Chapter 4

Getting Started with PySpark

Pyspark Algorithms Book[1]

1. Introduction

This chapter

- Introduces a real problem: DNA Base Count
- Provides a complete, end-to-end solution in PySpark
- Uses different reductions to solve the DNA Base Count problem
- Provides performances of using different reductions

Source Code

Complete PySpark programs for this chapter are presented in GitHub Chapter 4[2].

In this chapter, we will learn the most important Spark transformations (mappers and reducers) and then we will examine how to select specific transformations for targeted problems. This chapter shows that for a given problem (such as a DNA Base Count), there are multiple PySpark solutions, but efficiencies of them differ due to the selection of Spark transformations.

2. DNA Base Count Example

The purpose of this example is to count DNA Bases. Don't worry—you don't need to be an expert in biology and genomics to understand this example.

[1] https://github.com/mahmoudparsian/pyspark-algorithms
[2] https://github.com/mahmoudparsian/pyspark-algorithms/tree/master/code/chap04

What is DNA Base Counting? Given some string DNA containing the four letters `{A, T, C, G}`, representing the bases that make up DNA, the question is how many times does a certain base letter occur in the DNA string? For example, if a DNA string is `"AAATGGCATTA"` and we ask how many times the base `A` occurs in this string, the answer is `5`; if we ask how many times the base `T` occurs in this string, the answer is `3`. Therefore, we want to count the number of each base character, ignoring cases.

For this problem, I provide three distinct solutions using a set of very powerful and efficient Spark transformations such as `map()`, `flatMap()`, `filter()`, `reduceByKey()`, `groupByKey()`, and `mapPartitions()`. Even though all solutions generate the same results, their performances will be different due to the selection of different Spark transformations.

Figure 4.1 illustrates the process of solving DNA Base Count using Spark. To solve this problem, we

1. Write a driver program in Python, using the PySpark API (which is a series of Spark's transformations and actions)
2. Submit the program to a Spark cluster (explained in the following sections).

All the solutions will read input (FASTA files format — to be defined shortly) and produce a dictionary, where the key is a DNA letter and the value is the associated frequency.

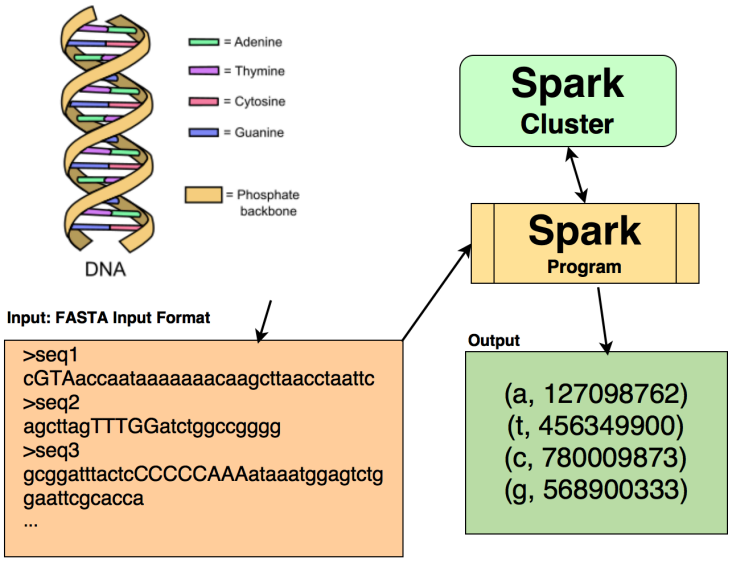

Figure 1. DNA Base Count

I will provide 3 different and distinct PySpark solutions (labeled as Versions 1, 2, and 3) using variety of Spark's transformations and actions. These 3 solutions will show that we have choices in selecting Spark transformations from solving our DNA Base Count problem and each transformation has a different performance. The complete PySpark solutions are provided as Table 4.1. By the end of this chapter you will learn that there are many options in selecting Spark's transformations and actions, but only your data and selected transformations will give you an insight to the performance of your data solution. This means that you should test your solution using real production data — **not just toy data** — before deploying your solution to the production environment. In the solution Version-3, I will introduce a very powerful Spark transformation called `mapPartitions()`, which is an ideal for reducing large volume of data into small amount of desired information (such as frequencies of DNA letters, which is a small hash map table).

Table 1. Solutions for DNA Base Count

Version	**Version-1**	**Version-2**	**Version-3**
Program	`dna_bc_ver_1.py`	`dna_bc_ver_2.py`	`dna_bc_ver_3.py`
Design Pattern	*Basic MapReduce*	*In Mapper Combiner*	*Mapping Partitions*

Version	Version-1	Version-2	Version-3
Transformations	1. `textFile()`	1. `textFile()`	1. `textFile()`
	2. `flatMap()`	2. `flatMap()`	2. `mapPartitions()`
	3. `reduceBykey()`	3. `reduceBykey()`	3. `reduceBykey()`

Performance of these 3 programs using my MacBook (with 16 GB RAM, 2.3 GHZ Intel Processor, 500 GB hard disk) are very different. Note that for these 3 programs, I just used the default parameters (no optimization is done for any solution). The performances are presented (used default parameters with `$SPARK_HOME/bin/park-submit` command) in Table 2.2.

Table 2. Performance for DNA Base Count

Input Data (in bytes)	Version-1	Version-2	Version-3
253,935,557	72 seconds	27 seconds	18 seconds
1,095,573,358	258 seconds	79 seconds	57 seconds

What does this basic performance table tell you? When you write your PySpark application, you have a lot of choices in selecting different transformations and actions. What are the rules for picking the "Right Transformations and Actions"? Only your data and Spark programs can answer this question. In general, when you try to write a PySpark application, you can usually choose from many arrangements of transformations and actions that will produce the same results. However, not all these arrangements of transformations and actions will result in the same performance: avoiding common pitfalls and picking the right arrangement can make a world of difference in an application's performance. We will touch on this subject in chapter 6: "Reductions in Spark". For example, for a large set of (key, value) pairs, `reduceByKey()` (or `combineByKey()`) is more efficient than using combination of `groupByKey()` and `mapValues()`, because `reduceByKey()` (or `combineByKey()`) reduces the shuffling time. For example, if your RDD (represented by variable `rdd`) is an `RDD[(String, Integer)]` (an RDD, where each element is a pair of (`key-as-String`, `value-as-Integer`)) then the following:

```
rdd.groupByKey()
   .mapValues(lambda values : sum(values))
```

will produce the same results as this:

```
rdd.reduceByKey(lambda x,y: x+y)
```

However, the `groupByKey()` will transfer the entire dataset across the cluster network (this is a performance penalty), while the `reduceByKey()` will compute local sums for each key (performance improvement) in each partition and combine those local sums into larger sums after shuffling. Therefore, `reduceByKey()` will transfer much less data than the `groupByKey()` across the cluster network. In most of the situations, `reduceByKey()` will out perform combination of `groupByKey()` and `mapValues()` transformations.

3. DNA Base Count Problem

What is the DNA Base Count Problem? According to the your genome[3]: "DNA's code is written in only four letters, called A, C, T and G. The meaning of this code lies in the sequence of the letters A, T, C and G in the same way that the meaning of a word lies in the sequence of alphabet letters. Your cells read the DNA sequence to make chemicals that your body needs to survive."

The goal of this example is to find frequencies (or percentages) of A, T, C, G, and N (anything other than A, T, C, or G) in a given set of DNA sequences. What does ATCG stand for when talking about DNA? It does refer to 4 of the nitrogenous bases associated with DNA (Table 2.3).

Table 3. DNA Base Letters

DNA Letter	Name
A	Adenine
T	Thymine
C	Cytosine
G	Guanine

[3] https://www.yourgenome.org/

DNA Letter	Name
N	any thing other than ATCG

For example, "ACGGGTACGAAT" is a very small DNA sequence. DNA sequences can be huge and can have uppercase and lowercase letters. For consistency, we will convert all letters to lowercase. For example, human genome consists of three billion DNA base pairs, while the diploid genome (found in somatic cells) has twice the DNA content. The goal of DNA base counting for our example is to generate the data in Table 2.4 (frequencies for each DNA base).

Table 4. DNA Base Count Example

Base	Count
a	4
t	2
c	2
g	4
n	0
z	1 (the total number of DNA Sequences)

4. FASTA Format

DNA sequences can be represented in many different formats including FASTA and FASTQ popular text-based formats (input is given as a text file). Our solution will only handle FASTA format since it is much easier to read FASTA files. Both FASTA and FASTQ formats store sequence data, and sequence metadata. With some minor modifications to presented solutions (modifications required to reading data portion), you can run the revised solutions for the FASTQ format.

A sequence file in FASTA format can contain several sequences. Each sequence in FASTA format begins with a single-line description, followed by one or many lines of sequence data. The description line must begin with a "greater-than" (">") symbol in the first column. Note that the description line may be used for counting the number of sequences and does not have any DNA sequence data.

4.1. FASTA Format Example

An example sequence in FASTA format is presented by the `sample.fasta` file. This FASTA file has has 4 DNA sequences and case (upper-case or lower-case) of characters are irrelevant, illustrated below, which will be used as a test case for our PySpark programs.

```
cat sample.fasta
>seq1
cGTAaccaataaaaaaacaagcttaacctaattc
>seq2
agcttagTTTGGatctggccgggg
>seq3
gcggatttactcCCCCCAAAAANNaggggagagcccagataaatggagtctgtgcgtccaca
gaattcgcacca
AATAAAACCTCACCCAT
agagcccagaatttactcCCC
>seq4
gcggatttactcaggggagagcccagGGataaatggagtctgtgcgtccaca
gaattcgcacca
```

4.2. FASTA Data Download

To test the DNA Base Count programs (provided in this chapter), you may download FASTA data from the University of California[4]. As of writing this book, GitHub does not permit files greater than 100MB. Some sample FASTA files are provided in this book's GIT repository[5].

5. DNA Base Count by PySpark — Version 1

This version provides a very basic solution for the DNA Base Count problem. This is the high-level solution:

1. Read FASTA input data and create an `RDD[String]`, where each RDD element is a FASTA record (it can be either a comment line or an actual DNA sequence)

[4] http://hgdownload.cse.ucsc.edu/
[5] https://github.com/mahmoudparsian/pyspark-algorithms/tree/master/code/chap04

2. Define a mapper function: for every DNA letter in a FASTA record, emit a pair of `(dna_letter, 1)` where `dna_letter` is in `{A, T, C, G}` and `1` is a frequency (similar to a wordcount solution)
3. Sum up frequencies for all DNA letters (this is a reduction step). For each unique `dna_letter`, group and add all frequencies.

For testing this solution, I will use the provided sample FASTA file (`sample.fasta` as an input) for running our PySpark solution.

The high level workflow for solution Version 1 is presented by the Figure 4.2.

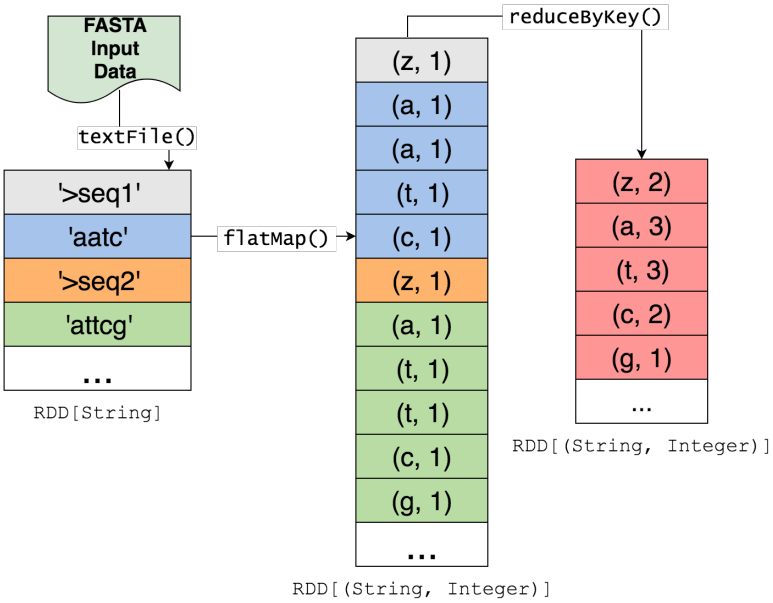

Figure 2. DNA Base Count: Solution Version 1

5.1. Step 1: Create an RDD of String from Input

The `SparkContext.textFile()` function is used to create an `RDD[String]` for input in FASTA text-based format. The `textFile()` can be used to read a text file from HDFS, Amazon S3, a local file system (available on all Spark nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings. Let "spark" be an instance of the `SparkSession` class, then to create FASTA records RDD (as denoted by `recordsRDD`), we have at least two options:

Option 1: Use SparkSession

```
>>># spark : instance of SparkSession
>>> input_path = "/pyspark_book/code/chap04/sample.fasta"  ❶
>>> recordsRDD = spark.read
                      .text(input_path)
                      .rdd.map(lambda r: r[0])  ❷
```

❶ Define input path

❷ Use DataFrameReader (`spark.read`) to create a DataFrame and then convert it to an `RDD[String]`

Option 2: Use SparkContext

```
>>> input_path = "/pyspark_book/code/chap04/sample.fasta"  ❶
>>># Let 'spark' be an instance of SparkSession
>>># access SparkContext from a SparkSession
>>> sc = spark.sparkContext  ❷
>>> recordsRDD = sc.textFile(input_path)  ❸
```

❶ Define input path

❷ Create an instance of a SparkContext (as `sc`)

❸ Use SparkContext to read input and create an `RDD[String]`

To read a text file and convert it to an RDD, Option 2 is preferable, because it easy and efficient. Option 1 works too, but first it creates a DataFrame, and then converts it to an RDD and eventually performs another mapper transformation.

Next we examine the content of the created RDD: each RDD element (as a String) is denoted by `u'<element>'`. The `RDD.collect()` is used to get the content as a list of String objects and display it. For large RDDs, you should not use `collect()`, which might cause OOM problem as well as a performance penalty. To just view the first N elements, you may use `RDD.take(N)`.

Next we examine the whole content of created RDD (since input file is small enough — for large RDDs, we should avoid the `collect()`):

```
>>> recordsRDD.collect()
[
 u'>seq1',
 u'cGTAaccaataaaaaaacaagcttaacctaattc',
 u'>seq2',
```

```
 u'agcttagTTTGGatctggccgggg',
 u'>seq3',
 u'gcggatttactcCCCCCAAAAANNaggggagagcccagataaatggagtctgtgcgtccaca',
 u'gaattcgcacca',
 u'AATAAAACCTCACCCAT',
 u'agagcccagaatttactcCCC',
 u'>seq4',
 u'gcggatttactcaggggagagcccagGGataaatggagtctgtgcgtccaca',
 u'gaattcgcacca'
]
```

5.2. Step 2: Define a Mapper Function

To map RDD elements into a set of pairs `(dna_letter, 1)`, we define a Python function, which is passed to the `flatMap()` transformation: `flatMap()` is a **"1-to-many"** transformation: returns a new RDD by first applying a function to all elements of this RDD, and then flattening the results. For, example if your Python fuction, which is passed to the `flatMap()` transformation, returns a list as `[`v_1`,` v_2`,` v_3`]`, then that will be flattened into 3 target RDD elements as v_1, v_2, and v_3.

To process a source RDD element, we define a function, `process_FASTA_record`, which accepts an RDD element (a single record of FASTA file as a String), and returns a list of pairs as `(dna_letter, 1)`. For example, if your input record to the `flatMap()` is "AATTG", then it will emit the following (key, value) pairs (note that all DNA letters are converted to lower case — input is mixed of uppercase and lowercase characters):

```
    ('a', 1)
    ('a', 1)
    ('t', 1)
    ('t', 1)
    ('g', 1)
```

Next, we define the `process_FASTA_record()` function, (denoted by Listing 4.1) which accepts a string (an element of source RDD) and returns a list of (key, value) pairs. If the input is a "description record" (no sequence data, which begins with `>seq`), then we emit ("z", 1). The key `z` is used to find the total number of sequences for a given input (the letter `z` is not a DNA letter and may be used to count the total number of DNA sequences). This will enable us to find the number of sequences as well. If the input is a DNA sequence, then we tokenize

it by characters and then for each DNA letter (denoted by `dna_letter`), we emit `(dna_letter, 1)`, and finally return list of these pairs. I have included some `print` statements for debugging purposes. For production environments, these `print` statements should be removed (they will cause performance penalties).

The process_FASTA_record() Function.

```
# Parameter : fasta_record : String,
#             a single FASTA record
#
# output: a list of (key, value) pairs,
#         where key is a DNA-Letter and
#         value is a frequency
#
def process_FASTA_record(fasta_record):
    key_value_list = [] ❶
    #
    if (fasta_record.startswith(">")):
        # z counts the number of FASTA sequences
        key_value_list.append(("z", 1)) ❷
    else:
        chars = fasta_record.lower()
        for c in chars:
            key_value_list.append((c, 1)) ❸
    #
    print(key_value_list) ❹
    return key_value_list ❺
#end-def
```

❶ Create an empty list, we will keep adding (key, value) pairs (this is our output from this function)

❷ Append `("z", 1)` to list

❸ Append `(c, 1)` to list, where `c` is a DNA-letter

❹ For debugging purposes only

❺ Return a list of (key, value) pairs, which will be flattened by the `flatMap()`

Now, we will use this function to apply the `flatMap()` transformation to the `recordsRDD` (`RDD[String]`) created above:

```
>>># rec refers to an element of recordsRDD
>>># Lambda is a notation which defines input and output
```

```
>>>#   input: "rec" as an recordsRDD element  ❶
>>>#   output: result of process_FASTA_record(rec)
>>> pairsRDD = recordsRDD.flatMap(lambda rec: process_FASTA_record(rec))
  ❷
```

❶ Source RDD (`recordsRDD`) is an `RDD[String]`

❷ We use a lambda expression, where `rec` denotes a single element of `recordsRDD`. The target RDD (`pairsRDD`) is an `RDD[(String, Integer)]`

or we may write its equivalent as (without using lambda expressions):

```
>>> pairsRDD = recordsRDD.flatMap(process_FASTA_record)
```

For example, if an element of `recordsRDD` contains DNA sequence as 'gaattcg', then it will be flattened into the following (key, value) pairs:

```
    (g, 1)
    (a, 1)
    (a, 1)
    (t, 1)
    (t, 1)
    (c, 1)
    (g, 1)
```

and if an element of `recordsRDD` contains '>seq', then it will be flattened into the following (key, value) pair (we use the key `z` to represent total number of DNA sequences for a given input):

```
    (z, 1)
```

5.3. Step 3: Find Frequencies of DNA Letters

Now, the `pairsRDD` contains a set of `(key, value)` pairs where `key` is a DNA letter and `value` is a frequency of that letter as 1. Next, we apply the `reduceByKey()` transformation to the `pairsRDD` to find the aggregated frequencies for all DNA letters.

The `reduceByKey()` transformation merges the values for each unique key using an associative and commutative reduce function (we use addition as our reduction

function). Therefore, we can now see that we are simply taking an **accumulated value** for the given key and summing it with the **next value** of that key. In the simplest form, let key `K` have five pairs in the RDD: `(K, 2)`, (K, 3), `(K, 6)`, `(K, 7)`, and `(K, 8)`, then the `reduceByKey()` transformation will transform these five pairs into a single pair as `(K, 26)` where `26=2+3+6+7+8`. For example, if we had 2 partitions for these 5 pairs, then each partition will be processed in parallel and independently:

- Processing Partition-1:

```
Partition-1: {
               (K, 2),
               (K, 3)
             }

   (K, 2), (K, 3) => (K, 2+3) = (K, 5)
   Result of Partition-1: (K, 5)
```

- Processing Partition-2:

```
Partition-2: {
               (K, 6),
               (K, 7),
               (K, 8)
             }

   (K, 6), (K, 7)  => (K, 6+7) = (K, 13)
   (K, 8), (K, 13) => (K, 8+13) = (K, 21)
   Result of Partition-2: (K, 21)
```

- Next, the partitions are merged:

```
Merge Partitions:
  => Partition-1, Partition-2
  => (K,5), (K, 21)
  => (K, 5+21) = (K, 26)

  Final result: (K, 26)
```

For viewing the final result, you may use different actions to get the job done. Below, I use `collect()` and `collectAsMap()` to view the final result. If you

want to persist your RDD (as a final result) on a disk, then you may use the `RDD.saveAsTextFile(path)` where `path` is your output directory name.

Next, I present the final reduction for DNA base count:

```
# x and y refers to the frequencies of the same key
# source: pairsRDD : RDD[(String, Integer)]
# target: frequenciesRDD : RDD[(String, Integer)]
frequenciesRDD = pairsRDD.reduceByKey(lambda x, y: x+y)
```

Note that the source and target data types for `reduceByKey()` are the same. That is if source RDD is `RDD[(K, V)]` then the target RDD will be the same as `RDD[(K, V)]`. Later we will see that the `combineByKey()` transformation does not have this restriction.

There are several ways that you can view the final output. For example, you may use the `RDD.collect()` function to get the final RDD's elements as a list of pairs:

```
frequenciesRDD.collect()
[
  (u'a', 73),
  (u'c', 61),
  (u't', 45),
  (u'g', 53),
  (u'n', 2),
  (u'z', 4)
]
```

Also, we may use the `RDD.collectAsMap()` action to return the result as a hash map:

```
>>> frequenciesRDD.collectAsMap()
{
  u'a': 73,
  u'c': 61,
  u't': 45,
  u'g': 53,
  u'n': 2,
  u'z': 4
}
>>>
```

We may use other Spark transformations to aggregate frequencies of DNA letters. For example, we may group frequencies (using `groupBykey()` transformation) of a DNA letter and then add all frequencies together. This solution is a less efficient than the one presented by the `reduceByKey()` transformation.

```
groupedRDD = pairsRDD.groupByKey()  ❶
frequenciesRDD = groupedRDD.mapValues(lambda values : sum(values)) ❷
frequenciesRDD.collect()
```

❶ `groupedRDD` : `RDD[(String, [Integer])]`, where key is a String and value is a list/iterable of Integers (as frequencies)

❷ `frequenciesRDD` : `RDD[(String, Integer)]`

For example, if `pairsRDD` contains `4` pairs of `('z', 1)`, then the `groupedRDD` will have a single pair of `('z', [1, 1, 1, 1])`. That is it groups values for the same key. While both of these transformations (`reduceByKey()` and `groupByKey()`) produce the correct answer, the `reduceByKey()` solution works much better on a large FASTA dataset. That's because Spark knows it can combine output with a common key (DNA Letter) on each partition before shuffling the data. It is recommended that we should "Avoid groupByKey()" and use `reduceByKey()` and `combineByKey()` whenever possible. In other words, we can say that `reduceByKey()` and `combineByKey()` scale-out better than `groupByKey()`.

5.4. Pros and Cons of Version 1

Pros:

- The provided solution works and is simple: uses minimal amount of code to get the job done and uses the Spark's simple `map()` and `reduceByKey()` transformations.
- There is no scalability issue since we use `reduceByKey()` for reducing all (key, value) pairs, which will automatically perform the `combine()` optimization (local aggregation) on all worker nodes.

Cons:

- This solution emits too many `(key, value)` pairs, where key is a DNA-letter and value is 1, as frequency. Sometimes, emitting too many (key, value) pairs might cause memory problems. If you get any error due to too many `(key,`

value) pairs, then you might adjust the RDD's StorageLevel. By default, Spark uses MEMORY, but you can set the StorageLevel to MEMORY and DISK combinations for that RDD.

- Performance is not an optimal since emitting too many (key, value) pairs will take network time and prolong the shuffle time. As during the second processing step we defined, too many single frequency tuples are emitted, network time will prove a bottleneck when scaling this solution.

6. DNA Base Count by PySpark — Version 2

Version 2 is an improved solution of Version 1. In version 1, we emitted pairs of (dna_letter, 1) for each DNA letter in given sequences. In solution Version 2, since some of the FASTA sequences can be very long, instead of emitting (dna_letter, 1) per DNA letter, we aggregate them into a hash map (dictionary concept in Python) and then flatten the hash map into a list and finally do the frequencies aggregation. For example, if a given FASTA sequence record is "aaatttcggggaa", then 2nd column (Version 2) will be emitted instead of 1st column (Version 1).

Table 5. Emitted (key, value) pairs

Version 1	Version 2
(a, 1)	(a, 5)
(a, 1)	(t, 3)
(a, 1)	(c, 1)
(t, 1)	(g, 4)
(t, 1)	
(t, 1)	
(c, 1)	
(g, 1)	
(g, 1)	
(g, 1)	
(g, 1)	
(a, 1)	

Version 1	Version 2
`(a, 1)`	

The advantage of solution Version 2 is that we emit much fewer (key, value) pairs which will ease up the cluster network traffic. Meanwhile, we do an "InMapper Combiner" (IMC) optimization. IMC is an approach to possibly improve the speed of a MapReduce job by reducing the number of intermediary (key, value) pairs emitted from mappers to reducers. For a given DNA string, in the mapper step, by aggregating the frequencies for the same DNA letter, we will emit much fewer (key, value) pairs, which can improve the cluster network traffic and hence can improve the overall performance of your program.

Therefore, our solution for Version 2 is presented as:

1. Read FASTA input data and create an `RDD[String]`, where each RDD element is a FASTA record. This step is the same as Version 1.
2. Mapper step: for every FASTA record, create a `HashMap[Key, Value]` (a dictionary or hash table) where Key is the DNA letter and Value is an aggregated frequency for that DNA letter. Then flatten (using Spark's `flatMap()`) the hash map into a list of (Key, Value) pairs. This step is different from Version 1. Compared with Version 1, this step enable us to emit less (key, value) pairs
3. Find frequencies for all DNA letters by aggregating frequencies of the same DNA letter (this is a reduction step). For each dna_letter, group and add all frequencies. This step is the same as Version 1.

The high level workflow for solution Version 1 is presented by the Figure 4.3.

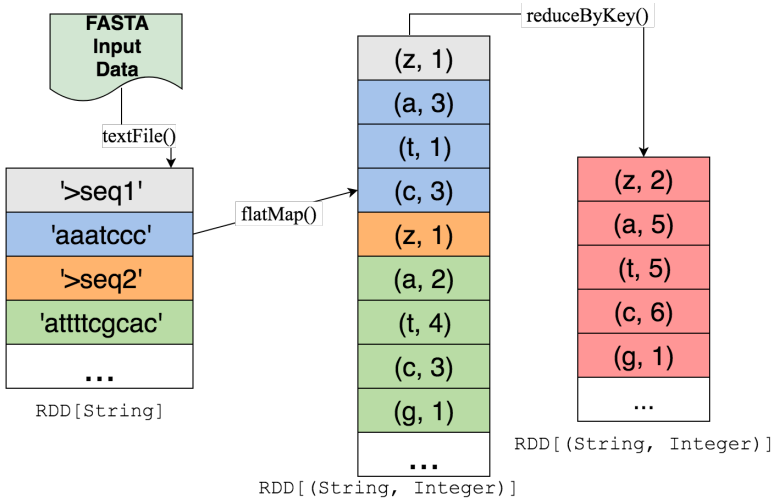

Figure 3. DNA Base Count: Solution Version 2

6.1. Step-1: Create an *RDD[String]* from Input

The `SparkContext.textFile()` function is used to create an RDD for input in FASTA text-based format. Let "spark" be a SparSession object.

```
>>># spark : an instance of SparkSession
>>># then access sparkContext from a SparkSession
>>> input_path = "/pyspark_book/code/chap04/sample.fasta"
>>> recordsRDD = spark.sparkContext.textFile(input_path) ❶
```

❶ `recordsRDD : RDD[String]`

6.2. Step 2: Define a Mapper Function

Map each RDD element (which represents a single FASTA record as a String) into a list of (key, value) pairs, where key is a unique DNA letter and value is an aggregated frequency for the entire record. We define a Python function, which is passed to the `flatMap()` transformation: `flatMap()` is a "1-to-many" transformation: return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

To process an RDD element, we define a Python function, `process_FASTA_as_hashmap` (Listing 4.2), which accepts an RDD element (as

a String), and returns a list of `(dna_letter, frequency)`. For debugging and teaching purposes, I have included some `print` statements, which should be removed for production environments.

The process_FASTA_as_hashmap() Function.

```
# Parameter: fasta_record : String,
#            a single FASTA record
#
# output: a list of (DNA-Letter, frequency)
#
def process_FASTA_as_hashmap(fasta_record):
    if (fasta_record.startswith(">")):  ❶
        return [("z", 1)]

    hashmap = defaultdict(int)  ❷
    chars = fasta_record.lower()
    for c in chars:  ❸
        hashmap[c] += 1
    #end-for
    print("hashmap=", hashmap)

    key_value_list = [(k, v) for k, v in hashmap.iteritems()]  ❹
    print("key_value_list=", key_value_list)
    return key_value_list  ❺
#end-def
```

❶ It is a comment line for a DNA sequence

❷ Create a dictionary[String, Integer]

❸ Aggregate DNA-letters

❹ Flatten dictionary into a list of (DNA-Letter, frequency)

❺ Return the flattened list of (DNA-Letter, frequency)

Now, we will use this Python function, `process_FASTA_as_hashmap()`, to apply the `flatMap()` transformation to the `recordsRDD` (as `RDD[String]`) created above:

```
>>># source: recordsRDD (as RDD[String])
>>># target: pairsRDD (as RDD[(String, Integer)])
>>> pairsRDD = recordsRDD.flatMap(lambda rec:
 process_FASTA_as_hashmap(rec))
```

or we may write this as:

```
>>># source: recordsRDD (as RDD[String])
>>># target: pairsRDD (as RDD[(String, Integer)])
>>> pairsRDD = recordsRDD.flatMap(process_FASTA_as_hashmap)
```

For example, if `recordsRDD` element contains `'gggggaaattccccg'`, then it will be flattened into the following (key, value) pairs:

```
    (g, 6)
    (a, 3)
    (t, 2)
    (c, 4)
```

To count the total number of DNA Sequences, if `recordsRDD` element begins with "`>seq`", then it will be flattened into the following single (key, value) pair:

```
    (z, 1)
```

6.3. Step 3: Find Frequencies of DNA Letters

Now, `pairsRDD` contains (key, value) pairs where key is a `dna_letter` and value is a frequency of that letter. Next, we apply the `reduceByKey()` transformation to the pairsRDD to find the aggregated frequencies for all DNA letters. Note that the key `'z'` is used to count the total number of DNA sequences and `'n'` is the key other than DNA bases.

```
# x and y refers to the frequencies of the same key
frequenciesRDD = pairsRDD.reduceByKey(lambda x, y: x+y) ❶
frequenciesRDD.collect() ❷
[
  (u'a', 73),
  (u'c', 61),
  (u't', 45),
  (u'g', 53),
  (u'n', 2),
  (u'z', 4)
]
```

❶ `pairsRDD` (as `RDD[(String, Integer)]`)

❷ `frequenciesRDD` (as `RDD[(String, Integer)]`)

Also, we may use the `collectAsMap()` action to return the result as a hash map:

```
>>> frequenciesRDD.collectAsMap()
{
  u'a': 73,
  u'c': 61,
  u't': 45,
  u'g': 53,
  u'n': 2,
  u'z': 4
}
>>>
```

Pros and Cons of Version 2

Pros:

- The provided solution works, simple, and semi-efficient. This solution improves on Version 1, by emitting much less (key, value) pairs, since we create a dictionary per input record and then flatten it into a list of (key, value) pairs, where key is a DNA-letter and value is an associated aggregated frequency of the DNA-letter.
- Network traffic is improved by emitting much fewer (key, value) pairs.
- There is no scalability issue since we use `reduceByKey()` for reducing all (key, value) pairs

Cons:

- For each DNA sequence, this solution emits up to 6 (key, value) pairs, where key is a DNA-letter and value is 1, frequency. This is a much improvement over solution version 1
- Performance is not an optimal since we are still emitting about 6 (key, value) pairs per DNA string
- This solution might be using too much memory due to creation of a dictionary per DNA sequence

7. DNA Base Count by PySpark — Version 3

This solution improves on Versions 1 and 2 and is an optimal solution with no scalability problem at all. Here we solve the DNA Base Count problem by a very powerful and efficient Spark transformation called `mapPartitions()`. Before presenting a solution for Version 3, I will deep dive into understanding of the `mapPartitions()` transformation.

7.1. Understanding `mapPartitions()`

The `mapPartitions()` transformation is defined as:

```
pyspark.RDD.mapPartitions(f, preservesPartitioning=False)

mapPartitions() is a method in the pyspark.RDD class.

Description:

    Return a new RDD (called target RDD)
    by  applying a function f() to  each
    partition of the source RDD.

    Input to f() is an iterator (of type T),
    which represents a single partition of
    the source RDD. Function f() returns an
    object of type U.

    f: Iterator<T> --> U ❶

    mapPartitions : RDD[T] -- > RDD[U] ❷

    Source RDD: RDD[T]
    Target RDD: RDD[U]
```

❶ The function `f()` accepts a pointer to a single partition (as an `iterator` of type `T`) and returns an object of type `U`; `T` and `U` can be any data types and they do not have to be the same

❷ Transform an `RDD[T]` to `RDD[U]`

To understand the semantics of the `mapPartitions()` transformation, first, we should understand the concept of a "partition" and partitioning in Spark. Informally, using Spark's terminology, input data (DNA sequences in FASTA format) is

represented as an RDD. To understand partitioning and `mapPartitions()` transformation, as an example say that we have 6,000 million records — about 6 billion records). Then the Spark partitioner partitions input data into 3,000 chunks/partitions and sends it to a mapper transformations. So, each partition will have roughly 2 million records — therefore each partition is processed by a single `mapPartitions()` transformation. If you divide 6 billion records into 3,000 partitions, then each partition will be about 2 million records. Therefore, a function `f()`, which is used in the `mapPartitions()` transformation, will accept an iterator (as an argument) to handle one partition — 2 million records. To make this transformation efficient, we will use a single dictionary per partition to aggregate DNA letters and its associated frequencies.

In solution Version 3, we will create one dictionary per partition rather than a dictionary per FASTA record. This is a huge improvement over solution of versions 1 and 2. Creation of 3,000 hash tables in a cluster uses very little memory compared in creating a dictionary per input record. This solution is a scale-out solution, since creating and aggregating 3,000 dictionaries takes minimal amount of memory and it is fast due to concurrent and independent processing of all partitions in the cluster.

So, what is the main difference between `map()` and `mapPartitions()` transformations? In a nutshell, `map()` is a **"1-to-1"** transformation: maps each element of the source RDD into a single element of target RDD. While `mapPartitions()` can be considered as a **"Many-to-1"** transformation: maps each partition (comprised of many elemets of the source RDD — each partition may have thousands or millions of elements) of the source RDD into a single element of target RDD.

The `map()` transformation converts each element of the source RDD into a single element of the target RDD by applying a mapper function. Let `func` be a user provided function, then `mapPartitions(func)` converts each partition (can be comprised of thousands or millions of RDD elements) of the source RDD into multiple elements of the result (possibly none) by applying the the function `func` to each partition. Therefore, the `mapPartitions(func)` returns a new RDD by applying a function to each partition of this input RDD. The `mapPartitions()` transformation is a map operation over partitions and not over the elements of the partition (the `func` receives an iterator, which you can iterate over elements of a partition). The `mapPartitions()` transformation is called once for each input

RDD partition unlike `map()` and `foreach()` which is called for each element in the RDD. The main advantage being that, we can do initialization on per-partition basis instead of per-element basis (as done by `map()` & `foreach()`).

The `mapPartitions()` transformation semantics is illustrated by the Figure 4.2.

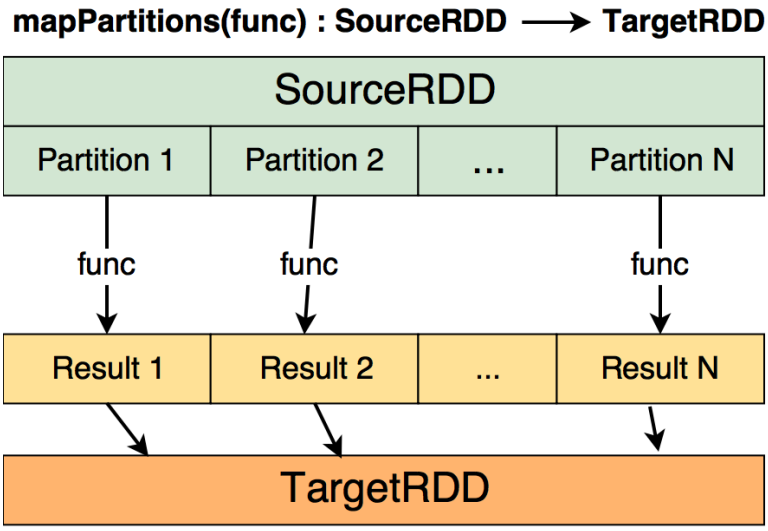

Figure 4. The mapPartitions() Transformation

The `mapPartitions()` transformation semantics for the DNA Base Count is illustrated by the Figure 4.2.

Let's walk through Figure 4.2:

- The SourceRDD represents all of our input as `RDD[String]`, since each record of FASTA file is a String object
- The entire input is partitioned into N chunks/partitions, where each chunk/partition may have thousands or million of DNA sequences (each DNA sequence is a record of String data type). Partitioning of source RDD into partitions is similar to the Linux's `split` command, which splits a file into pieces
- Each partition is sent to a `mapPartitions()` mapper/worker/executor to be processed by your provided `func()`. Your `func()` accepts a partition (as an iterator of `String` type) and returns at most 6 (key, value) pairs, where key is a DNA-letter and value is the total frequency of that letter for that partition. Note that partitions are processed in parallel and independently

- Once processing all partitions are completed, the results are merged into the target RDD (depicted as TargetRDD), which is an `RDD[(String, Integer)]`, where the key is a DNA-letter and value is the frequency of the DNA-letter

The detailed `mapPartitions()` transformation semantics for the DNA Base Count is illustrated by the Figure 4.3.

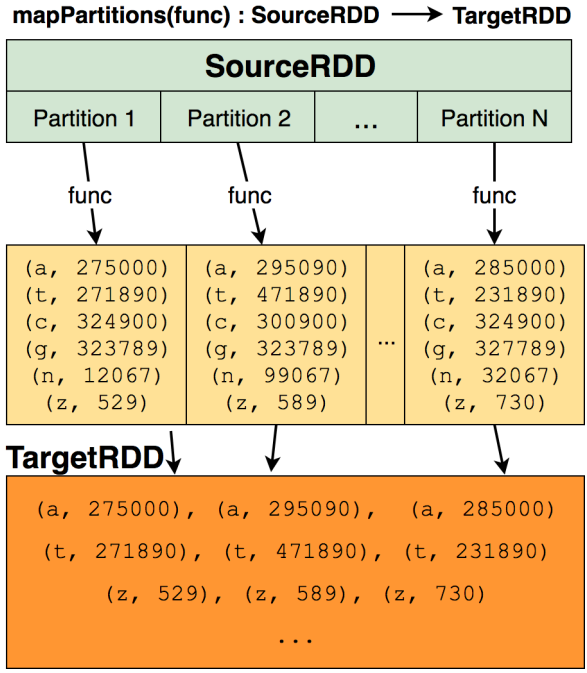

Figure 5. Detailed mapPartitions() Transformation

How does the `mapPartitions()` work? The Figure 4.3 shows that our input (FASTA format data) has been partitioned into `N` chunks/partitions where each partition can be handled by a mapper/worker/executor independently and in parallel. For example, if our input has total of `5 billion` records and `N = 50,000`, then each partition will get about `100,000` FASTA records (`5 billion = 50,000 * 100,000`). Therefore, each `func()` will process (by means of iteration) about `100,000` FASTA records. Each partition will emit at most 6 `(key, value)` pairs, where keys will be in `{"a", "t", "c", "g", "n", "z"}`.

According to Spark API: the `mapPartitions(func)` transformation is similar to `map()`, but runs separately on each partition (block) of the RDD rather than on elements of an RDD, so `func` must be of `iterator` type:

```
source: RDD[T] ❶
#
# Parameter p : iterator<T> ❷
func(p): ❸
   u = < create object of type U
        by iterating all elements
        of a single partition
        denoted by p >
   return u ❹
#end-def
#
target = source.mapPartitions(func) ❺
#
target:  RDD[U] ❻
```

❶ Each element of source RDD is type of `T`

❷ Parameter `p` is an `iterator<T>`, which represent a single partition

❸ Each iteration will return an object of type `T`

❹ Define a `func()`, which accepts a single partition as an `iterator<T>` (an iterator of type `T` — over a single partition of the source `RDD[T]`) and returns an object of type of `U`

❺ Apply the transformation

❻ The result is an `RDD[U]`, where each partition was converted (using `func()`) into a single object type of `U`

Let's assume that we have source `RDD[T]`. Therefore, for our example, `T` represents a String type (which represents a DNA sequence record) and `U` represents a hash table (a.k.a dictionary in Python) as `HashMap[String, Integer]`, where key is a DNA-letter (as a String object) and value is the associated frequency (as an Integer).

We can define "func" (as a generic template) in Python as denoted by the Listing 4.3.

Template for handling a partition.

```
# Parameter: iterator, which represents
#            a single partition
#
# Note that iterator is a parameter
```

```
# from the mapPartitions() transformation,
# through which we can iterate through all
# the elements in a single Partition.
#
# source : RDD[T]
# target : RDD[U]
#
def func(iterator): ❶
    1. make sure that iterator is not empty,
       if it is empty, then handle it properly,
       you can not ignore empty partitions

    2. initialize your desired "data structures DS"
       (such as dictionaries and lists)

    3. iterate for all records in a given partition
    for record in iterator: ❷
        3.1 process(record)
        3.2 update your "data structure DS"
    end-for

    4. if required, post process your "data structures DS"
    result_for_single_partition = post_process(DS) ❸

    5. return result_for_single_partition
#end-def
```

❶ `iterator` is a pointer to a single partition, which you can iterate on elements of a partition

❷ `record` is data type of `T`

❸ `result_for_single_partition` is a data type of `U`

When should we use the `mapPartitions()`? The `mapPartitions()` transformation should be used when you want to extract some condensed or minimal information or small amount of information (such as finding the minimum and maximum of numbers, top-10 URLs, and such as finding count of DNA bases) from each partition, where each partition is a large set of data. For example, if you want to find the minimum and maximum of all numbers in your input, then using `map()` can be pretty inefficient, since you will be generating tons of intermediate (K,V) pairs, but the bottom line is you just want to find two numbers: the minimum and maximum of all numbers in your input. Therefore, using `mapPartitions()` is the best choice to solve DNA Base Count problem.

The `mapPartitions()` can be used for other MapReduce design patterns. Suppose you want to find top-10 (or bottom-10) for your input, then `mapPartitions()` can work very well: find the top-10 (or bottom-10) per partition, then find the top-10 (or bottom-10) for all partitions: this way you are limiting emitting too many intermediate (key, value) pairs.

For counting DNA bases, the `mapPartitions()` transformation is an ideal solution: from each partition find frequencies of 4 DNA keys: `{A, T, C, G}`. This solution scales out very well even when your number of partitions is in high thousands. Let's say you partition your input into 100,000 (which is a very high number of partitions — typically the number of partitions will not be this high). Then aggregating 100,000 dictionaries (hash map) is a very trivial and can be accomplished in seconds and there will not be any "out of memory" problems at all and there will not be any scalability problems.

I will mention one more tip about using `mapPartitions()` before presenting a complete DNA Base Count solution using the powerful `mapPartitions()` transformation.

Suppose that you will be accessing a database for some of your data transformations. So, you need a connection to your database. As you know, creating a connection object is expensive and it will take some time (may be a second or two) to create a connection object. If you create a a connection object per source RDD element, then your soultion will not scale out all: you will run out of connections and resources and your solution will fail. Whenever you have heavyweight initialization that (such as creating a database connection object) should be done once for many RDD elements rather than once per RDD element, and if this initialization, such as creation of objects from an external library, cannot be serialized (so that Spark can transmit it across the cluster to the worker nodes), use `mapPartitions()` instead of `map()`. The `mapPartitions()` transformation provides for the **initialization to be done once per worker** task/partition instead of once per RDD data element.

This concept of initialization per partition/worker is presented by the following example:

```
# source_rdd : RDD[T]
# target_rdd : RDD[U]
```

```
target_rdd = source_rdd.mapPartitions(func)

def func(partition): ❶
   # create a heavyweight connection object
   connection = <creates a db connection per partition> ❷

   data_structures = <create and initialize your data structure> ❸

   # iterate all partition elements
   for rdd_element in partition: ❹
      <use  connection and rdd_element to
       make a query to your database>
      <update your data_structures>
   #end-for

   connection.close() # close db connection here ❺

   u = <prepare object of type U from data_structures> ❻
   return u ❼
#end-def
```

❶ The `partition` parameter is an `iterator<T>`, which represents a single partition of `source_rdd`; this `func()` returns an object of type `U`

❷ Create a single `connection` object to be used by all single partition elements

❸ The `data_structures` can be a list or dictionary or your desired data structures

❹ The `rdd_element` is a single element of type T`

❺ Close the `connection` object (to release allocated resources)

❻ Create an object of type `U` from your created `data_structures`

❼ Return a single object of type `U` per partition

7.2. DNA Base Count Solution by `mapPartitions()`

We will use the … `/chap04/sample.fasta` file for testing our PySpark solution Version 3.

The high level workflow for solution Version 3 is presented by the Figure 4.6.

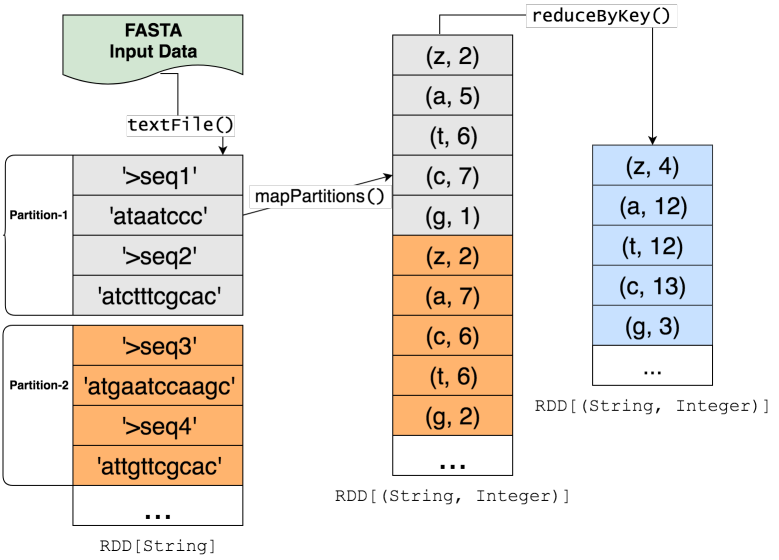

Figure 6. DNA Base Count: Solution Version 3

Some of key points about Figure 4.6:

1. I just show 4 records (2 FASTA sequences) per partition, but in reality, each partition may contain thousands or millions of records. If total of all your input is `N` records and `P` is the number of partitions, then each partition will have about `(N / P)` records.
2. If we have enough resources in our Spark cluster, then each partition can be processed in parallel and independently
3. If you have a lot of data, but your requirement is to extract small amout of information from all data, then `mapPartitions()` might be good choice and will out perform `map()` and `flatMap()` transformations.

Step 1: Create an RDD of String from Input

The `SparkContext.textFile()` function is used to create an RDD for input in FASTA text-based format. This step is identical to the Step 1 of the solution Version 1.

```
input_path = "/pyspark_book/code/chap04/sample.fasta"
>>> records = spark.sparkContext.textFile(input_path)  ❶
```

❶ Create records as an `RDD[String]`

Step 2: Define a Function to Handle a Partition

Let your RDD be an `RDD[T]` (in our example, `T` is a `String`). Spark splits your input data into partitions (where each partition is a set of elements of type `T`) and then executes computations on the partitions independently and in parallel (called divide and conquer model). In using the `mapPartitions()` transformation, the source RDD is partitioned into N partitions (the number of partitions are determined by the size and number of resources available in the Spark cluster) and each partition is passed to a function (this can be a user-defined function). You can control the number of partitions by using `coalesce()` as:

```
RDD.coalesce(numOfPartitions, shuffle=False)
```

which partitions the source RDD into a number of partitions (`numOfPartitions`). If you do not use `RDD.coalesce()` for explicit partitioning, then Spark engine will partition your input based on the cluster configurations and available number of resources in the cluster. For example, this is how, we can partition the RDD: the following example creates an RDD and then partitions it into 3 partitions.

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numOfPartitions = 3
>>> rdd = sc.parallelize(numbers, numOfPartitions) ❶

>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> rdd.getNumPartitions()  ❷
3
```

❶ Create an RDD and set the number of partitions to 3

❷ Check the number of partitions for an RDD

Next, I define a `scan()` function in Python to iterate a given iterator: you may use this function to debug small RDDs and check the partitioning (do not use `scan()` for production environments):

```
>>> def scan(iterator):  ❶
```

```
...     for x in iterator:
...             print(x)
...     print "==="
>>>#end-def
>>> rdd.foreachPartition(scan)  ❷
1
2
3
===
7
8
9
10
===
4
5
6
===
```

❶ Iterate elements of a partition

❷ Apply the scan() function to a given partition

To see the behavior of partitions, let's add the values of each partition be defining an adder() function in Python:

```
>>> def adder(iterator):
...     yield sum(iterator)
...
>>> rdd.mapPartitions(adder).collect()
[6, 15, 34]
```

For DNA Base counting, to handle (i.e., process all elements of a partition) an RDD partition, we define a function, `process_FASTA_partition` (Listing 4.4), which accepts a single partition (represented as an `iterator`). We then iterate on `iterator` to process all elements in the given RDD's partition. For DNA base counting, each partition produces a dictionary, which then we map it into a list of `(dna_letter, frequency)` pairs.

The process_FASTA_partition() Function.

```
#-------------------------------------
# Parameter: iterator
# we get an iterator: which represents a single
```

```
# partition of source RDD, through which we can
# iterate through all of the elements in a Partition.
# In a nutshell, a partition is a set of records.
#
# This function creates a hash map of DNA
# Letters and then flattens it to (key, value) pairs.
#-------------------------------------
from collections import defaultdict

def process_FASTA_partition(iterator):  ❶
    hashmap = defaultdict(int)  ❷

    for fasta_record in iterator:
        if (fasta_record.startswith(">")):  ❸
            hashmap["z"] += 1
        else:  ❹
            chars = fasta_record.lower()
            for c in chars:
                hashmap[c] += 1  ❺
#           end-for
#   end-for
    print("hashmap=", hashmap)
#
    key_value_list = [(k, v) for k, v in hashmap.iteritems()]  ❻
    print("key_value_list=", key_value_list)
    return  key_value_list  ❼
#
#-----------------------------------
```

❶ iterator is an handle to a single partition

❷ create a hash table of [String, Integer]

❸ handle comments for input data

❹ handle a DNA sequence

❺ populate hash table

❻ flatten a hash table into a list of (DNA-letter, frequency)

❼ return list of (DNA-letter, frequency)

In defining the `process_FASTA_partition()` function, we used a `defaultdict(int)`, which works exactly like a normal dictionary (as an associated-array), but it is initialized with a function ("default factory") that takes no arguments and provides the default value for a non-existent key. In our function,

process_FASTA_partition(), a defaultdict is used for counting DNA-letters. The default factory is 'int' (as an Integer data type), which in turn has a default value of zero. For each character in the list, the value is incremented by one where the key is the DNA letter. We do not need to make sure the DNA letter is already a key – it will use the default value of zero.

Step-3: Apply Custom Function to Each Partition

In this step, we apply the process_FASTA_partition() function to each partition. I have formatted output (Listing 4.5) and added some comments to show the output per partition (we have 2 partitions).

Output of mapPartitions() Transformation.

```
>>> recordsRDD.getNumPartitions()
2
>>> pairsRDD = recordsRDD.mapPartitions(process_FASTA_partition)

>>># output for partition 1
hashmap= defaultdict(<type 'int'>,
{
 u'a': 38, u'c': 28,
 u'g': 28, u'n': 2,
 u't': 24, 'z': 3
})
key_value_list= [
  (u'a', 38), (u'c', 28),
  (u'g', 28), (u'n', 2),
  (u't', 24), ('z', 3)]

>>># output for partition 2
hashmap= defaultdict(<type 'int'>,
{
 u'a': 35, u'c': 33,
 u't': 21, u'g': 25,
 'z': 1,
})
key_value_list= [
 (u'a', 35), (u'c', 33),
 (u't', 21), (u'g', 25),
 ('z', 1),
]
```

Note that for solution Version 3, therefore, each partition returns at most 6 (key, value) pairs as:

```
('a', count-of-a)
('t', count-of-t)
('c', count-of-c)
('g', count-of-g)
('n', count-of-n)
('z', count-of-sequences)
```

Final collection from all partitions will be:

```
>>> pairsRDD.collect()
[
 (u'a', 38), (u'c', 28), (u't', 24),
 ('z', 3), (u'g', 28), (u'n', 2),
 (u'a', 35), (u'c', 33), (u't', 21),
 (u'g', 25), ('z', 1)
]
```

Finally, we aggregate and sum up the output (generated from the `mapPartitions()`) for all partitions:

```
>>> frequenciesRDD = pairsRDD.reduceByKey(lambda a, b: a+b)
>>> frequenciesRDD.collect()
[
 (u'a', 73),
 (u'c', 61),
 (u'g', 53),
 (u't', 45),
 (u'n', 2),
 ('z', 4),
]
>>>
```

Pros and Cons of Version 3

Pros:

- This is the most optimal solution for the DNA Base Count problem. The provided solution works, simple, and efficient. This solution improves on

versions 1 and 2 by emitting the least number of (key, value) pairs, since we create a dictionary per partition (rather than each record) and then flatten it into a list of (key, value) pairs

- There is no scalability issue since we use `mapPartitions()` (for handling each partition) and `reduceByKey()` for reducing all (key, value) pairs emitted by partitions
- At most we will create N dictionaries, where N is the number of partitions for all input data (N can be in hundreds or thousands). This will not be a threat to scalability or memory outrage

Cons:

- None

7.3. Handling Empty Partitions

In our solution Version-3, we used the `mapPartitions(func)` transformation, which partitions input data into many partitions and then applies the function `func` (provided by the programmer) to each partition in parallel. What if any of these partitions are empty: it means that there is a partition, but there is no data (RDD elements) to iterate? We do need to write our function `func` (partition handler) in such a way that to handle an empty partition properly and gracefully. You can not just ignore them.

What if there is an exception (corrupted records after a network failure mid-transfer) for a Spark partitioner in partitioning the data, then some partitions might be empty. Also, empty partitions may happen for many other reasons: one reason can be that the partitioner does not have enough data to put for a given partition. This case can happen. Regardless of how empty partitions are created, we do need to handle all of them gracefully.

I will show the concept of an "empty partition" by a simple example. First, we define a function, `debug_partition()` (Listing 4.6 — to be used for testing and teaching only — not to be used in production environments), to show the content of each partition: this is the function, which will be invoked per partition.

Partition Debugger Function.

```
def debug_partition(iterator):
```

```
    print("type(iterator)=", type(iterator))
    print("begin partition ===")
    for x in iterator:
        print(x)
    print("end partition ===")
#end-def
```

You should note that displaying or debugging content of a partition can be costly and should be avoided in production environments. I have included `print` statements for teaching and debugging purposes only.

Next, we define an RDD and partition it into 4 partitions denoted by Listing 4.7.

Four Partitions for RDD.

```
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> rdd = sc.parallelize(numbers, 4)
>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Next, we examine each partition by using the `debug_partition()` function:

```
>>> rdd.foreachPartition(debug_partition)
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
8
9
10
end partition ===
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
5
6
7
end partition ===
begin partition ===   ❶
('type(iterator)=', <type 'itertools.chain'>)
end partition ===  ❷
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
1
2
```

```
3
4
end partition ===
```

❶ beginning of an empty partition

❷ end of an empty partition

From this test program we observe the following:

- A partition can be empty (with no RDD elements)
- Your custom function must handle an empty partitions grecefully (must return a proper value); empty partitions can not be just ignored.
- The data type of iterator (which represents a single partition and passed as a parameter to the `mapPartitions()`) is the `itertools.chain`. The `itertools.chain` is an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence.

Now the question is how to handle an empty n PySpark? The following pattern (Listing 4.8) may be used to handle an empty partition gracefully.

Template to Handle Empty Partition.

```
# This is the template function
# to handle a single partition
#
# source RDD: RDD[T]
#
# parameter: iterator
#
def func(iterator): ❶
    print("type(iterator)=", type(iterator))
    #   ('type(iterator)=', <type 'itertools.chain'>)
    #
    try:
        first_element = next(iterator)       ❷
        # if you are here it means that
        # the partition is NOT Empty
        ... process the partition
        ... and return a proper result

    except StopIteration:                    ❸
```

```
        # if you are here, it means that this
        # partition is Empty; now, you need
        # to handle and return a proper result
```

❶ `iterator` represent a single partition of elements of type `T`

❷ Try to get the first element (as `first_element` of type `T`) for a given partition. If this fails (throws an exception), then control will go to the `except` (exception happened section).

❸ You will be here when a given partition is empty. You can not just ignore empty partitions, you must return a proper value

8. FASTQ Format

DNA sequences can be represented in many different formats including FASTA and FASTQ popular text-based formats (input is given as a text file). In the following sections, DNA Base Count solution is given for the FASTQ format.

A FASTQ file normally uses four (4) lines per DNA sequence:

- Line 1 begins with a '@' character and is followed by a sequence identifier and an optional description (like a FASTA title line).
- Line 2 is the raw sequence of DNA letters.
- Line 3 begins with a '+' character and is optionally followed by the same sequence identifier (and any description) again.
- Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

8.1. FASTQ Format Example

A FASTQ file containing a single sequence might look like this:

```
@SEQ_ID ❶
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT ❷
+ ❸
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65 ❹
```

❶ Line 1: DNA sequence ID

❷ Line 2: DNA sequence

❸ Line 3: ignored for DNA Base Count

❹ Line 4: the quality values for the DNA sequence

8.2. FASTQ Data Download

To test the DNA Base Count programs (provided in this chapter), you may download FASTQ data from the SP1.fq[6]. Some sample FASTQ files are provided in this book's GIT repository[7].

9. DNA Base Count by PySpark — FASTQ Version

For FASTQ format, I only provide the most optimized solution using the `mapPartitions()` transformation. High-level solution is presented below:

```
Step-1: read data and create an RDD[String]

Step-2: drop out the non-needed records
        (keep only Line 2)

Step-3: for every 4 records, just keep
        Line 2 (the record of the raw
        sequence of DNA letters).

Step-4: apply the `mapPartitions()` transformation
        to perform DNA Base Counting
```

A complete solution is provided as a dna_base_count_FASTQ.py in this book's GIT repository[8].

10. Summary

- In solving big data problems, we have a lot of choices in selecting Spark's transformations and actions and their performances are different from each other.
- When selecting and using a trnsformation to solve a specific data problem, make sure that you test it with "real big data" rather than toy data

[6] https://molb7621.github.io/workshop/_downloads/SP1.fq
[7] https://github.com/mahmoudparsian/pyspark-algorithms/tree/master/code/chap04
[8] https://github.com/mahmoudparsian/pyspark-algorithms/tree/master/code/chap04

- For large volume of (key, value) pairs, overall, the `reduceByKey()` transformation performs better than `groupByKey()` due to different shuffling algorithms
- When you have a big data and you want to just extract and aggregate/derive small amount of information (such as Min-Max, Top-10, DNA Base Count, …), then the `mapPartitions()` transformation may be used.
- Emitting minimal number of (key, value) pairs improves the performance of your data solutions