A first take at building an inverted index

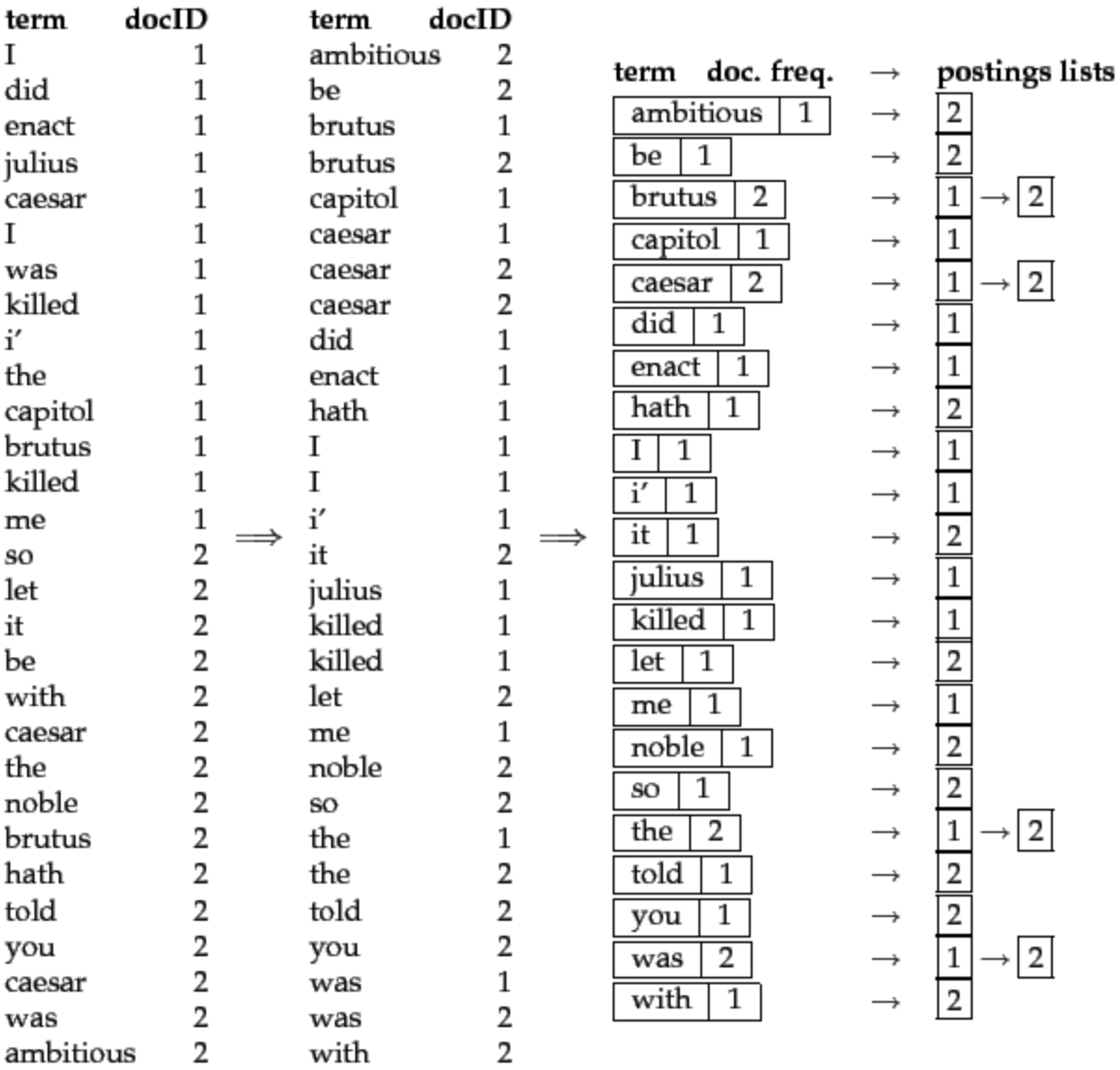
To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are:

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.

Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index by *sort-based indexing* .

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:



► **Figure 1.3** Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (left) is sorted alphabetically (middle). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (right). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (*docID*). During index construction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing is a list of normalized tokens for each document, which we can equally think of as a list of pairs of term and docID, as in Figure 1.4 . The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4 Multiple occurrences of the same term from the same document are then merged. Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings* , as shown in the right column of Figure 1.4 . Since a term generally occurs in a number of documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of documents which contain each term (the *document frequency* , which is here also the length of each postings list). This information is not vital for a basic Boolean search engine, but it allows us to improve the efficiency of the search engine at query time, and it is a statistic later used in many ranked retrieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. This inverted index structure is essentially without rivals as the most efficient structure for supporting ad hoc text search.

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is commonly kept in memory, while postings lists are normally kept on disk, so the size of each is important, What data structure should be used for a postings list? A fixed length array would be wasteful as some words occur in many documents, and others in very few. For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays. Singly linked lists allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated documents), and naturally extend to more advanced indexing strategies such as skip lists, which require additional pointers. Variable length arrays win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches. Extra pointers can in practice be encoded into the lists as offsets. If updates are relatively infrequent, variable length arrays will be more compact and faster to traverse. We can also use a hybrid scheme with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

Exercise.

- Write a program to create the inverted index and execute for the following document collection. (See Figure 1.3 for an example.)

  Doc 1 new home sales top forecasts
  Doc 2 home sales rise in july
  Doc 3 increase in home sales in july
  Doc 4 july new home sales rise