Problem Statement

Write a program that reads a file and finds matches against a predefined set of words. There can be up to 10K entries in the list of predefined words.
Requirement details:

- Input file is a plain text (ascii) file, every record separated by a new line.
- For this exercise, assume English words only
- The file size can be up to 20 MB
- The predefined words are defined in a text file, every word separated by a newline. Use a sample file of your choice for the set of predefined keywords for the exercise.

Understanding of the Problem

We have been given some predefined set of words; size of the set could be up to 10K. And then we have been given input file of size 20MB. Which is plain text file consisting new line separated records. Problem states to assume English words only. So, we could make assumption that input file will consist of whitespace separated English words and may contain grammatical special characters for example ‘!’, ‘.’, ‘,’, ‘;’.

Now, the ask we have to process input file and find matches against the predefined set of words. Input file is very large, and it will consist of very large number of English words, those words need to be extracted to be searched against the set of words. Extraction of words need to be taken care against the special characters that word may have in it.

Once we extracted the single word from input file. We could search against the set of predefined words and add it to output list. There could be many the same word in the input file, so we could just the add count of occurrence as well.

Approach

Data Structure

1. **Hash table**

Since we have predefined set of words, we could have them preprocessed and stored in in-memory data structure, we could then apply some efficient search when we iterate through the words from input file.

Since 10K words not considerably very big, ideally in any given standard system, they could easily fit in memory. So, we could use hash table to store the set of words.

Since words are known prior and size is fixed, we should be able to come up with perfect hashing function that minimizes hash collision, which could result in O (1) time complexity for search of given word.

2. **Trie**

   Alternatively, we could use Trie data structure. Advantages of using Trie, this could serve many use-cases, like not just word search, we could efficiently do prefix search, all the keys with common prefix. But the problem does not state any other use-cases.

Data Structure comparison

Time complexity:

If we have average length of word N and total of M words.

Trie: O (M * N) time complexity
Hash table: O (M * N) will also would need same amount if chosen hash function takes linear time, linear in length of the string, for calculating the hash value of string. There are hash functions whose hash value construction for strings faster than that.

Space complexity:

Trie: O (M * N* K) worst case if there no overlapping words, where K is the length of children node references at each Trie Node for the next level. For English words, it could be taken as 26 or 52 depending upper case or lower-case support.

Hash Table: O (M * N) it will have all the stored in the table.

Overall, Trie Data structure will be more compact and consume less memory on average case.

Algorithm

Load the input file for processing, though input is very large, underneath system call will not directly load entire file into memory, it will just give file handle with which we could just load the buffer from file as and how we want.

We could read line by line, extract whitespace separated word, trim any special characters if it has any.

Once we extract the single word, we could search that against the preprocessed set of words for its presence.

Searching for word in both Hash table or Trie both incur same time complexity. In Hash table search O (1) but Hash index calculation may take O (N), N is the length of word and Trie it would take O (N) iteratively traversing the tree.

Trie could have better time complexity on average because it could have early termination for the words not present.

Program

1. **Hash Table**

```
from collections import Counter
import re

# This is the hash table data structure for our predefined words.
predefined_words = set()

# This data structure used for storing the matched words for processing later.
# Problem does not state the complete use case. So, implementation writes the matched
word and it's count to file
matched_words = Counter()



def process_predefined_words(keyword_file: str, ignore_case=True) -> None:
    """
    This function process the keyword_file which contains the pre-defined words, parses
the file and stores the words
    into hash table data structure
    :param keyword_file: This file holds the all the predefined set of words, newline
separated.
    :param ignore_case: Indicates if case-sensitive or not
    :return:
    """
    with open(keyword_file, 'r') as file:
        for line in file:
            line = line.strip()
            word = line.lower() if ignore_case else line
            predefined_words.add(word)



def clean_word(word: str):
    return re.sub(r'[^a-zA-Z]', '', word)



def extract_word_from_input(input_file: str, ignore_case=True):
    """
    It processes and parses the input file. Which could be very large. This function
extracts the word yields it to the
    caller.

    :param input_file: Input file which need to processed and matched against the
predefined set of words.
    :param ignore_case: Indicates if case-sensitive or not
    :return:
    """
    with open(input_file, 'r') as in_file:
        for line in in_file:
            words = line.strip().split()
            for word in words:
                word = word.lower() if ignore_case else word
                yield clean_word(word)



def find_matches(input_file: str, keyword_file: str):
    """
    This function finds the matches against the predefined set of words by extracting
words form input file.

    :param input_file: Input file which need to processed and matched against the
predefined set of words.
    :param keyword_file: This file holds the all the predefined set of words, newline
separated.
    :return:
    """
    process_predefined_words(keyword_file)
    for word in extract_word_from_input(input_file):
        if word in predefined_words:
            matched_words[word] += 1



def main():
    input_file = r"C:\Users\kiran\PycharmProjects\pythonProject\input_file.txt"
    predefined_words_file =
r"C:\Users\kiran\PycharmProjects\pythonProject\predefined_words_file.txt"
    matched_words_output_file =
r"C:\Users\kiran\PycharmProjects\pythonProject\matched_words_output_file.txt"

    find_matches(input_file, predefined_words_file)

    with open(matched_words_output_file, 'w') as out_file:
        for key, val in matched_words.items():
            out_file.write(f"Word: {key} Count: {val}" + "\n")



if __name__ == "__main__":
    main()
```

Input file:

```
I have an apple, and a banana.
Would you like some cherry pie?
Dates are sweet.
Fig! It's good.
```

Predefined words file:

```
apple
banana
cherry
date
elderberry
fig
grape
honeydew
```

Program output file:

```
Word: apple Count: 1
Word: banana Count: 1
Word: cherry Count: 1
Word: fig Count: 1
```

2. Trie

```python
from collections import Counter
import re



class TrieNode(object):
    def __init__(self, ignore_case=True):
        self.is_word = False
        self.children = [None] * 26 if ignore_case else [None] * 52



class Trie(object):
    def __init__(self, ignore_case=True):
        self.ignore_case = ignore_case
        self.root = TrieNode(ignore_case)

    @staticmethod
    def _get_index(char):
        if 'a' <= char <= 'z':
            return ord(char) - ord('a')
        elif 'A' <= char <= 'Z':
            return ord(char) - ord('A') + 26
        else:
            raise ValueError("Invalid character")

    def insert_word(self, word: str) -> None:
        if self.ignore_case:
            word = word.lower()

        cur = self.root
        for ch in word:
            idx = Trie._get_index(ch)
            if cur.children[idx] is None:
                cur.children[idx] = TrieNode(self.ignore_case)
            cur = cur.children[idx]
        cur.is_word = True

    def search_word(self, word: str) -> bool:
        if self.ignore_case:
            word = word.lower()

        cur = self.root
        for ch in word:
            idx = Trie._get_index(ch)
            if cur.children[idx] is None:
                return False
            cur = cur.children[idx]
        return cur.is_word



# This is the Trie data structure for our predefined words.
predefined_words_trie = Trie()

# This data structure used for storing the matched words for processing later.
# Problem does not state the complete use case. So, implementation writes the matched
word and it's count to file
matched_words = Counter()

def process_predefined_words(keyword_file: str, ignore_case=True) -> None:
    """
    This function process the keyword_file which contains the pre-defined words, parses
the file and stores the words
    into hash table data structure

    :param keyword_file: This file holds the all the predefined set of words, newline
separated.
    :param ignore_case: Indicates if case-sensitive or not
    :return:
    """
    with open(keyword_file, 'r') as file:
        for line in file:
            line = line.strip()
            word = line.lower() if ignore_case else line
            predefined_words_trie.insert_word(word)



def clean_word(word: str):
    return re.sub(r'[^a-zA-Z]', '', word)
```

```python
def extract_word_from_input(input_file: str, ignore_case=True):
    """
    It processes and parses the input file. Which could be very large. This function
extracts the word yields it to the
    caller.

    :param input_file: Input file which need to processed and matched against the
predefined set of words.
    :param ignore_case: Indicates if case-sensitive or not
    :return:
    """
    with open(input_file, 'r') as in_file:
        for line in in_file:
            words = line.strip().split()
            for word in words:
                word = word.lower() if ignore_case else word
                yield clean_word(word)



def find_matches(input_file: str, keyword_file: str):
    """
    This function finds the matches against the predefined set of words by extracting
words form input file.

    :param input_file: Input file which need to processed and matched against the
predefined set of words.
    :param keyword_file: This file holds the all the predefined set of words, newline
separated.
    :return:
    """
    process_predefined_words(keyword_file)

    for word in extract_word_from_input(input_file):
        if predefined_words_trie.search_word(word):
            matched_words[word] += 1



def main():
    input_file = r"C:\Users\kiran\PycharmProjects\pythonProject\input_file.txt"
    predefined_words_file =
r"C:\Users\kiran\PycharmProjects\pythonProject\predefined_words_file.txt"
    matched_words_output_file =
r"C:\Users\kiran\PycharmProjects\pythonProject\matched_words_output_file.txt"

    find_matches(input_file, predefined_words_file)

    with open(matched_words_output_file, 'w') as out_file:
        for key, val in matched_words.items():
            out_file.write(f"Word: {key} Count: {val}" + "\n")



if __name__ == "__main__":
    main()
```

Input file:

```
I have an apple, and a banana.
Would you like some cherry pie?
Dates are sweet.
Fig! It's good.
```

Predefined words file:

```
apple
banana
cherry
date
elderberry
fig
grape
honeydew
```

Program output file:

```
Word: apple Count: 1
Word: banana Count: 1
Word: cherry Count: 1
Word: fig Count: 1
```

Optimization

Considering the input file of very large size, 20MB, we could process the file in parallel. We could divide the file in chunks of chosen chunk size and let the different threads operate at different chunks of the file and find the matches against the predefined words.

Program for this optimization submitted in the source code file.

Conclusion

Please find the source code files attached with the submission along with test files used for testing which maps to given scale in the problem statement.