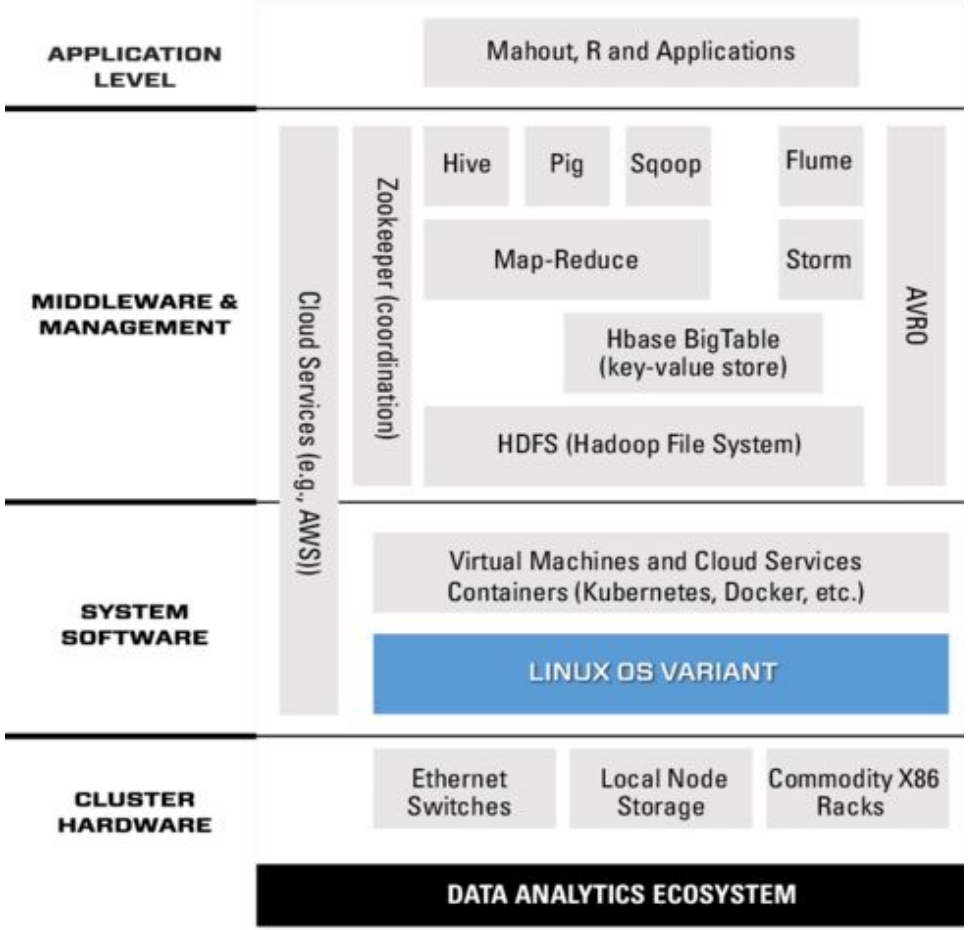
Large Scale Data Management

Ricardo Vilaça

rmvilaca@di.uminho.pt

https://rmpvilaca.github.io/

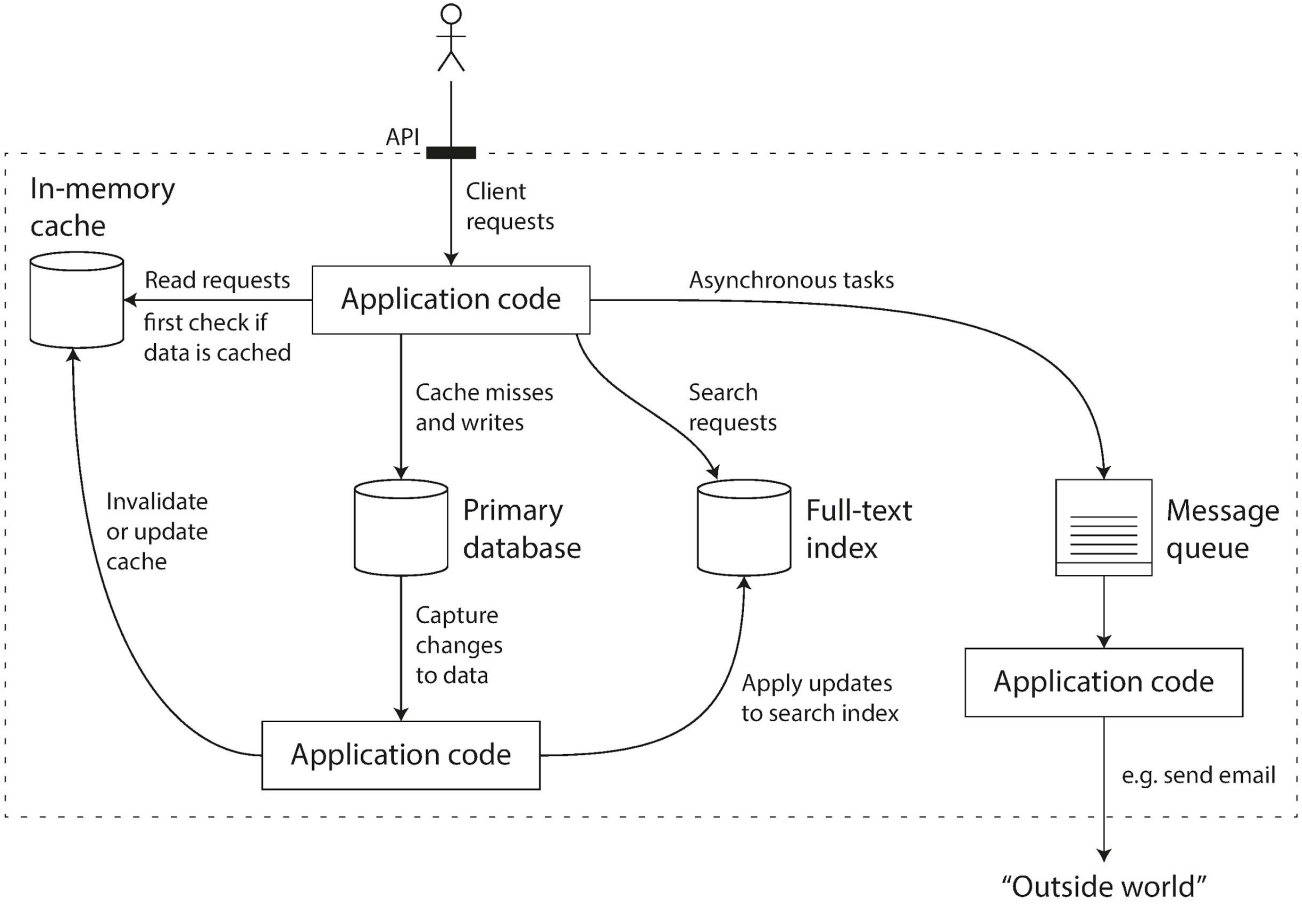
Reliable and Scalable Applications



Reed and Dongarra (2015). HDA:high-end data analysis.

Building Blocks

- A data-intensive application is typically built from standard building blocks that provide commonly needed functionality
  - Store data so that they, or another application, can find it again later (**databases**)
  - Remember the result of an expensive operation, to speed up reads (**caches)**
  - Allow users to search data by keyword or filter it in various ways (s**earch indexes**)
  - Send a message to another process, to be handled asynchronously (**stream processing**)
  - Periodically crunch a large amount of accumulated data (**batch processing**)



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Nonfunctional requirements

- Reliability
  - The system should continue to work correctly (performing the correct function at the desired level of performance) even in the face of adversity (hardware or software faults, and even human error)
- Scalability
  - As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth
- Maintainability
  - Majority of the cost of software is not in its initial development, but in its ongoing maintenance

Reliability

- The application performs the function that the user expected
- It can tolerate the user making mistakes or using the software in unexpected ways
- Its performance is good enough for the required use case, under the expected load and data volume
- The system prevents any unauthorized access and abuse

Fault-Tolerant

- The things that can go wrong are called faults
- Systems that anticipate faults and can cope with them are called **fault-tolerant** or **resilient**
- A **fault** is usually defined as one component of the system deviating from its spec, whereas a **failure** is when the system as a whole stops providing the required service to the user
- It is impossible to reduce the probability of a fault to zero
  - It is usually best to design fault-tolerance mechanisms that prevent faults from causing failures
  - Techniques for building reliable systems from unreliable parts
- In such fault-tolerant systems, it can make sense to increase the rate of faults by triggering them deliberately
  - Example: Netflix Chaos Monkey, https://github.com/Netflix/chaosmonkey

Types of Faults

- Unintended vs. malevolent failures
- Single vs. multiple failures
- Detectable vs. undetectable failures

Types of Faults

- Hardware Faults
  - Processor failures
    - Halt, delay, restart, erratic execution
  - Storage failures
    - Volatile vs. non-volatile storage failures
    - Atomic write violations, transient errors, localized vs. global failures
  - Network failures
    - Lost message, out-of-order messages, partitions, bounded delay
  - Until recently, redundancy of hardware components was sufficient for most applications, since it makes total failure of a single machine fairly rare
  - More applications have begun using larger numbers of machines, which proportionally increases the rate of hardware faults
  - There is a move toward systems that can tolerate the loss of entire machines, by using software fault-tolerance techniques in preference or in addition to hardware redundancy

Types of Faults

- Software Errors
  - Another class of fault is a systematic error within the system. A software bug that causes every instance of an application server to crash when given a particular bad input
    - A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth
    - A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.
    - Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults
  - There is no quick solution to the problem of systematic faults in software
    - Carefully thinking about assumptions and interactions in the system; thorough testing; process isolation; allowing processes to crash and restart; measuring, monitoring, and analyzing system behavior in production

Types of Faults

- Human Errors
  - Configuration errors by operators were the leading cause of outages, whereas hardware faults (servers or network) played a role in only 10–25% of outages
  - How do we make our systems reliable, in spite of unreliable humans?
    - Design systems in a way that minimizes opportunities for error
    - Decouple the places where people make the most mistakes from the places where they can cause failures
    - Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests
    - Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure
    - Set up detailed and clear monitoring, such as performance metrics and error rates
    - Implement good management practices and training

Importance

- Reliability is not just for nuclear power stations and air traffic control software
- Bugs in business applications cause lost productivity
- Outages of ecommerce sites can have huge costs in terms of lost revenue and damage to reputation
- Sacrifice reliability in order to reduce development cost or operational cost

Scalability

- If a system is working reliably today, that doesn't mean it will necessarily work reliably in the future
  - One common reason for degradation is increased load
- System's ability to cope with increased load
- If the system grows in a particular way, what are our options for coping with the growth?
- How can we add computing resources to handle the additional load?

Load

- Load parameters
  - Requests per second to a web server
  - The ratio of reads to writes in a database
  - The number of simultaneously active users in a chat room
  - The hit rate on a cache
  - Others?
  - Average case is what matters for you, or perhaps your bottleneck is dominated by a small number of extreme cases

Performance

- When you increase a load parameter and keep the system resources (CPU, memory, network bandwidth, etc.) unchanged, how is the performance of your system affected?
- When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged
- In a batch processing system such as Hadoop, we usually care about **throughput**
  - The number of records we can process per second
- In online systems, what's usually more important is the service's **response time**
  - The time between a client sending a request and receiving a response
  - Think of response time not as a single number, but as a **distribution of values**

Response Time

- Mean is not a very good metric if you want to know your “typical” response time
  - Doesn’t tell you how many users actually experienced that delay
- Percentiles are better
  - Take your list of response times and sort it from fastest to slowest, then the median is the halfway point
    - Half of user requests are served in less than the median response time
  - To figure out how bad your outliers you can look at higher percentiles
    - 95th, 99th, and 99.9th percentiles are common (abbreviated p95, p99, and p999)
- Amazon Example
  - The customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases
  - Amazon has also observed that a 100 ms increase in response time reduces sales by 1%

Queuing

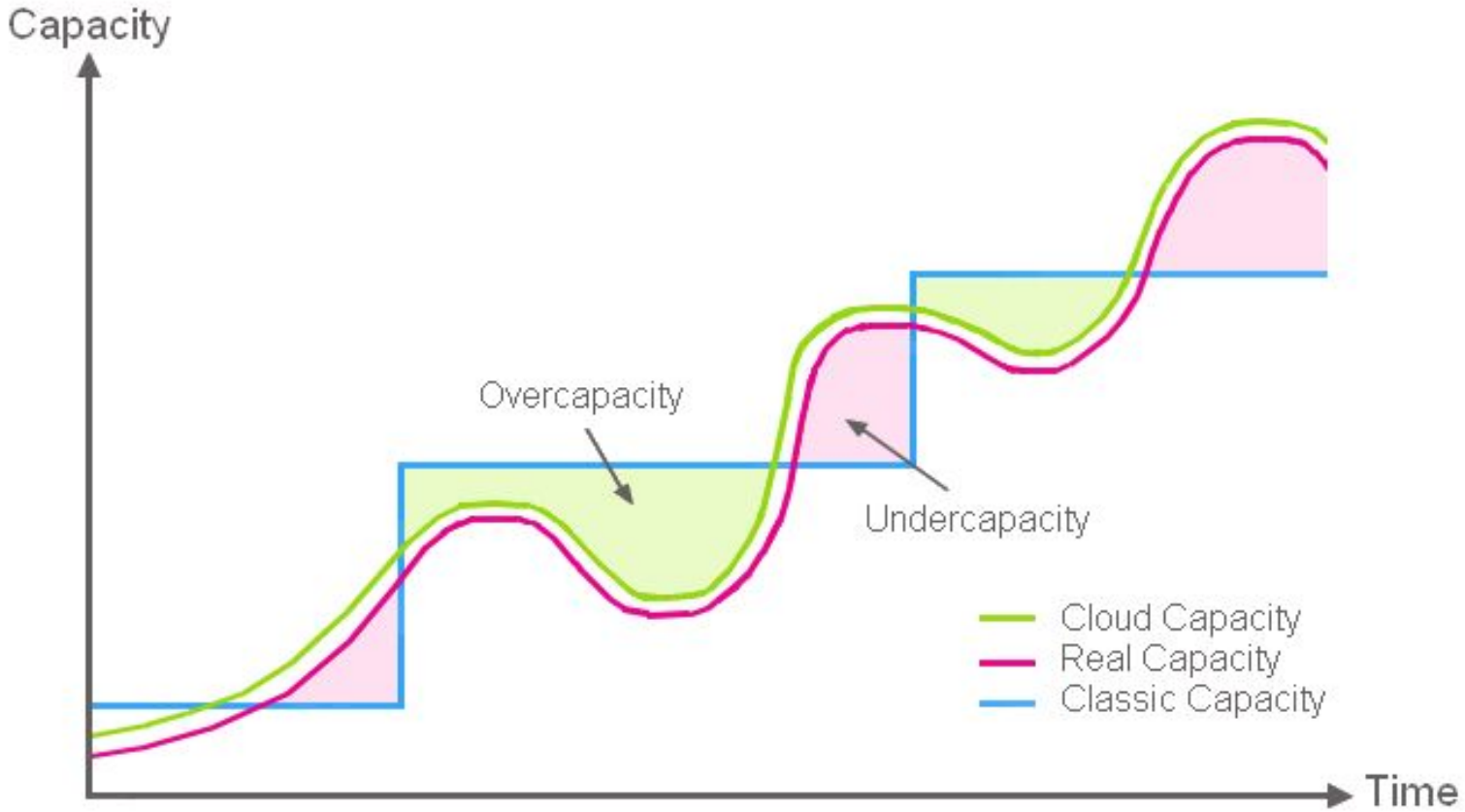
- Queueing delays often account for a large part of the response time at high percentiles
- A server can only process a small number of things in parallel
  - A small number of slow requests hold up the processing of subsequent requests
    - **head-of-line blocking**
- When generating load artificially in order to test the scalability of a system, the load-generating client needs to keep sending requests independently of the response time

Solutions

- Scaling up
  - Vertical scaling, moving to a more powerful machine
- Scaling out
  - Horizontal scaling, distributing the load across multiple smaller machines
    - Shared-nothing architecture
- A system that can run on a single machine is often simpler
  - But high-end machines can become very expensive, so very intensive workloads often can’t avoid scaling out.
- Good architectures usually involve a pragmatic mixture of approaches

Elasticity

- Systems that can automatically add computing resources when they detect a load increase
- An elastic system can be useful if load is highly unpredictable
- Manually scaled systems are simpler and may have fewer operational surprises



https://elasticsecurity.wordpress.com/2013/09/11/how-to-detect-side-channel-attacks-in-cloud-infrastructures/

Scalability

- Distributing stateless services across multiple machines is fairly straightforward
- Taking stateful data systems from a single node to a distributed setup can introduce a lot of additional complexity
- Distributed data systems are becoming the norm
- Highly specific to the application—there is no such thing as a generic, one-size-fits-all scalable architecture
  - The volume of reads, the volume of writes, the volume of data to store, the complexity of the data, the response time requirements, the access patterns, or (usually) some mixture of all of these plus many more issues
  - Built from general-purpose building blocks, arranged in familiar patterns

Distributed Systems

Need for distributed systems

- Scalability
  - If your data volume, read load, or write load grows bigger than a single machine can handle
- Fault tolerance/high availability
  - If your application needs to continue working even if one machine (or several machines, or the network, or an entire datacenter) goes down
- Latency
  - If you have users around the world, you might want to have servers at various locations worldwide so that each user can be served from a datacenter that is geographically close to them

Replication Versus Partitioning/Sharding

- Replication
  - Keeping a copy of the same data on several different nodes, potentially in different locations
  - Provides redundancy: if some nodes are unavailable, the data can still be served from the remaining nodes
  - Replication can also help improve performance
- Partitioning
  - Splitting a big database into smaller subsets called partitions so that different partitions can be assigned to different nodes (also known as sharding)

Shared Nothing Architecture

- Bunch of machines connected by a network
  - Each machine or virtual machine running the database software is called a node
  - Each node uses its CPUs, RAM, and disks independently
  - Any coordination between nodes is done at the software level, using a conventional network
- Has become the dominant approach for building internet services
  - It's comparatively cheap because it requires no special hardware
  - It can make use of commoditized cloud computing services
  - Can achieve high reliability through redundancy across multiple geographically distributed datacenters
- You need to be aware of the constraints and trade-offs that occur in such a distributed system

Why replication?

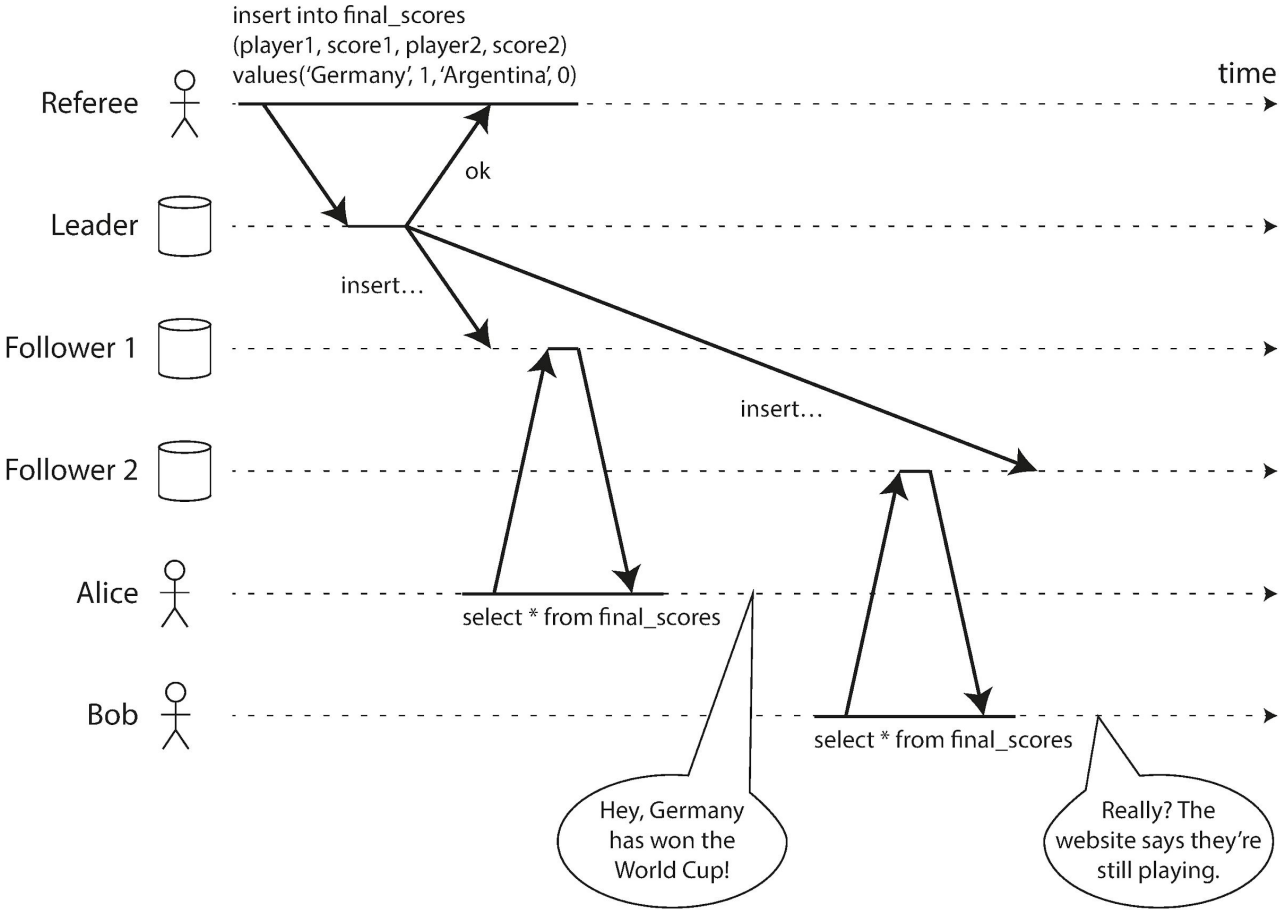
- To keep data geographically close to your users
  - Reduce latency
- To allow the system to continue working even if some of its parts have failed
  - Increase availability
- To scale out the number of machines that can serve read queries
  - Increase read throughput

Replication

- Replication is the solution to achieve fault tolerance
- Software based replication is an economically appealing technology to provide generic fault-tolerant services
- Replication is usually expected to be transparent to the user
  - transparency impacts on performance and it’s not always desirable
- Services are replicated to increase their dependability and often their performance
  - These goals are often conflicting
- Replica consistency is a key issue. Consistency enables reliability, availability and performance trade-offs

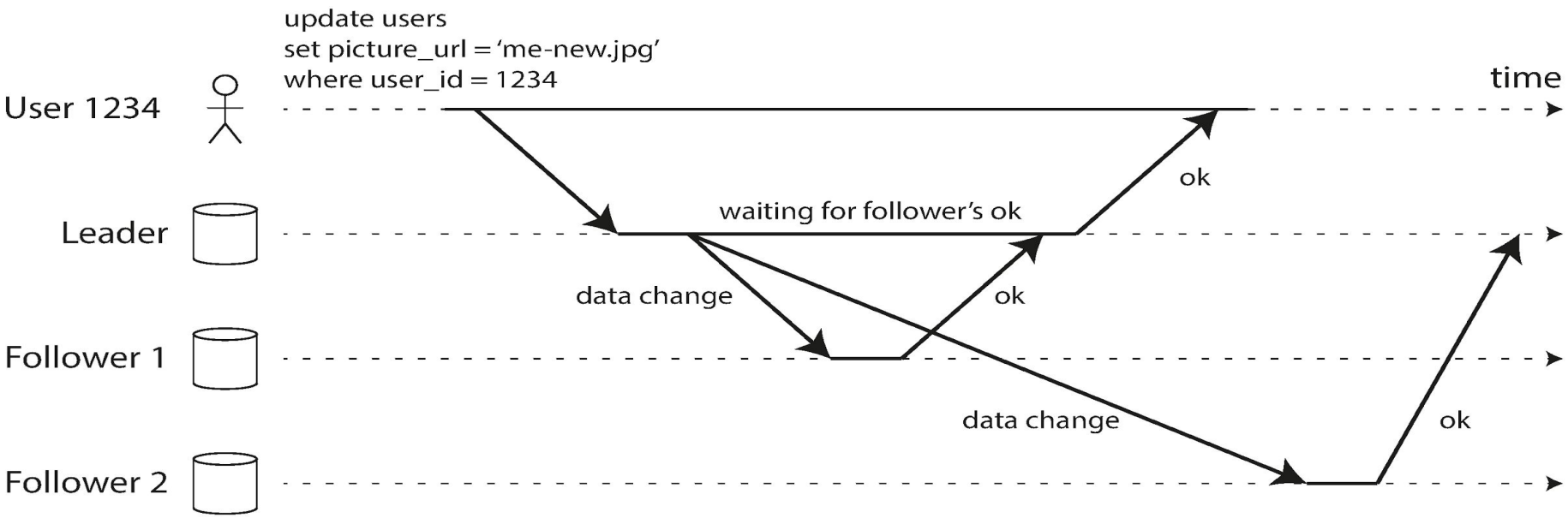
Replication

- Replication of immutable data is easy
  - Copy data to every node once
- Difficulty in replication lies in handling changes to replicated data
  - How consistent shall the replicas be?
  - What aspects shall the replication protocol favor?
  - What fault model shall be considered?



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Synchronous vs asynchronous



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Synchronous vs asynchronous

- Synchronous replication
  - Ensures consistency
  - May have high overhead
- Synchronous replication
  - Can be faster
  - Inconsistency
  - Staleness

Propagation (What)

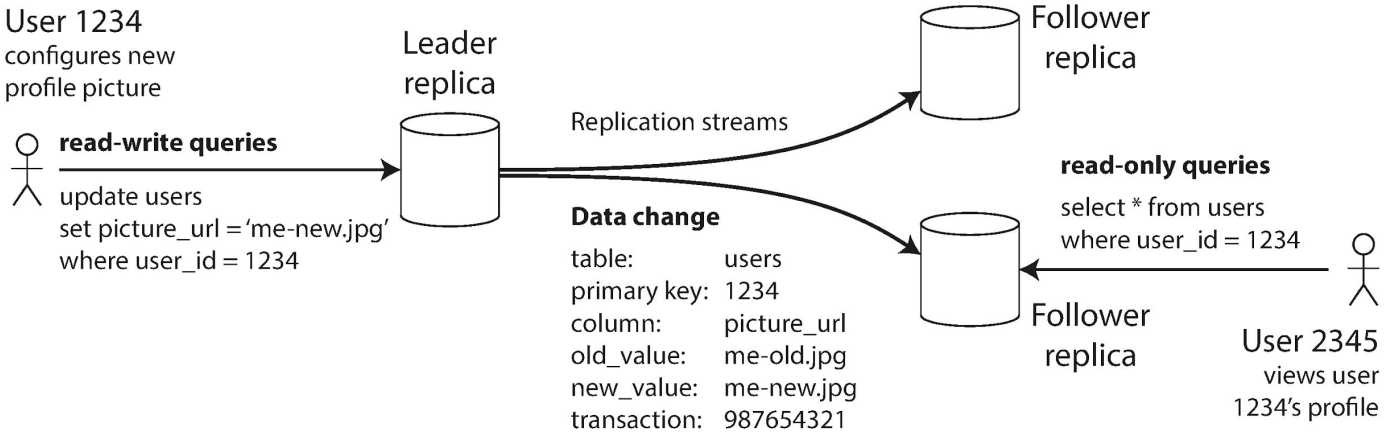
- How does replication work under the hood
  - Statement-based replication
    - Leader logs every write request (statement) that it executes and sends that statement log to its followers
    - Statements can call nondeterministic function
    - Example: VoltDB
  - Write-ahead log (WAL) shipping
    - The log is an append-only sequence of bytes containing all writes to the database
    - We can use the exact same log to build a replica on another node
    - Replication closely coupled to the storage engine
    - Examples: PostgreSQL and Oracle

Propagation (What)

- How does replication work under the hood
  - Logical (row-based) log replication
    - More easily be kept backward compatible
    - Easier for external applications to parse
    - Example: MySQL's binlog with row-based replication
  - Trigger-based replication
    - Use a trigger to log changes, from which it can be read by an external process
    - Has greater overheads than other replication methods
    - Is more prone to bugs and limitations
    - Example: Oracle GoldenGate

Passive Replication/Leader-based/ Master–Slave/ Primary Backup

1. Request goes to a distinguished replica - the primary
2. The primary processes the request, updates its own state, and sends a state update message to all other replicas
3. Each replica updates its own state, and sends an acknowledgment to the primary
4. The primary returns the response to the client



Passive Replication

- Passive replication is asymmetric: Only one replica actually processes the requests. Execution does not need to be deterministic
- The failure of the primary may not be transparent to the clients
- The choice of the primary, as well as the failure and reintegration of the replicas has to be dealt by the replication protocol

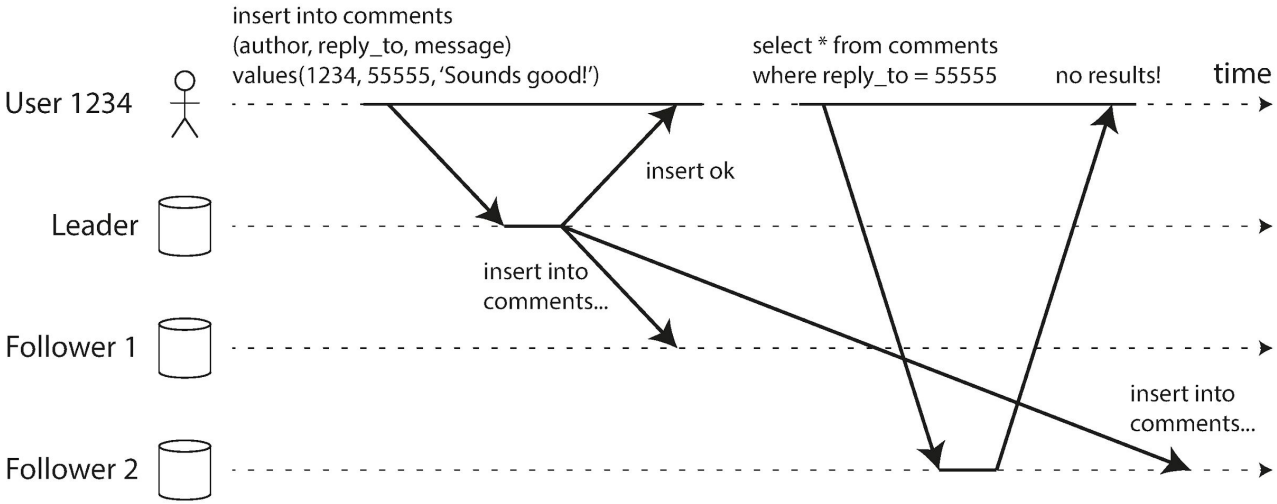
Consistency Criteria

- How consistent shall the replicas be?
- How to express replica consistency?
- The requirement of strictly consistent replicas could be stated as:
  - **Any read operation returns the value corresponding to the most recent written value**
  - Relies on absolute global time. Cannot be achieved in a distributed system
- Consistency criteria are usually stated as a set of predicates over system´s executions

Replication Lag

- Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica
- In this read-scaling architecture, you can increase the capacity for serving read-only requests simply by adding more followers
- If an application reads from an asynchronous follower, it may see outdated information

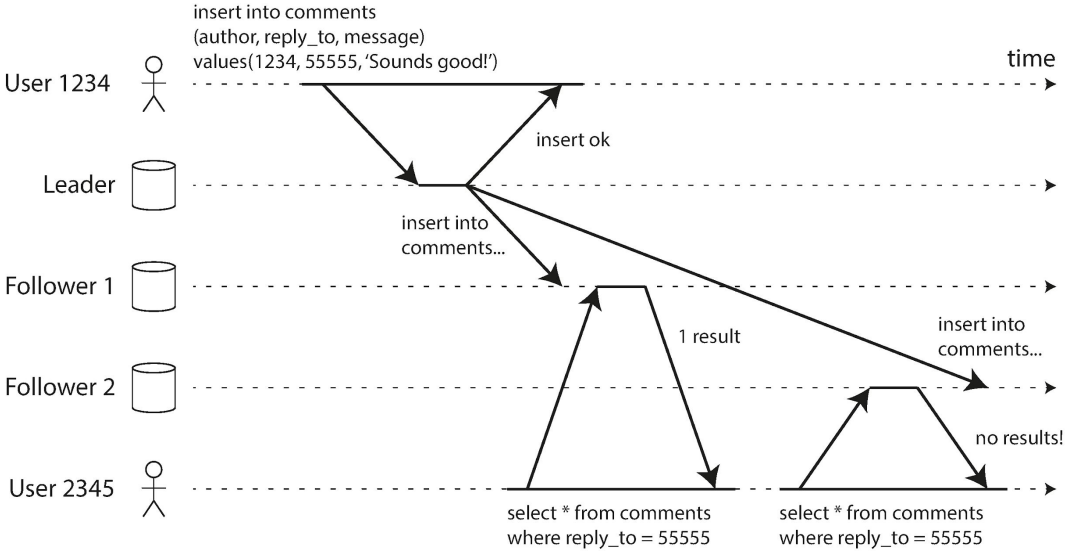
Read-after-write consistency



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

- Users should always see data that they submitted themselves

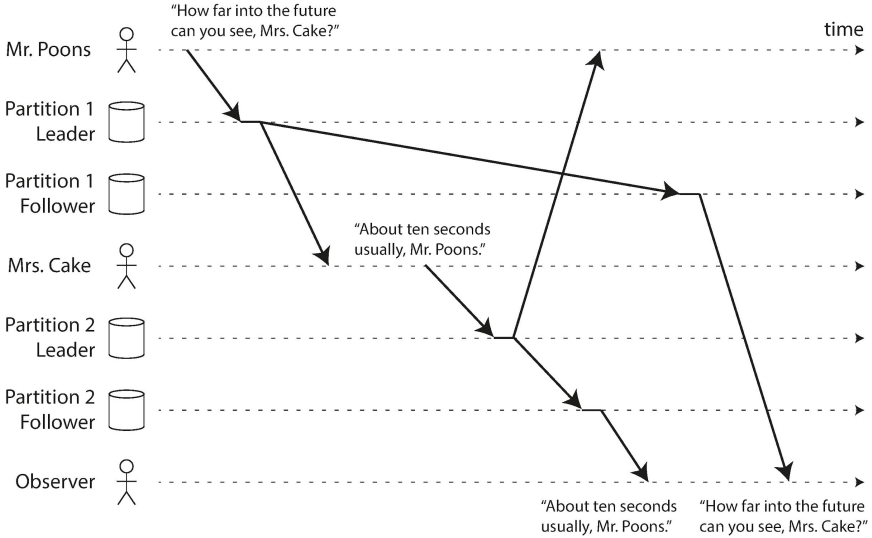
Monotonic reads



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

- After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time

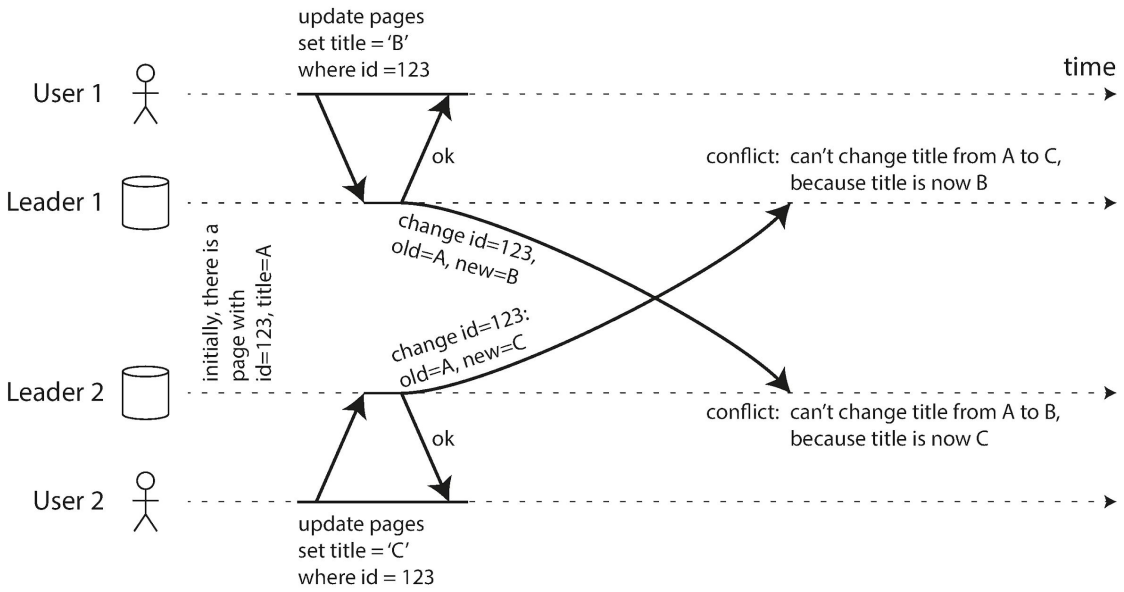
Consistent prefix reads



M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

- Users should see the data in a state that makes causal sense: for example, seeing a question and its reply in the correct order

Active Replication/Multi Leader/Multi Master



1. Request goes to all the replicas
2. Each replica processes the request, updates its own state, and returns the response to the client
3. Client waits until it receives the first response (or a majority of identical responses)

M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Active Replication

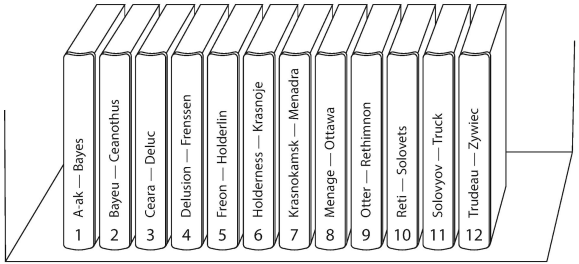
- Active replication requires request execution to be deterministic
- The failure of a replica tends to be transparent for the clients
- The failure and reintegration of the replicas has however to be dealt by the replication protocol

Partitioning

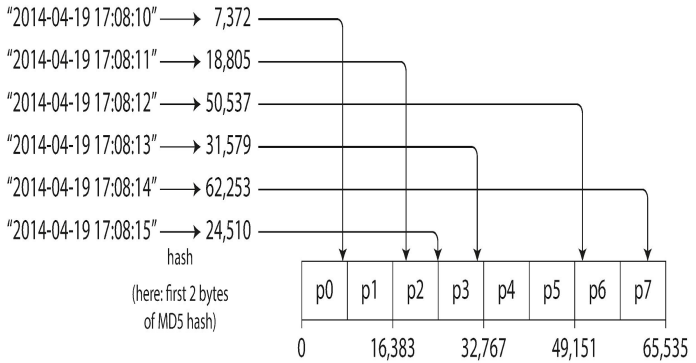
- Ideally split horizontally data such that each shard is responsible for:
  - The same amount of data
  - The same amount of traffic
- Distribute and shard parts over machines
  - Still fast traversal and read to keep related data together
  - Scale out instead scale up
  - Parallelizable data distribution and processing is key
- Avoid naïve hashing for sharding
  - Do not depend on the number of nodes
  - But difficult add/remove nodes
  - Trade off – data locality, consistency, availability, read/write/search speed, latency etc.
- Usually combined with replication so that copies of each partition are stored on multiple nodes

Partitioning

- Range
  - Range scans
  - Easier to reconfigure by splitting



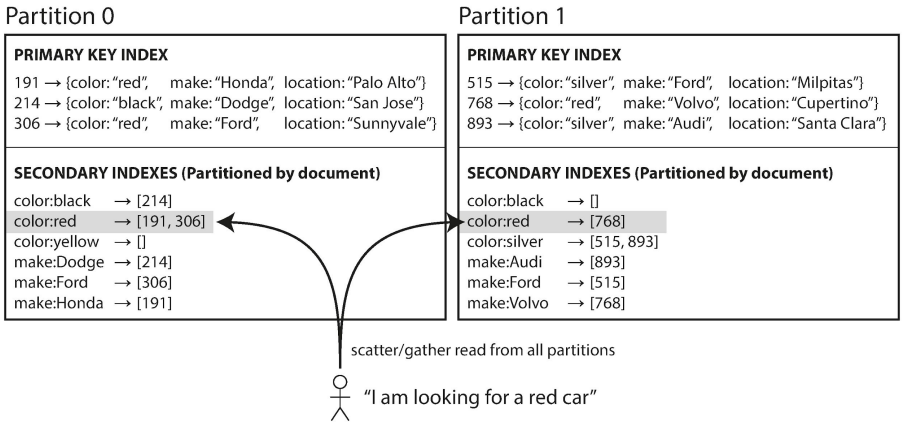
- Hash
  - Fast lookup
  - Better load distribution

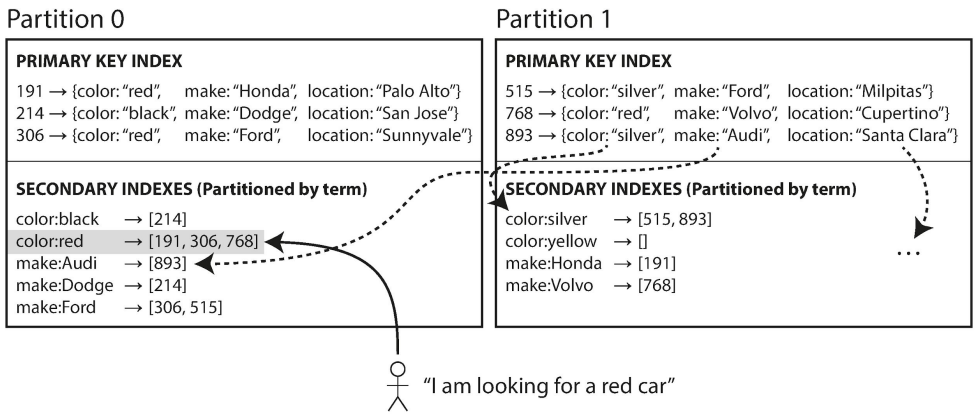


M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Partitioning Secondary Indexes

- Local
- Global

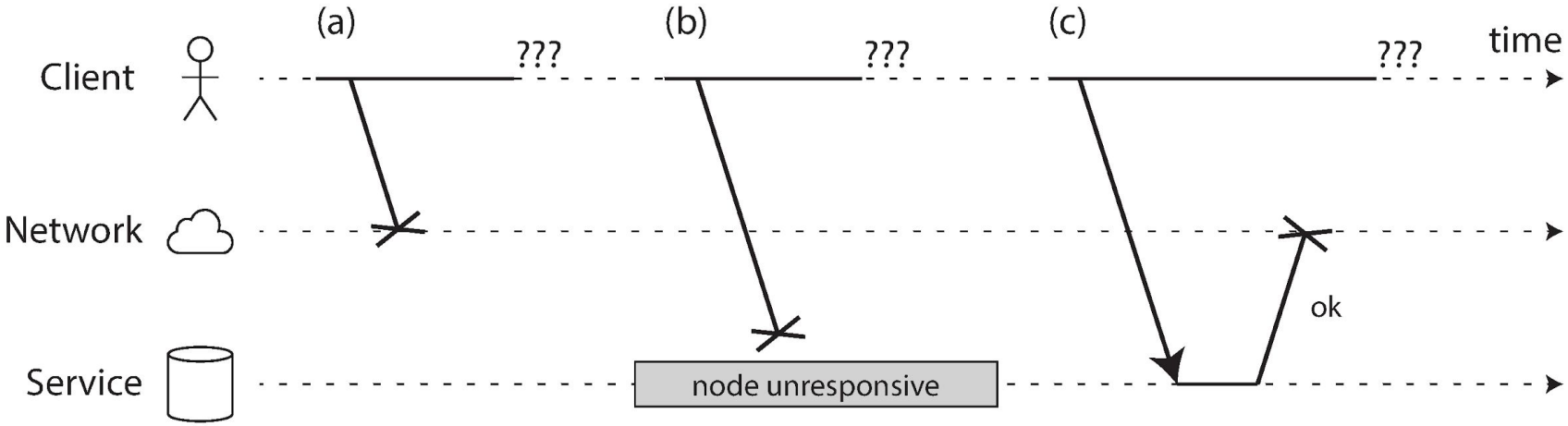




M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Unreliable networks

- Internet and most internal networks in datacenters (often Ethernet) are asynchronous packet networks



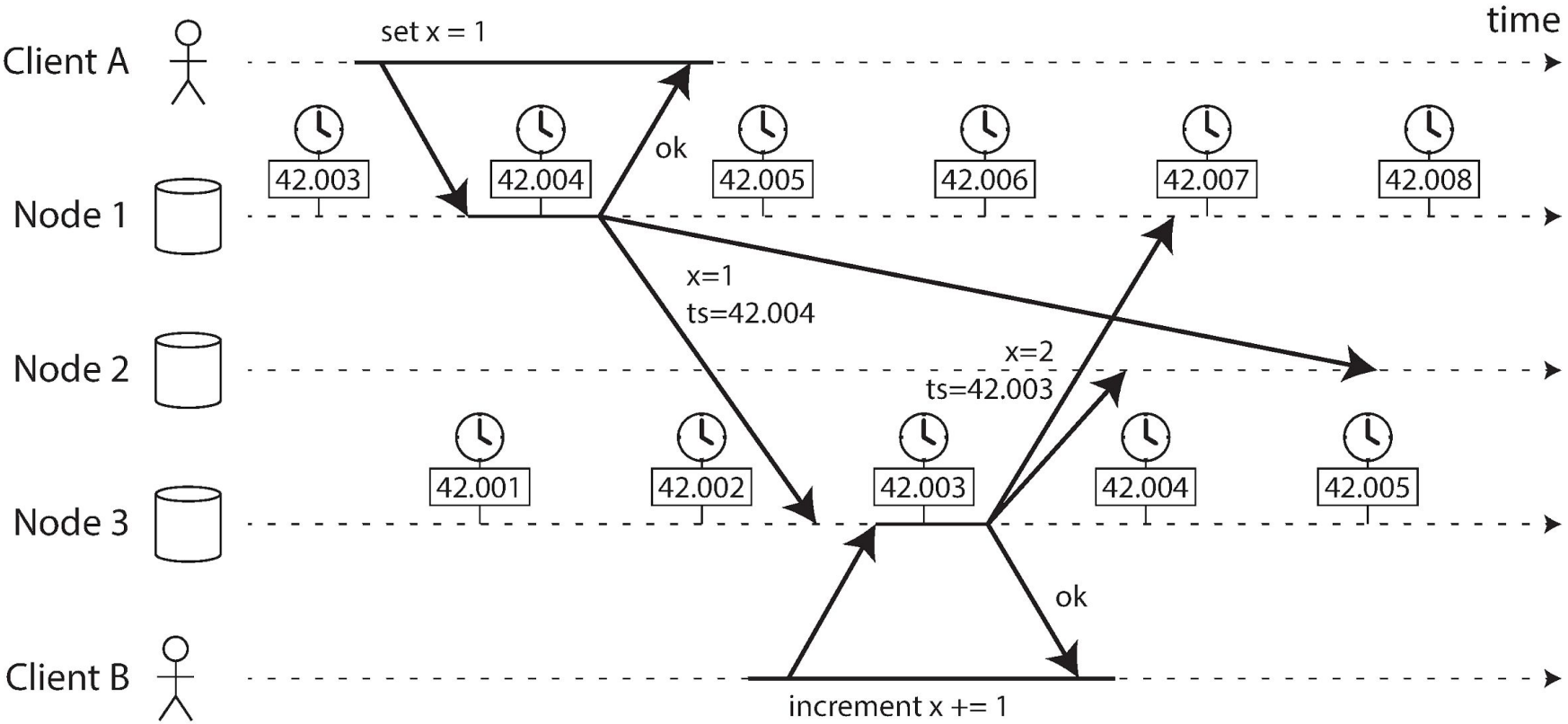
M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Unreliable Clocks

- Clocks and time are important
- Applications depend on clocks in various ways
- Time is a tricky business
  - Communication is not instantaneous
  - It takes time for a message to travel across the network from one machine to another
- Variable delays in the network
- Difficult to determine the order in which things happened when multiple machines are involved
- Each machine on the network has its own clock

Clock Synchronization and Accuracy

- Clocks need to be set according to an NTP server or other external time source in order to be useful
  - NTP synchronization can only be as good as the network delay
  - Some NTP servers are wrong or misconfigured
  - In virtual machines, the hardware clock is virtualized, which raises additional challenges
- Is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources
  - GPS receivers, the Precision Time Protocol(PTP)
  - Google TrueTime
    - Example: Google Spanner



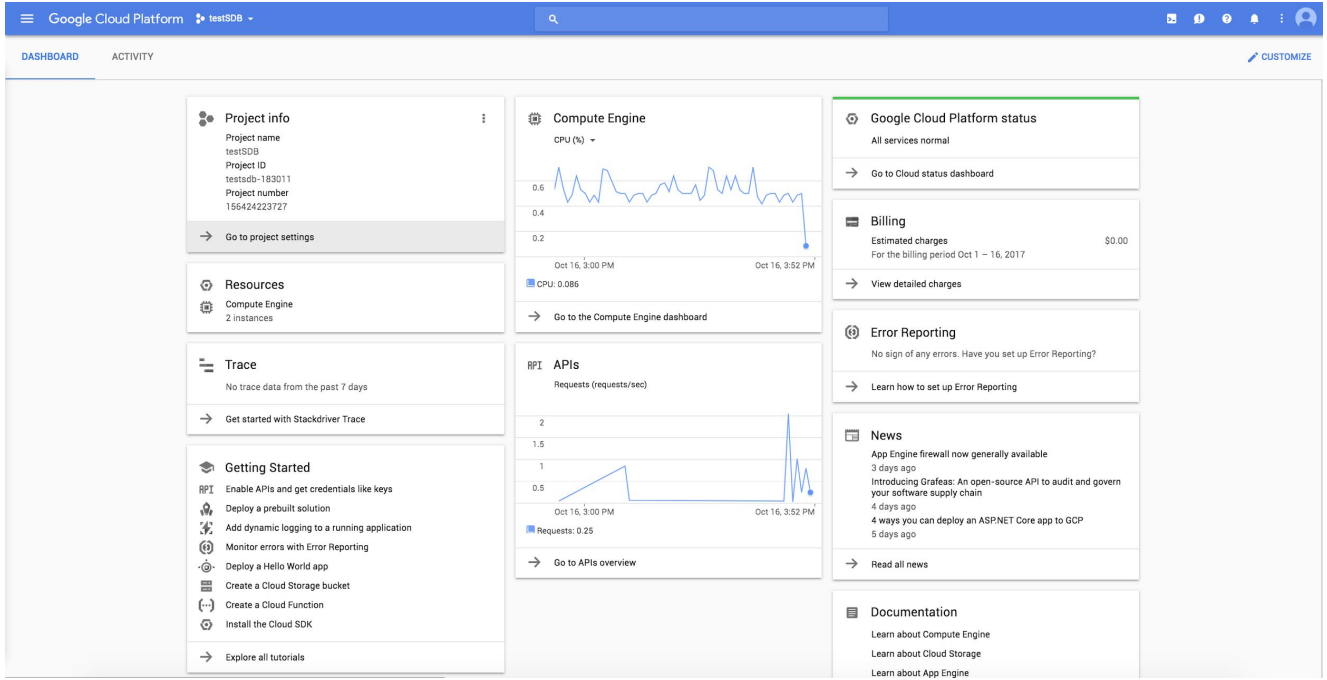
M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Logical Clocks

- Are based on incrementing counters
  - Do not measure the time of day or the number of seconds elapsed
  - Only the relative ordering of events
    - Whether one event happened before or after another
- Safer alternative for ordering events
- Examples: Riak

Google Cloud Platform

Google Cloud Platform
https://cloud.google.com

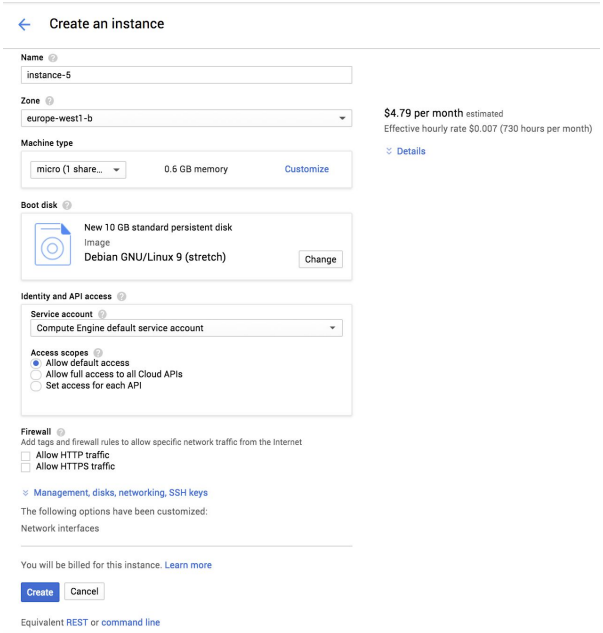


Google Cloud Platform

- Create project
  - Associate billing account (check e-mail for coupon)
  - Each account has 50$
- For project you may add team members (as Owners)
  - Side Bar -> IAM

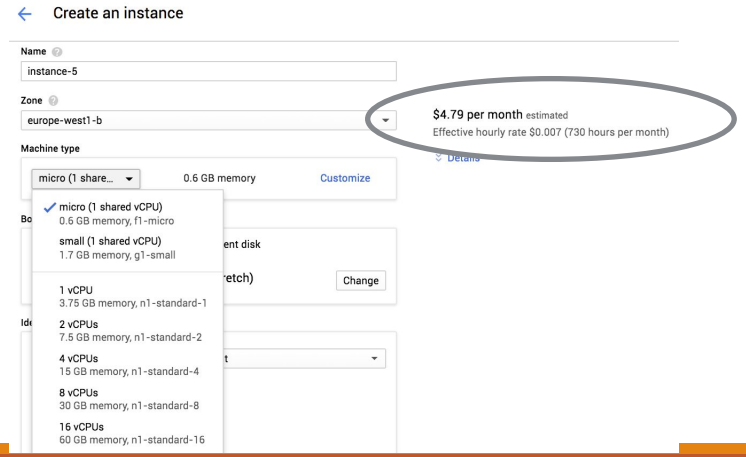
Google Cloud Platform
Create VM instance

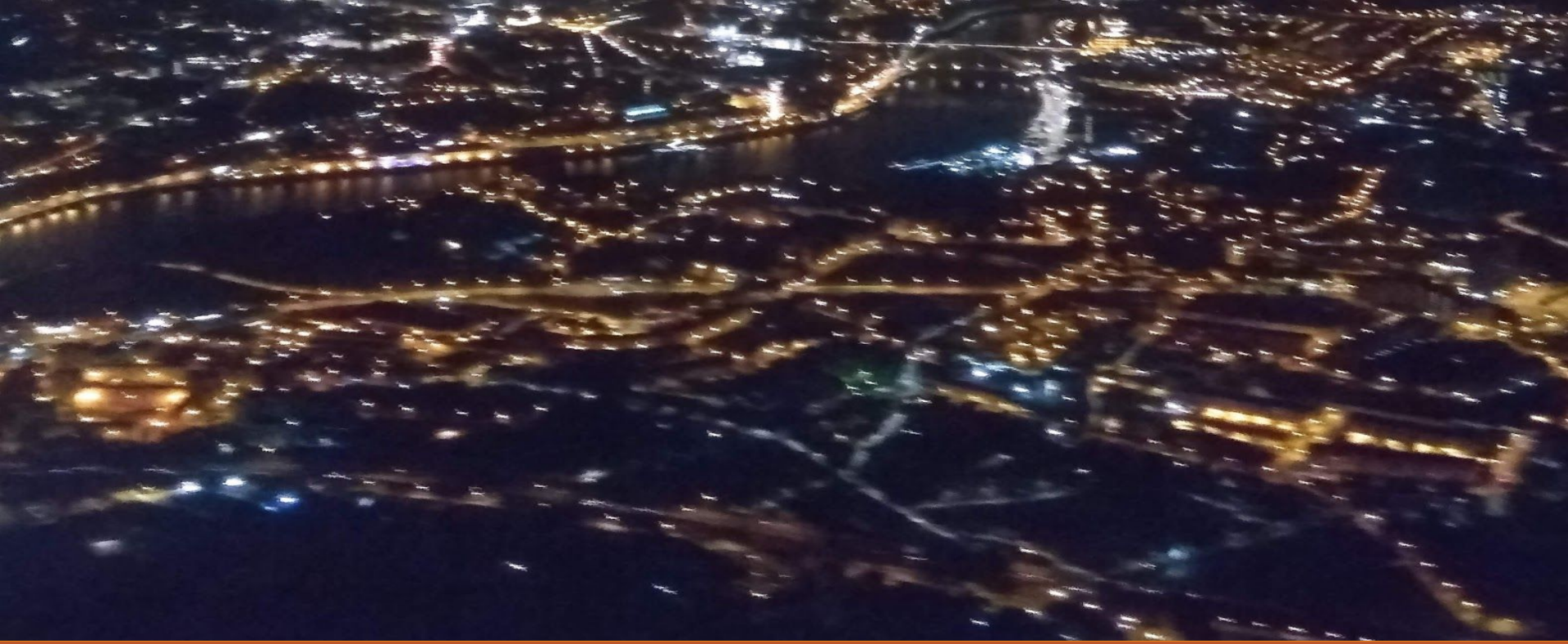
- Side Bar -> Compute Engine -> VM instances
- Create Instance
- Configure SSH Key (Metadata)



Google Cloud Platform
Create VM instance

- Each Zone has different prices
- Manage the budget wisely!!
- Advice: for testing purposes use micro instances, for the demo use small or 1vCPU instances
- Never forget to turn off/ STOP the VMs when not using them to save money…





Large Scale Data Management

rmvilaca@di.uminho.pt