Distributed systems concepts

- Scaling: Vertical vs Horizontal
- CAP Theorem
- ACID vs BASE
- Partitioning = Data Sharding
- Consistent Hashing

- Optimistic vs Pessimistic locking
- Strong vs Eventual consistency

- Relational SQL vs No-SQL
- No-SQLs: Key-value, wide-column, document-based, graph-based

- Caching
- Load Balancing: L4 vs L7
- Redundancy and Replications

- https vs TLS
- TCP vs UDP
- CDN vs Edge
- REST vs SOAP
- Long-pooling vs WebSockets vs Server-Send Events
- DNS lookup

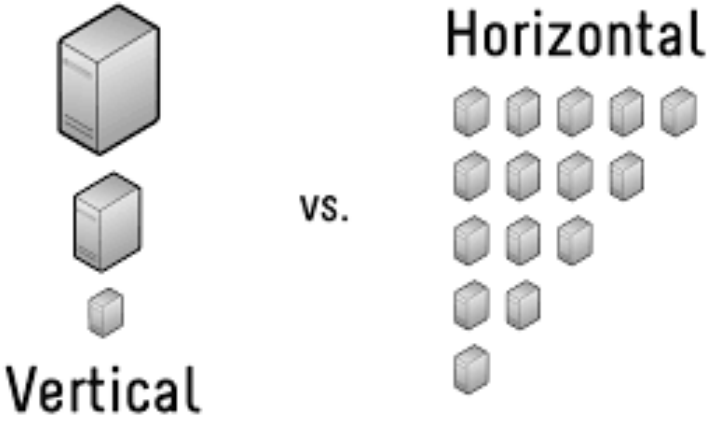
- Public key vs CA
- Publishers - Subscribers
- Map reduce

- http vs http2 vs WebSockets
- IPV4 vs IPV6

Scaling

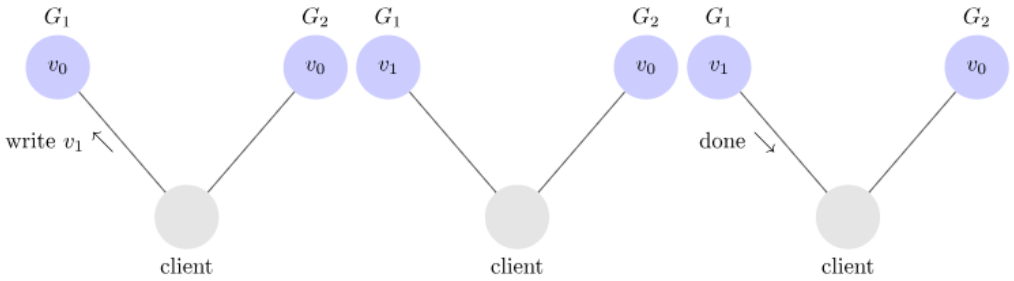
- Vertical: more RAM/CPU to existing host
  - Can be expensive
  - Limitations per max memory per single host
  - Less problem for distributed systems

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

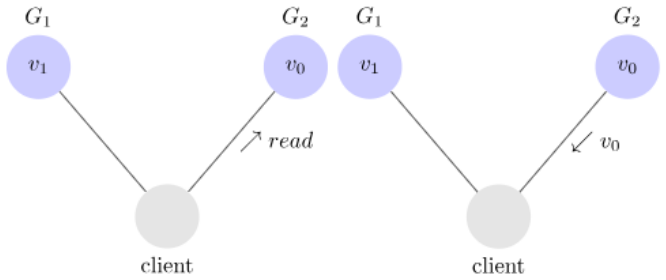


CAP

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.

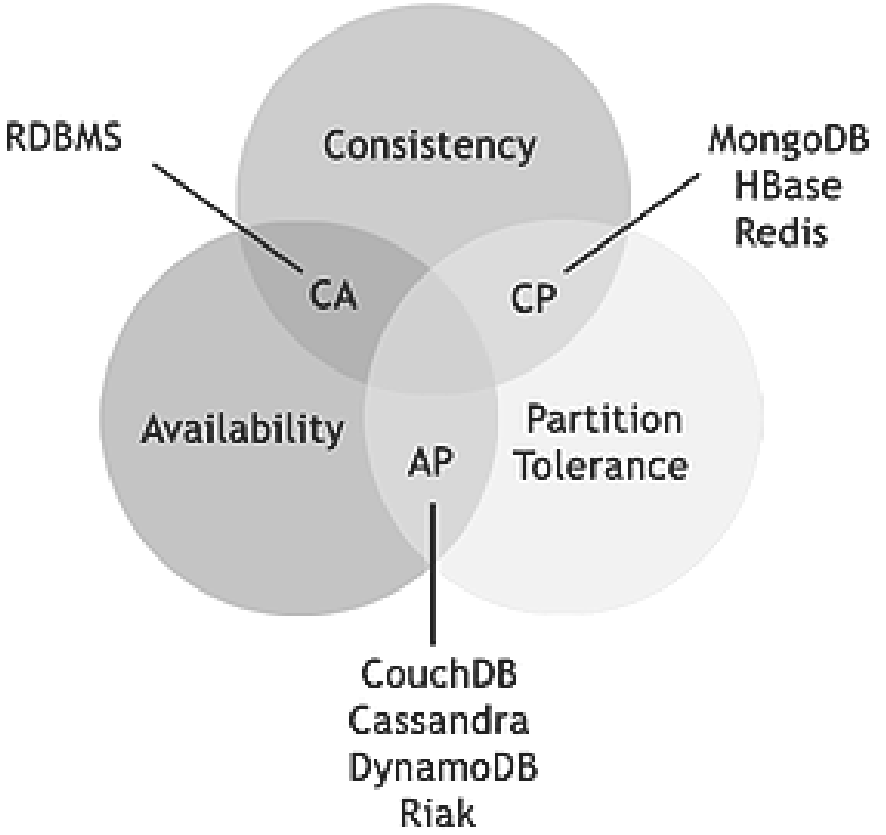
https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/

CAP

Consistency

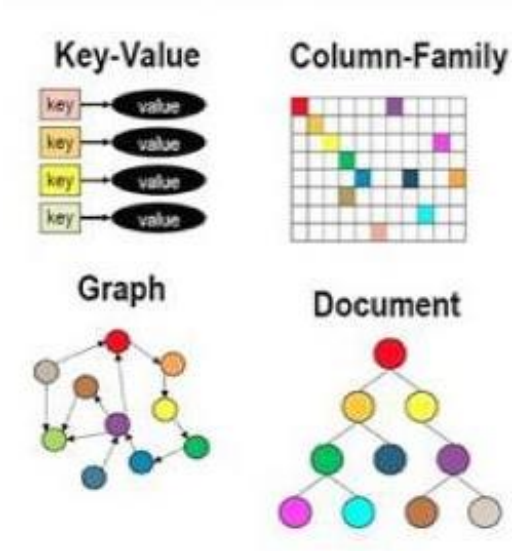
Availability

Partition Tolerance



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)		



No-SQL: Redis vs Cassandra

Cassandra
- column/tab based (goes well with the historical RDBMS)
- **A**vailability-Partition
- support HiveQL (SQL like syntax)

- Preferable option when # of writes > # of reads

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

ACID vs BASE

BASE

- **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

- **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

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

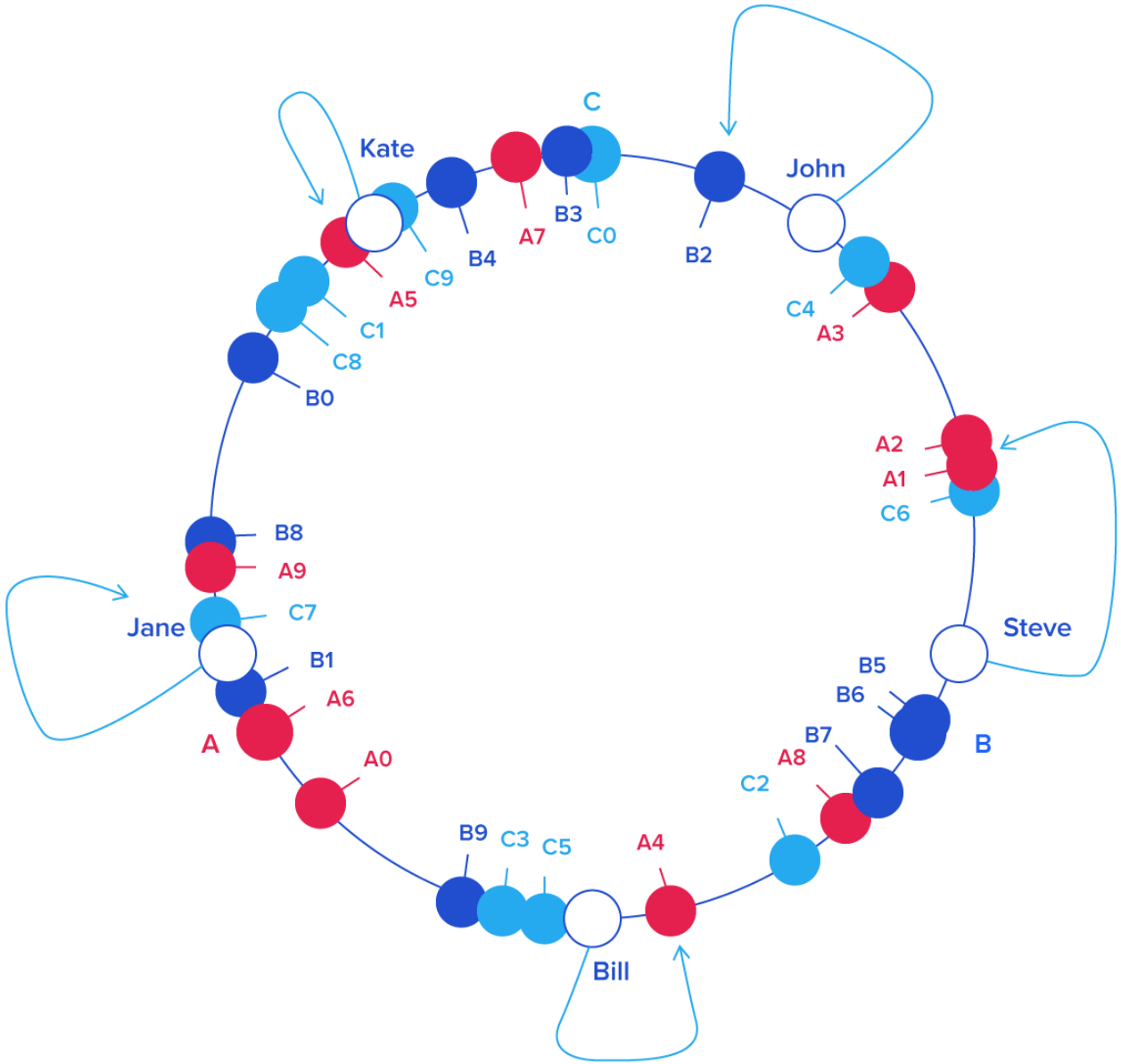
- data need to scale beyond a single storage node
- improve performance by reducing contention in a data store

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

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

Optimistic vs Pessimistic 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.

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

Optimistic vs Pessimistic 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

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**

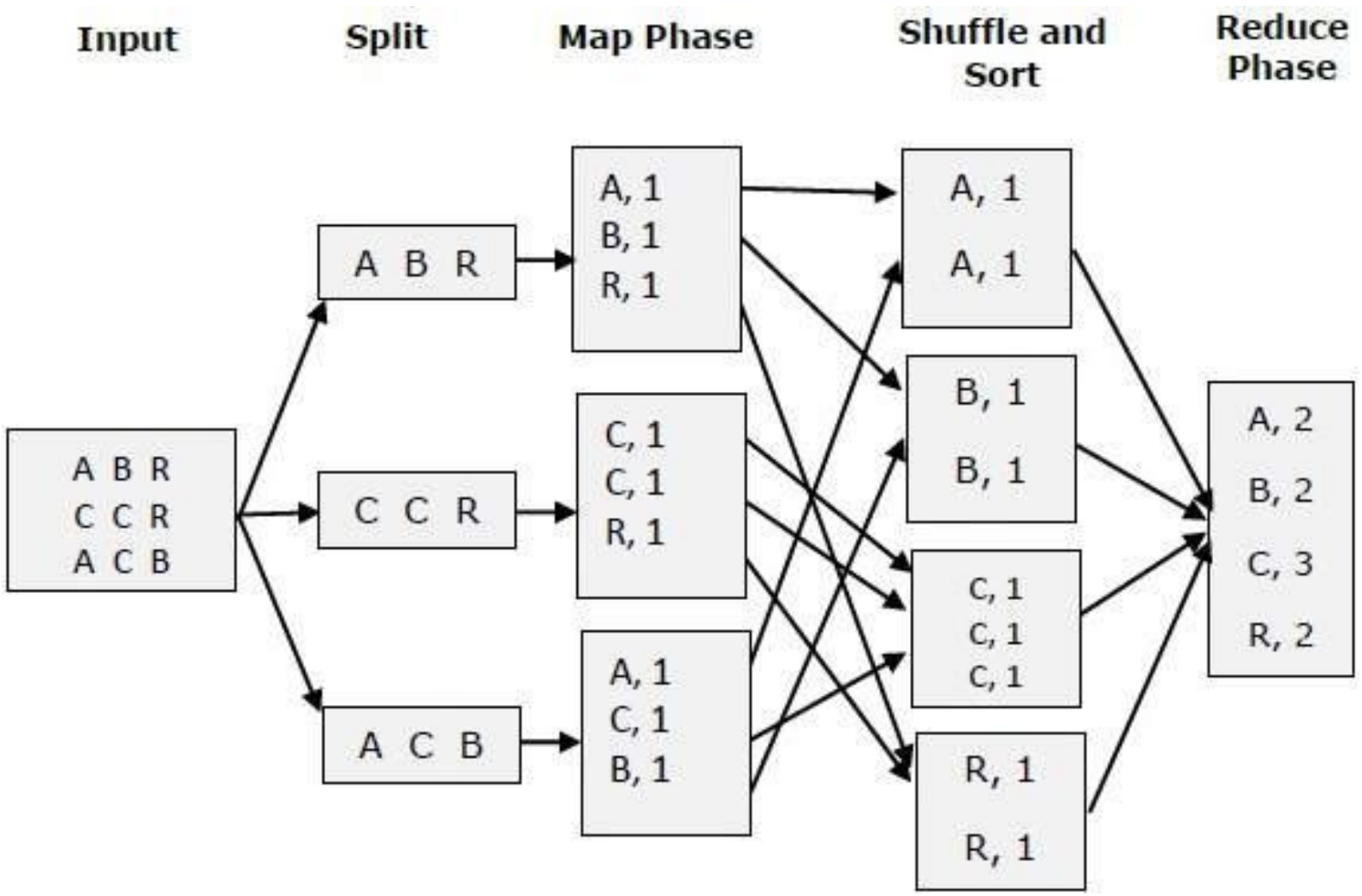
Load balancing: L4 vs L7

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

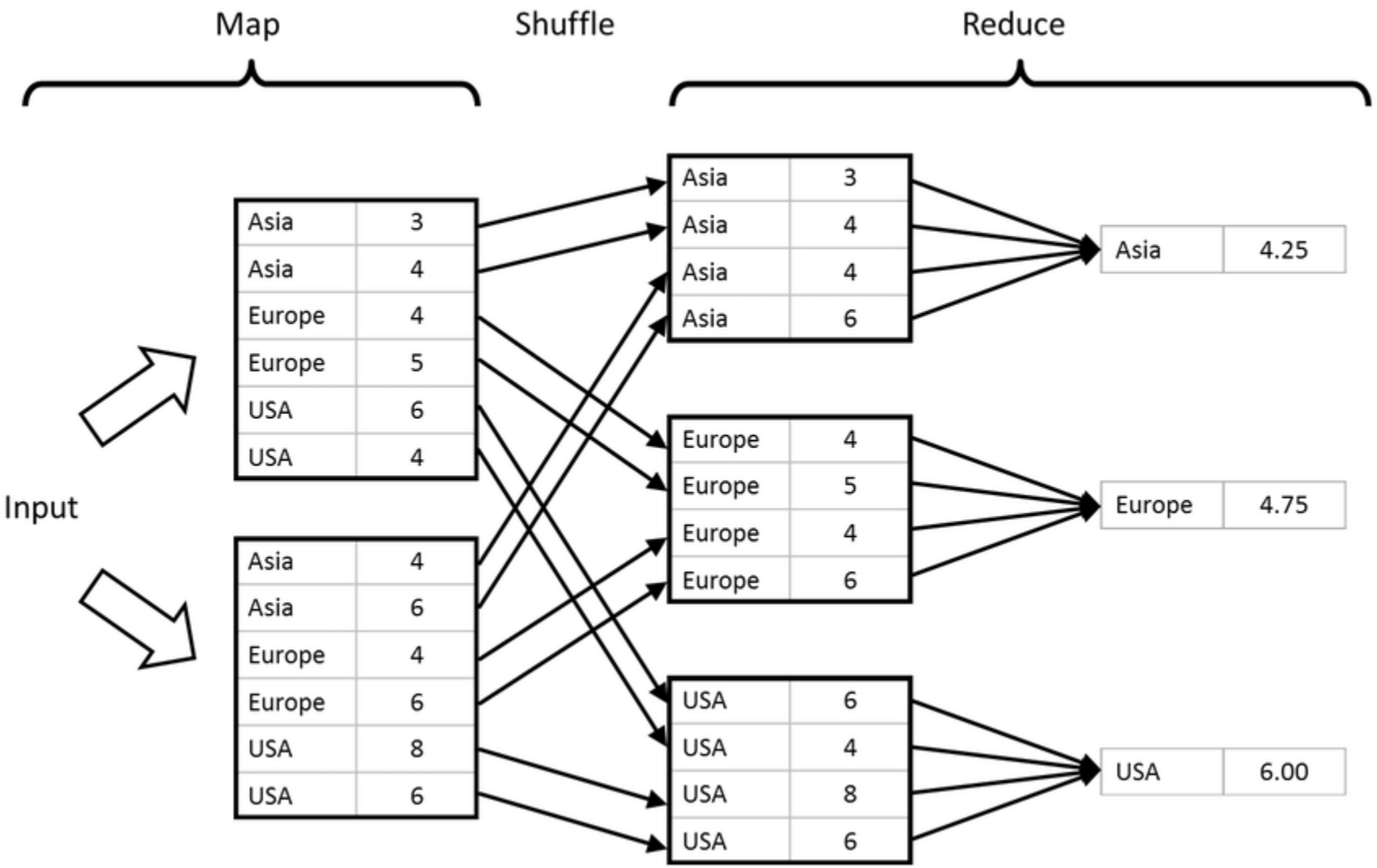
L7: adds content switching to load balancing. This allows routing decisions based on attributes like HTTP header, uniform resource identifier, SSL session ID and HTML form data

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

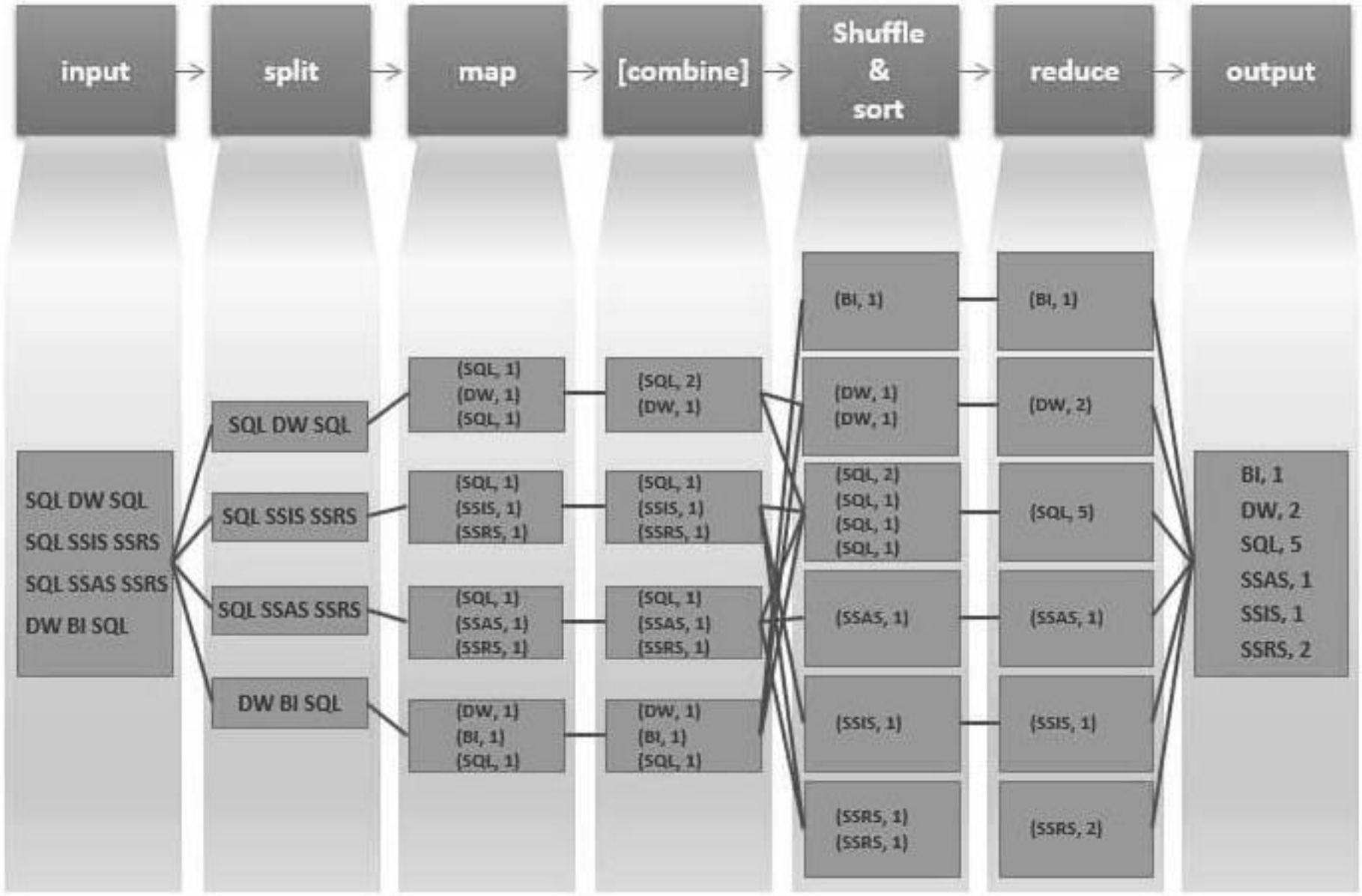
Map-Reduce


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 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)
(SQL, 1) (SSIS, 1) (SSRS, 1)
(SQL, 1) (SSAS, 1) (SSRS, 1)
(DW, 1) (BI, 1) (SQL, 1)
(BI, 1)
(DW, 1) (DW, 1)
(SQL, 2) (SQL, 1) (SQL, 1) (SQL, 1)
(SSAS, 1)
(SSIS, 1)
(SSRS, 1) (SSRS, 1)
(BI, 1)
(DW, 2)
(SQL, 5)
(SSAS, 1)
(SSIS, 1)
(SSRS, 2)
BI, 1
DW, 2
SQL, 5
SSAS, 1
SSIS, 1
SSRS, 2

xxx

- Kafka
- Ngninx
- HAProxy

Practice

- Designing URL Shortening service
- Designing Instagram
- Designing Twitter
- Designing Dropbox
- Designing Youtube
- Design a Parking Lot

The approach

- Requirements and Features: functional, non-functional

- APIs

- Durability, Availability, Performance
- Capacity estimation
- Scalability

- High level design, Entity diagram
- Data model
- Detailed design of each component
- Bottlenecks

- Security / Privacy
- Cost

Shortening URL

https://www.educative.io/collection/page/5668639101419520/5649050225344512/5668600916475904

https://www.quora.com/What-is-the-best-way-to-prepare-for-a-System-Design-interview-for-Amazon

https://www.youtube.com/user/tusharroy2525/videos?view=0&sort=dd&shelf_id=3

TODO

- ZooKeeper