Storing the index and Using the index to evaluate queries

Review: Inverted Index

- For each term, keep list of document entries, one for each document in which it appears: a postings list
  - Document entry is list of positions at which term occurs and attributes for each occurrence: a posting
- Keep summary term information
- Keep summary document information

meta-data

Consider "advanced search" queries

Content Coordination
- Phrases
- Numeric range
- NOT
- OR

Document Meta-data
- Language
- Geographic region
- File format
- Date published
- From specific domain
- Specific licensing rights
- Filtered by "safe search"

Issue of efficient retrieval

Basic operations consider

- One term
- AND of several terms
- OR of several terms
- NOT term
- proximity

Basic postings list processing: Merging posting lists

- Have two lists must coordinate
  - Find shared entries and do "something"
  - "something" changes for different operations
    - Set operations UNION? INTERSECTION? DIFFERENCE? …
  - Filter with document meta-data as process

5

Basic retrieval algorithms

- One term:
  - look up posting list in (inverted) index
- AND of several terms:
  - Intersect posting lists of the terms: a list merge
- OR of several terms:
  - Union posting lists of the terms
  - eliminate duplicates: a list merge
- NOT term
  - If *terms* AND NOT(other *terms*), take a difference
  - a list merge (similar to AND)
- Proximity
  - a list merge (similar to AND)

6

Merging two unsorted lists

X
- Read 2nd list over and over - once for each entry on 1st list
  - computationally expensive
    time $O(|L_1|*|L_2|)$ where |L| length list L
- Build hash table on entry values; insert entries of one list, then other; look for collisions
  - must have good hash table
  - unwanted collisions expensive
  - often can't fit in memory: disk version
- Sort lists; use algorithm for sorted lists
  - often lists on disk: external sort
  - can sort in O(|L| log |L|) operations

7

Sorted lists

- Lists sorted by some identifier
  - same identifier both lists; not nec. unique
- Read both lists in "parallel"
  - Classic list merge:
    (sorted list_1 , sorted list_2) ⇒ sorted set union
  - General merge: if no duplicates, get time $|L_1|+|L_2|$
- Build lists so sorted
  - pay cost at most once
  - maybe get sorted order "naturally"
- If only one list sorted, can do binary search of sorted list for entries of other list
  - Must be able to binary search! - rare!
    - can't binary search disk

8

Duplicates in sorted lists

- Sorted on a value vi that is not unique identifier.
- docID# identifies doc. uniquely

postings list "cat"	postings list "dog"
v1: docIDx	v1: docIDx
v2: docIDk	v3: docIDz
v4: docIDd	v4: docIDu
v4: docIDv	v4: docIDd
v4: dodIDf	v4: docIDv
v5: docIDq	v4: docIDp
v6: docIDw	v7: docIDr

Keys for documents

For posting lists, entries are documents
What value is used to sort?

- Unique document IDs
  - can still be duplicate documents
  - consider for Web when consider crawling
- document scoring function that is independent of query
  - PageRank, HITS authority
  - sort on document IDs as secondary key
  - allows for approximate "highest k" retrieval
    - approx. k highest ranking doc.s for a query

Keys within document list

Processing within document posting

- Proximity of terms
  - merge lists of terms occurrences within same doc.
- Sort on term position

Computing document score

1. "On fly"- as find each satisfying document
2. Separate phase after build list of satisfying documents

- For either, must sort doc.s by score

Web query processing: limiting size

- For Web-scale collections, may not process complete posting list for each term in query
  - at least not initially
- Need docs sorted first on global (static) quantity
  - why not by term frequency for doc?
- Only take first k doc.s on each term list
  - k depends on query - how?
  - k depends on how many want to be able to return
    - Google: 1000 max returns
  - Flaws w/ partial retrieval from each list?
- Other limits? query size
    - Google: 32 words max query size

13

Limiting size with term-based sorting

- Can sort doc.s on postings list by score of term
  - term frequency + …
- Lose linear merge - salvage any?
- Tiered index:
  - tier 1: docs with highest term-based scores, sorted by ID or global quantity
  - tier 2: docs in next bracket of score quality, sorted
  - etc.
  - need to decide size or range of brackets
- If give up AND of query terms, can use idf too
  - only consider terms with high idf = rarer terms

14

Data structure for inverted index?

How access individual terms and each associated postings list?

Assume a dictionary entry for each term points to its posting list

15

Last time

- Postings lists stored as lists
- Query processing based on merge-like operations on postings lists
  - action on duplicates
- Use of classic linear-time list merge algorithms:
  - postings lists sorted by a static value
  - build hash table; duplicates collide
- Disk properties

16

Today

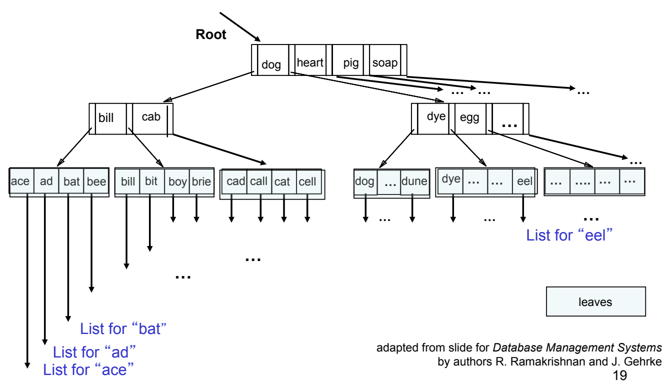
- Data structures for dictionary
  - how accessing postings lists
  - disk access costs
- Constructing inverted index

Data structure for dictionary?

- Sorted array:
  - binary search IF can keep in memory
  - High overhead for additions
- Hashing
  - Fast look-up
  - Collisions
- Search trees: B+-trees
  - Maintain balance - always log look-up time
  - Can insert and delete

Example B+ Tree

order = 2: 2 to 4 search keys per interior node



adapted from slide for *Database Management Systems* by authors R. Ramakrishnan and J. Gehrke

B+- trees

- All index entries are at leaves
- Order *m* B+ tree has *m+1* to *2m+1* children for each interior node
  - **except root** can have as few as 2 children
- Look up: follow root to leaf by keys in interior nodes
- Insert:
  - find leaf in which belongs
  - If leaf full, split
  - Split can propagate up tree
- Delete:
  - Merge or redistribute from too-empty leaf
  - Merge can propagate up tree

Disk-based B+ trees for large data sets

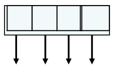
- Each leaf is file page (block) on disk
- Each interior node is file page on disk
- Keep top of tree in buffer (RAM)
- Typical sizes:
  - m ~ 200;
  - average fanout ~ 267
    - Height 4 gives ~ 5 billion entries

prefix key B+ trees

- Save space
- Each interior node key is shortest prefix of word needed to distinguish which child pointer to follow
  - Allows more keys per interior node
  - higher fanout
    - fanout determined by what can fit
    - keep at least 1/2 full

Revisit hashing - on disk

- hash of term gives address of bucket on disk
- bucket contains pairs
  (term, address of first page of postings list)
- bucket occupies one file page



Now

How construct inverted index from “raw” document collection?

- Don’t worry about getting into final index data structure

Preliminary decisions

- Define "document": level of granularity?
  - Book versus Chapter of book
  - Individual html files versus combined files that composed one Web page
- Define "term"
  - Include phrases?
    - How determine which adjacent words -- or all?
  - Stop words?

25

Pre-processing text documents

- Give each document a unique ID: docID
- Tokenize text
  - Distinguish terms from punctuation, etc.
- Normalize tokens
  - Stemming
    - Remove endings: plurals, possessives, "ing",
      - cats -> cat; accessible -> access
    - Porter's algorithm (1980)
  - Lemmatization
    - Use knowledge of language forms
      - am, are, is -> be
    - More sophisticated than stemming

(See *Intro IR* Chapter 2)

26

Construction of posting lists

- Overview
  - "document" now means preprocessed document
  - One pass through collection of documents
  - Gather postings for each document
  - Reorganize for final set of lists: one for each term
- Look at algorithms when can't fit everything in memory
  - Main cost file page reads and writes
    - "file page" minimum unit can read from drive
      - May be multiple of "sector" device constraint

27

Memory- disk management

- Have buffer in main memory (RAM)
  - Size = B file pages
  - Read from disk to buffer, page at a time
    - Disk cost = 1 per page
  - Write from buffer to disk, page at at time
    - Disk cost = 1 per page

28

Sorting List on Disk - External Sorting General techique

- Divide list into size-B blocks of contiguous entries
- Read each block into buffer, sort, write out to disk
- Now have ⌈L/B⌉ sorted sub-lists
  where L is size of list in file pages
- Merge sorted sub-lists into one list
  – How?

29

Merging Lists on Disk: General technique

- K sorted lists on disk to merge into one
- If K+1 <= B:
  – Dedicate one buffer page for output
  – Dedicate one buffer page for each list to merge input from different lists
  – Algorithm:
    Fill 1 buffer page from each list on disk
    Repeat until merge complete:
      Merge buffer input pages to output buffer pg
      When output buffer pg full, write to disk
      When input buffer pg empty, refill from its list

30

- If K+1 > B:
  – Dedicate one buffer page for output
  – B-1 buffer page for input from different lists
  – Define "level-0 lists": lists need to merge

31

If K+1 > B: Algorithm

```
j=0
Repeat until one level-j list:
  { Group level-j lists into groups of B-1 lists
            // ⌈K/(B-1)⌉ groups for j=0
    For each group, merge into one level-(j+1) list by:
      { Fill 1 buffer page from each level-j list in group
        Repeat until level-j merge complete:
          Merge buffer input pages to output buffer pg
          When output buffer pg full,
              write to group's level-(j+1) list on disk
          When input buffer pg empty, refill from its list
      }
    j++
  }
```

32

Number of file page read/writes?

- Merge lists?
- External sort?

33

So far

- Preprocessing the collection
- Sorting a list on disk (external sorting)
  - Cost as disk I/O

Now look at actually building

34

Index building Algorithm: "Block Sort-based"

1. Repeat until entire collection read:
   - Read documents, building (term, <attributes>, doc) tuples until buffer full
     - one tuple for each occurrence of a term
   - Sort tuples in buffer by term value as primary, doc as secondary
     - Tuples for one doc already together
     - Use sort algorithm that keeps appearance order for = keys: stable sorting
   - Build posting lists for each unique term in buffer
     - Re-writing of sorted info
   - Write partial index to disk

35

continuing "Blocked Sort-based"

2. Merge partial indexes on disk into full index

- Partial index lists of (term:postings list) entries must be merged
- Partial postings lists for one term must be merged
  - Concatenate
    - Keep documents sorted within posting list
- If postings for one document broken across partial lists, must merge

36

Remarks: Index Building

- As build index:
  - Build dictionary
  - Aggregate Information on terms, e.g. document frequency
    - store w/ dictionary
  - What happens if dictionary not fit in main memory as build inverted index?
- May not actually keep every term occurrence, maybe just first k.
  - Early Google did this for k=4095. Why?

37

What about anchor text?

- Complication
- Build separate anchor text index
  - strong relevance indicator
  - keeps index building less complicated

38

Other separate indexes?

Examples

- Other strong relevance indicators
  - abstracts of documents
    - compare listing abstract positions 1st in main index
  - tiered indexes based on term weights
- types of documents
  - volatility
    - news articles
    - blogs
    - etc.

39