Distributed Systems Primer

COMS4113: Fundamentals of Distributed Systems

Instructor: Hubertus Franke

Columbia University

https://frankeh.github.io/cu-ds1-2026-fall

Distributed Systems Primer

Outline

- Challenges, Goals, and Approaches for DS
- Example: Web Service Architecture
- The 11 Fallacies of Distributed Systems

Context

- Distribution is hard, for many reasons.
- Distributed Systems (DS) aim to provide the core mechanisms that address the challenges and hide them under abstractions.
- Not all challenges can be hidden, so DS developers and users must understand the challenges and approaches to address them.
- Next we'll talk about some of the challenges, goals, and approaches in DS design.

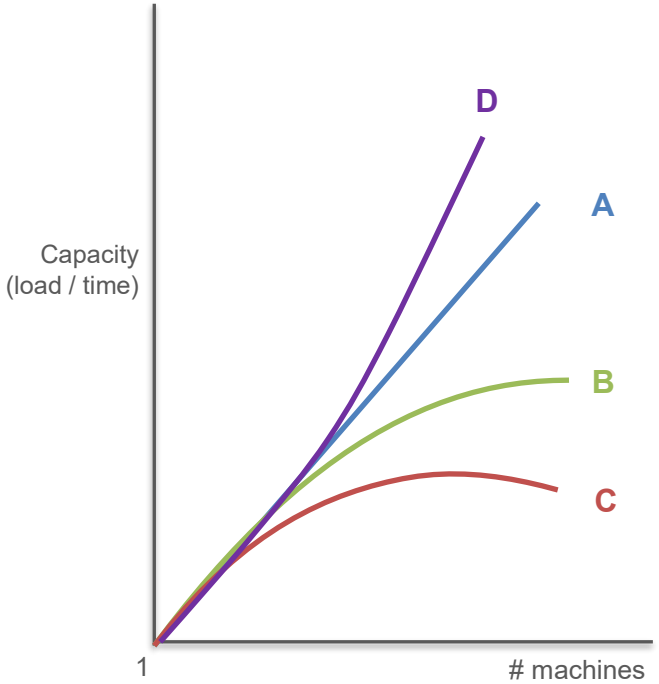
Challenge 1: Ever-Growing Load

- Data is big. Users are many. Requests are even more. No single machine can handle the ever-growing load.
- A primary goal of DSes is to enable *“multiple independent machines interconnected through a network to coordinate to achieve a common, consistent service or function.”*
- But effective coordination at scale is **hard**!

Goal 1: Scalability

- The more resource you add, the more data you should be able to store/process, and the more users you should be able to serve.
- But never hope for perfect scalability: add one machine, increase your capacity proportionally forever.
  - Most often the scalability curve tapers off as components reach their capacities (max handled load/unit of time, e.g. max requests completed per second).
  - Components that limit scalability are called **bottlenecks**, and sometimes they are very hidden (logging server, network router).

Sample Scalability Curves



- Which curve is realistic?
- **A** – (perfectl)y linear: capacity grows with every machine added, forever.
- **B** -- sub-linear: still gaining, but each machine buys less than the last.
- **C** -- flattens, then *degrades*: past some point another machine makes it worse.
- **D –** super-linear

POLL TIME!

Approach 1: Sharding

- Launch multiple processes (a.k.a. **workers**), split up your load into pieces (a.k.a. **shards**), and assign different shards to different workers.
- The workers should be designed to coordinate to achieve a common, consistent service despite the sharding.
- Unfortunately, sharding raises semantic challenges -- especially with failures.

Challenge 2: Failures Are Inevitable

- Many types of failures:
  - network, machine (software, hardware, flipped bits in memory, ...), datacenter, ...
  - some are small and isolated, others are major failures
  - some are persistent, others are temporary
  - some resolve themselves, others require human intervention
- At scale, failures are almost guaranteed to occur all the time, and most of them occur at **unpredictable** times.
- The big challenge: failures greatly complicate coordination between the machines of a distributed system.

Goal 2: Fault Tolerance

- Goal is to hide failures as much as possible: finish the computation fast despite failures, store data reliably despite failures, ...
- Fault tolerance subsumes:
  - **Availability:** the service/data continues to be operational despite failures.
  - **Durability:** data or updates acknowledged by the system will persist despite failures and will eventually become available.
- Durability differs from availability: durability can be thought of as “eventual availability” of some data or state.

Approach 2: Replication

- Have multiple **replicas** execute/store the same shard; if one replica dies, another can provide the data/computation.
- Example:
  - Say you are computing a maximum over a sharded dataset across multiple workers.
  - Have the maximum over each shard be computed by 2-3 replicas in parallel.
  - If one replica dies, another one can report the max to the other n-1 worker sets.

Challenge 3: Consistency Is Hard

- Example:
  - Say we want to compute an exact average over a sharded **and** replicated dataset.
  - How do we make sure we incorporate the average over each shard only once, if it is computed and reported by three replicas?
  - Assigning each shard a unique ID may help -- but what if the faulty replica does not die, and instead reports a wrong value due to, e.g., a memory error?

Goal 3: Well-Defined Semantics

- Dealing with these issues is hard for both the programmers and the users.
- The goal is to build infrastructure systems (storage systems, computation frameworks, ...) that provide **well-defined semantics** that hold in the face of failures -- and to express those semantics in the APIs.
- Example:
  - Say you decide that on failure your distributed computation system returns the results of a partial computation.
  - Then you must communicate that through your API, and provide error bounds for the results you report. That is semantics.

Approach 3: Rigorous Protocols

- The general approach is to develop rigorous protocols -- which we will generically call **agreement protocols** -- that let workers and replicas coordinate consistently despite failures.
- Agreement protocols often rely on the notion of **majorities**: as long as a majority agrees on a value, it can be safe to continue with that action.
- Different protocols exist for different consistency challenges, and protocols can often be composed to address bigger, more realistic challenges.

Recap: Challenge, Goal, Approach

Challenge	Goal	Approach
Ever-growing load	**Scalability**	**Sharding**
Failures are inevitable	**Fault tolerance**	**Replication**
Consistency is hard	**Well-defined semantics**	**Rigorous protocols**

Read **across**, not down: each row is one causal chain -- and each approach creates the next row's challenge.

Layered Design: The Google Stack

COMS4113

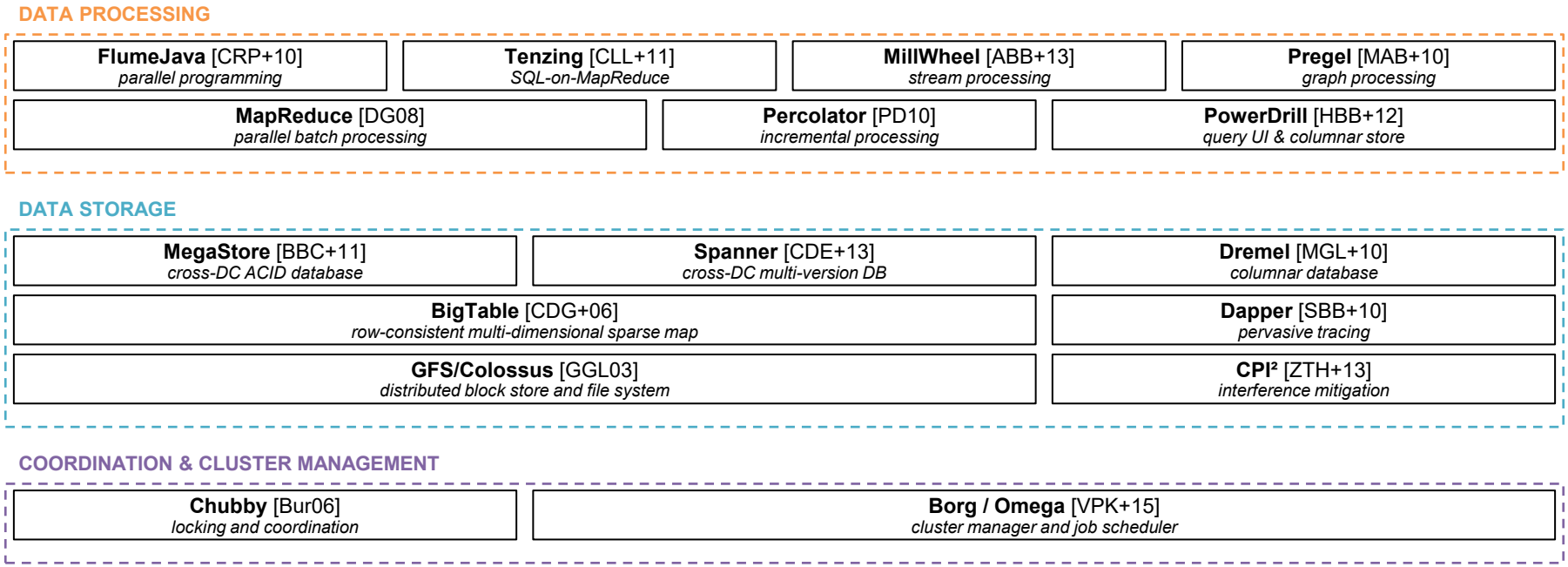


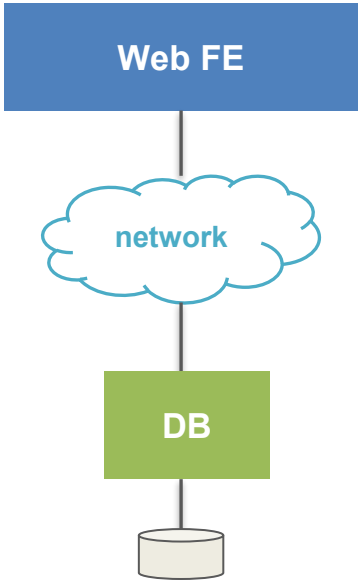
Figure after M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015 - redrawn from the author's vector original.

Three layers: **data processing** on top, **data storage** in the middle, **coordination & cluster management** underneath.

Example: Web Service Architecture

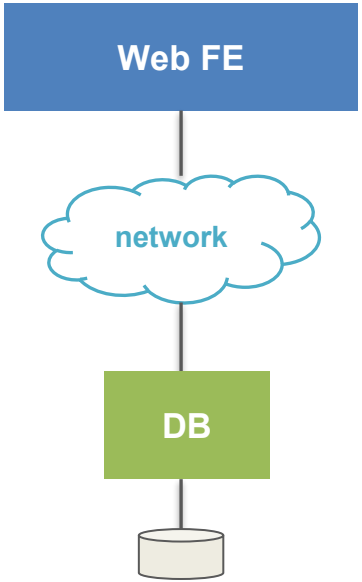
One service, four goals, and the order you are forced to fix them in

Basic Architecture



- Web front end (**FE**), database server (**DB**), network. The FE is stateless; all state lives in the DB.
- Properties to interrogate:
  - Scalability?
  - Fault tolerance?
  - Semantics?
  - Performance?

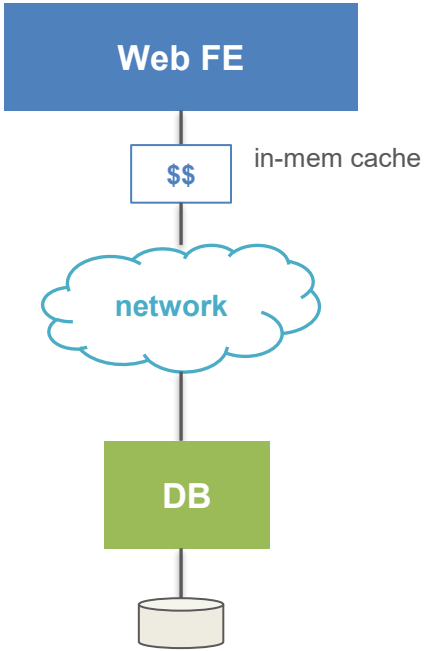
Basic Architecture



- **Performance:** poor.
- **Fault tolerance:** poor.
- **Scalability:** poor.
- **Semantics (consistency):** great!

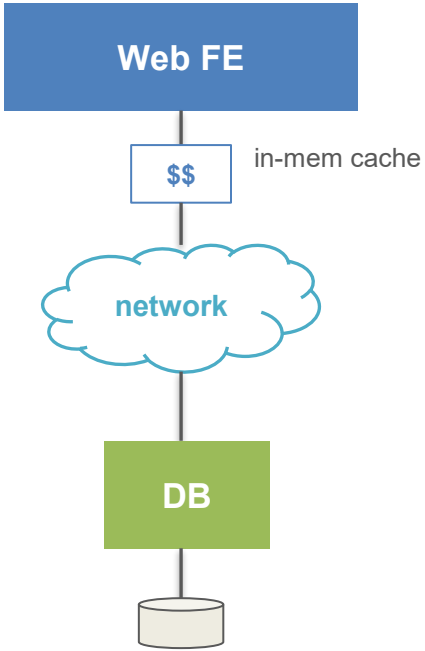
First, let us improve **performance** (latency).

Goal: Reduce Latency



- Web FE's reads/writes go through an in-memory cache ($).
- Performance?
  - For reads?
  - For writes?
- Fault tolerance?
  - Availability?
  - Durability?
- Scalability?
- Semantics / consistency?

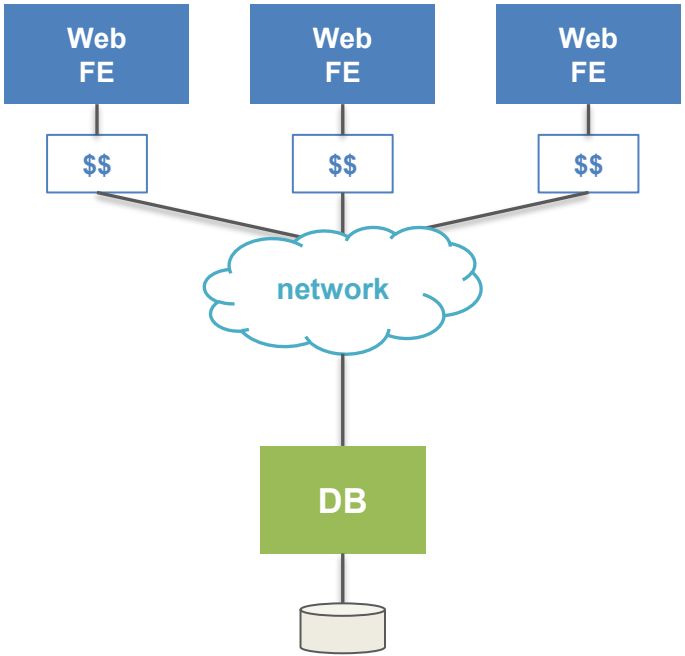
Goal: Reduce Latency



- **Read latency:** improved, if the working set fits in memory.
- **Durability:** depends on the cache -- good for write-through, poor for write-back.
- **Write latency** is the opposite: good with write-back, poor with write-through.
- **Consistency:** good. One FE reaches the DB through one cache, so behaviour is equivalent to a single machine.

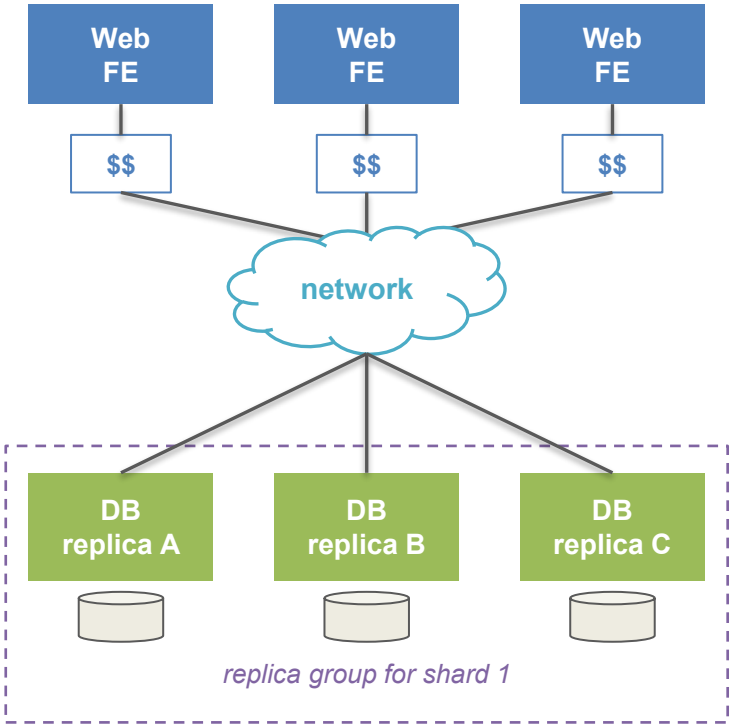
Now let us deal with **scalability** -- first the FE, then the DB.

Goal: Scale Out the FE



- Launch multiple FEs. Each has its own local cache, which we will assume is write-through.
  - Performance?
  - Fault tolerance?
  - Scalability?
  - Semantics (consistency)?

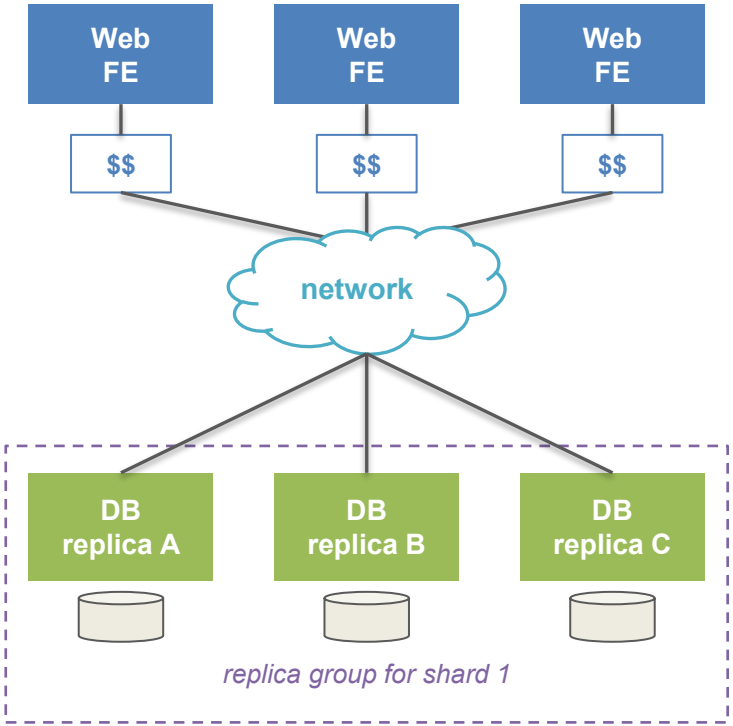
Goal: Fault Tolerance for DB



- Launch identical **replicas** of the DB server, each with its own disk. All replicas hold all data; writes go to all.
  - Performance?
  - Fault tolerance?
  - Scalability?
  - Semantics (consistency)?

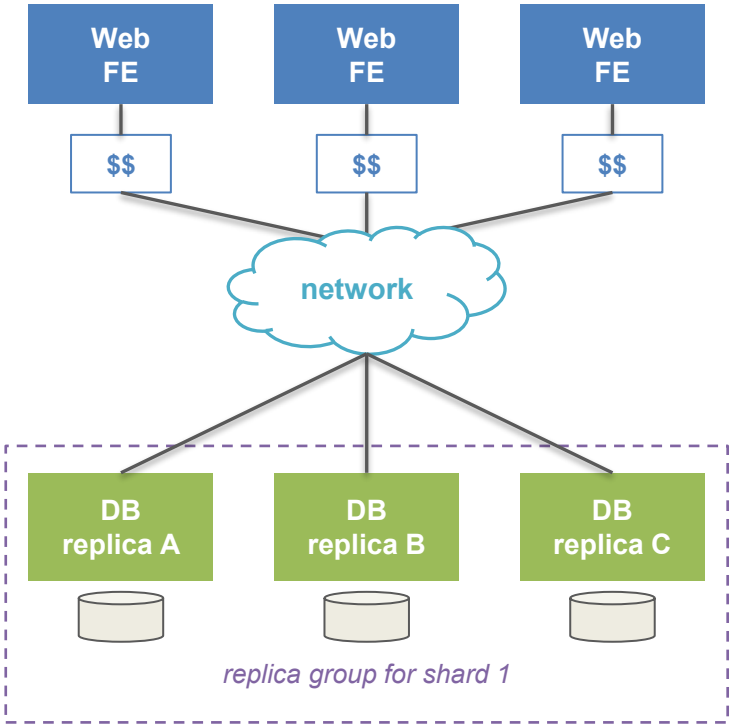
BREAKOUT

Goal: Fault Tolerance for DB



- Writes now propagate to **all** replicas, so they are much slower. Even done in parallel, the FE waits for the *slowest* replica to commit (assuming a write-through cache, which gives the best durability).
- All replicas must see all writes **in the same order**. If the order diverges they go out of sync -- so, many more consistency issues.

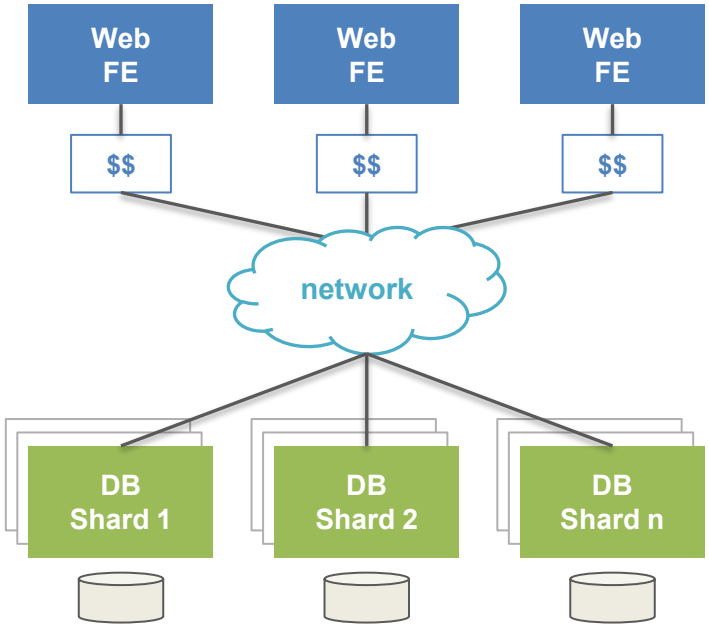
Goal: Fault Tolerance for DB



- There are also **availability** issues. Requiring all replicas to be available for a write to be satisfied (for durability) drives availability DOWN. **Consensus protocols**, which need only a majority, address this.
- Another consistency challenge: how are **reads** handled? If you read from one replica, which one -- given that updates to the item may be in flight as you read? We address this in later lectures by structuring the replica set.

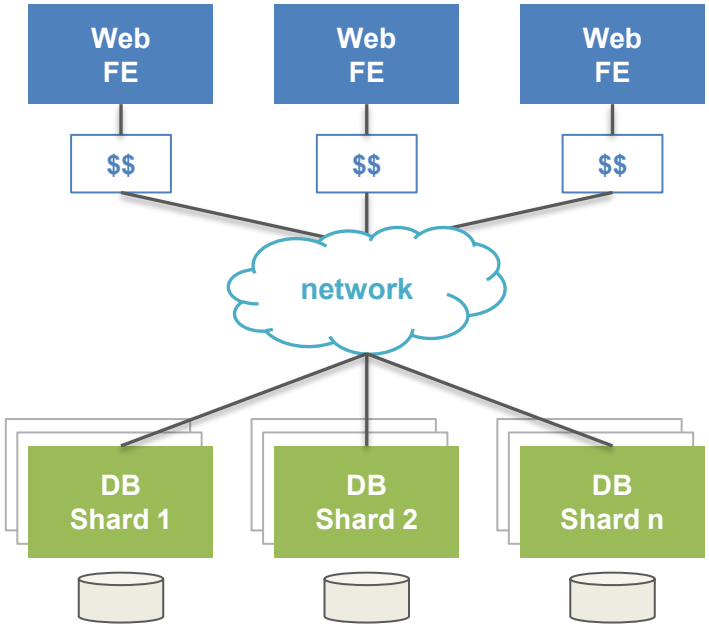
Now let us deal with **DB scalability**.

Last Goal: Scale Out the DB



- Partition the database into multiple **shards**, replicate each several times for fault tolerance. Requests for different shards go to different replica groups.
  - Performance?
  - Fault tolerance?
  - Scalability?
  - Semantics (consistency)?

Last Goal: Scale Out the DB



- New challenges that arise:
  - How should data be sharded -- by user, by some property of the data?
  - How are partitions assigned to replica groups? How do clients know which servers serve which shards?
  - If the FE reads/writes multiple entries spanning shards, how does it do that **atomically**? And if replica groups must coordinate for cross-shard updates, does that not hurt scalability?

... this class provides answers to such questions.

The 11 Fallacies of Distributed Systems

The false assumptions behind every challenge in part 1

The 11 Fallacies (1-6)

1. The network is reliable

Packets drop, connections sever, messages fail to arrive.

2. Latency is zero

Propagation, serialization and queuing delay are all real.

3. Bandwidth is infinite

Pipelines congest; throughput bottlenecks are the norm.

4. The network is secure

Internal networks are not inherently safe from interception or lateral movement.

5. Topology doesn't change

Routes, load balancers, DNS records and nodes all move.

6. There is one administrator

Subnets, firewalls and policies are managed by many teams.

Origin: formulated by Bill Joy & Tom Lyon at **Sun Microsystems**, expanded by L. Peter Deutsch, finalized by James Gosling; later extended for cloud platforms.

The 11 Fallacies (7-11)

7. Transport cost is zero

Serialization CPU, cloud egress fees and energy all add up.

8. The network is homogeneous

Hardware, kernels, protocols and stacks differ everywhere.

9. The system is monolithic

Changes ripple across coupled dependencies you did not expect.

10. The system is fully observable

Logging and tracing every state change across boundaries is hard.

11. The system is always on

High availability is not 100% uptime; regions fail.

1-8 are the classic list from Sun; **9-11** are later cloud-era additions.

Which Fallacy Bit Us Today?

Latency is zero

The cache slide: reads got faster only because we stopped crossing the network.

The network is reliable

Every replica slide: writes wait for the slowest replica, and one that never answers stalls the write.

Bandwidth is infinite / transport cost is zero

Replicating every write to every replica multiplies write traffic by the replica count.

Topology doesn't change

Sharding: who serves which shard, and how clients find out when that moves.

Take-Aways

- Scalability, fault tolerance, consistency, and performance are difficult goals to achieve **together** -- each one you fix makes another harder.
- Solving them requires rigorous protocols and deliberate system architectures, not just more machines.
- This class teaches those protocols and architectures, and how they are incorporated in real systems.
- The 11 fallacies are the names for the assumptions that break when you distribute. Every later lecture attacks at least one.