# 6.814 - Lecture 19; *Dynamo Recap, MapReduce & Spark*

Long Nguyen

November 15, 2017

## 1 Topics

Last time, we talked about Dynamo, Amazon's NoSQL system that trades consistency for availability, in accordance with Amazon's SLAs and services. Dynamo is a distributed key-value store that makes use of consistent hashing (storing data in a ring). Data is eventually consistent, that is, eventually replicated towards all $N$ nodes (overlay network).

In joining the ring, admins add or remove nodes, and other nodes broadcast/gossip about the structure of the ring at ping time. Any keys a new node $j$ is responsible are sent from a node $i$.

Dynamo makes use of quorum reads and writes, where we have $N$ replicas, and we write to $W$ and read from $R$ replicas. If $R + W > N$, then all reads will see $> 1$ copy of the most recent write.

Suppose successors aren't available. We can then use sloppy quorum and hinted handoff to write to some other node, which keeps track of the actual successor nodes the keys/values were meant for. Once those actual successors come back up, we can ping and send the data over. This gives more durability, but in partitioned networks, it can lead to divergent replicas.

Today, we'll talk about other distributed systems like Spark and Hadoop.

## 2 Spark & Data Analytics

### 2.1 Distributed dataflow

These systems include MapReduce and Spark.

**MapReduce:** We have a collection of partitions (GFS). We apply some `map` operation to them, producing intermediate outputs as key-value pairs. Then, we `reduce` these outputs by key. For these stages, we have $x$ map workers and $y$ reduce workers. Each reduce worker takes responsibility for a specific key and its values from the intermediate outputs. MapReduce is similar to `SELECT key, agg(value) GROUP BY key FROM (SELECT f1(v1), f2(v2) FROM t WHERE p)`. MapReduce has good midquery fault tolerance, because if some mapper fails, the master can assign the task to another worker.

In contrast, typical databases don't implement this midquery fault tolerance, because it's likely that we're not running thousand-node clusters that potentially work 24/7.

Spark is, in essence, a better MapReduce.

## 2.2 Spark

Spark provides distributed operations that run over datasets. Like MapReduce, Spark uses distributed file systems like HDFS (Hadoop File System). Data is partitioned, but we think of the entire dataset across all partitions as a single file.

**Spark operations:**

- `map(f): [T] -> [U]`
- `filter(f: T -> bool): [T] -> [T]`
- `groupBy, reduceBy, union, join`
- other operations as well.

The key idea we want here is **resilient distributed datasets**: if data in the system is immutable, then instead of writing intermediate results out and doing operations, why not just rerun tasks from the original, untouched file system? Instead of materializing intermediate outputs, let's just store the operations we ran for reference, and only persist to memory/disk when the user wants to.

With RDDs, we make use of lineage graphs, which give us DAGs of operations done on the dataset. This saves us from significant overhead associated with MapReduce (I/O, random access, etc.).

## 2.3 Dependencies

In Spark, we can have **narrow** or **wide** dependencies.

Using dependencies, we can optimize how we want to parallelize our tasks (the scheduler takes care of this). Narrow tasks can be done locally and independently, but wide tasks have to wait for their inputs from possibly multiple partitions.

## 2.4 Scheduler

The scheduler decides 1) what to run next and 2) where to run it. We have ready-to-run tasks that we can choose from, and from these, we should run tasks with data already cached in memory. This avoids eviction problems and removes overhead of disk spilling. We'd also like to run these tasks with locality exploitation (scheduling).

We have a buffer pool that uses an MRU eviction policy, since the most recently accessed data was probably just used for some task and will probably not be used again in the near future.