Introduction to Data Engineering

Week 9: Distribution is Challenging

PROFESSIONAL & CONTINUING EDUCATION
UNIVERSITY of WASHINGTON

Today's Lecture…

- Distribution of data and compute:
  - Why do it?
  - How did it affect the designs of things we've studied?
- Review of some Key Distribution Challenges:
  - Replication
  - Partitioning
  - Transactions
  - Consensus

Distribution: Why and What?

Distribution: Why Do It?

Why do we distribute data?

- ***Scalability:***
  - *Go Bigger:* Data too big to fit on a single machine.
  - *Go Faster:* Desired read/write rates too fast for a single machine.
- ***Fault Tolerance / High Availability:***
  - Keep working if machine(s) go down via redundancy
- ***Latency:***
  - Keep copies of data physically nearer to its users
  - Cache results of costly operations between user and source

Distribution: Sharing

Types of Sharing

- ***Shared-Memory on one Node:***
  - Buy a big machine; Fault tolerance may be limited and costly
- ***Shared Distributed-Memory:***
  - Data spread over memories on multiple nodes
- ***Shared-Disk:***
  - Array of disks connected over fast network
  - Scalability limited by contention, locking
- ***Share Nothing:***
  - Scale horizontally over independent nodes with own CPU/RAM/disk
  - Coordinate at software layer

Distribution: Replication and Partitioning

How Data Gets Distributed Over Nodes

- ***Replication:***
  - Keep a copy of same data on multiple nodes
  - Redundancy guards against node failure/unavailability
  - Might also improve performance
- ***Partitioning:***
  - Break large data into smaller partitions
  - Assign partitions to separate nodes (sharding)

Replication

Replication: Challenges and Opportunities

What and Why?

- ***Replication***: keep copies of the same data on multiple nodes, connected via a network.
- ***Why?***
  - Keep data near the user to *reduce latency*
  - Keep system working even with failed nodes to *increase availability*
  - To increase number of machines that can serve reads to *increase read throughput*

Replication: How Immutability Helps

Why can Immutability help?

- ***Immutability:*** if replicated data doesn't change over time it is easier to:
  - Maintain consistency
  - Support concurrent access
- Almost everything hard about replication involves handling ***changes*** to the replicated data.
- There are several approaches to replicating data between nodes.

Replication: Leaders and Followers

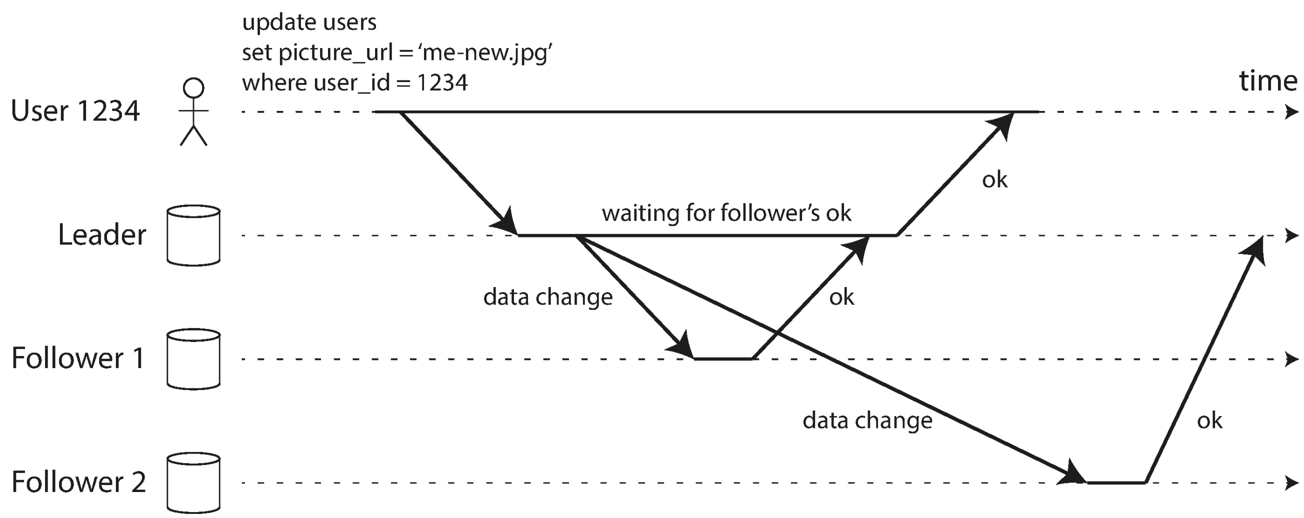
One Approach: Leaders and Followers

- Each node that stores a copy of a datum is a ***replica***
- Every write should eventually reach all desired replicas.
- Common approach: ***Leader and Follower***
  - One replica designated a leader (or master or primary)
  - Clients submit *writes* to the leader, which writes locally first
  - Leader sends changes to the followers, which then update their local storage, applying writes *in same order* as at leader.
- Sometimes have a long term designated leader; other times not.

Replication: Synchrony and Asynchrony

Leaders and Followers: Synchrony and Asynchrony

- ***Terminology: Synchronous vs. Asynchronous Updates***
  - ***Synchronous:*** Don't report success until replicas updated.
  - ***Asynchronous:*** May report success before some or all replicas are written.
  - ***Semi-synchronous:*** some followers synchronous, others not.



Replication: Handling Failed Replicas

Leaders and Followers: Handling Failed Replicas

- ***Handling Failed Replicas***
  - What if a leader fails? How do we tell and what do we do?
    - Become unavailable?
    - Decide on a new leader?
    - What about stuff that was only on leader at failure time?
  - What if new followers are added?
    - How do they catch up?
- Different systems do different things, sometimes it’s configurable.

Replication: Consistency

Leaders and Followers: Consistency

- We saw earlier that replication could lag at some nodes.
- Result: consistency issues.
- ***Consistency Levels***
  - ***Read-After-Write:*** i.e. “read your writes”
  - ***Monotonic Reads:*** you can’t see reads backward in time.
  - ***Consistent Prefix Reads:*** if sequence of writes happens in a certain order, any reader sees them in the same order.

Replication: Multiple Leaders

Multi-Leader Replication

- Single leader systems have downsides:
  - Bottleneck for write throughput
  - Single point of failure
- Multi-leader lets writes be accepted at multiple nodes
- Replication must still happen from leaders to followers
- Multiple reasons to do it
- Schemes can be complex
- Other names: master-master, active/active replication

Replication: Multiple Leaders

Multi-Leader Replication: Use Cases

- Multi-datacenter operation
  - Replicas in multiple datacenters
  - Leader(s) in each datacenter
- Challenges:
  - Between datacenters, latency may be high
  - Failure may force datacenters to operate independently for a while
  - Network conditions likely worse between DCs than within
  - Handling write conflicts: many approaches with different pros-cons

Replication: Leaderless Replication

Leaderless Replication

- Approach 1: Client sends write to several replicas.
- Approach 2: A coordinator node replicates on behalf of the client
  - Coordinator doesn't enforce a particular ordering of writes
  - Coordinator may read from multiple replicas
  - Related ideas: *read-repair*, *hinted-handoff*, *anti-entropy*.
  - Quorum reading and writing: can trade off latency for consistency

Partitioning

Partitioning

Partitioning:

- Break data into multiple chunks spread across nodes
- Partitions sometimes called shards, regions, tablets, vnodes, vBuckets…
- Once data is partitioned:
  - It can be bigger than any single node could hold
  - Computation can be independently executed on partitions
- Partitioning is usually combined with replication

Partitioning: Handling Key-Value Data

Partitioning Key-Value Data

- **Goal:**
  - Spread out the data distribution and query load
  - Avoid imbalanced partitions and hot spots
- **Schemes:**
  - Partition by key range (bad if keys very non-uniform)
  - Partition by hashes of keys (good hash function required) then give each partition a range of hashes
    - Lose efficient range queries unless we do extra work

Partitioning: Handling Key-Value Data

Hot Spots can Still Occur Naturally…

- **e.g.** Imagine a social media site where there are celebrities and schmoes:
- A celebrity may form a natural hot spot
- Can use tricks, e.g.
  - Figure out how many pieces a celebrity should be broken into…
  - Augment their key with a piece number…
  - …then hash that.

Partitioning: Handling Key-Value Data

Rebalancing Partitions

- Over a DB's lifetime things may change:
  - Query throughput increases so we want more nodes
  - Dataset size increases so we want more nodes
  - Nodes fail so we want failover or replacement
- All the above require data and queries to be *rebalanced*
- Desiderata:
  - After rebalancing things should be shared fairly
  - While rebalancing system should remain available
  - We should move as little data as possible

Partitioning

Rebalancing Partitions: Strategies

- **Bad Idea:** hash(data) mod N
  - We change N and everything has to move
- **Fixed Number of Partitions**
  - Fix a number of partitions, N, where N >> |nodes|
  - If a node is added it can steal a few partitions from every existing node; leaving nodes can donate back.
  - Only entire partitions move between nodes
  - Can leave old assignment of partition in place while data is moving

Partitioning

Rebalancing Partitions: Strategies

- **Dynamic Partitioning**
  - Create partitions dynamically
  - When a partition gets too big, split it in halves
  - When a partition shrinks below threshold, merge with adjacent partitions
  - Each partition is assigned to a node
    - Merges can happen locally
    - Splits can overflow to other nodes to rebalance

Partitioning: Routing Requests

Request Routing

- When a client requests data how does it know which node matters?
- This is a *service discovery* problem with various approaches:
  - 1. Let clients contact any node, forward requests and replies if they contacted the wrong node
  - 2. Use a routing tier to direct clients to correct node
  - 3. Force clients to be aware of where things are

Partitioning

Request Routing

- How do we get queries to the correct destination node?
- All participants must agree on:
  - What is where?
  - What is alive?
- Many systems use a separate ***coordination service*** for this, e.g. Zookeeper in Kafka and HBase
- Cassandra and its relatives use a ***gossip protocol*** among nodes to disseminate changes in cluster state

Transactions

Transactions

What and Why?

- Historically, transactions simplify the programming model in the face of:
  - Failures and crashes
    - Updates may not get everywhere
    - Updates may be incompletely applied at a node
  - Concurrent accesses
    - Readers may observe partial changes in progress
    - Writes may collide and produce corrupt results
    - Check and act operations may race
- Common argument: “every programmer understands transactions, so let’s just solve all our problems with them.”
  - No.

Transactions: Idea and Desiderata

Transaction: Notion and Properties

> A ***transaction*** lets an application:
>
> > Group multiple reads and writes…
> >
> > …in a single logical unit…
> >
> > …which either commits, or aborts.
>
> Traditionally transactions were ***ACID***:
>
> > **Atomic**: all or nothing effects
> >
> > **Consistent**: never leave database in a corrupt state
> >
> > **Isolated**: concurrent TXNs *appear* to run serially
> >
> > **Durable**: a committed TXN's result is stored safely

Transactions: Scope of a Transaction

Single vs Multiple Object Operations

- **Single Objects:**
  - Atomicity and isolation are easier
  - e.g. one database row in an RDBMS; one JSON doc in a doc DB
- **Multiple Objects:**
  - Harder
  - May have to use *distributed locks*
  - Failure more likely and pernicious, latency worse
  - Many distributed datastore abandon multi-object transactions entirely
- Scary Campfire Story: X/Open XA spec. for distributed transaction processing

Transactions: Isolation Levels

Isolation Levels

- TXNs touching different data can safely run in parallel.
- But, we may have problems, if two TXNs:
  - Modify the same data, or…
  - One reads data the other is modifying.
- The 'I' in ACID is about hiding such problems.
- There are different levels of isolation and:
  - They are hard to understand
  - Implemented inconsistently between databases.

Transactions: Serializability

Serializable Isolation: The Ideal

- The end result of transactions, even executing in parallel, is the same as if they had executed, one at a time, *serially*, without any concurrency.
- Thus:
  - If the transactions behave correctly run individually…
  - …they continue to be correct if run concurrently.
  - The database prevents all possible race conditions.
- So why not demand serializability always and everywhere?

Transactions: Serializability

Serializable Isolation: The Reality

- **Supporting Serializability is:**
  - Expensive
  - Complex, especially with multi-node systems
- **Common Techniques to support Serializability:**
  - Literally execute transactions in serial order
  - *Two Phase Locking* (only option for many years)
  - Optimistic Concurrency Control

Transactions

Other Isolation Levels

- If we don’t want to pay for Serializability what can we do?
- There are ***weaker isolation levels***, but:
  - Each doesn’t guard against certain concurrency anomalies
  - Your application will have to deal with the remainder, e.g. using explicit locking
- **Isolation levels to read up on:**
  - Read Committed
  - Snapshot Isolation / Repeatable Read

Consensus

Consensus

The Consensus Problem

- ***Consensus:*** getting all nodes to agree on something.
- Consensus is hard to do when there are:
    - Network failures
    - Process failures
- Many problems are *reducible* to consensus:
    - Leader election
    - Atomic commit
    - Total order broadcast
    - Distributed locking
    - Uniqueness constraint enforcement
    - Membership/coordination service
- Consensus mechanisms are complex and hard to implement

Consensus

Consensus in Practical Systems

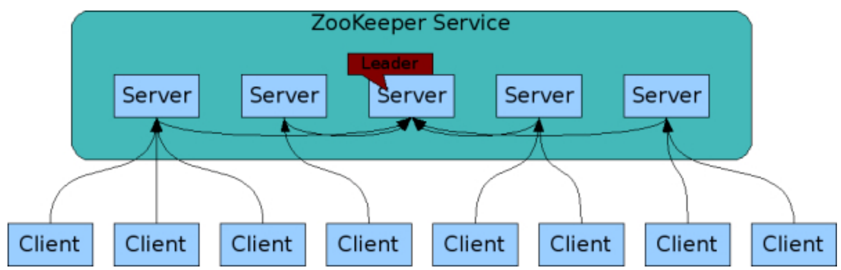
- We don't have time to say much about Consensus here
- Several Consensus algorithms exist:
  - e.g. Paxos, Raft, Viewstamped Replication
- There are systems that implement variations of it:
  - As a service usable by other systems
  - e.g. Zookeeper, Raft, Chubby, etcd, consul.
  - That are used by some systems seen in this program (e.g. HDFS, HBase, Kafka…)
- There's debate over whether it should be a *service* or a *library*

Consensus

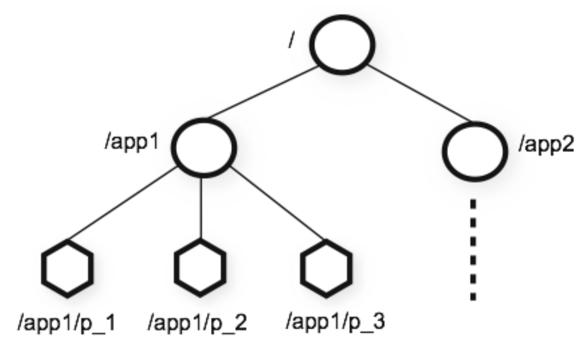
Zookeeper: An Example Consensus System

> Uses a protocol called Zookeeper Atomic Broadcast

> Deployed as a cluster of nodes (available when majority up):



> Exposes hierarchical namespace:



Consensus

Zookeeper: Exposed Abstractions

- **ZNodes:** each can have associated…
  - Children
  - Small amounts of data: e.g. status, configuration, location
  - Each node is read and written atomically
- **Ephemeral znodes:** exist as long as client session that created them remains active.
- **Sequence nodes:** monotonically increasing number appended to requested name.
- Nodes can have ***watches***, that are triggered on node changes, with client notified.

Consensus

Zookeeper: Guarantees

> **Sequential Consistency:** updates from a client will be applied in the order they were sent.
> **Atomicity:** Updates are all or nothing.
> **Single System Image:** client sees same view regardless of node it connects to.
> **Reliability:** Updates survive until overwritten.
> **Timeliness:** Client view of system guaranteed to be up to date within a certain time bound.

Summary:

- Systems are distributed because we want things like:
  - Scalability, fault tolerance, low latency
- Distribution requires a system to grapple with:
  - Replication
  - Partitioning
  - Transactions
  - Consensus
- Systems we studied:
  - Take various approaches to dealing with the above challenges…
  - …yielding various quirks and compromises
- Understanding why these things are the way they are makes them easier to deal with.

For Next Week: Project Presentations

- *Does there exist anybody who has not yet:*
  - *Formed a team?*
  - *Chosen a dataset?*
  - *Got started?*
- *We'll do presentations in class next week*
  - *If we can't fit them all in, we may have to overflow to do some by Zoom at another time*
- *Project work can continue for another week or two (until grades due) after next week*

References

> *Designing Data-Intensive Applications*, Kleppmann, Chapters 5—9

> *http://zookeeper.apache.org*

Q&A

UNIVERSITY OF WASHINGTON

Studying at UW

> *What defines the students and faculty of the University of Washington? Above all, it's our belief in possibility and our unshakable optimism. It's a connection to others, both near and far. It's a hunger that pushes us to tackle challenges and pursue progress. It's the conviction that together we can create a world of good. And it's our determination to Be Boundless. Join the journey at uw.edu.*