Inverted Index Construction

Introduction to Information Retrieval

Christof Monz and Maarten de Rijke

Spring 2002

Today's Program

- The need for indexes
- Accessing data
- Indexing choices
- Inverted index/inverted file
- Accessing the index
- Index construction
  - memory-based
  - sort-based

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 only for “small” collections

Accessing Data During Query Evaluation

- Use indexes for direct access
  - Evaluation time O(query term occurrences in collection)
  - Practical for "large" collections
  - Many opportunities for optimization

What Should the Index Contain?

- Database systems index primary and secondary keys
  - Index provides fast access to a subset of database records
  - Scan subset to find solution set
- IR Problem: Cannot predict keys that people will use in queries
  - Every word in a document is a potential search term
  - Solution: Index by all keys (words)

Accessing the Index

- Index accessed through **features** or **keys** or **terms**
  - Keys/terms can be atomic or complex
- Most common 'atomic' keys/terms:
  - Words in text, punctuation
  - Manually assigned terms (controlled and uncontrolled vocabulary)
  - Document structure (sentence and paragraph boundaries)
  - Inter- or intra-document links (e.g., citations)
- Composed features
  - Sequences (phrases, names, dates, monetary amounts)
  - Sets (e.g., synonym classes)

Indexing Choices

- What is a word?
  - Embedded punctuation (e.g., DC-10, long-term, AT&T)
  - 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 and effectiveness
  - Index stems only?
  - Index surface forms only?
  - Index both?

Index Contents

- Feature presence/absence
  - Boolean
  - Statistical (tf, df, ctf, doclen, . . .)
  - Often about 10% the size of the raw data, compressed
- Positional information
  - Feature location within 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

Implementation

- Common implementations of indexes
  - Bitmaps
  - Signature files
  - **Inverted files**
  - Hashing
  - n-grams
- Common index components
  - Dictionary (lexicon)
  - Postings (document ids, word positions)
- Inverted **files** (or **index**) vs inverted **list**
  - inverted file: each elt of a list points to a doc or file name
  - inverted list: our definition

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 Files

- Example text: each line is a document

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

Inverted Files

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

$\Longrightarrow$

Number	Text	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	nine	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

Word-Level Inverted File

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

$\Longrightarrow$

Number	Text	(Document; Word)
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	nine	(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 List Index: Access Methods

- Two basic data structures to organize data:
  - search trees
  - hashing
- Differ in how search is performed
  - trees define a lexicographic order over the data; the complete value of a key is used to direct search
  - hashing "randomizes" the data order, leading to faster searches on average, with the disadvatage that scanning in sequential order is not possible (e.g., range searches are expensive)

Search Trees

- Each internal node contains a key
  - left subkey stores all keys smaller than the parent key
  - right subtree stores keys larger than the parent key

- **B-tree** (balanced tree) of order m
  - root has between m and $2m$ keys, as do all other internal nodes
  - if k_i is the i-t key of a given internal node, then all keys in the $(i-1)$-th child are smaller than k, while all keys in the i-th child are bigger
  - all leaves are at the same depth

- Usually, a B-tree is used as an index, and all associated data are stored in the leaves or **buckets**: **B$^+$-tree**

B-Trees

- Usually, a B-tree is used as an index, and all associated data are stored in the leaves or **buckets**: $\mathbf{B^+}$**-tree**
- B-trees are mainly used as a primary key access method for large databases in secondary memory
- To search a given key, we go down the tree choosing the appropriate branch at each step
  - number of disk accesses = height of the tree

Hashing

- A **hashing function** $h(x)$ maps a key x to an integer in a given rang; e.g., $\mathbf{0}$ to $m - 1$
  - aim: produce values uniformly distributed in the given range
- A hashing function is used to map a set of keys to slots in a **hashing table**
- If the hashing function gives the same slot for two different keys, a **collision** occurs
  - collisions are possible if the domain of possible key values exceeds the number of locations in which they can be stored
  - whenever a collision occurs, some extra computation is necessary to further determine a unique location for a key
  - hashing techniques differ in how collisions are handled

More Hashing

- The best performance if the number of possible key values N equals the number of locations m, using a 1-to-1 mapping
  - Requires knowledge of the representation of the key domain
  - Example: if keys are consecutive numbers in the range (N_1, N_2) then $m = N_2 - N_1 + 1$ and the mapping on a key k is $k - N_1$
- In most applications the number actually stored keys is much smaller than the number of possible key values
- Mapping involved in hashing as two aspects
  - number of collisions
  - amount of unused storage
- Optimizing one occurs at the expense of the other

Inverted List: Access Methods

- How is a file of inverted lists accessed?
  - B-Tree (B+ Tree, B* Tree, etc)
    - Supports exact-match and range-based lookup
    - $O(\log n)$ lookups to find a list
    - Usually easy to expand
  - Hash table
    - Supports exact-match lookup
    - $O(1)$ lookups to find a list
    - May be complex to expand

Index Construction: Preview

- Today
  - memory-based inversion
  - sort-based inversion
  - (compression)
- Next time
  - FAST-INV

Index Construction: Computational Model

- Hypothetical collection of 5Gb and 5 million docs
- Some nominal performance figures

Parameter	Symbol	Assumed Value
Total text size	B	5×10^9 bytes
Number of docs	N	5×10^6
Number of distinct words	n	1×10^6
Total number of words	F	800×10^6
Number of index pointers	f	400×10^6
Final size of compressed inv. file	I	400×10^6 bytes
Disk seek time	t_s	10×10^{-3} sec
Disk transfer time per byte	t_r	0.5×10^{-6} sec
Inverted file coding per byte	t_d	5×10^{-6} sec
Time to compare and swap 10-byte records	t_c	10^{-6} sec
Time to parse, stem and look up one term	t_p	20×10^{-6} sec
Amount of main memory available	M	40×10^6 bytes

Index Construction: Preview

- Main memory requirements, disk space requirements beyond what is needed to store the inverted index

Method	Memory (Mb)	Disk (Mb)	Time (hours)
Linked lists (memory)	4000	0	6
Linked lists (disk)	30	4000	1100
Sort-based	40	8000	20
Sort-based (compressed)	40	680	26
Sort-based (multiway merge)	40	540	11
Sort-based (multiway in-place)	40	150	11
⋮			
Text-based partition	40	35	15

Memory-based Inversion: Outline

- Informal outline
  - Use a dynamic dictionary data structure (B-tree, hash table) to record distinct terms, with a linked list of nodes storing line numbers associated with each dictionary entry
  - Once all documents have been processed, the dictioary is traversed, and the list of terms and corresponding line numbers is written

Memory-based Inversion: Algorithm

1. `/* Initialization */`
   Create an empty dictionary structure S

2. `/* Phase one: collection of term appearances */`
   For each doc D_d in the collection $(1 \leq d \leq N)$
   (a) Read D_d, parsing it into index terms
   (b) For each index term $t \in D_d$
      i. Let $f_{d,t}$ be the frequency in D_d of term t
      ii. Search S for t
      iii. If t is not in S, insert it
      iv. Append a node storing $(d, f_{d,t})$ to the list corresponding to term t

Memory-based Inversion: Algorithm

3. `/* Phase two: output of inverted file */`
   For each term $1 \leq t \leq n$

   (a) Start a new inverted file entry
   (b) For each $(d, f_{d,t})$ in the list corresponding to t, append $(d, f_{d,t})$ to this inverted file entry
   (c) If required, compress the inverted file entry
   (d) Append this inverted file entry to the inverted file

Memory-based Inversion: Costs

- At the assumed rate of 2 Mb/sec, it takes about 40 minutes to read 5 Gb of text
- Parsing and stemming to create index terms, and searching for these terms in the dictionary takes 4 hours (at 20 microsec/wd)
- Phase 2: each list is traversed so that the corresponding inverted list can be encoded and written
  - encoding: 2000 sec
  - writing: 200 sec
- Total time $= Bt_r + Ft_p + I(t_d + t_r)$
- $\sim$ 6 hours

Memory-based Inversion: Costs

- Memory space requirements
  - each node in each list of doc numbers typically requires 10 bytes:
    - 4 for the doc number d
    - 4 for the "next" pointer
    - 2 or more for the frequency count $f_{d,t}$
- For the example doc collection there are 400 million nodes
  - 4 Gb of memory
  - unrealistic amount . . .
- Why not put the linked list of doc numbers from memory onto disk?

Memory-based Inversion: Disk-based

- Phase one: sequence of disk accesses is sequential
  - Generation of the threaded file containing the linked lists is largely unaffected
  - Each new node results in a record being appended to a file, so a file of 4 Gb is created in sequential fashion on disk ($\sim$ 30 min's)
- Second phase, when each list is traversed
  - stored list nodes are interleaved in the same order on disk as they appeared in the text
  - each node access requires a random seek into the file on disk
  - at assumed disk seek time of 10 millisecs/seek, with 10 bytes to be read/record, this is 4 million seconds

Memory-based Inversion: Disk-based

- Inversion time
  - $Bt_r + Ft_p + 10ft_r + ft_s + 10ft_r + I(t_d + t_r)$
- For gigabyte collections, linked-list approaches are inadequate because of memory and/or time requirements
- For small collections it is the best method though
  - For the **Bible**, in-memory inversion takes half a minute and requires about 10 Mb of main memory

Sort-Based Inversion

- Main problems with the two methods discussed so far
  - require too much memory
  - use data access sequence that is random, preventing an efficient mapping from memory onto disk
- **For large disk files, sequential access is the only efficient processing mode since transfer rates are usually high and random seeks are time-consuming**
- Moreover, for large volumes of data, the use of disk is inescapable
  - $\longrightarrow$ inversion should perform sequential processing on whatever disk files are required
  - $\longrightarrow$ **sort-based inversion**

Sort-Based Inversion

1. `/* Initialization */`

   Create an empty dictionary structure S
   Create an empty temporary file on disk

2. `/*Proces text and write temporary file */`

   For each document D_d in the collection, $1 \leq d \leq\leq N$

   (a) Read D_d, parsing it into index terms
   (b) For each index term $t \in D_d$
      i. Let $f_{d,t}$ be the frequency in D_d of term t
      ii. Search S for t
      iii. If t is not in S, insert it
      iv. Write record $(t, d, f_{d,t})$ to the temporary file, where t is represented by its term number in S

Sort-Based Inversion

3. `/* Internal sorting to make runs */`

   Let k be the number of records that can be held in memory

   (a) Read k records from the temporary file
   (b) Sort into nondecreasing t order, and for equal values of t, nondecreasing d order
   (c) Write the sorted run back to the temporary file
   (d) Repeat until there are no more runs to be sorted

4. `/* Merging */`

   Pairwise merge runs in the temporary file until it is one sorted run

Sort-Based Inversion

5. `/* Output inverted file */`

   For each term $1 \leq t \leq n$

   (a) Start a new inverted file entry
   (b) Read all triples $(t, d, f_{d,t})$ from the temporary file and form the inverted file entry for term t
   (c) If required, compress the inverted file entry
   (d) Append this inverted file entry to the inverted file

Sort-Based Inversion: Example

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

Initial		Sorted runs		Merged runs (fully sorted)
⟨1,1,2⟩		⟨1,1,2⟩		⟨1,1,2⟩
⟨2,1,2⟩		⟨1,2,1⟩		⟨1,2,1⟩
⟨3,1,1⟩		⟨2,1,2⟩		⟨2,1,2⟩
⟨4,1,1⟩		⟨2,2,1⟩		⟨2,2,1⟩
⟨1,2,1⟩		⟨3,1,1⟩		⟨3,1,1⟩
⟨2,2,1⟩		⟨4,1,1⟩		⟨3,4,1⟩
⟨5,2,1⟩		⟨5,2,1⟩		⟨4,1,1⟩
⟨6,2,1⟩		⟨6,2,1⟩		⟨4,4,1⟩
⟨7,2,1⟩		⟨7,2,1⟩		⟨5,2,1⟩
⟨8,3,1⟩		⟨8,3,1⟩		⟨5,5,1⟩
⟨9,3,1⟩		⟨9,3,1⟩		⟨6,2,1⟩
⟨10,3,1⟩	→	⟨10,3,1⟩	→	⟨6,5,1⟩
⟨11,4,2⟩		⟨11,4,2⟩		⟨7,2,1⟩
⟨12,4,2⟩		⟨12,4,2⟩		⟨7,5,1⟩
⟨13,4,2⟩		⟨3,4,1⟩		⟨8,3,1⟩
⟨3,4,1⟩		⟨4,4,1⟩		⟨8,6,1⟩
⟨4,4,1⟩		⟨5,5,1⟩		⟨9,3,1⟩
⟨11,5,1⟩		⟨11,5,1⟩		⟨9,6,1⟩
⟨12,5,1⟩		⟨12,5,1⟩		⟨10,3,1⟩
⟨13,5,1⟩		⟨13,4,2⟩		⟨10,6,1⟩
⟨5,5,1⟩		⟨13,5,1⟩		⟨11,4,2⟩
⟨6,5,1⟩		⟨6,5,1⟩		⟨11,5,1⟩
⟨7,5,1⟩		⟨7,5,1⟩		⟨12,4,2⟩
⟨8,6,1⟩		⟨8,6,1⟩		⟨12,5,1⟩
⟨9,6,1⟩		⟨9,6,1⟩		⟨13,4,2⟩
⟨10,6,1⟩		⟨10,6,1⟩		⟨13,5,1⟩

Sort-Based Inversion: Costs . . . Time

- Read and parse, write file
  - $Bt_r + Ft_p + 10ft_r$
- Sort runs
  - $20ft_r + R(1.2k \log k)t_c$
- Merge runs
  - $\lceil \log R \rceil (20ft_r + ft_c)$
- Write compressed inverted file
  - $10ft_r + I(t_d + t_r)$
- $\sim$ 20 hours, using 40 Mb of main memory

Sort-Based Inversion: Costs . . . Space

- The sorting algorithm requires **two** copies of the data at any given time
- Halfway during the last merge:
  - Two runs are being merged, each appr half the size of the original file
  - At the halfway stage of the merge, both of these runs have been partially consumed
  - Because of this, the merged output cannot be written sequentially back to the same file since it might overwrite data yet to be processed
  - At the last instant, just before this merge finishes, the output contains all of the records being sorted, and so do the two input files

Sort-Based Inversion: Costs . . . Space

- So, two temporary input files must be allowed for
  - For the example inversion, each of these contains $\mathbf{10} \times \mathbf{400}$ million bytes $\longrightarrow$ 8 Gb
- Simple sort-based inversion is the best method for moderate sized collections (10–100 Mb range), but not suitable for truly large collections

What Have We Done Today?

- Index construction
- Components
- Memory-Based algorithms
- Sort-Based algorithms

Lab Session

- Experiment with indexing
  - Input: Test collection
  - Output: Index

