Chapter 5 Design Patterns for Scaling

Presenters:

Pranali Shenvi

Ameya Mangaonkar

What is Scaling?

A system's ability to process a growing workload, measured in transactions per second, amount of data, and number of users

Distributed systems must be built to be scalable because growth is expected

Make sure that a service is fast & stays fast (It's critical!)

Strategies for building a Scalable System

Design

Design the system with scalability from the start

Identify

Identify Bottlenecks:

- A point in the system where congestion occurs
- Bottleneck is where backlog accumulates

Reengineer

Reengineer Components:

- Rewriting parts of system to improve speed, functionality or resource consumption
- Restructuring an existing code, mostly its internal structure without changing external behavior is called refactoring

Measure Results:

- Scaling solutions must be evaluated using evidence- take measurements, try a solution, and repeat the same measurements to see the effect
- If the effect is negative, the solution isn't deployed well

Be Proactive:

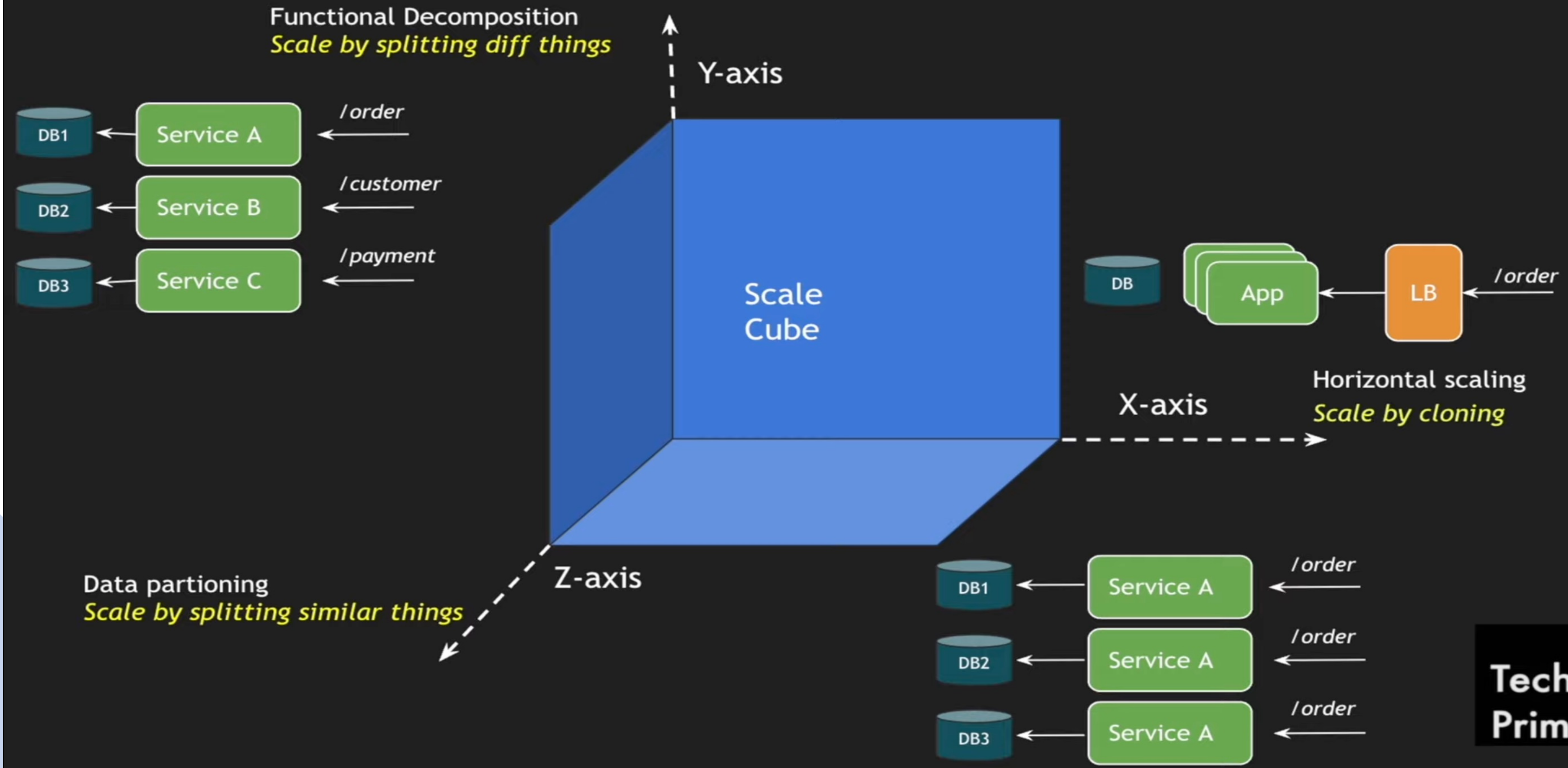
- The best time to fix a bottleneck is **BEFORE** it becomes a problem
- Predict problems enough in advance that there is time to engineer a solution
- Always remember the Goldilocks solution: ***Not too early, not too late, just right***

Scaling Up

- The simplest method to scale is a system is to use bigger, faster equipment
- An existing computer can have its attributes improved without replacing the whole machine, which is called scaling up
- Cons:
  1. Limits to system size
  2. Might not always be economical
  3. Won’t work in all situations

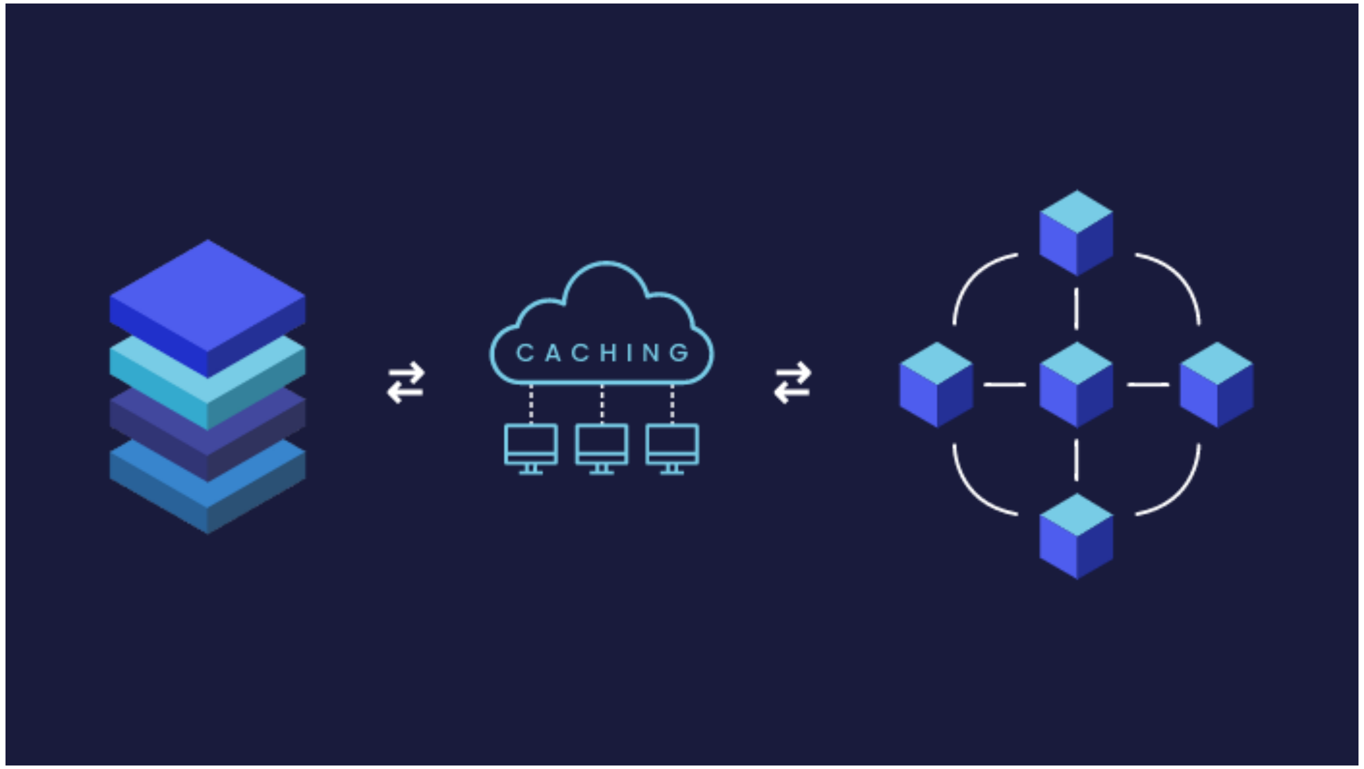
AFK Scaling Cube

- X-axis:
  - Also called as Horizontal Duplication or Scale by Cloning
- Y-axis:
  - Also called as Functional Decomposition or Scaling by Splitting Different things
- Z-axis:
  - Also called as Data Partitioning or Scale by splitting similar things


Functional Decomposition
Scale by splitting diff things
Y-axis
DB1
Service A
/order
DB2
Service B
/customer
DB3
Service C
/payment
Scale Cube
DB
App
LB
/order
Horizontal scaling
Scale by cloning
X-axis
Z-axis
Data partioning
Scale by splitting similar things
DB1
Service A
/order
DB2
Service A
/order
DB3
Service A
/order

Caching

- Software/ hardware components which allows you to store data so that future requests for that data can be served faster
- Cache Hit: Successful Lookup from data table found in cache is called cache hit
- Cache Miss: Unsuccessful Lookup from data table not found in cache is called cache miss



Cache Effectiveness

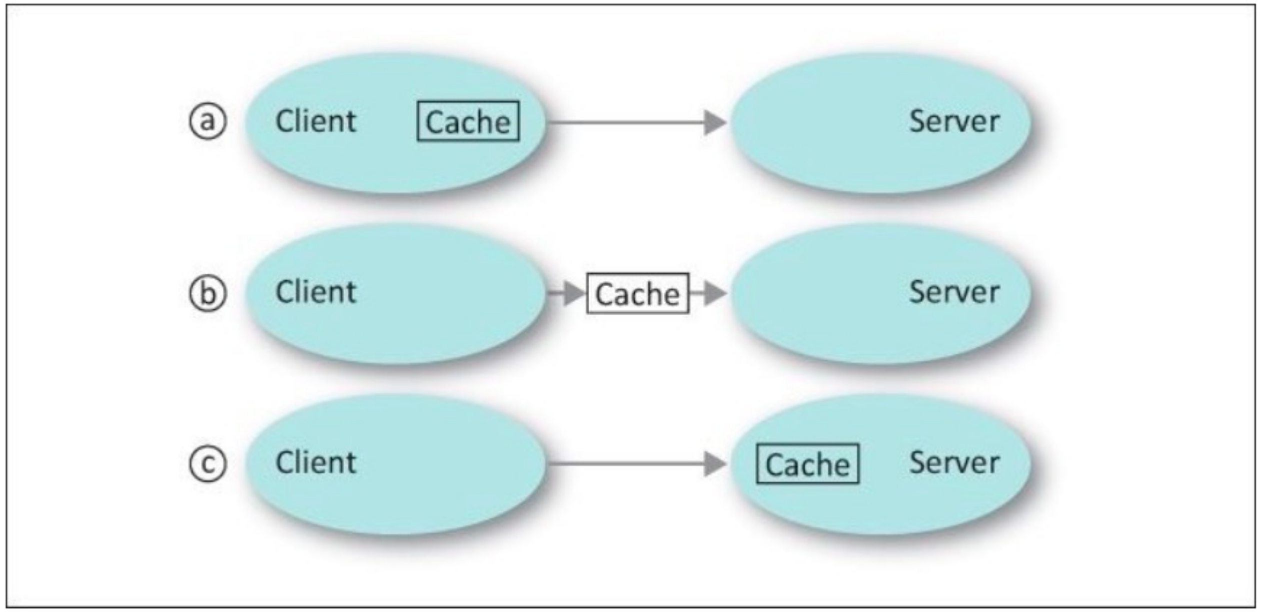
Measured using Cache Hit Ratio:

CHR= Number of Cache Hits/ Number of Lookups

For eg: Lookups performed are 500; 100 were serviced from cache; then CHR will be 1/5 or 20%

Cache Placement

1. **Local caching:** Software performs its own caching
2. **External caching:** Cache is placed between the server and external resources
3. **Caching at the server side:** Server has its own caching



Cache Persistence

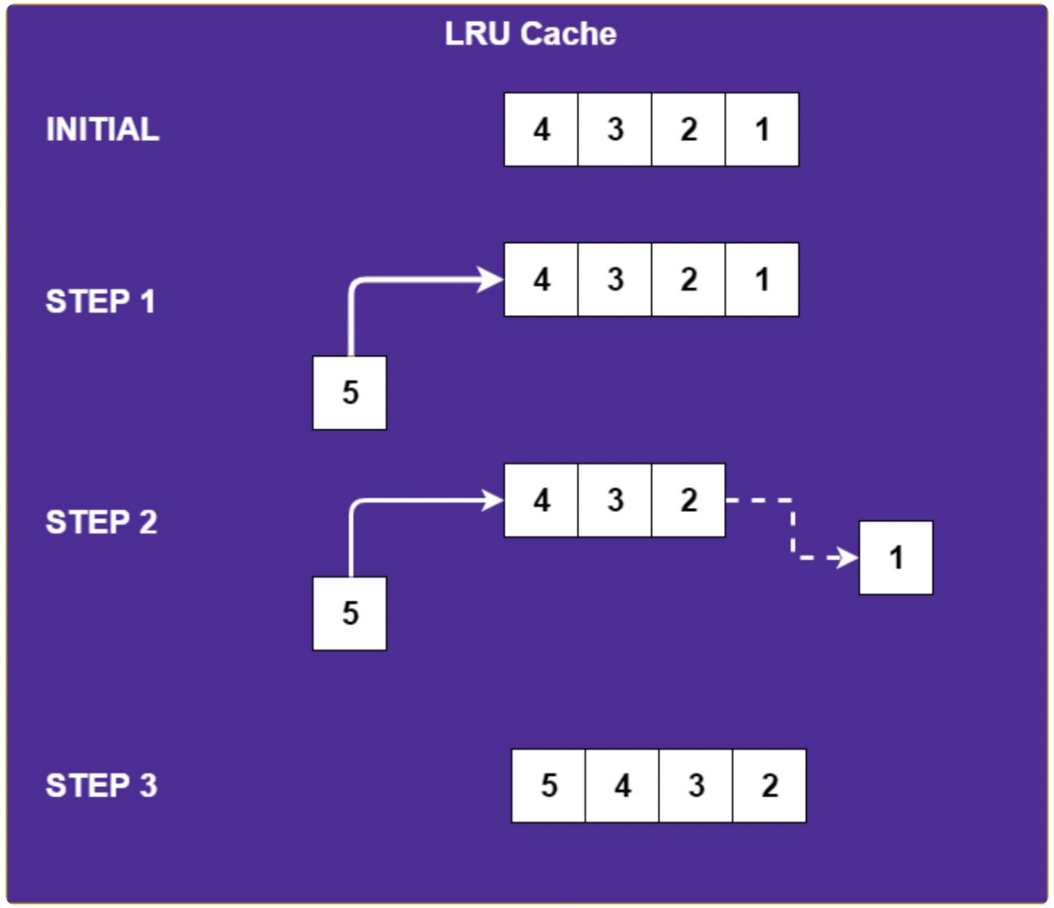
- When a system starts, the cache is usually empty, or **cold**
- The cache hit ratio will be very low and performance will remain slow until enough queries have **warmed** the cache
- Some caches are persistent, meaning they survive restarts
- **For example**, a cache stored on disk is persistent across restarts. RAM is not persistent and is lost between restarts. If a system has a cache that is slow to warm up and is stored in RAM, it may be beneficial to save the cache to disk before a shutdown and read it back in on startup

Cache Replacement Algorithms

- Our cache storage is finite
- We have no choice but to evict some objects and keep others
- Cache Replacement Algorithms decide which objects can stay and which objects should be evicted
- Better algorithms keep track of more usage information to improve the cache hit ratio
- Cache hit ratio is a measurement of how many content requests a cache can fill successfully, compared to how many requests it receives
- Different algorithms work best for different data access patterns

Least Recently Used Algorithm (LRU)

- LRU keeps the least recently used objects at the top and evicts objects that haven't been used in a while if the list reaches the maximum capacity
- It is an ordered list where objects are moved to the top every time they're accessed; pushing other objects down

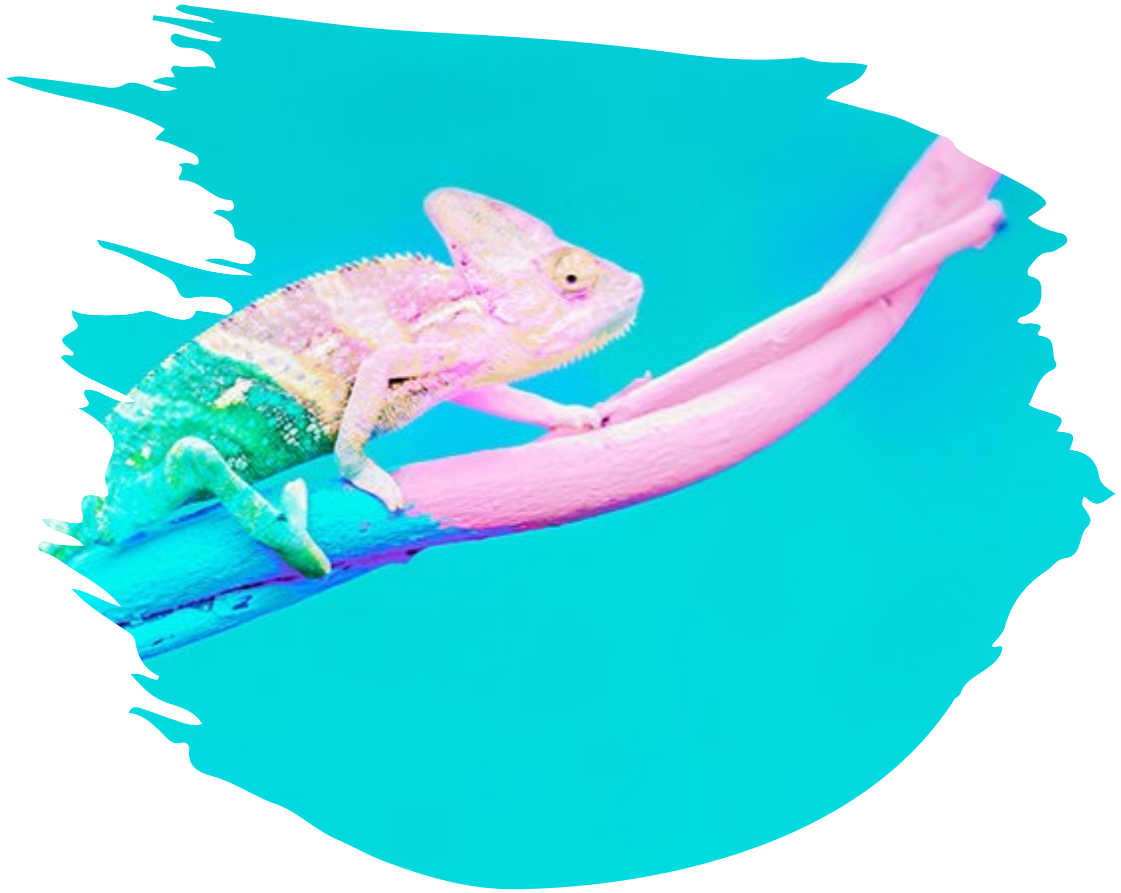


Least Frequently Used Algorithm (LFU)

- It keeps track of how many times an object was accessed instead of how recently it was accessed
- Each object has a counter that counts how many times it was accessed
- When the list reaches the maximum capacity, objects with the lowest counters are evicted
- **Problem** - an object was repeatedly accessed for a short period only. Its counter increases by a magnitude compared to others so it's very hard to evict this object even if it's not accessed for a long time

Adaptive Replacement Cache (ARC)

- Sudden influx of little data does not make the algorithm work well
- ARC puts the newly cached data in a probationary state
- It then moves to the main cache if accessed again
- A single pass flushes the probationary cache and not the main cache



Cache Entry Invalidation

When data in the primary storage location changes, any related cache entries become obsolete

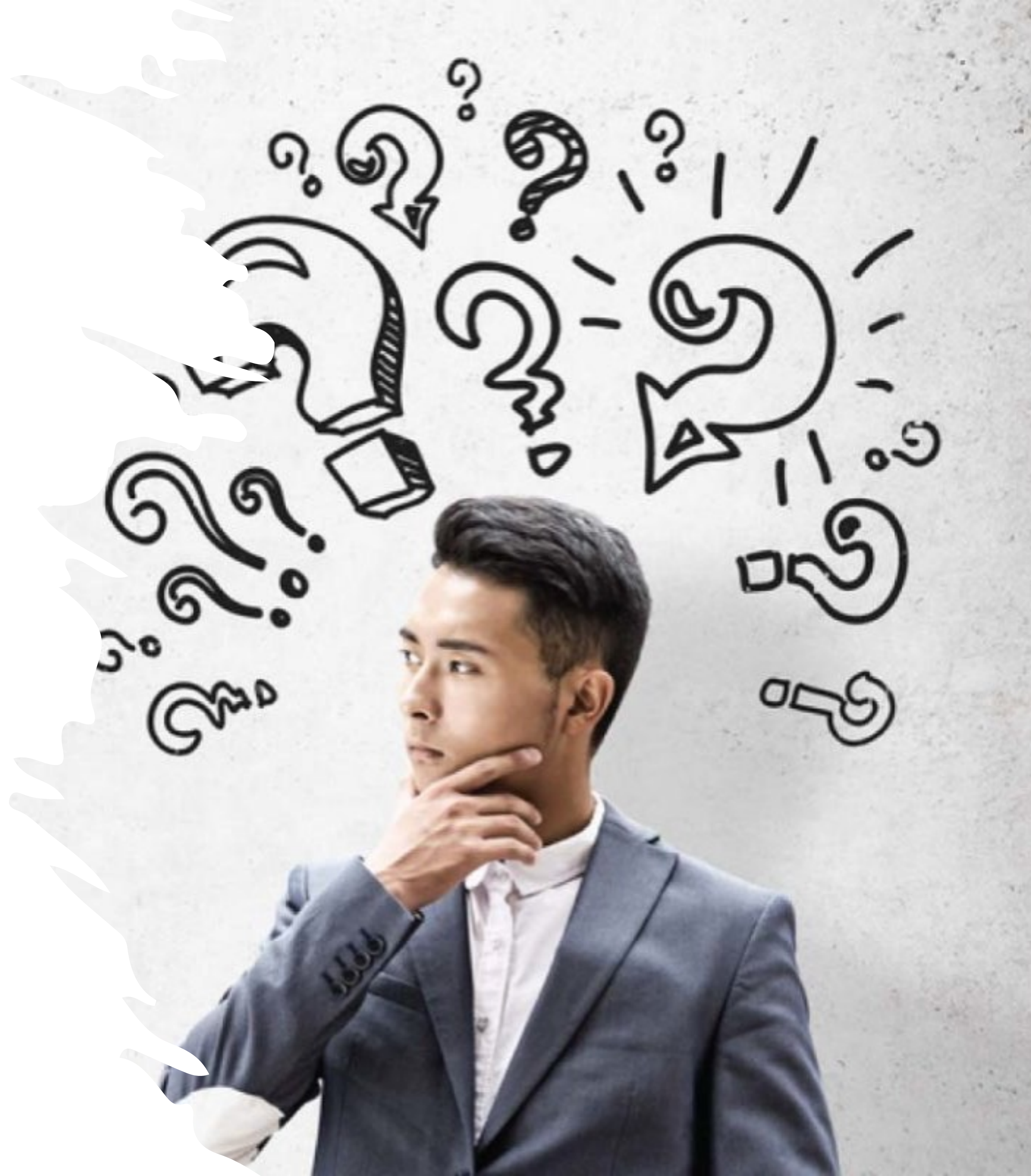
- Ignore it – the cache is very small and obsolete entries will be eventually be replaced via the cache replacement algorithm
- Invalidate the entire cache anytime the database changes. It leaves the cache cold, but performance suffers. It can be used when the cache warms quickly
- The main storage may communicate to the cache the need to invalidate an entry whenever the data in the cache has changed
- The cache can record a timestamp on each entry when it is created and expire entries after a certain amount of time
- The server can help by including how long an entry may be cached when it answers a query
- The cache can poll the server to see if a local cache should be invalidated

Cache Size

- Picking the right cache size is difficult but important
- Caches are usually a fixed size

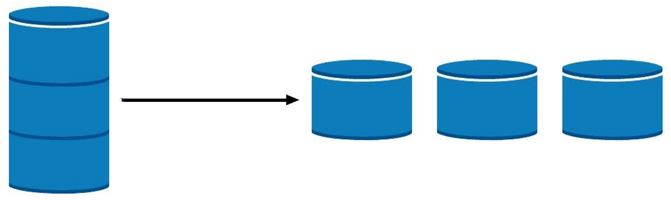
- Methods:

1. Take measurements on a running system
2. Run the system with a variety of cache sizes, measuring performance achieved by each one
3. Configure a system with a larger cache size than the application could possibly need
4. Estimate the cache hit ratio
5. We can improve our estimate by using a cache simulator



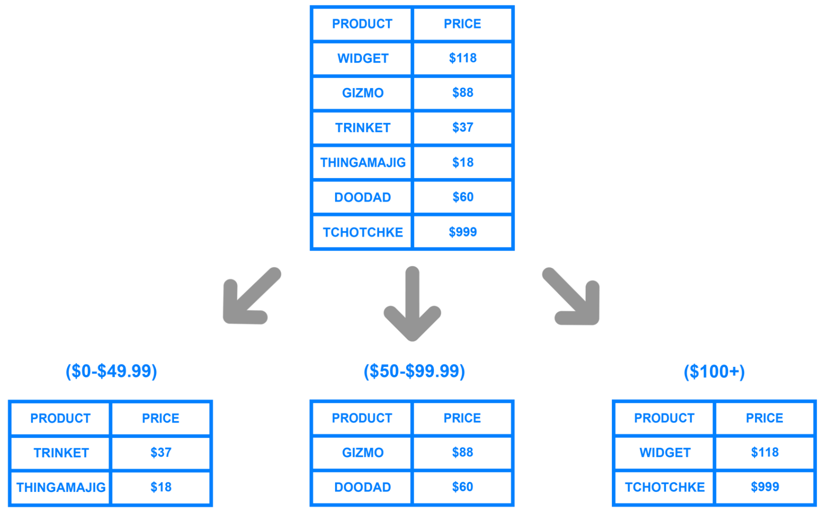
Data Sharding

- Sharding is a way to segment a database (*z*-axis) that is flexible, scalable, and resilient
- It divides the database based on the hash value of the database keys
- A hash function is an algorithm that maps data of varying lengths to a fixed-length value
- To divide a database into two shards, generate the hash of the key and store keys with even hashes in one database and keys with odd hashes in the other database
- Because the hash values are randomly distributed between the shards, each shard will store approximately the same number of keys automatically
- This pattern is called a **distributed hash table (DHT)** since it distributes the data over many machines and uses hashes to determine where the data is stored



Data Sharding (continued)

- Shards can be replicated on multiple machines to improve performance
- As more data are stored, the shards may outgrow the machine
- One must increase the number of shards based on the largest shard, which could be considerably bigger than the smallest
- Create more, smaller shards and store multiple shards on each machine
- New hardware models should be benchmarked to determine the usable capacity before being put into service
- Shards are often used to distribute a read-only corpus of information
- The process of unadvertising until no new requests are received is called **draining**
- One must be aware that while a shard is being drained, there is one fewer replica



PRODUCT	PRICE
WIDGET	$118
GIZMO	$88
TRINKET	$37
THINGAMAJIG	$18
DOODAD	$60
TCHOTCHKE	$999

($0-$49.99)

PRODUCT	PRICE
TRINKET	$37
THINGAMAJIG	$18

($50-$99.99)

PRODUCT	PRICE
GIZMO	$88
DOODAD	$60

($100+)

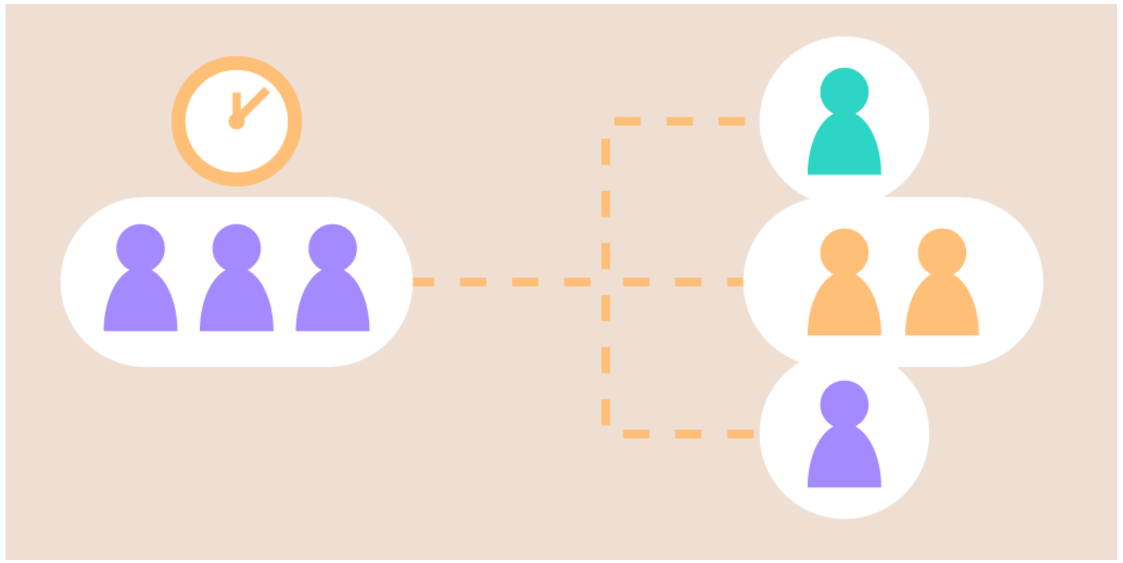
PRODUCT	PRICE
WIDGET	$118
TCHOTCHKE	$999

Threading

- Threading is a technique that can be used to improve system throughput by processing many requests at the same time
- In a single-thread process, we receive a query, process it, send the result, and get the next query This is simple and direct
- **Head of line blocking-** the head of the line is blocked by a big request
- In a flood of requests, some requests will be dropped
- In a multi-core machine, the single thread will be bound to a single CPU. Multithreaded code can take advantage of all cores, thereby making maximum use of a machine
- In multithreading, a **main thread** receives new requests. For each request, it creates a new thread, called a **worker thread**
- If multiple threads must access the same resources in memory, locks or signaling flags we need to prevent resource collision

Queueing

- A **queue** is a data structure that holds requests until the software is ready to process them
- Most queues release elements in the FIFO order
- There is a master thread and worker threads. It follows a **feeding from a queue workflow**
- Queueing prevents overloading the machine, retains the same threads to service multiple requests
- High-priority requests can go to the head of the queue. In **fair queueing**, the algorithm prevents a low-priority item from being unattended by a flood of high-priority items

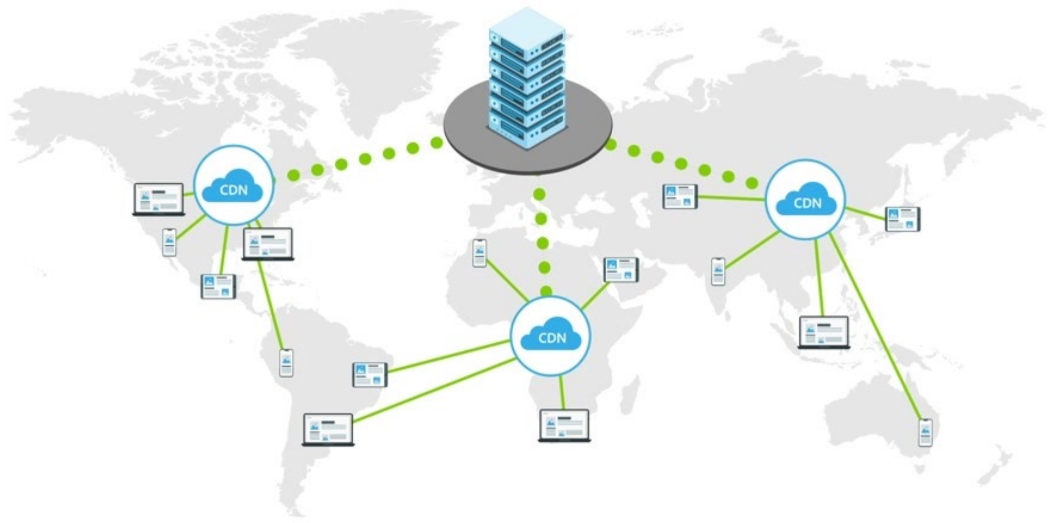


Variations

- Variations are used to optimize performance; it can shrink or grow the number of threads
- Threads can kill and re-create themselves periodically so that they remain fresh
- It is common practice to use processes instead of threads. The fixed population of worker processes means that you pay that overhead once and then get the benefit of using processes. They have their own address space, memory, and open file tables. Processes are self-isolating
- An example of queueing implemented with processes is the Prefork processing module for the Apache web server

Content Delivery Networks

- A **content delivery network** (CDN) is a web-acceleration service that delivers content (web pages, images, video) more efficiently on behalf of your service
- CDNs notice usage trends and determine where to cache certain content
- CDNs have extremely large, fast connections to the internet and have more bandwidth to the internet than most web sites
- CDNs often place their cache servers in the datacenters of ISPs, in arrangements called **colocation**
- While a new web site is in the testing phase, you may not want to use a CDN
- The next generation of CDN products added video hosting. In the past, file sizes were limited, but video hosting requires the CDN to be able to handle larger content plus deal with protocols related to skipping around within a video
- The connection between your web servers and the CDN can then use an easier-to-manage transport mechanism, or no encryption at all



Questions?