Sampling

Reservoir Sampling

Common Operations on Streams

- Sampling
- Filtering
- Counting Distinct
- Clustering

Sampling

- Extract samples from a stream such that they bear the approximately the same statistical properties- serves as the representative set.

Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of *k* items from a list *S* containing *n* items, where *n* is either a very large or unknown number.

Assumptions-

> ***n*** is large enough that the whole dataset does not fit into main memory, whereas **k**, the desired sample does.

Pseudo Code

```
array R[k];    // result
integer i, j;

// fill the reservoir array with the first k incoming elements
for each i in 1 to k do
    R[i] = S[i]
done;

// replace elements probabilistically
for each i in k+1 to length(S) do
    j = random(1, i);   // important: inclusive range
    if j <= k then
        R[j] = S[i]
    end if;
Done;
```

Proof of Correctness

- Equal Likelihood:
    - Probability that any item *S[i]* where *0 <= i < n* will be in final *R[]* is *k/n*.

Proof of Correctness

- Let x_i be the i-th element and S_i be the solution obtained after examining the first i elements.
- RTS:-

  $Pr[x_j \in S_i] = k/i$ for all $j \leq i$ with $k \leq i \leq n$.

  This will imply that the probability that any element is in the final solution S_n is exactly k/n.

A Different Perspective

- The core insight behind reservoir sampling is that picking a random sample of size k is equivalent to *generating a random permutation (ordering) of the elements and picking the top k elements*.
- Associate a random float id with each element and pick the elements with the k largest ids. Since the ids induce a random ordering of the elements (assuming the ids are distinct), it is clear that the elements associated with the k largest ids form a random subset.

A Different Perspective

- The goal here is to incrementally keep track of the k elements with largest ids seen so far.
- (Streaming Sequential Setting)

```
import sys, random
from heapq import heappush, heapreplace

k = int(sys.argv[1])
H = []

for x in sys.stdin:
  r = random.random() # the randomly associated id.
  if len(H) < k: heappush(H, (r, x))
  elif r > H[0][0]: heapreplace(H, (r, x)) # replace

print ''.join([x for (r,x) in H]),
```

Map Reduce Approach

- Let *K* the number of samples you want. We'll assume that this is small enough to hold in memory on one of your nodes.
- Each mapper associates a random id with each element and keeps track of the top k elements.
- The top k elements of each mapper are then sent to a single reducer which will complete the job by extracting the top k elements among all.
- Data sent to the Reducer is now restricted to top k found in each mapper instead of the whole data set.

Mapper

```python
# mapper.py
 import sys, random
from heapq import heappush, heapreplace

k = int(sys.argv[1])
H = []

for x in sys.stdin:
  r = random.random() #randomly associated key, x is the value
  if len(H) < k: heappush(H, (r, x))
  elif r > H[0][0]: heapreplace(H, (r, x))

for (r, x) in H:
  #negating the keys, the reducer receives the elements from highest to lowest
  print '%f\t%s' % (-r, x),
```

Reducer

- Hadoop framework will automatically present the values to the reducer in order of keys from lowest to highest.

```
# reducer.py

import sys

k = int(sys.argv[1])
c = 0

for line in sys.stdin:
   (r, x) = line.split('\t', 1)
   print x, # emit the values
   c += 1
   if c == k: break
```

Compromise on linear time complexity?

- Each heap operation takes O(logk) time, so a trivial bound for the overall running time would be O(nlogk).
- However, this bound can be improved as the heap replace operation is only executed when the i-th element is larger than the root of the heap.
- This happens only if the i-th element is one of the k largest elements among the first i elements, which happens with probability k/i.
- Therefore the expected number of heap replacements is $\sum_{\text{over } i= k+1 \text{ ->} n}$ k/i ≈klog(n/k).
- The overall time complexity is then O(n+klog(n/k)logk), which is substantially linear in n unless k is comparable to n.

Big Data it is!

- So far we worked under the assumption that the desired sample would fit into memory.
- After all, in the big data world, 1% of a huge dataset may still be too much to keep in memory!

Solution

- Use multiple reducers.
- The key idea is:
  - suppose we have ℓ buckets and generate a random ordering of the elements first by putting each element in a random bucket and then by generating a random ordering in each bucket.
  - The elements in the first bucket are considered smaller (with respect to the ordering) than the elements in the second bucket and so on.
  - if we want to pick a sample of size k, we can collect all of the elements in the first j buckets if they overall contain a number of elements t less than k, and then pick the remaining $k-t$ elements from the next bucket.

- Here ℓ is a parameter such that n/ℓ elements fit into memory.
- Note the key aspect that buckets can be processed distributively.

Mapper

- Mappers associate with each element an id (j,r) where j is a random index in $\{1,2,\ldots,\ell\}$ to be used as key, and r is a random float for secondary sorting. In addition, mappers keep track of the number of elements with key less than j (for $1 \leq j \leq \ell$) and transmit this information to the reducers.

Mapper

```python
# largeK_mapper.py

import sys, random
# number of buckets
l = int(sys.argv[1])
S = [0 for j in range(l)]

for x in sys.stdin:
   (j,r) = (random.randint(0,l-1), random.random()) #key
  S[j] += 1
  print '%d\t%f\t%s' % (j, r, x), #key, value pair

for j in range(l): # compute partial sums
  prev = 0 if j == 0 else S[j-1]
  S[j] += prev # number of elements with key less than j
  print '%d\t-1\t%d\t%d' % (j, prev, S[j]) # secondary key is -1 so reducer gets this first
```

Reducer

- The reducer associated with some key (bucket) j acts as follows: if the number of elements with key less or equal than j is less or equal than k then output all elements in bucket j; otherwise, if the number of elements with key strictly less than j is $t<k$, then run a reservoir sampling to pick $k-t$ random elements from the bucket; in the remaining case, that is when the number of elements with key strictly less than j is at least k, don't output anything.

*C:\Users\saumya_space\Desktop\rand_subset_seq.py - Notepad++

```python

k = int(sys.argv[1])
 line = sys.stdin.readline()
while line:
  # Aggregate Mappers information
  less_count, upto_count = 0, 0
  (j, r, x) = line.split('\t', 2)
  while float(r) == -1:
    l, u = x.split('\t', 1)
    less_count, upto_count = less_count + int(l), upto_count + int(u)
    (j, r, x) = sys.stdin.readline().split('\t', 2)
  n = upto_count - less_count # elements in bucket j

  # Proceed with one of the three cases
  if upto_count <= k: # in this case output the whole bucket
    print x,
    for i in range(n-1):
      (j, r, x) = sys.stdin.readline().split('\t', 2)
      print x,

  elif less_count >= k: # in this case do not output anything
    for i in range(n-1):
      line = sys.stdin.readline()

  else: # run reservoir sampling picking (k-less_count) elements
    k = k - less_count
    S = [x]
    for i in range(1,n):
      (j, r, x) = sys.stdin.readline().split('\t', 2)
      if i < k:
        S.append(x)
      else:
        r = random.randint(0,i-1)
        if r < k: S[r] = x
    print ''.join(S),
  line = sys.stdin.readline()
```

Limitations

- Communication overhead of transferring *the whole* dataset from the mappers to the reducers as opposed to the k items only previously.
- The previous approach should be preferred if the sample size k fits in memory.