Exam Solutions

Ahmed Alhasan 'ahmal787'

2020-08-20

Q1

- since we have the key (the ID) we could just search for that key using get(userID) and then we could search the value optained by a user writen application since key-value databases does not have indexes over values

Q2

- Secondary indexes over fields in the documents are possible which in key-value store there is no indexes over values. This gives flexibility when searching for both key-pairs inside a document

Q3

- write scalability can be achieved by adding more resources to the system, like more CPU power to increase throughput, more RAM to increase high accessability and more disk storage to increase amount of data that can be stored

Q4

- Wrong, because the master node has the privilege to write in the slave nodes to update the content to achieve consistency.

Q5

- to achieve write consistency we need $2W > N$, therefore we need $w = 3$ so $2*3 > 4$

Q6

- Only Record Reader and Output Formatter involve network I/O

  Record Reader: Parses an input file block from stdin into key value pairs that define input data records

  Output Formatter: Translates the final (key,value) pair from the reduce function and writes it to stdout to a file in HDFS

Q7

```
import sys

for number in sys.stdin:
  # assuming one floating number per line

  # converting to floats
  try:
    number = float(number)
  except(ValueError):
    # silently ignore invalid line
    continue

  total += number**2

# get the geometirc mean as per the formula
g_mean = sqrt(total)
print(g_mean)
```

Q8

- in memory

Q9

```
import math
from pyspark import SparkContext
sc = SparkContext(appName="K-means")

# map the data to a desired form and cache it
# because we will use it later
data = sc.textFile().split().map().cache()


def closestPoint(p, centers):
  """
  calculates the distance between data points and the current
  centroids and returns the index of the closest centroid which
  represents the the cluster number
  """
  bestIndex = 0
  closest = float(10000) # set it to a very high value
  for i, center in enumerate(centers):
    # Or we could you use distance(p, centers[i])
    tempDist = math.sqrt(((p[0]-center[0])**2)+((p[1]-center[1])**2))
    if tempDist < closest:
      closest = tempDist
      bestIndex = i
  return bestIndex
```

```
tempDist = # some large value with respect to the data
convergeDist = # some low value with respect to the data

while tempDist > convergeDist:
  # spliting the data randomly to 2 clusters
  cluster1, cluster2 = data.randomSplit(weights = [0.5, 0.5], seed = 1). \
                          map(lambda a: (a, 1)).reduceByKey(lambda a,b: a+b)
  total1 = cluster1.keys().sum() # take the sum of points in a cluster
  total2 = cluster2.keys().sum()

  centroid1 = total1 / cluster1[1] # get the average
  centroid2 = total2 / cluster2[1]

  kPoints = (centroid1, centroid2) # make a tuple of all centroids

  closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
  # combine into (points sum, points count)
  pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
  # take the new centroids as the average of each cluster and collect the results as
  # (cluster number, new centroid) at the driver node
  newPoints = pointStats.map(lambda st: (st[0], (st[1][0][0] / st[1][1], st[1][0][1] / st[1][1]))).colle
  # calculate the distance between the old centroids and the new ones to measure convergance
  tempDist = sum(math.sqrt(((kPoints[iK][0]-newp[0])**2) +
             ((kPoints[iK][1]-newp[1])**2)) for (iK, newp) in newPoints)
  # update to the new centroids
  for (iK, newp) in newPoints:
    kPoints[iK] = newp


print("Final centers: " + str(kPoints))
```