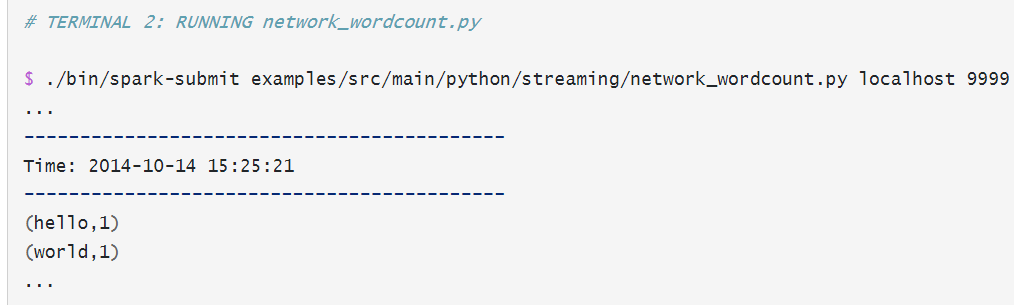
**Big Data Assignment 5**

**1. Perform random data lookup and streaming in SPARK?**

The data lookup in spark can be performed using the lookup operation. It returns the list of values in the RDD for key ‘key’. An example for the same is shown below:

* val rdd1 = sc.parallelize(Seq(("Spark",78),("Hive",95),("spark",15),("HBase",25),("spark",39),("BigData",78),("spark",49)))
* rdd1.lookup("spark")
* **Output:**
* Seq[Int] = WrappedArray(15, 39, 49)

**Streaming:**

* Let’s say we want to count the number of words in text data received from a data server listening on a TCP socket. All we need to do is as follows.
* First, we import StreamingContext, which is the main entry point for all streaming functionality. We create a local StreamingContext with two execution threads, and batch interval of 1 second.
  + **from** **pyspark** **import** SparkContext
  + **from** **pyspark.streaming** **import** StreamingContext
  + *# Create a local StreamingContext with two working thread and batch interval of 1 second*
  + sc = SparkContext("local[2]", "NetworkWordCount")
  + ssc = StreamingContext(sc, 1)
* Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).
  + *# Create a DStream that will connect to hostname:port, like localhost:9999*
  + lines = ssc.socketTextStream("localhost", 9999)
* This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.
  + *# Split each line into words*
  + words = lines.flatMap(**lambda** line: line.split(" "))
* flatMap is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream. In this case, each line will be split into multiple words and the stream of words is represented as the words DStream. Next, we want to count these words.
  + *# Count each word in each batch*
  + pairs = words.map(**lambda** word: (word, 1))
  + wordCounts = pairs.reduceByKey(**lambda** x, y: x + y)
  + *# Print the first ten elements of each RDD generated in this DStream to the console*
  + wordCounts.pprint()
* Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet. To start the processing after all the transformations have been setup, we finally call
  + ssc.start() *# Start the computation*
  + ssc.awaitTermination() *# Wait for the computation to terminate*
* Then, in a different terminal, you can start the example by using
  + $ ./bin/spark-submit examples/src/main/python/streaming/network\_wordcount.py localhost 9999
* Then, any lines typed in the terminal running the netcat server will be counted and printed on screen every second. It will look something like the following.
  + 

**2. What are the benefits of partitioning in Hive? Could you give an example?**

Hive organizes tables into partitions. Using partition, it is easy to query a portion of the data. Partitioning is a way of dividing a table into related parts based on the values of columns like date, city, and department. In Hive, partitioning is supported for both managed and external tables. Some benefits of portioning in Hive are:

* It distributes execution load horizontally.
* In partition, faster execution of queries with the low volume of data takes place. For example, search population from Vatican City returns very fast instead of searching entire world population.

Example:

* Let’s assume we have a US census table which contains zip code, city, state and other columns. Creating a partition on state splits the table into around 50 partitions, when searching for a zipcode with in a state (state=’CA’ and zipCode =’92704′) results in faster as it need to scan only in a state=CA partition directory. When creating partitions you have to be very cautious with the number of partitions it creates, as having too many partitions creates too many sub-directories in a table directory which bring unnecessarily and overhead to NameNode since it must keep all metadata for the file system in memory.

**3. What commands should I use to start and stop Hadoop daemons?**

* **start-all.sh & stop-all.sh** 
  + Used to start and stop Hadoop daemons all at once.
  + Issuing it on the master machine will start/stop the daemons on all the nodes of a cluster.
  + This method is deprecated though.
* **start-dfs.sh, stop-dfs.sh and start-yarn.sh, stop-yarn.sh** 
  + Same as above but start/stop HDFS and YARN daemons separately on all the nodes from the master machine.
  + It is advisable to use these commands now over start-all.sh & stop-all.sh
* hadoop-daemon.sh namenode/datanode and yarn-deamon.sh resourcemanager
  + To start individual daemons on an individual machine manually. You need to go to a particular node and issue these commands.
* **Use case:** Suppose you have added a new DN to your cluster and you need to start the DN daemon only on this machine,
  + bin/hadoop-daemon.sh start datanode
* **Note :** You should have ssh enabled if you want to start all the daemons on all the nodes from one machine.

**4. What exactly is in-memory computing, and how does Spark achieve it?**

* Spark is considered to be 10x times quicker than traditional HDFS or MapReduce techniques, to process Big Data. How can Spark do it? It’s because of ‘In-Memory Computation’ that Spark uses to achieve these performances.
* In Spark, the Big Data is not stored in the data drive which makes the system slow, for pulling and processing the data, rather Spark takes another route by keeping data in Random Access Memory (RAM), and that helps Spark to detect a pattern, analyze large data quickly, and also reduce the dependency on memory, eventually cutting down costs for memory requirement.
* The in-memory capability of Spark is good for machine learning and micro-batch processing. It provides faster execution for iterative jobs.

**How does Spark achieve this?**

* The main abstraction of Spark is its RDDs. And the RDDs are cached using the **cache()** or **persist()** method.
* When we use cache() method,
  + all the RDD stores in-memory. When RDD stores the value in memory, the data that does not fit in memory is either recalculated or the excess data is sent to disk.
  + Whenever we want RDD, it can be extracted without going to disk.
  + This reduces the space-time complexity and overhead of disk storage.
* When we use persist() method,
  + the RDDs can also be stored in-memory, we can use it across parallel operations.
  + The difference between cache() and persist() is that using cache() the default storage level is MEMORY\_ONLY while using persist() we can use various storage levels such as:
    - MEMORY\_ONLY
    - MEMORY\_AND\_DISK
    - MEMORY\_ONLY\_SER
    - MEMORY\_AND\_DISK\_SER
    - DISK\_ONLY
    - MEMORY\_ONLY\_2 and MEMORY\_AND\_DISK\_2

**5. Could you elaborate on Hadoop configuration files?**

Configuration Files are the files which are located in the extracted tar.gz file in the etc/hadoop/ directory. All Configuration Files in Hadoop are listed below:

* **HADOOP-ENV.sh**
  + It specifies the environment variables that affect the JDK used by Hadoop Daemon (bin/hadoop).
  + We know that Hadoop framework is wriiten in Java and uses JRE so one of the environment variable in Hadoop Daemons is $Java\_Home in Hadoop-env.sh.
* **CORE-SITE.XML**
  + It is one of the important configuration files which is required for runtime environment settings of a Hadoop cluster.
  + It informs Hadoop daemons where the NAMENODE runs in the cluster.
  + It also informs the Name Node as to which IP and ports it should bind.
* **HDFS-SITE.XML**
  + It is one of the important configuration files which is required for runtime environment settings of a Hadoop.
  + It contains the configuration settings for NAMENODE, DATANODE, SECONDARYNODE.
  + It is used to specify default block replication.
  + The actual number of replications can also be specified when the file is created,
* **MAPRED-SITE.XML**
  + It is one of the important configuration files which is required for runtime environment settings of a Hadoop.
  + It contains the configuration settings for MapReduce.
  + In this file, we specify a framework name for MapReduce, by setting the MapReduce.framework.name.
* **Masters**
  + It is used to determine the master Nodes in Hadoop cluster.
  + It will inform about the location of SECONDARY NAMENODE to Hadoop Daemon.
  + The Mater File on Slave node is blank.
* **Slave**
  + It is used to determine the slave Nodes in Hadoop cluster.
  + The Slave file at Master Node contains a list of hosts, one per line.
  + The Slave file at Slave server contains IP address of Slave nodes.