# MapReduce in Streaming Data

October 21, 2013

**Instructor : Dr. Barna Saha**

**Course : CSCI 8980 : Algorithmic Techniques for Big Data Analysis**

**Scribe by : Neelabjo Shubhashis Choudhury**

## Introduction to MapReduce, Minimum Spanning Tree, Matching and Dense sub-graph computation in Streaming Data

In this lecture, we introduce MapReduce algorithms for the data streaming approach. In MapReduce framework multiple servers exists. The method is similar to parallel processing on different servers where each server has restrictions on the amount of data that it can contain. There is enough space overall to store the data but not enough to store it on a particular server. Hence multiple servers are used. MapReduce is a relatively new method to tackle such situations compared to parallel processing way. In parallel processing, the methods are difficult to debug and it becomes difficult to synchronize between the calls. MapReduce does parallel programming in an easier way by not considering low level synchronization and distributing data over the servers easily. It imposes certain structure to the data. Certain trade-offs are associated with it too such as only certain kinds of parallel processing are allowed in MapReduce.

## 1 MapReduce : Data and Operations

In MapReduce, data is represented as $< Key, Value >$ pairs. For example, a graph can be represented as a list of edges with associated edge weights represented as $< Key, Value >$ pairs in the following way :

Key $= (u, v)$

Value = Edge weight of $(u, v)$

Three types of operation exists :

**Map function** : It takes $< Key, Value >$ pairs as input and generates another set (preferably a list) of $< Key, Value >$ pairs. Mapper decomposes the key into its 2 constituent values thus reducing the $< Key, Value >$ pair. This function is specified by the user.

**Shuffle function** : It groups the $< Key, Value >$ pairs with the same key into a single $< Key, Value >$ pair. After shuffle, all $< Key, Value >$ pairs with the same key goes to the same server.

**Reduce function** : It takes the $< Key, Value >$ pair from the shuffle and does some function or performs some operation on the keys or the values to generate another $< Key, Value >$ pair (key and list of values). This sequence of Mapper to Reducer and again to Mapper can proceed in multiple rounds.

**Example 1 : Total weighted degree of a graph**

Map function breaks down the graph in the following way - A vertex becomes a key and an the weight of the incident edge on that vertex becomes the value. A vertex can have a number of $< Key, Value >$ pairs with key being the vertex itself and values equal to the weights of edges incident on it. Hence number of values equals the degree of the vertex.

Shuffle function arranges the $< Key, Value >$ pair in the following way - All $< Key, Value >$ pairs with the same key are put in a single $< Key, Value >$ pair with a single key and the list of incident edges on that vertex as the list of corresponding weights for the key. Each reducer (server) gets all edges incident on the vertex. Reduce function (reducer) adds all values for each key to produce a single value for the $< Key, Value >$ pair. This value equals the total weighted degree of a vertex.

**Example 2 : Given a sparse matrix in row major order, output same matrix in column major order**

The cells with value 0 are not stored. For a specific row as the key and the $< columnnumber, number >$ as the value, the mapper converts the column as the key and transforms the value as $< rownumber, number >$. For a specific column, shuffle aggregates the ¡row number, number¿ value into a list of values. Reducer sorts the values depending on the row number and output it in a column major order.

The goal is to process these elements using space (ideally) in poly-log of $m$ and $n$, but definitely sub-linear in $m$ and $n$. In the basic streaming setting, only a single pass over the data is allowed. The time to process each update must be low.

**Parameters to evaluate MapReduce algorithms :**

- **Space per server**
- **Number of servers used**
- **Number of MapReduce rounds**

## 2 Minimum Spanning Tree Algorithm - MapReduce framework

Given a graph G=(V,E) where $|V| = n$ and $|E| = m$, find a spanning tree of minimum weight.

**Property** In a graph G=(V,E) if an edge $e \notin MST(G)$, then there must exist a cycle C such that "e" is the heaviest edge on that cycle C.

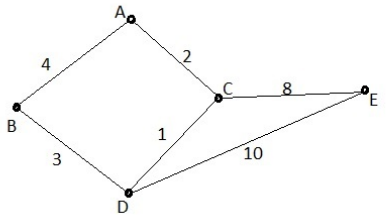


Figure 1: Graph G = (V,E)

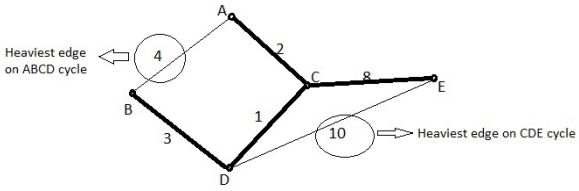


Figure 2: Minimum Spanning tree of graph G

*Proof.* Clearly, a Minimum Spanning Tree (MST) is a spanning tree with minimum weight. It does not contain any cycle. In the original graph G, there may be an edge "e" which completes the cycle. Let us consider the cycle which it completes. The edge "e" is dropped and we find another minimum edge to connect the vertices (such a vertex exists since in a cycle a vertex is connected with other vertices by two paths). Hence all the vertices are connected and "e", which is the heaviest edge on the cycle, is dropped to create a spanning tree of lower cost. □

There are two methods in which the problem can be modeled in the MapReduce framework.

**1. Vertex Partitioning :** Randomly partition the vertices to the servers and the edges get distributed accordingly. In the above diagram, A, C can be given to server 1, B to server 2 and D, E to server 3.

**2. Edge Partitioning :** Randomly distribute the edges to the servers and accordingly the vertices get distributed.

## Vertex Partitioning

$|V| = n$. There exists $l$ servers. $l = k^2$. $|V|$ is decomposed randomly into $k$ parts.

$G(i,j) = (V_i \cup V_j), E(i,j)$ where $E(i,j)$ are all the edges incident on $V_i \cup V_j$.

There are $C(k,2)$ ways of forming $G(i,j)$. Each $G(i,j)$ is send to a single separate server. $H(i,j)$ is the MST of $G(i,j)$ and this is computed on each server. We now have $H = \cup_{i,j} H(i,j)$. There exists $k^2$ sub-graphs and the MST for each sub-graph is computed.

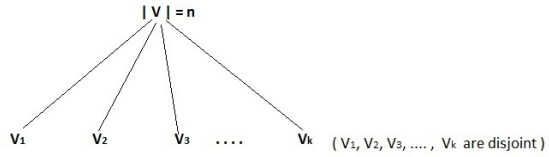


Figure 3: Vertex partitioning into k servers

Size of $|E(H)| = O(nk)$.

$|V_i| = \frac{n}{k}$ (Each vertex assigned randomly, hence average is taken) $|V_i \cup V_j| = \frac{2n}{k}$ $|E(H_{i,j})| = (\frac{2n}{k} - 1)$

We have $C(k, 2)$ or O($k^2$) of $|E(H_{i,j})|$

$$= k^2 \cdot (\frac{2n}{k} - 1) = O(n \cdot k)$$

In the final step, send H to one server. Compute MST(H) and return.

$< Key, Value >$ : $< (V_i \cup V_j)$, (Edges incident on $(V_i \cup V_j) >$

## Edge Partitioning

*Proof.* $MST(H) = MST(G)$. If this does not hold, then there must exist an edge $e \in MST(G)$ but not in $MST(H)$. $u \in V_i$ and $u \in V_i, e(u, v)$ is dropped while computing. $e \notin H_{i,j}$ but $e \in G_{i,j}$ for some i,j. This implies there exists a cycle in $G_{i,j}$ for which "e" was the heaviest. So $e \notin MST(G)$. □

**Number of rounds :**

First Pass : Send $G_{i,j}$ and compute $H_{i,j}$

Second pass : Find Spanning Tree of $H_{i,j}$

**$k^2$ servers are required.**

**Space required on each server :**

First round : Bound that can be provided on $|E_{i,j}|$

Second round: $O(nk)$

Total space required is the maximum space required among the first and second rounds.

$0 \lneq degree(v) \leq (n - 1)$ $(v \in V)$

$< Key, Value >$ : $< (E_i \cup E_j)$, (Vertices corresponding to $(E_i \cup E_j) >$

Consider,

$W_1 = v | 0 \lneq degree(v) \leq 2$,

$W_2 = v|(2^1) \lneq degree(v) \leq (2^2)$,

$W_3 = v|(2^2) \lneq degree(v) \leq (2^3)$, ....

$W_{log(n)} = v|(2^{log(n)-1}) \lneq degree(v) \leq (2^{log(n)})$

We find how many $W$ fall on $E_{i,j}$ and then find a bound on $E_{i,j}$.

$|E_{i,j}| \leq E[x] \leq \sum_{v \in V_i} degree(v) + \sum_{v \in V_j} degree(v)$

Consider any particular group $W_i$.Each vertex goes to one of the partition with probability $\frac{1}{k}$.

$W_i = v|(2^{i-1} \lneq degree(v) \leq (2^i)$. If $|W_i| \leq (k \cdot log(n))$, we leave it. Else if $|W_i| \gneq (k \cdot log(n))$, for each $v \in W_i$, $V_j$ goes to $W_i$ with probability $\frac{1}{k}$. On expectation, the number of vertices from $W_i$ which goes to $V_j = \frac{|W_i|}{k}$, $\frac{|W_i|}{k} \geq log(n)$. Hence we can apply Chernoff bound. Deviation form expectation is high with probability $\lneq e^{-expectedvalue} = e^{-log(n)} = \frac{1}{n}$.

The expectation value denotes the number of vertices that fall on $V_j$. It is important to find the large value of deviation on expectation.

**Space Requirement :**

$$O(\sum_{i=1}^{log(n)} \frac{|W_i| \cdot (2^i)}{k}), |W_i| \leq n$$

$$= O(\sum_{i=1}^{log(n)} \frac{|2m| \cdot (2^i)}{(2^{i-1})k}$$

$$= O(\frac{m \cdot log(n)}{k})$$

Total degree in G = 2m. For $W_i$, minimum degree of vertices in $W_i \gneq (2^{i-1})$

$$|W_i| \cdot (2^{i-1}) \leq 2m$$

$$|W_i| \leq \frac{2m}{2^{i-1}}$$

Total degree of vertices in $V_i$ similar to total degree of vertices in $V_j$

$$E_{i,j} = O(\frac{m \cdot log(n)}{k}) = \sum_{v \in V_i} degree(v) + \sum_{v \in V_j} degree(v)$$

**Number of rounds** = 2

**Number of servers** = $k^2$, $k = n^{\frac{c}{2}}$ where $m = n^{1+c}$, $0 \lneq c \lneq 1$.

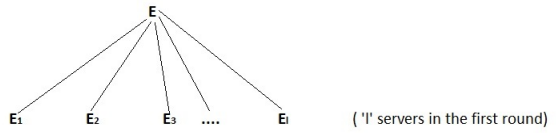


Figure 4: Edge partitioning

This is called a c-sparse graph. $k^2 = n^c$ servers.

**Space per server :**

$$max(O(\frac{m \cdot log(n)}{k}), O(n \cdot k))$$

$$= max(O(\frac{n^{1+c} \cdot log(n) \cdot n^{c/2}}{k}), O(n \cdot k))$$

$$= O(n^{1+\frac{c}{2}} \cdot log(n), 0 \lneq c \lneq 1$$

So the number of servers is sub-linear in n and for each server space is $O(n^{1+\frac{c}{2}})$. Space requirement is also sub-linear in the number of edges as we can omit log(n) factor.

$$(n^c) \cdot (n^{1+\frac{c}{2}})$$

$$= n^{\frac{3c}{2}}$$

$$\gneq n^{1+c}(Numberofedges)$$

We would ideally like to store only the edges and hence have space proportional to the number of edges. So edge partitioning is used.

**Space per server** $= n^{1+\epsilon}, \epsilon \gneq 0$

**Number of servers** $= n^{c-\epsilon}$

$m = n^{1+\epsilon} \cdot n^{c-\epsilon} = n^{1+c}$

This barely fits all the edges in the overall server space.

The number of edges are reduced significantly. There were $n^{1+c}$ edges. After dividing it into $n^{c-\epsilon}$ servers, in each server $(n-1)$ edges remain. So, after first round, total number of edges that remain is nearly equal to $n^{c-\epsilon} \cdot n = n^{1+c-\epsilon}$.

Similarly, after second round, total number of edges remaining nearly equals $n^{c-2\epsilon} \cdot (n-1) = n^{1+c-2\epsilon}$.

**Algorithm 1** Edge partitioning

Set $|E'| = E, l' = l$
Partition E' into l' servers such that each server has $n^{1+\epsilon}$ edge at most. Else we keep on throwing edges.
Compute $H_i = \text{MST}(E_i)$ on server "i"
Combine all spanning trees computed.
$H = \cup H_i$
$E' = E(H), l' = \frac{E(H)}{n^{1+\epsilon}}$
**if** the summation of the number of edges fits into a single server **then**
  RETURN
  **if** $|E'| \gneq n^{1+\epsilon}$ **then**
    repeat from the beginning
  **else**
    Compute MST(H)
    RETURN
  **end if**
**end if**

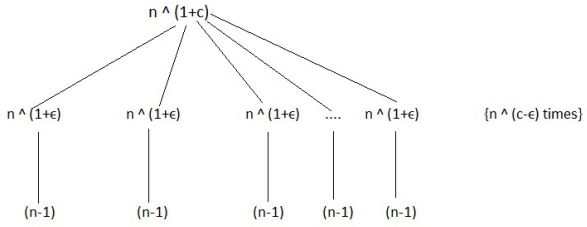


Figure 5: First round of reducing vertices

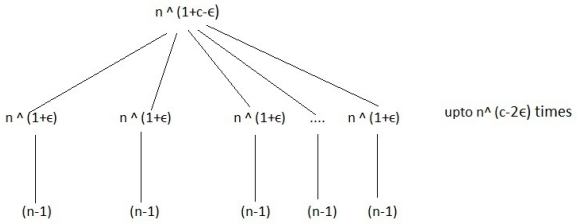


Figure 6: Second round of reducing vertices

After $\frac{c}{\epsilon}$ rounds, number of edges remain $= n^{1+c-\frac{c}{\epsilon}\cdot\epsilon} = n$. Now this will fit into the memory and we can compute the MST.

**Rounds** $= \frac{c}{\epsilon}$.

**Number of servers** $= n^{c-\epsilon}$.

**Space on each server required** $= n^{1+\epsilon}$.

In edge partitioning, number of edges is reduced by $n^{\epsilon}$. So $\frac{c}{?\epsilon}$ rounds are required. In vertex partitioning, number of edges is reduced by $\frac{n}{2}$. So $log(n)$ rounds are required.

# References


[1] Howard Karloff, Siddarth Suri, Sergei Vassilvitskii. A Model of Computation for MapReduce. In SODA 2010 (Austin, Texas)

[2] Bahman Bahmani, Ravi Kumar, Sergei Vassilvitskii. Densest Subgraph in Streaming and MapReduce. In VLDB 2012

[3] Graph sparsification in the semi-streaming model, Kook Jin Ahn, Sudipto Guha