Data structures in Information Retrieval

Building index
Build index
(pseudocode)

Index = new Index()
for document in (collection):
    for word in (document):
        index[word].insert(document.id);
Questions

1. Implementation iterators/extractors
2. Indexing big collections/speed
3. Compression of indexes
4. Updating index
5. Parallelization/cluster use
Linearization (word extraction)
Build one piece of index

Word, ID

1, 30, 45, 77, 134, 9099, ...

22, 44, 77, 132, ...

13
Post lists in memory

Growing array: 1, 30, 45, 77, 134, 9099, ...

List: 1 • 30 • 45 • 77 •

List of arrays: 1, 30, 45 • 77, 134, 9099 •

List of adapt. arrays: 1 • 30, 45 • 77, 134, 9099, 10056 •

Decrease number of allocations, avoid fragmentation
Build big indexes

RAM

1, 30, 45, 77, 134, 9099, ...

22, 44, 77, 132, ...

13

Temp dumps
Temporary dump

RAM

1,30,45,77,134,9099,…

22,44,77,132,…

13

... 20 “B” 22,5,3 3 “AB” 18,5 2 “A”
Multiway merge
[merging dumps]

1. Load first words into memory
2. Select smallest words
3. Merge post-lists for the words
4. Move to next words in pr. dumps
5. Remove empty dumps
6. Goto 2 if dumps are not empty
Multiway merge
[example]

```
...  20
22,5,3  3  "AB"  18,5  2  "A"
...
50  "B"  99,50  2  "A"
```

```
"A"  5,18,50,99
"AB"  3,5,22
"B"  ...
```
Multiway merge
[summary]

1. Linear number of operations
2. Fixed amount of RAM in memory
3. Linear access to data
Compression

Data → Compression model → States (probability) → Encoder → Enc. Data
Model for post-lists


postings
Model for post-lists

\[ N = \text{ID}_{\text{cur}} - \text{ID}_{\text{prev}} - 1 \]
Universal code

• Binary code
• Prefix code (without out-of-band markers)
• If integers distribution \( p(i+1) < p(i) \) →
  \( E(\text{codelength}) = E(\text{optimal\_code}) + C \)
Unary code

<table>
<thead>
<tr>
<th>n</th>
<th>Unary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>001</td>
</tr>
<tr>
<td>4</td>
<td>0001</td>
</tr>
<tr>
<td>5</td>
<td>00001</td>
</tr>
<tr>
<td>6</td>
<td>000001</td>
</tr>
<tr>
<td>7</td>
<td>0000001</td>
</tr>
<tr>
<td>8</td>
<td>00000001</td>
</tr>
<tr>
<td>9</td>
<td>000000001</td>
</tr>
<tr>
<td>10</td>
<td>0000000001</td>
</tr>
</tbody>
</table>
Elias code

Beta code:
1. Unary code for \([\log_2 i]\)
2. \(i\) in binary without leading 1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
</tr>
</tbody>
</table>
Elias $\omega$ code (universal case)

\[
\begin{align*}
\text{write}(0) \\
\text{while}([\log_2 i]! = 0) \\
\quad \text{prefix}(C_b(i)) \\
\quad i = [\log_2 i]
\end{align*}
\]
### Rice-Golomb code

<table>
<thead>
<tr>
<th>Value</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 01</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1 10</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1 11</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0 1 00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0 1 01</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0 1 10</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0 1 11</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>00 1 00</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>1</td>
<td>00 1 01</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>00 1 10</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>3</td>
<td>00 1 11</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>0</td>
<td>000 1 00</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>1</td>
<td>000 1 01</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>2</td>
<td>000 1 10</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>3</td>
<td>000 1 11</td>
</tr>
</tbody>
</table>

**Param Q**

- $X:Q \rightarrow$ unary code
- $X \% Q \rightarrow$ binary code

Optimal for geometric distribution
Variable-byte code

0 0 0 0 1 1 1 1

= 15

0 0 0 0 1 0 1 0
1 0 1 0 0 1 0 1

= 1317

0 0 1 0 1 0 0 0
1 0 1 0 0 0 0 1
1 0 1 0 0 1 0 1

= 329893
Block coding

1. Divide post-list into blocks
2. Encode every block independently
3. Use all postings in block (more information)

How to define block:

• Fixed number of postings
• Fixed encoded size
Interpolative encoding

Example of **block coding**: 
1. Encode first post (Elias, etc...) 
2. Encode delta with last code in block 
3. Encode middle with truncated binary 
4. Repeat recursively prev. step
Interpolative encoding (2)
Interpolative encoding(3)
Index update polices

• Full rebuild (bulk data load)
• Rebuild/merge strategies
• In-place update
Full rebuild

Conditions:
1. Static collection
2. Big changes in collection
3. Indexing and Search are physically separated
Logarithm merge
[rebuild/merge]
Inverted index over B-Tree

- Well-studied “classical” structure
- Many good implementations
- Good solution “little memory” and “in-place update”

Tree slot:  

| Word | Block of post-list(compressed) |

Cmp. function:  

\[(w1 < w2) ? -1: (w1 == w2) ? d1 - d2 : 1\]
Building Inverted Index over B-Tree

Full rebuild B-Tree bulk load:

As effective as any other approach
Update B-Tree in-place
Building in cluster

• Distribute process over cluster
• Effectively use resources
• Failure handling
Channels

Controller

Generator

Channel adapter

Copy

Data, Marker

Consumer

Channel adapter

notification
Channel adapter logic

1. Find consumer
2. Sending to consumer and making copy
3. Wait for notification
4. If consumer failure Goto 1
Build index with channels

Linearization → Indexer → Merger
Map Reduce

• Use ideas of functional programming
• Simple paradigm, well understandable for programmers
• Match algorithms
Functional programming

• Avoids states and side effects
• “Pure functions” – get structures and return new structures

**Map**(fun,list) - apply fun for every element and return new list

**Reduce**(fun,list) – apply fun(prevVal,list_elem) to every list element and return fun. result
Build index in map-reduce [map]

def map(document):
    for each word w in document.text:
        Emit (w, (document.id,));
Build index in map-reduce
[reduce]

Def Reduce(Word,ids_list)

    joined_list = ()
    for lst in ids_list:
        joined_list.append(joined_list)

    Emit(Word,joined_list)
Map-reduce environment

- Distribute tasks in cluster
- Move elements of list (group by key)
- Restart tasks in case of failure
Summary

• Build index in memory and merge
• Compress post-lists
• Build in cluster
Q&A