Web Data Compression and Search

Search, index construction and compression

Slides modified from Hinrich Schütze and Christina Lioma slides
Inverted Index

For each term \( t \), we store a list of all documents that contain \( t \).

- **Brutus** → 1 2 4 11 31 45 173 174
- **Caesar** → 1 2 4 5 6 16 57 132 ...
- **Calpurnia** → 2 31 54 101

- dictionary
- postings
Inverted index construction

1. Collect the documents to be indexed:
   
   Friends, Romans, countrymen. So let it be with Caesar . . .

2. Tokenize the text, turning each document into a list of tokens:
   
   Friends Romans countrymen So . . .

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms:
   
   friend roman countryman so . . .

4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.
Tokenizing and preprocessing

**Doc 1.** I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

**Doc 2.** So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

**Doc 1.** i did enact julius caesar i was killed i’ the capitol brutus killed me

**Doc 2.** so let it be with caesar the noble brutus hath told you caesar was ambitious
Generate posting

<table>
<thead>
<tr>
<th>term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
Sort postings

<table>
<thead>
<tr>
<th>term</th>
<th>docID</th>
<th></th>
<th>term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td></td>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td></td>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td></td>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td></td>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
<td></td>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td></td>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
<td></td>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td></td>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td></td>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td></td>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td></td>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
<td></td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td></td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td></td>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
<td></td>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
<td></td>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
<td></td>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td></td>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
<td></td>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td></td>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td></td>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
<td></td>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td></td>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
<td></td>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
<td></td>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
<td></td>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td></td>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td></td>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
<td></td>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>
Create postings lists, determine document frequency

<table>
<thead>
<tr>
<th>term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>doc. freq.</th>
<th>postings lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>→ 1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>→ 2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>→ 1 → 2</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>→ 2</td>
</tr>
</tbody>
</table>
Split the result into dictionary and postings file

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

: dictionary
: postings
Simple conjunctive query (two terms)

- Consider the query: BRUTUS AND CALPURNIA
- To find all matching documents using inverted index:
  1. Locate BRUTUS in the dictionary
  2. Retrieve its postings list from the postings file
  3. Locate CALPURNIA in the dictionary
  4. Retrieve its postings list from the postings file
  5. Intersect the two postings lists
  6. Return intersection to user
Intersecting two posting lists

- This is linear in the length of the postings lists.
- Note: This only works if postings lists are sorted.
Intersecting two posting lists

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

1. \( \text{answer} \leftarrow \langle \rangle \)
2. \( \textbf{while} \ p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL} \)
3. \( \textbf{do if} \ \text{docID}(p_1) = \text{docID}(p_2) \)
4. \( \quad \textbf{then} \ \text{ADD}(\text{answer, docID}(p_1)) \)
5. \( \quad p_1 \leftarrow \text{next}(p_1) \)
6. \( \quad p_2 \leftarrow \text{next}(p_2) \)
7. \( \textbf{else if} \ \text{docID}(p_1) < \text{docID}(p_2) \)
8. \( \quad \textbf{then} \ p_1 \leftarrow \text{next}(p_1) \)
9. \( \quad \textbf{else} \ p_2 \leftarrow \text{next}(p_2) \)
10. \( \textbf{return} \ \text{answer} \)
Typical query optimization

- Example query: BRUTUS AND CALPURNIA AND CAESAR
- Simple and effective optimization: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further
- In this example, first CAESAR, then CALPURNIA, then BRUTUS

```
Brutus: 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174
Calpurnia: 2 → 31 → 54 → 101
Caesar: 5 → 31
```
Optimized intersection algorithm for conjunctive queries

\[
\text{INTERSECT}\left(\langle t_1, \ldots, t_n \rangle \right)
\]

1. \( \text{terms} \leftarrow \text{SORTBYINCREASINGFREQUENCY}\left(\langle t_1, \ldots, t_n \rangle \right) \)
2. \( \text{result} \leftarrow \text{postings}(\text{first(terms)}) \)
3. \( \text{terms} \leftarrow \text{rest(terms)} \)
4. \( \textbf{while} \ \text{terms} \neq \text{NIL} \ \text{and} \ \text{result} \neq \text{NIL} \)
5. \( \textbf{do} \ \text{result} \leftarrow \text{INTERSECT}(\text{result}, \text{postings}(\text{first(terms)})) \)
6. \( \text{terms} \leftarrow \text{rest(terms)} \)
7. \( \text{return result} \)
Recall basic intersection algorithm

- Linear in the length of the postings lists.
- Can we do better?
Skip pointers

- Skip pointers allow us to skip postings that will not figure in the search results.
- This makes intersecting postings lists more efficient.
- Some postings lists contain several million entries – so efficiency can be an issue even if basic intersection is linear.
- Where do we put skip pointers?
- How do we make sure intersection results are correct?
Basic idea

Brutus: 2 → 4 → 8 → 34 → 35 → 64 → 128

Caesar: 1 → 2 → 3 → 5 → 8 → 17 → 21 → 31 → 75 → 81 → 84 → 89 → 92
Skip lists: Larger example

![Skip list diagram](image)

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

- **Caesar**
  - 1 → 2 → 3 → 5 → 8 → 41 → 51 → 60 → 71

Numbers 16, 28, and 72 are highlighted, indicating potential skip list nodes.
Intersection with skip pointers

\textbf{INTERSECTWITHSKIPS}(p_1, p_2)

1 \quad \text{answer} \leftarrow \langle \rangle
2 \quad \textbf{while } p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL}
3 \quad \textbf{do if } \text{docID}(p_1) = \text{docID}(p_2)
4 \quad \quad \textbf{then } \text{ADD}(\text{answer}, \text{docID}(p_1))
5 \quad \quad p_1 \leftarrow \text{next}(p_1)
6 \quad \quad p_2 \leftarrow \text{next}(p_2)
7 \quad \textbf{else if } \text{docID}(p_1) < \text{docID}(p_2)
8 \quad \quad \textbf{then if } \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))
9 \quad \quad \quad \textbf{then while } \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))
10 \quad \quad \quad \quad \textbf{do } p_1 \leftarrow \text{skip}(p_1)
11 \quad \quad \quad \textbf{else } p_1 \leftarrow \text{next}(p_1)
12 \quad \quad \textbf{else if } \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))
13 \quad \quad \quad \textbf{then while } \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))
14 \quad \quad \quad \quad \textbf{do } p_2 \leftarrow \text{skip}(p_2)
15 \quad \quad \textbf{else } p_2 \leftarrow \text{next}(p_2)
16 \quad \textbf{return } \text{answer}
Where do we place skips?

- Tradeoff: number of items skipped vs. frequency skip can be taken
- More skips: Each skip pointer skips only a few items, but we can frequently use it.
- Fewer skips: Each skip pointer skips many items, but we cannot use it very often.
Phrase queries

- We want to answer a query such as [stanford university] – as a phrase.
- Thus *The inventor Stanford Ovshinsky never went to university* should not be a match.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists.
- Two ways of extending the inverted index:
  - biword index
  - positional index
Positional indexes

- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions
Positional indexes: Example

Query: “to$_1$ be$_2$ or$_3$ not$_4$ to$_5$ be$_6$”

TO, 993427:

1: <7, 18, 33, 72, 86, 231>
2: <1, 17, 74, 222, 255>
4: <8, 16, 190, 429, 433>
5: <363, 367>
7: <13, 23, 191>; . . .

BE, 178239:

1: <17, 25>
4: <17, 191, 291, 430, 434>
5: <14, 19, 101>; . . . Document 4 is a match!
Inverted index

For each term $t$, we store a list of all documents that contain $t$.

- $\text{BRUTUS} \rightarrow \{1, 2, 4, 11, 31, 45, 173, 174\}$
- $\text{CAESAR} \rightarrow \{1, 2, 4, 5, 6, 16, 57, 132, \ldots\}$
- $\text{CALPURNIA} \rightarrow \{2, 31, 54, 101\}$

\[
\begin{array}{c}
\text{dictionary} \\
\text{postings}
\end{array}
\]
Dictionaries

- The dictionary is the data structure for storing the term vocabulary.
- **Term vocabulary**: the data
- **Dictionary**: the **data structure** for storing the term vocabulary
Dictionary as array of fixed-width entries

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...  
- Assume for the time being that we can store this information in a fixed-length entry.
- Assume that we store these entries in an array.
Dictionary as array of fixed-width entries

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td>→</td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td>→</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td>→</td>
</tr>
</tbody>
</table>

space needed: 20 bytes 4 bytes 4 bytes

How do we look up a query term $q_i$ in this array at query time? That is: which data structure do we use to locate the entry (row) in the array where $q_i$ is stored?
Data structures for looking up terms

- Two main classes of data structures: hashes and trees
- Some IR systems use hashes, some use trees.
- Criteria for when to use hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?
Hashes

- Each vocabulary term is hashed into an integer.
- Try to avoid collisions
- At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array
- Pros: Lookup in a hash is faster than lookup in a tree.
  - Lookup time is constant.
- Cons
  - no way to find minor variants (resume vs. résumé)
  - no prefix search (all terms starting with automat)
  - need to rehash everything periodically if vocabulary keeps growing
Trees

- Trees solve the prefix problem (find all terms starting with *automat*).
- Simplest tree: binary tree
- Search is slightly slower than in hashes: $O(\log M)$, where $M$ is the size of the vocabulary.
- $O(\log M)$ only holds for balanced trees.
- Rebalancing binary trees is expensive.
- **B-trees** mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval $[a, b]$ where $a, b$ are appropriate positive integers, e.g., $[2, 4]$. 
Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- But in-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.
Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?

- No: Sorting for example 100,000,000 records on disk is too slow – too many disk seeks.

- We need an **external** sorting algorithm.
“External” sorting algorithm (using few disk seeks)

- We must sort 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, document frequency).
- Define a block to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks.
- Basic idea of algorithm:
  - For each block: (i) accumulate postings, (ii) sort in memory, (iii) write to disk
  - Then merge the blocks into one long sorted order.
Merging two blocks

**postings to be merged**

**Block 1**
- brutus d3
- caesar d4
- noble d3
- with d4

**Block 2**
- brutus d2
- caesar d1
- julius d1
- killed d2

**merged postings**
- brutus d2
- brutus d3
- caesar d1
- caesar d4
- julius d1
- killed d2
- noble d3
- with d4

**disk**
Blocked Sort-Based Indexing

BSBIndexConstruction()
1   $n \leftarrow 0$
2   while (all documents have not been processed)
3     do $n \leftarrow n + 1$
4     \hspace{1em} block $\leftarrow$ ParseNextBlock()
5     BSBI-Invert(block)
6     WriteBlockToDisk(block, $f_n$)
7     MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)
Problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)
Single-pass in-memory indexing

- Abbreviation: SPIMI
- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.
SPIMI-Invert

\[
\text{SPIMI-Invert}(\text{token\_stream})
\]

1. \text{output\_file} \leftarrow \text{NEWFILE()}
2. \text{dictionary} \leftarrow \text{NEWHASH()}
3. \textbf{while} (free memory available)
4. \textbf{do} \quad \text{token} \leftarrow \text{next}(\text{token\_stream})
5. \quad \textbf{if} \quad \text{term(token)} \notin \text{dictionary}
6. \quad \quad \text{then} \quad \text{postings\_list} \leftarrow \text{addToDictionary(\text{dictionary}, \text{term(token)})}
7. \quad \quad \text{else} \quad \text{postings\_list} \leftarrow \text{getPostingsList(\text{dictionary}, \text{term(token)})}
8. \quad \textbf{if} \quad \text{full(postings\_list)}
9. \quad \quad \text{then} \quad \text{postings\_list} \leftarrow \text{doublePostingsList(\text{dictionary}, \text{term(token)})}
10. \quad \quad \text{addToPostingsList(postings\_list, docId(token))}
11. \quad \text{sorted\_terms} \leftarrow \text{sortTerms(\text{dictionary})}
12. \quad \text{WriteBlockToFile(sorted\_terms, dictionary, output\_file)}
13. \textbf{return} \quad \text{output\_file}

Merging of blocks is analogous to BSBI.
Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We will devise various compression schemes for dictionary and postings.
Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.
Recall: Dictionary as array of fixed-width entries

Space needed: 20 bytes  4 bytes  4 bytes
for Reuters: (20+4+4)*400,000 = 11.2 MB
Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can’t handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?
Dictionary as a string

...systileszygeticszygialsyzygyszaibelyiteszecinszono...

<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
Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400,000$ positions)
- Space: $400,000 \times (4 +4 +3 + 8) = 7.6$MB (compared to 11.2 MB for fixed-width array)
Dictionary as a string with blocking

...7syltle9syzygetic8syzygial6syzygy11szaibelyite6szecin...

freq. postings ptr. term ptr.
9 →
92 →
5 →
71 →
12 →
... →
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used $4 \times 3$ bytes for term pointers without blocking . . .
- . . .we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400,000/4 \times 5 = 0.5$ MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
Lookup of a term without blocking
Lookup of a term with blocking: (slightly) slower
Front coding

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

\[\text{8 automatica 8 automatica 9 automatica 10 automatication} \]

\[\Downarrow\]

. . . further compressed with front coding.

\[\text{8 automatic*3 automatication}\]
## Dictionary compression for Reuters: Summary

<table>
<thead>
<tr>
<th>data structure</th>
<th>size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>~, with blocking, k = 4</td>
<td>7.1</td>
</tr>
<tr>
<td>~, with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
</tbody>
</table>
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use \( \log_2 800,000 \approx 19.6 < 20 \) bits per docID.
- Our goal: use a lot less than 20 bits per docID.
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, . . .
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps: COMPUTER: 283154, 5, 43, . . .
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.
Gap encoding

<table>
<thead>
<tr>
<th>encoding</th>
<th>postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td>doclIDs</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>doclIDs</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td>doclIDs</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
</tr>
</tbody>
</table>
Variable length encoding

- **Aim:**
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).

  - In order to implement this, we need to devise some form of variable length encoding.
  - Variable length encoding uses few bits for small gaps and many bits for large gaps.
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit $c$.
- If the gap $G$ fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more additional bytes to encode the higher order bits using the same algorithm.
- At the end set the continuation bit of the last byte to 1 ($c = 1$) and of the other bytes to 0 ($c = 0$).
# VB code examples

<table>
<thead>
<tr>
<th>docIDs</th>
<th>824</th>
<th>829</th>
<th>215406</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaps</td>
<td>5</td>
<td>5</td>
<td>214577</td>
</tr>
<tr>
<td>VB code</td>
<td>00000110 10111000</td>
<td>10000101</td>
<td>00001101 00001100 10110001</td>
</tr>
</tbody>
</table>
VB code encoding algorithm

```vb
VBEncodeNumber(n)
1  bytes ← ∅
2  while true
3    do Prepend(bytes, n mod 128)
4      if n < 128
5        then Break
6      n ← n div 128
7  bytes[Length(bytes)] += 128
8  return bytes
```

```vb
VBEncode(numbers)
1  bytestream ← ∅
2  for each n ∈ numbers
3    do bytes ← VBEncodeNumber(n)
4      bytestream ← Extend(bytestream, bytes)
5  return bytestream
```
VB code decoding algorithm

```vbnet
VBDDecode(bytestream)
1    numbers ← ⟨⟩
2    n ← 0
3    for i ← 1 to Length(bytestream)
4        do if bytestream[i] < 128
5            then n ← 128 × n + bytestream[i]
6        else n ← 128 × n + (bytestream[i] − 128)
7            Append(numbers, n)
8    n ← 0
9    return numbers
```
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: *bitlevel* code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent $n$ as $n$ 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is
    111111111111111111111111111111111111111111111111111111111111111111111110
  - Unary code for 70 is:
    111111111111111111111111111111111111111111111111111111111111111111111110
Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example 13 → 1101 → 101 = offset
- Length is the length of offset.
- For 13 (offset 101), the length is 3.
- Encode length in unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.
### Gamma code examples

<table>
<thead>
<tr>
<th>number</th>
<th>unary code</th>
<th>length</th>
<th>offset</th>
<th>( \gamma ) code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10,0</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>10</td>
<td>1</td>
<td>10,1</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>110</td>
<td>00</td>
<td>110,00</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>1110</td>
<td>001</td>
<td>1110,001</td>
</tr>
<tr>
<td>9</td>
<td>1111111110</td>
<td>1110</td>
<td>101</td>
<td>1110,101</td>
</tr>
<tr>
<td>13</td>
<td>1110</td>
<td>1000</td>
<td></td>
<td>11110,1000</td>
</tr>
<tr>
<td>24</td>
<td>11110</td>
<td>11111111110</td>
<td>1111111110,1111111111</td>
<td>11111111110,00000000001</td>
</tr>
</tbody>
</table>
Properties of gamma code

- Gamma code is prefix-free
- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- $\Upsilon$ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$. 
Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at granularity of bits can be slow.
- Variable byte encoding is aligned and thus potentially more efficient.
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost.
## Compression of Reuters

<table>
<thead>
<tr>
<th>data structure</th>
<th>size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>~, with blocking, k = 4</td>
<td>7.1</td>
</tr>
<tr>
<td>~, with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
<tr>
<td>collection (text, xml markup etc)</td>
<td>3600.0</td>
</tr>
<tr>
<td>collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>T/D incidence matrix</td>
<td>40,000.0</td>
</tr>
<tr>
<td>postings, uncompressed (32-bit words)</td>
<td>400.0</td>
</tr>
<tr>
<td>postings, uncompressed (20 bits)</td>
<td>250.0</td>
</tr>
<tr>
<td>postings, variable byte encoded</td>
<td>116.0</td>
</tr>
<tr>
<td>postings, gamma encoded</td>
<td>101.0</td>
</tr>
</tbody>
</table>