CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 16, 5/18/2016. Scribed by Milind Rao.

16 Lecture

In this lecture, we go over

1. Distributed Sorting
2. Partitioning - cleverly storing the RDD on different machines to lower the communication costs associated with operations like `Join` and `reduce` which need all-to-all communication patterns.
3. Pregel - Brief look at an alternate *bulk synchronous dataflow paradigm*.

16.1 Sorting

The sorting problem is defined as follows:

- The data starts out distributed on p machines and there is some ordering present between the data points that we sort on.
- All-to-All communication is unavoidable and the best we can hope for is that the data points are sent over the network only once.
- The sorted dataset is stored in a distributed manner as well.

This is need in the Shuffle step of distributed algorithms.

16.1.1 Algorithm

1. Each machine sends a uniform sample of its elements to the driver. Streaming algorithms (covered in Lecture 9) allow each machine to efficiently compute a uniform sample and the number of data points it has.
2. The driver uses the samples from each machine weighted by the number of data points the machine has to compute an approximate histogram or distribution of the data points to be sorted. It then uses this to compute p split-points where the number of data points between two consecutive points is approximately the same. The final result will have all data points between d_{i-1} and d_i in machine i. This is highlighted in Fig. 1. These split points are broadcast to the nodes.

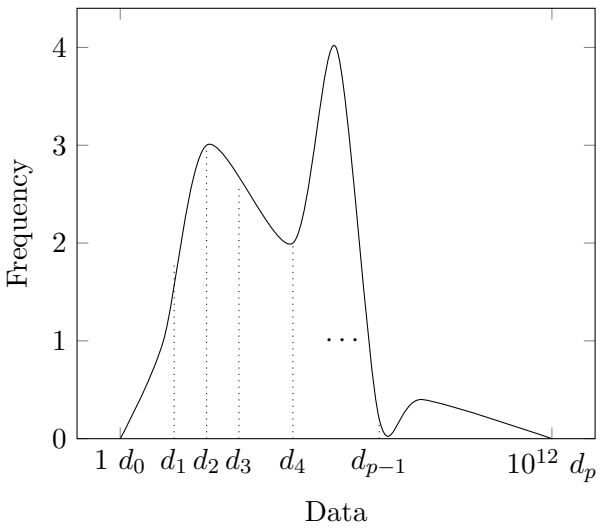

Figure 1: Example histogram or pdf of the data points from the samples sent by p machines and the split points $\{d_i\}_1^p$ computed.

3. Each machine does a local sort (eg. TimSort or variants of QuickSort dealing with loading data from harddisk).
4. Using splitpoints, each machine builds index of which data point goes to which machine.
5. Machine i asks machine j for its portion - machine j does efficient index lookup and can stream directly from harddisk to the network as the sorted datapoints are stored contiguously on harddisk. All-to-all communication occurs in this step.
6. Each machine does a p way merge of p different sorted data points that it has received from each of the machines. This is similar to the two-way merge seen in lecture 4.

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Figure 2: Performance of distributed sorting in Spark

In this algorithm, we have seen that the best algorithms for local sorting are used and data is indeed sent on the network only once reinforcing the belief that this is an efficient approach. Asymptotics may not as useful for practising engineers as empirical evidence.

In Fig. 2, we see that Spark with distributed TimSort and the algorithm we have described can sort at the rate of 4.27 TB/min and 20 GB/min/node which set a record. This needed an optimized network as the network turned out to be a bottleneck.

16.2 Partitioning

The motivation for partitioning arises from the fact that sending data via the network is expensive. The hope is that smartly partitioning an RDD in its constituent machines will reduce the need for communication. We highlight the need for partitioning by highlighting an example through the *PageRank* algorithm.

16.2.1 Sparse Pagerank

The problem is to compute node importance given

- RDD of sparse graph `links` which stores the pair inscribing directed edges.
- RDD storing the rank of the nodes. In this situation, we consider guess for node importance which we refine through iterations.

An application is computing importance of urls (nodes) in the internet(graph). One way is to compute the topmost eigenvalue of the adjacency graph matrix A. Let x_t be the guess of the rank. One way to compute this:

$$x_{t+1} \leftarrow Ax_t.$$

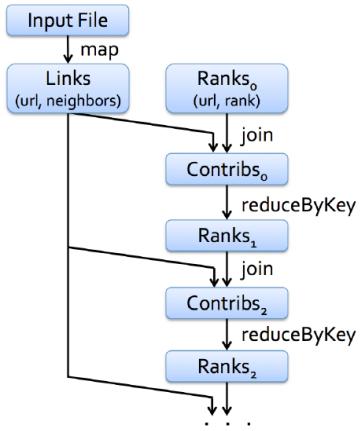

Figure 3: Pagerank without partitioning incurs multiple all-to-all communication from repeated joins.

Here, it is assumed that entries of the adjacency matrix are 1 if there is an outgoing link and this is normalized by the number of neighbors the node has. We use the following modification:

$$x_{t+1} \leftarrow 0.15\mathbf{1} + 0.85Ax_t.$$

In other words,

1. Each page (node) is started with rank 1. Or x_t is initialized with the uniform vector.
2. On each iteration page p computes its contribution $(x_t)_p/|\text{neighbors}_p|$ and sends it to its neighbors.
3. Each page's rank is $0.15 + 0.85 \times$ contribs. We are implementing this matrix-vector multiplication using `join` and aligning the RDDs.

The algorithm is now written:

```
val links =  // RDD of (url, neighbors) pairs
var ranks =  RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
        val contribs = links. join(ranks). flatMap {
                case (url,(links,rank)) => links.map(dest => (dest,rank/links.size))
        }
        ranks = contribs. reduceByKey(_+_). mapValues(.15+0.85*_)
}
```

The parts in red are local operations and the parts in blue are distributed operations. *All-to-all* communication is required in the `join` and `reduceByKey` operations. As can be seen from Fig. 3, joins necessitating all-to-all communication are required for each iteration.

We *partition* the links such that all links within the same partition sit on the same machine. This can be done with the following line-

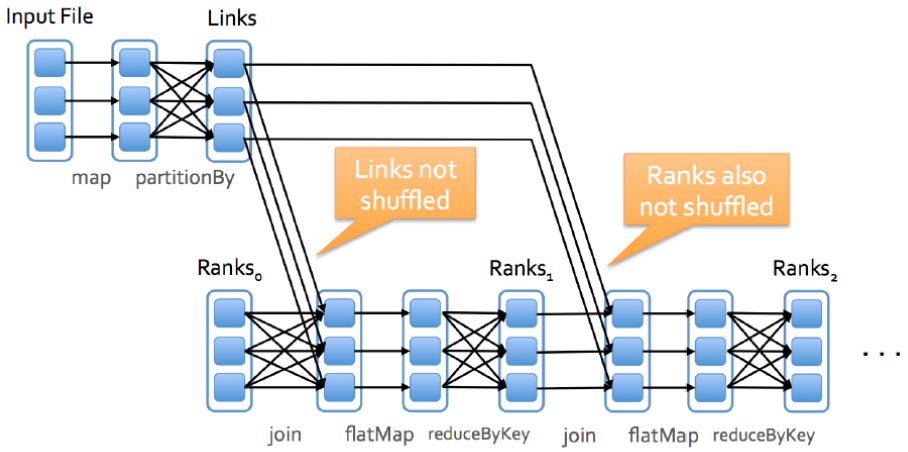

Figure 4: With partitioning, repeated joins does not happen.

```
val links = sc.textfile(...)....partitionBy(new HashPartitioner(8))
```

Any shuffle operation (such as one with join) will respect the partitioner if set. As can be seen from Fig. 4, new ranks after the first iteration sit in the same machine as the old links and repeated joins do not occur.

A note of caution here is that `map` may change key values and partitioning is not respected. If knowledge of partitioning is to be preserved, `mapValue` can be employed.

Further communication can be cut down from domain knowledge. For example, we know that most links from a webpage are to another page with the same domain name. Thus we can partition all webpages with the same domain in the same machine.

16.3 Pregel

Pregel, like MapReduce and Spark, is a *bulk synchronous dataflow paradigm* but is adapted for graph operations. Synchrony comes from the fact that map and reduce operations occur at the same time in the distributed computation network. In Pregel too, the programming interface is restricted so that the system can do more automatically.

In Pregel, the key difference is that nodes send messages to neighbors and update their state based purely on its state and messages received from neighbors. This dataflow paradigm is illustrated in Fig. 5.

Pregel was motivated by excessive communication in MapReduce. Pregel is implemented in Spark through GraphX. Separate RDDs are needed for graph state, vertex states and messages at each iteration. `groupByKey` is needed to perform each state. This is however not the most efficient implementation of Pregel. A discussion of cases where scaling does not help is described in [1].

We note that Pagerank is easily implemented in Pregel as the message computed in each iteration is the contributions to its neighbours and state at each node is its rank. An example of Pregel for finding connected components is:

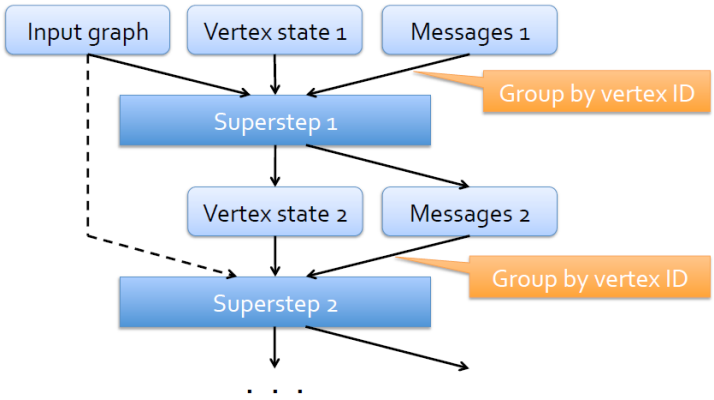

Figure 5: Pregel dataflow paradigm.

1. The state at each node is initialized to be its nide id.
2. The message a node sends to its neighbors in each iteration is its state
3. State update is done by taking the minimum of its state and the messages received from all neighbors.
4. After n (number of nodes) iterations, if all nodes have the same state, the graph is connected.

References

[1] F. McSherry, M. Isard and D. Murray. *Scalability! but at what COST?*. 15th Workshop on Hot Topics in Operating System (HotOS XV).