# Design decisions document

## Anagram grouper

It’s a class that reads and writes to files based on a specific format:

* new line separated words
* the words must have basic alphabet letters from ‘a’ to ‘z’
* must all be lower case

The reason there is no internal state except read filename is to prevent bugs with multi-threading, is stupid proof since the user doesn’t need the extra step of initializing the state. The class implements the strategy pattern to allow for better maintainability. At any time you can swap between parallel and sequential strategies, and it also allows for faster and more intuitive testing. This design also allows for scalability, because you can use multiple instances to work on multiple files at the same time without interference.

There are multiple anagram grouping strategies, so the user can choose the fastest approach for each context. I also decided to implement that many strategies for fun and to learn. I got captivated quickly.

### Anagram grouper sequential

Used for files with less than 100 thousand words.

### Anagram grouper parallel

Used for files with less 100 million words. It doesn’t support super massive files like with 100 billion words. For that, I think it’s better to separate data into smaller files and use multiple AnagramGrouperParallel instances asynchronously (like in test2 from the TestKit).

### Anagram grouper sharding

Used for files larger than 100 million. I did not have time to properly limit test or optimize the algorithm. It uses a technique called sharding to distribute workload. Words are stored in different files based on their hash value. Later the files each anagram group is merged in separate respective files.

## TestKit

Class used for testing parameters for better performance, such as number of threads used and batches of words per thread. By comparison, I concluded that using 16 threads with batch size of: 100-000 is best for my setup (for AnagramGrouperParallel).

## Scalability considerations

In case of 100 billion words, I think it’s better to separate the data set into multiple files and run the asynchronous parallel operations within the RAM limit. Another thought is to use sharding to keep track of the processed words and later combine them or work with them from their respective shards. Another idea would be to use existing solutions like MapDB or RocksDB to store memory directly to disk while processing them.