System Design

Cracking a code interview

Dmitry Ryabokon, github.com/dryabokon

Concepts

- Scaling
- Storage
- Databases
- Messaging
- Caching
- Load Balance
- Map-Reduce
- Tools
- Consistent Hashing

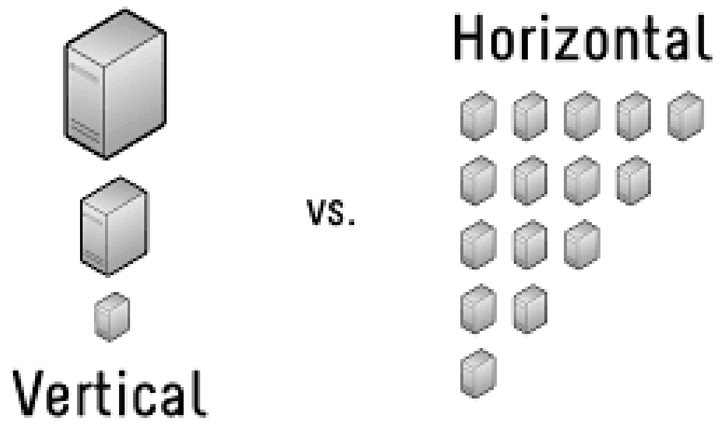
https://www.freecodecamp.org/news/systems-design-for-interviews/

Scaling

Scaling

- Vertical: more RAM/CPU to existing host
  - Can be expensive
  - Limitations per max memory per single host
  - Less problem for distributed systems

- Horizontal : add another host
  - preferred over vertical scaling
  - Should handle issues with distributed systems



Microservices vs monolithic app

Microservices are small applications, usually running on their own infrastructure or within a virtual machine, which is responsible for one segment of a larger application's business requirements

Monolithic application exists on one centralized piece of infrastructure

Breaking application into micro-services = separate application for each of the entities
each app exposes its own smaller REST API for interacting with the separate entities

8 Misconceptions

- Reliable network
- Zero latency
- Infinite bandwidth
- Secure network
- Frozen network topology (ring, bus, star, meshed)
- One Admin
- Zero transport cost
- Homogeneous network protocol

Parallel vs distributed computing

- Parallel : each component has access to shared memory
- Distributed: there is no shared memory and each component must communicate information about its internal state via message passing

Failure types

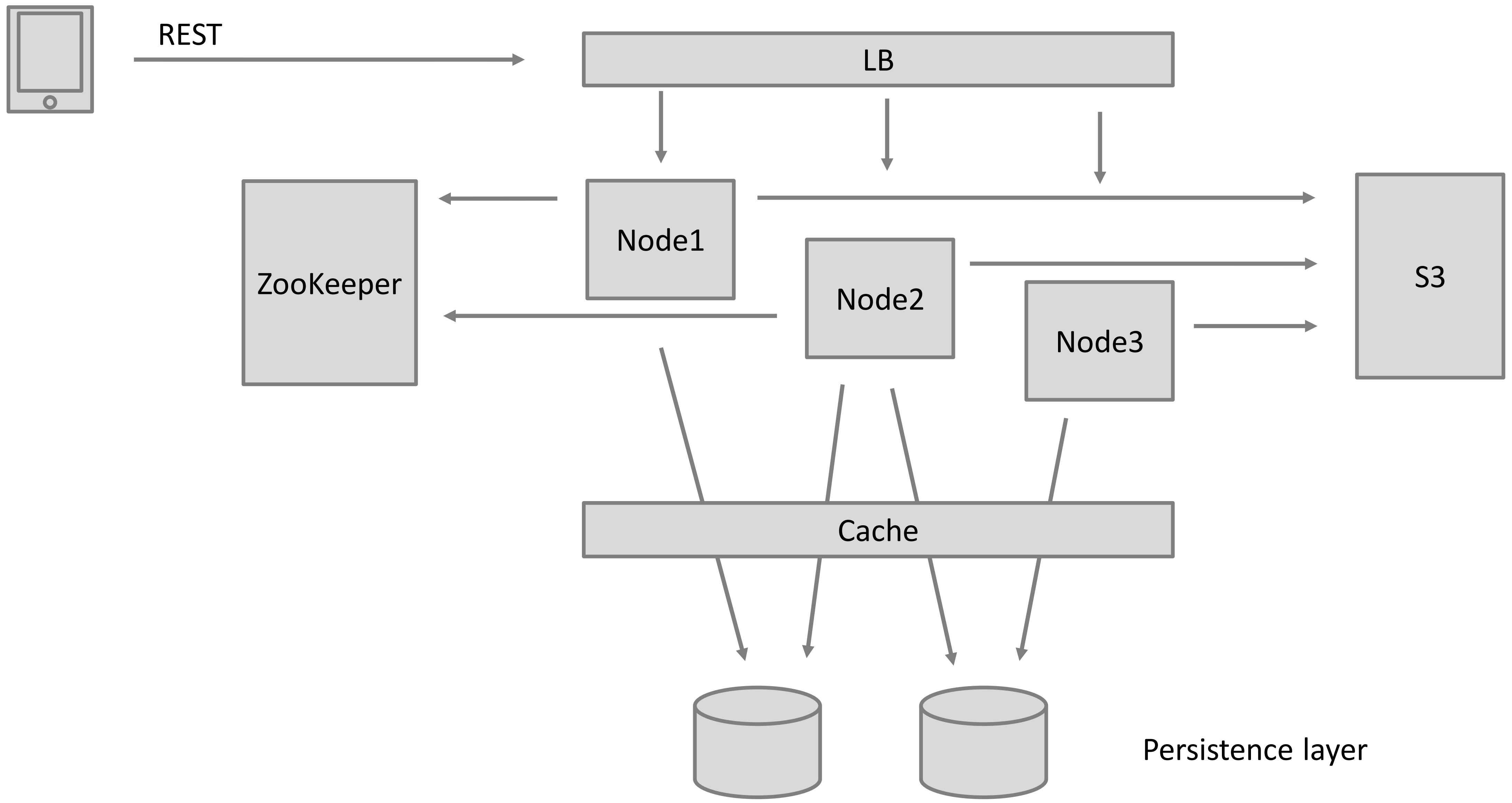
•Halting failures: A component simply stops. There is no way to detect the failure except by timeout: it either stops sending "I'm alive" (heartbeat) messages or fails to respond to requests. Your computer freezing is a halting failure.
•Fail-stop: A halting failure with some kind of notification to other components. A network file server telling its clients it is about to go down is a fail-stop.
•Omission failures: Failure to send/receive messages primarily due to lack of buffering space, which causes a message to be discarded with no notification to either the sender or receiver. This can happen when routers become overloaded.
•Network failures: A network link breaks.
•Network partition failure: A network fragments into two or more disjoint sub-networks within which messages can be sent, but between which messages are lost. This can occur due to a network failure.
•Timing failures: A temporal property of the system is violated. For example, clocks on different computers which are used to coordinate processes are not synchronized; when a message is delayed longer than a threshold period, etc.
•Byzantine failures: This captures several types of faulty behaviors including data corruption or loss, failures caused by malicious programs, etc. [1]

Performance vs Scalability

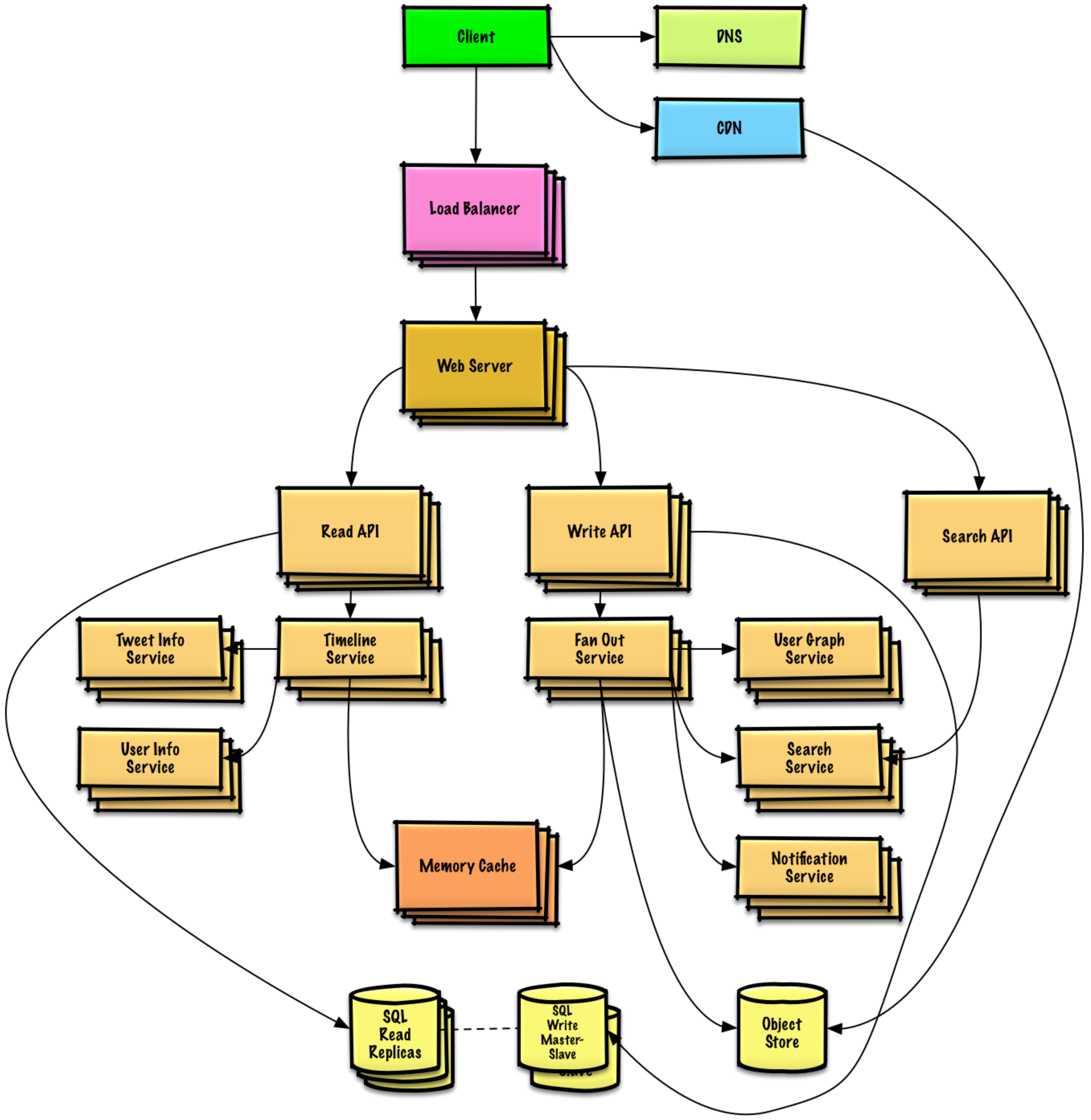
A service is **scalable** if it results in increased **performance** proportional to resources added.

performance problem: system is slow for a single user.
scalability problem: system is fast for a single user but slow under heavy load

The approach



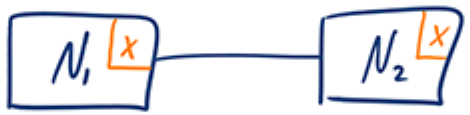
The approach



Storage

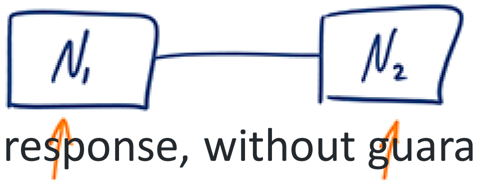
Storage: CAP

Consistency



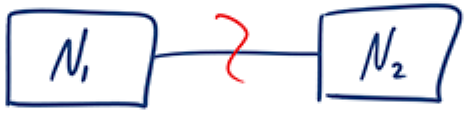
Every read receives the most recent write or an error

Availability

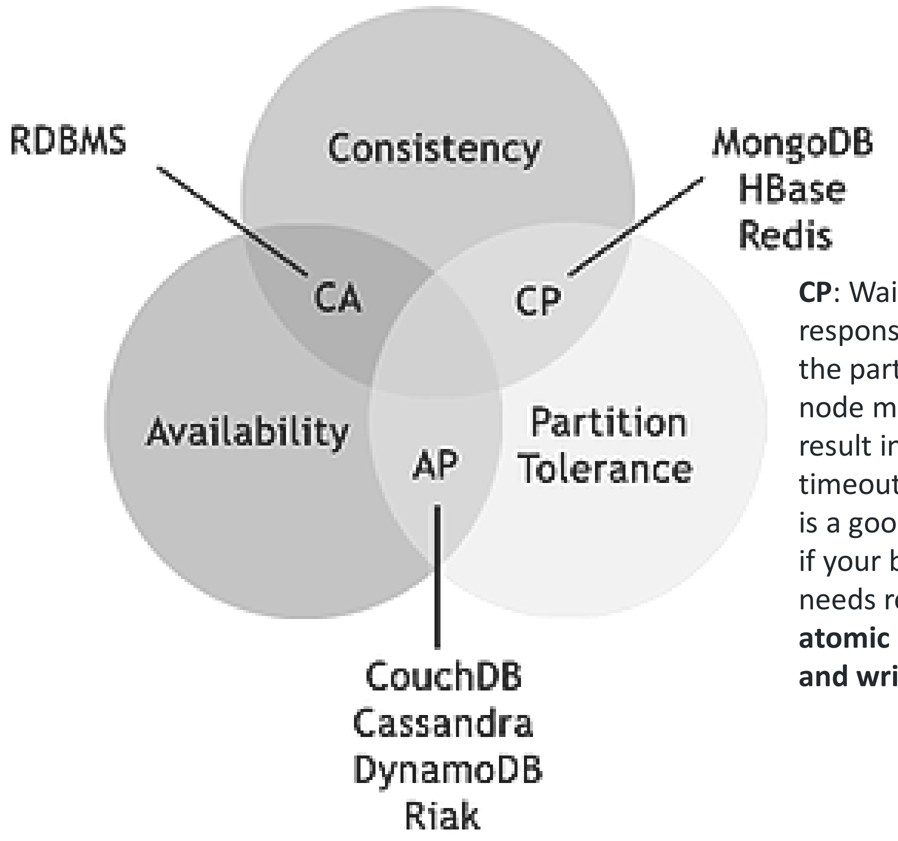


Every request receives a response, without guarantee that it contains the most recent version of the information

Partition Tolerance



The system continues to operate despite arbitrary partitioning due to network failures

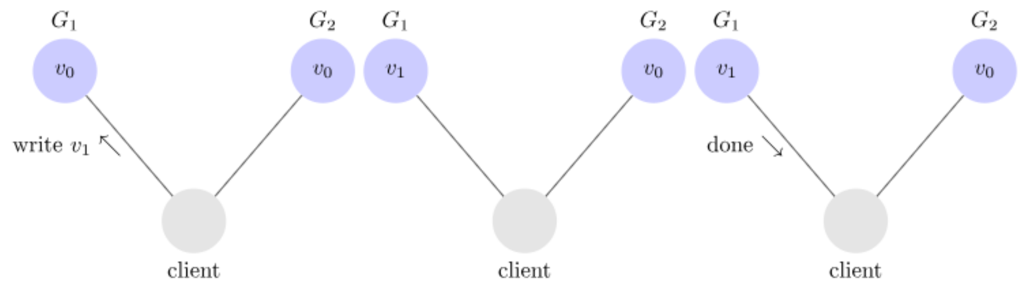


CP: Waiting for a response from the partitioned node might result in a timeout error. CP is a good choice if your business needs require **atomic reads and writes**

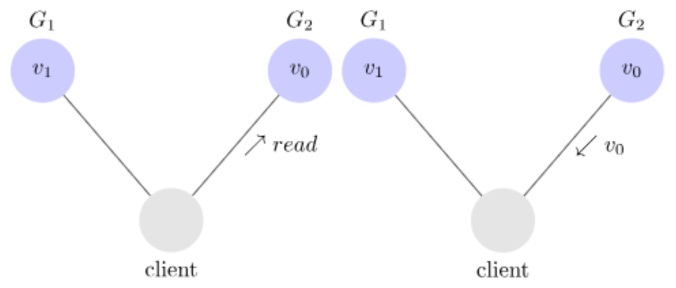
AP: Responses return the most recent version of the data available on a node, which might not be the latest. Writes might take some time to propagate when the partition is resolved. AP is a good choice if the business needs allow for **eventual consistency** or when the system needs to continue working despite external errors.

Storage: CAP

Next, we have our client request that v_1 be written to G_1. Since our system is available, G_1 must respond. Since the network is partitioned, however, G_1 cannot replicate its data to G_2. Gilbert and Lynch call this phase of execution α_1.



Next, we have our client issue a read request to G_2. Again, since our system is available, G_2 must respond. And since the network is partitioned, G_2 cannot update its value from G_1. It returns v_0. Gilbert and Lynch call this phase of execution α_2.



G_2 returns v_0 to our client after the client had already written v_1 to G_1. This is inconsistent.

We assumed a consistent, available, partition tolerant system existed, but we just showed that there exists an execution for any such system in which the system acts inconsistently. Thus, no such system exists.

https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/

Consistency: Eventual, Strong Eventual, Strong

Conflicts arise because each node can update its own copy. If we read the data from different nodes we will see different values

Strong consistent
Data will get passed on to all the replicas as soon as a write request comes to one of the replicas of the database.
But during the time these replicas are being updated with new data, response to any subsequent read/write requests by any of the replicas will get delayed as all replicas are busy in keeping each other consistent

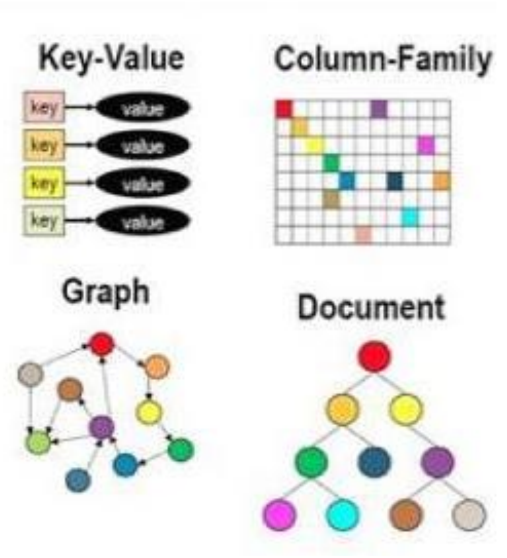
Eventual consistency
Used in distributed computing to achieve high availability that informally guarantees that,
if no new updates are made to a given data item, **eventually** all accesses to that item will return the last updated value

Weak consistency
After a write, reads may or may not see it. A best effort approach is taken.
This approach is seen in systems such as memcached. Weak consistency works well in real time use cases such as VoIP, video chat, and realtime multiplayer games. For example, if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.

Databases

No-SQL

Key-Value Persistent	Key-Value Volatile	Column	Graph	Document
Redis (CP)	memcached	Cassandra (AP)	FlockDB (twitter)	MongoDB (CP)
Membase (memcached)	Hazelcast	BigTable (Google)		CouchDB (AP)
Dynamo (AWS)		SimpleDB (AWS)		



ACID vs BASE

BASE (no-SQL)

• **B**asically **A**vailable:	system guarantee availability in terms of the CAP
• **S**oft state:	state of the system may change over time, even without input
• **E**ventual consistency:	updates will eventually ripple through to all servers, given enough time.

ACID (traditional relational DB)

• **A**tomicity:	each transaction is a "unit“ which either succeeds completely, or fails completely
• **C**onsistency:	ensures transaction can only bring the DB from one valid state to another
• **I**solation	state(concurrent transactions) = state(executed sequentially transactions)
• **D**urability	once transaction is committed, it will remain committed even in the case of a system failure

Pessimistic vs Optimistic locking

Pessimistic Locking

- lock the record for your exclusive use until you have finished with it

This strategy is most applicable
- direct connection to the database or
- an externally available transaction ID that can be used independently of the connection
- cases when a collision is anticipated

Pessimistic vs Optimistic locking

Optimistic Locking

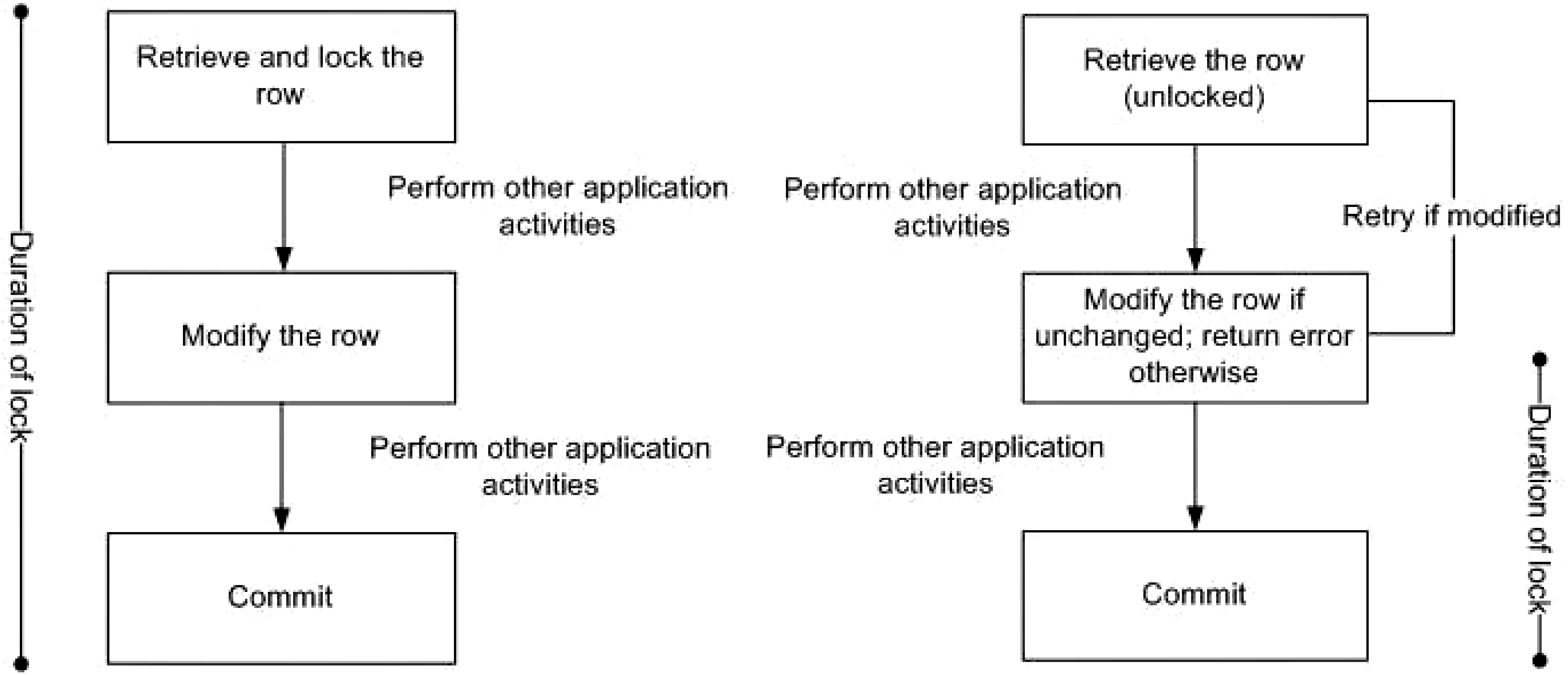
- read a record,
- take note of a version number (dates/timestamps/checksums/hashes)
- check that the version hasn't changed before you do a write operation
- Write the record
- filter the update on the version to make sure it's atomic and update the version in one hit.
    - Record should not be updated between when you check the version and write the record

If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable for

- high-volume systems where you do not necessarily maintain a connection to the DB
- cases when you don't expect many collisions.

Pessimistic vs Optimistic locking


Retrieve and lock the row
Modify the row
Commit
Duration of lock
Perform other application activities
Perform other application activities
Perform other application activities
Perform other application activities
Retrieve the row (unlocked)
Modify the row if unchanged; return error otherwise
Commit
Retry if modified
Duration of lock

Data sharding = Partitioning

- Horizontal sharding = Range based sharding
- Vertical sharding = Different features of an entity are placed in different shards
- Key or hash based sharding = This hash value determines which database server(shard) to use.
  - if we want to add X more servers, keys would need to be remapped and migrated to new servers
  - Both new and old hash function are not valid. So requests cannot be serviced till the migration completes
  - Consistent hashing can solve this

Drawbacks

- Database Joins become more expensive and not feasible in certain cases
- application layer needs additional level of asynchronous code and exception handling
- cross machine joins may not be an option for high availability SLA

Use sharding when

- data need to scale beyond a single storage node
- improve performance by reducing contention in a data store

https://www.acodersjourney.com/database-sharding/

Caching

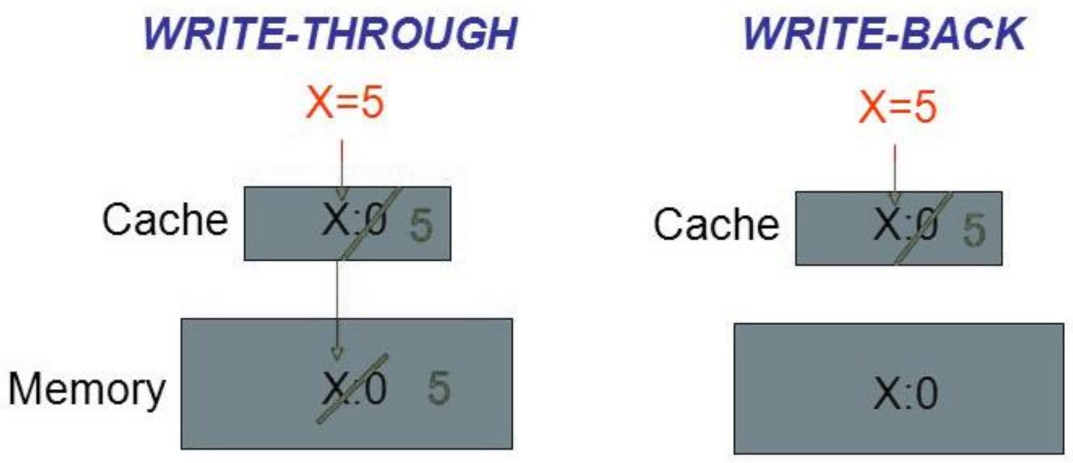
Caching

- Should be small to fin in memory
- Is not a source of True
- Should comply with **caching eviction policy**

FIFO, LIFO, Least Recently Used (LRU), Most Recently Used (MRU), Least Frequently Used (LFU), Random Replacement (RR).

- write-through: write to the cache causes a synchronous write to the backing store
- write-back: writes are not immediately mirrored to DB store. Instead, the cache tracks which of its locations have been written over and marks these locations as dirty. The data in these locations is written back to the backing store when those data are evicted from the cache, an effect referred to as a lazy write.

- Reduce DB/network load (get recent data from cache)
- Reduce recalculations (e.g. aggregations)



Load Balancing

Load balancing: methods

- Round robin: an incoming request is routed to each available server in a sequential manner.

- Weighted round robin: a static weight is preassigned to each server

- Least connection: This method reduces the overload of a server by assigning an incoming request to a server with the lowest number of connections currently maintained.

- Weighted least connection: In this method, a weight is added to a server depending on its capacity. This weight is used with the least connection method to determine the load allocated to each server.

- Least connection slow start time -- Here, a ramp-up time is specified for a server using least connection scheduling to ensure that the server is not overloaded on startup.
- Agent-based adaptive balancing -- This is an adaptive method that regularly checks a server irrespective of its weight to schedule the traffic in real time.
- Fixed weighted -- In this method, the weight of each server is preassigned and most of the requests are routed to the server with the highest priority. If the server with the highest priority fails, the server that has the second highest priority takes over the services.
- Weighted response -- Here, the response time from each server is used to calculate its weight.
- Source IP hash -- In this method, an IP hash is used to find the server that must attend to a request.

Load balancing: L4 vs L7

Open systems interconnection - Transmission Control Protocol - Internet protocol

	OSI Layer	TCP/IP	Datagrams are called
Software	**Layer 7** Application	HTTP, SMTP, IMAP, SNMP, POP3, FTP	**Upper Layer Data**
	Layer 6 Presentation	ASCII Characters, MPEG, SSL, TSL, Compression (Encryption & Decryption)	
	Layer 5 Session	NetBIOS, SAP, Handshaking connection	
	Layer 4 Transport	TCP, UDP	**Segment**
	Layer 3 Network	IPv4, IPv6, ICMP, IPSec, MPLS, ARP	**Packet**
Hardware	**Layer 2** Data Link	Ethernet, 802.1x, PPP, ATM, Fiber Channel, MPLS, FDDI, MAC Addresses	**Frame**
	Layer 1 Physical	Cables, Connectors, Hubs (DLS, RS232, 10BaseT, 100BaseTX, ISDN, T1)	**Bits**

Load balancing: L4 vs L7

L4: directs traffic based on data from network and transport layer protocols, such as IP address and TCP port.

L7: adds content switching to load balancing. This allows routing decisions based on attributes like HTTP header, uniform resource identifier, SSL session ID and HTML form data. Most of LBs are working like this

https://www.educative.io/collection/page/5668639101419520/5649050225344512/5747976207073280

https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236

Messaging

Publish-Subscribe vs Queues

A message queue receives incoming messages and ensures that each message for a given topic or channel is delivered to and processed by exactly one consumer. Message queues can support high rates of consumption by adding multiple consumers for each topic, but only one consumer will receive each message on the topic. Which consumer receives which message is determined by the implementation of the message queue. To ensure that a message is only processed by one consumer, each message is deleted from the queue once it has been received and processed by a consumer (i.e. once a consumer has acknowledged consumption of the message to the messaging system).

in contrast to message queuing, publish-subscribe messaging allows multiple consumers to receive each message in a topic. Further, pub-sub messaging ensures that each consumer receives messages in a topic in the exact order in which they were received by the messaging system

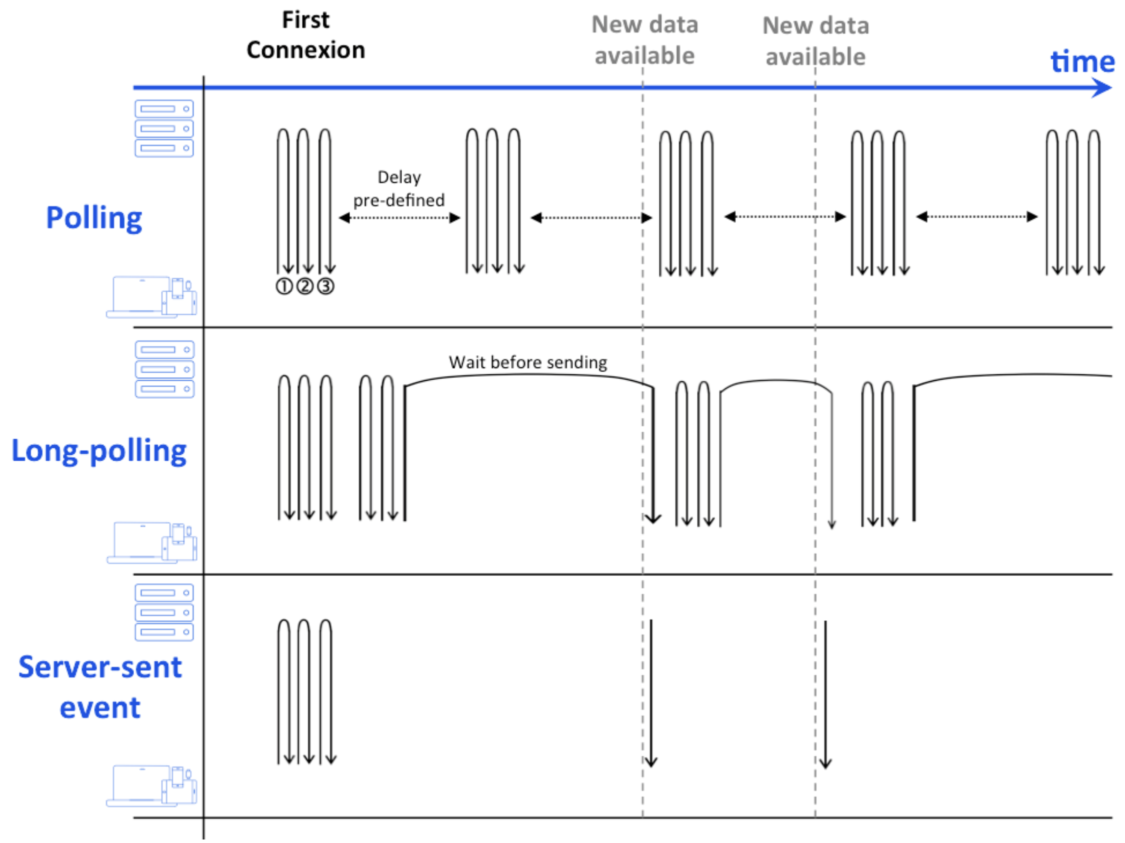
https://dzone.com/articles/comparing-publish-subscribe-messaging-and-message

Long-pooling vs WebSockets vs Server-Send Events

- **Long/short polling (client pull)**
- client asking server for updates at certain regular intervals
- *Short polling* is an AJAX-based timer that calls at fixed delays
- *Long polling* is based on [Comet]

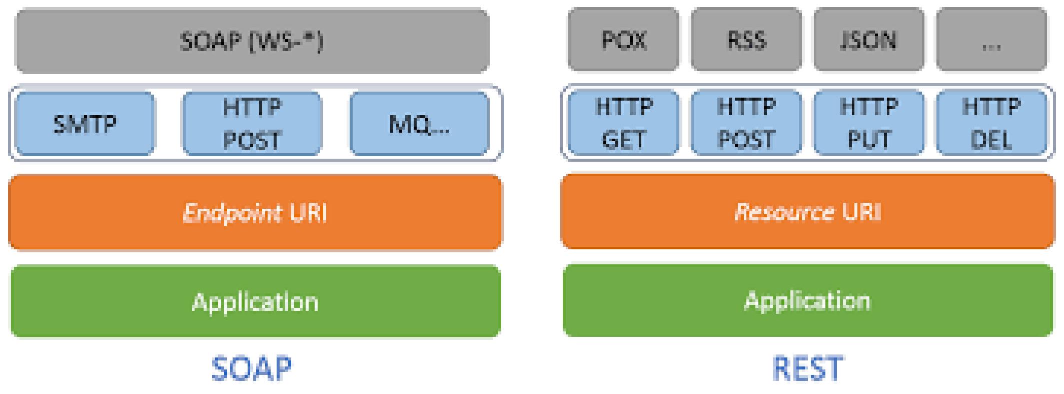
- **WebSockets** (server push)
- server is proactively pushing updates to the client (reverse of client pull)

- **Server-Sent Events** (server push)



https://www.educative.io/collection/page/5668639101419520/5649050225344512/5715426797420544

REST vs SOAP



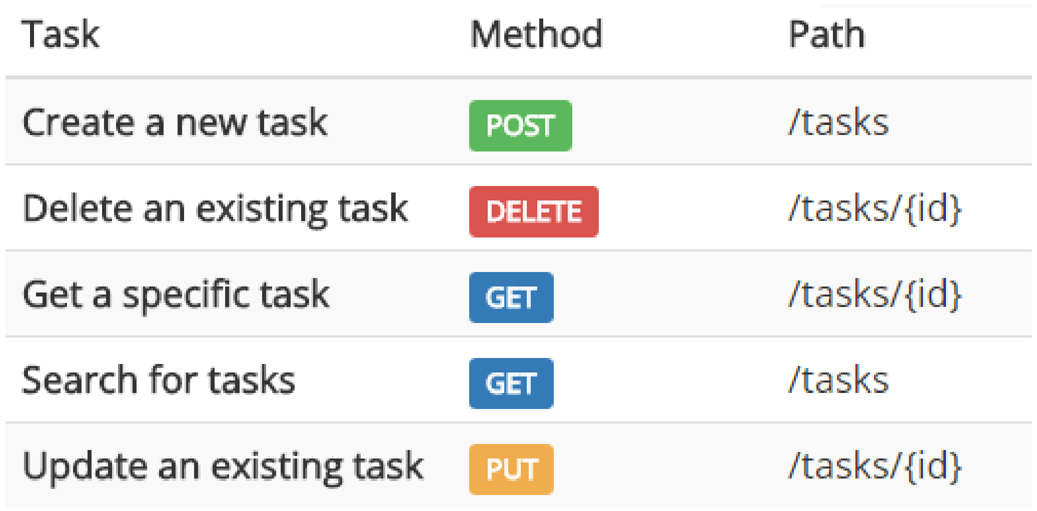
https://www.educba.com/soap-vs-rest/

REST: Representational State Transfer

Key principles:

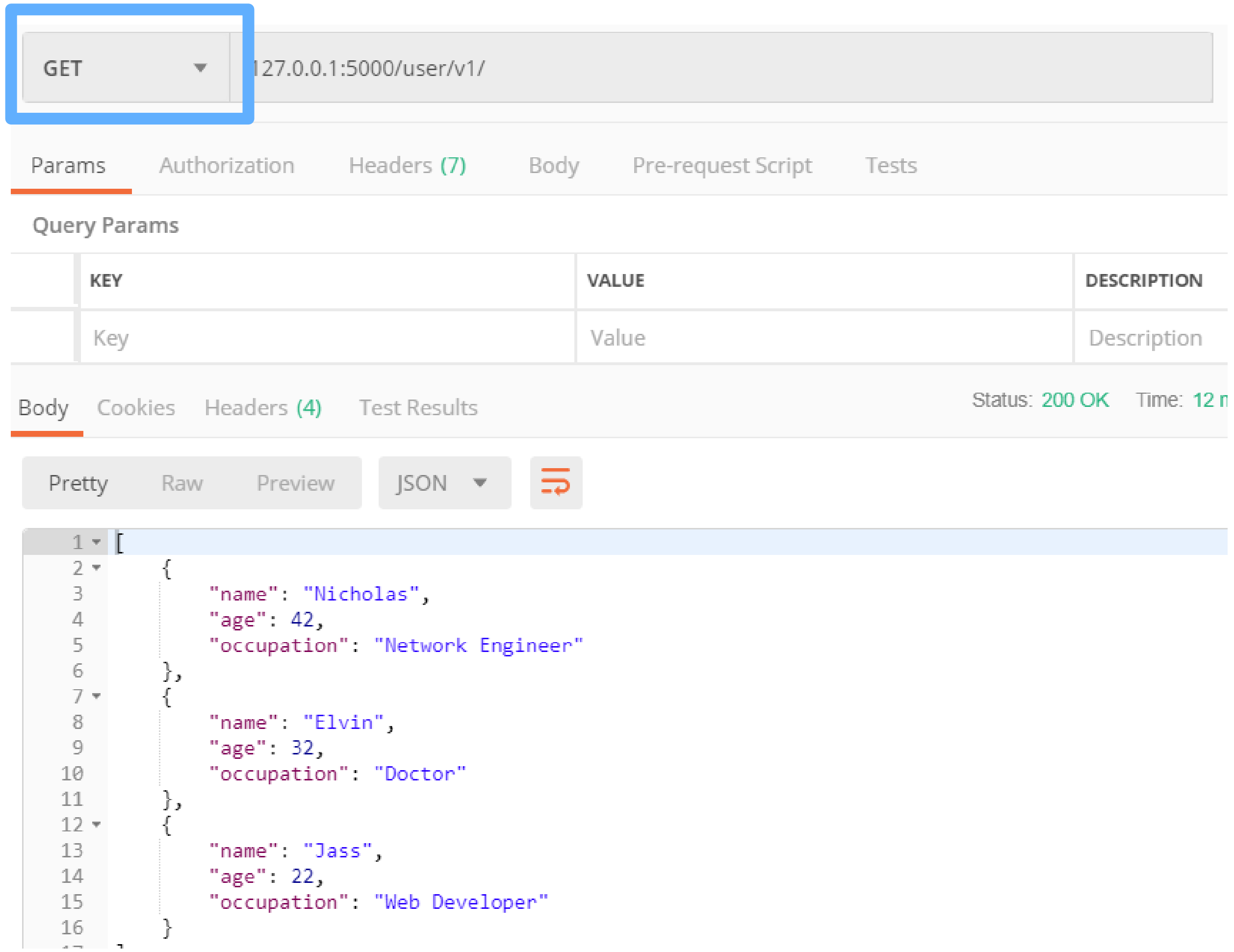
- this is a client-server architecture
- **It’s stateless**: *communication between the client and the server always contains all the information needed to perform the request.*
- There is **no session state** in the server, it is kept entirely on the client's side. *(e.g. if access to a resource requires authentication, then the client needs to authenticate itself with every request)*

- Cacheable
- provides a uniform interface between components.



Task	Method	Path
Create a new task	POST	/tasks
Delete an existing task	DELETE	/tasks/{id}
Get a specific task	GET	/tasks/{id}
Search for tasks	GET	/tasks
Update an existing task	PUT	/tasks/{id}

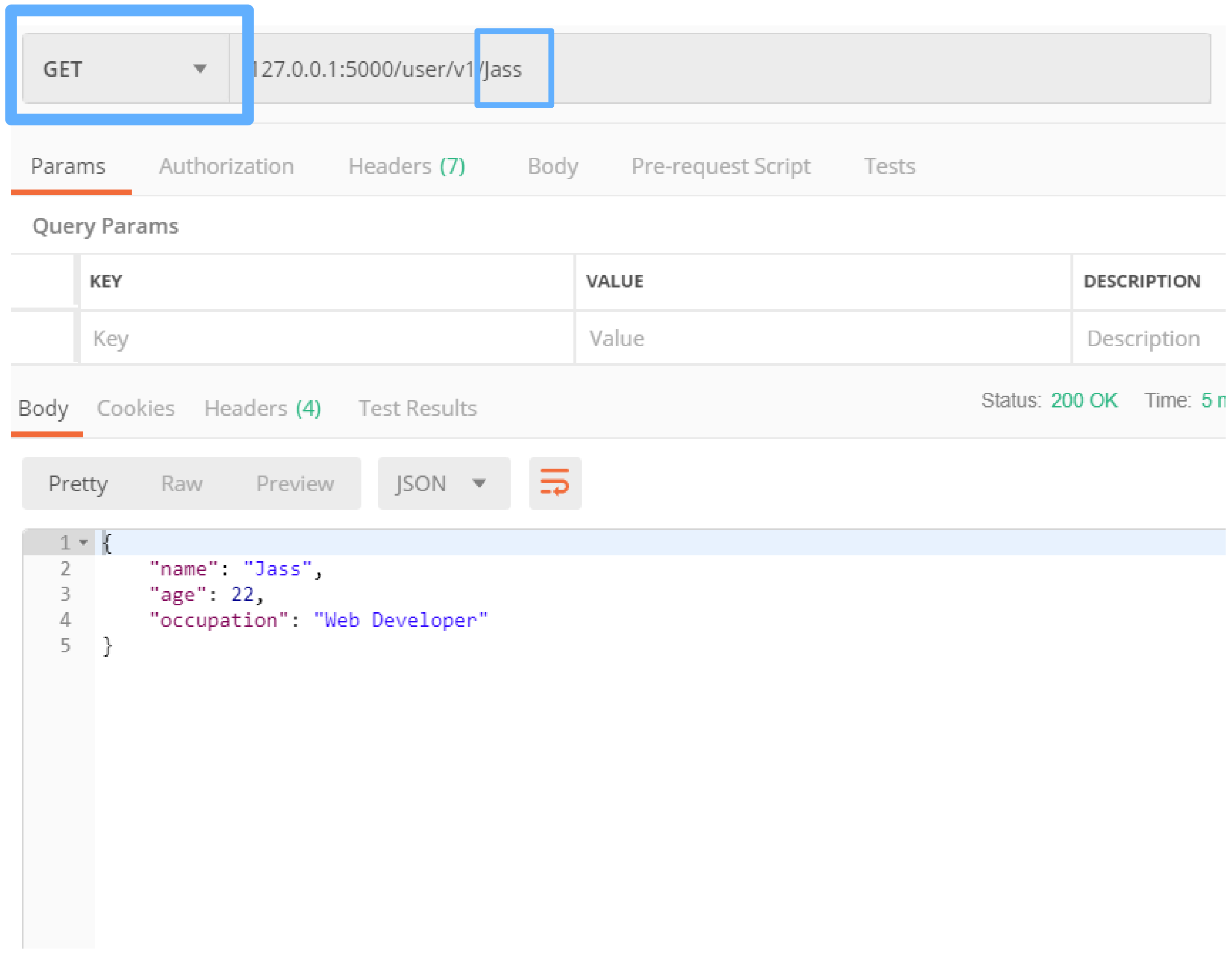
REST: GET - collection



```
[
    {
        "name": "Nicholas",
        "age": 42,
        "occupation": "Network Engineer"
    },
    {
        "name": "Elvin",
        "age": 32,
        "occupation": "Doctor"
    },
    {
        "name": "Jass",
        "age": 22,
        "occupation": "Web Developer"
    }
```

```
@namespace_user.route('/')
class UserList(Resource):
    def get(self):
        return users
```

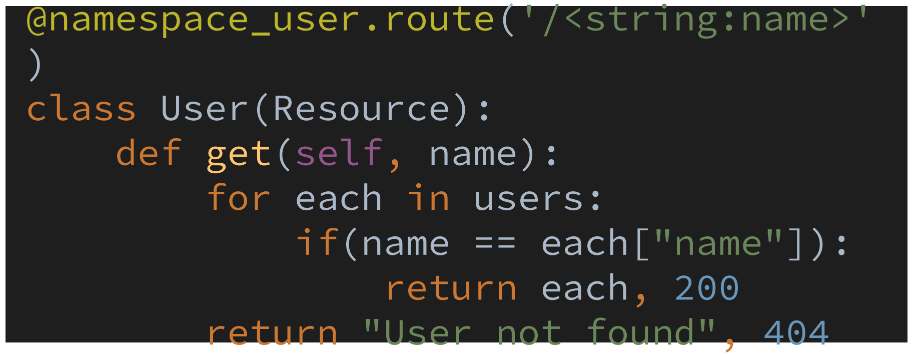
REST: GET - item



GET 127.0.0.1:5000/user/v1/Jass

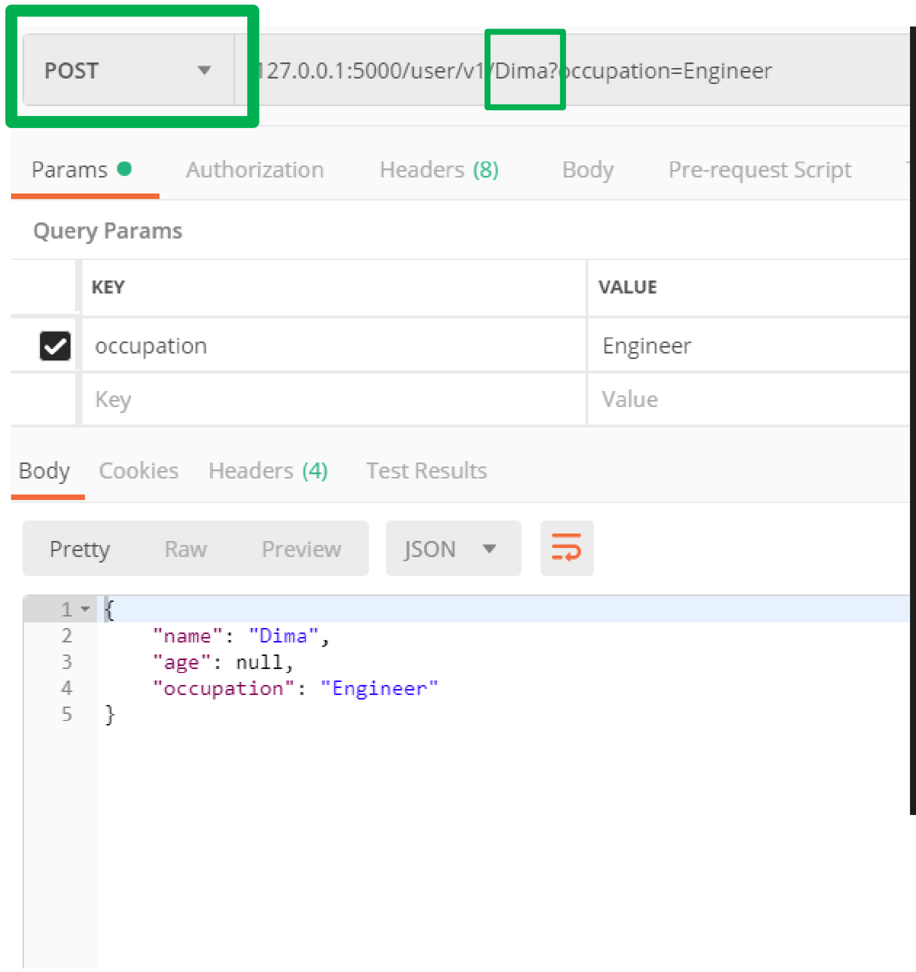
Status: 200 OK

```
{
    "name": "Jass",
    "age": 22,
    "occupation": "Web Developer"
}
```



```
@namespace_user.route('/<string:name>')
class User(Resource):
    def get(self, name):
        for each in users:
            if(name == each["name"]):
                return each, 200
        return "User not found", 404
```

REST: POST - item

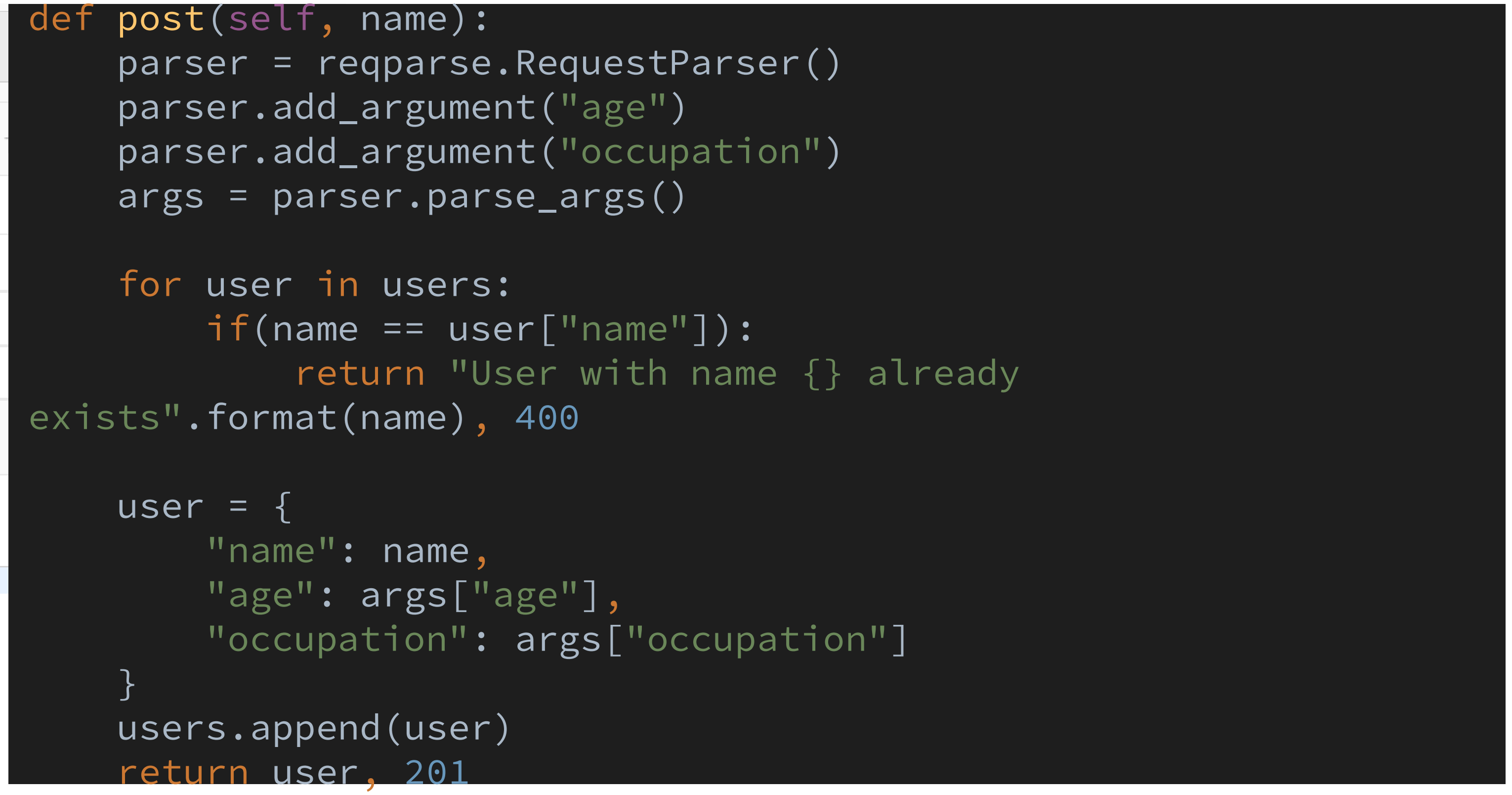


POST 127.0.0.1:5000/user/v1/Dima?occupation=Engineer

Query Params

	KEY	VALUE
✓	occupation	Engineer
	Key	Value

```
{
    "name": "Dima",
    "age": null,
    "occupation": "Engineer"
}
```

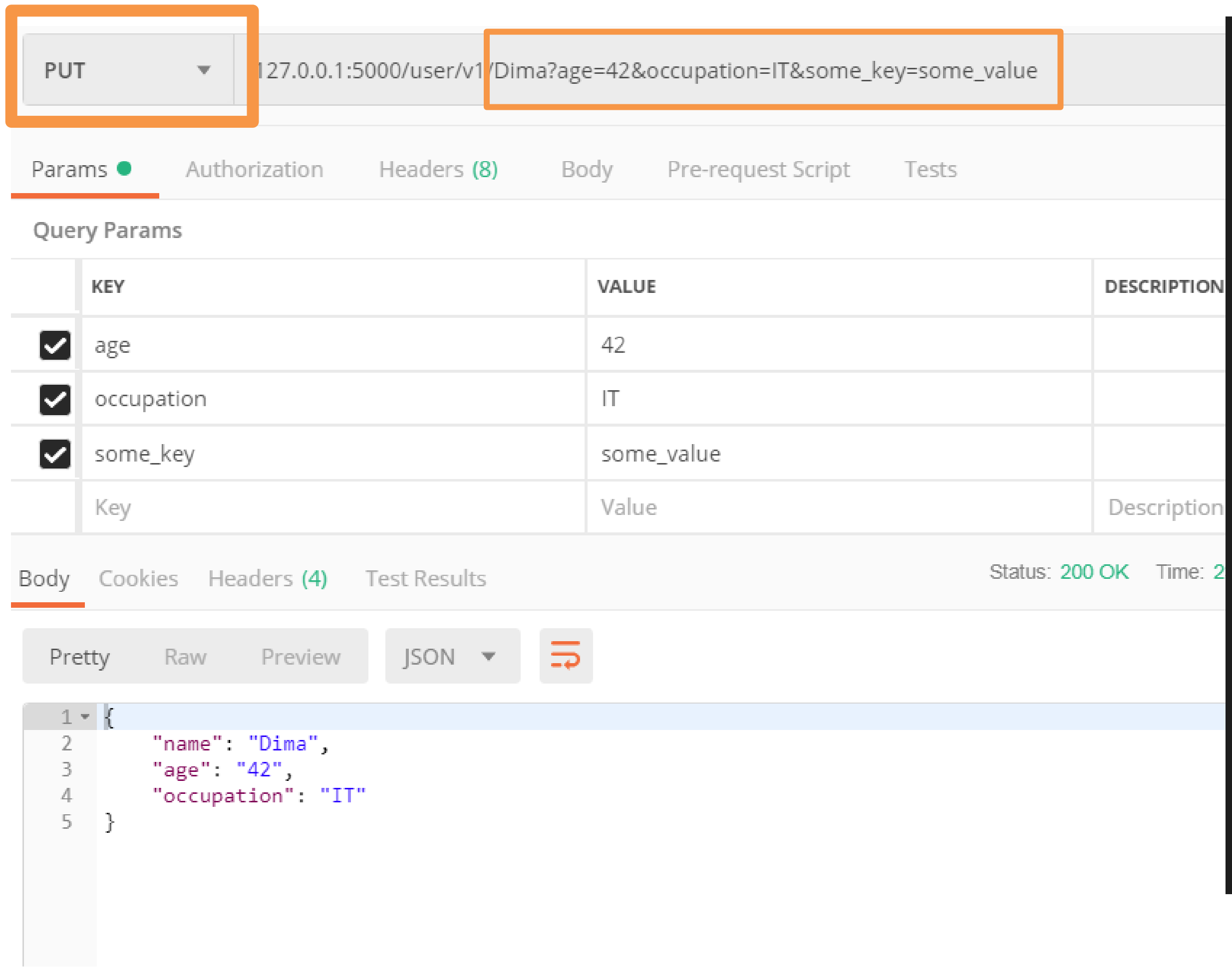


```
def post(self, name):
    parser = reqparse.RequestParser()
    parser.add_argument("age")
    parser.add_argument("occupation")
    args = parser.parse_args()

    for user in users:
        if(name == user["name"]):
            return "User with name {} already
exists".format(name), 400

    user = {
        "name": name,
        "age": args["age"],
        "occupation": args["occupation"]
    }
    users.append(user)
    return user, 201
```

REST: PUT - item



PUT ▾ 127.0.0.1:5000/user/v1/Dima?age=42&occupation=IT&some_key=some_value

Query Params

	KEY	VALUE	DESCRIPTION
✓	age	42	
✓	occupation	IT	
✓	some_key	some_value	
	Key	Value	Description

Status: 200 OK Time: 2

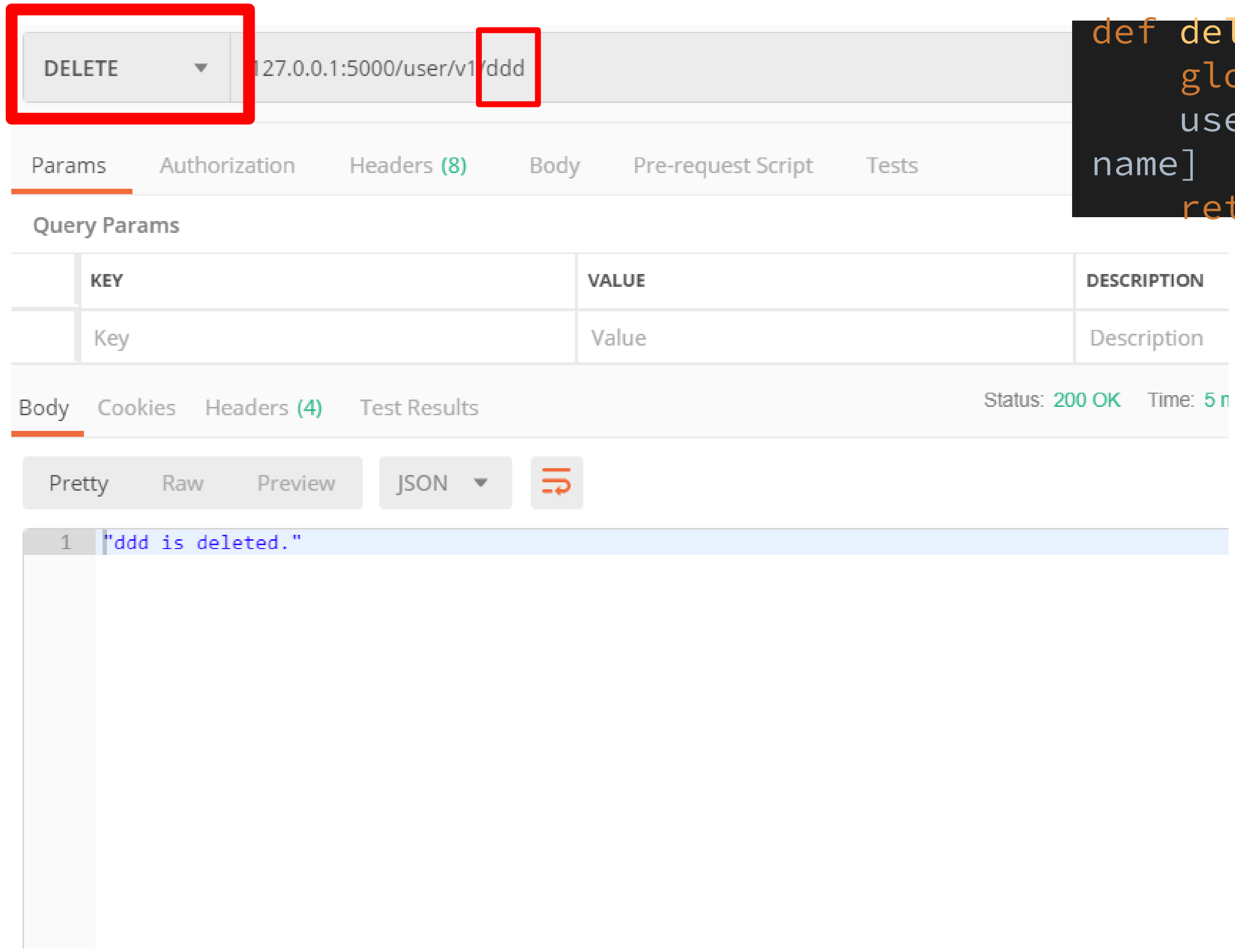
```
{
    "name": "Dima",
    "age": "42",
    "occupation": "IT"
}
```

```
def put(self, name):
    parser = reqparse.RequestParser()
    parser.add_argument("age")
    parser.add_argument("occupation")
    args = parser.parse_args()

    for user in users:
        if(name == user["name"]):
            user["age"] = args["age"]
            user["occupation"] = args["occupation"]
            return user, 200

    user = {
        "name": name,
        "age": args["age"],
        "occupation": args["occupation"]
    }
    users.append(user)
    return user, 201
```

REST: DELETE - item

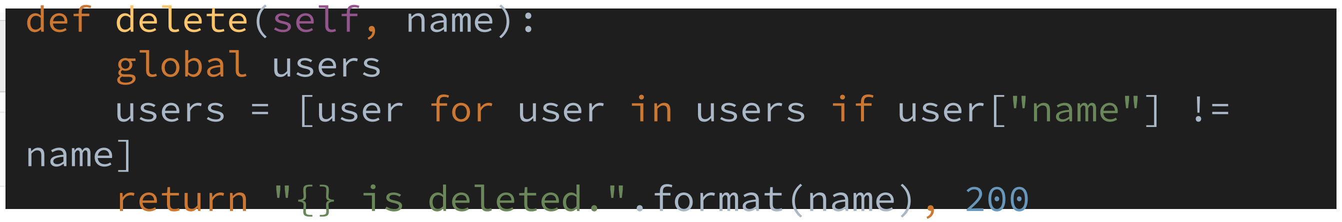


DELETE 127.0.0.1:5000/user/v1/ddd

KEY	VALUE	DESCRIPTION
Key	Value	Description

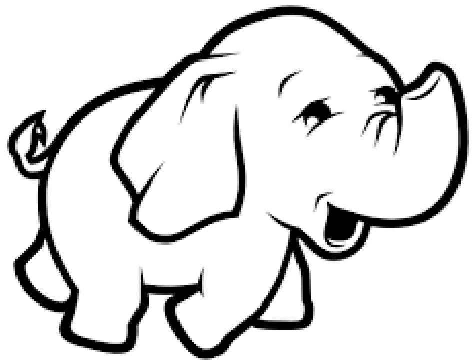
Status: 200 OK Time: 5 m

```
"ddd is deleted."
```

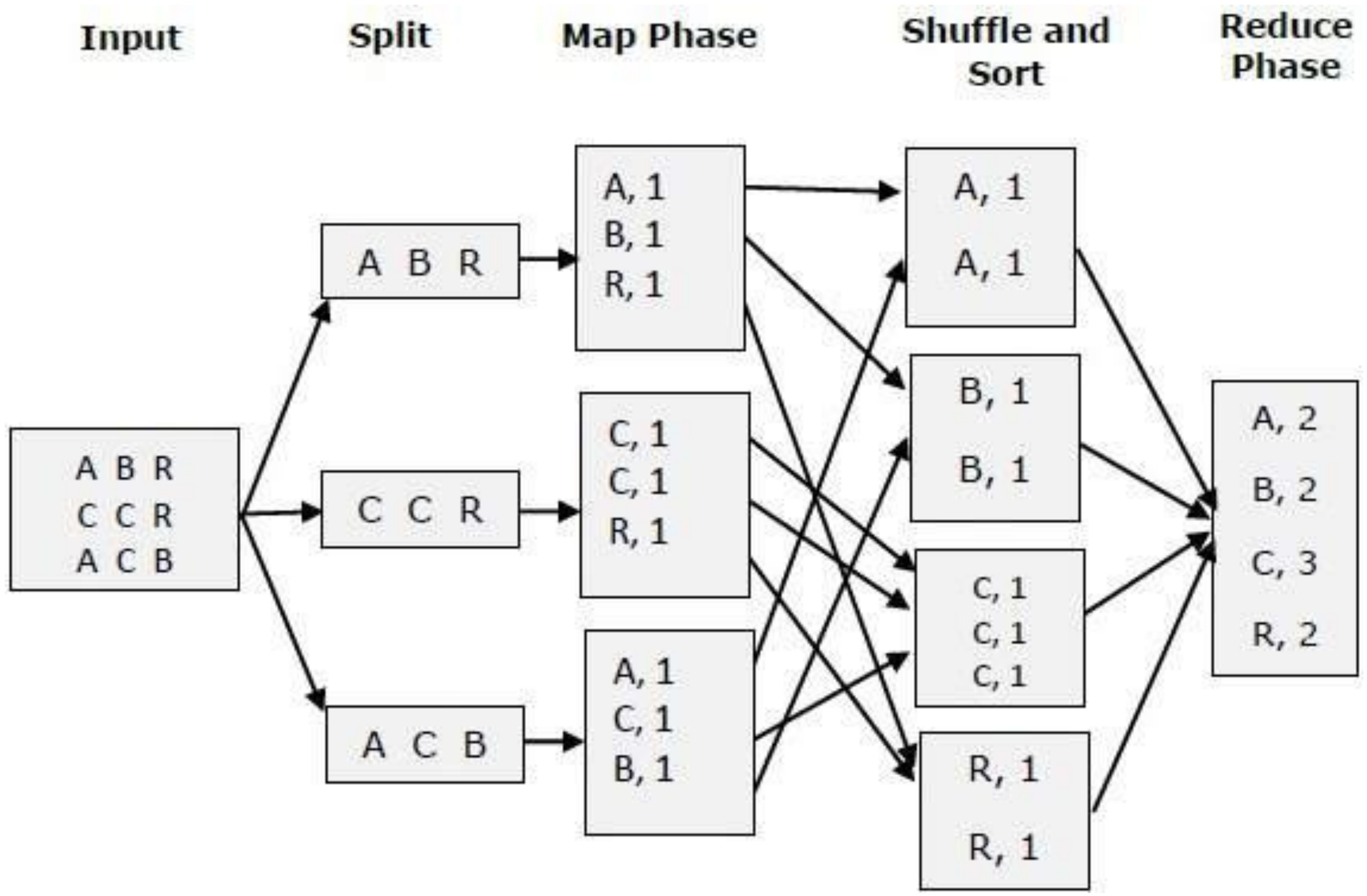


```
def delete(self, name):
    global users
    users = [user for user in users if user["name"] !=
name]
    return "{} is deleted.".format(name), 200
```

Map-Reduce

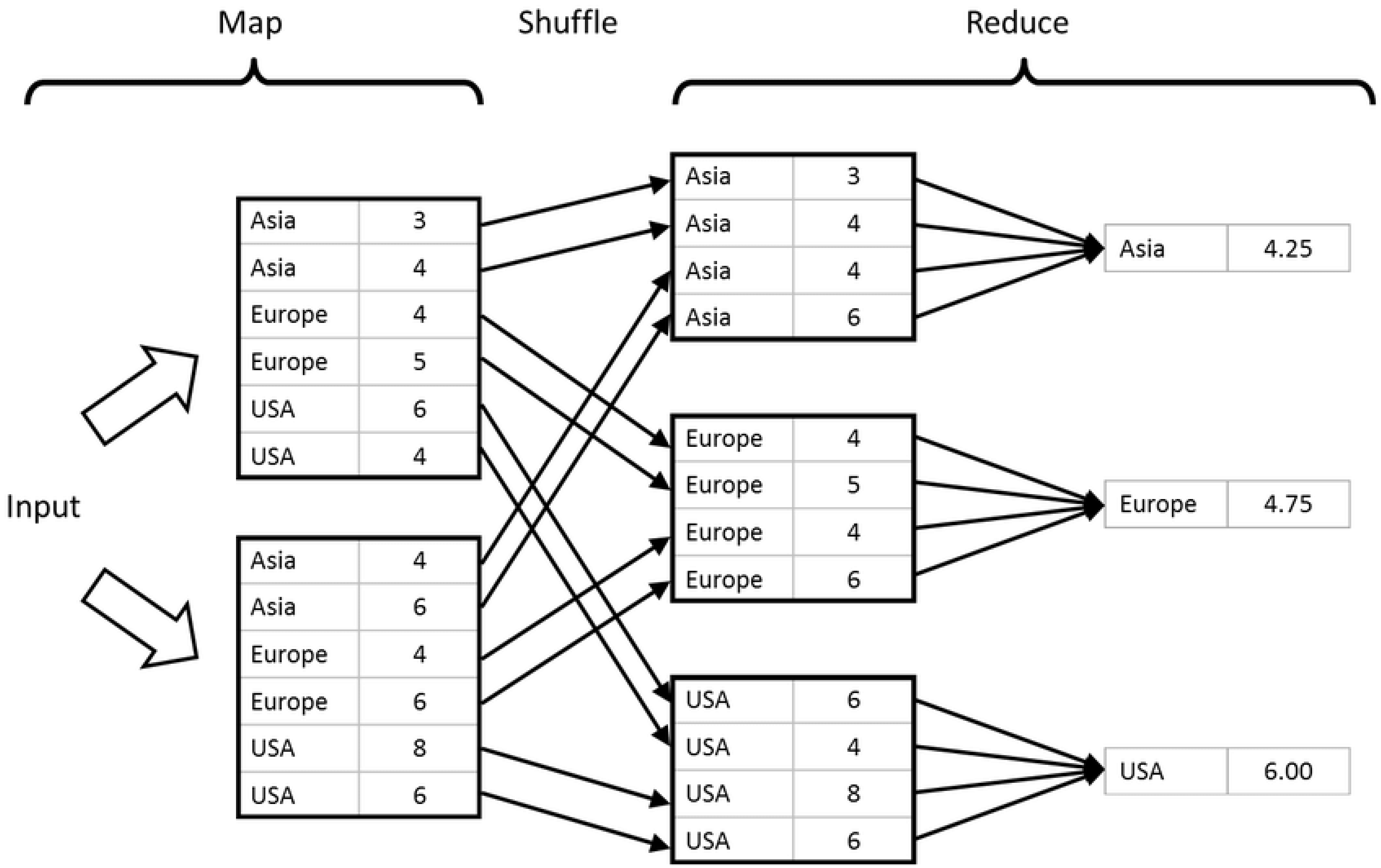


Map-Reduce

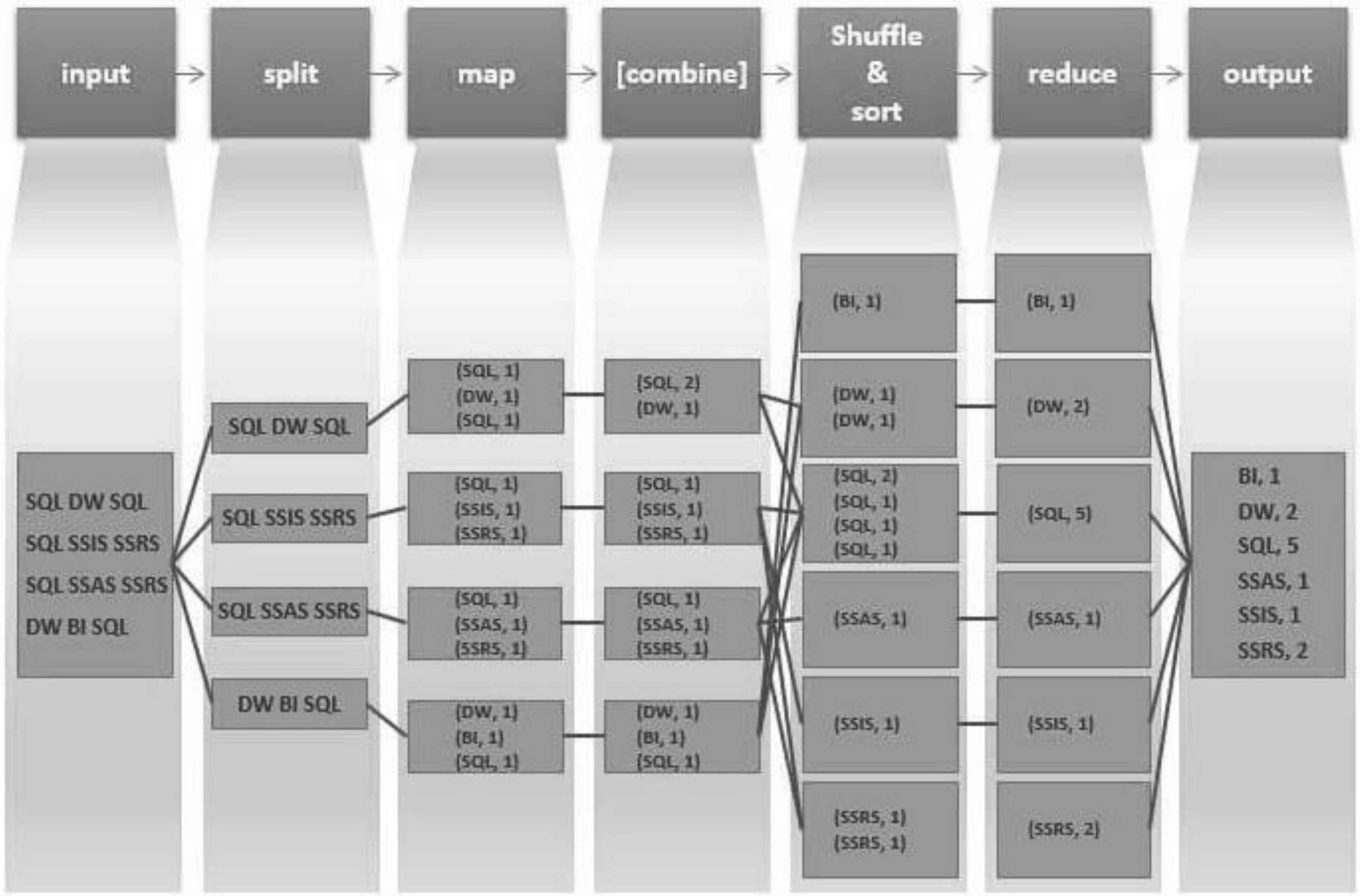


Input
Split
Map Phase
Shuffle and Sort
Reduce Phase
A B R
C C R
A C B
A B R
C C R
A C B
A, 1
B, 1
R, 1
C, 1
C, 1
R, 1
A, 1
C, 1
B, 1
A, 1
A, 1
B, 1
B, 1
C, 1
C, 1
C, 1
R, 1
R, 1
A, 2
B, 2
C, 3
R, 2

Map-Reduce


Map
Shuffle
Reduce
Input
Asia 3
Asia 4
Europe 4
Europe 5
USA 6
USA 4
Asia 4
Asia 6
Europe 4
Europe 6
USA 8
USA 6
Asia 3
Asia 4
Asia 4
Asia 6
Europe 4
Europe 5
Europe 4
Europe 6
USA 6
USA 4
USA 8
USA 6
Asia 4.25
Europe 4.75
USA 6.00

Map-Reduce


input
split
map
[combine]
Shuffle & sort
reduce
output
SQL DW SQL
SQL SSIS SSRS
SQL SSAS SSRS
DW BI SQL
(SQL, 1) (DW, 1) (SQL, 1)
(SQL, 1) (SSIS, 1) (SSRS, 1)
(SQL, 1) (SSAS, 1) (SSRS, 1)
(DW, 1) (BI, 1) (SQL, 1)
(SQL, 2) (DW, 1)
(BI, 1)
(DW, 1) (DW, 1)
(SQL, 2) (SQL, 1) (SQL, 1) (SQL, 1)
(SSAS, 1)
(SSIS, 1)
(SSRS, 1) (SSRS, 1)
(DW, 2)
(SQL, 5)
(SSRS, 2)
BI, 1
DW, 2
SQL, 5
SSAS, 1
SSIS, 1
SSRS, 2

Tools

Tools

- **No-SQL**
  - Cassandra
  - MongoDB
- **Blob Storage**
  - AWS S3
- **Load balancers**
  - Nginix
  - HAProxy
- **Cache**
  - Memcached
  - Redis
- **Config management**
  - Zookeeper

- **Queue publisher-subscriber**
  - Kafka
- **Search**
  - Elastic Search
  - Apache Solr
- **MapReduce**
  - Hadoop
  - Spark

No-SQL Cassandra: column based

Cassandra

- column/tab based (goes well with the historical RDBMS)
- **A**vailability-Partition
- support HiveQL (SQL like syntax)
- Supports sharding
- Preferable option when # of writes > # of reads
- Great for both key-value and time series date
- Can provide both eventual and strong consistency

No-SQL: Redis key-value

Redis

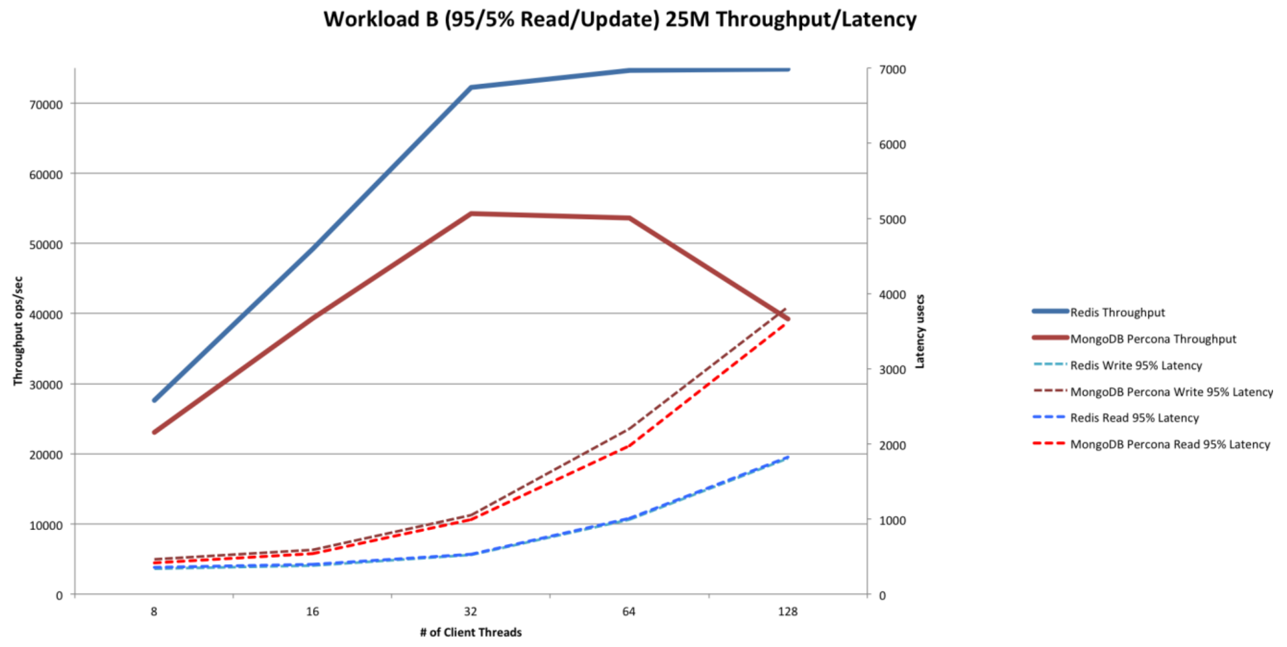
- key-value based
- **C**onsistency-Partition
- great for rapid changing data

Load balancing: nginix vs HAProxy

1000+ connections

Cache

- Memcached vs Redis vs Hazelcast



https://dzone.com/articles/comparing-in-memory-databases-redis-vs-mongodb

Search:

ElasticSearch

Solr

Blob Storage:

Amazon S3

S3 vc EC2

Amazon S3

Zookeeper

Centralized configuration management

- Distributed locking
- Leader election
- Scales great for reads
- Scales worse for writes

Apache
Zookeeper

Messaging

Fault-tolerant highly available queue for publisher-subscriber

- Can deliver message exactly once
- Keep messages ordered inside the partition

Queues

Apache ActiveMQ, Amazon SQS, RabbitMQ

Pub-Sub

Apache Kafka , Google Cloud Pub/Sub

Map-Reduce

- Hadoop
- Spark: in-memory

5+ Best Distributed Systems Interview Questions & Answers
https://www.algrim.co/408-distributed-systems-interview-questions

https://gist.github.com/vasanthk/485d1c25737e8e72759f

https://dzone.com/articles/top-20-system-design-interview-questions-for-java

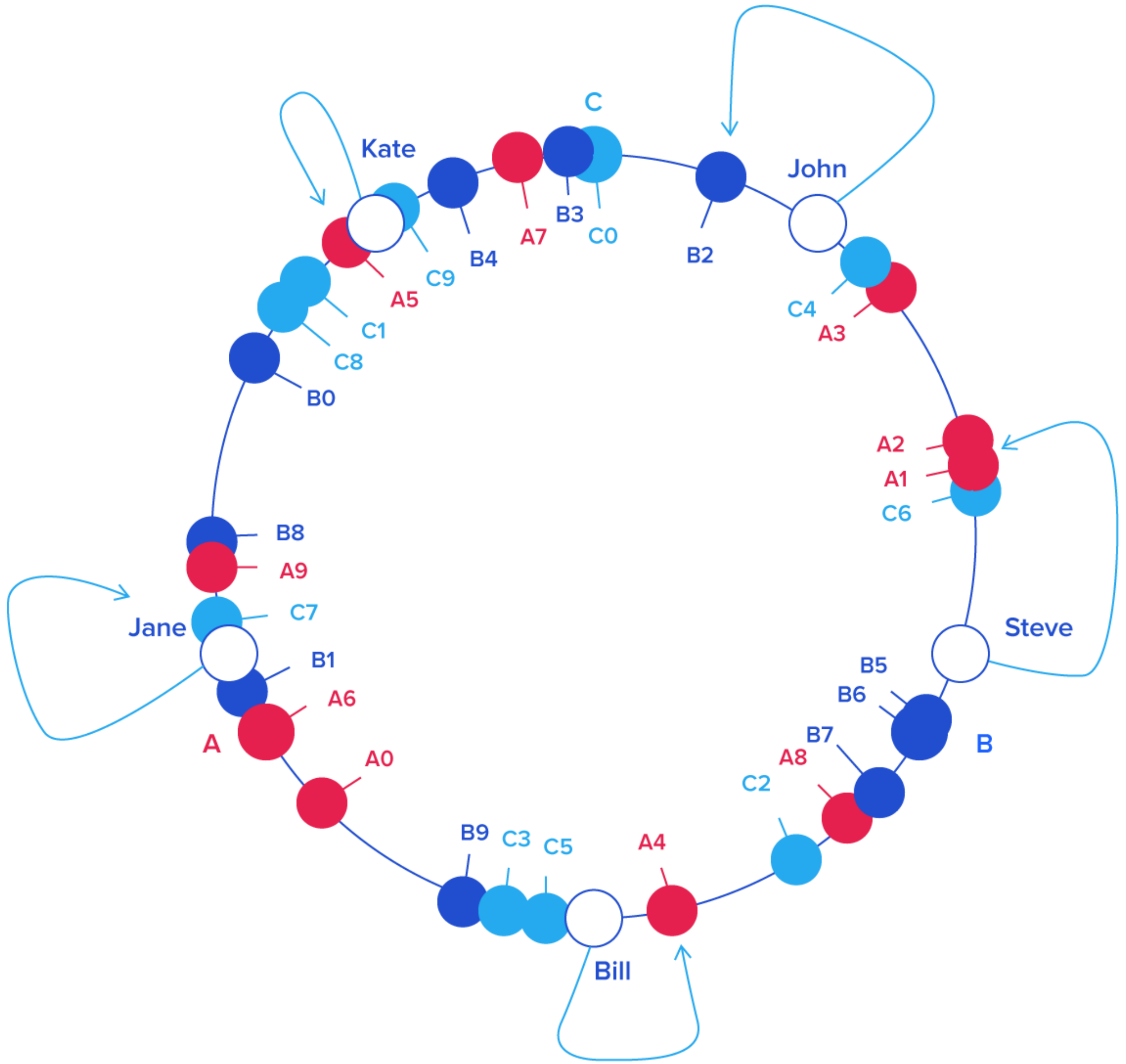
https://github.com/donnemartin/system-design-primer/blob/master/README.md

More ..

Consistent hashing

special kind of hashing: when a hash table is resized, only K/n keys need to be remapped on average

- K is the number of keys
- n is the number of slots.

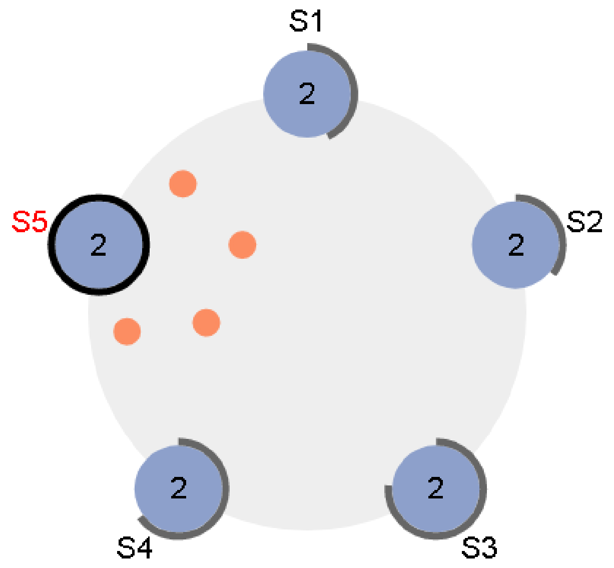


https://www.toptal.com/big-data/consistent-hashing

Consensus Solving

Leader election
Log replication

Paxos vs RAFT



http://thesecretlivesofdata.com/raft/

https://www.youtube.com/watch?v=IE3r12vBx6I

References

5+ Best Distributed Systems Interview Questions & Answers
https://www.algrim.co/408-distributed-systems-interview-questions

https://gist.github.com/vasanthk/485d1c25737e8e72759f

https://dzone.com/articles/top-20-system-design-interview-questions-for-java

https://github.com/donnemartin/system-design-primer/blob/master/README.md