COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 5: Sept. 20th

Lecturer: Emery Berger *Scribe(s): Lakshmi Rajan, Timm Allman*

5.1 Master-Slave Architecture

Consider a distributed system that uses 1000 computers for performing tasks. If the chance of any one computer failing during a given computation is 1/1000 then the probability that all the systems succeed is (1 - 1/1000)^1000 or about 0. This means that the probability of at least one failure is 1 - (1 - 1/1000)^1000 or about 100%.

A master-slave architecture is common solution to this problem. It deals with failure by having a leader (the master) and some number of workers (the slaves). The leader divides tasks and assigns them to the workers. If the workers are slow to finish their tasks, or fail entirely, the leader reassigns those tasks to other workers so that all the work will get done (fault tolerance). The only single point of failure is if the leader: if any of the workers die the leader will take care of it, if the leader dies there is noone to take over.

While beyond the scope of the class there are ways to elect a new leader. If interested look up: Byzantine faults/fault tolerance - Model of how failures happen and how to deal with them. Leslie Lamport - Computer scientist who wrote a bunch about this. PAXOS/RAFT - two different leader election algorithms.

5.2 MapReduce

The map-reduce paradigm has been around since Lisp. It consists of two operations: `map(f, list) -> list` and `reduce(g, list(value)) -> value` with f and g being functions. f is a unary operator that can be run in parallel on all elements of the list (it only deals with one element at a time and should be idempotent) and g is a binary operator that is (almost always) associative and commutative so the order it is applied to the list doesnt matter.

MapReduce (MR) frameworks are more recent. The user gives the MR framework two functions: `Map(key, value) -> (key, value)` and `Reduce(key, list(value))`. Map is run once for each item in a set and Reduce is run once for each unique key that Map emits.

MR uses a master-slave architecture to parallelize map and reduce over many machines. Many map and reduce tasks are runs in parallel across many systems. Because Map is idempotent (no side-effects) and stores its output into a private file it is easy for the leader to kill and reschedule map tasks without any side-effects. Similarly since Reduce is run only once per unique key any such task can also be easily killed and rescheduled.

Reduce is always preceded by sort operation. This is done for the following reasons:

- Sort ensures that all the runs produce the same output.
- Results are deterministic. Hence, if an entire job is executed on one machine, or divided into tasks and executed on multiple machines, the results would be the same. This enables developers to debug and test the code locally.

Applications of MapReduce include grep, reversible web-link graph, and inverted index.

Example: Inverted Index

```
Map(Url, contents) -> list(word:url)
Reduce(word, list(URLs)) -> list(word:urls)
```

Map(A, "the sandwich is delicious") -> the:A; sandwich:A; is:A; delicious:A
Map(B, "the pizza is delicious") -> the:B; pizza:B; is:B; delicious:B
Map(C, "the beer is great") -> the:C; beer:C; is:C; great:C

Reduce(the, list(A, B, C)) -> "the":"A,B,C"
Reduce(sandwich, list(A)) -> "sandwich":"A"
Reduce(is, list(A,B,C)) -> "is":"A,B,C"
Reduce(delicious, list(A,B)) -> "delicious":"A,B"
Reduce(pizza, list(A)) -> "pizza":"A"
Reduce(beer, list(C)) -> "beer":"C"
Reduce(great, list(C)) -> "great":"C"

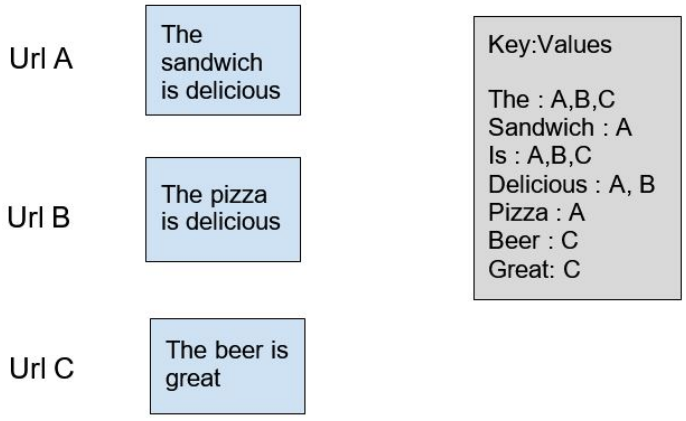

Figure 5.1: Example illustrated