**Week 9: Distributing and Replicating Data**

**Overview**

So far, we've only considered services that run on a single compute node.  These solutions have several weaknesses:

* Single point of failure (SPOF): if the node fails, the database will become unavailable.
* Data set size limited to the capacity of the single node
* Number of clients that can be served is limited by the capability of the single node

Traditional RDMSs like PostgreSQL have implemented the capability to replicate the database across multiple nodes to address the SPOF issue.  These replicated setups are usually configured with only one or two secondary compute nodes (for a total of three nodes).  All writes are issued by clients connected to the primary server; the primary server then replicates the writes to the secondary compute nodes.  Should the primary server failure, one of the secondary servers can be promoted to the primary server and clients can connect to it instead.  This is called active-passive replication.  The data set sizes are stilled limited by what can be stored on a single server since data are not partitioned across servers but rather the entire data set is replicated to the secondary servers.  Similarly, the active-passive replication configuration does not provide any load balancing since clients still need to connect to the primary server.

More recently, these systems have begun to support a replication mode (active-active) where all of the servers are considered equal peers.  The active-active approach allows clients to connect to any server in the cluster, allowing some load balancing.  I say "some" because the clusters are still limited (in practice) to a maximum of 2-3 nodes.

NoSQL databases paved the way for distributing data across large clusters of nodes.  Early NoSQL databases achieved this by giving up strict guarantees of consistency, support for transactions, and/or switching to append-only data structures.  Examples of these systems include the Hadoop Distribution File System, the Apache Cassandra database, and the Voldemort distributed key-value store.  Early on, it was assumed that it wasn't feasible to create large, distributed databases with the SQL data model, because of its strict ACID guarantees.  Later on, however, engineers realized that you the SQL query and the relational data model do not automatically imply strict consistency guarantees.  Apache Hive and Presto (created by engineers at Facebook) are examples of read-only SQL query engines that can be used on data sets too large to fit on a single machine and employ parallelism for faster query processing.  Other SQL query engines such as Apache Spark allow writes but use an append-only model and do not support transactions (meaning, multiple users writing at once).  You'll dive more into these technologies in the last part of this class when you research, analyze, and present on them.

In this module, we're going to learn about strategies for distributing data across multiple server processes.  We'll start by discussing rendezvous hashing, a technique for determining which keys are stored on which servers.  The primary advantage of hashing techniques is that clients can compute the same partitioning rules with no communication -- they only need the same input data.  In future readings, we'll discuss the Consistency-Availability-(Network) Partitioning (CAP) theorem and the implications for tradeoffs when designing and implementing distributed systems.  Lastly, we'll cover a few algorithms used maintaining operation in the case of failure.

**Reflection Questions**

* What does it mean to partition data? What does it mean to replicate data? What are the reasons for doing these two things?
* What are round-robin, hash, and range partitioning? What are their advantages and disadvantages?
* What is skew?
* What is dynamic repartitioning and why is it a desired ability?
* What is a distributed hash table?
* What are the reasons for using replication and the challenges introduced by replication?
* What are some of the use cases of CHORD? What is it trying to achieve?
* Explain the structure of a CHORD cluster. Do all nodes know about each other? Or do nodes all only know about some of the other nodes?
* What happens to the data if a node leaves or joins the cluster?
* Explain consistent hashing. How is a value assigned to a node?
* What are the requirements specified by Thaler and Ravishankar?
* What are the tradeoffs of the various approaches they evaluated?
* What is rendezvous hashing? How does it compare with consistent hashing? Is it trying to achieve the same goals? Does it?