Indexing and Ranking
Inverted Indexes

Query “Brutus” AND “Calpurnia”

<table>
<thead>
<tr>
<th>Term</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>Brutus</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td>31</td>
<td></td>
<td>45</td>
<td>173</td>
</tr>
<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>
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

```plaintext
BSBIIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3    do n ← n + 1
4      block ← PARSENextBlock()
5      BSBI-Invert(block)
6      WRITEBlockToDisk(block, f_n)
7      MERGEBlocks(f_1, ..., f_n; f_merged)
```
Example

Building inverted index in main memory

Merging

postings lists to be merged

merged postings lists

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

`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

- The time complexity is based on the assumption 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

parser

a-f g-p q-z

inverter

a-f

inverter

g-p

inverter

q-z

map phase

segment files

reduce phase

J. Pei: Information Retrieval and Web Search -- Indexing and Ranking
Pseudocode

• Mapper generates pairs (word, document:position)

• Reducer generates posting lists for words

```plaintext
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
  – I₀ contains as much data as can be fit into main memory
  – I₁ contains r times as much data as I₀
  – Iₘ contains between n x rᵐ and n x rᵐ⁺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

\texttt{LMergeAddToken(indexes, Z_0, token)}

1. \( Z_0 \leftarrow \text{Merge}(Z_0, \{\text{token}\}) \)
2. \( \text{if } |Z_0| = n \)
3. \( \text{then for } i \leftarrow 0 \text{ to } \infty \)
4. \( \text{do if } I_i \in \text{indexes} \)
5. \( \text{then } Z_{i+1} \leftarrow \text{Merge}(I_i, Z_i) \)
6. \( (Z_{i+1} \text{ is a temporary index on disk.}) \)
7. \( \text{indexes } \leftarrow \text{indexes } \setminus \{I_i\} \)
8. \( \text{else } I_i \leftarrow Z_i \) \( (Z_i \text{ becomes the permanent index } I_i.) \)
9. \( \text{indexes } \leftarrow \text{indexes } \cup \{I_i\} \)
10. \( \text{break} \)
11. \( Z_0 \leftarrow \emptyset \)

\texttt{LogarithmicMerge()}\n
1. \( Z_0 \leftarrow \emptyset \) \( (Z_0 \text{ is the in-memory index.}) \)
2. \( \text{indexes } \leftarrow \emptyset \)
3. \( \text{while } \text{true} \)
4. \( \text{do } \text{LMergeAddToken(indexes, Z_0, getNextToken())} \)
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

...stilesyzygeticssyzygialsyzygyszaibelyiteszecinszano...

freq. postings ptr. term ptr.
9 →
92 →
5 →
71 →
12 →
...
...
...
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

```
...7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin...
```

<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

\[\downarrow\]

… further compressed with front coding.

8 automate a1 e2 ic3 ion
To-Do List

• Read Chapter 4 and Section 5.2
• Exercises in Chapter 4, Exercises 5.2 and 5.3
• 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?
Why Compression?

- Reducing memory/disk requirement and accesses

Computer Memory Hierarchy

- Processor registers: very fast, very expensive
- Processor cache: very fast, expensive
- Random access memory: fast, affordable
- Flash/USB memory: slower, cheap
- Hard drives: slow, very cheap
- Tape backup: very slow, affordable
Tradeoff Between Space and Time

- A processor can process $p$ inverted list postings per second.
- A memory system can supply the processor with $m$ postings per second.
- The number of postings processed per second is $\min(m, p)$.
  - If $p > m$, the processor will sometimes wait.
  - If $m > p$, the memory system will sometimes be idle.
Tradeoff Between Space and Time

• Compression
  – Compression ratio is $r$ ($r \geq 1$) → the memory system can supply $m \times r$ postings per second
  – Decompression factor $d$ ($d \leq 1$) → the processor can process $d \times p$ postings per second
  – The larger $r$, the smaller $d$
  – Processing $\min(mr, dp)$ postings per second

• Task: maximizing $\min(mr, dp)$ and ensuring $m \times r \approx d \times p$
  – Classroom discussion: why do we want $m \times r \approx d \times p$?
Why Is Compression Possible?

- Representing common data elements with short codes, while representing uncommon data elements with longer codes.
- Inverted lists are lists of numbers, some are more frequent than the others.
  - Encode the frequent numbers with short codes and the infrequent numbers with longer codes.
Ambiguity

- Sequence “0, 1, 0, 3, 0, 2, 0” can be encoded as 00010011001000
- 0 appears 4 times – can we use one bit “0” to represent it?
  - Encoding: 0010110100 – saving 4 bits
  - Ambiguity: the decoding is not unique
  - One possible decoding other than the original sequence: 0, 0, 2, 3, 0, 1, 0, 0
- Unambiguous encoding
  - 0 – 0, 1 – 101, 2 – 110, 3 – 111
  - 0101011101100 – still saving 1 bit
Compressibility

- There is no such thing as an unambiguous code that can compress every possible input: some inputs will get bigger!
- Assumption: small numbers are more likely to occur than larger ones
  - Holding for word count data – power law distribution
  - Not holding for document numbers
  - Holding for the differences between document numbers
Delta Decoding

- Document number sequence: 1, 5, 9, 18, 23, 24, 30, 44, 45, 48
- Document number difference sequence: 1, 4, 4, 9, 5, 1, 6, 14, 1, 3
- Delta decoding: number sequence $\rightarrow$ number difference sequence
  - The differences are called d-gaps
- In posting lists
  - Frequent words have long posting lists where small d-gaps frequently appear
  - Rare words have short posting lists where big d-gaps appear a lot
Byte-Aligned Codes

- Code words are restricted to end on byte boundaries
  - Compressing and de-compressing faster
- Variable byte encoding (v-byte or VB)
  - The last 7 bits of a byte are “payload” and contain numeric data in binary
  - The high bit is a terminator/continuation bit – 1 for the last byte of an encoded gap and 0 otherwise
  - Decoding: read a sequence of bytes with continuation bit 0 terminated by a byte with termination bit 1, extract and concatenate the 7-bit part
# Space Requirement in V-Byte

<table>
<thead>
<tr>
<th>$k$</th>
<th>Number of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k &lt; 2^7$</td>
<td>1</td>
</tr>
<tr>
<td>$2^7 \leq k &lt; 2^{14}$</td>
<td>2</td>
</tr>
<tr>
<td>$2^{14} \leq k &lt; 2^{21}$</td>
<td>3</td>
</tr>
<tr>
<td>$2^{21} \leq k &lt; 2^{28}$</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k$</th>
<th>Binary Code</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0000001</td>
<td>81</td>
</tr>
<tr>
<td>6</td>
<td>1 0000110</td>
<td>86</td>
</tr>
<tr>
<td>127</td>
<td>1 1111111</td>
<td>FF</td>
</tr>
<tr>
<td>128</td>
<td>0 00000001 1 0000000</td>
<td>01 80</td>
</tr>
<tr>
<td>130</td>
<td>0 00000001 1 00000010</td>
<td>01 82</td>
</tr>
<tr>
<td>20000</td>
<td>0 00000001 0 0011100 1 0100000</td>
<td>01 1C A0</td>
</tr>
</tbody>
</table>
Bit-Aligned Codes

- The breaks between the coded regions can happen after any bit position
- Unary codes: using $k$ 1s to encode number $k$
  - Very efficient for small numbers like 0 and 1
  - Quickly become very expensive
- Can we combine the strengths of unary and binary codes?
  - Elias-$\gamma$ codes and Elias-$\delta$ codes
Elias-$\gamma$ Codes

- A number $k$ can be represented in two parts
  - Length: $k_d = \lfloor \log_2 k \rfloor$, the number of binary digits you need to write, encoded in unary codes
  - Offset: $k_r = k - 2^{k_d}$, the remaining binary digits

<table>
<thead>
<tr>
<th>Number ($k$)</th>
<th>$k_d$</th>
<th>$k_r$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>21010</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>1110111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>11110000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>111111101111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>1111111111011111111</td>
</tr>
</tbody>
</table>
Why Do Elias-\(\gamma\) Codes Work?

• The unary part of the code tells us how many bits to expect in the binary part.
• No more bits than the unary code for any number.
• For any numbers larger than 2, Elias-\(\gamma\) codes use fewer bits.
  – Example: 19 bits for 1023, instead of 1024 bits using unary code.
• For any number \(k\), the Elias-\(\gamma\) codes need \(2 \left\lfloor \log_2 k \right\rfloor + 1\) bits.
Saving by Elias-$\gamma$ Codes

- An optimal coding uses $H(P) = - \sum_{x \in X} P(x) \log_2 P(x)$ bits in expectation
- Elias-$\gamma$ codes achieve constant approximation rate to the optimal coding in any case
  \[ \frac{E(L_\gamma)}{H(P)} \leq 2 + \frac{1}{H(P)} \leq 3 \]
  - Universal code: within a factor of optimal for an arbitrary distribution
- Elias-$\gamma$ codes achieve great compression ratio if gaps follow power law distribution approximately
  - Details in the textbook
- Other nice properties of Elias-$\gamma$ codes
  - Prefix free: no $\gamma$ code is the prefix of another – there is always a unique decoding of a sequence of $\gamma$ codes, no delimiters are needed
  - Parameter free
Elias-$\delta$ Codes

- Elias-$\gamma$ codes is not good for big numbers
  - Number $k$ can be expressed in $\log_2 k$ binary digits, but Elias-$\gamma$ codes require twice as many bits to make the encoding unambiguous

- Elias-$\delta$ codes: encode $k_d + 1$ in Elias-$\gamma$ codes
  - $k_{dd} = \lfloor \log_2 (k_d + 1) \rfloor$, $k_{dr} = k_d - 2 \lfloor \log_2(k_d + 1) \rfloor$
    - Why $k_d + 1$?
  - Encode $k_{dd}$ in unary, $k_{dr}$ and $k_r$ in binary
## Example

<table>
<thead>
<tr>
<th>Number ((k))</th>
<th>(k_d)</th>
<th>(k_r)</th>
<th>(k_{dd})</th>
<th>(k_{dr})</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1010</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>11000111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>110010000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>3</td>
<td>0</td>
<td>11100001111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>3</td>
<td>2</td>
<td>1110010111111111</td>
</tr>
</tbody>
</table>
Skipping

- Query “galago AND animal”
  - Suppose “animal” and “galago” appear in 300 million and 1 million documents, respectively
  - A naïve method scanning the two posting lists spends most of the time on the 299 million documents that do not contain “galago”

- Skipping: every time we find a document $d_a < d_g$, we skip ahead $k$ documents in the “animal” list to a new document until $d_a \geq d_g$
  - We only need to search up to $k$ document backward linearly to find whether there is a document that may contain $d_g$
Cost Analysis of Skipping

- 1 million linear search of length $k$
  - $\frac{1}{2} \times 1,000,000 \times k$ steps in expectation
- Skipping forward in “animal” list
  - $300,000,000 / k$ times
- Total cost:
  - $500,000 k + 300,000,000 / k$

<table>
<thead>
<tr>
<th>$k$</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>62.5 million</td>
</tr>
<tr>
<td>10</td>
<td>35 million</td>
</tr>
<tr>
<td>20</td>
<td>25 million</td>
</tr>
<tr>
<td>25</td>
<td>24.5 million</td>
</tr>
<tr>
<td>40</td>
<td>27.5 million</td>
</tr>
<tr>
<td>50</td>
<td>31 million</td>
</tr>
<tr>
<td>100</td>
<td>53 million</td>
</tr>
</tbody>
</table>
Skipping in Compressed Data

- Why not use binary search instead of linear search?
  - Data is compressed!

- Using skip pointers
  - A skip pointer \((d, p)\) gives the document number \(d\) at position \(p\) in the original posting list
  - Example: finding document 80

\[
5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119
\]

Delta-encoding

\[
5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15
\]

Generating skipping pointers

\[
(17, 3), (34, 6), (45, 9), (52, 12), (89, 15), (101, 18)
\]
Summary

• Unambiguous encoding
• Delta decoding
• Byte-aligned codes and v-bytes
• Bit-aligned codes
  – Elias-γ codes, Elias-δ codes
• Skipping in compressed data
To-Do List

• Read Section 5.3
• Exercises 5.4-5.18
• Let Shrink be any unambiguous lossless compression scheme for 2-bit numbers. Prove either Shrink never uses less space than an uncompressed encoding or there is an input to Shrink such that the compressed version is larger than the uncompressed input
Scoring
Fields and Zones

• Is a document simply a sequence of words?
  – Many structural components, e.g., authors, title, date of publication, …

• Meta data: the data about documents

• Fields: document features where the possible values are finite
  – Examples: dates, ISBN

• Zones: document features whose content can be arbitrary free text
  – Examples: title, abstracts
Parametric Search

- A user may specify requirements on fields and zones

**Bibliographic Search**

<table>
<thead>
<tr>
<th>Search category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author</strong></td>
<td>Example: Widom, J or Garcia-Molina</td>
</tr>
<tr>
<td><strong>Title</strong></td>
<td>Also a part of the title possible</td>
</tr>
<tr>
<td><strong>Date of publication</strong></td>
<td>Example: 1997 or &lt;1997 or &gt;1997 limits the search to the documents appeared in, before and after 1997 respectively</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Language the document was written in English</td>
</tr>
<tr>
<td><strong>Project</strong></td>
<td>ANY</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>ANY</td>
</tr>
<tr>
<td><strong>Subject group</strong></td>
<td>ANY</td>
</tr>
<tr>
<td><strong>Sorted by</strong></td>
<td>Date of publication</td>
</tr>
</tbody>
</table>
Parametric Indexes

- One parametric index for each field/zone
Zone Index

- Encoding fields and zones where a term occurs
- Advantages
  - Reducing the size of dictionary
  - Efficient query answering
Weighted Zone Scoring

- Different fields/zones may have different importance in evaluating how a document matches a query
- For a query q and a document d, weighted zone scoring assigns to pair (q, d) a score in range [0, 1] by computing a linear combination of zone scores
- Suppose each document has l zones, let g₁, ..., gₙ ∈ [0, 1] such that ∑₁ⁿ gᵢ = 1
  - Each field.zone of the document contributes a Boolean value – let sᵢ be the Boolean score denoting a match or absence between q and the i-th zone
  - The weighted zone score is ∑₁ⁿ gᵢ x sᵢ
Example

- Let $g_{\text{abstract}} = 0.5$, $g_{\text{title}} = 0.3$, and $g_{\text{author}} = 0.2$
- Score(D2) = 0.3 + 0.2 = 0.5
- Score(D11) = 0.5
Learning Weights

• Using training examples that have been judged editorially

• Each training example is a tuple consisting of a query \( q \) and a document \( d \), and a relevance judgment for \( d \) on \( q \)
  – The judgment can be binary – relevant or not
  – A judgment score can also be used

• Compute the weights such that the learned scores approximate the relevance judgments as much as possible
  – An optimization problem
A Simple Case

- Only two zones: title and body

\[
\text{score}(d, q) = g \cdot s_{\text{Title}}(d, q) + (1 - g) s_{\text{Body}}(d, q)
\]

- A training example is a tuple \( \Phi_j = (d_j, q_j, r(d_j, q_j)) \)
  - \( r(d_j, q_j) \) is either relevant or irrelevant

<table>
<thead>
<tr>
<th>Example</th>
<th>DocID</th>
<th>Query</th>
<th>( s_T )</th>
<th>( s_B )</th>
<th>Judgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi_1 )</td>
<td>37</td>
<td>linux</td>
<td>1</td>
<td>1</td>
<td>Relevant</td>
</tr>
<tr>
<td>( \Phi_2 )</td>
<td>37</td>
<td>penguin</td>
<td>0</td>
<td>1</td>
<td>Non-relevant</td>
</tr>
<tr>
<td>( \Phi_3 )</td>
<td>238</td>
<td>system</td>
<td>0</td>
<td>1</td>
<td>Relevant</td>
</tr>
<tr>
<td>( \Phi_4 )</td>
<td>238</td>
<td>penguin</td>
<td>0</td>
<td>0</td>
<td>Non-relevant</td>
</tr>
<tr>
<td>( \Phi_5 )</td>
<td>1741</td>
<td>kernel</td>
<td>1</td>
<td>1</td>
<td>Relevant</td>
</tr>
<tr>
<td>( \Phi_6 )</td>
<td>2094</td>
<td>driver</td>
<td>0</td>
<td>1</td>
<td>Relevant</td>
</tr>
<tr>
<td>( \Phi_7 )</td>
<td>3191</td>
<td>driver</td>
<td>1</td>
<td>0</td>
<td>Non-relevant</td>
</tr>
</tbody>
</table>
Learning

- Using examples

\[
\text{score}(d_j, q_j) = g \cdot s_{\text{Title}}(d_j, q_j) + (1 - g) s_{\text{Body}}(d_j, q_j)
\]

- Error of scoring function

\[
\varepsilon(g, \Phi_j) = (r(d_j, q_j) - \text{score}(d_j, q_j))^2
\]

- Optimization goal: minimizing the total error

\[
\sum_j \varepsilon(g, \Phi_j)
\]
Optimal Weights

- Let \( n_{01r} \) (\( n_{01i} \)) be the numbers of training examples that \( S_{\text{Title}} = 0 \) and \( S_{\text{Body}} = 1 \) and the judgment is relevant (irrelevant). The contribution of those examples that \( S_{\text{Title}} = 0 \) and \( S_{\text{Body}} = 1 \) to the total error is

\[
[1 - (1 - g)]^2 n_{01r} + [0 - (1 - g)]^2 n_{01i}
\]

- The total error is 

\[
(n_{01r} + n_{10i})g^2 + (n_{10r} + n_{01i})(1 - g)^2 + n_{00r} + n_{11i}
\]

- By differentiating with respect to \( g \) and setting the result to 0, the optimal value of \( g \) is

\[
g_{\text{optimal}} = \frac{n_{10r} + n_{01i}}{n_{01r} + n_{10i} + n_{10r} + n_{01i}}
\]
Term Frequency and Weighting

• A document or zone that mentions a query term more often is likely more relevant to the query → a higher score
• Term frequency $\text{TF}(t, d)$: the number of occurrences of term $t$ in document $d$
• Are all terms in a document equally important?
  – Word “the” is frequent in many documents → even though “the” is frequent in $d$, “the” may not capture the topic of $d$
  – We also need to consider how popular a term is in the whole document collection
Document Frequency

- Document frequency $DF(t)$: the number of documents in the collection that contain a term $t$
- Collection frequency $CF(t)$: the total number of occurrences of a term $t$ in the collection
- Document frequency is more meaningful
  - We want the few documents that contain “insurance” to get a higher boost for a query on “insurance” than the many documents containing “try” to get from a query on “try”

<table>
<thead>
<tr>
<th>Word</th>
<th>cf</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td>try</td>
<td>10422</td>
<td>8760</td>
</tr>
<tr>
<td>insurance</td>
<td>10440</td>
<td>3997</td>
</tr>
</tbody>
</table>
Inverse Document Frequency

- $IDF(t) = \log \frac{N}{DF(t)}$, $N$: the total number of documents in the collection
- $IDF(t)$ is high if $t$ is a rare term
- $IDF(t)$ is likely low if $t$ is a frequent term

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>18,165</td>
<td>1.65</td>
</tr>
<tr>
<td>auto</td>
<td>6723</td>
<td>2.08</td>
</tr>
<tr>
<td>insurance</td>
<td>19,241</td>
<td>1.62</td>
</tr>
<tr>
<td>best</td>
<td>25,235</td>
<td>1.5</td>
</tr>
</tbody>
</table>
TF/IDF Weighting

- TF(t, d) measures the importance of a term t in document d
- IDF(t) measures the importance of a term t in the whole collection of documents
- TF/IDF weighting: putting TF and IDF together  \( \text{TFIDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t) \)
  - High if t occurs many times in a small number of documents, i.e., highly discriminative in those documents
  - Not high if t appears infrequent in a document, or is frequent in many documents, i.e., not discriminative
  - Low if t occurs in almost all documents, i.e., no discrimination at all
- A query may contain multiple query words
  - \( \text{Score}(q, d) = \sum_{t \in q} \text{TFIDF}(t, d) \)
Document Vectors

- Each term can be treated as a dimension
- A document is a vector $\vec{V}(d)$ with value TF($t_i$) on term $t_i$
- Can the similarity between two documents be measured as the magnitude of the vector difference?
  - If one document is long and the other is short, their absolute term frequencies may differ substantially
  - However, the relative distributions of terms may be similar
Cosine Similarity

- Measure the similarity using the cosine of the vector representations
  \[
  \text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)| \times |\vec{V}(d_2)|}
  \]

- Dot product
  \[
  \vec{V}(d_1) \cdot \vec{V}(d_2) = \sum_t TF(t, d_1) \times TF(t, d_2)
  \]

- Euclidean length
  \[
  |\vec{V}(d_1)| = \sqrt{\sum_t TF(t, d_1)^2}
  \]

- Length-normalization
  \[
  \vec{v}(d_1) = \frac{\vec{V}(d_1)}{|\vec{V}(d_1)|}
  \]

  \[
  \text{sim}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2)
  \]
Finding Similar Documents

\[ \vec{v}(d_1) \]
\[ \vec{v}(q) \]
\[ \vec{v}(d_2) \]
\[ \vec{v}(d_3) \]
## Term-Document Matrix

- Each row represents a term (dimension), in total $M$ terms
- Each column represents a document, in total $N$ documents

<table>
<thead>
<tr>
<th>Terms</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aquarium</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bowl</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>care</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fish</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>freshwater</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>goldfish</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>homepage</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>keep</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>setup</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tank</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>tropical</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

1. Tropical Freshwater Aquarium Fish.
2. Tropical Fish, Aquarium Care, Tank Setup.
3. Keeping Tropical Fish and Goldfish in Aquariums, and Fish Bowls.
4. The Tropical Tank Homepage - Tropical Fish and Aquariums.
Query Vector

• A query can be viewed as a very short document
  – Query vector: the vector of the query document

• Query answering: find the documents which are most similar to the query vector

\[
\text{score}(q,d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| \times |\vec{V}(d)|}
\]

  – Can be very expensive in practice – many words in a document and many documents
Sublinear TF Scaling

- If a term appears 20 times, the significance may not be 20 times of a single occurrence
  - The marginal increase of significance decreases

- Using the logarithm of the term frequency

\[
WF(t, d) = \begin{cases} 
1 + \log TF(t, d) & \text{if } TF(t, d) > 0 \\
0 & \text{otherwise}
\end{cases}
\]

\[
WFIDF(t, d) = WF(t, d) \times IDF(t)
\]
Maximum TF Normalization

• A document repeating a term may gain TF
  – Anomaly: if a document copies itself, the TF score doubles but the relevance to any query should not!
• Solution: normalize the TF weights of all terms occurring in a document by the maximum TF in that document
• For document d, let \( TF_{\text{max}}(d) = \max_{\tau \in d} TF(\tau, d) \)
• Normalized term frequency \( NTF(t, d) = a + (1-a) \frac{TF(t, d)}{TF_{\text{max}}(d)} \)
  – Smoothing term \( a = 0.4 \)
Maximum TF Normalization: Cons

- Hard to tune: a change in the stop word list can dramatically alter term weightings and thus rankings
- A document may contain an outlier term with an unusually large TF but not representative of the content of the document
- Distribution of TF in a document should be considered: a document where the most frequent term appears roughly as often as many other terms should be treated differently from one with a more skewed distribution
Document/Query Weighting Schemes

- Notion: ddd.qqq – ddd for documents, qqq for queries
  - Example: Inc.ltc

<table>
<thead>
<tr>
<th>term frequency</th>
<th>document frequency</th>
<th>normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>n (natural)</td>
<td>tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;</td>
<td>n (no)</td>
</tr>
<tr>
<td>l (logarithm)</td>
<td>1 + log(tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;)</td>
<td>t (idf)</td>
</tr>
<tr>
<td>a (augmented)</td>
<td>0.5 + 0.5×tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;/max&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;(tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;)</td>
<td>p (prob idf)</td>
</tr>
<tr>
<td>b (boolean)</td>
<td>{1 if tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt; &gt; 0, 0 otherwise}</td>
<td>u (pivoted unique)</td>
</tr>
<tr>
<td>L (log ave)</td>
<td>1 + log(tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;) / (1 + log(ave&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;(tf&lt;sub&gt;&lt;i&gt;d&lt;/i&gt;&lt;/sub&gt;)))</td>
<td>b (byte size)</td>
</tr>
</tbody>
</table>

n (none) 1  
\[
\frac{N}{df_i}
\]

\[
\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}
\]

\[
\frac{1}{u}
\]

\[
\frac{1}{CharLength^\alpha}, \alpha < 1
\]
Summary

• Parametric search and parametric indexes
• TF/IDF weighting
• Document vectors
• Document and query weight schemes
Query Answering Using Inverted Indexes
Inverted Indexes

Query “Brutus” AND “Calpurnia”

<table>
<thead>
<tr>
<th>Term</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>Brutus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caesar</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>2</td>
<td>31</td>
<td>54</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dictionary

Postings
Document-at-a-time Evaluation

- The conceptually simplest query answering method

<table>
<thead>
<tr>
<th>Query</th>
<th>salt</th>
<th>water</th>
<th>tropical</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1:1</td>
<td>1:1</td>
<td>1:2</td>
<td>1:4</td>
</tr>
<tr>
<td></td>
<td>2:1</td>
<td>2:2</td>
<td>3:1</td>
<td>4:2</td>
</tr>
<tr>
<td></td>
<td>3:1</td>
<td>4:1</td>
<td>4:1</td>
<td>2:3</td>
</tr>
<tr>
<td></td>
<td>4:1</td>
<td>2:1</td>
<td>3:1</td>
<td>2:2</td>
</tr>
<tr>
<td></td>
<td>4:1</td>
<td>3:1</td>
<td>4:2</td>
<td>1:2</td>
</tr>
</tbody>
</table>

- Query: salt, water, tropical, score
- Document: 1, 2, 3, 4
Algorithm

```plaintext
procedure DOCUMENTATATIMEREtrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms $w_i$ in $Q$ do
        $l_i$ ← InvertedList($w_i$, I)
        L.add($l_i$)
    end for
    for all documents $d \in I$ do
        for all inverted lists $l_i$ in $L$ do
            if $l_i$ points to $d$ then
                $s_D ← s_D + g_i(Q)f_i(l_i)$ ▷ Update the document score
                $l_i$.movePastDocument($d$)
            end if
        end for
        R.add($s_D$, $D$)
    end for
    return the top $k$ results from $R$
end procedure
```

Find posting lists

Can be implemented efficiently by keeping the top-k list at anytime
Term-at-a-time Evaluation

<table>
<thead>
<tr>
<th>Term</th>
<th>Partial Scores</th>
<th>New Partial Scores</th>
<th>Old Partial Scores</th>
<th>Tropical</th>
<th>Final Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>salt</td>
<td>1:1</td>
<td>4:1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partial scores</td>
<td>1:1</td>
<td>4:1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>old partial scores</td>
<td>1:1</td>
<td></td>
<td>4:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>water</td>
<td>1:1</td>
<td>2:1</td>
<td>4:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>new partial scores</td>
<td>1:2</td>
<td>2:1</td>
<td>4:2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>old partial scores</td>
<td>1:2</td>
<td>2:1</td>
<td>4:2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tropical</td>
<td>1:2</td>
<td>2:2</td>
<td>3:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>final scores</td>
<td>1:4</td>
<td>2:3</td>
<td>2:2</td>
<td>4:2</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm

procedure TERMATATIMERETRIEVAL(Q, I, f, g, k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms \( w_i \) in Q do
        \( l_i \) ← InvertedList(w_i, I)
        L.add( \( l_i \) )
    end for
    for all lists \( l_i \in L \) do
        while \( l_i \) is not finished do
            \( d \) ← \( l_i \).getCurrentDocument()
            \( A_d \) ← \( A_d \) + \( g_i(Q) \cdot f(l_i) \)
            \( l_i \).moveToNextDocument()
        end while
    end for
    for all accumulators \( A_d \) in A do
        \( s_D \) ← \( A_d \)  \( \triangleright \) Accumulator contains the document score
        R.add( \( s_D, D \) )
    end for
    return the top \( k \) results from \( R \)  \( \triangleright \) Can be implemented efficiently by keeping the top-k list at anytime
end procedure
Comparison

• Memory usage
  – The document-at-a-time method only needs to maintain a priority queue $R$ of a limited number of results
  – The term-at-a-time method needs to store the current scores for all documents

• Disk access
  – The document-at-a-time method needs more disk seeking and buffers for seeking since multiple lists are read in a synchronized way (Discussion: typically are there many lists being read in a synchronized way?)
  – The term-at-a-time method reads through each inverted list from start to end – requiring minimal disk seeking and buffer
Computing Cosine Score

$\text{CosineScore}(q)$

1. float $Scores[N] = 0$

2. Initialize $Length[N]$

3. for each query term $t$

4. do calculate $w_{t,q}$ and fetch postings list for $t$

5. for each pair $(d, tf_{t,d})$ in postings list

6. do $Scores[d] += wf_{t,d} \times w_{t,q}$

7. Read the array $Length[d]$

8. for each $d$

9. do $Scores[d] = Scores[d]/Length[d]$

10. return Top $K$ components of $Scores[]$
Efficient Scoring

• For a query \( q = w_1 \, w_2 \)
  – The unit vector \( \mathbf{v}(q) \) has only two nonzero components
  – If query terms are not weighted, the nonzero components are equal to \( \sqrt{2}/2 = 0.707 \)

• Generally, for any two documents \( d_1 \) and \( d_2 \)
  \[ \mathbf{V}(q) \cdot \mathbf{v}(d_1) > \mathbf{V}(q) \cdot \mathbf{v}(d_2) \]  if and only if \( \mathbf{v}(q) \cdot \mathbf{v}(d_1) > \mathbf{v}(q) \cdot \mathbf{v}(d_2) \)

• \( \mathbf{V}(q) \cdot \mathbf{v}(d) \) is the weighted sum over all terms in query \( q \), of the weights of those terms in \( d \)
Efficient Scoring Algorithm

\begin{algorithm}
\textbf{FastCosineScore}(q)
\begin{algorithmic}[1]
\State float $Scores[N] = 0$
\For {each $d$}
\Do Initialize $Length[d]$ to the length of doc $d$
\For {each query term $t$}
\Do calculate $w_{t,q}$ and fetch postings list for $t$
\For {each pair $(d, tf_{t,d})$ in postings list}
\Do add $wf_{t,d}$ to $Scores[d]$
\EndDo
\EndFor
\EndDo
\EndFor
\Read the array $Length[d]$
\For {each $d$}
\Do Divide $Scores[d]$ by $Length[d]$
\EndDo
\EndFor
\State \textbf{return} Top $K$ components of $Scores[]$
\end{algorithmic}
\end{algorithm}

Using a heap, selecting top $k$ answers can be done with $2J$ comparisons where $J$ is the number of answers of nonzero scores.
Approximate Top-K Retrieval

• Retrieve K documents that are likely to be among the K highest scoring documents
  – Goal: lower down the query answering cost
  – Cosine measure is also an approximation of information need

• Major cost: computing cosine similarities between the query and a large number of documents

• Approximation strategies
  – Find a set A of documents that are contenders, where K < |A| « N, such that A is likely to have many documents with scores near those of the top K
  – Return the top-K documents in A
Index Elimination

• For a multi-term query q, we only need to consider documents containing at least one of the query terms

• Only consider documents containing terms whose IDF exceeds a preset threshold
  – Only check those discriminative words
  – Benefit: the postings lists of low-IDF terms are generally long (many are stop words)

• Only consider documents that contain many of the query terms
Champion Lists

• For each term $t$ in the dictionary, pre-compute the top-$r$ documents of the highest weights for $t$, where $r$ is a preset parameter
  – Set different $r$ for different terms – larger for rare terms and smaller for frequent terms

• Given a query $q$, let $A$ be the union of the champion lists for each of the terms comprising $q$
  – Compute cosine similarity only between $q$ and those documents in $A$
Static Quality Scores and Ordering

- Different documents have different importance
  - Example: how good are reviews on a web page?
  - Modeled by a quality measure \( g(d) \in [0, 1] \)

\[
TotalScore(q, d) = g(d) + \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| \times |\vec{V}(d)|}
\]

- Sort documents in posting lists in \( g(d) \) descending order

Suppose \( g(1) = 0.25 \), \( g(2) = 0.5 \), and \( g(3) = 1 \)
Using Quality Score Ordering

• For a well-chosen value r, maintain for each term t a global champion list of the top-r documents with the highest value of $g(d) + \text{TFIDF}(t, d)$
  – At query time, only compute TotalScore for documents in the union of those global champion lists

• Maintain for each term t two posting lists
  – High list: m documents with the highest TF values for t
  – Low list: the other documents containing t
  – Use high list only if at least K answers can be generated
Tiered Indexes

- Generalization of champion lists
Clustering and NN Search

- Clustering
  - Pick $\sqrt{N}$ documents as leaders at random from the collection
  - For each document that is not a leader (called a follower), compute its nearest leader
    - Each cluster has $\frac{N}{\sqrt{N}} = \sqrt{N}$ followers
  - Alternatively, a follower can be assigned to $b_1$ leaders

- Query answering as nearest neighbor search
  - For a query $q$, find the leader $L$ (or $b_2$ leaders) that is closest to $q$ – computing cosine similarities between $q$ and $\sqrt{N}$ leaders
  - The candidate set $A$ contains the closest leader and the followers
Example

- Leader
- Follower

Query
Putting All Together

Diagram showing the process of information retrieval and web search, including steps such as parsing linguistics, indexers, and scoring and ranking.
Summary and To-Do-List

• Query evaluation
  – Document-at-a-time versus term-at-a-time
• List skipping
• Efficient scoring
• Approximate top-K retrieval
  – Index elimination, champion lists, quality score and ranking, clustering and nearest neighbor search
• Chapter 7 and the exercises
Efficient Implementation of Postings Lists
Inverted Indices

Query “Brutus” AND “Calpurnia”

<table>
<thead>
<tr>
<th>Term</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>Brutus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
Skip Pointers

Brutus → 2 → 4 → 8 → 16 → 19 → 23 → 28 → 43

Caesar → 1 → 2 → 3 → 5 → 8 → 41 → 51 → 60 → 71
Using Skip Pointers

\[\text{IntersectWithSkips}(p_1, p_2)\]

1. \textit{answer} ← \{\}
2. \textbf{while} \(p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL}\)
3. \textbf{do if} \(\text{docID}(p_1) = \text{docID}(p_2)\)
   \hspace{1cm} \textbf{then} \text{ADD}(\textit{answer}, \text{docID}(p_1))
   \hspace{1cm} p_1 ← \text{next}(p_1)
   \hspace{1cm} p_2 ← \text{next}(p_2)
4. \textbf{else if} \(\text{docID}(p_1) < \text{docID}(p_2)\)
   \hspace{1cm} \textbf{then if} \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))
   \hspace{1cm} \textbf{then while} \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))
   \hspace{1cm} \textbf{do} p_1 ← \text{skip}(p_1)
   \hspace{1cm} \textbf{else} p_1 ← \text{next}(p_1)
5. \textbf{else if} \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))
   \hspace{1cm} \textbf{then while} \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))
   \hspace{1cm} \textbf{do} p_2 ← \text{skip}(p_2)
   \hspace{1cm} \textbf{else} p_2 ← \text{next}(p_2)
6. \textbf{return} \textit{answer}
Efficiency of Using Skip Pointers

• What is the best case?
  – W1: 2(40) → 5 → 8 → 40(76) → 42 → 65 → 76
  – W2: 40(120) → 85 → 100 → 120

• What is the worst case?
  – W1: 1(7) → 3 → 5 → 7(13) → 9 → 11 → 13
  – W2: 2(8) → 4 → 6 → 8(14) → 10 → 12 → 14

• Can you propose an idea to make use of skip pointers more aggressively?
  – Then, what is the cost associated with the aggressive usage?
Where Do We Place Skip Pointers?

- More skip pointers → shorter skip spans → more likely to skip → many comparisons to skip pointers → much space storing skip pointers
- Fewer skip points → fewer pointer comparison → long skip spans → fewer opportunities to skip
- A heuristic: for a postings list of length $P$, use $\sqrt{P}$ evenly spaced skip pointers
  - Can you propose an idea to be aware of the distribution of the postings?
Phrase Queries

- Query “Stanford University”
  - Using singleton terms, “The inventor Stanford Ovshinsky never went to university” is an answer
- 10% of web queries are phrase queries, many more are implicit phrase queries (e.g., person names)
Biword Indices

- Consider every pair of consecutive terms in a document as a phrase
  - “Friends, Romans, Countrymen” → “friends romans”, “romans countrymen”
  - “stanford university palo alto” → “stanford university” AND “university palo” AND “palo alto”

- Biwords using nouns
  - “renegotiation of the constitution” matches rule NX*N
  - May not always work well
    - “cost overruns on a power plant” → “cost overruns” AND “overruns power” AND “power plan”
    - “cost overruns” AND “power plan” seem more appropriate
Phrase Indices

- Biword indices can be extended to variable length word sequences
- The longer the phrase, the lower the chance of false-positive
- Including long phrases increases the size of vocabulary
  - A tradeoff between space and query effectiveness as well as efficiency
Positional Indices

• A positional index stores, for each term in the vocabulary, postings of the form docID: <position1, position2, ...>

• Search for “to be and not to be”
  – How many times do we need to search for “to be”?

  to, 993427:
  \[1, 6: \langle 7, 18, 33, 72, 86, 231 \rangle; \]
  \[2, 5: \langle 1, 17, 74, 222, 255 \rangle; \]
  \[4, 5: \langle 8, 16, 190, 429, 433 \rangle; \]
  \[5, 2: \langle 363, 367 \rangle; \]
  \[7, 3: \langle 13, 23, 191 \rangle; \ldots \]

  be, 178239:
  \[1, 2: \langle 17, 25 \rangle; \]
  \[4, 5: \langle 17, 191, 291, 430, 434 \rangle; \]
  \[5, 3: \langle 14, 19, 101 \rangle; \ldots \]
Proximity Intersection

**Positional Intersect**$(p_1, p_2, k)$

1. $answer \leftarrow \langle \rangle$
2. **while** $p_1 \neq \text{NIL}$ and $p_2 \neq \text{NIL}$
3. **do if** $docID(p_1) = docID(p_2)$
4. \hspace{1em} **then** $l \leftarrow \langle \rangle$
5. \hspace{2em} $pp_1 \leftarrow \text{positions}(p_1)$
6. \hspace{2em} $pp_2 \leftarrow \text{positions}(p_2)$
7. **while** $pp_1 \neq \text{NIL}$
8. \hspace{2em} **do while** $pp_2 \neq \text{NIL}$
9. \hspace{3em} **do if** $|\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k$
10. \hspace{4em} **then** $\text{ADD}(l, \text{pos}(pp_2))$
11. \hspace{4em} **else if** $\text{pos}(pp_2) > \text{pos}(pp_1)$
12. \hspace{5em} **then** break
13. \hspace{2em} **else if** $\text{pos}(pp_1) > \text{pos}(pp_2)$
14. \hspace{4em} $pp_2 \leftarrow \text{next}(pp_2)$
15. **while** $l \neq \langle \rangle$ and $|l[0] - \text{pos}(pp_1)| > k$
16. \hspace{2em} **do** $\text{DELETE}(l[0])$
17. \hspace{2em} **for each** $ps \in l$
18. \hspace{3em} **do** $\text{ADD}(answer, \langle docID(p_1), \text{pos}(pp_1), ps \rangle)$
19. \hspace{3em} \hspace{2em} $pp_1 \leftarrow \text{next}(pp_1)$
20. \hspace{3em} \hspace{2em} $p_1 \leftarrow \text{next}(p_1)$
21. \hspace{3em} \hspace{2em} $p_2 \leftarrow \text{next}(p_2)$
22. \hspace{3em} **else if** $docID(p_1) < docID(p_2)$
23. \hspace{4em} **then** $p_1 \leftarrow \text{next}(p_1)$
24. \hspace{4em} **else** $p_2 \leftarrow \text{next}(p_2)$

**return** $answer$
Positional Index Size

- Record positions of all occurrences of a term
  - Boolean query complexity $O(T)$, where $T$ is the number of tokens in the document collection
- Suppose a term has frequency 0.1% on average
  - The term is expected to appear once in a document of 1,000 terms, 100 times in a document of 100,000 terms
  - Document size affects the positional index size
- In practice, a positional index is about 2-4 times as large as a nonpositional index
Combination Schemes

- Biword indices and positional indices can be combined
  - For commonly queried phrases, use biword indices
  - For others, use positional indices
- Consider relative frequencies
  - Use biword indices for phrases where the individual words are common but the desired phrase is comparatively rare
- Partial next word index: for each term, record terms that follow it in a document
  - Save 75% query time, with 26% more space
Example Text Collection

$S_1$ Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

$S_2$ Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

$S_3$ Tropical fish are popular aquarium fish, due to their often bright coloration.

$S_4$ In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.
Inverted Index with Word Counts

- Help to rank the most relevant documents
Posting Lists of Scores

- We can store the relevance/similarity between a document and a query word
- A posting list of scores
  - Fish: (1: 3.6), (3: 2.2), ...
- Increasing flexibility: computationally expensive scoring becomes possible
  - Scoring is moved to the index construction process
- Losing flexibility: the scoring mechanism is fixed once the index is built
  - Information about word proximity is lost
Ordering Using Scores

• An inverted list can be ordered by score so that the highest scored documents come first

• The query processing engine can focus only on the top part of each inverted list
  – Only need to read a small number of postings to find the top-k documents
Threshold Adjustment Algorithm

- **Query**: A and B, find top-2 documents
  - Scoring function: score(A) + score(B)
- **Each posting list is sorted in score descending order**
  - A: (D1, 3.8), (D10, 3.7), (D5, 3.6), (D8, 2.9), ...
  - B: (D10, 3.4), (D2, 2.5), (D5, 2.2), (D9, 1.9), (D1, 1.8)
- **Step 1**: read (D1, 3.8) from A and (D10, 3.4) from B
  - Upper bounds: any document cannot be higher than 7.2
  - Lower bounds: $D1 \geq 3.8$, $D10 \geq 3.4$
- **Step 2**: read (D10, 3.7) from A and (D2, 2.5) from B
  - D10 = 7.1
  - Upper bounds: $D1 \leq 6.3$, all others $\leq 6.2$
  - Lower bounds: $D2 \geq 2.5$
  - D10 is the highest
Threshold Adjustment Algorithm

- Step 3: read (D5, 3.6) from A and (D5, 2.2) from B
  - D5 = 5.8
  - Upper bounds: D1 \leq 6, D2 \leq 6.1, all others \leq 5.8
  - Lower bounds:
    - D5 is a candidate

- Step 4: read (D8, 2.9) from A and (D9, 1.9) from B
  - Upper bounds: D1 \leq 5.7, D2 \leq 5.4
  - D5 is another document in top-2
  - The search terminates
Summary

- Inverted indices can be improved by efficient implementation of postings lists
- Using skip pointers
- Phrase queries
  - Biword indices
  - Positional postings
  - The two can be combined
- Inverted indexes can be extended in many ways
- The TA algorithm
To-do List

• Read Sections 2.3 and 2.4 in the textbook
• Exercises 2.5-2.14
• Give the pseudocode of the TA algorithm
  – How can the lower bounds be used? This is not discussed in the class
  – Can you generalize it for multiple lists?