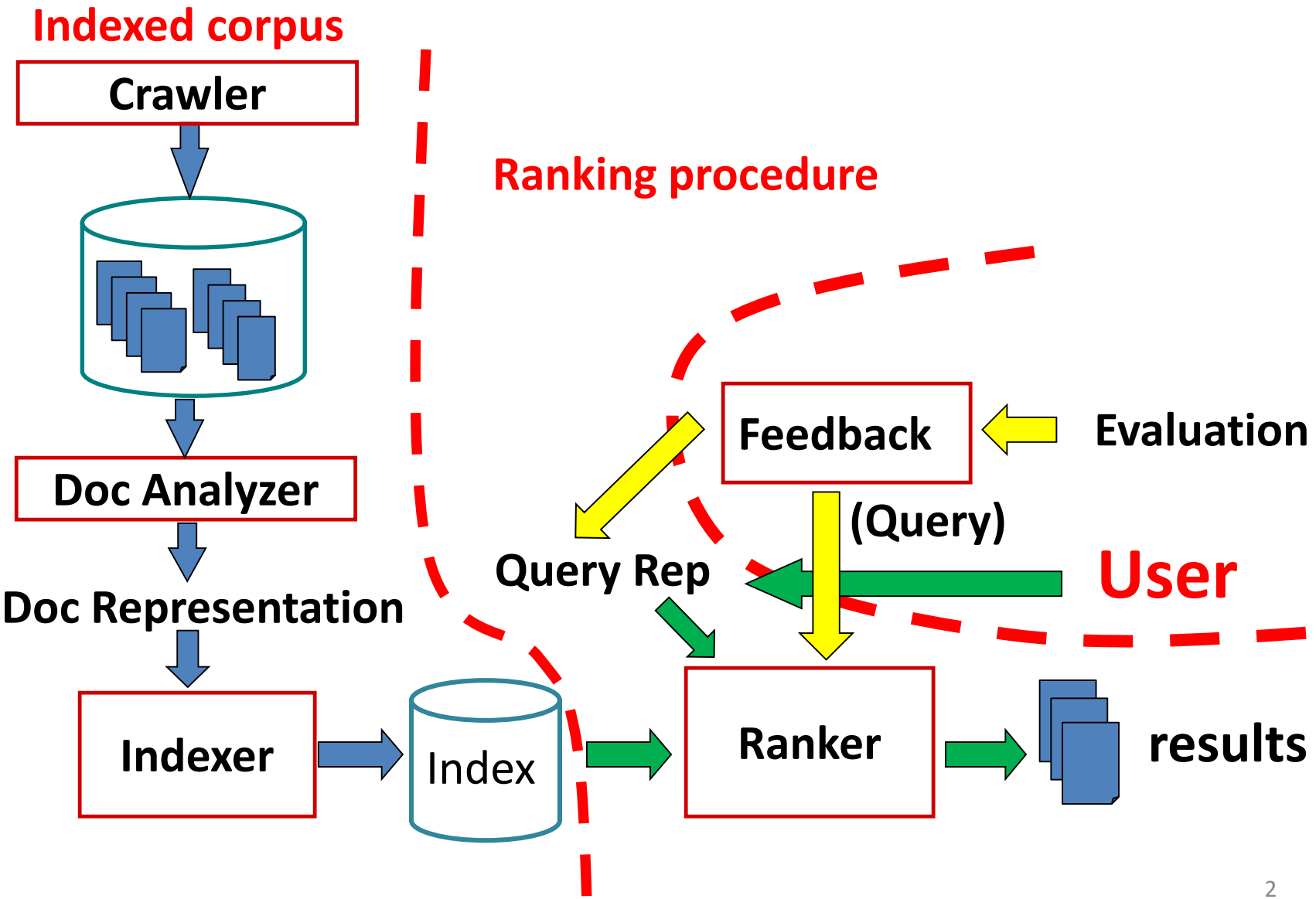
Inverted Index

Lecture 2

Abstraction of search engine architecture



What we have now

- Documents have been
  - Crawled from Web
  - Tokenized/normalized
  - Represented as Bag-of-Words
- Let's do search!
  - Query: "information retrieval"

	information	retrieval	retrieved	is	helpful	for	you	everyone
Doc1	1	1	0	1	1	1	0	1
Doc2	1	0	1	1	1	1	1	0

Term-document incidence matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if play contains word, 0 otherwise

Brutus *AND* ***Caesar*** *BUT NOT* ***Calpurnia***

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➔ bitwise *AND*.
  - 110100 *AND*
  - 110111 *AND*
  - 101111 =
  - **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Bigger collections

- Consider *N* = 1 million documents, each with about 1000 words.

- Avg 6 bytes/word including punctuation
  - 6GB of data in the documents.

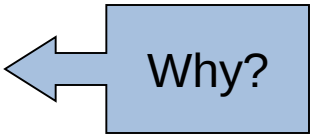
- Say there are *M* = 500K *distinct* terms among these.

- Size of collection = 10^9
- Size of term Doc matrix = $5*10^5 * 10^6$ $= 5*10^{11}$

- $= 5*10^{11} \quad / \quad 10^9$
- = 500 times

Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
  - matrix is extremely sparse.



- What's a better representation?
  - We only record the 1 positions.

Complexity analysis

- Space complexity analysis
  - $O(D * V)$
    - D is total number of documents and V is vocabulary size
  - Zipf's law: each document only has about 10% of vocabulary observed in it
    - 90% of space is wasted!
  - Space efficiency can be greatly improved by only storing the occurred words

Solution: linked list for each document

Solution: linked list for each document

Doc1: Term1, Term2, Term5….

Doc2: Term4, Term6, Term8….

Doc3: Term9, Term21, Term15….

Doc4: Term1, Term12, Term5….

Doc5 Term3, Term27, Term5….

……

Complexity analysis

- Time complexity analysis
  - $O(|q| * D * |D|)$
    - $|q|$ is the length of query, $|D|$ is the length of a document

```
doclist = []
for (wi in q) {
    for (d in D) {
        for (wj in d) {
            if (wi == wj) {
                doclist += [d];
                break;
            }
        }
    }
}
return doclist;
```

Bottleneck, since most of them won't match!

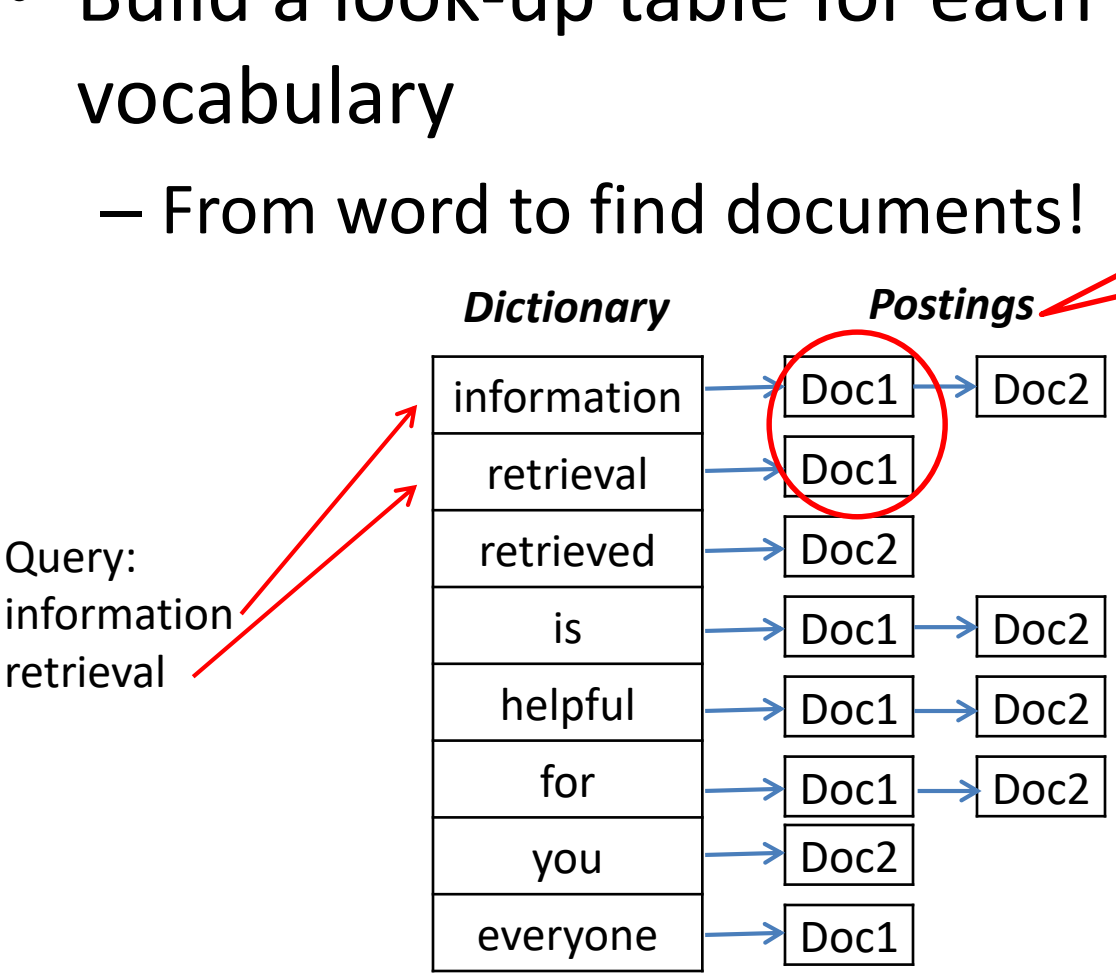
Solution: Inverted index

The Inverted Index

The key data structure underlying modern IR

Inverted index

- Build a look-up table for each word in vocabulary
  - From word to find documents!



Time complexity:

- $O(|q| * |L|)$, $|L|$ is the average length of posting list

Inverted Index

- If we have a corpus of 1 million documents, each of length 1,000 words, and a total vocabulary size of 500,000, what is the approximate maximum size of the postings and the size of the matrix (which contains a 1 in row *i* and column *j* if word *i* occurs in document *j* and a 0 otherwise), respectively?

A. 10^9 and 5×10^{11}
B. Both 10^9
C. 10^3 and 5×10^{11}
D. Both 5×10^{11}

Inverted Index

Answer: Option A: 10^9 and 5×10^{11}

If no word is repeated twice in a document, then the postings can at most reach a size of $1{,}000\times10^6=10^9$, whereas the matrix has 500,000 rows and 10^6 columns, yielding a total size of 5×10^{11} elements.

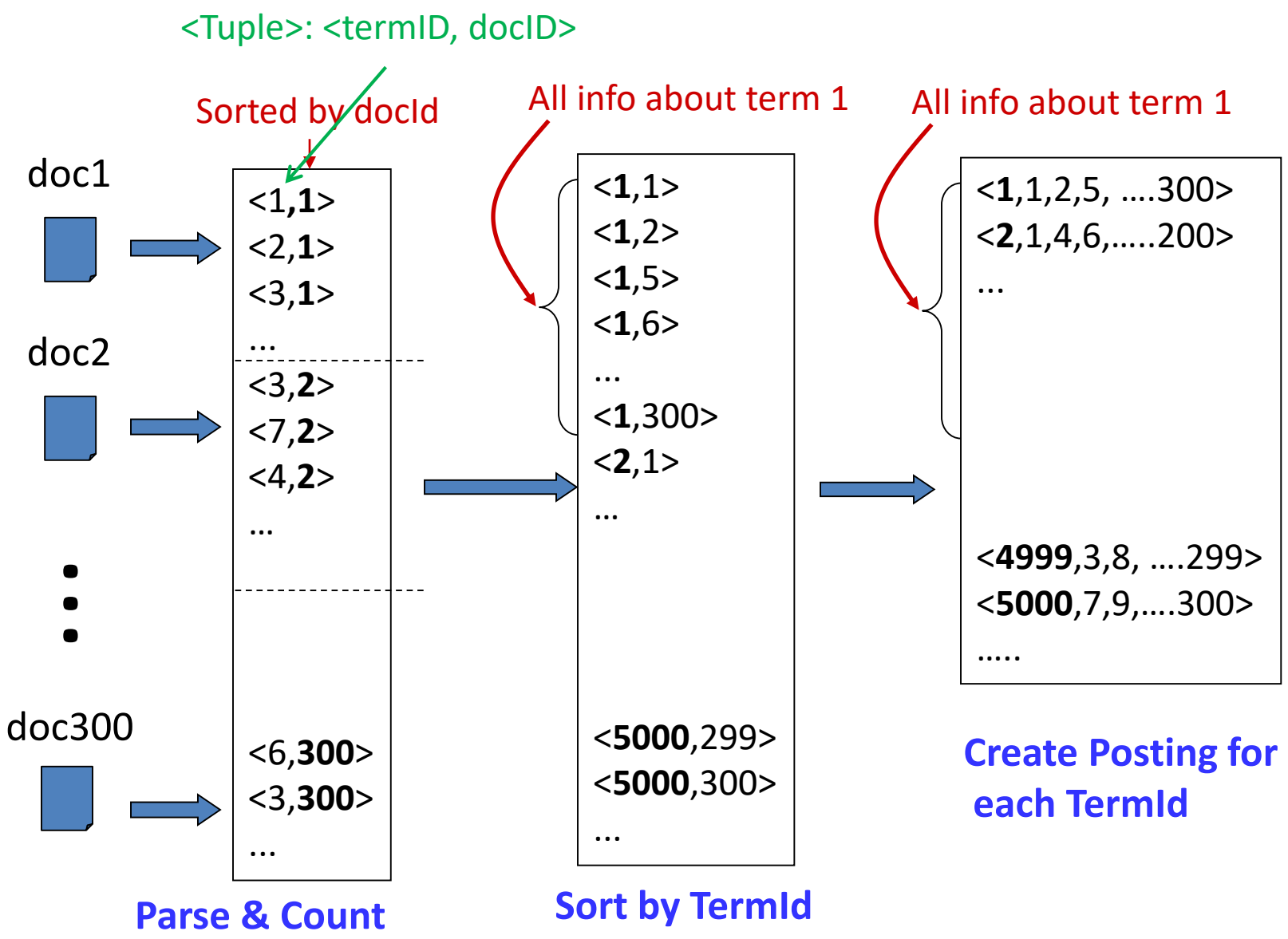
Structures for inverted index

- Dictionary: modest size
  - Needs fast random access
  - Stay in memory
    - Hash table, B-tree, …
- Postings: huge
  - Stay on disk
  - Contain docID, term freq, term position, …
  - Compression is needed

"Key data structure underlying modern IR"

- Christopher D. Manning

Sorting-based inverted index construction



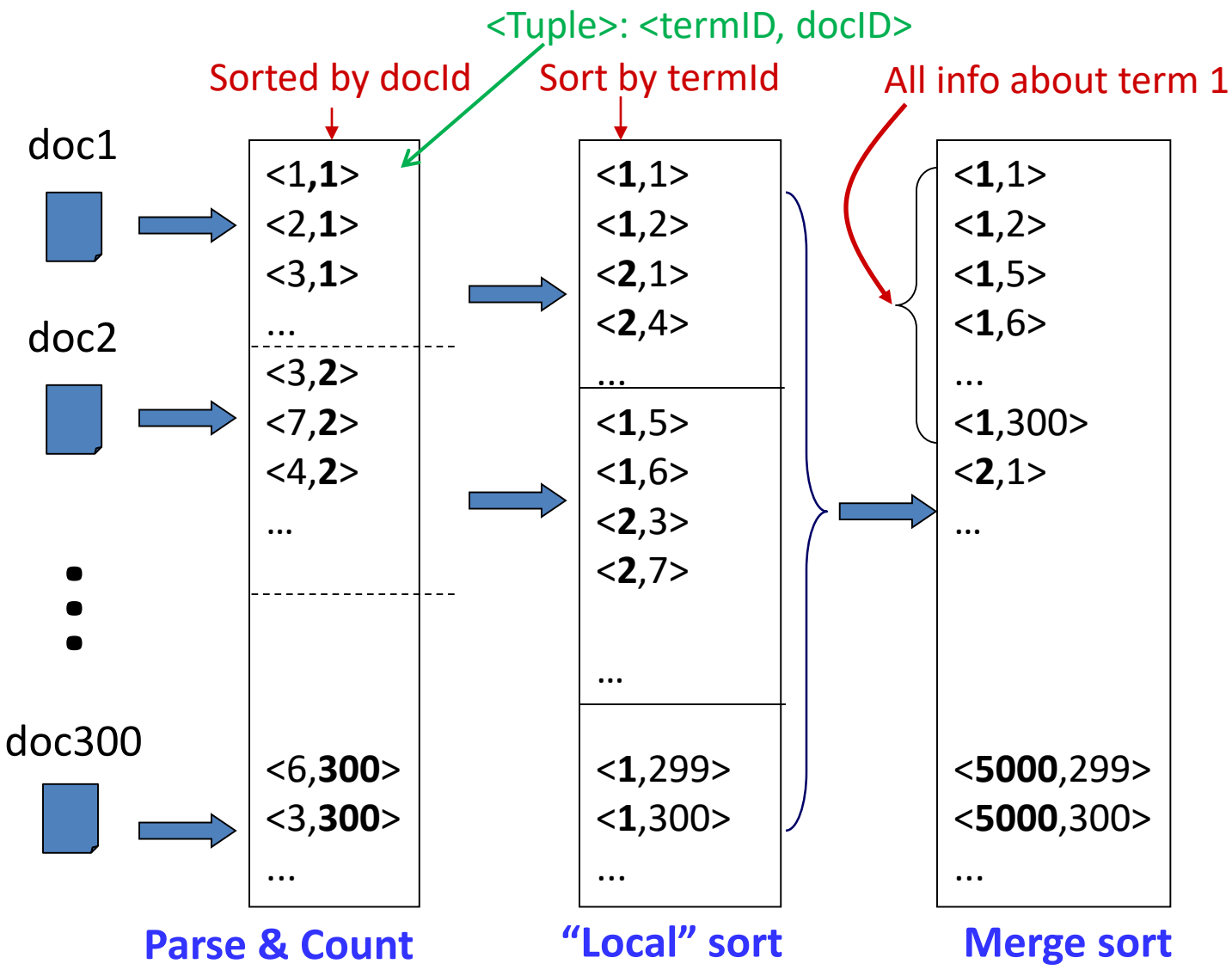
Problem

- All index data cannot fit in memory so how to sort all data based on terms

Solution

- Divide data in fixed size blocks that can fit in memory
- Create separate index for each block sorted on terms and then merge all indexes using Merge Sort

Sorting-based inverted index construction



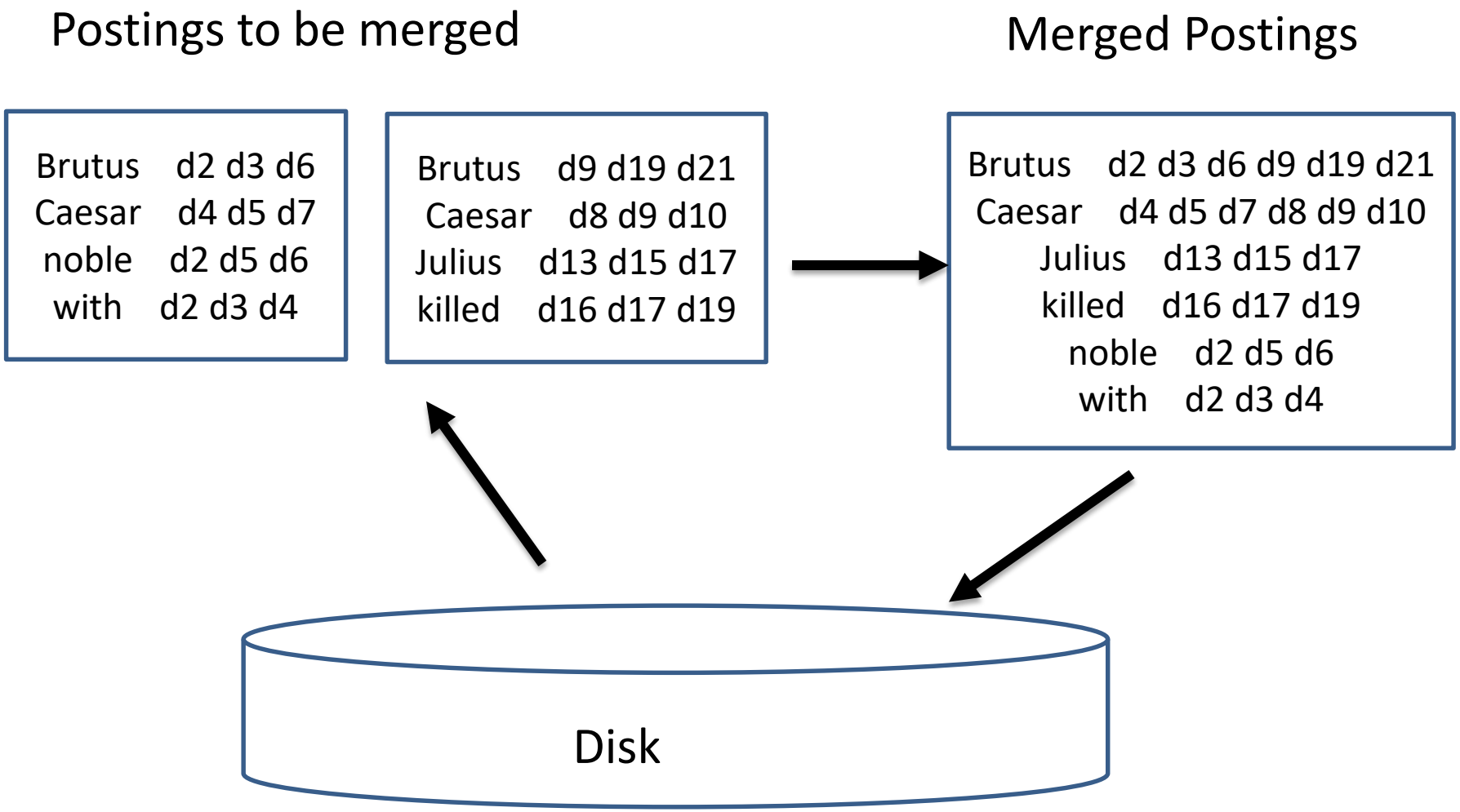
HashMap or Dictionary

- Do we need to sort all (termId, DocId) pairs ?
- We can use HashMap or Dictionary to create postings on the fly as we parse documents
- **Benefit**
- Number of items that need to be sorted is substantially reduced

SPIMI: Single-pass in-memory indexing

- Key idea 1: Divide data in fixed size blocks that can fit in memory
- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur using dictionary.
- 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.

Postings Merge



SPIMI-Invert

```
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
```

How to merge the sorted runs?

- It is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously

- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks

Slide Credits

- Lecture Notes, Text Retrieval and Mining by Christopher Manning and Prabhakar Raghavan, Stanford University