**Q1.**

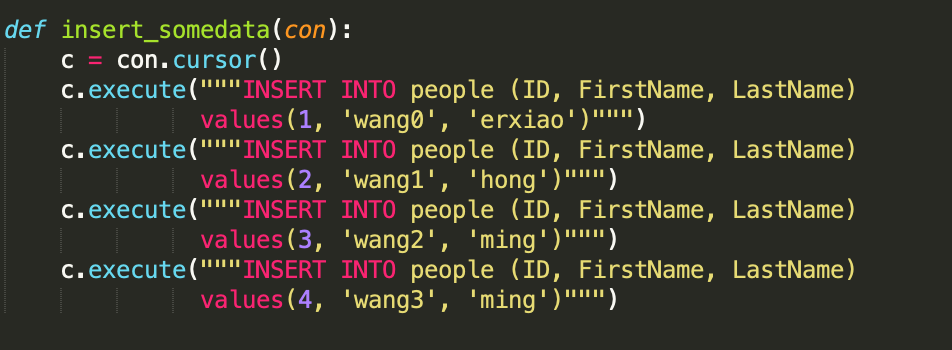
1.1



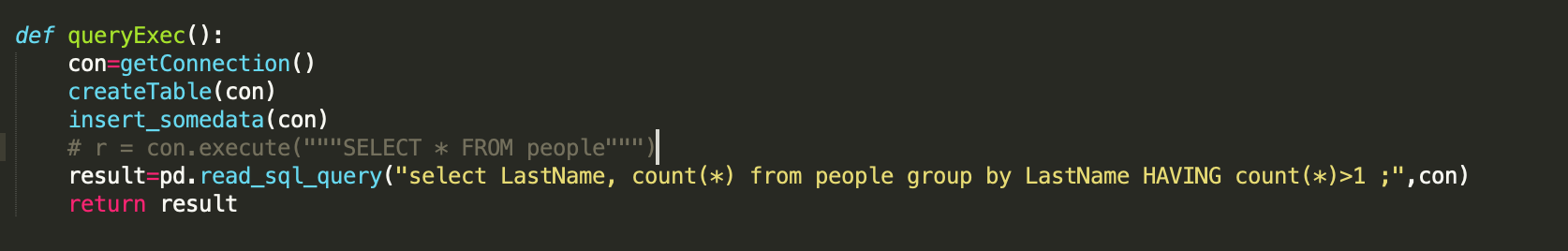
In the folder:



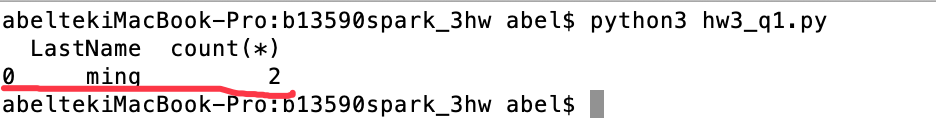
1.2



1.3



The outuput of program result is:



**Q2**

Spark RDD operations:

transformation operations is such kind of RDD operation: it changes rdd data

action operation is such RDD operation: it’s not change rdd data ,but give an output.

So a **transformation** operations is a function that produces new RDD from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed

Example 1:

numList = [1,2,3]

firstRDD = sc.parallelize(numList)

# An example of a transformation

# Multiply the values by 2

secondRDD = firstRDD.map(lambda x: x\*2)

# An example of an action

secondRDD.collect()

Example 2:

# An example of a transformation

# Multiply the values by 2

wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']

wordsRDD = sc.parallelize(wordsList)

wordCountsCollected = (wordsRDD

.map(lambda x: (x,1))

.reduceByKey(lambda a,b: a+b)

# An example of an action

wordCountsCollected.count()

Example3:

wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']

wordsRDD = sc.parallelize(wordsList)

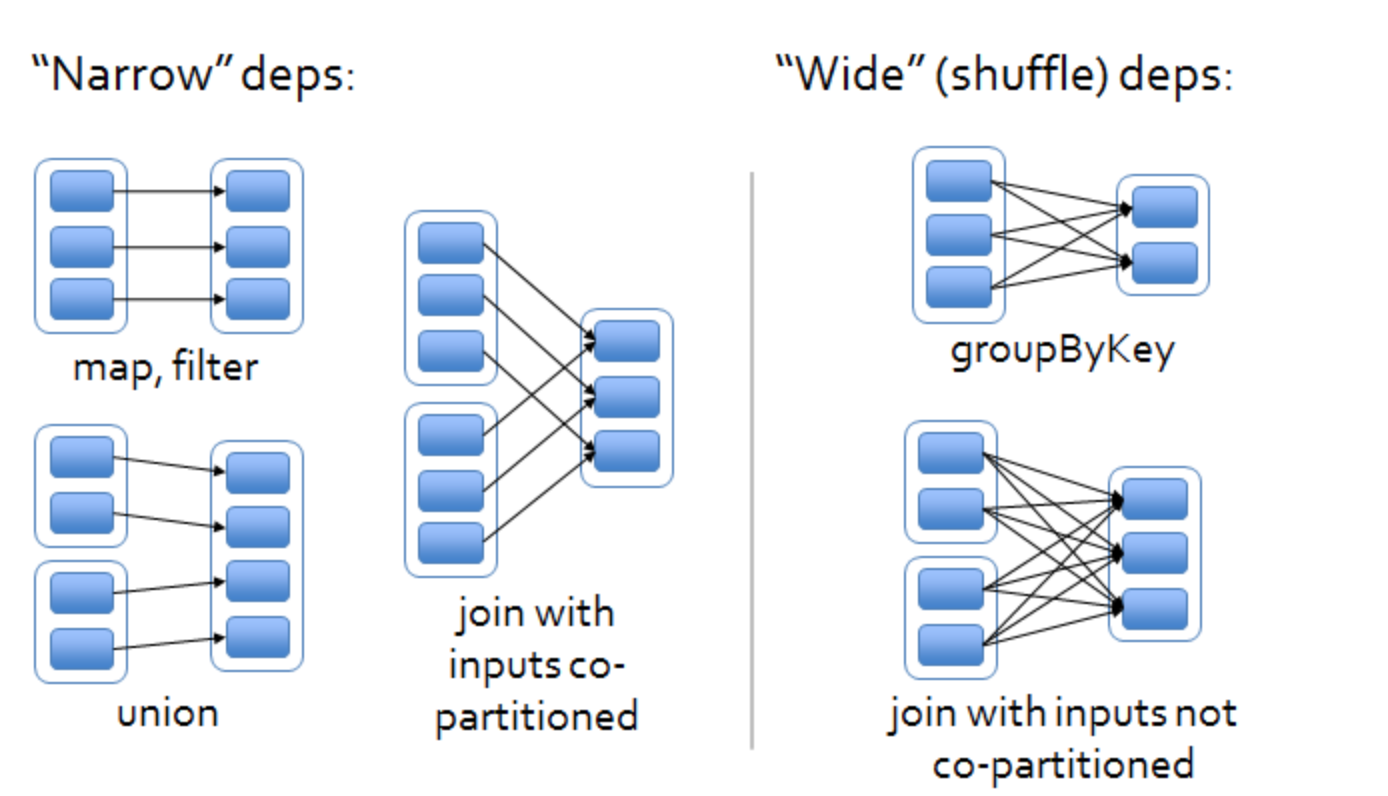
filterrdd = wordsRDD.flatMap(lines => lines.split(” “)).filter(value => value=”cat”)

# An example of an action

filterrdd.saveAsTextFile(“tempFile2.name”)

如果觉得还不够详细，你可以补充解释：“DD 支持两种操作：转化操作和行动操作。RDD 的转化操作是返回一个新的 RDD 的操作，比如 map() 和 filter()，而行动操作则是向驱动器程序返回结果或把结果写入外部系统的操作，会触发实际的计算，比如 count() 和 first()。Spark 对待转化操作和行动操作的方式很不一样，因此理解你正在进行的操作的类型是很重要的。如果对于一个特定的函数是属于转化操作还是行动操作感到困惑，你可以看看它的返回值类型：转化操作返回的是 RDD，而行动操作返回的是其他的数据类型。”

**Q3**

****

**Narrow dependencies**

When each partition at the parent RDD is used by at most one partition of the child RDD, then we have a narrow dependency. Computations of transformations with this kind of dependency are rather fast as they do not require any data shuffling over the cluster network. In addition, optimizations such as *pipelining* are also possible.

## Wide dependencies

When each partition of the parent RDD may be depended on by multiple child partitions (wide dependency), then the co**mputation spee**d might be significantly affected as we might need to shuffle data around different nodes when creating new partitions.

**Basically:Narrow transformation** - doesn't require the data to be shuffled across the partitions. for example, Map, filter etc… But **wide transformation**- requires the data to be shuffled for example, reduceByKey …

Stage:

T*o distinguish between wide and narrow dependencies, we mainly look at the Partition flow of the parent RDD: flow to a single RDD is narrow dependence, and flow to multiple RDDs is wide dependence.*

*The Spark Stage division is calculated from the last RDD forward, and it is added to the stage when it encounters a narrow dependency (NarrowDependency), and it is disconnected when it encounters a wide dependency (ShuffleDependency). The number of tasks in each stage is determined by the number of partitions in the last RDD of the stage. If the stage is to generate a Result, the Tasks in the stage are all ResultTasks, otherwise it is ShuffleMapTask.*

*The calculation result of ShuffleMapTask needs to be shuffled to the next stage, which is essentially equivalent to the mapper in MapReduce. Result Task is equivalent to the reducer in MapReduce. Therefore, the entire calculation process will be established from back to front according to the data dependency, and a new stage will be formed when a wide dependency is encountered.*

*Stage scheduling is completed by DAG Scheduler. The directed acyclic graph DAG of the RDD segmented out the directed acyclic graph DAG of the stage. Stage takes the last executed Stage as the root for breadth-first traversal, and traverses to the initial stage execution. If the submitted Stage still has an unfinished parent stage, the stage needs to wait for its parent stage to execute before executing*

不够的对于关于stage划分的附加中文解释：区分宽窄依赖，我们主要从父RDD的Partition流向来看：流向单个RDD就是窄依赖，流向多个RDD就是宽依赖。

Spark Stage划分，就是从最后一个RDD往前推算，遇到窄依赖（NarrowDependency）就将其加入该Stage，当遇到宽依赖（ShuffleDependency）则断开。每个Stage里task的数量由Stage最后一个RDD中的分区数决定。如果Stage要生成Result，则该Stage里的Task都是ResultTask，否则是ShuffleMapTask。

ShuffleMapTask的计算结果需要shuffle到下一个Stage，其本质上相当于MapReduce中的mapper。Result Task则相当于MapReduce中的reducer。因此整个计算过程会根据数据依赖关系自后向前建立，遇到宽依赖则形成新的Stage。

Stage的调度是由DAG Scheduler完成的。由RDD的有向无环图DAG切分出了Stage的有向无环图DAG。Stage以最后执行的Stage为根进行广度优先遍历，遍历到最开始执行的Stage执行，如果提交的Stage仍有未完成的父Stage，则Stage需要等待其父Stage执行完才能执行

**Q4**

4.1 Spark Streaming uses the concept of DStreams, which are basically microbatches of data that are RDDs. We also saw some transformations that can be applied to DStreams. DStream transformations can be grouped into two types: stateless and stateful transformations.

In a stateless transformation, whether or not each microbatch of data is processed does not depend on the previous data batches, so each batch is fully independent of whatever batches of data preceded it.

In stateful transformations, whether or not each microbatch of data is processed depends partially or wholly on the previous batches of data, so each batch considers what happened prior to it and uses that information while being processed

Stateless demo1:

map() f: (T) -> U

s.map(x => x + 1)

Stateless demo2:

filter() f: T -> Boolean

ds.filter(x => x != 1)

state demo1（简介：“在 Scala 中使用 updateStateByKey() 运行响应代码的计数）

def updateRunningSum(values: Seq[Long], state: Option[Long]) = {

Some(state.getOrElse(0L) + values.size)

}

val responseCodeDStream = accessLogsDStream.map(log => (log.getResponseCode(), 1L))

val responseCodeCountDStream = responseCodeDStream.updateStateByKey(updateRunningSum \_)

demo2(简介：Scala 中的窗口计数操作)

“val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()}

val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))

val requestCount = accessLogsDStream.countByWindow(Seconds(30), Seconds(10))”

附加解释：和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流（discretized stream）作为抽象表示，叫作 DStream。DStream 是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 所组成的序列（因此得名“离散化”）。DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作，一种是转化操作（transformation），会生成一个新的 DStream，另一种是输出操作（output operation），可以把数据写入外部系统中。DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。”

Q5

The design concept of MLlib is very simple: express the data in the form of RDD, and then call various algorithms on the application data set"

"Next step.

(1) First use the string RDD to represent your message.

(2) Run a feature extraction (feature extraction algorithm) in MLlib to convert text data into digital features (suitable for machine learning algorithm processing); this operation will return an RDD.

(3) Call a classification algorithm (such as logistic regression) on the linear RDD; this step will return a model object, which can be used to classify new data points.

(4) Use MLlib's evaluation function to evaluate the model on the test data set.

中文附加解释：MLlib 的设计理念非常简单：把数据以 RDD 的形式表示，然后在分布式数据集上调用各种算法”

“下步骤操作。

(1) 首先用字符串 RDD 来表示你的消息。

(2) 运行 MLlib 中的一个特征提取（feature extraction）算法来把文本数据转换为数值特征（适合机器学习算法处理）；该操作会返回一个向量 RDD。

(3) 对向量 RDD 调用分类算法（比如逻辑回归）；这步会返回一个模型对象，可以使用该对象对新的数据点进行分类。

(4) 使用 MLlib 的评估函数在测试数据集上评估模型。”