Using and building the index

1

Review: Model

- Document: sequence of {terms + attributes}
- Query: sequence of terms
  - Can make more complicated: Advanced search
- Satisfying: in current search engines, documents “containing” all terms
  - AND model
  - “containing” includes anchor text of pointers to this doc from other docs
- Ranking: wide open function of document and terms

2

Review: Posting list

- For each document, keep list of terms appearing and attributes for each term
  - Actually list of positions at which each term occurs and attributes for that occurrence

Invert:

- For each term, keep list of documents in which it appears and attributes
  - For each document, list of positions at which term occurs and attributes for each occurrence

=> Inverted file/ Inverted index

3

Consider “advanced search” queries

To know if satisfied need:

Content
- Phrases
- OR
- NOT
- Numeric range
- Where in page

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

4

Retrieval of satisfying documents

- Inverted index will allow retrieval for content queries
- Keep meta-data on docs for meta-data queries
  - Need length even for tf.idf
- Issue of efficient retrieval

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
- NOT term
  - If *terms* AND NOT(*terms*), take a difference
  - a list merge (similar to AND)
- Proximity
  - a list merge (similar to AND)

6

Merging posting lists

- Have two lists must coordinate
  - Find shared entries and do something
  - Use for intersection and related
- Algorithms?
  - Unsorted lists: read 2nd list over and over- once for each entry on 1st list
  - Unsorted lists: build hash table on entry values; insert entries of one list, then other, looking for collisions
  - Lists sorted by some entry ID: Read both lists in "parallel"
    - Classic list merge algorithm for sorted lists
  - If one list sorted, can do binary search of sorted list for entries of other list
    - Must be able to binary search!
- For posting lists, entries are documents
  - More processing within document postings

7

Data structure for inverted index?

- 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

8

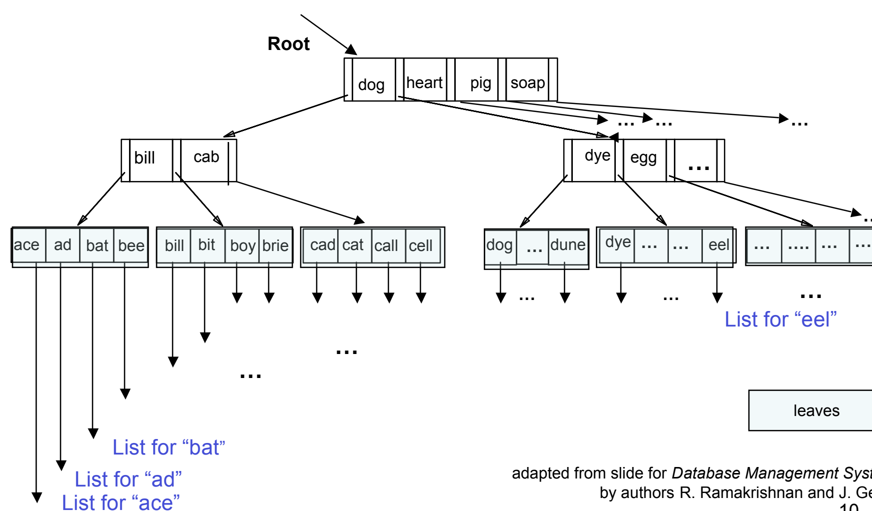
B+- trees

- All index entries are at leaves
- Order *m* B+ tree has *m* to *2m* children for each interior node
- 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

9

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

10

- B+ trees used for large data sets
  - Leaves are file pages on disk
  - Each interior node is file page on disk
  - Keep top of tree in buffer (RAM)
  - M is typically 100; average fanout 133
    - Height 4 gives ~ 300 Million entries
- Save more space: prefix B+ trees for words
  - Each interior node key is shortest prefix of word that need 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

Another tree structure: tries

- Strictly for character strings
- Each edge out of node labeled with one character
- Follow path root to leaf to spell word
- Leaf contain data for word
  - Usually pointer