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 $O(\text{characters in collection})$
  - 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

<table>
<thead>
<tr>
<th>Document</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pease porridge hot, pease porridge cold</td>
</tr>
<tr>
<td>2</td>
<td>Pease porridge in the pot</td>
</tr>
<tr>
<td>3</td>
<td>Nine days old</td>
</tr>
<tr>
<td>4</td>
<td>Some like it hot, some like it cold</td>
</tr>
<tr>
<td>5</td>
<td>Some like it in the pot</td>
</tr>
<tr>
<td>6</td>
<td>Nine days old</td>
</tr>
</tbody>
</table>
# Inverted Files

<table>
<thead>
<tr>
<th>Document</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pease porridge hot, pease porridge cold</td>
</tr>
<tr>
<td>2</td>
<td>Pease porridge in the pot</td>
</tr>
<tr>
<td>3</td>
<td>Nine days old</td>
</tr>
<tr>
<td>4</td>
<td>Some like it hot, some like it cold</td>
</tr>
<tr>
<td>5</td>
<td>Some like it in the pot</td>
</tr>
<tr>
<td>6</td>
<td>Nine days old</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Text</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cold</td>
<td>1, 4</td>
</tr>
<tr>
<td>2</td>
<td>days</td>
<td>3, 6</td>
</tr>
<tr>
<td>3</td>
<td>hot</td>
<td>1, 4</td>
</tr>
<tr>
<td>4</td>
<td>in</td>
<td>2, 5</td>
</tr>
<tr>
<td>5</td>
<td>it</td>
<td>4, 5</td>
</tr>
<tr>
<td>6</td>
<td>like</td>
<td>4, 5</td>
</tr>
<tr>
<td>7</td>
<td>nine</td>
<td>3, 6</td>
</tr>
<tr>
<td>8</td>
<td>old</td>
<td>3, 6</td>
</tr>
<tr>
<td>9</td>
<td>pease</td>
<td>1, 2</td>
</tr>
<tr>
<td>10</td>
<td>porridge</td>
<td>1, 2</td>
</tr>
<tr>
<td>11</td>
<td>pot</td>
<td>2, 5</td>
</tr>
<tr>
<td>12</td>
<td>some</td>
<td>4, 5</td>
</tr>
<tr>
<td>13</td>
<td>the</td>
<td>2, 5</td>
</tr>
</tbody>
</table>
## Word-Level Inverted File

<table>
<thead>
<tr>
<th>Document</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pease porridge hot, pease porridge cold</td>
</tr>
<tr>
<td>2</td>
<td>Pease porridge in the pot</td>
</tr>
<tr>
<td>3</td>
<td>Nine days old</td>
</tr>
<tr>
<td>4</td>
<td>Some like it hot, some like it cold</td>
</tr>
<tr>
<td>5</td>
<td>Some like it in the pot</td>
</tr>
<tr>
<td>6</td>
<td>Nine days old</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Text</th>
<th>(Document; Word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cold</td>
<td>(1; 6), (4; 8)</td>
</tr>
<tr>
<td>2</td>
<td>days</td>
<td>(3; 2), (6; 2)</td>
</tr>
<tr>
<td>3</td>
<td>hot</td>
<td>(1; 3), (4; 4)</td>
</tr>
<tr>
<td>4</td>
<td>in</td>
<td>(2; 3), (5; 4)</td>
</tr>
<tr>
<td>5</td>
<td>it</td>
<td>(4; 3, 7), (5; 3)</td>
</tr>
<tr>
<td>6</td>
<td>like</td>
<td>(4; 2, 6), (5; 2)</td>
</tr>
<tr>
<td>7</td>
<td>nine</td>
<td>(3; 1), (6; 1)</td>
</tr>
<tr>
<td>8</td>
<td>old</td>
<td>(3; 3), (6; 3)</td>
</tr>
<tr>
<td>9</td>
<td>pease</td>
<td>(1; 1, 4), (2; 1)</td>
</tr>
<tr>
<td>10</td>
<td>porridge</td>
<td>(1; 2, 5), (2; 2)</td>
</tr>
<tr>
<td>11</td>
<td>pot</td>
<td>(2; 5), (5; 6)</td>
</tr>
<tr>
<td>12</td>
<td>some</td>
<td>(4; 1, 5), (5; 1)</td>
</tr>
<tr>
<td>13</td>
<td>the</td>
<td>(2; 4), (5; 5)</td>
</tr>
</tbody>
</table>
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 disadvantage 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 $i$-th 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: $\textbf{B}^+$-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 $= \text{height of the tree}$
Hashing

- A hashing function $h(x)$ maps a key $x$ to an integer in a given range; e.g., $0$ 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Assumed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total text size</td>
<td>$B$</td>
<td>$5 \times 10^9$ bytes</td>
</tr>
<tr>
<td>Number of docs</td>
<td>$N$</td>
<td>$5 \times 10^6$</td>
</tr>
<tr>
<td>Number of distinct words</td>
<td>$n$</td>
<td>$1 \times 10^6$</td>
</tr>
<tr>
<td>Total number of words</td>
<td>$F$</td>
<td>$800 \times 10^6$</td>
</tr>
<tr>
<td>Number of index pointers</td>
<td>$f$</td>
<td>$400 \times 10^6$</td>
</tr>
<tr>
<td>Final size of compressed inv. file</td>
<td>$I$</td>
<td>$400 \times 10^6$ bytes</td>
</tr>
<tr>
<td>Disk seek time</td>
<td>$t_s$</td>
<td>$10 \times 10^{-3}$ sec</td>
</tr>
<tr>
<td>Disk transfer time per byte</td>
<td>$t_r$</td>
<td>$0.5 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Inverted file coding per byte</td>
<td>$t_d$</td>
<td>$5 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Time to compare and swap 10-byte records</td>
<td>$t_c$</td>
<td>$10^{-6}$ sec</td>
</tr>
<tr>
<td>Time to parse, stem and look up one term</td>
<td>$t_p$</td>
<td>$20 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Amount of main memory available</td>
<td>$M$</td>
<td>$40 \times 10^6$ bytes</td>
</tr>
</tbody>
</table>
Main memory requirements, disk space requirements beyond what is needed to store the inverted index

<table>
<thead>
<tr>
<th>Method</th>
<th>Memory (Mb)</th>
<th>Disk (Mb)</th>
<th>Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked lists (memory)</td>
<td>4000</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Linked lists (disk)</td>
<td>30</td>
<td>4000</td>
<td>1100</td>
</tr>
<tr>
<td>Sort-based</td>
<td>40</td>
<td>8000</td>
<td>20</td>
</tr>
<tr>
<td>Sort-based (compressed)</td>
<td>40</td>
<td>680</td>
<td>26</td>
</tr>
<tr>
<td>Sort-based (multiway merge)</td>
<td>40</td>
<td>540</td>
<td>11</td>
</tr>
<tr>
<td>Sort-based (multiway in-place)</td>
<td>40</td>
<td>150</td>
<td>11</td>
</tr>
<tr>
<td>Text-based partition</td>
<td>40</td>
<td>35</td>
<td>15</td>
</tr>
</tbody>
</table>
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 dictionary 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 ≤ t ≤ 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 (≈ 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
  - inversion should perform sequential processing on whatever disk files are required
  - sort-based inversion
Sort-Based Inversion

1. /* Initialization */

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

2. /* Process text and write temporary file */

   For each document $D_d$ 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. 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

<table>
<thead>
<tr>
<th>Term</th>
<th>Term number</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>4</td>
</tr>
<tr>
<td>days</td>
<td>9</td>
</tr>
<tr>
<td>hot</td>
<td>3</td>
</tr>
<tr>
<td>in</td>
<td>5</td>
</tr>
<tr>
<td>it</td>
<td>13</td>
</tr>
<tr>
<td>like</td>
<td>12</td>
</tr>
<tr>
<td>nine</td>
<td>8</td>
</tr>
<tr>
<td>old</td>
<td>10</td>
</tr>
<tr>
<td>peas</td>
<td>1</td>
</tr>
<tr>
<td>porridge</td>
<td>2</td>
</tr>
<tr>
<td>pot</td>
<td>7</td>
</tr>
<tr>
<td>some</td>
<td>11</td>
</tr>
<tr>
<td>the</td>
<td>6</td>
</tr>
</tbody>
</table>

Initial | Sorted runs | Merged runs (fully sorted)
Sort-Based Inversion: Costs . . . Time

- Read and parse, write file
  - $Bt_r + Ft_p + 10 ft_r$

- Sort runs
  - $20 ft_r + R(1.2k \log k)t_c$

- Merge runs
  - $\lceil \log R \rceil (20 ft_r + ft_c)$

- Write compressed inverted file
  - $10 ft_r + I(t_d + t_r)$

- $\sim 20$ 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 $10 \times 400$ million bytes $\rightarrow 8$ 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