Distribution and Replication

Spring 2025

Last time…

We discussed how relational databases handle concurrent access

- multiple users accessing a database at the same time

Main abstraction: transactions

- transactions are atomic, consistent, isolated, and durable
- aka the ACID properties

There's some engineering required to ensure those properties of transactions

- logs for aborting transactions, two-phase locking to ensure isolation, …

Today…

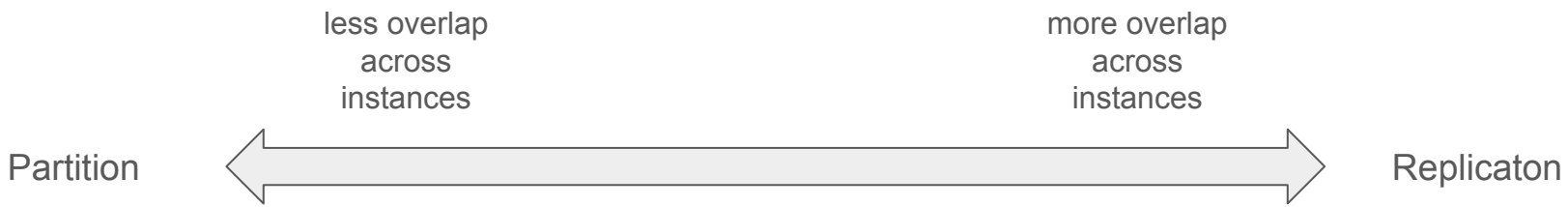
Moving from multiple users accessing a database to having multiple users accessing data spread (distributed) across multiple instances of a database

Distributing data across multiple instances helps with:

- fault tolerance — serving data even if database instances fail
- localization — keeping relevant data geographically close to users
- scalability — serve more data to more users

Distributing data

There's a spectrum of possibilities for what "distributing data" means:



Distributed computing and distributed systems

How do you compute when data or programs run on different machines (nodes) that communicate together?

That's the domain of **distributed computing**

Core problem: **consensus**

- have N nodes agree on a value of interest
- consensus is easy when there are no failures and communication is reliable
- otherwise: Paxos protocols

Example: primary-based replication

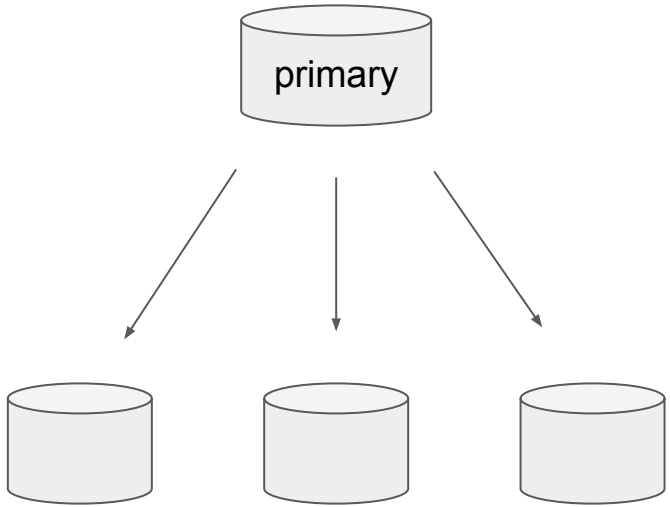
Out of the box replication for many DBs

- Postgres, MongoDB, …

Choose one instance as the read/write instance ⇒ **primary**

Propagate all changes to the read **replicas**

Use the primary instance for updates
Use any instance for reads



Example: primary-based replication

How do you propagate to the read replicas?

- use the log that we already have for rollback/recovery!
- logs record every update to the database
- send those logs to replicas who run them forward

This is reasonable for fault-tolerance and scalability

- easy to set up
- what happens when the primary is unreachable or fails?

Fragmentation

A fragment is a subset of a database

- fragments are what get distributed and live in different DB instances

Fragments may overlap

- disjoint fragments ⇒ partitioning
- completely overlapping fragments ⇒ replication
- overlapping fragments may require synchronization for consistency

Fragmentation can be:

- horizontal, vertical, mixed

Vertical fragmentation

Fragment = a subset of **columns** for each table

Different columns of tables may be stored in different database instances

- often used in columnar databases
- each table fragment has the primary key column

Think of splitting a tables into multiple tables by columns

- can always join to get back the original table

Horizontal fragmentation

(Also known as **sharding**)

Fragment = a subset of **rows** for each table

Different rows of tables may be stored in different database instances

- often split by value of a specific field (distribution key)
- often correlates with geographic distribution (e.g., country)
- keep data that is often queried together in the same instance

Transparency

To what extent does a user know how data is distributed?

Location transparency:
user doesn't need to know on which instance the data lives

Transaction transparency:
a transaction should behave as though it runs on a single database, even if the work is distributed across multiple nodes

Transparency is good, but adds a lot of complexity

Distributed query processing

For a query to return data, it needs to have all the data it needs at hand

In the presence of fragmentation:

- need to bring all needed data to the database instance processing the query

To process a query:

- decompose into the primitive operations (filter, project, join, …)
- split into fragment queries for each instance based on fragmentation
- return data of each fragment query to the processing node

CAP Theorem

(A mathematical result, so be careful applying it…)

A distributed system can only guarantee **at most 2** of the following properties at the same time:

Consistency	all nodes have the same data
Availability	every node can respond to any request
Partition tolerance	works even is messages are slow or dropped

P is pretty much required, because internet

So you can either guarantee C or A — but not both

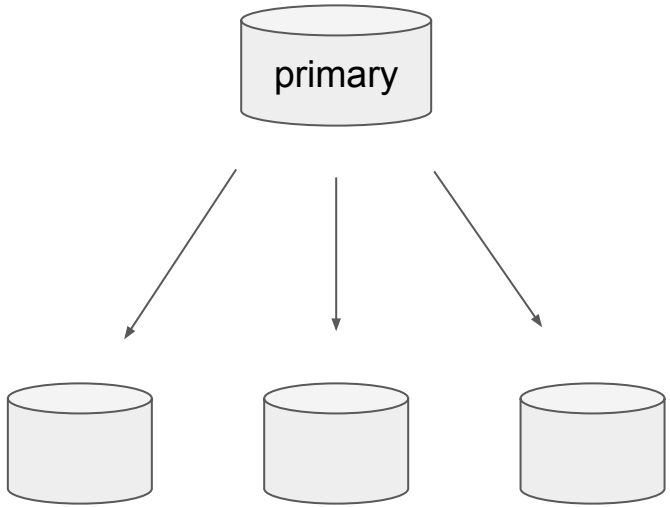
Revisit: primary-based replication

- **Primary** is the read/write instance
- Propagate all changes to the read **replicas**

No A because read replicas cannot be updated

No C when propagation is asynchronous

Synchronous propagation gains C but loses even read A



Consistency: distributed transactions

Recall ACID properties:

Atomicity
Consistency
Isolation
Durability

In a distributed context, these properties are more expensive to enforce

Atomic transaction = transaction commits or fails, across all instances holding data for that transaction

- each instance runs its own local transaction for their part of the data
- each of those local transaction may abort
- if any of those local transactions abort, it aborts the distributed transaction and all other local transactions

Two-phase commit protocol

2PC — do not confuse with the two-phase locking protocol

A consensus protocol to vote on whether to commit or abort a distributed transaction

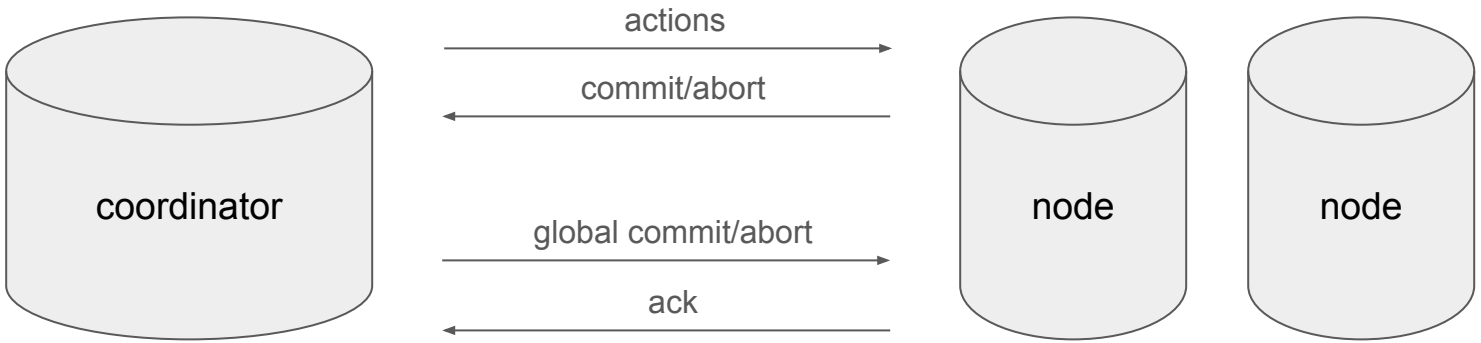
Phase 1: Voting phase

Coordinator sends transaction details to all nodes
All nodes perform their action and respond with **commit** or **abort**

Phase 2: Commit phase

If all nodes says **commit**, coordinator tells all nodes to commit locally
If a node says **abort** (or times out), coordinators tells all node to abort locally
Nodes respond with **ack** and the coordinator aborts/commits the transaction

Two-phase commit protocol



Two-phase commit protocol

Per CAP, 2PC gives you consistency (C)

- when the protocol completes, all nodes have the same data
- protocol gets stuck if coordinator fails
- nodes are left with unreleasable locks until the coordinator is restarted
- you do not have availability (A)

Relational databases: generally ensure C and give up on A

NoSQL databases (next time): generally ensure A but give up on C

That's all, folks!