System Design Basics

Agenda

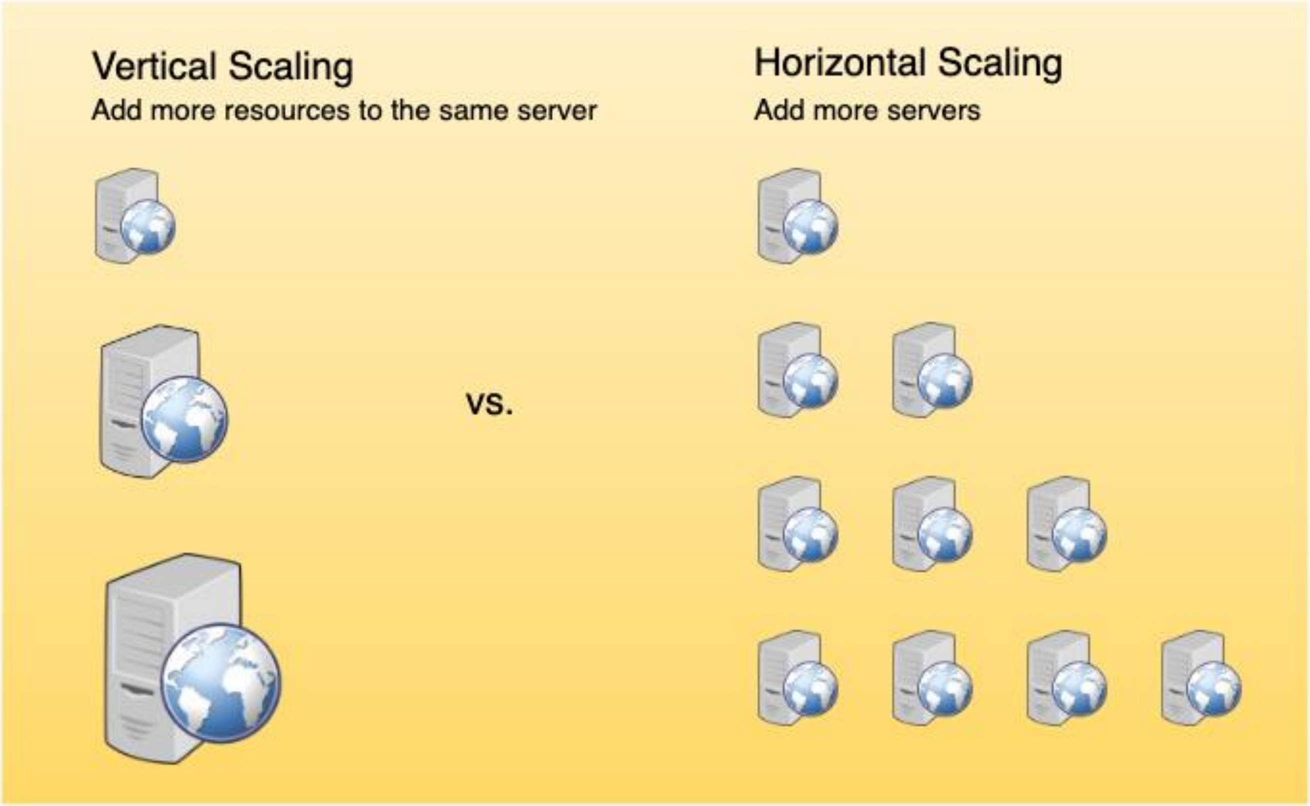
- (Some) Key Characteristics of Distributed Systems
- Load Balancing
- Caching
- Asynchronism
- Scaling the DB
- Consistent Hashing

(Some) Key Characteristics of Distributed Systems

What are DS and why do we need them?

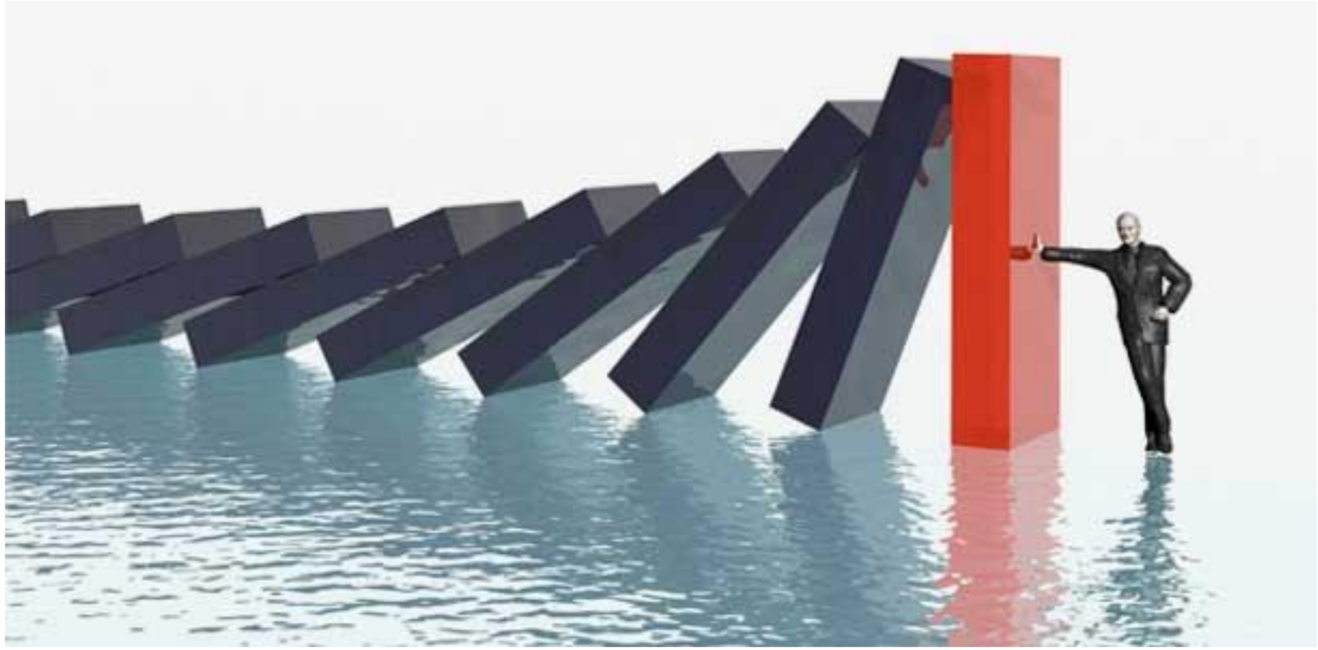
Scalability

- What? - the capability of a system, process, or a network to grow and manage increased demand without performance loss
- Why? - a system may have to scale because of many reasons like increased data volume or increased amount of work, e.g., number of transaction
- How? - **Horizontally** and **Vertically**

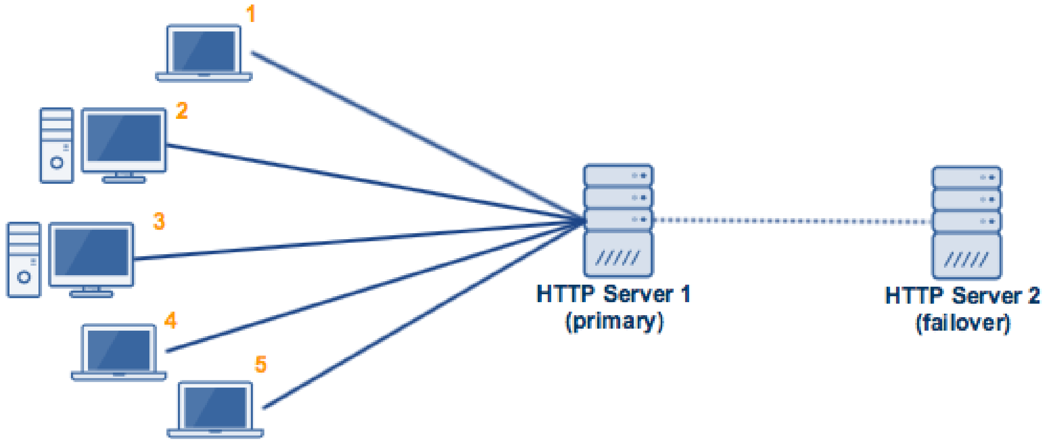


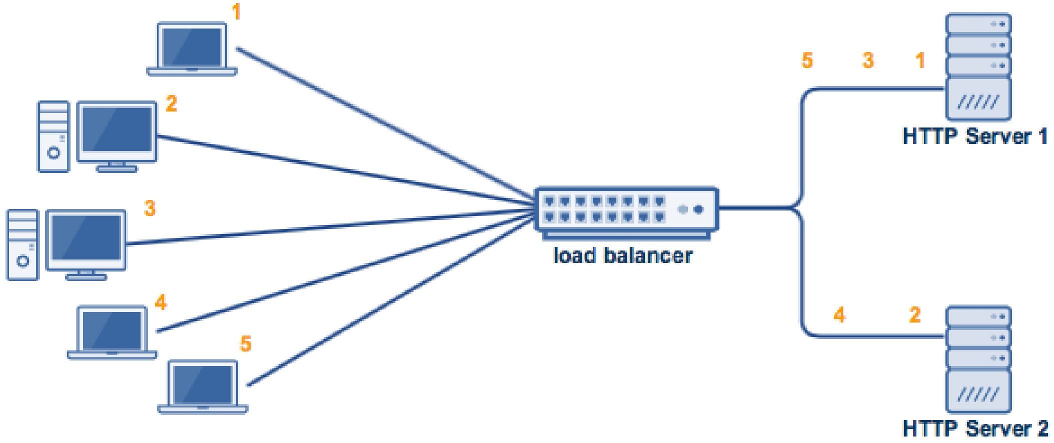
Reliability

- What? - the probability a system will fail in a given period
- Why? - a system needs to be reliable in order to keep delivering its services even when one or several of its software or hardware components fail
- How? - **Redundancy** of both the **software components** and **data**



Ex: Server Redundancy

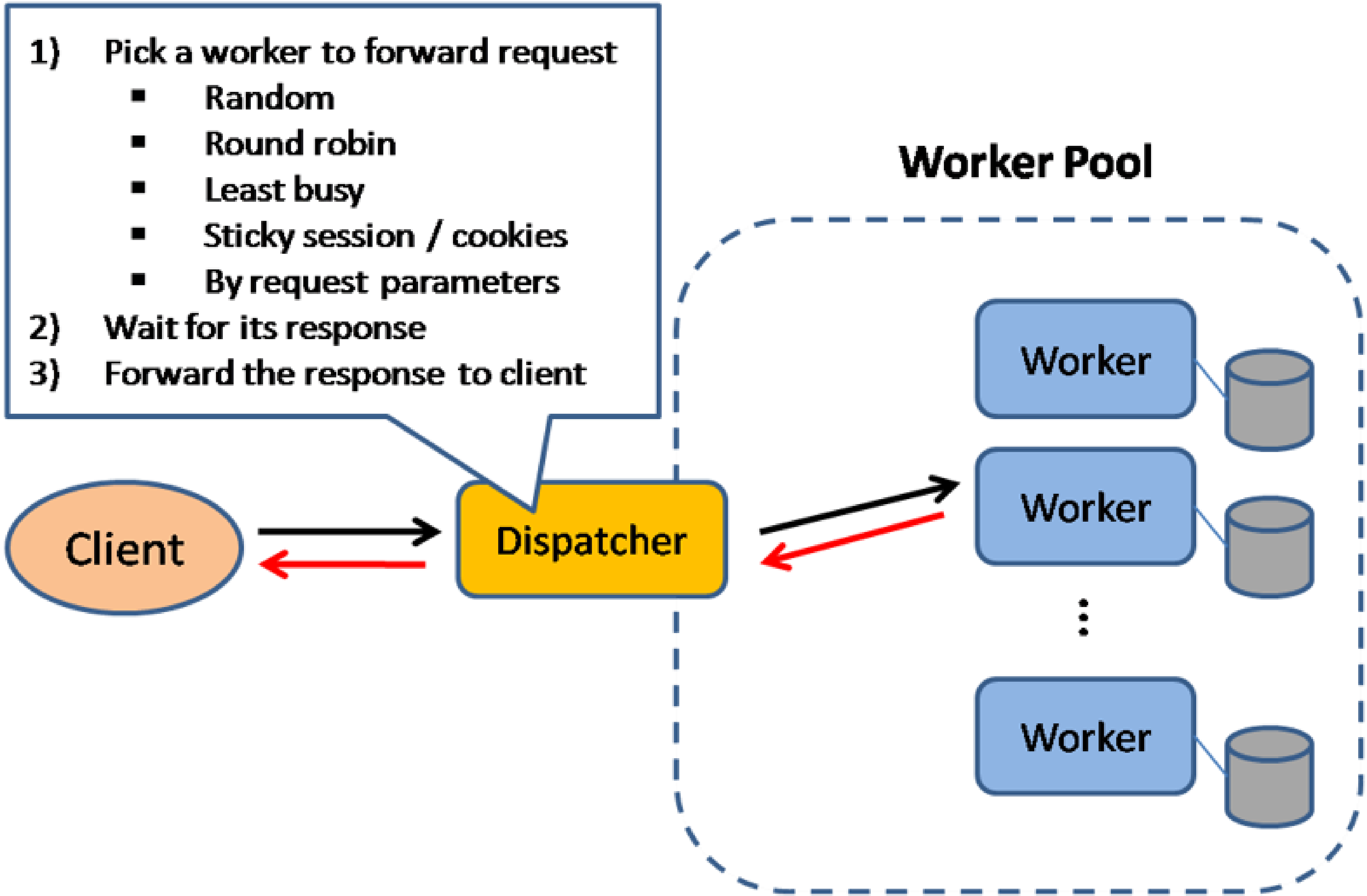


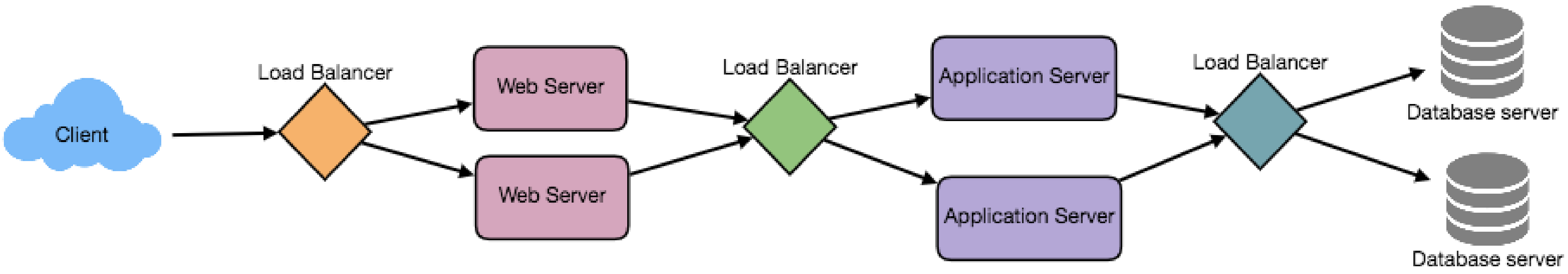


Availability

- What? - the time a system remains operational to perform its required function in a specific period under normal conditions
- **How? - redundancy** and **replication**

Load Balancer


1) Pick a worker to forward request
Random
Round robin
Least busy
Sticky session / cookies
By request parameters
2) Wait for its response
3) Forward the response to client
Worker Pool
Client
Dispatcher
Worker
Worker
Worker

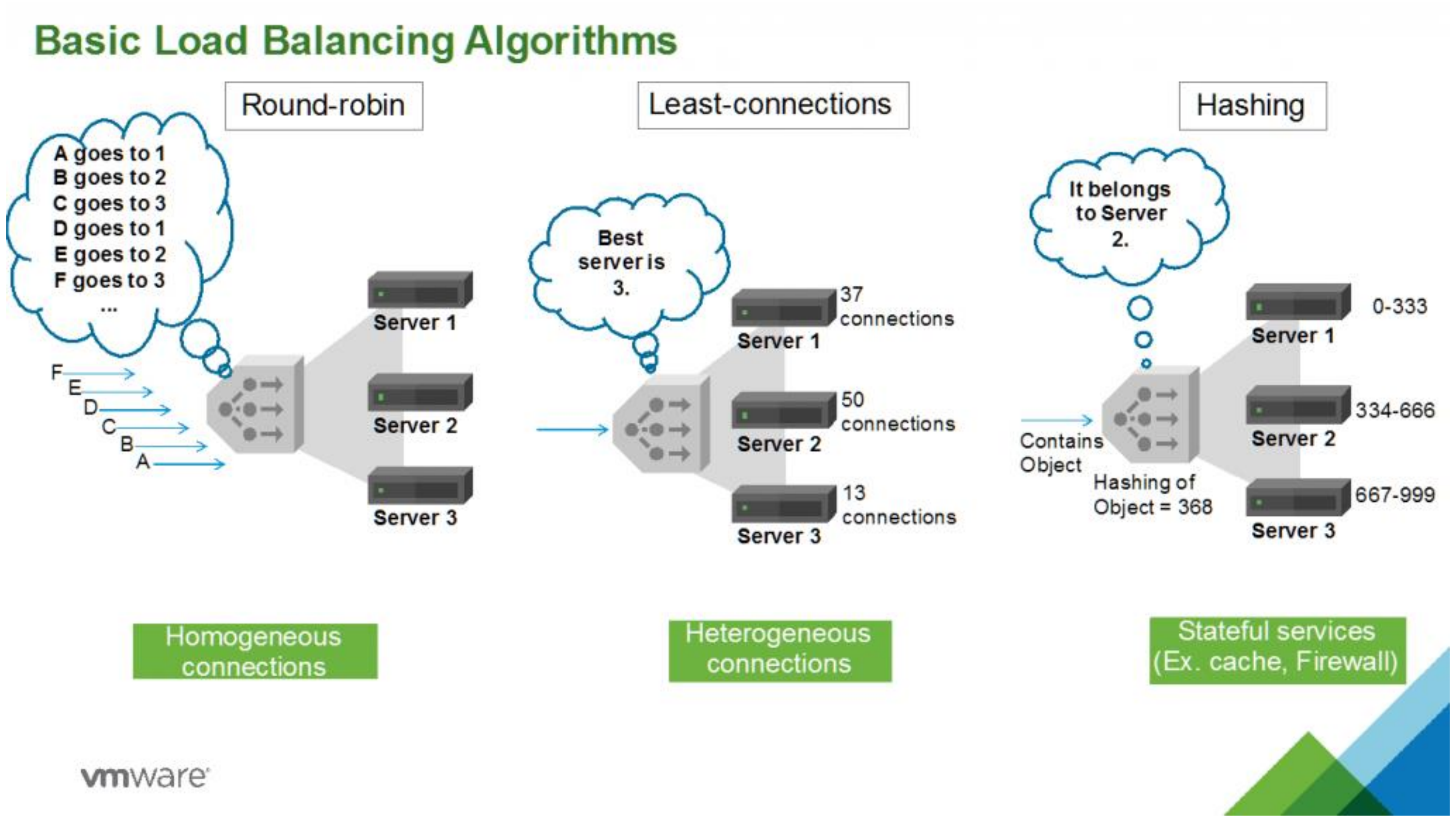


Where?

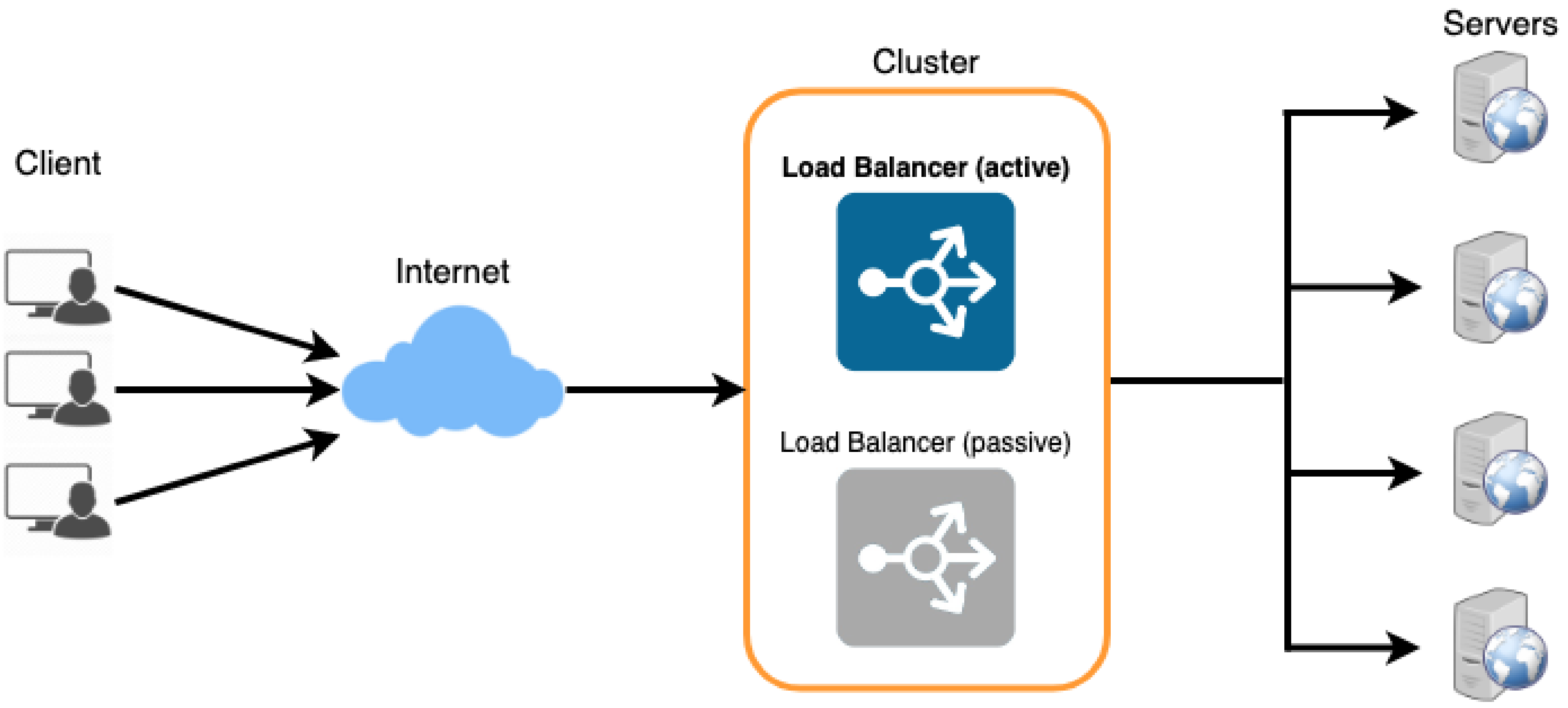
- Between the user and the web server
- Between web servers and an internal platform layer, like application servers or cache servers
- Between internal platform layer and database.

Load Balancing Algorithms

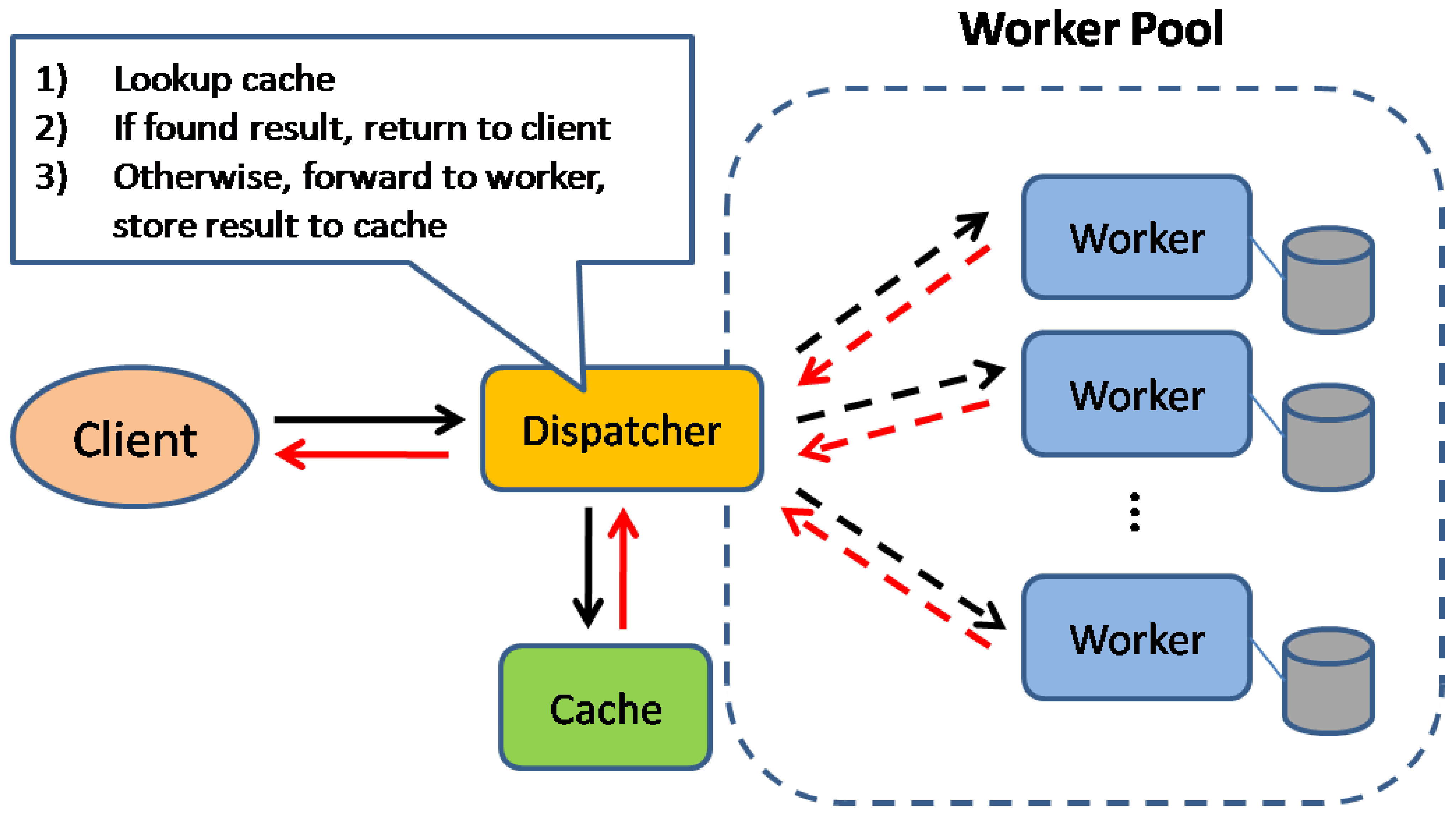
- **Least Connection Method**
- **Least Response Time Method**
- **Least Bandwidth Method**
- **(Weighted) Round Robin Method**
- **IP Hash**
- **Consistent Hashing**

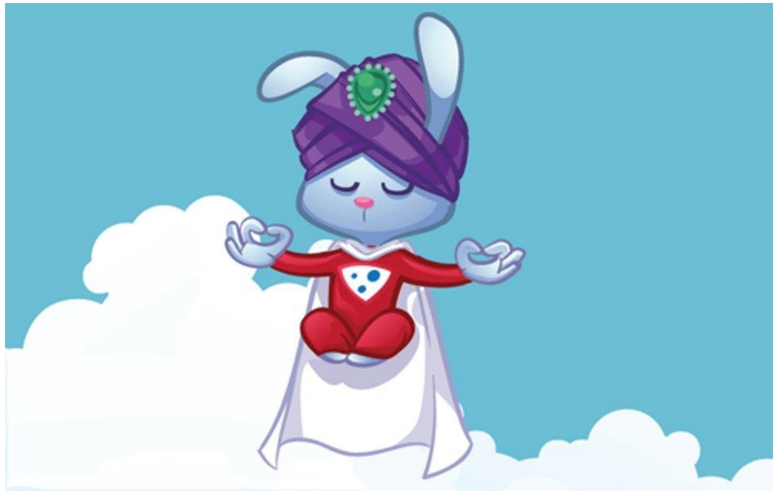


Redundant Load Balancers



Caching

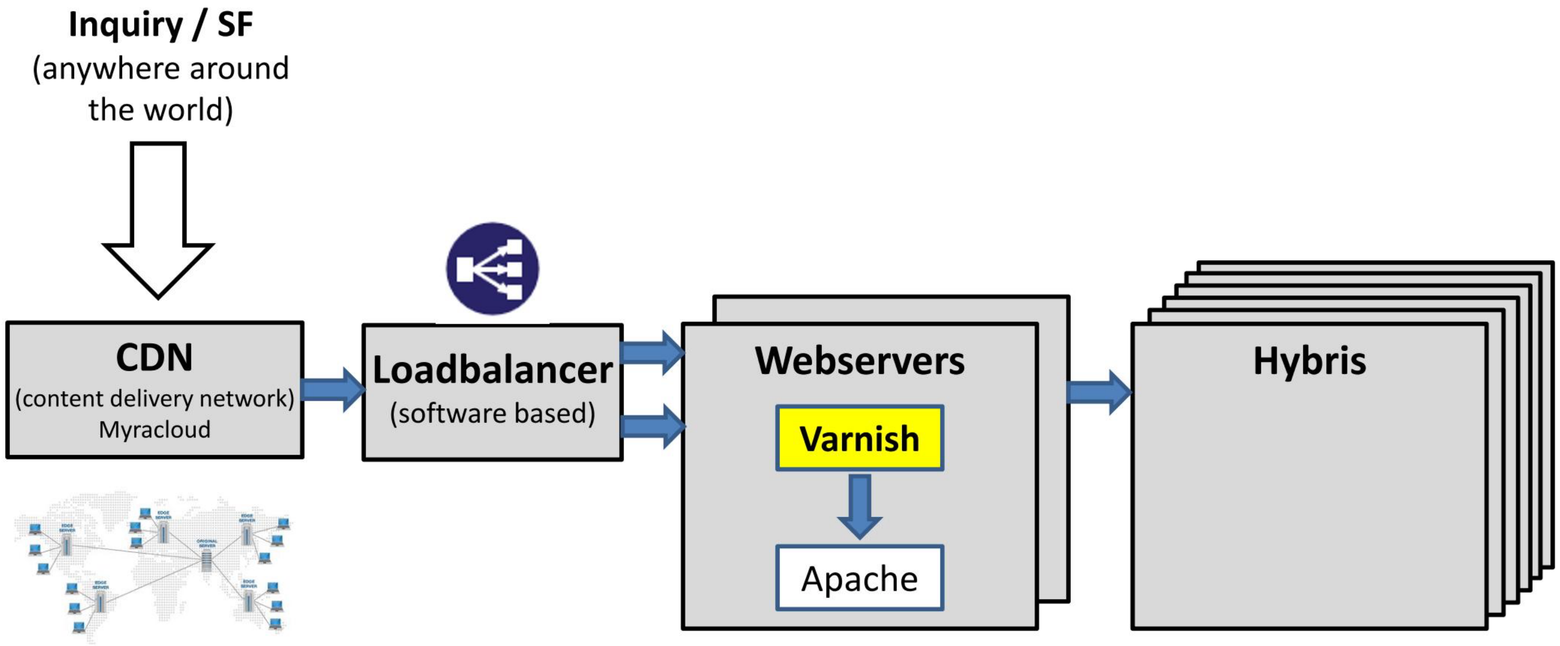

Worker Pool
1) Lookup cache
2) If found result, return to client
3) Otherwise, forward to worker, store result to cache
Client
Dispatcher
Cache
Worker
Worker
Worker



Where?

- CDN caching (Myra)
- Web Server caching – Reverse Proxies (Varnish)
- Application Server caching (Redis)
- Database Caching

Varnish in its ECI environment



Varnish = http accelerator / page cache

KE (mgm), 04.04.17

can't have every user who loads her timeline look up all their followers and then their photos

instead, everyone gets their own list in Redis

Redis is awesome for this; rapid insert, rapid subsets

media ID is pushed onto a list for every person who's following this user

when time to render a feed, we take small # of IDs, go look up info in memcached

Instagram Feed and Redis

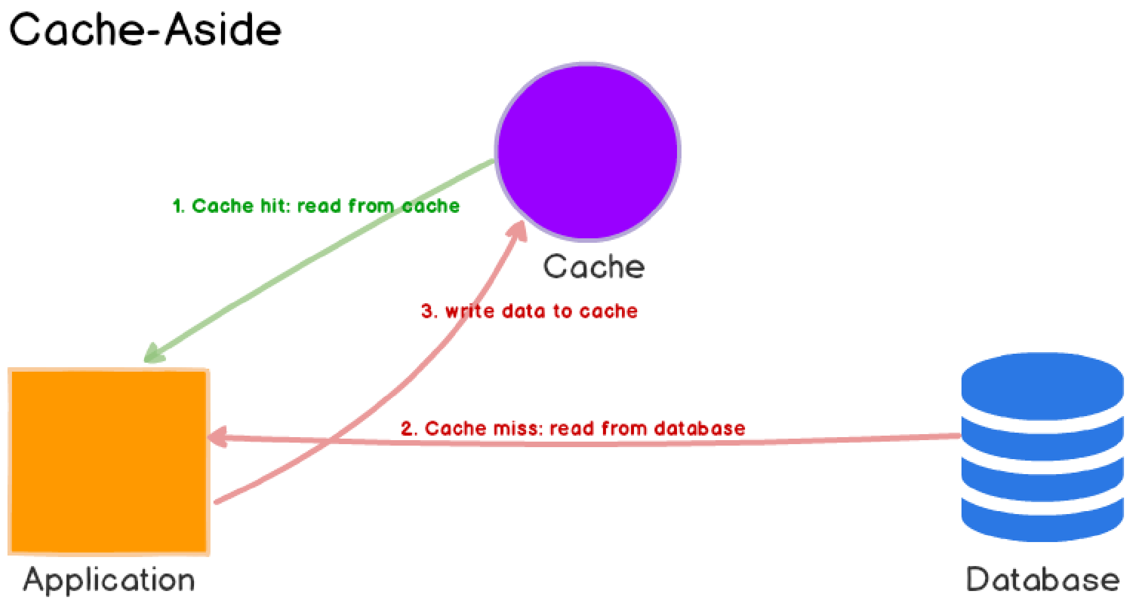
Cache eviction policies

- First In First Out (FIFO)
- Last In First Out (LIFO)
- Least Recently Used (LRU)
- Most Recently Used (MRU)
- Least Frequently Used (LFU)
- Random Replacement (RR)

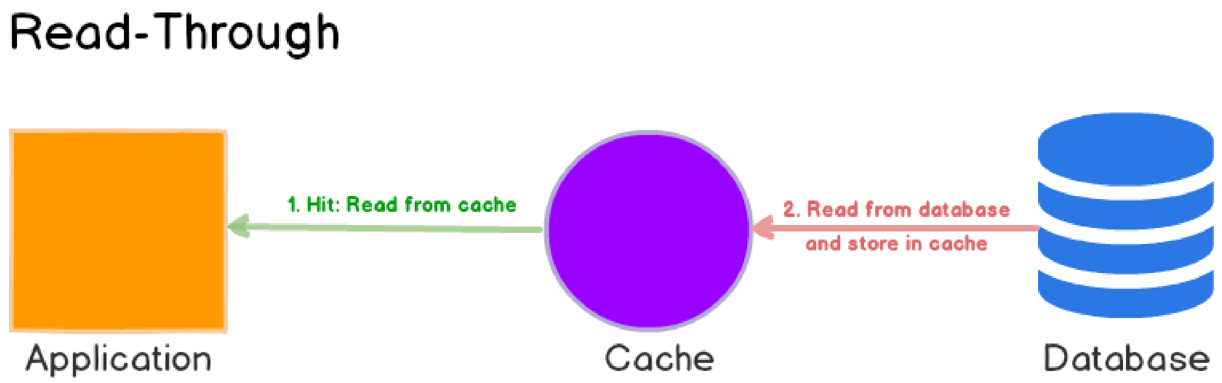
Cache update strategies

- Cache-aside (lazy loading)
- Read-through
- Write-through
- Write-behind (write-back)
- Write-around

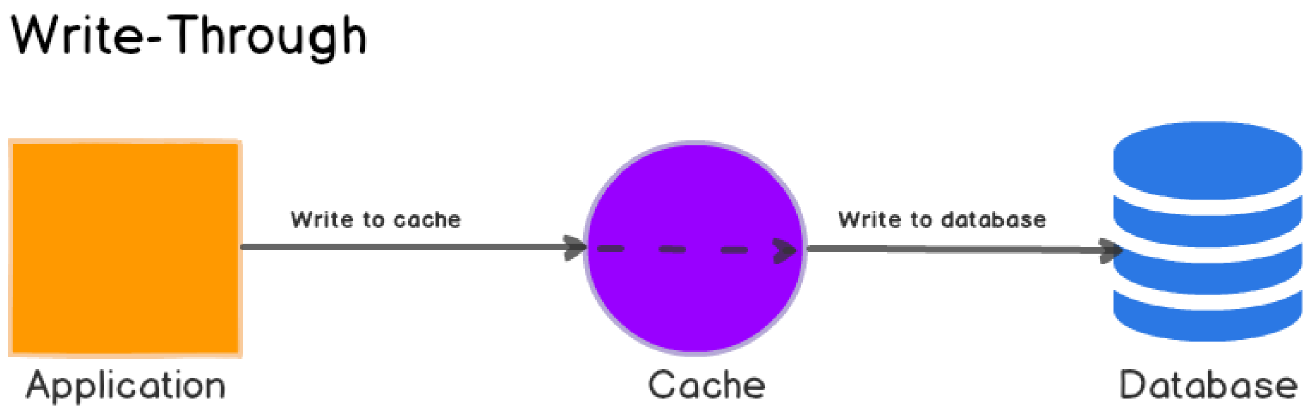
Cache-aside (lazy loading)



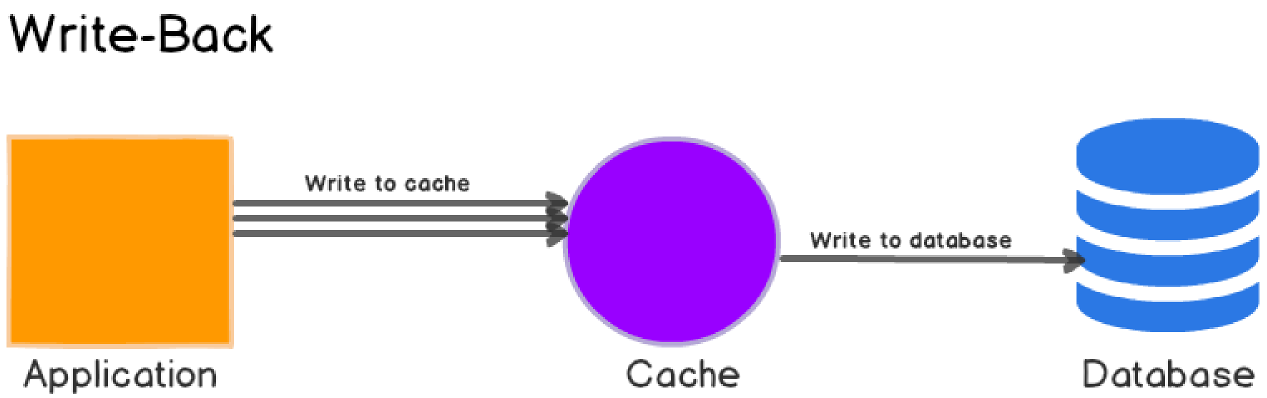
Read-through



Write-through

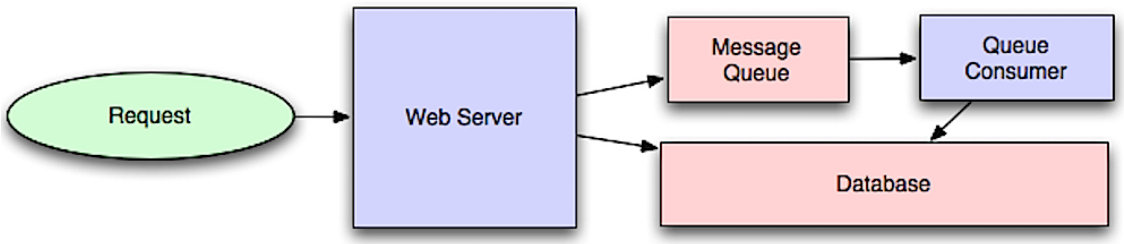


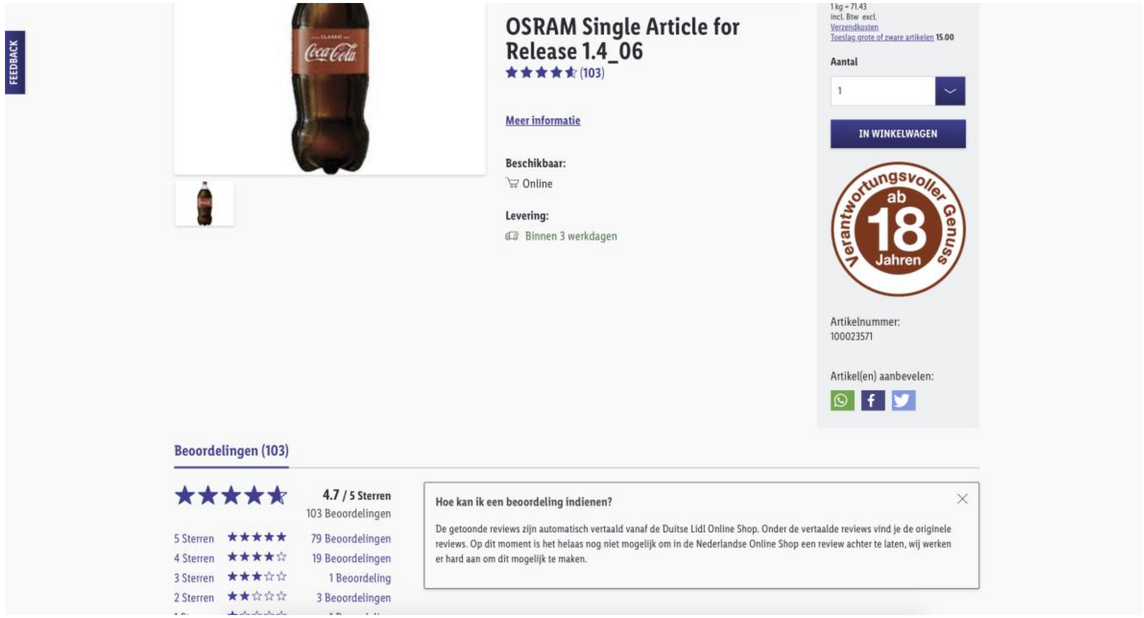
Write-back


Write-Back
Write to cache
Write to database
Application
Cache
Database

Asynchronism

Message Queue





Scaling the DB

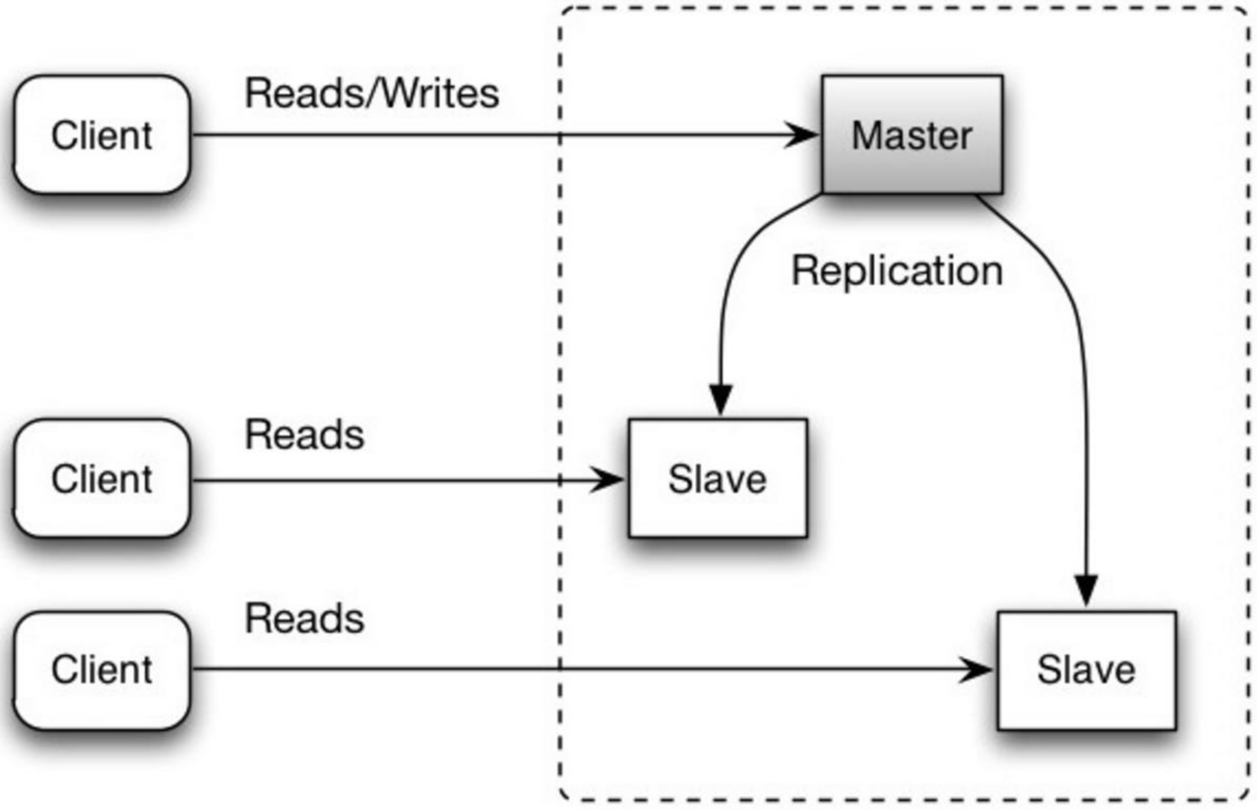
Replication: Master - slave

Cons:

- single point of failure
- Master slave cannot be scaled horizontally
- when failure, a slave has to be promoted to master to take over its place. No automatic failover
- Downtime and possibly loss of data when a master fails

Pros:

- You can split read and write requests to different servers



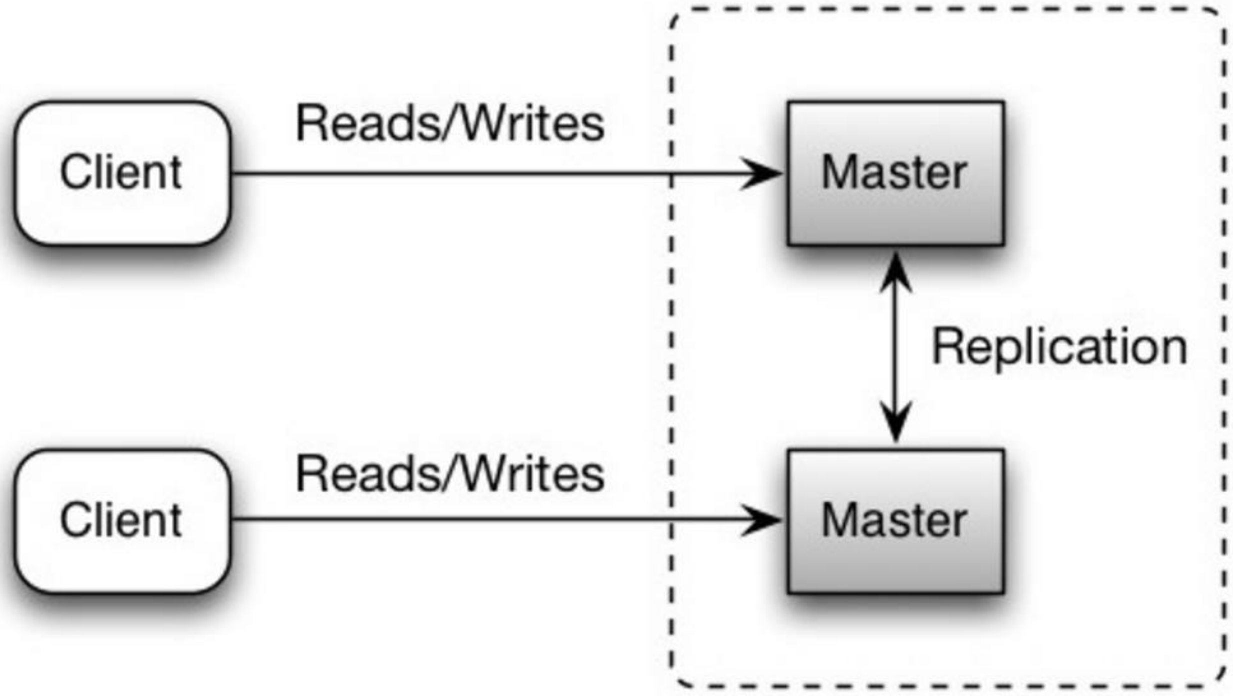
Replication: Multi – master

Cons:

- a load balancer or need of making changes to your application logic to determine where to write
- either loosely consistent or increased write latency due to synchronization

Pros:

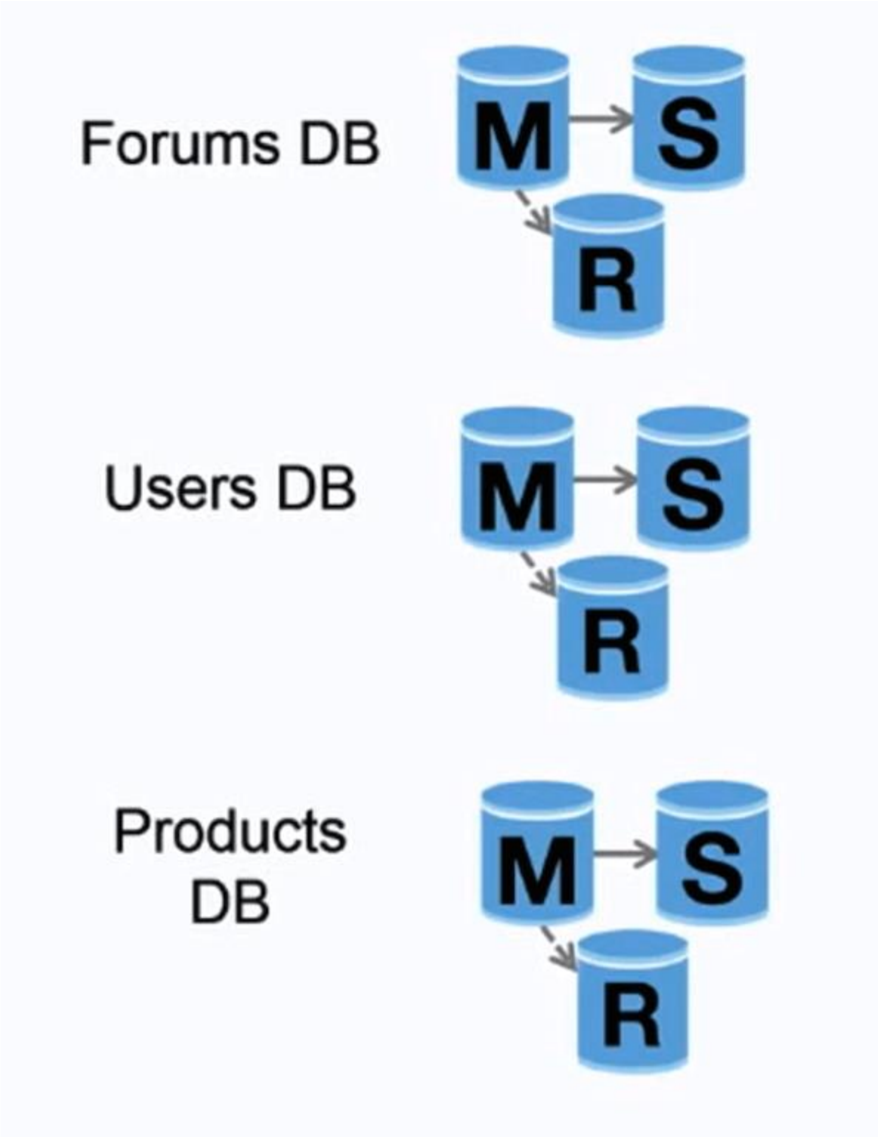
- Failover semi-automatic because of multiple master nodes
- Distributes write load



Partitioning: Federation (Vertical)

Disadvantages

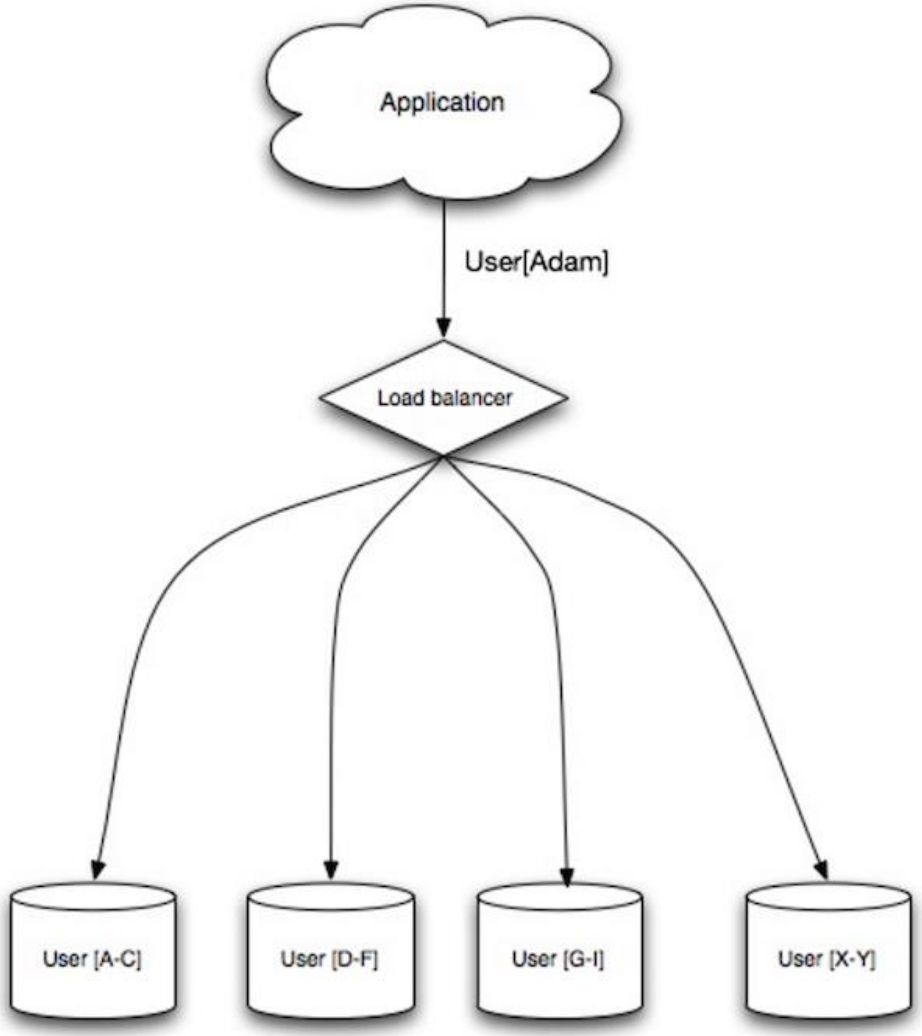
- not effective if your schema requires huge functions or tables
- update application logic to determine which database to read and write
- Joining data from two databases is more complex (FK)
- more hardware and additional complexity



Partitioning: Sharding (Hozirontal)

Disadvantages:

- update application logic to work with shards; complex SQL queries.
- rebalancing
- joining data from multiple shards is more complex (FK)
- more hardware and additional complexity



25k signups in the first day

scaling = replacing all components of a car while driving it at 100mph

moved db to its own machine

but photos kept growing and growing...

Instagram DB Scaling

vertical partitioning

photosdb > 60GB

horizontal partitioning!

aka: sharding

Instagram DB Scaling

what's painful about sharding?

1 data retrieval

2 what happens if one of your shards gets too big?

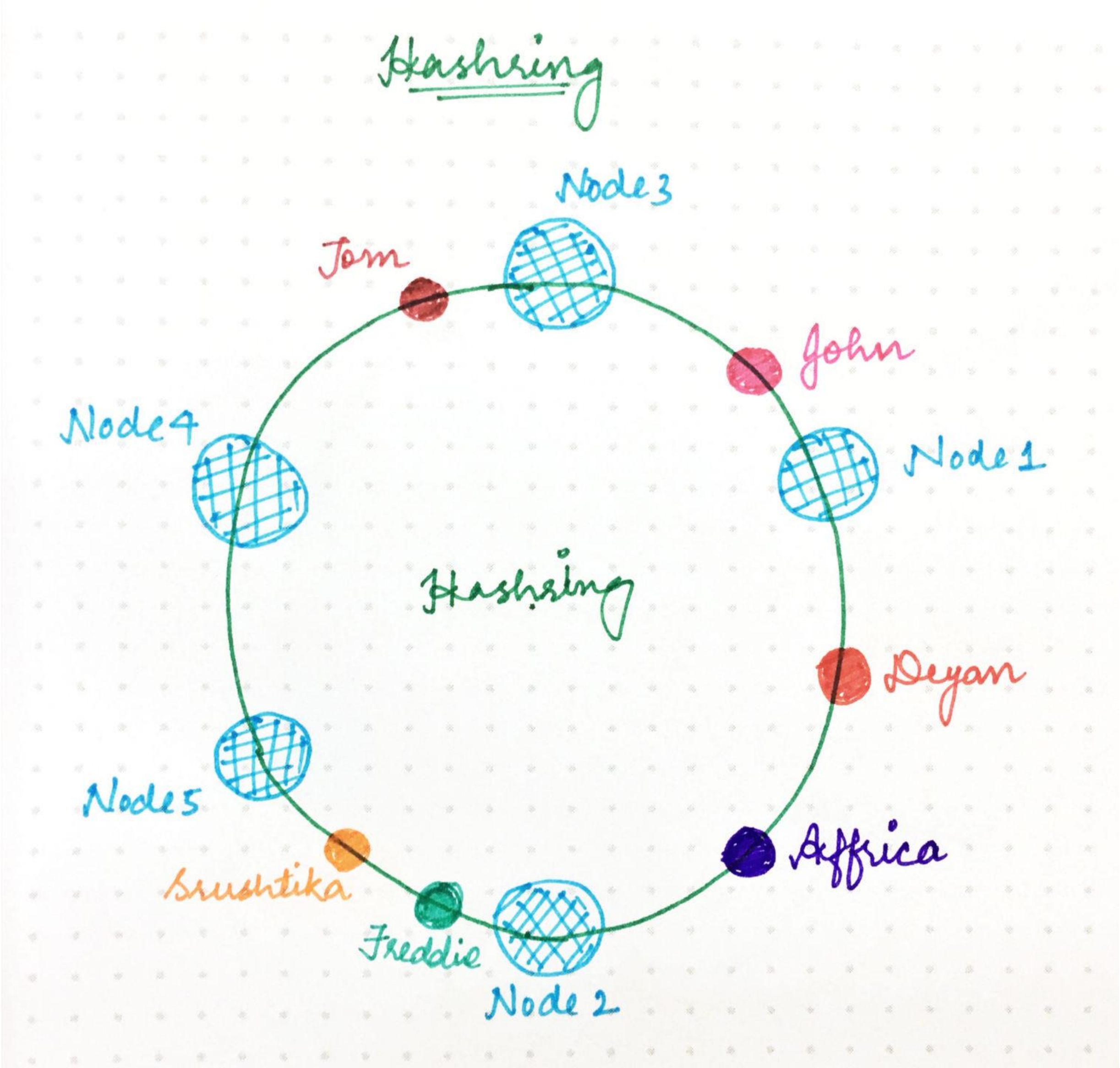
// 8 logical shards on ~~2~~ 4 machines

```
user_id % 8 = logical shard

logical shards -> physical shard map

{
  0: A, 1: A,
  2: C, 3: C,
  4: B, 5: B,
  6: D, 7: D
}
```

can do this as long as you have more logical shards than physical ones

Instagram DB Scaling

Consistent Hashing


Hashring
Node 3
Tom
John
Node 4
Node 1
Hashring
Deyan
Node 5
Affrica
Srushtika
Freddie
Node 2

Resources

- https://github.com/donnemartin/system-design-primer (Donne Martin github)
- Educative.io System Design course
- https://www.slideshare.net/iammutex/scaling-instagram (Scaling Instagram)
- https://www.youtube.com/watch?v=zaRkONvyGr8 (Consistent Hashing)
- https://svenbayer.blog/2018/09/30/accelerate-microservices-with-refresh-ahead-caching/
- Google :)