc generates the parser
  - Few people can write a parser that is faster than a yacc parser

Stopword Recognition

- **There are usually fewer than 500 stopwords**
  - Some systems have very few
- **<u>Every</u> word token is checked, so the test must be very fast**
- **Store the stopword list in a hash table**
  - Since stopword lists evolve slowly, calculate a perfect hash code
- **Lookup each word token in the hash table**
  - If found, the token is a stopword, so ignore it
- **Document length & word locations should count stopwords**
  - Example: "Library of Congress" is 3 words

    Locations: 1 2 3

Term Dictionary

- **Main purpose:** Map terms (strings) to term identifiers (integers)
  - Example: stocks → 14,319
- **Other purposes:**
  - Combine with stopword recognition
    - » Lookup "stocks", get term id <u>or</u> N/A if a stopword
  - Combine with stemming
    - » Lookup "stocks", get term id for "stock" (or N/A if a stopword)
  - Provide access to other information about the term
    - » E.g., corpus statistics (df, ctf, idf, …)
    - » E.g., pointer to inverted list

Term Dictionary: Storage

Hash tables

- **O(1) lookup**
  - Very fast
  - But…can suffer if terms don't distribute evenly
- **A little space inefficient**
  - Empty slots
- **Usually a big in-memory data structure**
  - Is this how you want to spend your RAM?

B-Trees

- **O (log n) lookup**
  - Fast, but not O(1)
  - Terms guaranteed to distribute evenly
- **A little space inefficient**
  - Pointers
- **Often implemented as a two-stage data structure**
  - Frequent terms in memory
  - Infrequent terms on disk
  - Efficient use of RAM
  - Occasional slow access

Document Indexing:
Why Index?

Choices for accessing data during query evaluation

- **Scan the entire collection**
  - Typical in early (batch) retrieval systems
  - Still used today, in hardware form (e.g., Fast Data Finder)
  - Computational and I/O costs are *O*(characters in collection)
  - Practical for only "small" collections
- **Use indexes for direct access**
  - An index associates a document with one or more *keys*
    - » Present a key, get back the document
  - Evaluation time *O*(query term occurrences in collection)
  - Practical for "large" collections
  - Many opportunities for optimization
- **Hybrids: Use small index, then scan a subset of the collection**

Document Indexing: What to Index

What should the index contain?

- **Database systems index primary and secondary keys**
  - This is the hybrid approach
  - Index provides fast access to a subset of database records
  - Scan subset to find solution set
- **Title, author, id, creation date, …**
  - Good idea, but none of these support content-based retrieval
- **IR Problem:** Can’t predict the keys that people will use in queries
  - Every word in a document is a potential search term
- **IR Solution:** Index by *all* keys (words)
  - “full text indexing”

Indexes

The index is accessed by the atoms of a query language

- **The atoms are called "features" or "keys" or "terms"**
- **Most common feature types**:
  - Words in text, punctuation
  - Manually assigned terms (controlled & uncontrolled vocabulary)
  - Document structure (sentence & paragraph boundaries)
  - Inter- or intradocument links (e.g., citations)
- **Composed features**
  - Feature sequences (phrases, names, dates, monetary amounts)
  - Feature sets (e.g., synonym classes)

Indexes

Indexing choices (there is no "right" answer)

- **What is a word?**
  - Embedded punctuation (e.g., DC-10, long-term)
  - Case folding (e.g., New vs new, Apple vs apple)
  - Stopwords (e.g., the, a, its)
  - Morphology (e.g., computer, computers, computing, computed)
- **Index granularity has a large impact on speed & effectiveness**
  - Index stems only?
  - Index surface forms only?
  - Index both?

Index Contents

The contents depend upon the retrieval model

- **Feature presence/absence (e.g., SMART)**
  - Boolean
  - Statistical (*tf, df, ctf, doclen, maxtf)*
  - Often about 10% the size of the raw data, compressed
- **Positional (e.g., InQuery, Lemur)**
  - Feature location in document
  - Granularities include word, sentence, paragraph, etc
  - Coarse granularities are less precise, but take less space
  - Word-level granularity about 20-30% the size of the raw data, compressed

Indexes: Implementation

Common implementations of indexes

- **Bitmaps**
- **Signature files**
- **Inverted files**
  - The most common choice today

Common index components

- **Dictionary (lexicon)**
- **Postings**
  - document ids, word positions

Indexes:
Inverted Lists

Inverted lists are today the most common indexing technique

- **Source file:** collection, organized by document
- **Inverted file:** collection organized by term
  - one record per term, listing locations where term occurs
- **During evaluation, traverse lists for each query term**
  - OR: the *union* of component lists
  - AND: an *intersection* of component lists
  - Proximity: an *intersection* of component lists
  - SUM: the *union* of component lists; each entry has a score

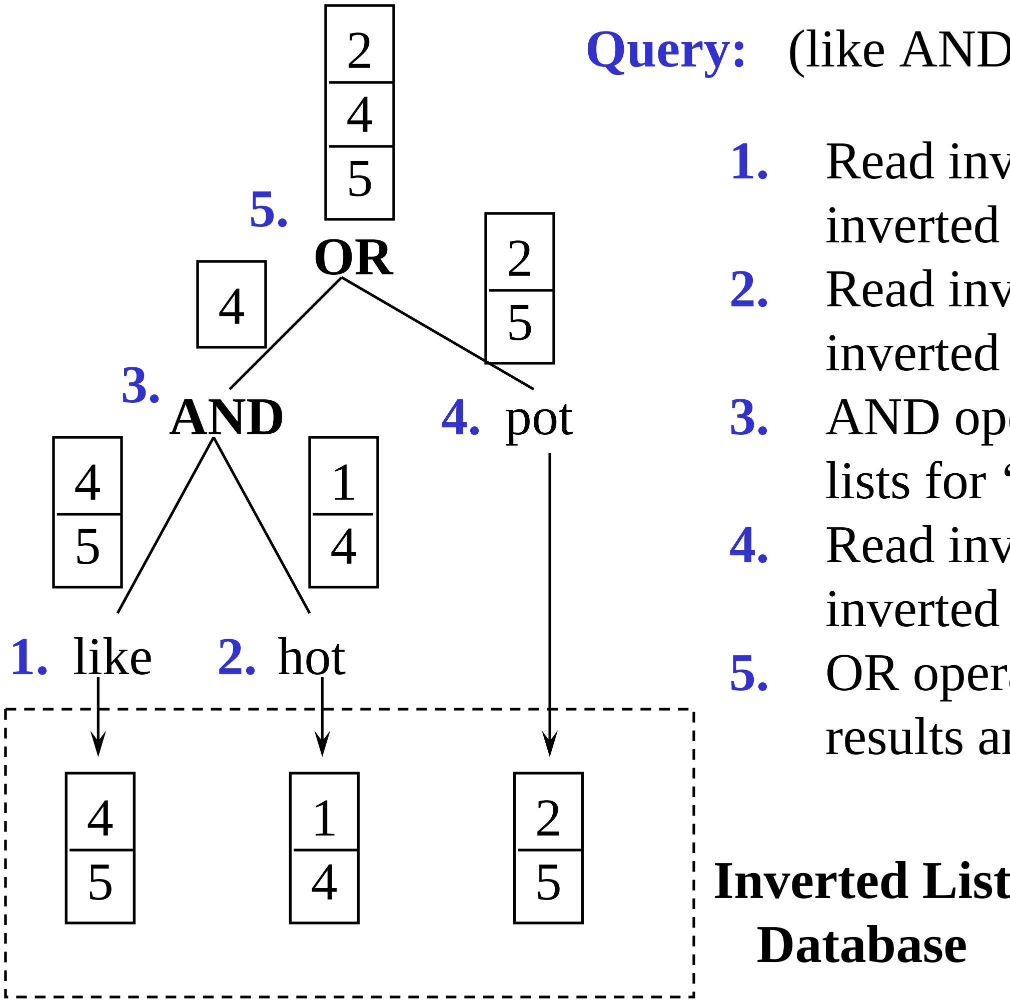
Inverted Lists

Document	Text
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot,
6	Nine days old.

6 documents
Each line is one document.

ID	Term	Documents
1	cold	1,4
2	days	3,6
3	hot	1,4
4	in	2,5
5	it	4,5
6	like	4,5
7	mine	3,6
8	old	3,6
9	pease	1,2
10	porridge	1,2
11	pot	2,5
12	some	4,5
13	the	2,5

Using Inverted Lists: Term-at-a-Time Query Evaluation



Query: (like AND hot) OR pot

1. Read inverted list for ‘like’ from inverted list database
2. Read inverted list for ‘hot’ from inverted list database
3. AND operator: Intersect the inverted lists for ‘like’ and ‘hot’
4. Read inverted list for ‘pot’ from inverted list database
5. OR operator: Union of AND operator results and ‘pot’ inverted list

Inverted Lists
With Word Positions

Document	Text
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot,
6	Nine days old.

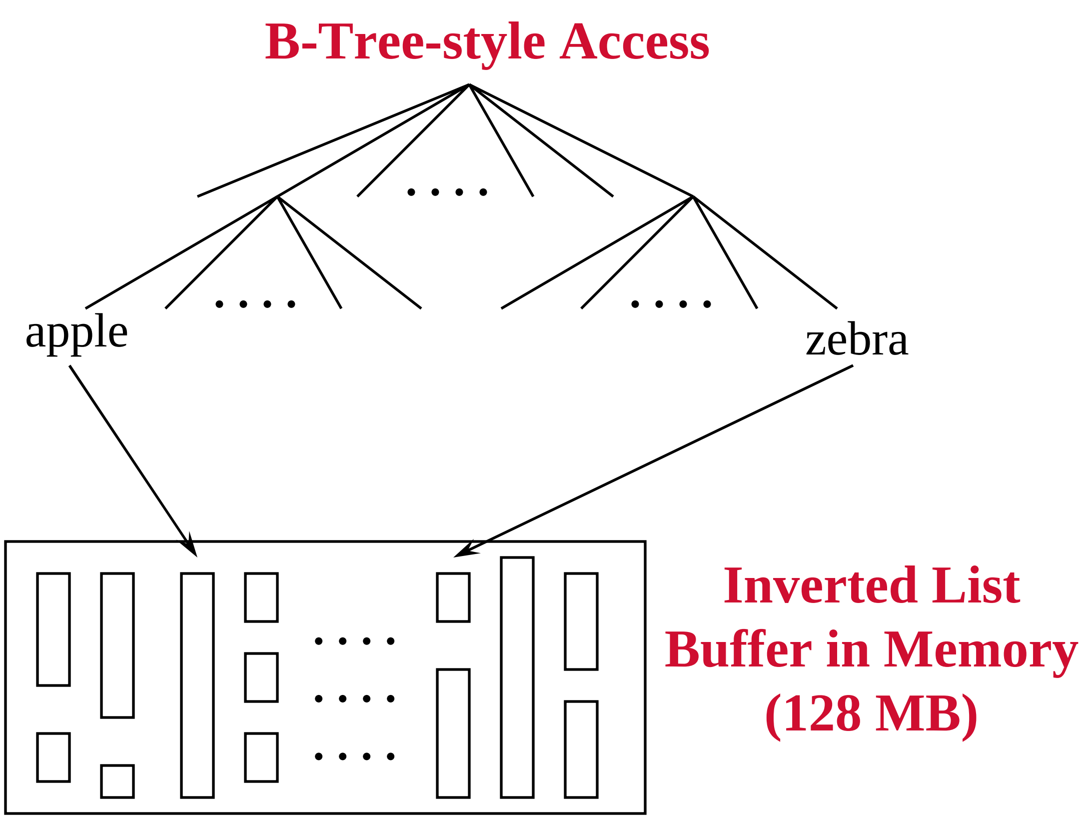
6 documents
Each line is one document.

ID	Term	Documents
1	cold	(1:6) (4:8)
2	days	(3:2) (6:2)
3	hot	(1:3) (4:4)
4	in	(2:3) (5:4)
5	it	(4:3,7) (5:3)
6	like	(4:2,6) (5:2)
7	mine	(3:1) (6:1)
8	old	(3:3) (6:3)
9	pease	(1:1,4) (2:1)
10	porridge	(1:2,5) (2:2)
11	pot	(2:5) (5:6)
12	some	(4:1,5) (5:1)
13	the	(2:4) (5:5)

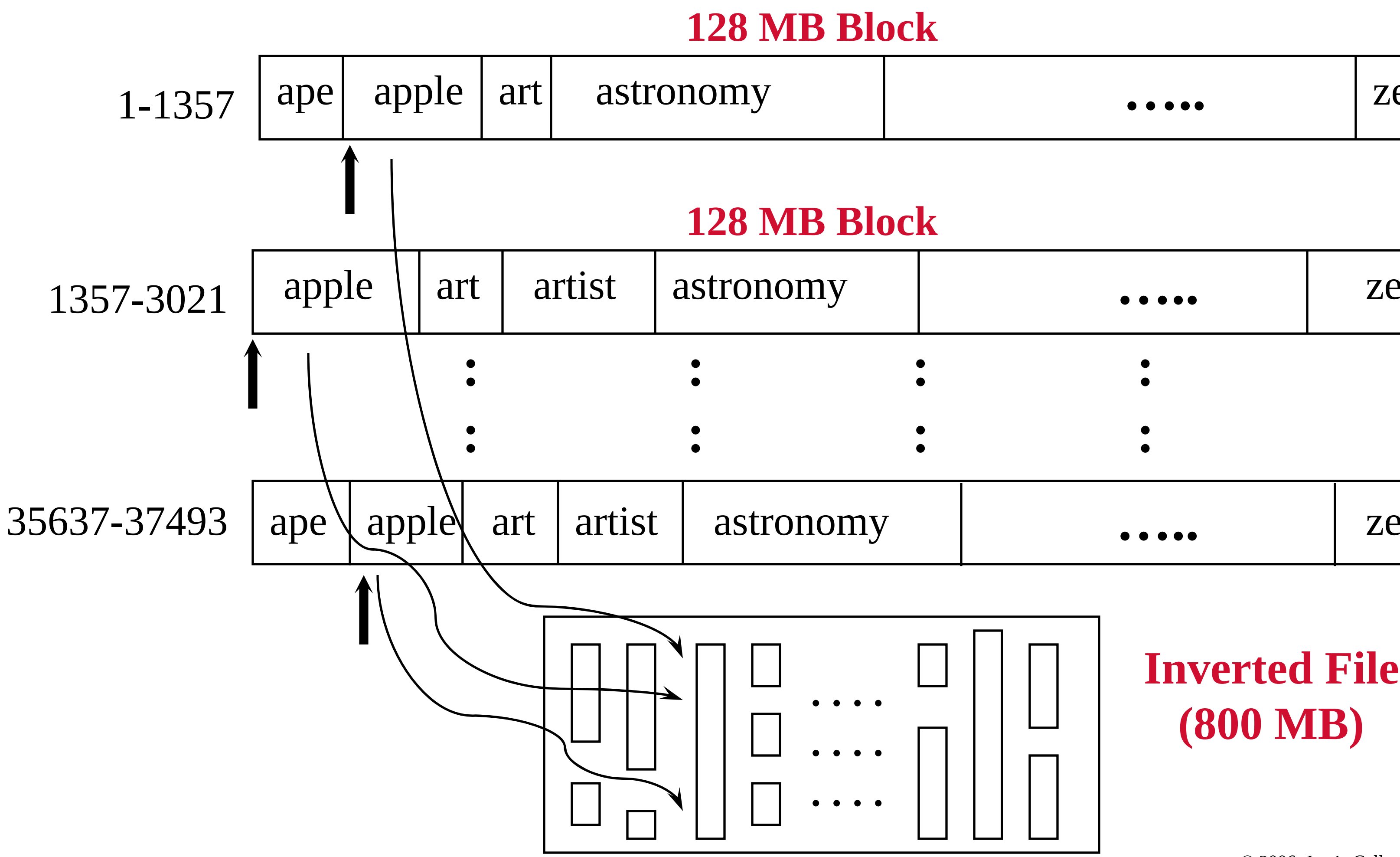
Inverted Lists and Inverted Files: Inverted List Fragments in Memory

Allocate RAM buffer for inverted list fragments

- Usually small compared to size of collection (1-5%)
- When a token is recognized, update inverted list fragment
- When buffer is full
  - write fragments to disk
  - reinitialize buffer
  - continue parsing
- When all documents have been parsed, merge inverted list fragments



Inverted Lists and Inverted Files: Merge Inverted List Fragments on Disk



Inverted File Management

Requirements:

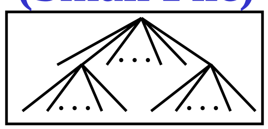
- Many inverted lists (e.g., half a million)
- Very skewed size distribution (e.g., 16 bytes to several MB)
- Need to provide fast I/O
- Possibly need to support efficient updates
  - Or, possibly just rebuild the index periodically to update

Solution:

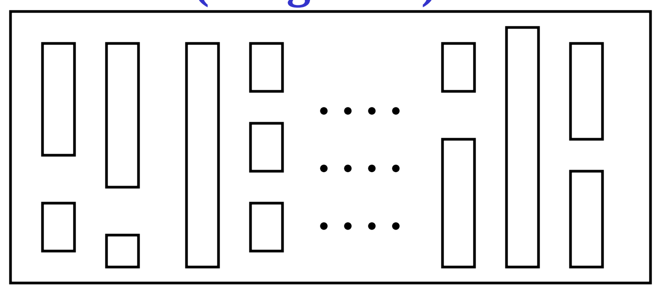
- Index consists of two types of information:
  - Access mechanism(s), e.g., B-Tree, Hash table
  - Inverted lists
- Different management techniques for each type of information

Inverted File Management: Static File (No Updates)

Access Information (Small File)



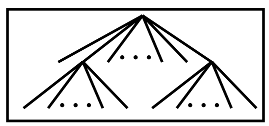
Inverted Lists (Large File)



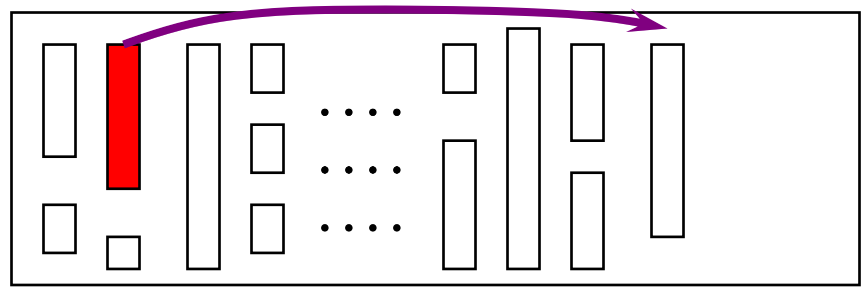
- **Create files when inverted list fragments are merged**
- **Inverted lists are packed tightly together**
  - One list follows another immediately
- **Lists are stored in canonical order (e.g., alphabetic)**
- **Easy to create, very space efficient**
- **Very difficult to update; easier to rebuild**
  - Update by merging fragments with file to create new file

Inverted File Management: Supporting Updates (Version 1)

Access Information



Inverted Lists

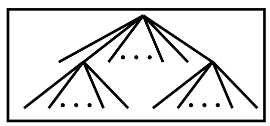


Approach:

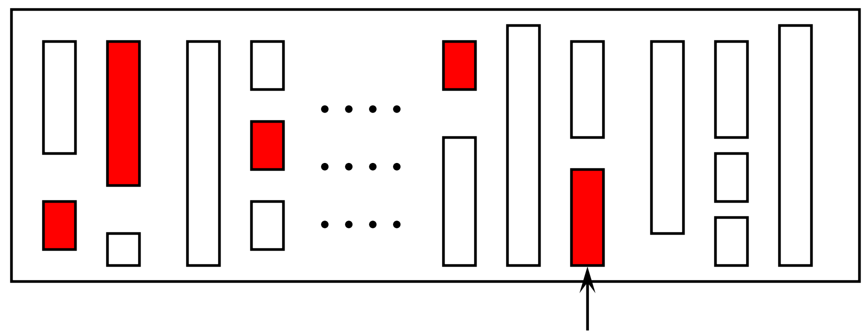
- Append updated information to inverted list
- Inverted list is now too long for original location in file
- Move to new location (usually nearer to end of file)
- Update access information (B-Tree, Hash table, etc) with new location and length

Inverted File Management: Supporting Updates (Version 1)

Access Information



Inverted Lists

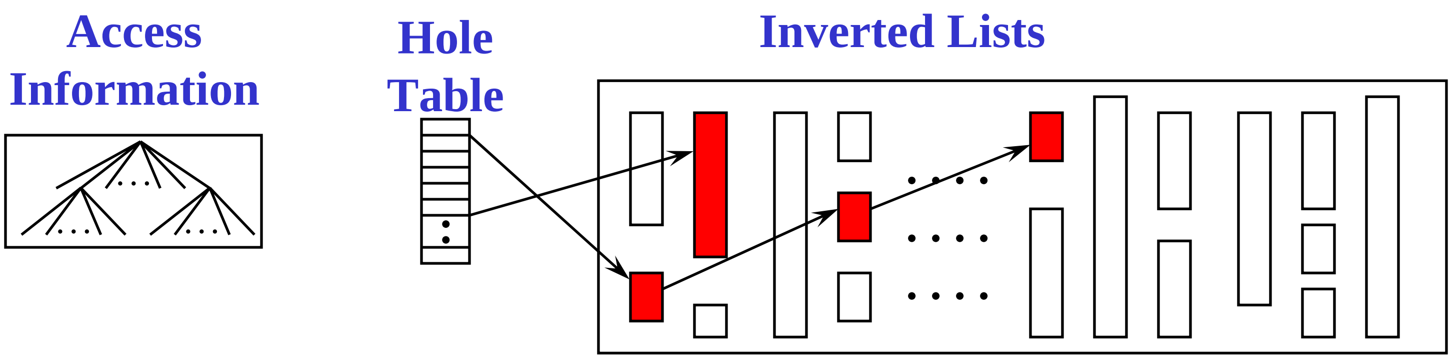


Unused space (“hole”)

Problem:

- After awhile, there are a lot of “holes” of different sizes
  - Significant wasted space
- Recycling holes takes a surprisingly large amount of time
  - Many holes to consider, many sizes
- Garbage collecting the holes takes time

Inverted File Management: Supporting Updates (Version 2)



- **"Round up" each inverted list length to the next power of 2**
  - 16 bytes, 32 bytes, 64 bytes, 128 bytes, 256 bytes, ….
- **Only a small number of hole sizes (e.g., 30)**
  - For each hole size, maintain a linked list of available holes
  - Recycling holes is much easier (direct lookup, pop from list)

Inverted File Management: Supporting Updates (Version 2)

- **On average, each list has about 25% free space at end**
- **Some inverted list updates can be done "in place"**
  - Small update: Write new information into space at end of list
- **Moving list to a new location is much easier**
  - Need a hole of size 2^n? Pop it off the appropriate linked list
  - If no hole is available, create new space at the end of the file
- **Trading space for speed**
  - This approach does waste space
  - But, the amount of wasted space is predictable (about 25%)

Indexes: Building Indexes

Indexes are expensive to update; Usually done in batches

- **Typical build/update procedure:**
  - One or more documents arrive to be added / updated
  - Documents parsed to generate index modifications
  - Each inverted list updated for *all* documents in the batch
- **Concurrency control required**
  - To synchronize changes to documents and index
  - To prevent readers & writers from colliding
- **Common to split index into static / dynamic components**
  - All updates to dynamic components
  - Search both static and dynamic component
  - Periodically merge dynamic into static

Inverted List Indexes: Compression

Inverted lists are usually compressed

- **Inverted files with word locations are about the size of the raw data**
- **Distribution of numbers is skewed**
  - Most numbers are small (e.g., word locations, term frequency)
- **Distribution easily can be made more skewed**
  - Delta encoding: 5, 8, 10, 17 --> 5, 3, 2, 7
- **Simple compression techniques are often the best choice**
  - Simple algorithms nearly as effective as complex algorithms
  - Simple algorithms much faster than complex algorithms
  - Goal: Time saved by reduced I/O > Time required to uncompress

Inverted List Indexes: Compression

- **The longest lists, which take up the most space, have the most frequent (probable) words.**
- **Compressing the longest lists saves the most space.**
- **The longest lists compress easily because they contain the least information.**
- **Algorithms:**
  - Delta encoding
  - Variable-length encoding
  - Unary codes
  - Gamma codes
  - Delta codes

Inverted List Indexes: Compression

Delta Encoding ("Storing Gaps")

- Store the differences between numbers
- Reduces range of numbers.
- Produces a more skewed distribution.
- Increases probability of smaller numbers.
- (Stemming also increases the probability of smaller numbers.)

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

Restricted Variable-Length Codes: Generalization for Numeric Data

- **Store first 2^7 numbers in 7 bits: 1xxxxxxx**
- **Store next 2^{14} numbers in 14 bits: 0xxxxxxx1xxxxxxx**
- **Store next 2^{21} numbers in 21 bits: 0xxxxxxx0xxxxxxx1xxxxxxx**
- **And so on….**
- **Often used on inverted lists, after delta encoding integer data**
  - Many numbers fit in one byte
  - It is rare to exceed two bytes (16,511)
- **Advantages:**
  - Effective, non-parametric
  - Encoding and decoding can be done very efficiently
  - Easy to find number boundaries without decoding
  - Integer encoding is Endian-independent

Inverted List Compression: Unary Code

- Represent a number n >= 0 as n 1 bits and a terminating 0.
- Great for small numbers.
- Terrible for large numbers.

Inverted List Compression: Gamma Code

A combination of unary and binary codes

- **The unary code stores the number of bits needed to represent n in binary.**
- **The binary code stores the information necessary to reconstruct n.**
- **Unary code stores 1 + floor (log n)**
- **Binary code stores n - 2^(floor (log n))**
- **Example: n = 9**
  - floor (log 9) = 3, so unary code is 1110.
  - 9-8=1, so binary code is 001.
  - The complete encoded form is 1110001 (7 bits).
- **This method is superior to a binary encoding**

Inverted List Compression: Delta Code (Not Delta Encoding)

A generalization of the Gamma code

- **Encode the length portion of a Gamma code in a Gamma code.**
- **Gamma codes are better for small numbers.**
- **Delta codes are better for large numbers.**
- **Example:**
  - For gamma codes, number of bits is

    $$1 + 2 * \lfloor \log n \rfloor$$

  - For delta codes, number of bits is

    $$\lfloor \log n \rfloor + 1 + 2 * \lfloor \log \left(1 + \lfloor \log n \rfloor \right) \rfloor$$

Inverted File Compression: Comparison

	Bits Per Number		
Number	RVL	Gamma	Delta
1	8	1	1
2	8	3	4
4	8	5	5
8	8	7	8
16	8	9	9
32	8	11	10
64	8	13	11
128	16	15	14
256	16	17	15
512	16	19	16
1,024	16	21	17

	Bits Per Number		
Number	RVL	Gamma	Delta
2,048	16	23	18
2,048	16	23	18
4,096	16	25	19
8,192	16	27	20
16,384	24	29	21
32,768	24	31	24
65,536	24	33	25
131,072	24	35	26
262,144	24	37	27
524,288	24	39	28
1,048,576	24	41	29

Inverted File Compression: Comparison

	Bits per pointer			
Method	**Bible**	**GNUbib**	**Comact**	**TREC**
Unary	264.00	920.00	490.00	1719.00
Binary	15.00	16.00	18.00	20.00
Bernouli	9.67	11.65	10.58	12.61
Gamma	6.55	5.69	4.48	6.43
Delta Code	6.26	5.08	4.36	6.19
Observed Freq	5.92	4.83	4.21	5.83
Bernoulli	6.13	6.17	5.40	5.73
Hyperbolic	5.77	5.17	4.65	5.74
Skewed Bernoulli	5.68	4.71	4.24	5.28
Batched Freq	5.61	4.65	4.03	5.27

(Managing Gigabytes)

Inverted File Compression: Summary

- **A compressed inverted file, without positional information:**
  - About 10% the size of the original text
    - » Delta encoding, variable length encoding
- **A compressed inverted file with positional information:**
  - About 20-30% the size of the original text
    - » Delta encoding, variable length encoding
- **More aggressive compression**
  - Yields small improvements
  - Is often much slower
- **Adaptive compression never used on inverted lists**
  - Why?