Index Construction
Inverted Indexes

Query “Brutus” AND “Calpurnia”

<table>
<thead>
<tr>
<th>Brutus</th>
<th>→</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>11</th>
<th>31</th>
<th>45</th>
<th>173</th>
<th>174</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar</td>
<td>→</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>16</td>
<td>57</td>
<td>132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>→</td>
<td>2</td>
<td>31</td>
<td>54</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dictionary

Postings
Blocked Sort-Based Indexing (BSBI)

- Divide text collection into blocks
  - Each block can be held into main memory

- For each block
  - Sort the termID-docID pairs in the block in main memory
  - Store intermediate sorted results on disk

- Merge all intermediate results into the final result

```
BSB INDEX CONSTRUCTION()
1  n ← 0
2  while (all documents have not been processed)
3    do n ← n + 1
4       block ← PARSE NEXT BLOCK()
5       BSBI-INVERT(block)
6       WRITE BLOCK TO DISK(block, fn)
7       MERGE BLOCKS(f1, ..., fn; f merged)
```
Example

Building inverted index in main memory

Merging

postings lists to be merged

brutus d1,d3
cæsar d1,d2,d4
noble d5
with d1,d2,d3,d5

brutus d6,d7
cæsar d8,d9
julius d10
killed d8

merged postings lists

brutus d1,d3,d6,d7
cæsar d1,d2,d4,d8,d9
julius d10
killed d8
noble d5
with d1,d2,d3,d5

disk
Cost Analysis

• Time complexity: $\Theta(T \log T)$
  – $T$: the maximum number of termID-docID pairs that can be held into main memory
  – Theoretical bottleneck: sorting termID-docID pairs within a block

• Practical bottleneck
  – The time parsing documents
  – Final merge step

• Assumption: the dictionary can be held into main memory so that termID can be obtained online for each document
Single-Pass in-Memory Indexing

• For very large text collections, the dictionary may not be held in main memory

• Major ideas
  – Create a dictionary for each block using a hash function
  – Add a posting directly to its posting list – no sorting or storage of termID-docID pairs
Algorithm

```plaintext
SPIMI-INVERT(token_stream)
1  output_file = NewFile()
2  dictionary = NewHash()
3  while (free memory available)
4    do token ← next(token_stream)
5      if term(token) ∉ dictionary
6          then postings_list = AddToDictionary(dictionary, term(token))
7          else postings_list = GetPostingsList(dictionary, term(token))
8      if full(postings_list)
9          then postings_list = DoublePostingsList(dictionary, term(token))
10         AddToPostingsList(postings_list, docID(token))
11    sorted_terms ← SortTerms(dictionary)
12   WriteBlockToDisk(sorted_terms, dictionary, output_file)
13 return output_file
```
Complexity

• Time complexity: $\Theta(T)$
  – SPIMI can index collections of any size as long as there is enough disk space available

• Classroom discussion: the time complexity is based on some assumption, what is that?
  – The number of terms is much smaller than $|T|$
  – In the extreme case where every term in the block is new, the complexity is still $\Theta(T \log T)$
MapReduce

- A two phase-process
  - Mapping: each mapper independently processes a chunk of data and generate a set of data entries with keys
  - Shuffling (done by masters): shuffle the data entries generated by mappers and assign data entries of the same key to the same reducers
  - Reducing: each reducer independently processes a chunk of entries assigned and generates output
- Idempotence: if the mapper or reducer is called multiple times on the same input, the output will always be the same
  - Fault-tolerance: if a mapper or reducer fails or just slow, the same job can be assigned to some other machine
MapReduce
MapReduce Index Construction

splits

assign

master

assign

postings

map
phase

segment
files

reduce
phase

parser

parser

parser

a-f g-p q-z

a-f g-p q-z

a-f g-p q-z

inverter

inverter

inverter

a-f

g-p

q-z
Pseudocode

- Mapper generates pairs (word, document:position)
- Reducer generates posting lists for words

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, document:position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```
Incremental Maintenance

• The text collection may change over time
  – Insertion: new documents may be added
  – Deletion: some documents may be removed

• Index merging
  – Data $D = D_1$ (original data) $\cup \Delta D$ (new data)
  – Index $I$ for $D_1$, build index $\Delta I$ for $\Delta D$
  – Merge $I$ and $\Delta I$
  – Particularly useful when updates come in large batches (e.g., thousands of documents at a time)
  – Inefficient if updates comes in small batches (e.g., one new document at a time)
Result Merging

- Build a small index for the new data, but not merge it into the large index
  - The new small index can be held in main memory and thus is easy to update
- Queries are evaluated separately against the small index and the large index
  - The results lists are merged
  - A deleted document list can be used to handle deleted documents
Using Multiple Indexes

• Using too many indexes slows down query processing
• Using too few indexes slows down index construction throughput due to excessive disk traffic
• Geometric partitioning
  – $I_0$ contains as much data as can be fit into main memory
  – $I_1$ contains $r$ times as much data as $I_0$
  – $I_m$ contains between $n \times r^m$ and $n \times r^{m+1}$ bytes of data, where $n$ is the main memory size
  – If $r = 2$, it is called logarithmic merging and can hold $1000n$ bytes of index data using 10 indexes
Logarithmic Merging Algorithm

\textbf{L\textsc{mergeAddToken}}(indexes, \(Z_0\), \textit{token})
\begin{verbatim}
1  \(Z_0 \leftarrow \text{Merge}(Z_0, \{\text{token}\})\)
2  \textbf{if} \(|Z_0| = n\)
3  \textbf{then} \textbf{for} \(i \leftarrow 0\) \textbf{to} \(\infty\)
4    \textbf{do} \textbf{if} \(I_i \in \text{indexes}\)
5      \textbf{then} \(Z_{i+1} \leftarrow \text{Merge}(I_i, Z_i)\)
6      \hspace{1em} (\(Z_{i+1}\) is a temporary index on disk.)
7      \textbf{indexes} \leftarrow \text{indexes} \setminus \{I_i\}
8    \textbf{else} \(I_i \leftarrow Z_i\) \hspace{1em} (\(Z_i\) becomes the permanent index \(I_i\).)
9      \textbf{indexes} \leftarrow \text{indexes} \cup \{I_i\}\n10     \textbf{Break}\n11  \end{verbatim}
\(Z_0 \leftarrow \emptyset\)

\textbf{Logarithmic\textsc{merge}}()
\begin{verbatim}
1  \(Z_0 \leftarrow \emptyset\) \hspace{1em} (\(Z_0\) is the in-memory index.)
2  \text{indexes} \leftarrow \emptyset
3  \textbf{while} \text{true}
4  \textbf{do} \textsc{LmergeAddToken}(\text{indexes}, \(Z_0\), getNextToken())
\end{verbatim}
Vocabulary

• Inverted lists alone cannot answer any queries
  – We need a vocabulary to find the inverted list for a particularly term

• A naïve method – store each inverted list as a separate file named after the term
  – Millions of files are needed, most of them are very short – no file systems can handle such a huge number of files in an efficient way
Dictionary as a String

- Using 20 bytes for a term, 4 bytes for document frequency, and 4 bytes for pointer to postings list
  - To store 400,000 entries, we need $400,000 \times 28 = 11.2$ MB
  - Many words are much shorter than 20 bytes – wasting a lot of space in storing entries
- Inverted file: all inverted lists are stored together in a single file
  - A vocabulary contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - If the vocabulary is too big, use a tree-based index structure
Example

• Suppose on average a term has 8 bytes
  – We only need $400,000 \times 8 = 3.2$ MB to store terms

• We need $\log_2 3.2 \times 10^6 = 22$ bits $\approx 3$ bytes for each term pointer

• In total, we need $400,000 \times (4 + 4 + 3 + 8) = 7.6$ MB for the whole dictionary
  – Saving 3.6 MB space
Example

...systilesyzygeticsyzygialsyzygyszaibelyiteszecinszono...

<table>
<thead>
<tr>
<th>freq.</th>
<th>postings ptr.</th>
<th>term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

4 bytes 4 bytes 3 bytes
Blocked Storage

- Group terms in the string into blocks of size $k$ and keep a term pointer only for the first term of each block

```
... 7 systemSyzygetic 8 systemystery 11 system Szablonyite 6 Szecin...
```

<table>
<thead>
<tr>
<th>freq.</th>
<th>postings ptr.</th>
<th>term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Saving in Space

• In each block
  – Eliminate \((k - 1)\) pointers, when \(k = 4\), saving \((4 - 1) \times 3 = 9\) bytes
  – Need \(k\) bytes for the length of terms, when \(k = 4\), need 4 bytes
  – When \(k = 4\), saving 5 bytes per block, total saving = \(400,000 / 4 \times 5 = 0.5\) MB

• Effect of parameter \(k\)
  – Binary search on indexed terms, linear search within a block
  – Large \(k\) gains more savings in space, but slows down the search within a block
Front Coding

- Consecutive entries in an alphabetically sorted list share common prefixes
  - Common prefixes can be omitted
  - Save about 10% of the original size in many cases

One block in blocked compression ($k = 4$) ...

8 automata 8 automate 9 automatic 10 automation

... further compressed with front coding.

8 automate* a 1 e 2 ic 3 i on
Vocabulary in Galago

- A hybrid strategy
- A vocabulary is an abbreviated lookup table
  - One vocabulary entry for each 32K data
  - A 32TB inverted file requires less than 1GB of vocabulary space
  - The lists in the inverted file are stored in alphabetical order
- Lookup operation
  - Use binary search to find the nearest entry in the vocabulary, read the offset from that entry
  - Read 32KB of the inverted file starting at the offset using one disk seek
Summary

• Index construction on single machine
  – Blocked sorted based indexing
  – Single-pass in-memory indexing
• MapReduce index construction for computer clusters
• Incremental maintenance of inverted indexes
• Vocabulary organization
To-Do List

• Read Chapter 5.6
• Suppose that you want to count the distinct number of credit card numbers in a huge number of credit card transactions. How can the task be achieved using MapReduce?