Warehouse Scale Computing

1. Amdahl's Law

1) You are going to train the image classifier with 50,000 images on a WSC having more than 50,000 servers. You notice that 99% of the execution can be parallelized. What is the speedup?

1 / (0.01 + 0.99 / 50,000) ≈ 1 / 0.01 = 100

2. Failure in a WSC

1) In this example, a WSC has 55,000 servers, and each server has four disks whose annual failure rate is 4%. How many disks will fail per hour?

(55,000 x 4 x 0.04) / (365 x 24) = 1.00 → MTTF = 1 hour

2) What is the availability of the system if it does not tolerate the failure? Assume that the time to repair a disk is 30 minutes.

MTTF = 1, MTTR = 0.5 → Availability = 1 / (1 + 0.5) = 2/3 = 66.6%

3. Power Usage Effectiveness (PUE) = (Total Building Power) / (IT Equipment Power)

Sources speculate Google has over 1 million servers. Assume each of the 1 million servers draw an average of 200W, the PUE is 1.5, and that Google pays an average of 6 cents per kilowatt-hour for datacenter electricity.

1) Estimate Google's annual power bill for its datacenters.

1.5 x 1,000,000 servers x 0.2kW/sever x $0.06/kW-hr x 8760 hrs/yr = $157.68 M/yr

2) Google reduced the PUE of a 50,000 machine datacenter from 1.5 to 1.25 without decreasing the power supplied to the servers. What's the cost savings per year?

(1.5 - 1.25) x 50,000 servers x 0.2kW/server x $0.06/kW-hr x 8760 hrs/yr = $1.314M/yr

Map Reduce

Use pseudocode to write MapReduce functions necessary to solve the problems below. Also, make sure to fill out the correct data types. Some tips:

- The input to each MapReduce job is given by the signature of the **`map()`** function.
- The function **`emit(key k, value v)`** outputs the key-value pair **`(k, v).`**
- The **`for(var in list)`** syntax can be used to iterate through **`Iterable`**s or you can call the **`hasNext()`** and **`next()`** functions.
- Usable data types: **`int, float, String`**. You may also use lists and custom data types composed of the aforementioned types.
- The method **`intersection(list1, list2)`** returns a list that is the intersection of list1 and list2.

1. Given the student's name and the course taken, output each student's name and total GPA.

Declare any custom data types here:

```
CourseData:
    int courseID
    float studentGrade  // a number from 0-4
```

```
map(String student, CourseData value):
  emit(student, value.studentGrade)
```

```
reduce( String key,
     Iterable< float > values):
  totalPts = 0
  totalClasses = 0
  for ( grade in values ):
      totalPts += grade
      totalClasses++
  emit(key, totalPts / totalClasses)
```

2. Given a person's unique int ID and a list of the IDs of their friends, compute the list of mutual friends between each pair of friends in a social network.

Declare any custom data types here:

```
FriendPair:
  int friendOne
  int friendTwo
```

```
map(int personID, list<int> friendIDs):
  for ( fID in friendIDs ):
    if ( personID < fID ):
      friendPair = ( personID, fID )
    else:
      friendPair = ( fID, personID )
    emit(friendPair, friendIDs)
```

```
reduce( FriendPair key,
     Iterable< list<int> > values):
  mutualFriends =
    intersection(
      values.next(), values.next())
  emit(key, mutualFriends)
```

3. a) Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has.

Declare any custom data types here:

```
CoinPair:
  String person
  String coinType
```

map	reduce

`map(String person, String coinType):`
`    key = (person, coinType)`
`    emit(key, 1)`
	`reduce(CoinPair key,`
	`       Iterable< int > values):`
	`  total = 0`
	`  for ( count in values ):`
	`    total += count`
	`  emit(key, total)`

b) Using the output of the first MapReduce, compute the amount of money each person has. The function `valueOfCoin(String coinType)` returns a float corresponding to the dollar value of the coin.

map	reduce

`map(CoinPair key, int amount):`
`  emit(coinPair.person,`
`  valueOfCoin(coinPair.coinType)*amount)`
	`reduce(String key,`
	`       Iterable< float > values):`
	`  total = 0`
	`  for ( amount in values ):`
	`    total += amount`
	`  emit(key, total)`

Spark

- RDD: primary abstraction of a distributed collection of items
- Transforms: RDD → RDD

`map``(`*func*`)`	Return a new distributed dataset formed by passing each element of the source through a function *func*.
`flatMap(`*func***`)`**	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
`reduceByKey(`*func***`)`**	When called on a dataset of `(K,V)` pairs, returns a dataset of `(K,V)` pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type `(V,V) => V`.

• Actions: RDD → Value

reduce(*func*)	Aggregate the elements of the dataset ***regardless of keys*** using a function *func*

1. Implement Problem 1 of MapReduce with Spark

```
# students: list((studentName, courseData))
studentsData = sc.parallelize(students)
out = studentsData.map(lambda (k, v): (k, (v.studentGrade, _1_)))
                  .reduceByKey(lambda v1, v2: (v1[0] + v2[0], v1[1] + v2[1]))
                  .map(lambda (k, v): (k, v[0] / v[1]))
```

2. Implement Problem 2 of MapReduce with Spark

```
def genFriendPairAndValue(pID, fIDs):
  return [((pID, fID), fIDs) if pID < fID else (fID, pID) for fID in fIDs]
def intersection(l1, l2):
  return [x for x in b1 if x in b2]
# persons: list((personID, list(friendID))
personsData = sc.parallelize(persons)
out = personsData.flatMap(lambda (k, v): genFriendPairAndValue(k, v))
                 .reduceByKey(lambda v1, v2: intersection(v1, v2))
```

3. Implement Problem 3 of MapReduce with Spark

```
# coinPairs: list((person, coinType))
coinData = sc.parallelize(coinPairs)
#(3.a) out: list(((person, coinType), count))
out1 = coinData.map(lambda (k1, k2): ((k1, k2), 1))
               .reduceByKey(lambda v1, v2: v1 + v2)
#(3.b)
out2 = out1.map(lambda (k, v): (k[0], v * valueOfCoin(k[1])))
           .reduceByKey(lambda v1, v2: v1 + v2)
```