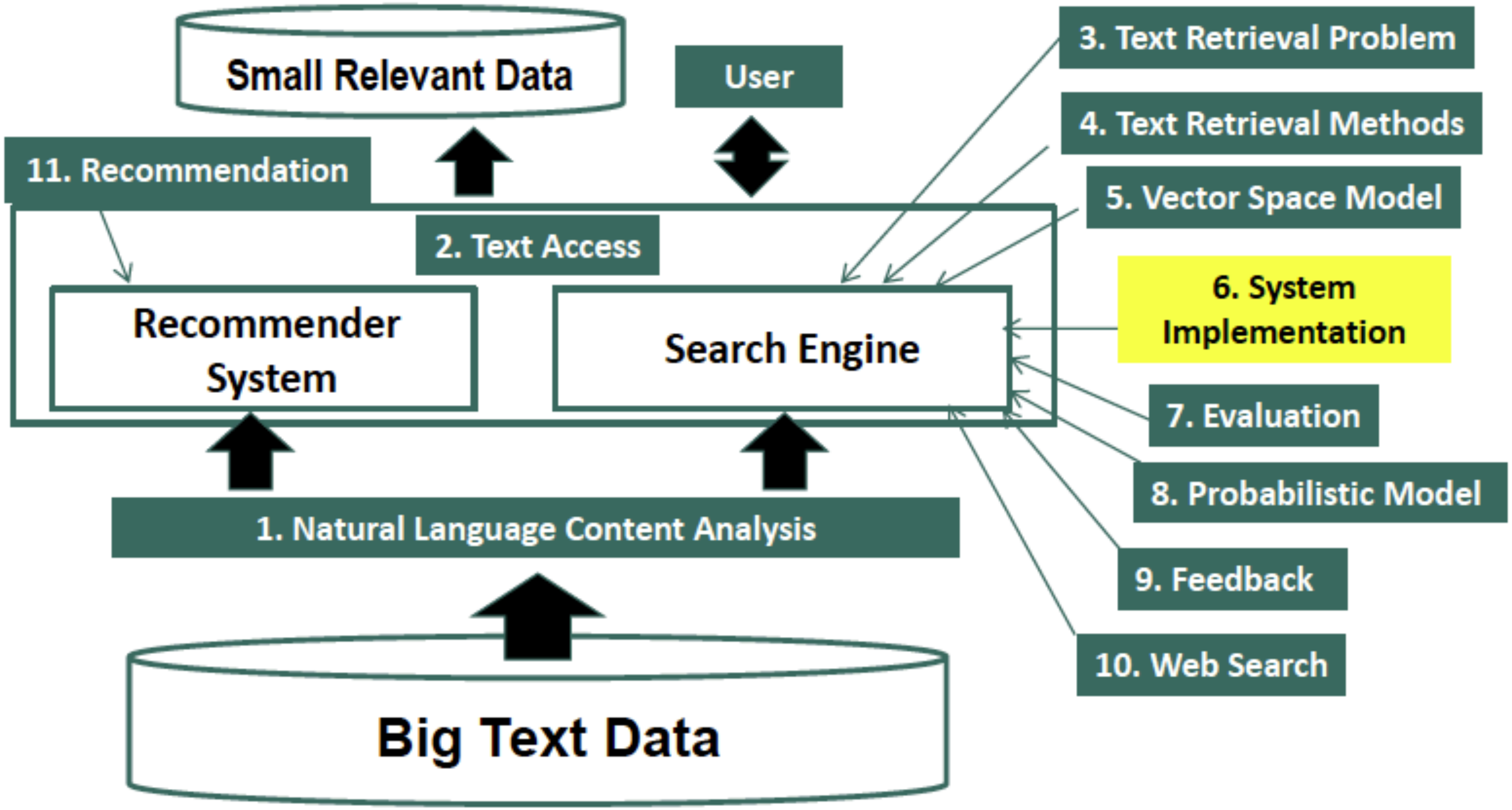
Text Retrieval & Search Engines

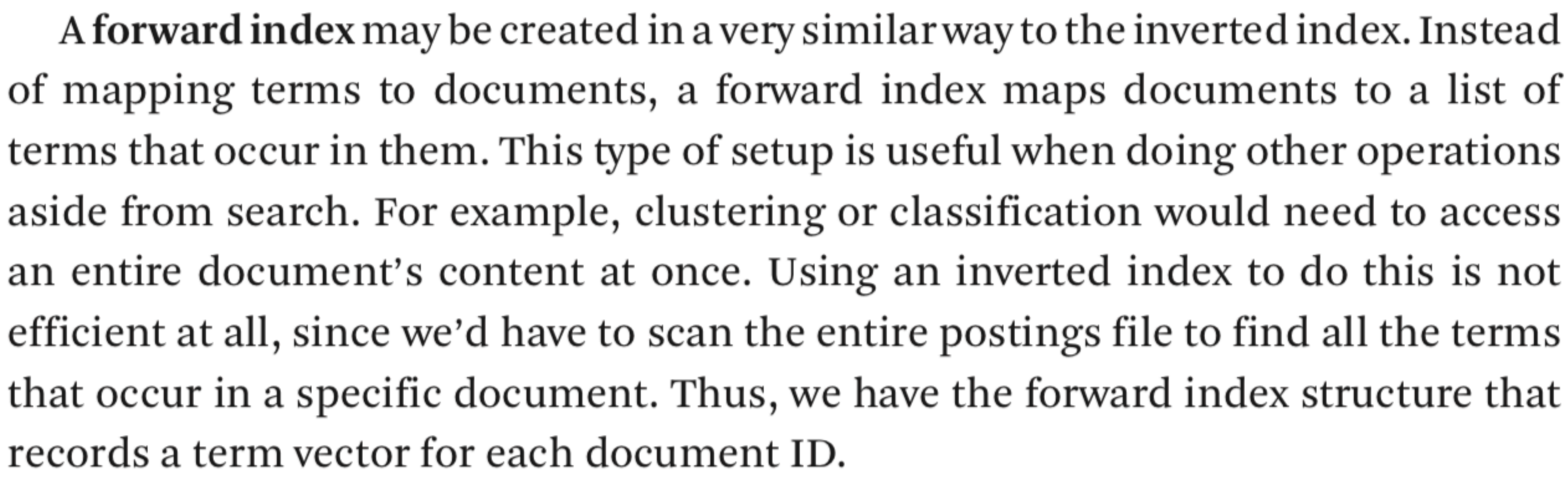
System Implementation: Inverted Index Construction

Dr. Iqra Safder

Implementation of Text Retrieval Systems



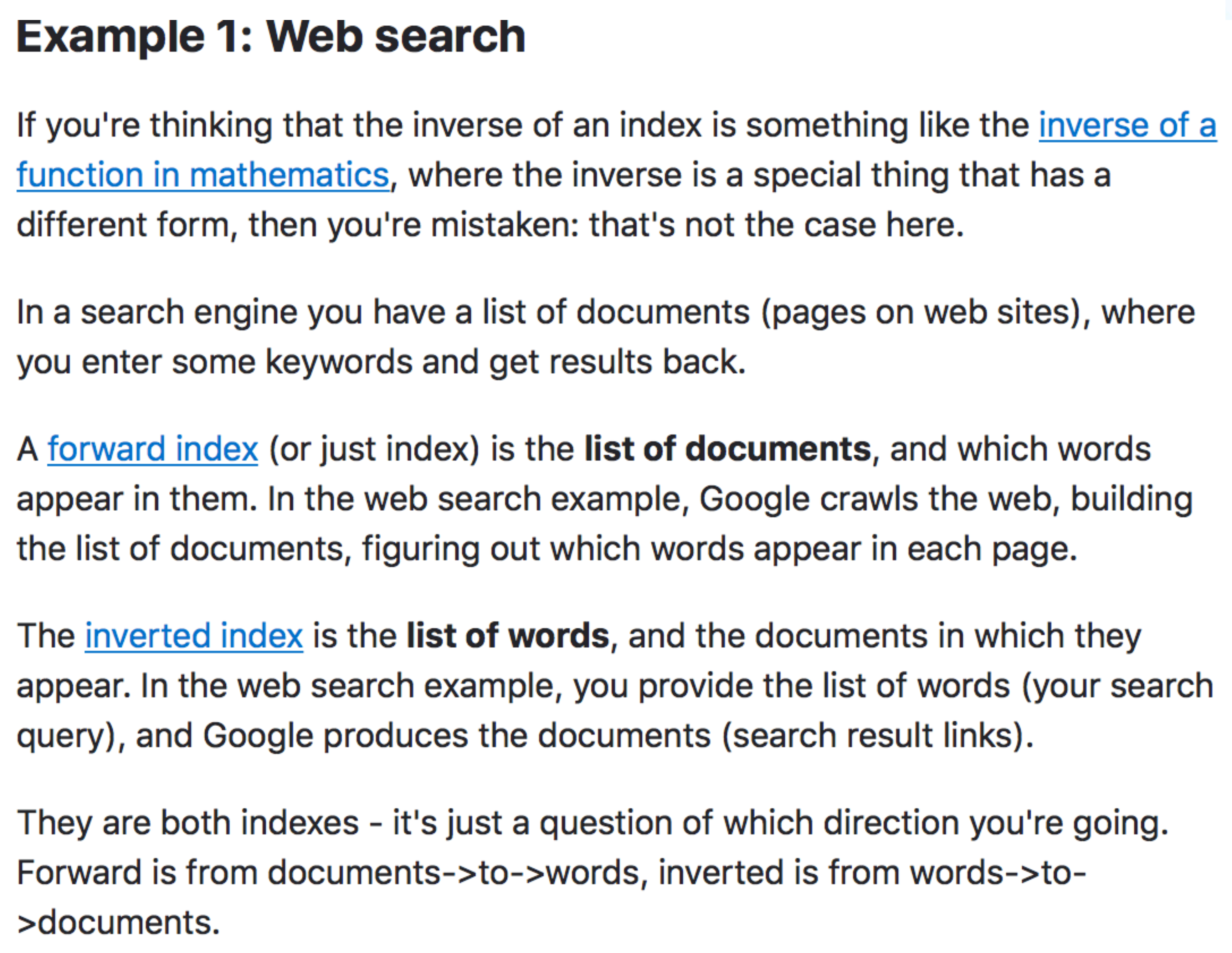
Forward Index



A **forward index** may be created in a very similar way to the inverted index. Instead of mapping terms to documents, a forward index maps documents to a list of terms that occur in them. This type of setup is useful when doing other operations aside from search. For example, clustering or classification would need to access an entire document's content at once. Using an inverted index to do this is not efficient at all, since we'd have to scan the entire postings file to find all the terms that occur in a specific document. Thus, we have the forward index structure that records a term vector for each document ID.

Indexing is the process of creating these data structures based on a set of tokenized documents

Forward Index VS Inverted Index



Example 1: Web search

If you're thinking that the inverse of an index is something like the inverse of a function in mathematics, where the inverse is a special thing that has a different form, then you're mistaken: that's not the case here.

In a search engine you have a list of documents (pages on web sites), where you enter some keywords and get results back.

A forward index (or just index) is the **list of documents**, and which words appear in them. In the web search example, Google crawls the web, building the list of documents, figuring out which words appear in each page.

The inverted index is the **list of words**, and the documents in which they appear. In the web search example, you provide the list of words (your search query), and Google produces the documents (search result links).

They are both indexes - it's just a question of which direction you're going. Forward is from documents->to->words, inverted is from words->to->documents.

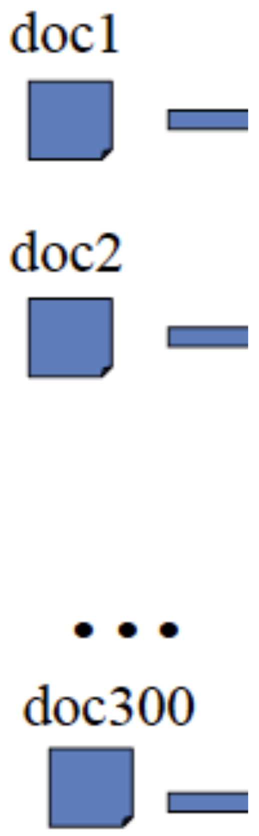
http://stackoverflow.com

Constructing Inverted Index

- The main difficulty is to build a huge index with limited memory
- Memory-based methods: not usable for large collections
- Sort-based methods:
  - Step 1: Collect local (termID, docID, freq) tuples
  - Step 2: Sort local tuples (to make "runs")
  - Step 3: Pair-wise merge runs
  - Step 4: Output inverted file

Indexing is the process of creating these data structures based on a set of tokenized documents

Sort-based Inversion



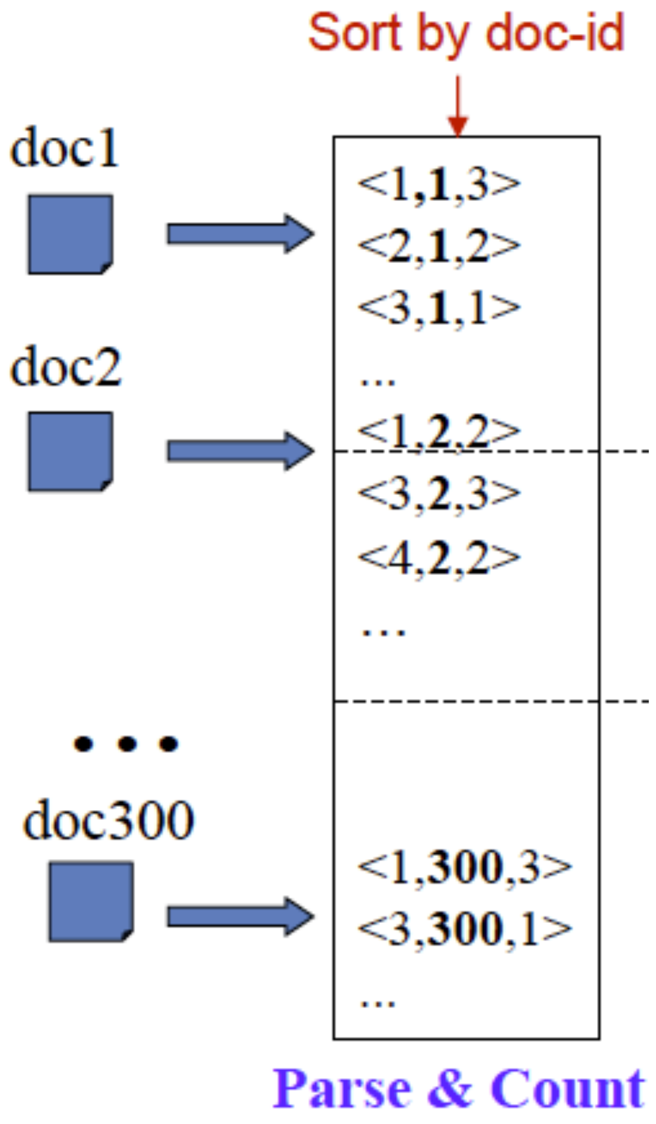
Term Lexicon:

the	1
campaign	2
news	3
a	4
…	

DocID Lexicon:

doc1	1
doc2	2
doc3	3
…	

Sort-based Inversion



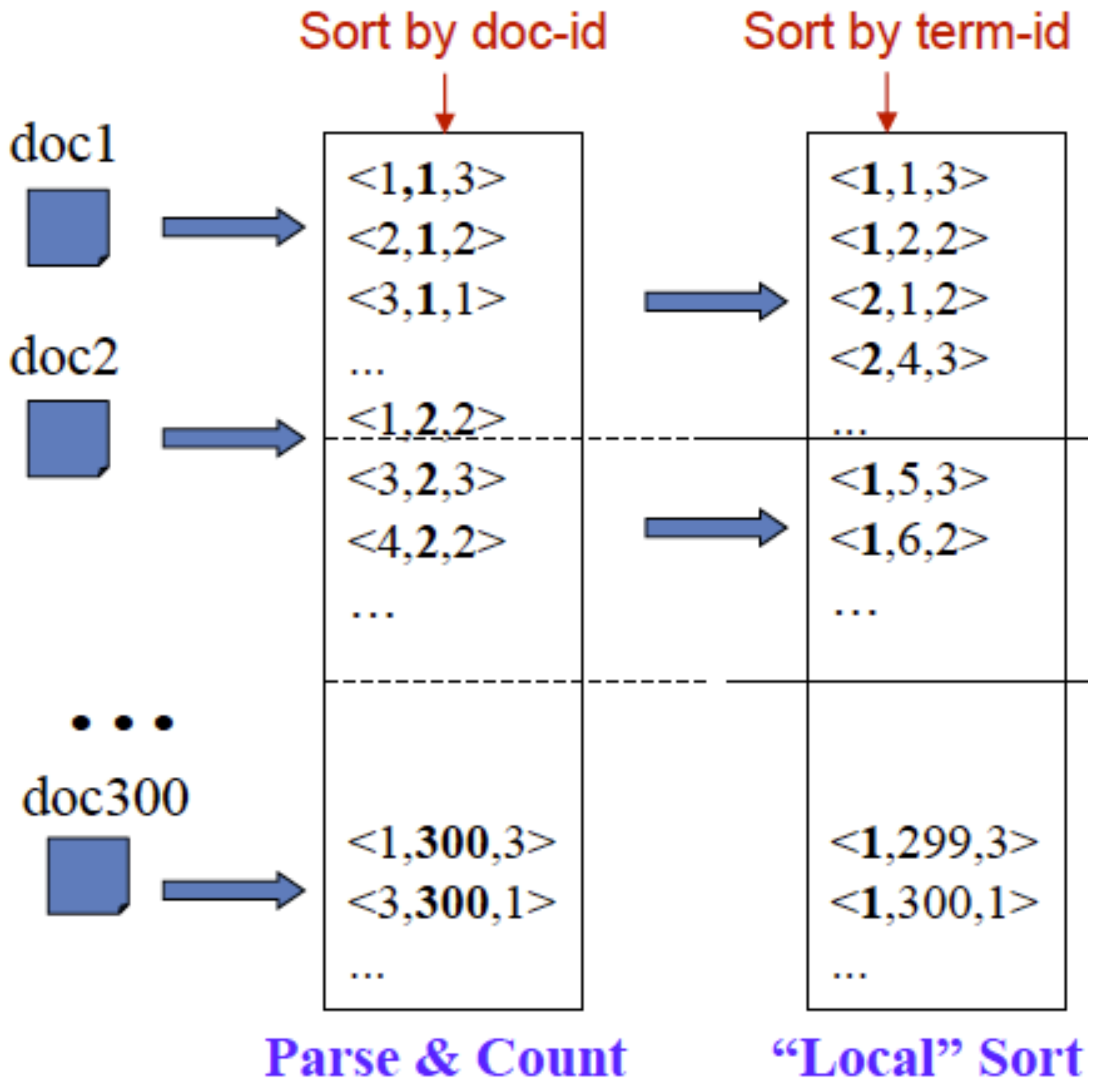
Term Lexicon:

the	1
campaign	2
news	3
a	4
…	

DocID Lexicon:

doc1	1
doc2	2
doc3	3
…	

Sort-based Inversion



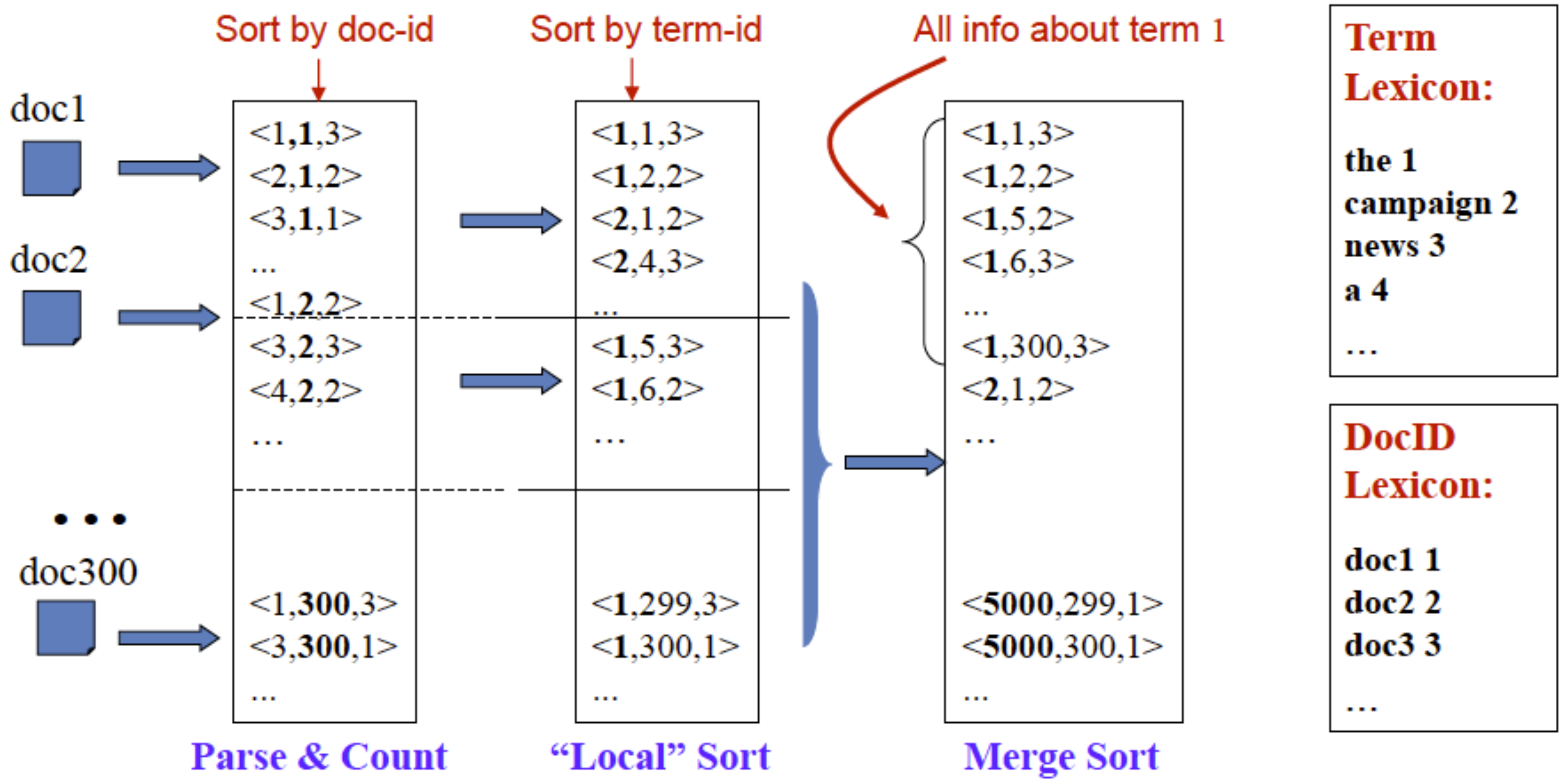
Term Lexicon:

the	1
campaign	2
news	3
a	4
…	

DocID Lexicon:

doc1	1
doc2	2
doc3	3
…	

Sort-based Inversion



Term Lexicon:
the 1
campaign 2
news 3
a 4
…

DocID Lexicon:
doc1 1
doc2 2
doc3 3
…

Inverted Index Compression

- In general, leverage skewed distribution of values and use variable-length encoding
- TF compression
  - Small numbers tend to occur far more frequently than large numbers (why?)
  - Fewer bits for small (high frequency) integers at the cost of more bits for large integers
- Doc ID compression
  - "d-gap" (store difference): d1, d2-d1, d3-d2,...
  - Feasible due to sequential access
- Methods: Binary code, unary code, γ-code, δ-code, ...

GAP Encoded Compression

It is important that all of our compression methods need to support random access decoding; that is, we could like to seek to a particular position in the postings file and start decompressing without having to decompress all the previous data.

$$\{23, 25, 34, 35, 39, 43, 49, 51, 57, 59, \ldots\}.$$

$$\boxed{\{23, 2, 9, 1, 4, 4, 6, 2, 6, 2, \ldots\}.}$$

To get the actual document ID values, simply add the offset to the previous value. So the first ID is 23 and the second is $23 + 2 = 25$. The third is $25 + 9 = 34$, and so on.

With bitwise compression, instead of writing out strings representing numbers (like "1624"), or fixed byte-width chunks (like a 4-byte integer as "00000658"), we are writing raw binary numbers. When the representation ends, the next number begins. There is no fixed width, or length, of the number representations. Using bitwise compression means performing some bit operations for every bit that is encoded in order to "build" the compressed integer back into its original form.

Integer Compression Methods

- Binary: equal-length coding
- Unary: $x \geq 1$ is coded as x-1 one bits followed by 0, e.g., 3=> 110; 5=>11110
- γ-code: x=> unary code for $1+\lfloor \log x \rfloor$ followed by uniform code for $x-2^{\lfloor \log x \rfloor}$ in $\lfloor \log x \rfloor$ bits, e.g., 3=>101, 5=>11001
- δ-code: same as γ-code ,but replace the unary prefix with γ-code. E.g., 3=>1001, 5=>10101

Unary. Unary encoding is the simplest method. To write the integer k, we simply write $k-1$ zeros followed by a one. The one acts as a delimiter and lets us know when to stop reading:

$$\begin{aligned}
1 &\rightarrow 1\\
2 &\rightarrow 01\\
3 &\rightarrow 001\\
4 &\rightarrow 0001\\
5 &\rightarrow 00001\\
19 &\rightarrow 0000000000000000001
\end{aligned}$$

Note that we can't encode the number zero—this is true of most other methods as well. An example of a unary-encoded sequence is

$$000100100010000000101000100001 = 4, 3, 4, 8, 2, 4, 5.$$

Gamma. To encode a number with γ-encoding, first simply write the number in binary. Let k be the number of bits in your binary string. Then, prepend $k-1$ zeros to the binary number:

$$\begin{aligned} 1 &\rightarrow 1 \\ 2 &\rightarrow 010 \\ 3 &\rightarrow 011 \\ 4 &\rightarrow 00100 \\ 5 &\rightarrow 00101 \\ 19 &\rightarrow 000010011 \\ 47 &\rightarrow 00000101111 \end{aligned}$$

To decode, read and count k zeros until you hit a one. Read the one and additional k bits in binary. Note that all γ codes will have an odd number of bits.

Uncompress Inverted Index

- Decoding of encoded integers
  - Unary decoding: count 1's until seeing a zero
  - γ-decoding
    - first decode the unary part; let value be k+1
    - read k more bits decode them as binary code; let value be r
    - the value of the encoded number is $\mathbf{2^k+r}$
- Decode doc IDs encoded using d-gap
  - Let the encoded ID list be x1, x2, x3, ….
  - Decode x1 to obtain doc ID1; then decode x2 and add the recovered value to the doc ID1 just obtained
  - Repeatedly decode x3, x4, …., and the recovered value to the previous doc ID.