17-423/723: Designing Large-scale Software Systems

Design for Scalability

Mar 18, 2024

Leaning Goals

- Describe scalability as a QA of a software system and its relationship with other QAs, such as performance, availability, and reliability.
- Specify a scalability QA in terms of load and performance metrics.
- Describe common design problems in building a scalable software system.
- Describe the differences between vertical and horizontal scaling.
- Describe the benefits and downsides of replication and partitioning approaches to distributed data.

Scalability

What is Scalability?

- The ability of a system to handle growth in the amount of workload while maintaining an acceptable level of performance
- Why is scalability important?

Prime Big Deal Days

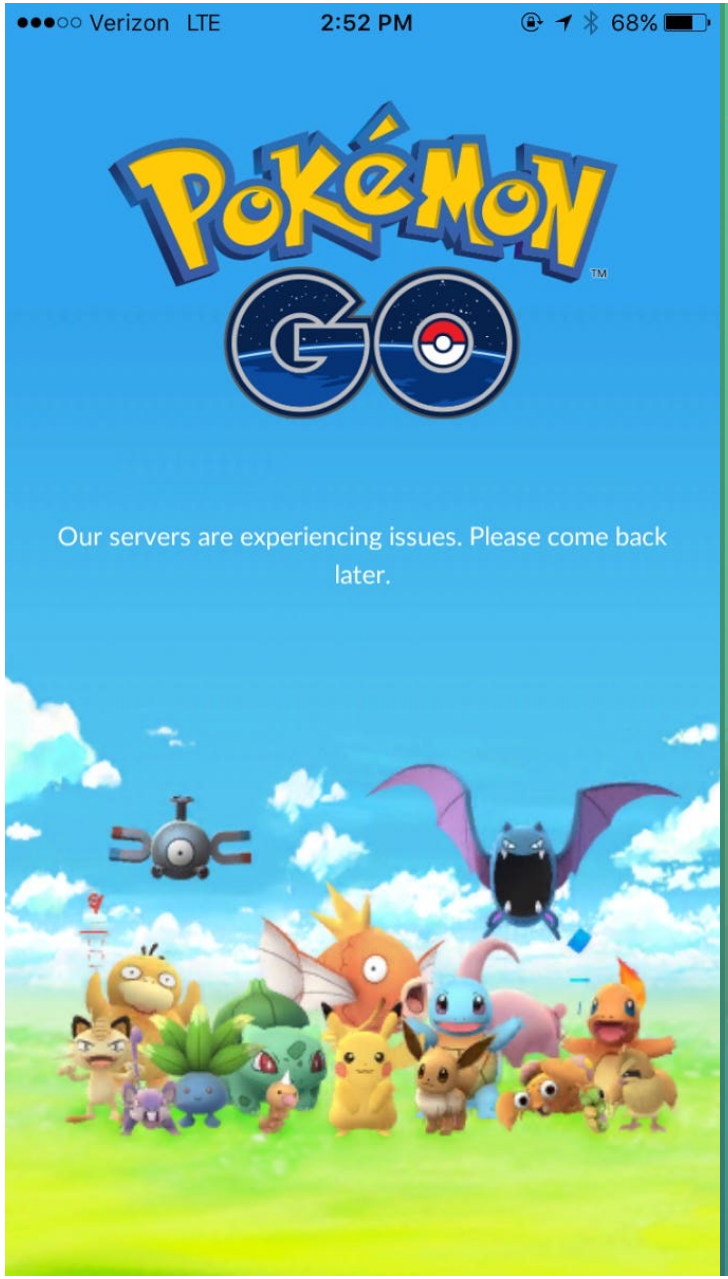
Included with a Prime membership

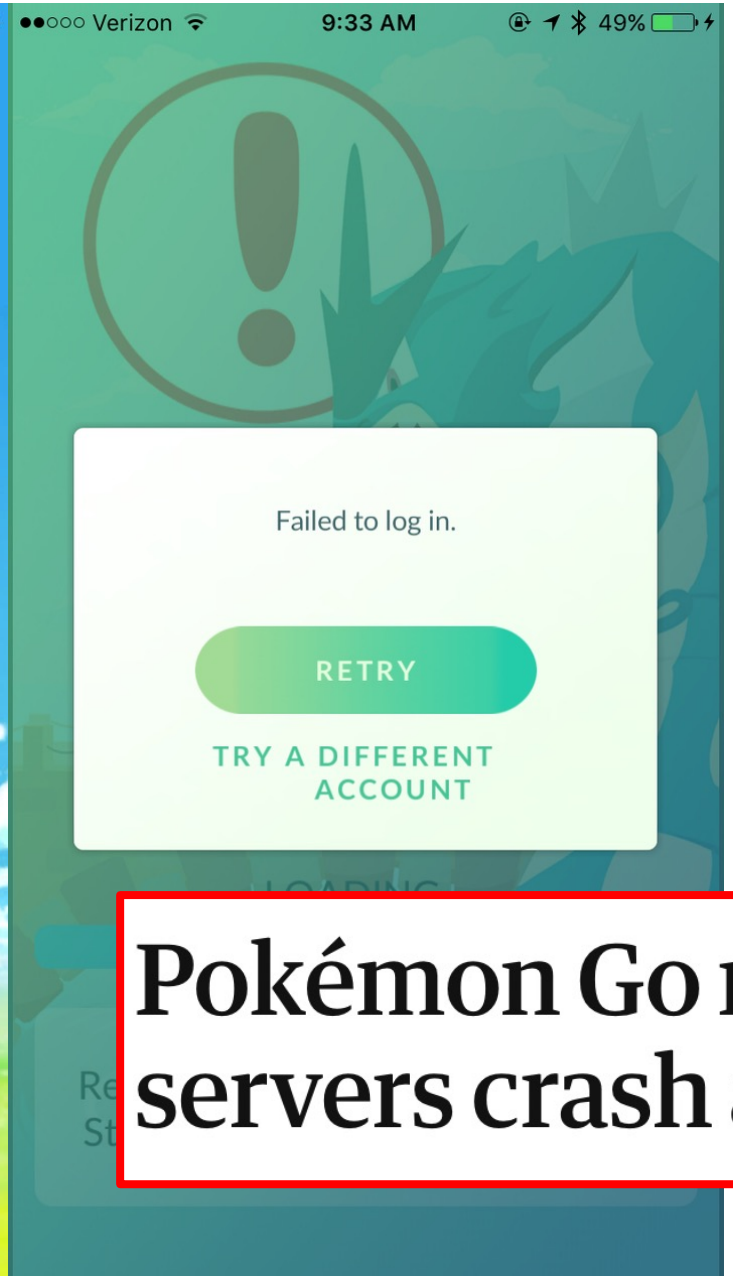
TECH

Internal documents show how Amazon scrambled to fix Prime Day glitches

PUBLISHED THU, JUL 19 2018•2:43 PM EDT | UPDATED THU, JUL 19 2018•6:16 PM EDT

- Amazon wasn't able to handle the traffic surge and failed to secure enough servers to meet the demand on Prime Day, according to expert review of internal documents obtained by CNBC.



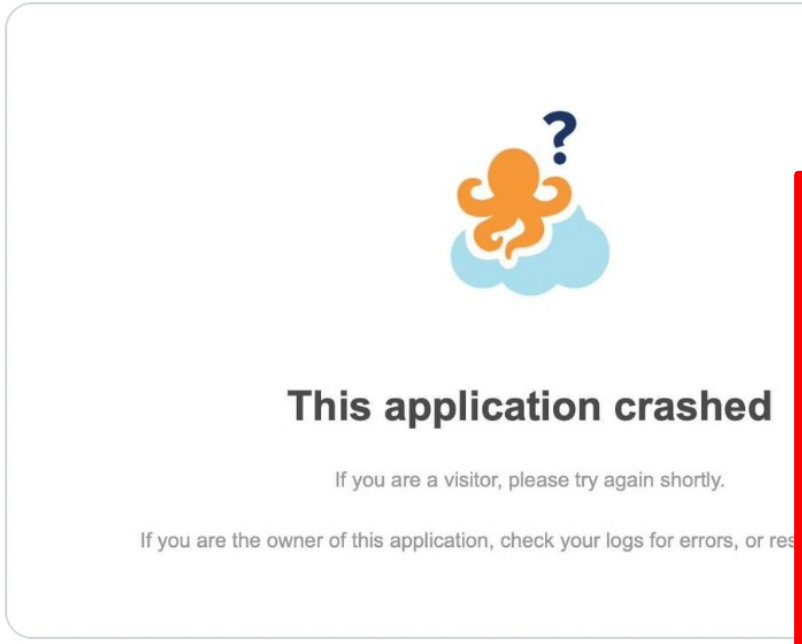


Pokémon Go makers call for calm as servers crash across Europe and US

Sat 16 Jul 2016 16.33 EDT

Travis (travtufts.bsky.social)
@travtufts · **Follow**

Using the Massachusetts vaccination website is like feverishly clicking on Ticketmaster with millions of other people, except instead of trying to see Beyoncé you're trying to keep parents alive in a pandemic. #mapoli

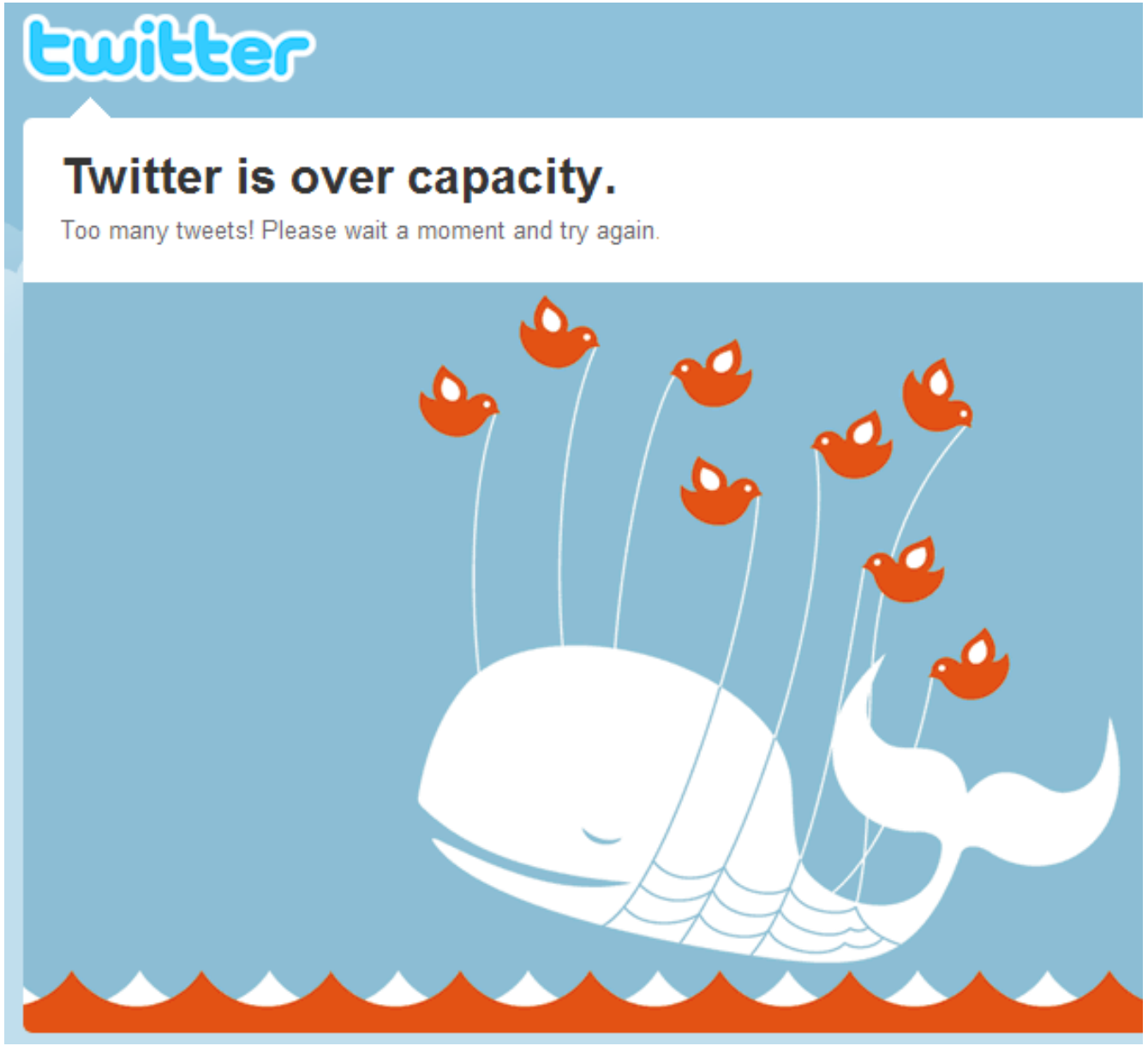


CORONAVIRUS

Massachusetts Vaccination Website Crash: What Went Wrong?

The state thinks the high volume of traffic may have been the cause, but they still aren't 100% certain

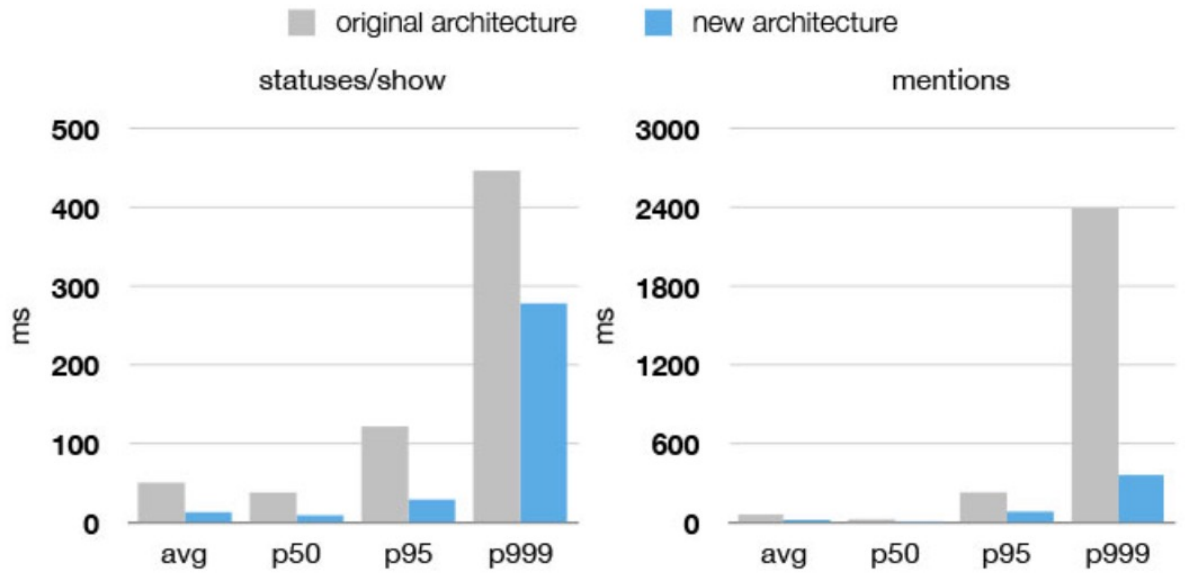
By **Staff and wire reports** • Published February 19, 2021 • Updated on February 19, 2021 at 9:01 pm



“Twitter fail whale”

Twitter Redesign for Scalability

- Early Twitter architecture (~2010): Monolithic design running Ruby on Rails, connected to MySQL databases
- Difficulty handling traffic spikes during major events (World Cup, Super Bowl, etc.,)
- Redesign decisions
  - Ruby -> JVM/Scala
  - Monolith -> Microservices
  - Load balancing, continuous monitoring, failover strategies
  - New, distributed database solution (Gizzard)



https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how

Related Concepts

- **Performance**: Amount of resources (e.g., time, memory, disk space) that the system expends to perform a function
  - It's not just about time or responsiveness (but in this class, we will mostly talk about time-related attributes)
  - It's part of a scalability QA, but not the same!
- **Availability**: Degrees to which the system is available to perform its function(s) at the request of a client
  - Usually expressed in terms of probabilities (99.99% available)
- **Reliability**: Degrees to which the system performs its functions correctly
  - e.g., Mean time between failures (1000 hours before a sensor failure)
  - Availability does not imply reliability!

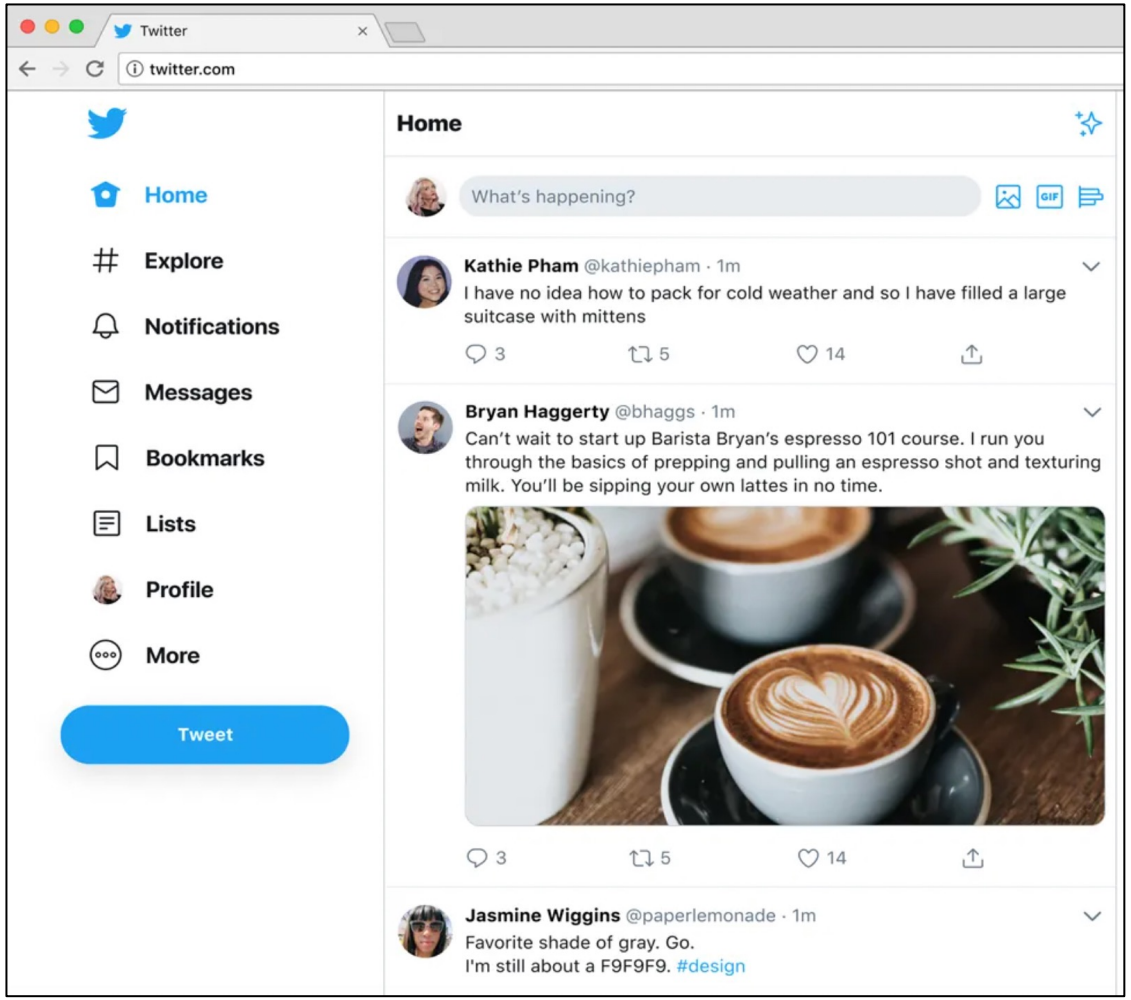
Specifying Scalability

- The ability of a system to handle growth in the amount of **workload** while maintaining an acceptable level of **performance**
- **Workload** (or simply, **load**): Amount of work that the system is given to perform
  - Number of client requests per second, average size of input data, number of concurrent users, etc.,
- **Performance**: Amount of resources that the system expends to perform a function
  - Average response time, average throughput (i.e., number of requests successfully processed per hour), peak response time, CPU utilization, etc.,

Scalability Specification: Good & Bad Examples

- “Twitter must be able to handle 100 million additional users in the next year” ❌
- “The average time to load a tweet must be no more than 100 ms” ❌
- “Twitter must be able to handle 1 million concurrent requests for viewing tweets with an average response time of 100 ms” ✅
- “On Prime Day, the Amazon storefront should be 100% up and available” ❌
- “Netflix should be able to process addition of 1000 shows per day in its catalog with no more than 2% increase in average latency” ✅
- “The company should achieve active daily users of 10 million by the end of 2024” ❌

Load Parameters Example: Twitter

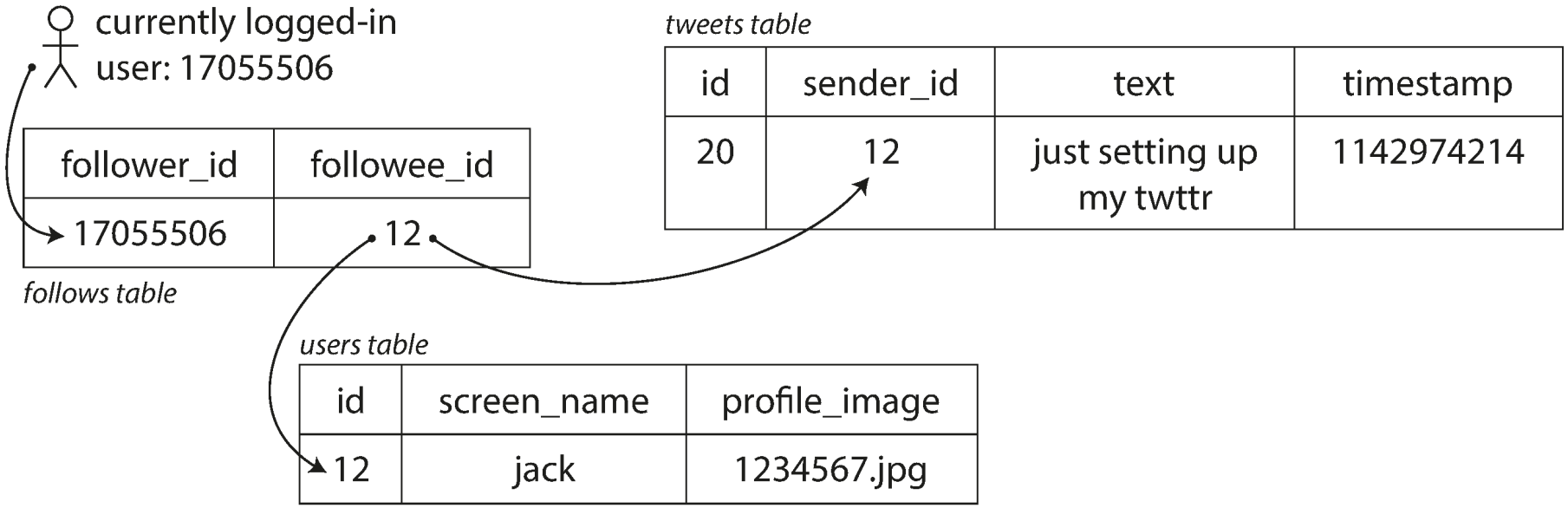


- Post tweet: Publish a tweet to followers
  - 4.6k requests/sec on average
  - 12k requests/sec at peak
- View home timeline: View tweets posted by the people that the user follows
  - 300k requests/sec on average
- Fan-out problem: Each user follows many people & each user is followed by many people
- **Q**. How would you design these two operations?

Example from: *Designing Data Intensive Applications,* Chapter 1, by M. Kleppmann

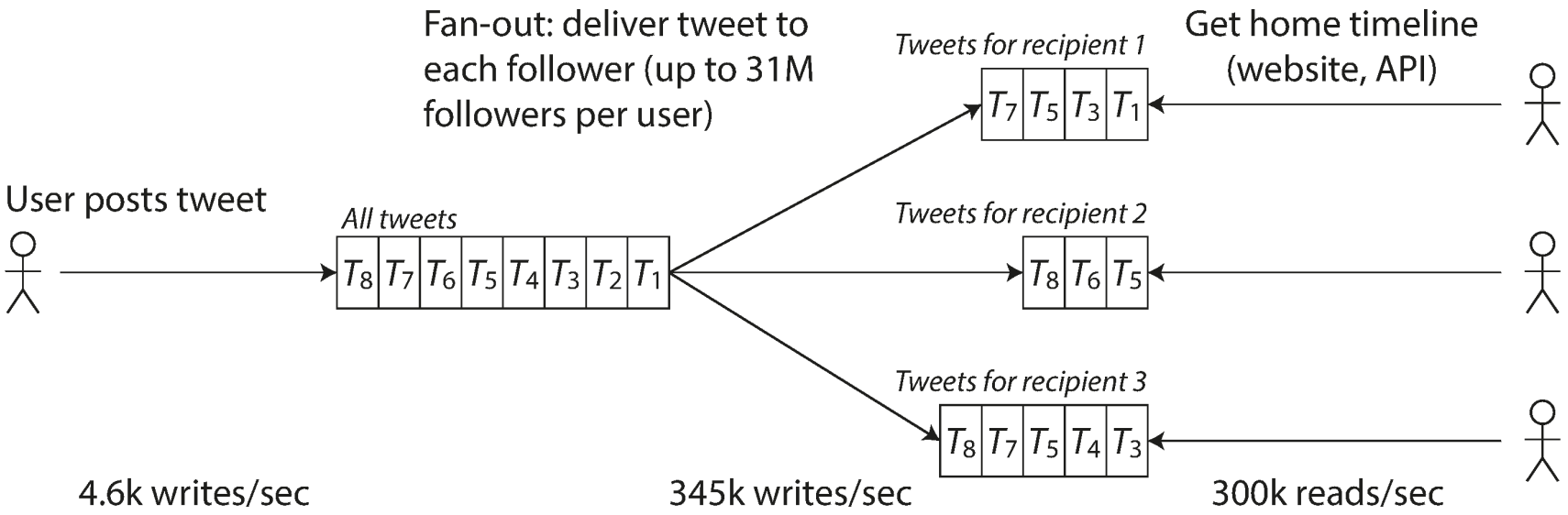
Design Option #1: Timeline Reconstruction

- **Post tweet**: Insert the new tweet into a global database of tweets.
- **View timeline**: Reconstruct the timeline for each request - (1) look up all the people the user follows, (2) find all the tweets for each of these people, (3) merge & sort by time



Design Option #2: Timeline Cache

- For each user, maintain the current view (cache) of their timeline
- **Post tweet**: (1) Look up all the people who follows the user and (2) insert the new tweet into each of their timeline
- **View timeline**: Inexpensive; no need to re-compute the timeline

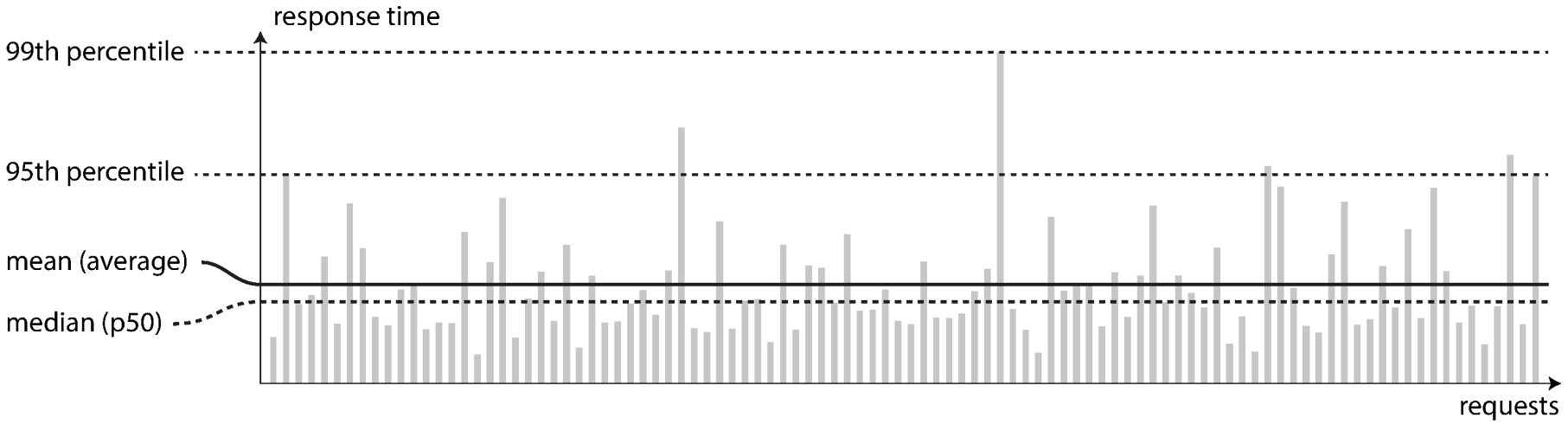


Discussion: Which Option?

- Design Option #1: Reconstruct the timeline at every request
- Design Option #2: Maintain & update a cache of timeline
- **Q. Which option would you prefer for better scalability, and under what assumptions?**
  - **What additional information about the load do you need to make the decision?**

Describing Performance: Common Metrics

- Throughput: Amount of work processed per time period
- Response time (RT): Time between a client's request & response received
  - Average RT: Commonly used, but not very useful (**Q. why not?)**
  - Percentile RT: "1.5s at 95th" means 95% of requests take < 1.5s
  - Median (50th percentile, or p50): How long users typically wait



Performance Matters!

- Performance affects other business metrics, such as revenue, conversions/downloads, user satisfaction/retention, time on site, etc.,

Source: The Real Cost of Slow Time vs Downtime, Tammy Everts (2014)

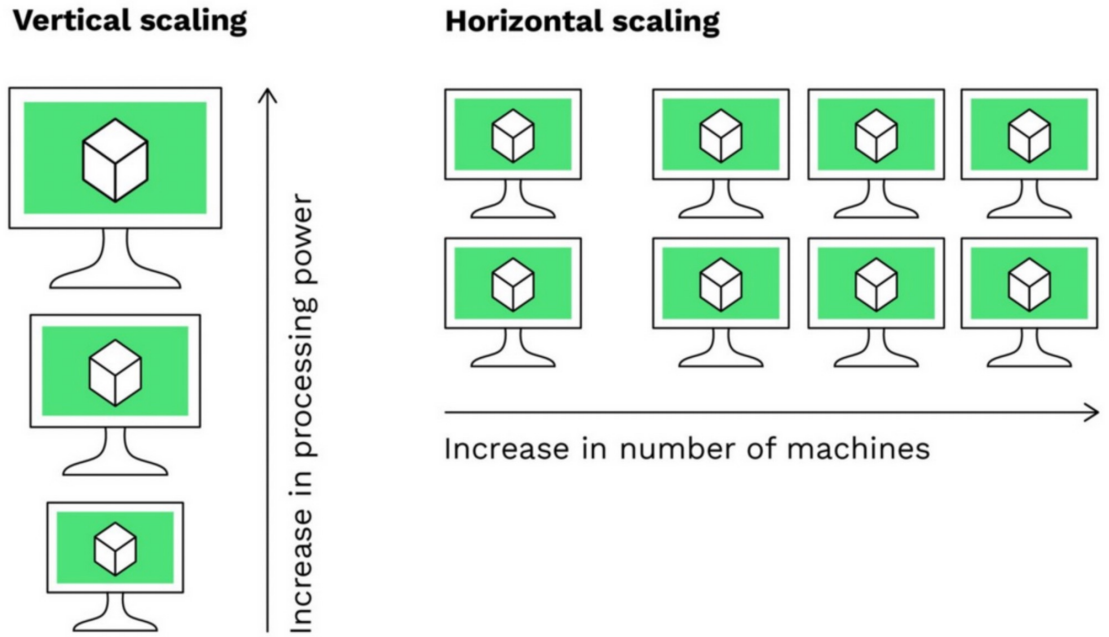
Design Patterns for Scalability

Common Design Problems for Scalability

- **How do we increase capacity to handle additional load? Vertical & horizontal scaling**
- **How do we avoid overloading one part of the system due to increased load? Load balancing**
- **How do we reduce bottleneck in the overall workload? Caching**

Vertical vs. Horizontal Scaling

- **Problem: How do we increase capacity to handle additional load?**
- **Vertical scaling** (*scaling up*): Get a more powerful machine!
- **Horizontal scaling** (*scaling out*): Distribute the load across multiple machines!



Vertical vs. Horizontal Scaling

- **Problem: H** **additional load?**
- **Vertical sc** achine!
- **Horizontal** cross multiple machines!
- **Q. What ar** **roach?**
  - So does er choice?

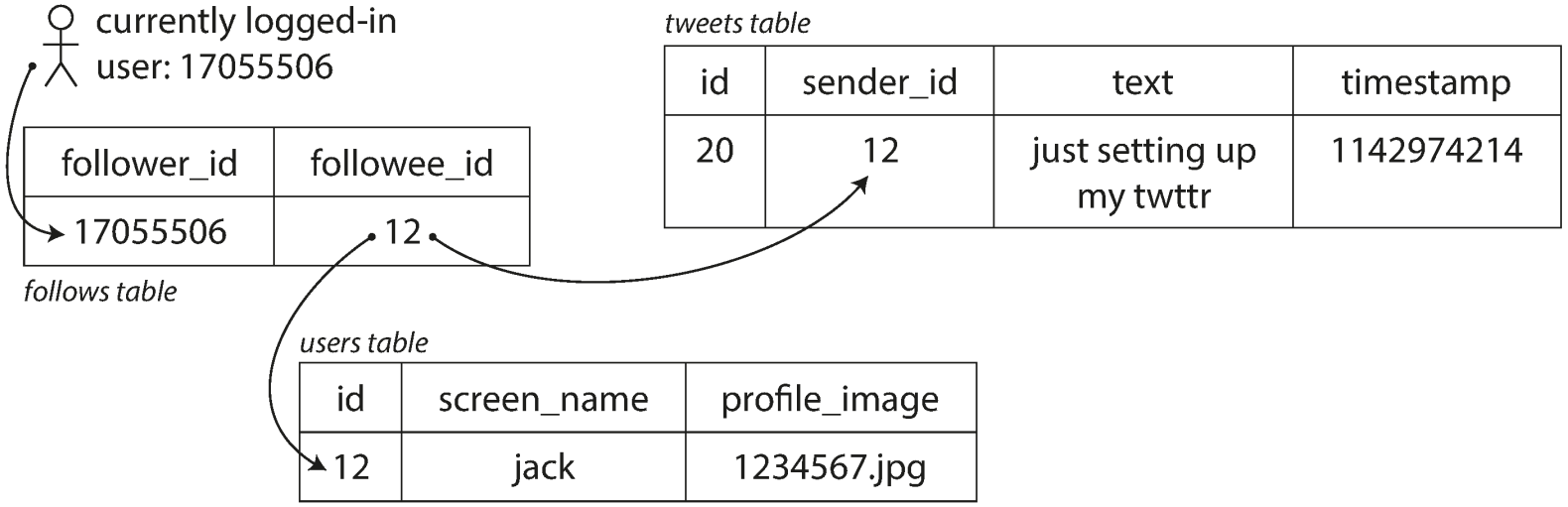
Vertical vs. Horizontal Scaling

- **Problem: How do we increase capacity to handle additional load?**
- **Vertical scaling** (*scaling up*): Get a more powerful machine!
- **Horizontal scaling** (*scaling out*): Distribute the load across multiple machines!
- **Q. What are the benefits & downsides of each approach?**
  - So does this mean horizontal scaling is always the better choice?
- In practice, most systems use a hybrid approach
  - Vertical scaling, where possible, is simpler and more efficient
  - **Example**: StackExchange Architecture

Distributed Data

Digression: Relational vs. Document Model

- **Relational data model**: Schemas, tables & queries (e.g., SQL)




```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

Digression: Relational vs. Document Model

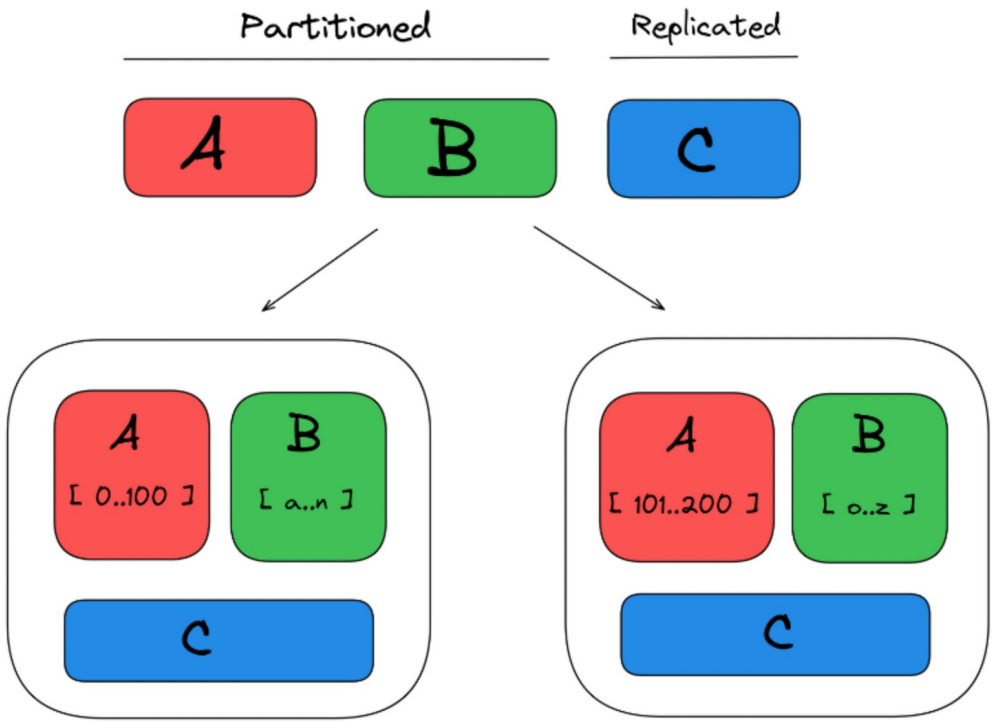
- **Relational data model**: Schemas, tables & queries
- **Document model**: No fixed schema, semi-structured (e.g., JSON/XML)

```
{
  "id": 2,
  "firstName": "Niels",
  "lastName": "Bohr",
  "address": {
    "streetName": "Flemsevej",
    "streetNumber": "31A",
    "city": "København"
  },
  "emergencyPhoneNumbers": []
},
...
```

Q. Benefits & drawbacks of each model?
When would you choose one over the other (in relation to scalability)?

Horizontal Scaling through Distributed Data

- Distribute load across multiple machines
  - Typically involves distributing & storing data across those machines
- Two ways to distribute data: **Replication** and **partitioning**
- Many systems use a hybrid approach that combines both



Distributed Data: Replication

- **Replication**: Copy & store data across multiple machines (or *nodes*), possibly in different locations.
  - **Fault-tolerant**: If some nodes become unavailable, data can be access from the remaining nodes
  - **Performance**: Requests can be directed to a node that is physically closer (reduced latency)
  - **Scalability**: Increased load can be handled by adding more nodes with replicated data

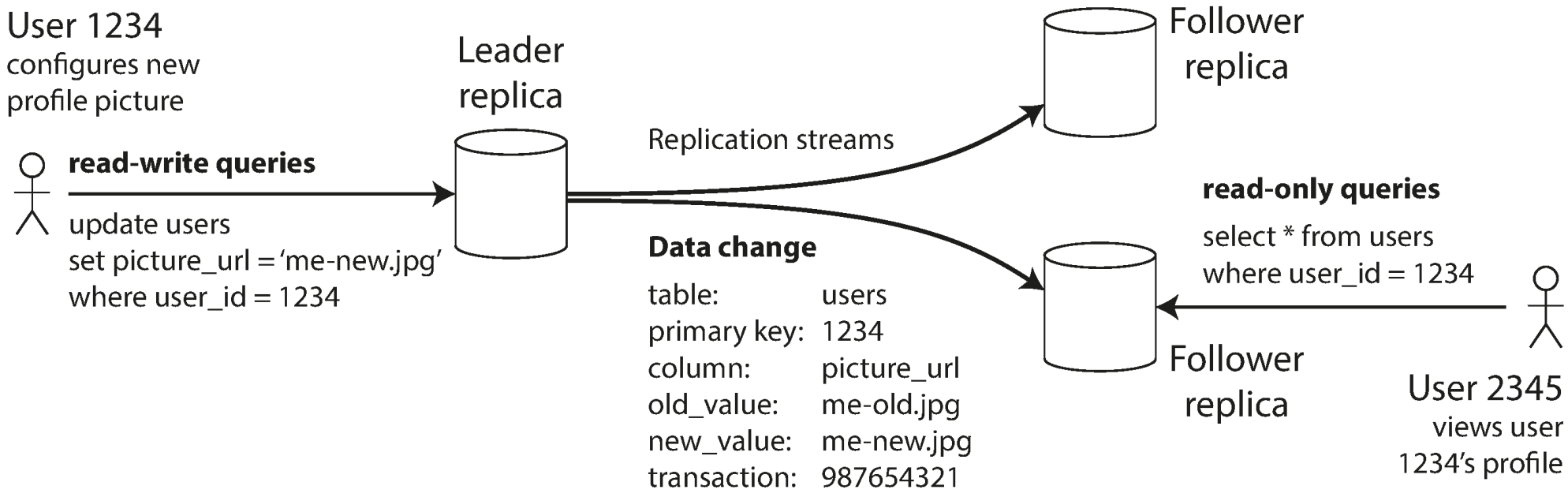
Q. This sounds great! What's the catch?

Replication: Challenges

- Consistency: If data on one node changes, how do we ensure that all its replicas have the same, consistent data?
  - Clients may read outdated data from inconsistent nodes
  - Node failures: What if some of the nodes fail before updating its data?
- There are several different approaches to dealing with these challenges
- This is an active area of research (called *distributed systems*); you can take multiple courses on this topic alone
- We will cover one well-known approach: **Leader-follower model**

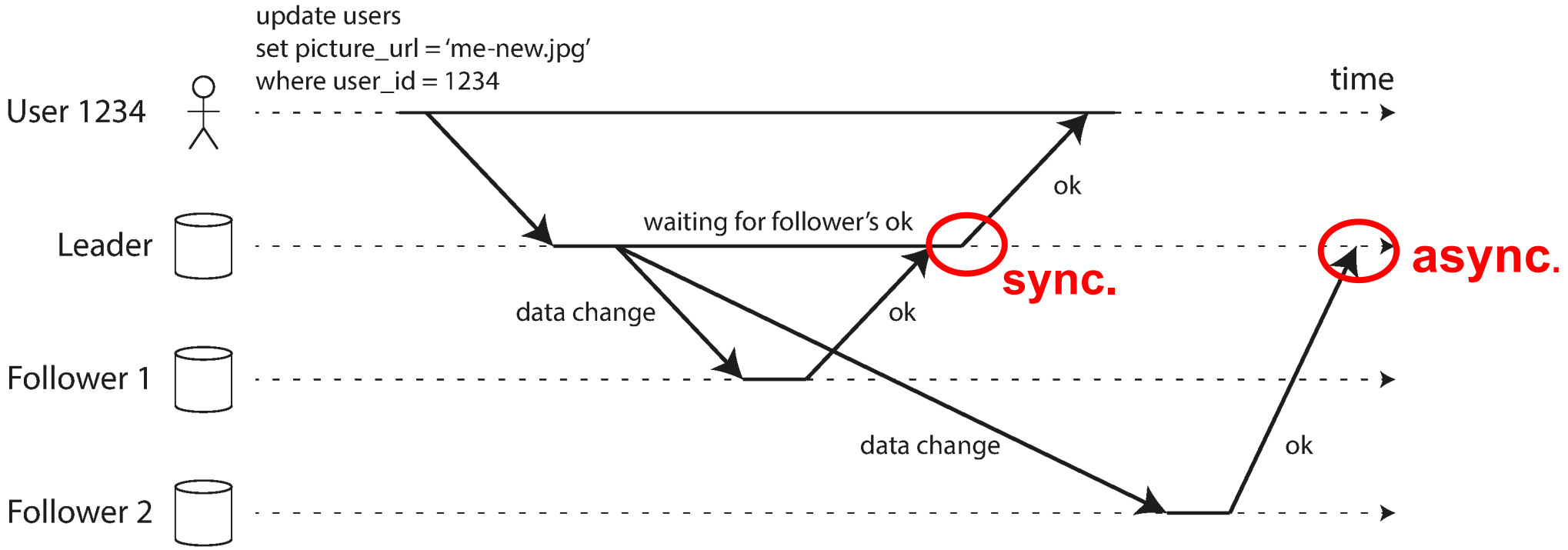
Leader-Follower Model

- Designate one of the replicas as the **leader**; the rest are **followers**
- Write operations are allowed only on the leader
- When data changes on the leader, send update to every follower



Synchronous vs. Asynchronous Replication

- **Synchronous**: The leader waits until the follower confirms that it has received the update
- **Asynchronous**: The leader sends the update and continues without confirmation



Synchronous vs. Asynchronous Replication

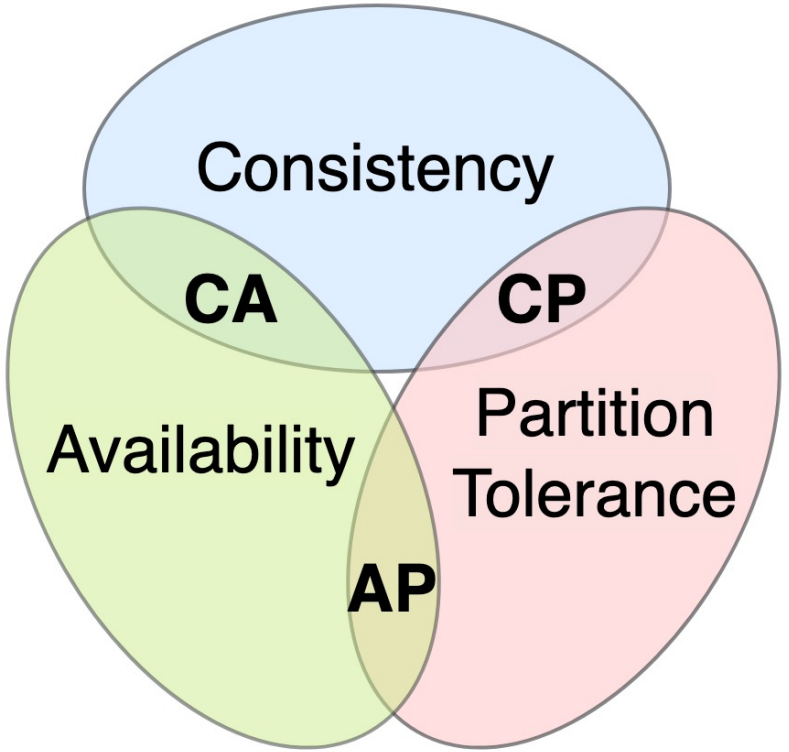
- **Synchronous**: The leader waits until the follower confirms that it has received the update
- **Asynchronous**: The leader sends the update and continues without confirmation
- **Q. What are the benefits & downsides of each design? What types of applications does one approach makes more sense over the other?**

Synchronous vs. Asynchronous Replication

- **Synchronous**: The leader waits until the follower confirms that it has received the update
  - Pros: Ensures that followers have updated data. If the leader fails, latest data can still be read from the followers
  - Cons: Higher latency for the client; some followers may fail and never return a confirmation
- **Asynchronous**: The leader sends the update and continues without confirmation
  - Pros: Higher performance; the leader can continue to process client requests
  - Cons: Weaker guarantees on consistency across replicas
- **Hybrid model:** Assign some followers to be synchronous, the others asynchronous

Digression: CAP Theorem

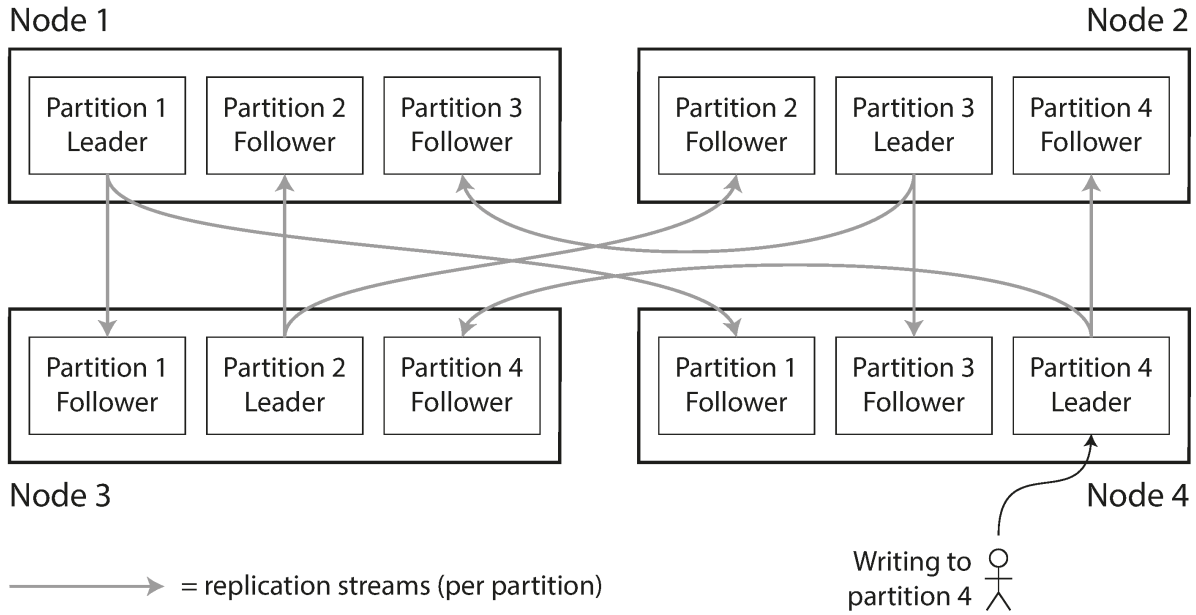
- **Consistency:** Clients always read the latest data
- **Availability:** Services are available for clients to access
- **Partition tolerance:** System continues to operate despite network failures
- **CAP theorem**: Choose two out of three
  - e.g., if a network failure occurs (and system tolerates it), choose consistency or availability



- Demonstrates trade-offs between different qualities of scalable systems
  - But somewhat controversial; some people argue it as being misleading

Distributed Data: Partitioning

- **Partitioning** (also called *sharding*): Split the data into smaller, independent units & distribute them across nodes
  - Useful and necessary when one dataset is too large to be fit onto a single node (i.e., replication alone is not sufficient!)
  - Usually combined with replication: Each partition is replicated stored across multiple nodes



Distributed Data: Partitioning

- **Partitioning** (also called *sharding*): Split the data into smaller, independent units & distribute them across nodes
  - Useful and necessary when one dataset is too large to be fit onto a single node (i.e., replication alone is not sufficient!)
  - Usually combined with replication: Each partition is replicated stored across multiple nodes
- **Design considerations**
  - How to partition the data (key-based vs. hash-based
  - How to rebalance partitions (when new data is added over time)
  - How to route client requests to the right partition (e.g., Zookeeper)
  - More details in the assigned reading (Chapter 6, Kleppman)

Exercise: Designing Movie Streaming Service

- Sketch a design of a movie streaming service (e.g., Netflix) that should scale to 1 million concurrent users with rebuffering rate of less than 2%
  - Focus on two operations: **Searching movies** and **playing a movie**
- **Questions to discuss**:
  - What types of data do we need store?
  - What are the major components in the system?
  - What type of scaling (vertical, horizontal, or both) do we apply?
  - How do we distribute data (replication, partitioning, or both)?
  - Synchronous vs. asynchronous replication?

Summary: Vertical vs. Horizontal Scaling

- Vertical and horizontal scaling are two major ways of adding capacity to a system
- Horizontal scaling typically involves distributing data across multiple nodes, to allow load to be divided among the machines
- Replication and partitioning are two common ways of distributing data
- Despite multiple benefits (performance, scalability, fault-tolerance), distributing data introduces new challenges into the design task
- Ultimately, improving scalability adds costs and complexity!
  - We should first ask: Do we actually need scalability right now? (More in the next lecture)

Summary

- Exit ticket!