CSE 106

Lecture 15 – Scalable Web App

Acknowledgement:

http://tutorials.jenkov.com/software-architecture/scalable-architectures.html

https://igotanoffer.com/blogs/tech/caching-system-design-interview

https://www.conceptatech.com/blog/dos-donts-designing-scalable-architecture

https://www.ibm.com/cloud/learn/microservices

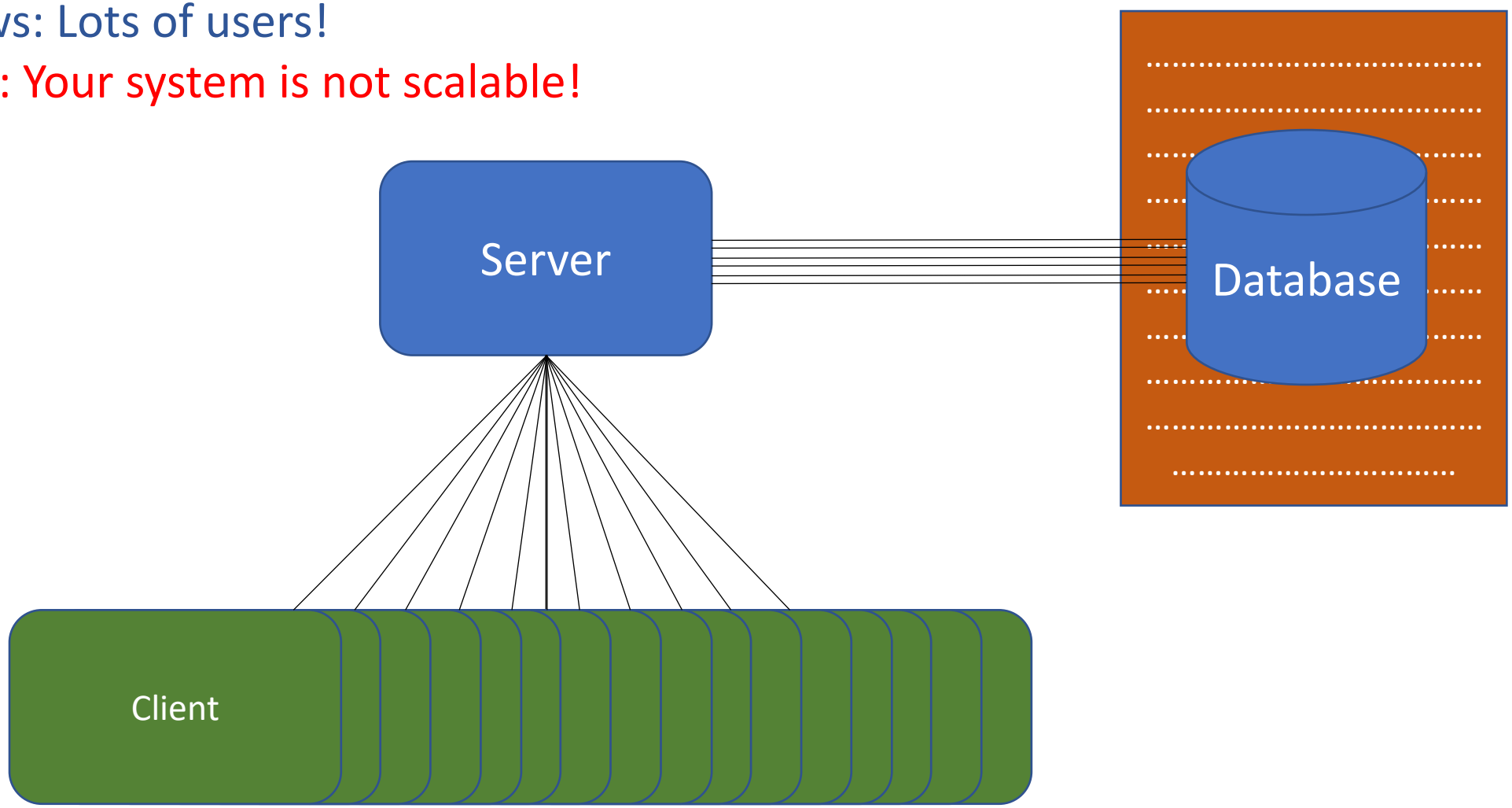
Scalable Architecture

- An architecture that can scale up to meet increased work loads
- If the workload suddenly exceeds the capacity of your system, you can scale up to meet the increased workload
- Typically refers to the backend of your system
- Considers lots of users (i.e. requests) and lots of data stored

Scalable?

Good News: Lots of users!

Bad News: Your system is not scalable!

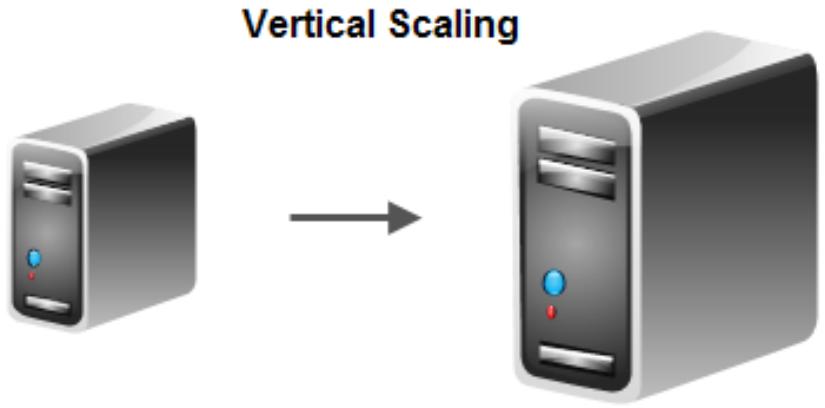


Principles of Scalability

- Availability
  - The system should be available for use as much as possible (ideally, always)
  - Uptime percentage has the most immediate effect on customer experience
- Performance
  - The system maintains a high level of performance even under heavy loads
  - Needs to be fast enough to have a good user experience
- Reliability
  - The system must accurately store, retrieve, and edit data under stress
  - It should return the most current data on request

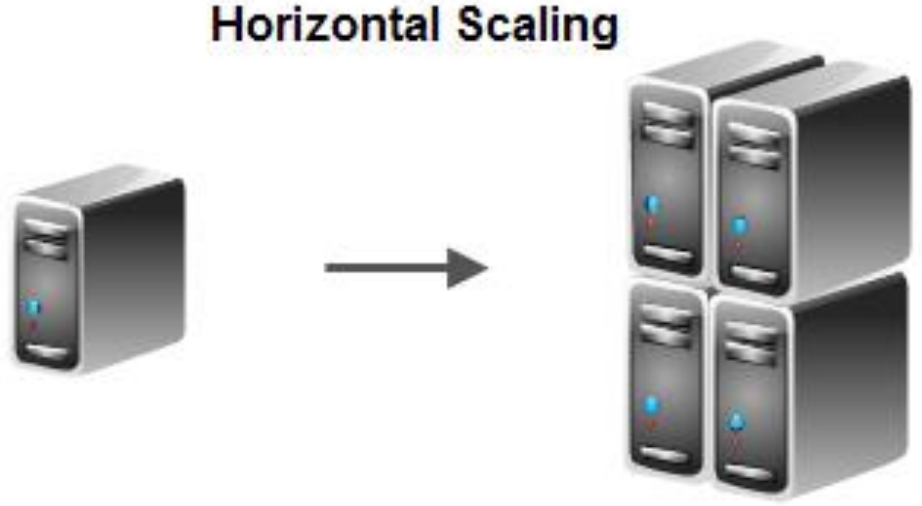
Vertical Scaling

- Upgrade your hardware
  - Better CPU
  - More memory
  - Faster and larger hard disk
- Pro: Easy to do
- Cons:
  - Expensive
  - Limited scalability
  - Requires downtime to scale
- Vertical scaling is NOT recommended



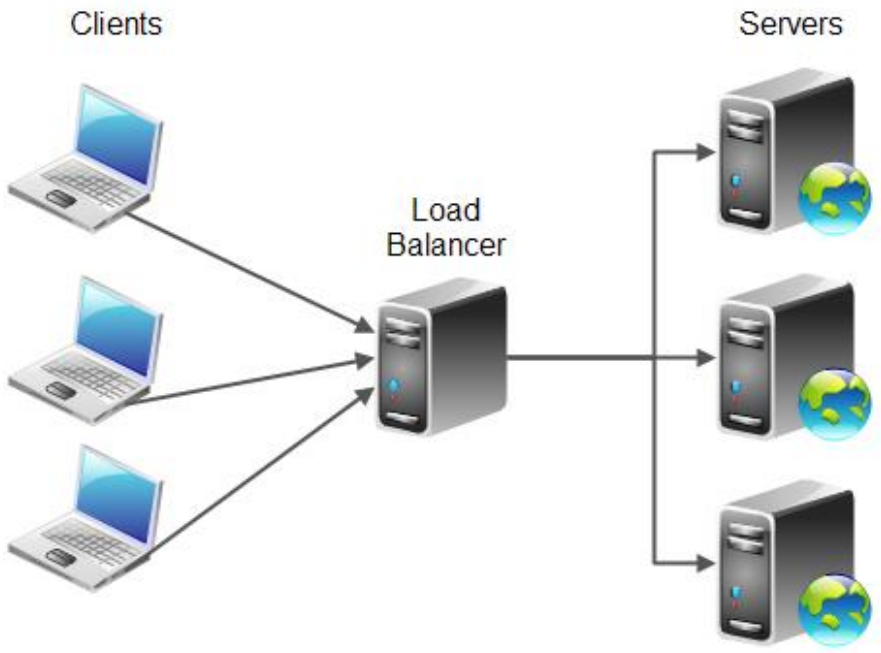
Horizontal Scaling

- Run the server on more computers (distributed)
- Con: More complicated architecture
- Pros:
  - Theoretically no limit to scale
  - No downtime needed to scale
  - Lower cost
  - Redundancy



Load balancing

- Distributes the workload of an application onto multiple computers
- Provides redundancy in your application

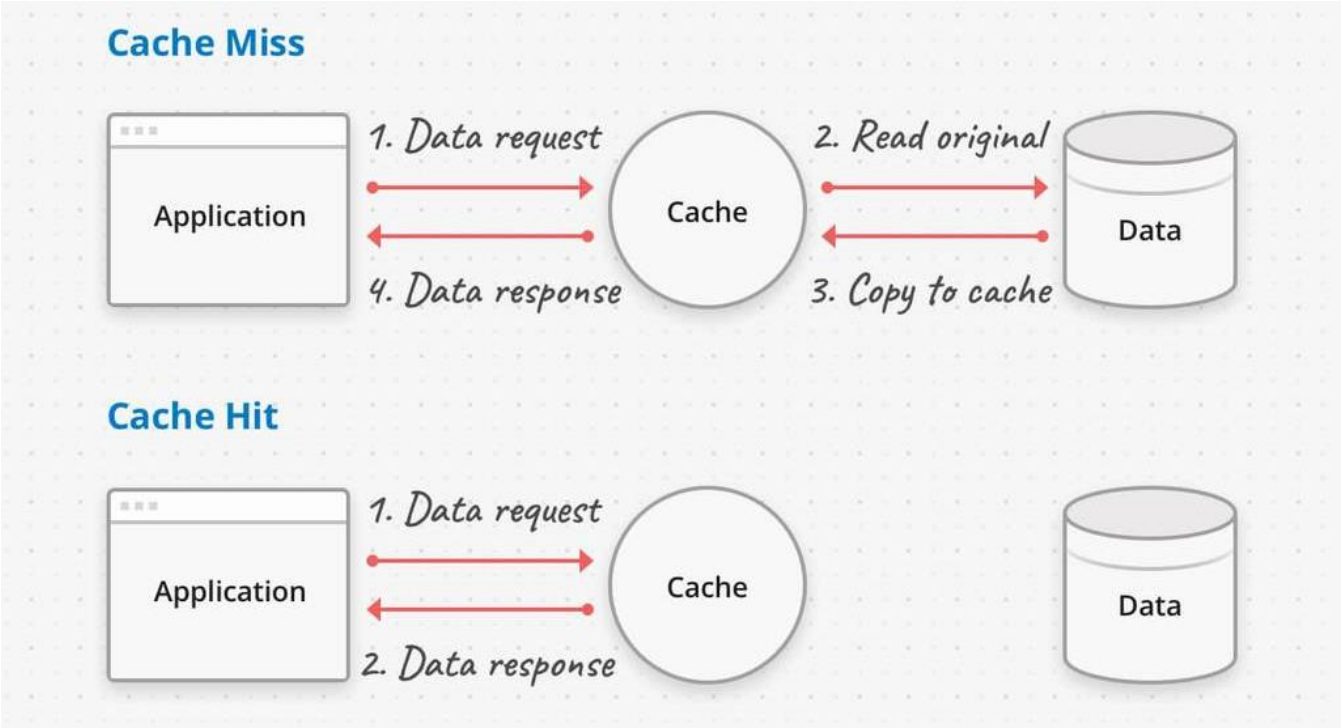


Load balancing techniques

- Distribution to servers based on load balancing techniques
  - **Round Robin**: Client requests distributed to servers in rotation (simplest)
  - **Weighted Round Robin**: Adds weights depending on server characteristics
  - **Least Connection**: Requests distributed to the server with the least active connections
  - **Weighted Response Time**: The server that responds the fastest, receives the next request
  - **Resource-based**: Monitoring agent helps decide how to balance load

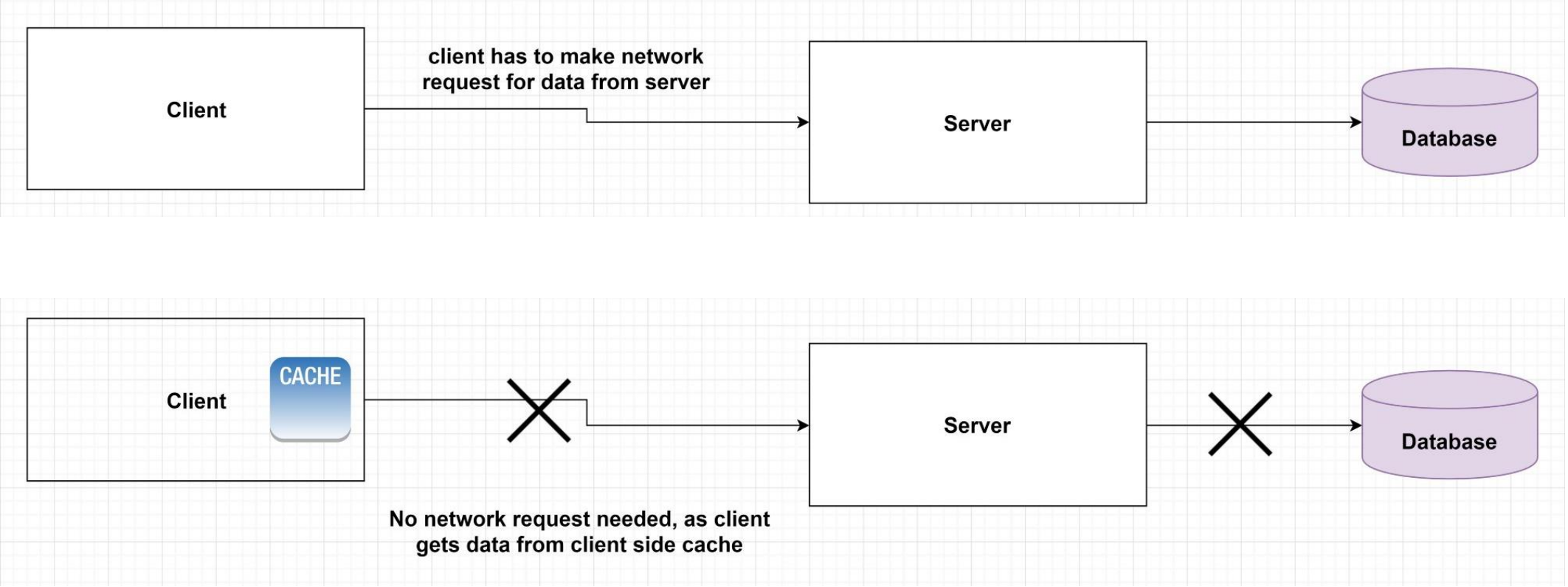
Caching

- Stores frequently accessed data in a way that it’s faster to retrieve
- Applications access the cache if the data is available there, then go to the DB



Client-Side Caching

- Cache is stored on the client so no need to send request to the server



Client-side Storage Options

- Browser can store data on the user's hard drive
- Main options include:
  1. Web Storage (Local Storage and Session Storage)
  2. Cookies (Session Cookies & Persistent Cookies)
  3. IndexedDB
  4. Cache API

Web Storage

- Synchronous and will block the main thread - **use with caution**
- Limited to about 5MB and can only contain strings
- Session storage
  - available for the duration of the page session (as long as the browser is open)
- Local storage
  - does the same thing, but persists even when the browser is closed and reopened

Cookies

- Sent with every HTTP request
- Should not be used for storing more than a small amount of data
- Synchronous and limited to only strings
- Used to tell if two requests came from the same browser
- Remembers stateful information for the stateless HTTP protocol
- Session Cookies
  - Removed when browser is closed
- Persistent cookies
  - Expire at a specific date (Expires) or length of time (Max-Age).

IndexDB

- Transactional database system but is a JS object-oriented database
- Store much bigger volumes of data than localStorage
- Store any kind of value based on { key: value } pairs
- Asynchronous, so it won’t block main thread
- Useful for web applications that don’t require a persistent internet connection

Cache API

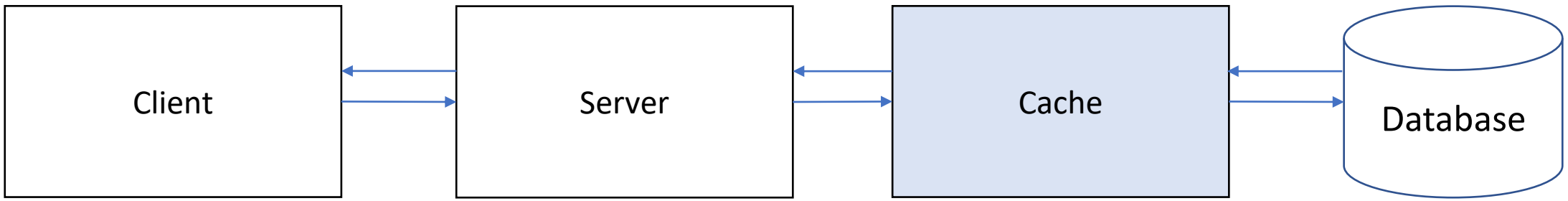
- Stores HTTP request and response objects
- Asynchronous, so it won’t block main thread
- Used in Progressive Web Apps to cache network responses so an app can serve cached resources when it’s offline
- Not practical for storing other types of data

Wake-up!

- https://youtu.be/G3UtAW8hc-o

Server-Side Caching

- Server accesses cache because it is faster than the database
- In memory caches store data in RAM, which is faster to access than disk and are very commonly used for server-side caching
  - Examples: Redis or Memcached



Caching Considerations

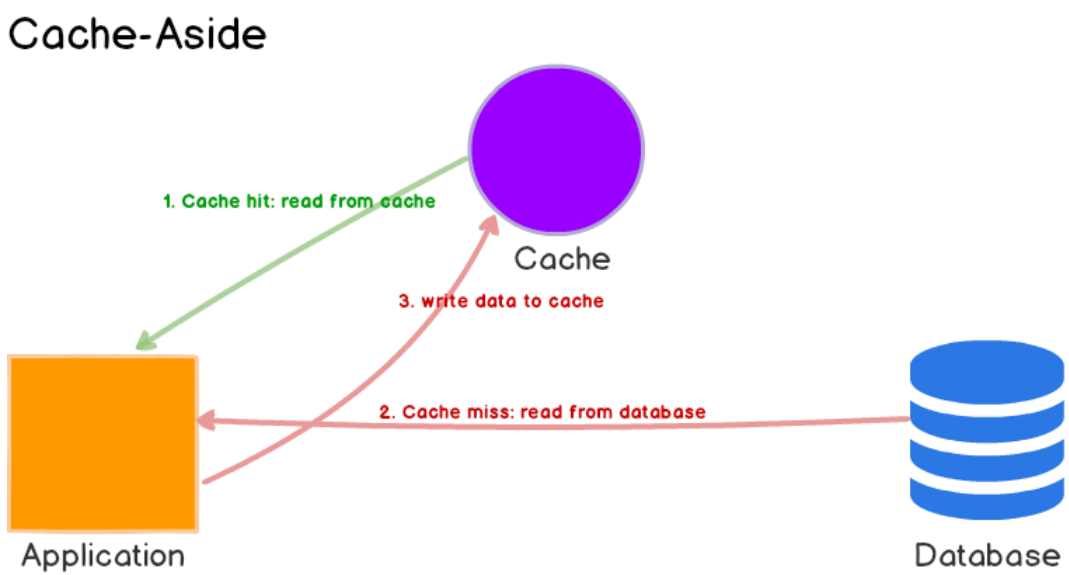
- Requires cache invalidation techniques
  - Ensures data is up to date
  - Don’t want stale data
- Requires cache eviction policy
  - Frees up space and removes stale data

Cache Invalidation Techniques

- Ensures cache data is up to date
- Techniques
  - Cache Aside - only reads DB if data is not in cache, then writes to cache
  - Read Through Cache - a cache miss loads data from DB and updates cache
  - Write Through Caching - writes to cache, then to DB
  - Write Back Caching - writes to cache, then writes to DB on time interval
- More on this here:
  - https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one

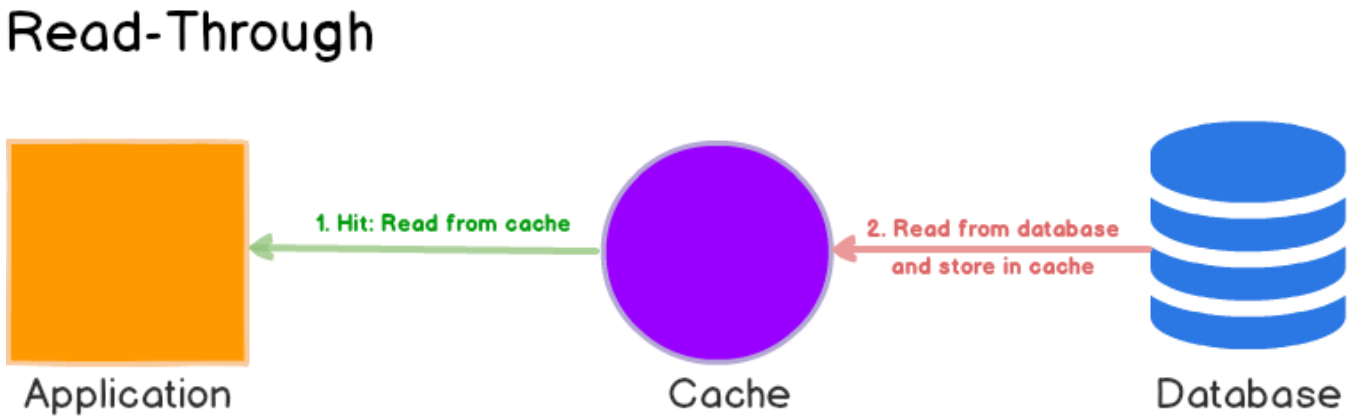
Cache Aside

- The cache is on the side; app talks to both the cache and database
- No connection between the cache and the database
- Most common cache approach
- Requires extra work from the app
- Works best for **read-heavy** workloads



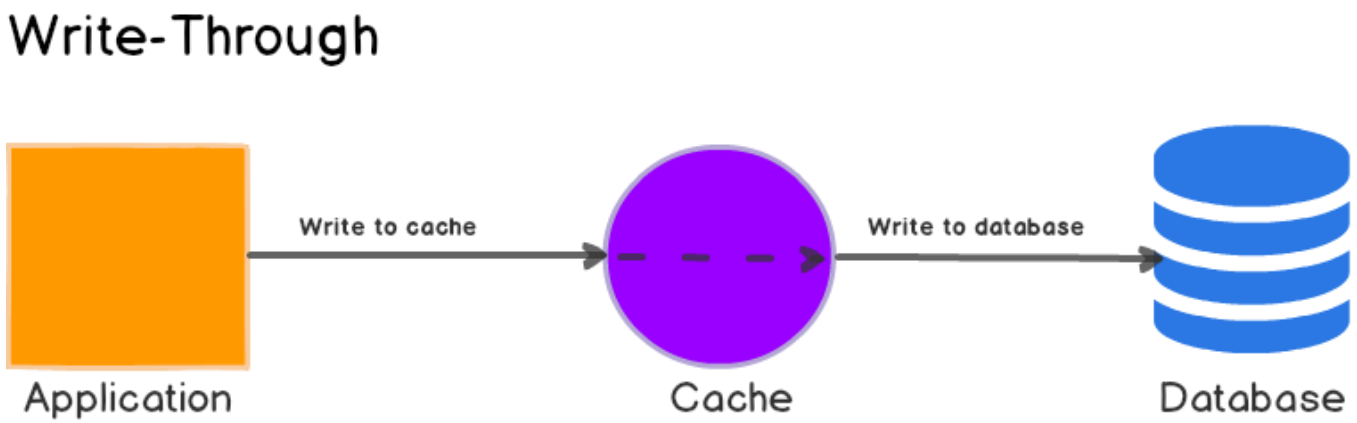
Read Through Cache

- Cache sits in-line with the database
- When there is a cache miss, the cache loads missing data from database, populates the cache and returns it to the application
- Works best for **read-heavy** workloads when the same data is requested many times



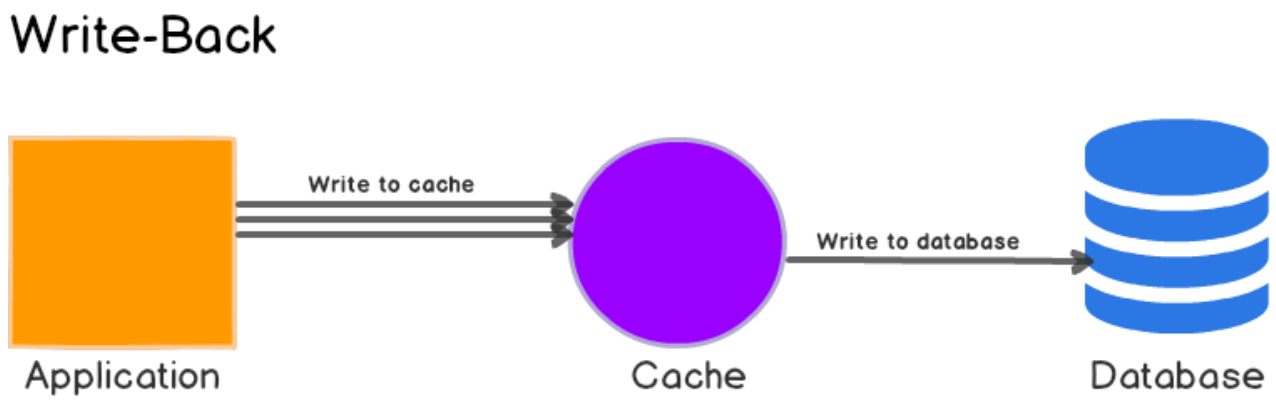
Write Through Caching

- Data is first written to the cache and then to the database
- Writes always go *through* the cache to the main database helping cache maintain consistency with the main database
- Alone it introduces extra write latency (two write operations)
- When paired with read-through caches, you get all the benefits of read-through and get data consistency guarantee (no cache invalidation)



Write Back Caching

- Same as write through, but data written to the cache is asynchronously updated in the main database
- Seems faster because only the cache needs to be updated before returning a response
- Good for write-heavy workloads

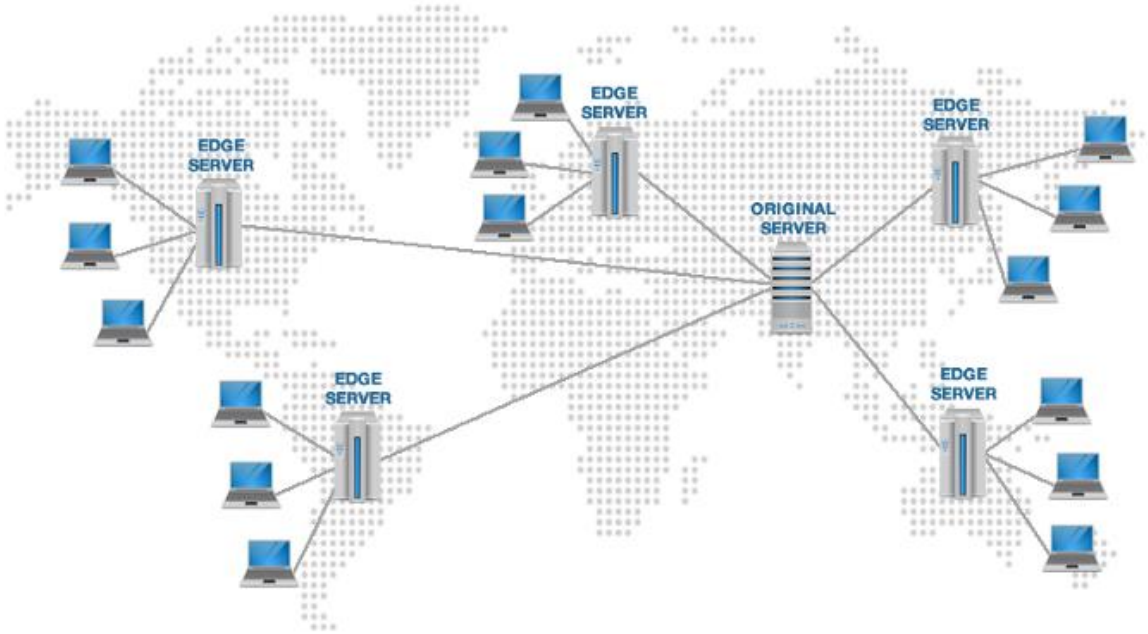


Cache Eviction Policies

- Frees up space and removes stale data
- Policies
  - First In First Out (FIFO) - discards data added first (like a queue)
  - Last In First Out (LIFO) - discards data that is most recently added (like a stack)
  - Least Recently Used (LRU) - discards the least recently (timestamp) used data
  - Least Frequently Used (LFU) - discards the least frequently used data
  - Random Selection - discards data randomly

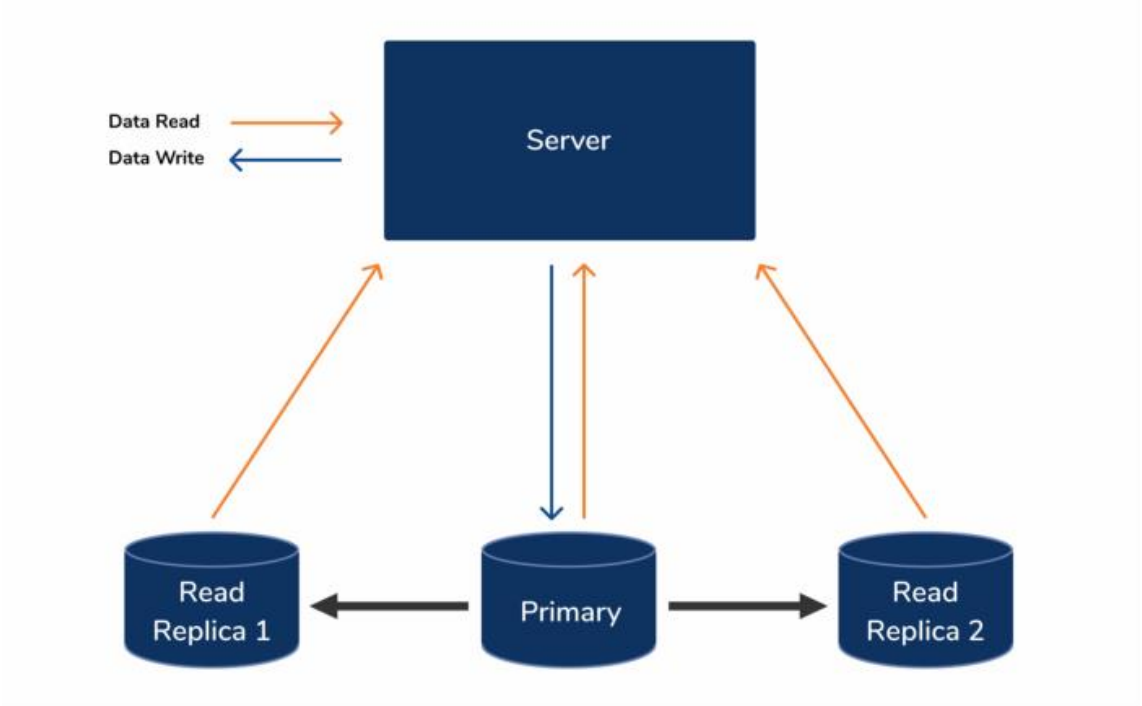
Content Delivery Network (CDN)

- Global network servers storing static files (image, videos, code files)
- Users far from web server can access files quickly with CDN because the CDN is closer to the user



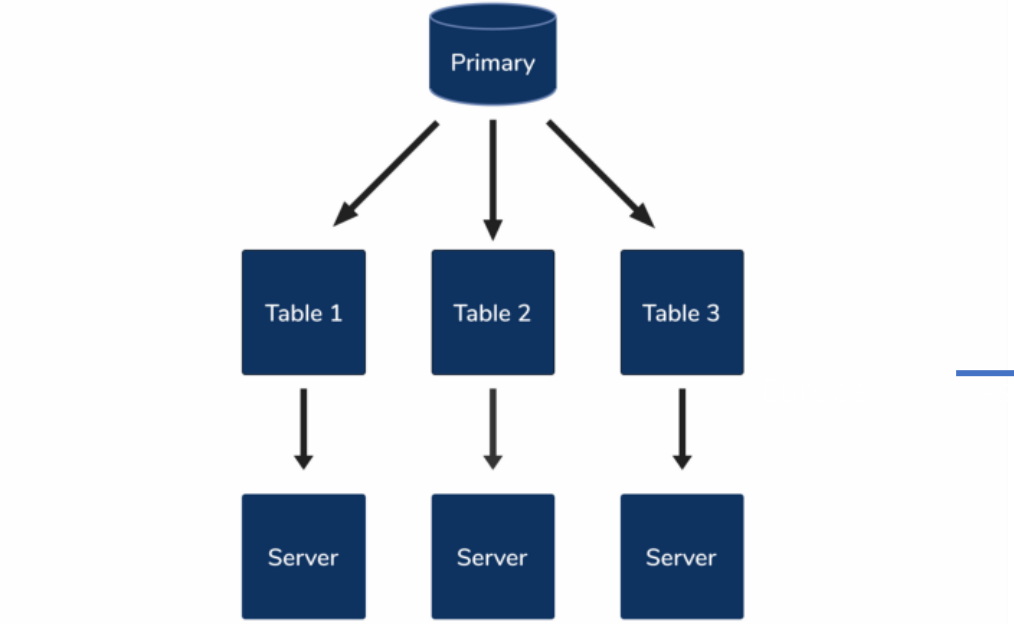
Database Replication

- Replicate your database to ensure data is not lost if in a disaster
- *Database read replication* can help to scale a read heavy system
  - Cloned DBs are used to read from where the primary database is to write



Database Sharding

- Horizontal sharding separates tables with identical columns but distinct rows are put on different machines
  - e.g. Divide users in table by continent (Europe, Asia, N. America)



UserID	Email	Country	Phone
ayeh	ayeh@yahoo.com	USA	555-555-5555
pila	pila@gmail.com	USA	555-555-5555
loper	loper@gmail.com	China	555-555-5555
yuta	yuta@aol.com	China	555-555-5555

Database Sharding

- Vertical sharding separates tables by columns (normalization)

UserID	Email	Country	Phone
ayeh	ayeh@yahoo.com	USA	555-555-5555
pila	pila@gmail.com	USA	555-555-5555
loper	loper@gmail.com	China	555-555-5555
yuta	yuta@aol.com	China	555-555-5555

Monolith

- Server-side based on single application
- Easy to develop, deploy, and manage
- Challenge
  - Different parts of app are tightly coupled
  - Difficult to work on with a large team
  - App scales as a whole (can’t scale specific high traffic pieces)

Microservices

- A single application is composed of many loosely coupled and independently deployable smaller services which typically:
  - Have their own tech stack
  - Communicate via REST APIs, event streaming, and message brokers
  - Are organized by business capability
- Benefits:
  - New features can be added without touching the entire application
  - Teams can use different stacks and different programming languages for different services
  - Components can be scaled independently of one another

Cloud Services for Scaling

- Cloud services offer resources that you can use to scale
- Services
  - Servers
  - Load balancing
  - Databases (SQL and NoSQL)
  - Caching services
- Advantages:
  - No limit to hardware capacity
  - Pay for only what you use
  - Instantaneous availability
  - Easy to resize to your needs