file organization

many slides courtesy James Allan@umass
file organization

• indexes
• bitmaps
• signature files
• inverted files
• wildcards
file organization

Choices for accessing data during query evaluation

• Scan the entire collection
  – Typical in early (batch) retrieval systems
  – Computational and I/O costs are \( O(\text{characters in collection}) \)
  – Practical for only small text collections
  – Large memory systems make scanning feasible

• Use indexes for direct access
  – Evaluation time \( O(\text{query term occurrences in collection}) \)
  – Practical for large collections
  – Many opportunities for optimization
  – ML(Nearest Neighbor) research with image databases shows that indexes can be difficult

• Hybrids: Use small index, then scan a subset of the collection
indexes

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

• IR Problem: Cannot predict keys that people will use in queries
  – Every word in a document is a potential search term

• IR Solution: Index by all keys (words)
indexes

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 and uncontrolled vocabulary)
  - Document structure (sentence and paragraph boundaries)
  - Inter- or intra-document links (e.g., citations)

- Composed features
  - Feature sequences (phrases, names, dates, monetary amounts)
  - Feature sets (e.g., synonym classes)

- Indexing and retrieval models drive choices
  - Must be able to construct all components of those models
Indexing choices (there is no “right” answer)

• What is a word (tokenization)?
  – 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?
indexes

The contents depend upon the retrieval model

• Feature presence/absence
  – Boolean
  – Statistical \((tf, df, ctf, doclen, maxtf)\)
  – Often about 10% the size of the raw data, compressed

• Positional
  – Feature location \textit{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
indexes: implementation

Common implementations of indexes
• Bitmaps
• Signature files
• Inverted files

Common index components
• Dictionary (lexicon)
• Postings
  – document ids
  – word positions
indexes: bitmaps

- Bag-of-words index only
- For each term, allocate vector with one bit per document
  - If feature present in document $n$, set $n$th bit to 1, otherwise 0
- Boolean operations very fast
- Space efficient for common terms (why?)
- Space inefficient for rare terms (why?)
- Good compression with run-length encoding (why?)
- Difficult to add/delete documents (why?)
- Not widely used
indexes: signature files

- Bag-of-words only
- Also called *superimposed coding*
- For each term, allocate fixed size $s$-bit vector($signature$)

- Define hash function:
  - Single function: word $\rightarrow 1..2^s$ [sets all $s$-bits]
  - Multiple functions: word $\rightarrow 1..s$ [selects which bits to set]
- Each term has an $s$-bit signature
  - may not be unique!

- OR the term signatures to form document signature
- Long documents are a problem (why?)
  - Usually segment them into smaller pieces
indexes: signature files

<table>
<thead>
<tr>
<th>Term</th>
<th>Hash string</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>1000 0000 0010 0100</td>
</tr>
<tr>
<td>days</td>
<td>0010 0100 0000 1000</td>
</tr>
<tr>
<td>hot</td>
<td>0000 1010 0000 0000</td>
</tr>
<tr>
<td>in</td>
<td>0000 1001 0010 0000</td>
</tr>
<tr>
<td>it</td>
<td>0000 1000 1000 0010</td>
</tr>
<tr>
<td>like</td>
<td>0100 0010 0000 0001</td>
</tr>
<tr>
<td>nine</td>
<td>0010 1000 0000 0100</td>
</tr>
<tr>
<td>old</td>
<td>1000 1000 0100 0000</td>
</tr>
<tr>
<td>pease</td>
<td>0000 0101 0000 0001</td>
</tr>
<tr>
<td>porridge</td>
<td>0100 0100 0010 0000</td>
</tr>
<tr>
<td>pot</td>
<td>0000 0010 0110 0000</td>
</tr>
<tr>
<td>some</td>
<td>0100 0100 0000 0001</td>
</tr>
<tr>
<td>the</td>
<td>1010 1000 0000 0000</td>
</tr>
</tbody>
</table>

16 bit signatures
indexes: signature files

<table>
<thead>
<tr>
<th>Document</th>
<th>Text</th>
<th>Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pease porridge hot, pease porridge cold,</td>
<td>1100 1111 0010 0101</td>
</tr>
<tr>
<td>2</td>
<td>Pease porridge in the pot,</td>
<td>1110 1111 0110 0001</td>
</tr>
<tr>
<td>3</td>
<td>Nine days old.</td>
<td>1010 1100 0100 1100</td>
</tr>
<tr>
<td>4</td>
<td>Some like it hot, some like it cold,</td>
<td>1100 1110 1010 0111</td>
</tr>
<tr>
<td>5</td>
<td>Some like it in the pot,</td>
<td>1110 1111 1110 0011</td>
</tr>
<tr>
<td>6</td>
<td>Nine days old.</td>
<td>1010 1100 0100 1100</td>
</tr>
</tbody>
</table>
indexes: signature files

• At query time:
  – Lookup signature for query (how?)
  – If all corresponding 1-bits are “on” in document signature, document probably contains that term

• Vary $s$ to control $\Pr[\text{false alarm}]$
  – Note space tradeoff
• Optimal $s$ changes as collection grows (why?)

• Many variations
• Widely studied
• Not widely used
signature files trivia

• Punch card as “card catalogue”
• Punch out bits of signature
  – Tolkein = 4, 11, & 14
  – Harper = 4, 14, 19
  – Lee = 11, 18
  – ...

• To find an item
  – Calculate its signature
  – Run rods through “bits”
  – “Harper Lee” is 5 bits
• “Tolkein” is 3 bits

Tolkein, J.R.R.
The Two Towers
PR6039.O32 L6 1966 v.2

Lee, Harper
To Kill a Mockingbird
PS3562.E353 T64 1960
Inverted lists are currently 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

<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>

Example text; each line is one document.
## 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>Term</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>
inverted files: word level

<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>Term</th>
<th>(Document; Words)</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>
indexes and language models

• Assume query likelihood approach

\[ P(q_1, \ldots, q_k|MD) = \prod_{i=1}^{k} P(q_i|MD) \]

• Jelinek-Mercer smoothing for each query term

\[ P(q_t|MD) = \lambda ML_{estim}(q_t|MD) + (1-\lambda) BKGRND_{prob} \]

• Probably use logs to avoid tiny numbers

\[ \log P(q_1, \ldots, q_k|MD) = \sum_{i=1}^{k} \log P(q_i|MD) \]
document-based approach

• For each document $D$ in collection
  – Calculate $\log P(Q|MD)$

• Sort scores
• Drawbacks
  – Most documents have no query terms
  – Very slow
using inverted files

• Simple approach to using inverted list

• Use list to find documents containing any query term
  – All others assumed to have low and constant probability

• For each document in that pool
  – Calculate log P(Q|MD)

• Sort scores
• Better
  – Only plausible documents considered
  – Still requires accessing entire document
better use of inverted files

• Recall score being calculated

\[ \log P(q_1, \ldots, q_k | M_D) = \sum_{i=1}^{k} \log P(q_i | M_D) \]

• Can be done in parts
  – Do \( q_1 \) for every document that contains \( q_1 \)
  – Then \( q_2 \) then \( q_3 \) then ...

• Keep array Score[] with cumulative scores
better use of inverted files

• For each query word \( q_i \)
  – Fetch its inverted list
  – For each document \( D_k \) in list

• Calculate \( \log P(q_i \mid D_k) \)
• Add to accumulator Score\([k]\)

• Sort array
  – Or keep running list of top \( n \) documents
• Why do the calculation?
  – Store pre-computed \( \log P(q_i \mid D_k) \) in list
  – Or store partial results if necessary

• May have to smooth at query time
inverted lists: 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
inverted lists: access optimizations

• Skip lists:
  – A table of contents to the inverted list
  – Embedded pointers that jump ahead \( n \) documents

• Separating presence information from location information
  – Many operators only need presence information
  – Location information takes substantial space (I/O)
  – If split,
    • reduced I/O for presence operators
    • increased I/O for location operators (or larger index)
  – Common in CD-ROM implementations
wildcard matching

- $X^*$ is probably easy (why? when not?)
- What about $*X$, $*X^*$, $X^*Y$?

- Permuterm index
  - Prefix each term $X$ with a $\|$.
  - Rotate each augmented term cyclically (with wraparound) by one character, to produce $n$ new terms.
  - Append an $\|$ to the end of each word form.
  - Insert all forms in the dictionary.

- Lookup
  - $X$: search for $\|X\|$
  - $X^*$: search for all terms beginning with $\|X$
  - $*X$: search for all terms beginning with $X\|$
  - $*X^*$: search for all terms beginning with $X$
  - $X^*Y$: search for all terms beginning with $Y\|X$
building indexes

Indexes 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 and 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
indexes: summary

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Inverted Files</th>
<th>Bitmaps</th>
<th>Signature Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of update (edit a doc)</td>
<td>-</td>
<td>+</td>
<td>+–</td>
</tr>
<tr>
<td>Query evaluation speed</td>
<td>+</td>
<td>+–</td>
<td>+–</td>
</tr>
<tr>
<td>Uncompressed space efficiency</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Compressed space efficiency</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Index fidelity</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Can store word positions</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>