

# Programming Assignment 1

Marcel Kyas

23rd January 2026

This assignment accounts for 10 % of your final grade. We have estimated that typical students in a team of two should each allocate 16 h to complete the assignment. This time does not include *debugging time*.

## Objectives and Overview

This assignment aims to get you familiar with programming in Go. The goal is to write a simple document indexer. The user provides a directory on the command line. The program shall scan all files in this directory and its subdirectories, building an index of documents that contain key terms.

We aim to use concurrency to index the files as quickly as possible.

## 1 Components

You shall build a single program. To speed up indexing, use the *map-reduce* pattern. One function may traverse the directory hierarchy. A second function may read a file and generate a word-frequency map, associating each word with its number of occurrences. Execute this function in a goroutine.

### 1.1 Document Indexing

You will build a simplified component of a search engine, comprising a document indexer and a system for determining document relevance to a specific search term. For this assignment, a search engine is a stateful object that ‘knows about’ a set of documents and supports various queries on those documents and their contents.

Identify documents by their file path names as unique identifiers. Documents consist of a sequence of terms. Terms are (always) the lowercase version of words; operations on the search engine and documents should therefore be case-insensitive. Be careful with your definition of term. When Shakespeare writes ‘o’er’, we would like to store this as one term instead of the terms ‘o’ and ‘er’.

The search engine functions as an index. That is, given a term, the search engine can return the set of documents (that it knows about) that contain that term.

The search engine can also find a list of documents (again, from among the set it knows about) relevant to a given term, ordered from most relevant to least relevant. It does so using a specific version of the tf-idf statistic (Spärck Jones, 1972).

The tf statistics measure the frequency of a term  $t$  in a document  $d$ . Let  $n_{t,d}$  be the count of occurrences of the term  $t$  in the document  $d$ . Then, we can define the term frequency as:

$$td(t, d) = \frac{n_{t,d}}{\sum_{t' \in d} n_{t',d}} . \quad (1)$$

The inverse document frequency (idf) relates to the number of documents in which a term occurs. Let  $D$  be the set of documents,  $N = |D|$  the number of documents, and  $n_t = |\{d \in D \mid t \in d\}|$  is the total number of documents that contain the term  $t$ . Then the idf can be defined as:

$$idf(t, D) = \log \frac{N}{n_t} . \quad (2)$$

Many systems weight documents by the *count-idf*:

$$tfidf(t, d, D) = td(t, d) \cdot idf(t, D) \quad (3)$$

We consider a term to have high weight if it occurs infrequently (high idf) and more frequently in that document (high tf). Terms that occur in every document have an idf of 0, and we do not order such documents.

The SearchEngine needs to keep track of documents for two purposes: to perform term index lookups (IndexLookup) and to compute the two components of the tf-idf statistics (TermFrequency and InverseDocumentFrequency). You can maintain this state with any data structures you prefer, but my suggestions follow.

I suggest you implement AddDocument and IndexLookup working first. To support the index, a straightforward mapping of terms to DocumentIDs will suffice. We can represent sets as a map from strings to empty structures, e.g. by declaring `type DocumentIDs map[string]struct{}`, and the index can be defined as a `map[string]DocumentIDs`.

I recommend reading the files line by line and splitting them using a regular expression. The `regexp` package and the `Split()` function should be used for this purpose. Remember to lowercase the result.

`TermFrequency` requires that you compute the number of times a given term appears in a given document. You should have a data structure that keeps track of the term count per document: a `map[string]int`. However, this frequency-counting structure is per document; you need to keep track of the counts for

each document. Overall, I suggest using a `map[string]map[string]int`. The outer map corresponds to the file name and leads to the inner frequency-counting structure. You'll have to update `AddDocument` to populate and update these structures.

Once you have this structure, implementing `InverseDocumentFrequency` is pretty straightforward. Use `math.Log` to calculate the logarithm.

Use these two methods to compute the TfIdf value for a given document-term pair.

Finally, implement `RelevanceLookup`, which returns a list of all documents containing a given term, sorted from largest tf-idf to smallest. If you do implement it, ensure it returns a value that sorted from largest to smallest, and consider the tie-breaker requirement.

## 1.2 Map-Reduce

Listing 1 illustrates how we can implement the map-reduce pattern. The function `Pmap` creates a go routine and supplies the data to process to the function `f`. The go routine sends the result on the channel `r`. The function `Reduce` receives those results and accumulates them with the function `g`. The result is accumulated in the variable `a`, which ought to be initialised to a neutral value.

A proper implementation of map-reduce uses a shuffle function to determine how to allocate tasks to machines or processes. We can leave this shuffling to the go scheduler for this assignment.

Note that file descriptors, the operating system's way of tracking opened files, are a limited resource. Ensure that the program does not exhaust this resource by limiting the number of goroutines that keep a file open.

## 2 Interfaces

Your program shall be invoked by

```
go run indexer.go ${DIRECTORY}
```

The parameter `${DIRECTORY}` shall be the name of a directory on your file system. You can open and read a directory using the `ReadDir()` function of the `os` package.

Once the program has built the index, it shall read terms from standard input. For each term, it should look up that term and output lines of the form

```
== term (count)
document,tdidf
document,tdidf
```

to report the lookup result. Observe the double equals followed by the search term on the first line.

For example, searching for `Juliet` may result in the output

```

func Pmap(data []int, f func(int)int, r chan int) {
    for _, e := range data {
        go func(v int, r chan int) { r <- f(v) }(e, r)
    }
}

func Reduce(N int, r chan int, g func(int,int)int, a int) int {
    for range(N) {
        a = g(a, <-r)
    }
    return a
}

func main() {
    var data []int = []int{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
    ret := make(chan int, len(data))
    Pmap(data, func(v int) int { return v * v }, ret )
    x := Reduce(len(data), ret,
                func(x, y int) int { return x + y }, 0)
    println(x)
}

```

Listing 1: An example of the map-reduce pattern

```
== juliet (2)
plays/romeo-and-juliet_TXT_FolgerShakespeare.txt,0.018725
plays/measure-for-measure_TXT_FolgerShakespeare.txt,0.002032
```

## 3 Testing

We provide a zip file of test files. The file contains text files of Shakespeare's works.

We will call your program using

```
cat terms | go run indexer.go ${DIRECTORY} > output
diff reference output
```

We will test your code using an undisclosed set of files. This test is to ensure that you do not overspecialise your implementation to the work set.

We will also perform a code review to ensure that you process documents concurrently.

## 4 Explain your code

You have a certain scope of discretion. If you are not sure about how to interpret the requirements, document your interpretation.

Your explanation shall address the following questions:

- (a) How do you avoid data races?
- (b) Why is your solution deadlock-free?
- (c) How do you handle errors in go-routines?

Always reference your code by filename and function. Add comments or indicate line numbers to help us locate the relevant section of code more quickly.

## 5 Milestones

You have two weeks to complete this assignment. The weekly milestones below outline what you should be able to complete by the end of each week.

**Milestone 1** You should be able to parse the command line arguments, count the words in a file, read terms from standard input, and print relevant words.

**Milestone 2** You scan files concurrently without exhausting system resources.

Make sure that inputs and outputs conform to the specification. And do not forget the documentation.

## 6 Submission

Your submission should be a single ZIP file named `indexer-{$GROUP}.zip`. This file must include:

- A file called `AUTHORS` that lists the name and e-mail at `ru.is` of every group member contributing to this assignment.
- A `README` file containing a description of each file and any information and instructions you feel the TA needs to grade your assignment. The file may be in Markdown, AsciiDoc, or reStructuredText format. It must not have a suffix.
- A directory called `agentlogs` containing all logs of interactions with AI systems, as specified in Section 7.2.
- All go files related to the assignment in the root directory.

Your answers and justifications to the questions in Section 4 can be part of the `README` file. It can also be in source code comments. It can be any combination of these. Please clearly specify where we find your documentation in the `README` file.

You **must not** include any compiled files, the problem PDF, or the test cases. Keep your submission as small as possible and do not include what we already provide.

## 7 Grading

This assignment counts for 10 % towards your final grade and has a maximum score of 20 points. The programming component accounts for 40 % with the explanation accounting 60 %.

We award points according to the rubric in Table 1.

### 7.1 Deduction

You should document all code according to the guidelines at `go.dev/doc/comment`. Points can be deducted for code features mentioned in Table 1 if:

1. Bad formatting and poor readability (use `go fmt` to fix your formatting).
2. Lack of documentation and explanations for design choices.

We will deduct points for written explanations that do not reference code by file name, function name, line numbers or labels.

Table 1: Rubric

Points	Feature
1	The submitted zip file meets the requirements of Section 6, e.g. files are in the right directory, filenames are correct, and all content is there.
1	The command-line arguments are parsed correctly. The terms are read correctly from standard input.
1	The output conforms to the expected format in Section 2.
1	All files are read, and their terms are indexed.
1	The tf, idf, and td-idf statistics are calculated correctly
2	The program performs the indexing concurrently and uses a go routine for each file.
1	The program does not run out of resources.
3	There is a precise definition of data flow. What data flows through channels, what flows through shared data structures? Shared data for each part of the code, i.e. what each function reads or writes.
3	There is a valid explanation of how the number of goroutines is determined.
3	There is a valid explanation that as many goroutines are executed as possible. We assess this criterion based on the number of goroutines your solution returns.
3	There is a valid explanation of deadlock freedom and progress.

## 7.2 AI Policy

All use of artificial intelligence tools shall be in compliance with the ‘Guiding Principles for the Use of Artificial Intelligence in Teaching and Learning at RU’.

You must disclose all use of AI. Since the logs on your assistant’s server are ephemeral, you must download and archive them. Submit them in a directory called `agentlogs`. Give each file a unique name ending in `.log`. Each file must be a UTF8 coded plain text file.

Document use of coding assistants in your source file. If your code is the result of a conversation with a coding agent, archive this conversation in the directory `agentlogs`.

## 7.3 Your own work

Plagiarism results in 0 points for this assignment. See RU Library site about Plagiarism, RU Project Code of Conduct and Section 7: Procedures and appeals of RU’s General rules on study and assessment for more information.

If you use another person's work in yours, you must give credit and add a proper citation to your `README` file. If you import and adapt code, add a comment in front of that code explaining its origin, if and why it was changed, and how it was changed.

Note that not documenting your use of AI according to the rules in Section 7.2 are considered plagiarism.

## References

- Spärck Jones, Karen (Jan. 1972). 'A Statistical Interpretation of Term Specificity and Its Application in Retrieval'. In: *Journal of Documentation* 28.1, pp. 11–21. DOI: [10.1108/eb026526](https://doi.org/10.1108/eb026526).

## 8 Changelog

**1.0.1** (2026-01-23)

### **Changed**

- Changed the output format in section 2. The document frequency is the same for each file, so display it after the term.

### **Added**

- Commented on the definition of term and that ‘o’er’ is best seen as one rather than two terms in section 1.1.
- Require to document discretionary scope in section 4.
- Added the requirement to only include necessary files in the submission in section 6.

**1.0.0** (2026-01-13)

### **Added**

- Initial version