DSC-204A: Scalable Data Systems, Winter 2024

15: Parallelism

Lecturer: Hao Zhang | *Scribe: Albert Zhong*

1 3 Paradigms of Multi-Node Parallelism Implementations

- 3 paradigms: shared nothing, shared disk, shared memory

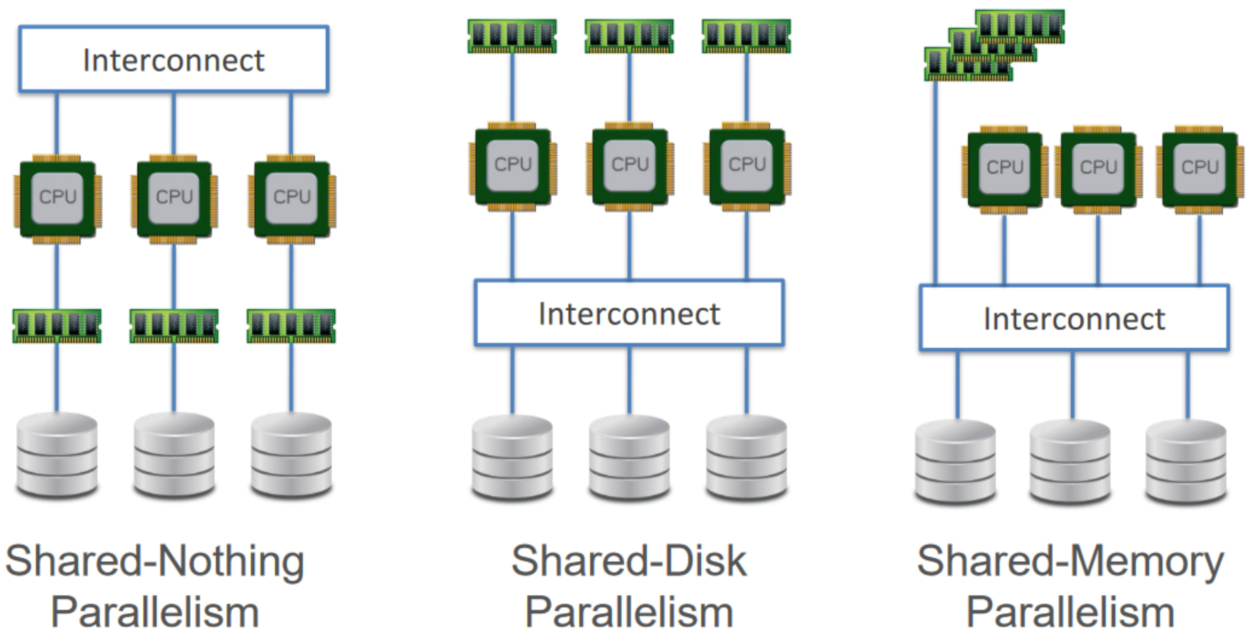

Figure 1: The 3 Paradigms

1.1 Shared Nothing Parallelism(Horizontal Scaling)

- Most popular
- Each node uses its CPUs, RAM, and disks independently
- The most vanilla but also most complex system
  - Need to somehow maintain consistency, communication, coordination
- Advantages: performance, cost
- Disadvantages: complexity, involves many constraints and trade offs
- Database cannot hide any of the issues involved from you

2 Metrics to Evaluate Distributed Big Data Systems

- Scalability - data volume and read/write/compute speed
- Consistency and correctness - read/write sees consistent data, computations produce the correct results
- Fault tolerance/high availability - when one system fails (there is always a chance for failure, which only increases as you add more machines), another can take over
- Latency - distribute machines worldwide and reduce network latency

3 Problems Distributed Systems Need To Solve

- Communication
- How to distribute data?
- How to distribute computations?
- How to coordinate/synchronize?

3.1 How to distribute data

- One of the problems distributed systems need to solve
- Two methods: replicate/partition the data

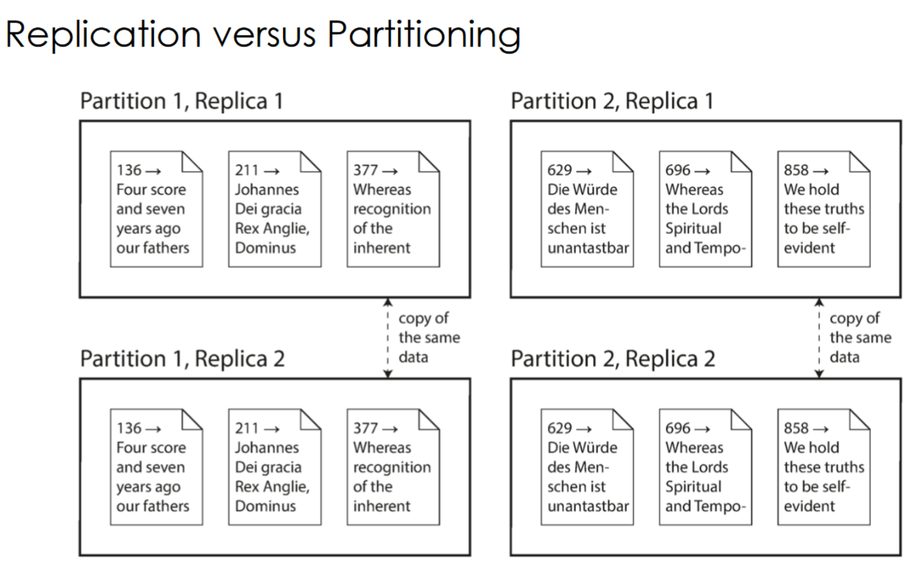

Figure 2: Replication vs Partitioning

- Key assumptions:
  - We are using a distributed data or database system

- The computations are lightweight (like a SQL query or light computation)
- Core challenge - how to deal with distributed data
- These mainly apply to old fashioned systems before 2000
- Today, this isn't as relevant due to ML systems

3.1.1 Replication

- Is it good?
  - Scalability - bad at dealing with high data volume because it cannot scale, good read speed, writing speed is also good but more complex (since each replication needs to write)
  - Consistency/correctness - also complex due to writing being complex
  - Fault tolerance/high availability - very good due to the redundancy
- Core challenge: How to handle changes to replicated data?

3.1.2 Single Leader Replication

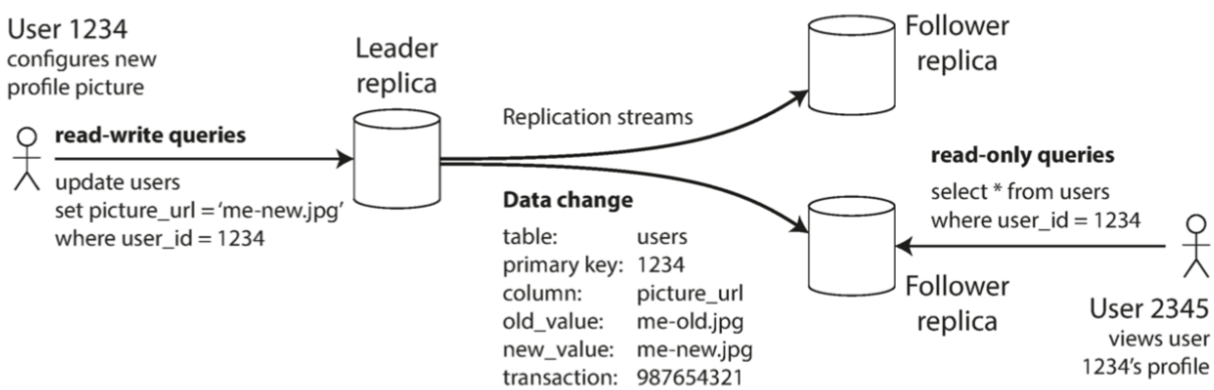

Figure 3: Single Leader Replication

- Client sends requests to the leader to write to the db, which the leader saves locally
- The leader then sends the data change to all of its followers
- Clients read data from the leader or followers
- This leader-follower idea is one of the big ideas in Computer Science and distributed message brokers such as Kafka
- Advantages - simplicity (easy to understand), easy to coordinate (making it consistent)
- Disadvantage - single point of failure (if the leader fails everything falls apart)

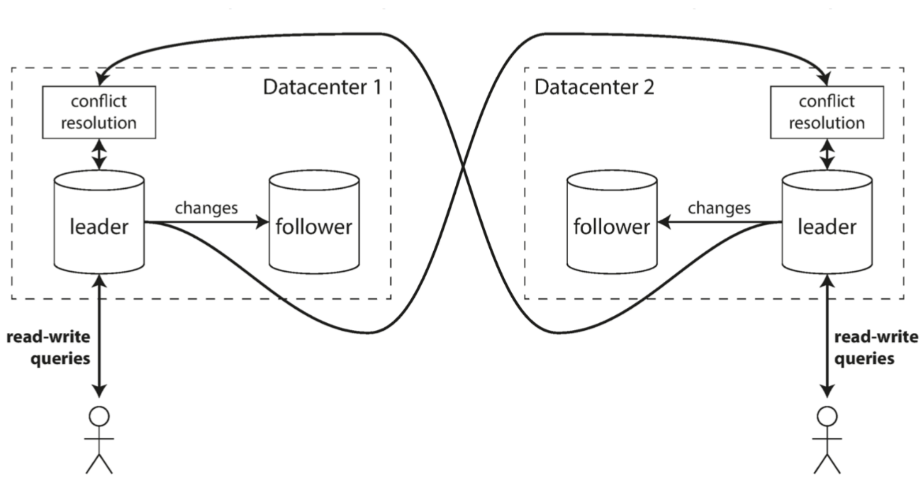

Figure 4: Multi Leader Replication

3.1.3 Multi Leader Replication (Multi Datacenter Operation)

- Avoids the single point of failure problem
- Consistency is now more complex because things need to be submitted to multiple leaders (making it not very popular)
- Advantages - performance, tolerance of data center outages and network problems
- Disadvantages - harder to coordinate due to potential write conflicts (since the same data can be concurrently modified in two different data centers)
- Example: google docs undo-redo, which needs to have many replicas since there can be multiple people concurrently writing on a single document

3.1.4 Leaderless Replication

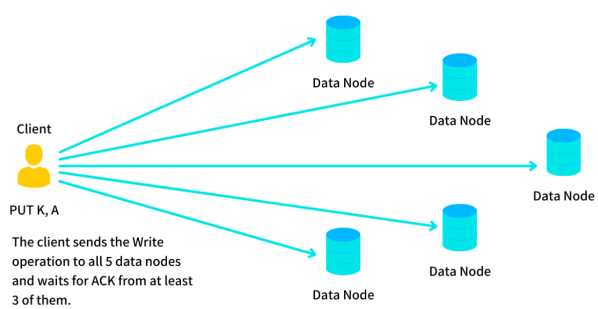

Figure 5: Leaderless Replication

- Client sends write operations to all data nodes
- Even more complex because we need to make distributed decision making

3.1.5 Concluding Replication

- Tradeoffs:
  - Simplicity
  - Conflict across nodes (consistency)
  - Faulty nodes
  - Network interruption
  - Latency spikes
- Overall, replication is only good for small data since it cannot scale

3.1.6 Partitioning

- Can be combined with replication

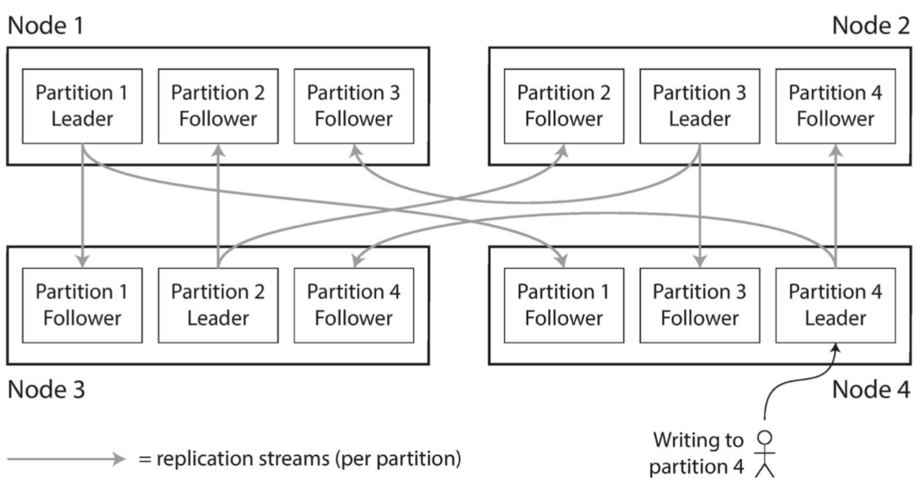

Figure 6: Combining Replication and Partitioning

- Used when the dataset is too big for a single machine
- Used when a large dataset has to be distributed across many disks (the extra storage is needed)
- Used when the query load can be distributed across many processors (allows for more computing)
- Challenges: how to partition and index? How to add or remove nodes (rebalancing)? How to route the requests and execute queries?
- How to partition - key metric = load balancing (and query efficiency)
- Query efficiency - ideally a system that is good at performing range queries
  - In reality there are tradeoffs and need to find workarounds
- Load balancing - ideally hope to spread the data and query load evenly across nodes with all nodes having balanced workloads
  - In reality there will often be hot spots (a popular partition with a high load)

3.1.7 Partitioning by Key Range

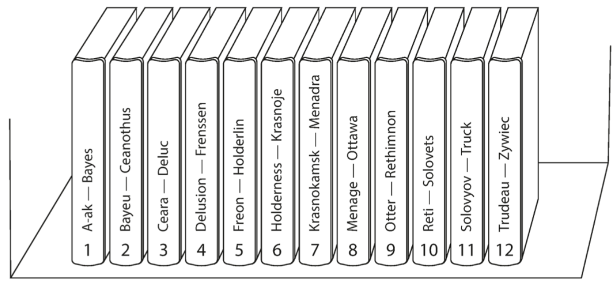

Figure 7: Partitioning by Key Range

- Advantage - can do range queries
- Problems:
  - ranges of keys are not necessarily evenly spaced (some ranges can have a lot more data, creating unbalanced workloads)
  - It is a manual process that requires domain expertise to build the semantic keys
  - Hard to rebalance (what happens if you want to add or remove nodes? Requires new key ranges)
  - Hot spot issues (ex: for a list of names, some letters like T are much more common)
- Common keys: name, titles, dates

3.1.8 Partition by hash of key

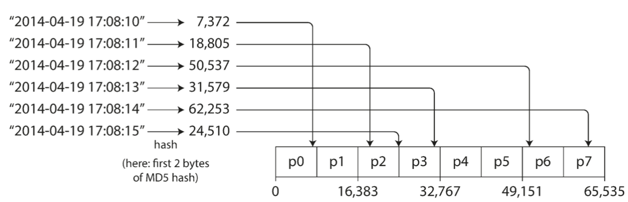

Figure 8: Partition by hash of key

- Advantage - automatic (only need to develop a good hash function), easy to balance (since the hash does it for you)
- Problems - cannot efficiently perform range queries anymore (hash doesn't preserve key values without being called first)

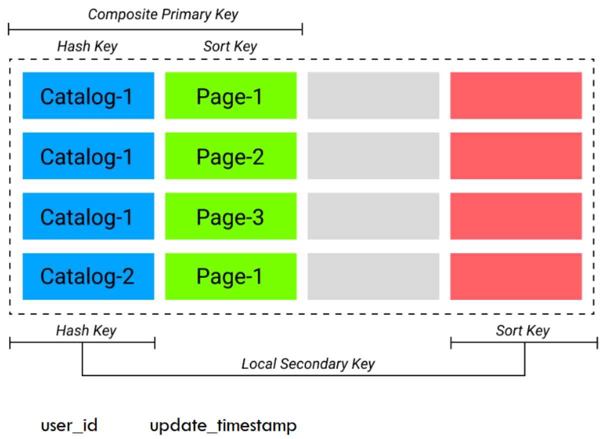

Figure 9: Partitioning by hash of key + key range

3.1.9 Partitioning by hash of key + key range

- Trying to combine the best of both worlds of methods 1 and 2
- Makes the system more complicated, but it might enjoy the benefits of both (and suffer both the weaknesses)
- Challenge: How to add or remove nodes?
- Rebalancing - move the load from one node in a cluster to another
  - Motivations: when the query throughput increases we want to add more CPUs, when the dataset size increases we want to add more disks and RAM, when a machine fails we want its workload given to functional machines
  - Goals: share the load fairly after rebalancing, service needs to still be live while rebalancing, minimize data moving
- Strawman Solution: hash mod N (N = amount of nodes)
  - Main issue: is very slow if lots of data is needed at once

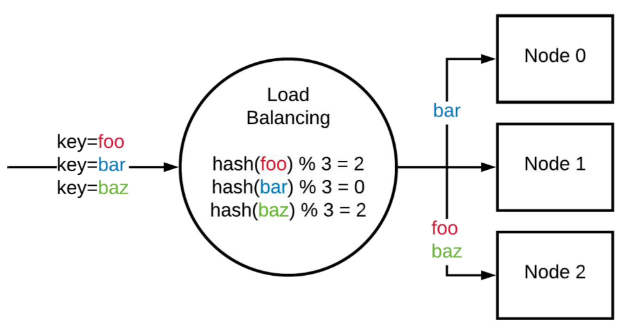

Figure 10: Hash mod N

- Better solution: consistent hashing ring (hashing data keys and node names)

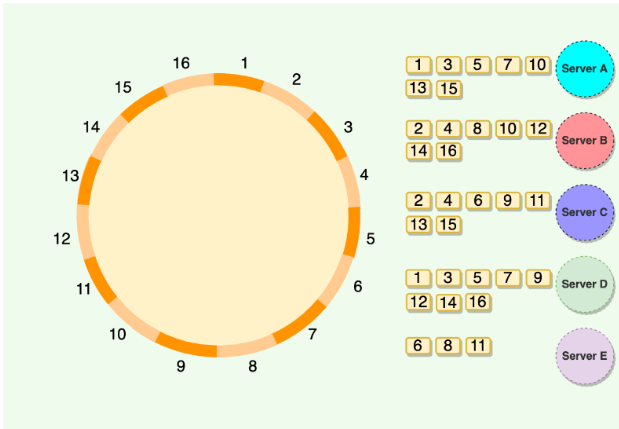

Figure 11: Hashing ring

- Done by hashing both machines and objects in the same range
  - To assign an object to a machine, you first computer the object's hash and then traverse right until you find a machine mash. That object is then assigned to that machine.
- This means you only have to move data from one server instead of all servers
- Searching can be done in log(n) time when using binary trees
- Used to address technical challenges that arise in peer to peer networks (helps deal with the issue of finding file locations)