

1.

Checkpoint on slide 11:

```
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Checkpoint on slide 55:

```
(SparkCamp,4)
(Spark,3)
(spark,1)
(SparkSQL,1)
(..../spark/bin/spark-submit,1)
```

Code for slide 60:

```
val rdd1 = sc.textFile("README.md").filter(_ contains "Spark")
val rdd2 = sc.textFile("spark/docs/contributing-to-spark.md").filter(_ contains "Spark")
val wc1 = rdd1.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
val wc2 = rdd2.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
val joined = wc1.join(wc2)
```

Checkpoint on slide 60:

```
(Spark, (3,2))
```

2.

The mapper emits one key-value output pair for every input pair of vertices: the output key is the *sorted* vertices and the output value indicates the direction of the edge. The mapper takes directed edge  $\langle a, b \rangle$ : if  $a < b$ , it emits  $\langle (a, b), 1 \rangle$  and if  $a > b$ , it emits  $\langle (b, a), 2 \rangle$ .

The reducer sees all the values  $v_1, \dots, v_n$  for a given key  $(c, d)$ . If  $\{1, 2\} \subseteq \{v_1, \dots, v_n\}$  then it emits  $(c, d)$ ; otherwise it emits nothing.

No combiner is used; combiners are not likely to help in this problem.

---

**Algorithm 1** Map

---

```
function MAP( $\langle a, b \rangle$ )
  if  $a < b$  then
    Emit  $\langle (a, b), 1 \rangle$ 
  else
    Emit  $\langle (b, a), 2 \rangle$ 
  end if
end function
```

---

---

**Algorithm 2** Reduce

---

```
function REDUCE(Key  $(c, d)$ , Values  $\{v_1, \dots, v_n\}$ )  
  if  $n < 2$  then return  
  end if  
  for  $i = 2, \dots, n$  do  
    if  $v_i \neq v_{i-1}$  then  
      Emit  $\langle c, d \rangle$   
    return  
  end if  
  end for  
end function
```

---

3.

The combiner is the same as the reducer. Let  $N$  be the total number of words.

---

**Algorithm 3** Map

---

```
function MAP(String  $s$ )  
  for Word  $w$  in  $s$  do  
    Emit  $\langle w, 1 \rangle$   
  end for  
end function
```

---

---

**Algorithm 4** Reduce/Combine

---

```
function REDUCE(Key  $w$ , Values  $\{v_1, \dots, v_n\}$ )  
   $s \leftarrow 0$   
  for  $i = 1, \dots, n$  do  
     $s \leftarrow s + v_i$   
  end for  
  Emit  $\langle w, s \rangle$   
end function
```

---

*Without combiners*— The shuffle size is  $N$ , and the reduce takes  $N \pm O(B)$  operations.

*With combiners*— After the combine step, there are at most  $k$  key-value output pairs, since there are at most  $k$  distinct keys. The shuffle size is  $kB$ , and the reduce takes  $kB \pm O(B)$  operations.

4.

Let me briefly state the naive solution, which does not parallelize effectively. Run one map-reduce to count the number of elements  $N$ . Next, map each input pair  $\langle i, a_i \rangle$  to  $N-i+1$  output pairs  $\langle i, a_i \rangle, \langle i+1, a_{i+1} \rangle, \dots, \langle N, a_N \rangle$ . Reduce by summing all values for a given key. With combiners, the shuffle size is  $N$ , and the number of reduce operations is  $NB$  where  $B$  is the number of mappers. The problem with this solution is that each mapper requires  $O(N)$  storage to hold the output keys—but this is on the same order as the size of the entire data.

A better idea is to use divide-and-conquer. The idea is to divide the key-set into  $B$  equally-sized partitions:  $P_1 = \{1, \dots, n_1\}, P_2 = \{n_1+1, \dots, n_2\}, \dots, P_B = \{n_{B-1} + 1, \dots, N\}$ . Accordingly define

$$\phi(i) = b \text{ such that } i \in P_b.$$

Then define the partial sums  $p_1, \dots, p_B$  by

$$p_b = \sum_{i \in P_b} a_i,$$

and define quantities

$$u_b = \sum_{c < b} p_c.$$

For  $i = 1, \dots, n$  define the within-partition prefix sums as

$$t_i = \sum_{j \in P_{\phi(i)}: j \leq i} a_j$$

Now observe that

$$s_i = t_i + u_{\phi(i)}$$

Therefore the procedure is as follows

1. (Map/Reduce 1) Count the number of keys  $N$
2. (Map 2) Input: the original key-value pairs. Partition the keys sequentially into  $B$  workers
3. (Reduce 2) Each worker  $b = 1, \dots, B$  computes  $p_b$  and sends it to the driver
4. The driver computes  $u_b = \sum_{c < b} p_c$  and sends  $u_b$  to each worker  $b$  for  $b = 1, \dots, B$ .

5. (Reduce 3) Input: the output of step 2. Each worker  $b = 1, \dots, B$  computes  $s_i = u_b + t_i$  and emits  $\langle i, s_i \rangle$  for each  $i \in P_b$ .

To formalize this procedure in the map/reduce framework we have to designate the partition number  $b$  as a key throughout steps 2-5. In steps 2 and 5, we have  $(i, a_i)$  as values. Note that step 5 uses the same input as step 3.

---

**Algorithm 5** Step 2: Map 2

---

Parameters  $n_1, \dots, n_{B-1}$  determined in Step 1, and  $n_B = N$ .

**function** MAP( $\langle i, a \rangle$  from original inputs)

**for**  $b \in 1, \dots, B$  **do**  
   **if**  $n_b > i$  **then**  
      Emit  $\langle b, (i, a) \rangle$   
      **return**  
   **end if**  
  **end for**  
**end function**

---



---

**Algorithm 6** Step 3: Reduce 2

---

**function** REDUCE(Key  $b$ , values  $(i, a)$  from step 2)

$p \leftarrow 0$   
  **for**  $(i, a)$  in values **do**  
     $p \leftarrow p + a$   
  **end for**  
  Emit  $\langle b, p \rangle$   
**end function**

---



---

**Algorithm 7** Step 5: Reduce 3

---

Parameters  $u_1, \dots, u_B$  computed by driver in step 3.

**function** REDUCE(Key  $b$ , values  $(i, a)$  from step 2)

  Sort values  $(i, a)$  by  $i$   
   $s \leftarrow u_b$   
  **for** Value  $(i, a)$  in sorted list **do**  
     $s \leftarrow s + a$   
    Emit  $\langle i, s \rangle$   
  **end for**  
**end function**

---

The cost of the computation is dominated by step 2, when the data is partitioned: this requires a shuffle size of  $N$ . This is followed by a reduce in step 3 requiring  $O(N/B)$  operations and  $O(1)$  space. The driver has to complete  $O(B)$  operations in step 4. Finally, each worker has to complete  $O(N/B)$  operations in step 5, requiring  $O(1)$  memory. The overall number of Map/Reduce iterations is 3, including the initial count.

## 5.

a. Let  $A$  be the adjacency matrix. Then the  $i, j$  entry of  $A^2$  is nonzero if and only if there is a path of length 2 from vertex  $i$  to  $j$ . Therefore the procedure is to evaluate  $A^2$ , which costs  $O(n^{2.376})$  operations, and check the  $n(n-1)/2$  off-diagonals, which costs  $O(n^2)$  operations.

b. Let  $b = \lfloor n/r \rfloor$ , and write  $A = [A_1, \dots, A_b]^T$  where  $A_1, \dots, A_{b-1}$  are  $r \times r$  and  $A_b$  is  $r \times (n - (b-1)r)$ . Also write  $B = [B_1, \dots, B_b]$  where  $B_1, \dots, B_{b-1}$  are  $r \times r$  and  $B_b$  is  $r \times (n - (b-1)r)$ . Then  $C = AB$  is composed of blocks  $C_{i,j} = A_i^T B_j$ . The cost to compute  $C_{i,j}$  is the cost to multiply  $A_i^T$  and  $B_j$ , which is bounded by  $O(r^{2.376})$ . Meanwhile, there are  $b^2$  such blocks, where  $b^2 = O((n/r)^2)$ . Therefore the cost is  $O((n/r)^2 r^{2.376})$ .

c.

Let  $m = n^q$ . Then our goal is to provide an algorithm with runtime  $O(m^{.55} n^{1.45}) = O(n^{1.45+0.55q})$ .

In the proof we will make use of the following Lemma:

**Lemma 1** *Let  $x = (x_1, \dots, x_n)$ , such that  $x \geq 0$ ,  $\sum_i x_i \leq m$ , and  $\max_i x_i \leq t$ . Then*

$$\|x\|^2 \leq mt$$

**Proof.** Since  $x \geq 0$ , we have  $\sum_i x_i = \sum_i |x_i| = \|x\|_1$  and also  $\max_i x_i = \max_i |x_i| = \|x\|_\infty$ . Now apply Holder's inequality:

$$\|x\|^2 \leq \|x\|_1 \|x\|_\infty \leq mt$$

We first deal with the case  $q < 1$ . Then there exists at least one node with no edges, hence the graph is not shallow. For any  $\epsilon > 0$ , it is possible to distinguish the case  $q < 1 - \epsilon$  from the case  $q \geq 1$  with high probability by computing the proportion of  $O(n)$  sampled vertex pairs which have edges. Since  $O(n) < O(n^{1.45+0.55q})$  the cost of this statistical test is a non-issue. Henceforth we assume  $q \geq 1$ .

We will define an algorithm with a parameter  $p > 0$ . Given any  $q \geq 1$ , we claim that there exists  $p > 0$  such that the algorithm has complexity

$O(n^{1.45+0.55q})$ . Note that for any  $q \geq 1$ ,  $O(n^{1.45+0.55q}) \geq O(n^2)$ , so checking all pairs of vertices is free.

We define the algorithm as follows. Order the vertices in descending order by degree. Let  $1, \dots, n_h$  denote all the vertices with degree larger than  $n^p$ . Since the number of edges is twice the sum of the degrees, we have  $n_h < n^{q-p}$ . Let  $A$  be the adjacency matrix of the graph, and write  $A = [A_1 A_2]$  where  $A_1$  is  $n \times n_h$  and  $A_2$  is  $n \times (n - n_h)$ . By construction, each column of  $A_2$  has at most  $n^p$  entries. We will compute

$$A^2 = AA^T = A_1 A_1^T + A_2 A_2^T$$

where  $A_v$  is the  $v$ th column of  $A$ .

To compute  $A_1 A_1^T$ , apply the result of part *b* with  $r = n_h < n^{q-p}$ . Therefore the cost to compute  $A_1 A_1^T$  is  $O(n^{2+0.376q-0.376p})$ .

To compute  $A_2 A_2^T$ , note that

$$A_2 A_2^T = \sum_{v=n_h+1}^n A_v A_v^T$$

Now let  $S$  denote a  $n \times n$  matrix initialized to 0. For each  $v = n_h + 1, \dots, n$ ,  $A_v A_v^T$  is a matrix of zeros and ones, with exactly  $d_v^2$  ones. Therefore, we can compute  $A_2 A_2^T$  by iterating over all  $v = n_h + 1, \dots, n$  and incrementing the  $d_v^2$  entries of  $S$  which correspond to nonzero entries of  $A_v A_v^T$ . In this way, the cost of computing  $\sum_{v=n_h+1}^n A_v A_v^T$  is  $\sum_{v=n_h+1}^n d_v^2$ . Applying the lemma we have  $\sum_{v=n_h+1}^n d_v^2 \leq 2mn^p$ , so the cost of this step is  $O(n^{p+q})$ .

Now we will prove the claim. Given  $q \geq 1$ , choose  $p = 1.45 - 0.45q$ . Then the cost of computing  $A_1 A_1^T$  is

$$O(n^{2+0.376q-0.376p}) = O(n^{1.45+0.55q})$$

and the cost of computing  $A_2 A_2^T$  is

$$O(n^{p+q}) = O(n^{1.45+0.55q})$$

Having computed  $A^2 = A_1 A_1^T + A_2 A_2^T$ , it remains to check the off-diagonals of  $A^2$ . But this takes  $O(n^2)$  operations, which is less than  $O(n^{1.45+0.55q})$  for any  $q \geq 1$ .

Hence the cost of the whole procedure is  $O(n^{1.45+0.55q}) = O(n^{1.45} m^{0.55})$ , as claimed.