Map Reduce

Information Retrieval

Example: term frequency

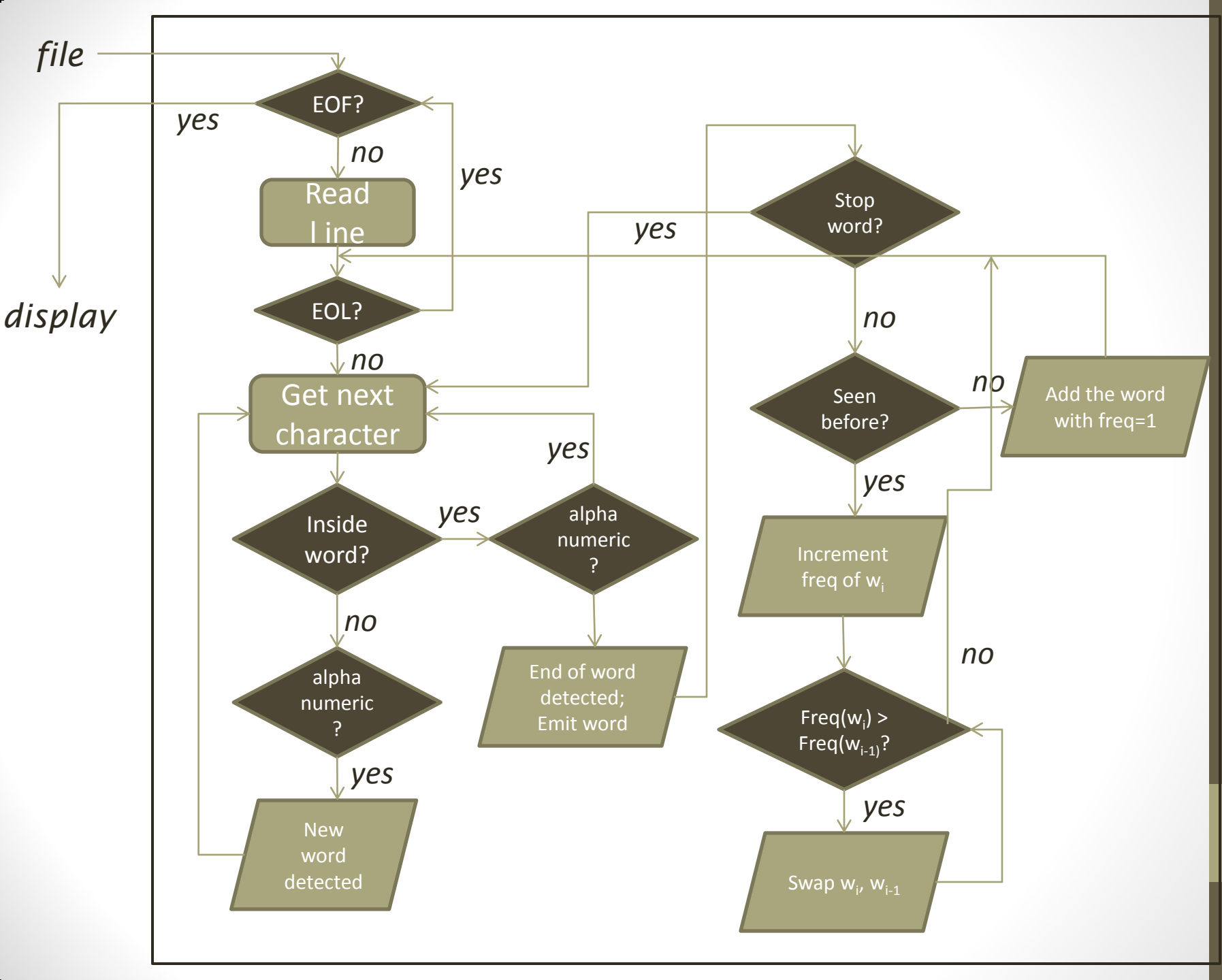
- Given one or more text files, produce the list of words and their frequencies, and display the N (e.g. 25) most frequent words and corresponding frequencies
  - Trivial problem

- How to do this?
  - Many *styles*, i.e. ways of thinking about the problem

Style 1: Monolithic

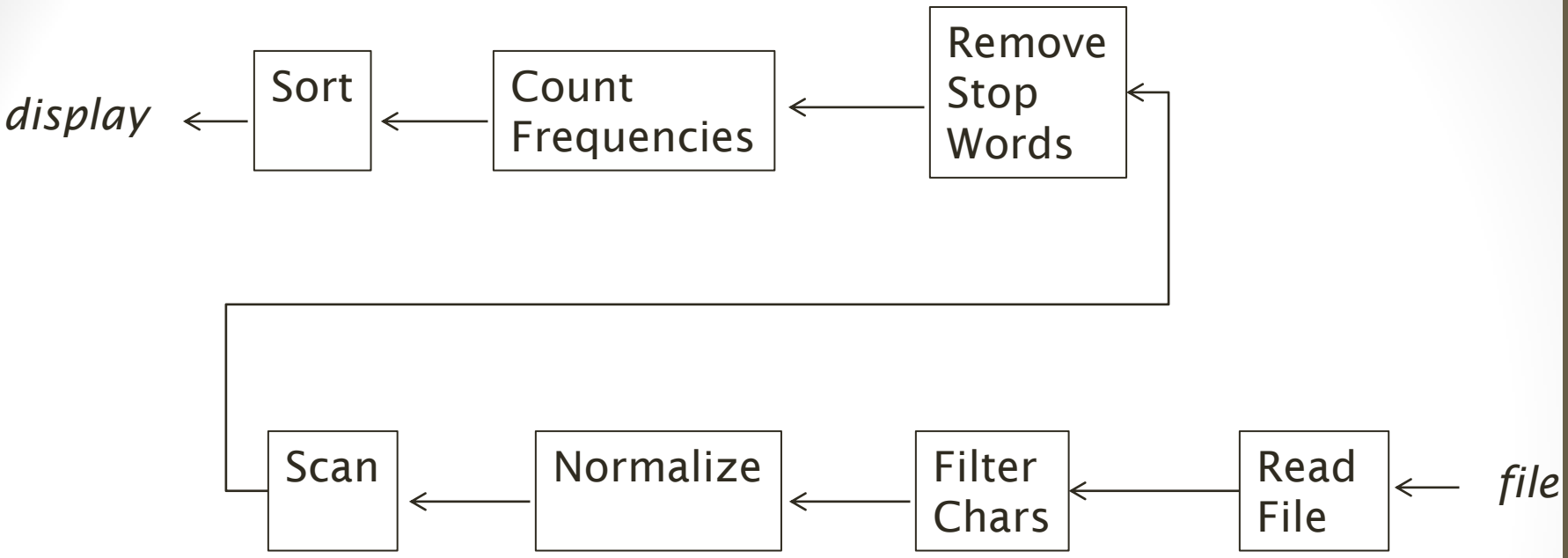
- The problem is not subdivided into smaller problems at all; instead, it is solved as a whole. The programming task consists in defining the data and control flow that rules this monolith.

- Main characteristics of this style:
  - One monolithic conceptual unit that takes input and produces the desired output
  - For all but the simplest problems, the control flow becomes very large
  - Potentially small memory footprint


file
EOF?
yes
no
Read
l ine
yes
display
EOL?
no
Get next
character
yes
Inside
word?
yes
alpha
numeric
?
yes
no
alpha
numeric
?
End of word
detected;
Emit word
yes
New
word
detected
Stop
word?
yes
no
Seen
before?
no
Add the word
with freq=1
yes
Increment
freq of w_i
no
Freq(w_i) >
Freq(w_{i-1})?
yes
Swap w_i, w_{i-1}

Style 2: Functional

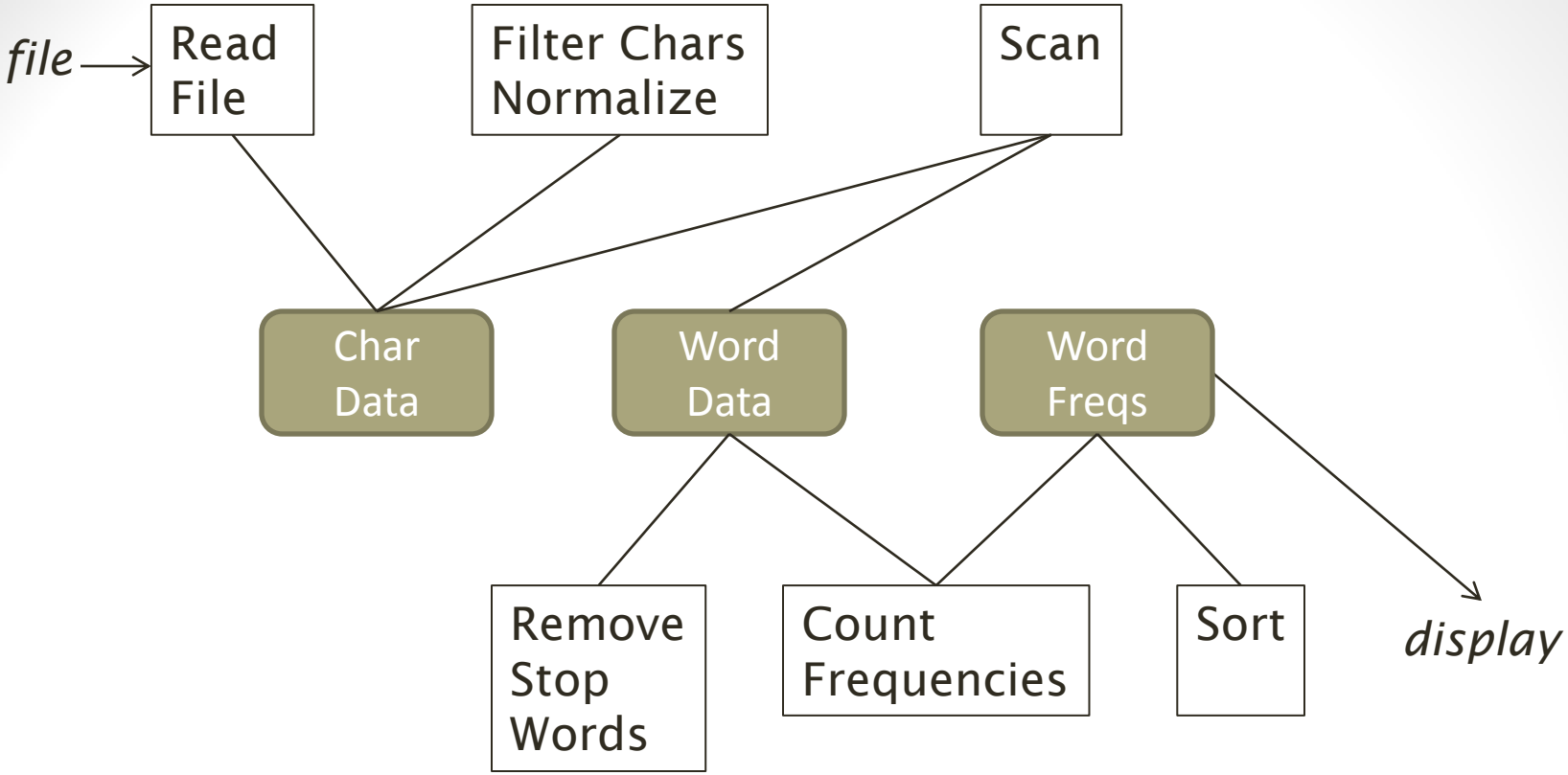
- Main characteristics of this style:
  - Functions take input, produce output, don't have side effects
  - There is no shared state
  - The large problem is solved by composing functions one after the other, as a faithful reproduction of mathematical function composition f ◦ g ("f after g")




```
sort(frequencies(remove_stop_words(scan(normalize(filter_chars(\
                read_file(sys.argv[1]))))))))
```

Style 3: Imperative

- Conceptual processing units similar to the functional style. Radical difference in the way that the units share data. In this style, the units all share a pool of state, which they read and modify.
- Main characteristics of this style:
  - Units may or may not take input, don't produce output relevant to the main problem
  - Units change the state of global, shared data
  - The large problem is solved by issuing commands, one after the other, that change, or add to, the global state



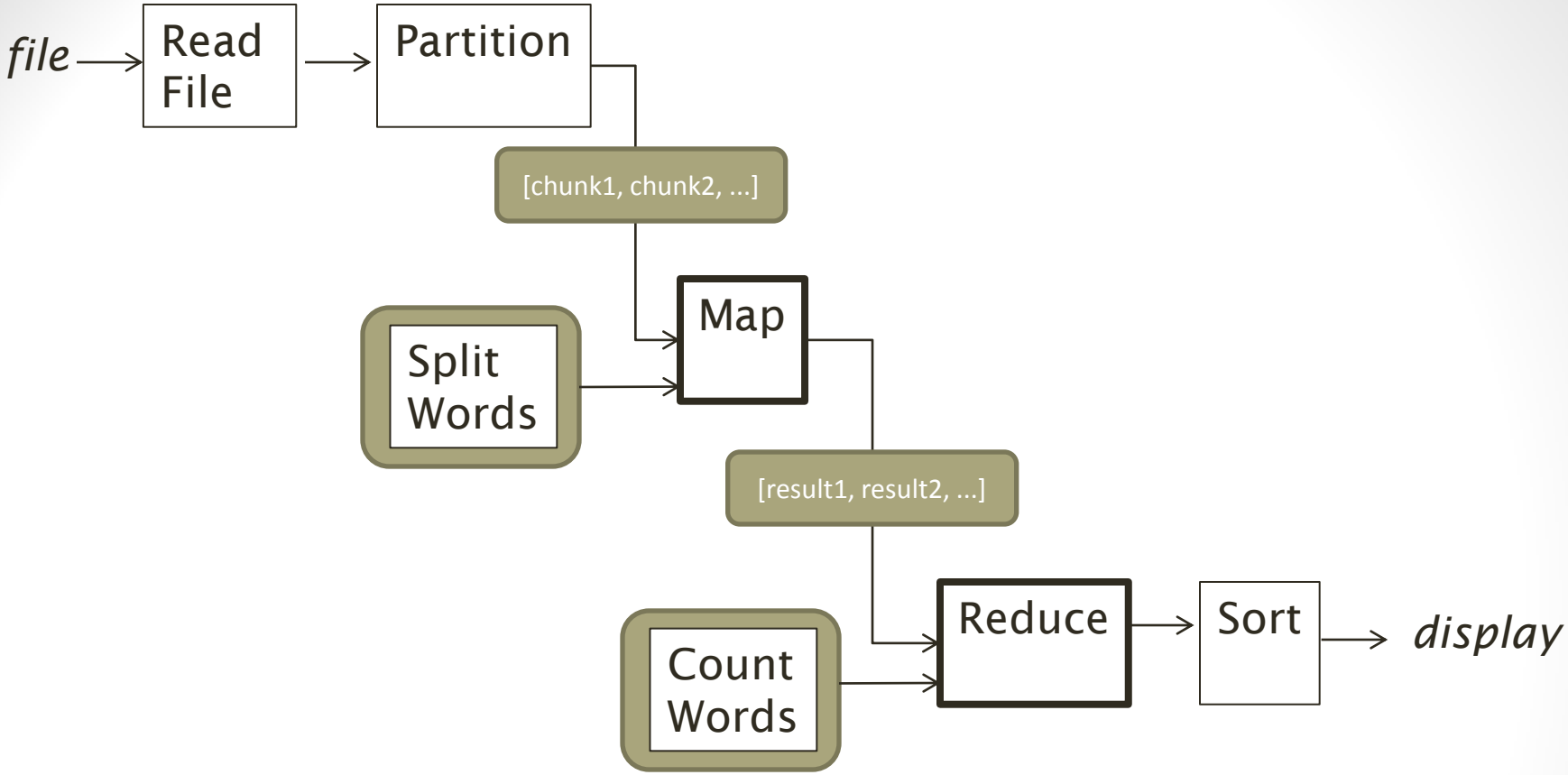

```
read_file(sys.argv[1])
filter_chars_and_normalize()
scan()
remove_stop_words()
frequencies()
sort()
```

Many more styles.....

Map-Reduce, Basic

- Key observation is that this problem of counting words can be done in a divide-and-conquer style over the input data.:
  - split the input into chunks, and process each chunk independently so to produce data that can then be reduced into counts of words.

- Main characteristics of this style:
  - Two key abstractions:
    - (1) **map** takes chunks of data and applies a given function to each chunk independently, producing a collection of results;
    - (2) **reduce** takes a collection of results and applies a given function to them in order to extract some global knowledge out of those results
  - Uses functions as data, i.e. as input to other functions
  - The map operation over independent chunks of data can be parallelized, potentially resulting in considerable performance gains



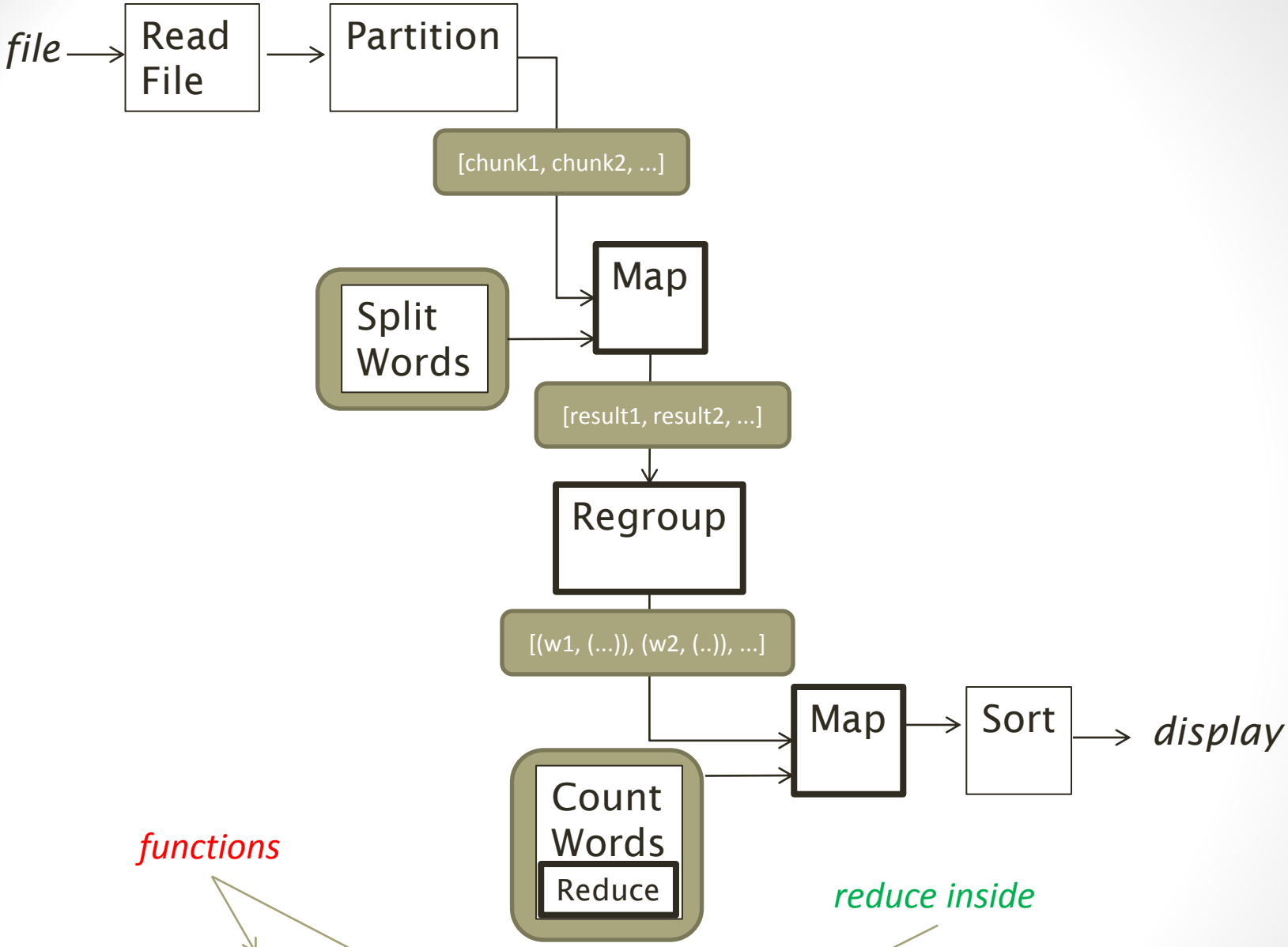
functions

```
splits = map(split_words,partition(read_file(sys.argv[1]),200))
splits.insert(0, []) # normalize input to reduce
word_freqs = sort(reduce(count_words, splits))
```

Map-Reduce, Hadoop

- The previous style allows for parallelization of the map step, but requires serialization of the reduce step. Google map-reduce and Hadoop use a slight variation that makes the reduce step also potentially parallelizable. The main idea is to regroup, or reshuffle, the list of results from the map step so that the regroupings are amenable to further mapping of a reducible function.

- Main characteristics of this style:
  - Similar to basic map-reduce, but with additional regroup (aka shuffling) step, followed by another map that maps a reducible function to a collection of inputs.




```
splits = map(split_words,partition(read_file(sys.argv[1]),200))
splits_per_word = regroup(splits)
word_freqs = sort(map(count_words, splits_per_word.items()))
```