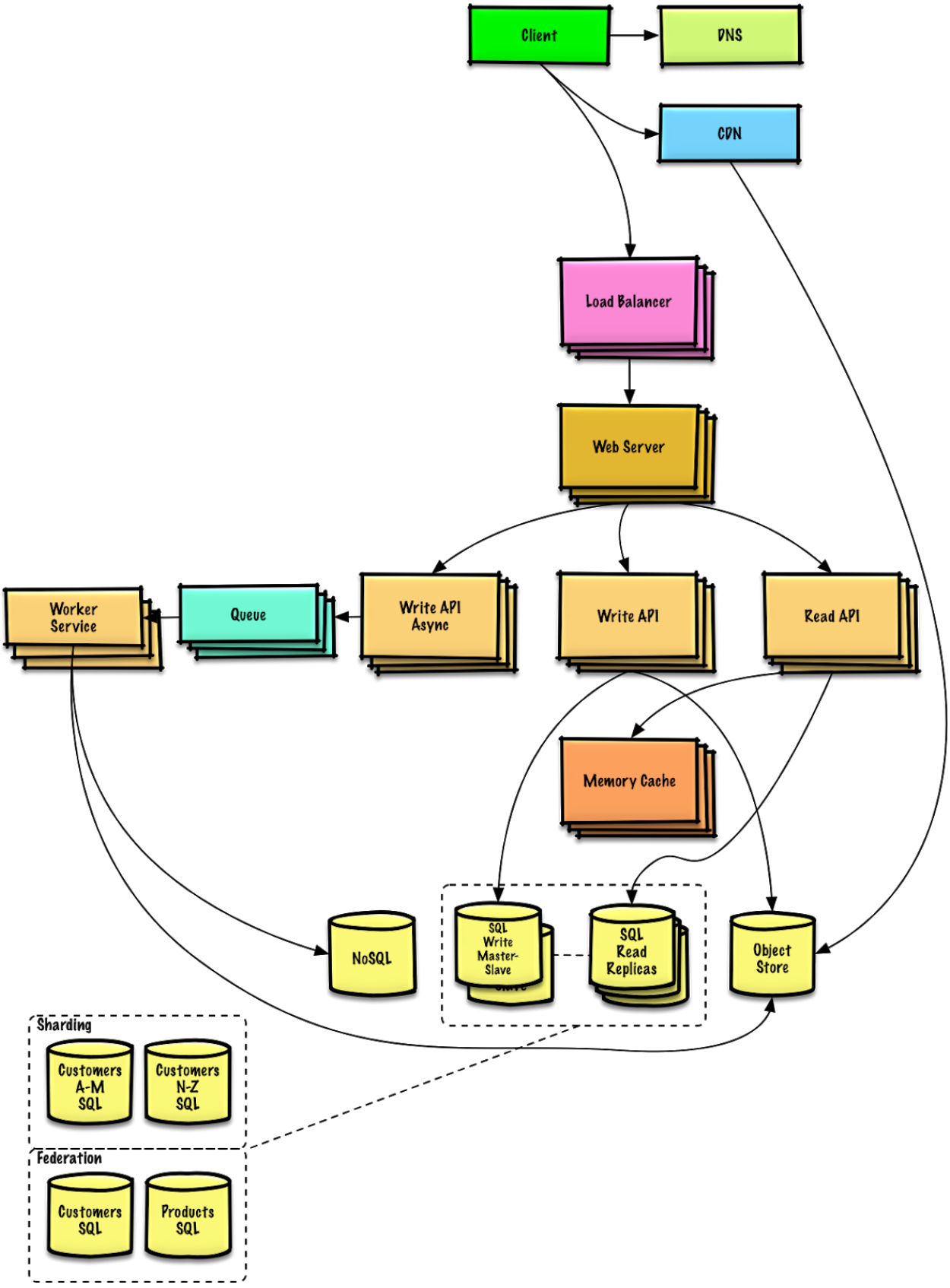
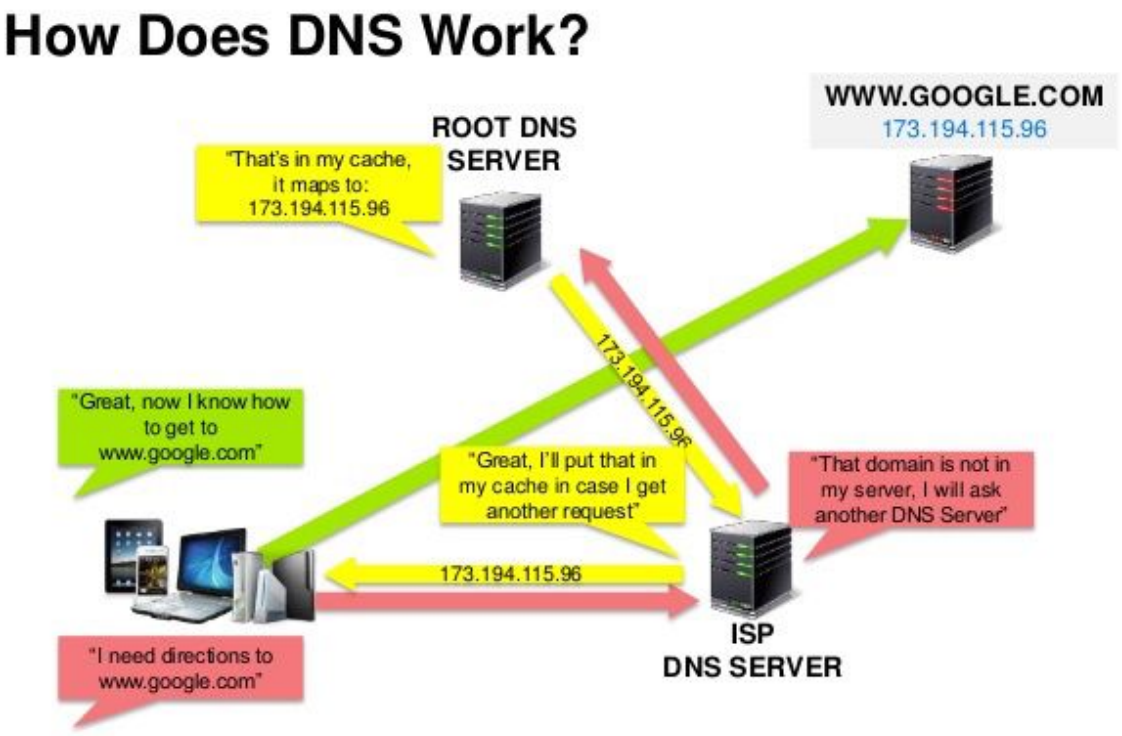
Overview


Client
DNS
CDN
Load Balancer
Web Server
Worker Service
Queue
Write API Async
Write API
Read API
Memory Cache
NoSQL
SQL Write Master-Slave
SQL Read Replicas
Object Store
Sharding
Customers A-M SQL
Customers N-Z SQL
Federation
Customers SQL
Products SQL

Client

DNS

How does DNS work?



Lower DNS servers cache mappings, and the DNS results also can be cached by browser or OS.

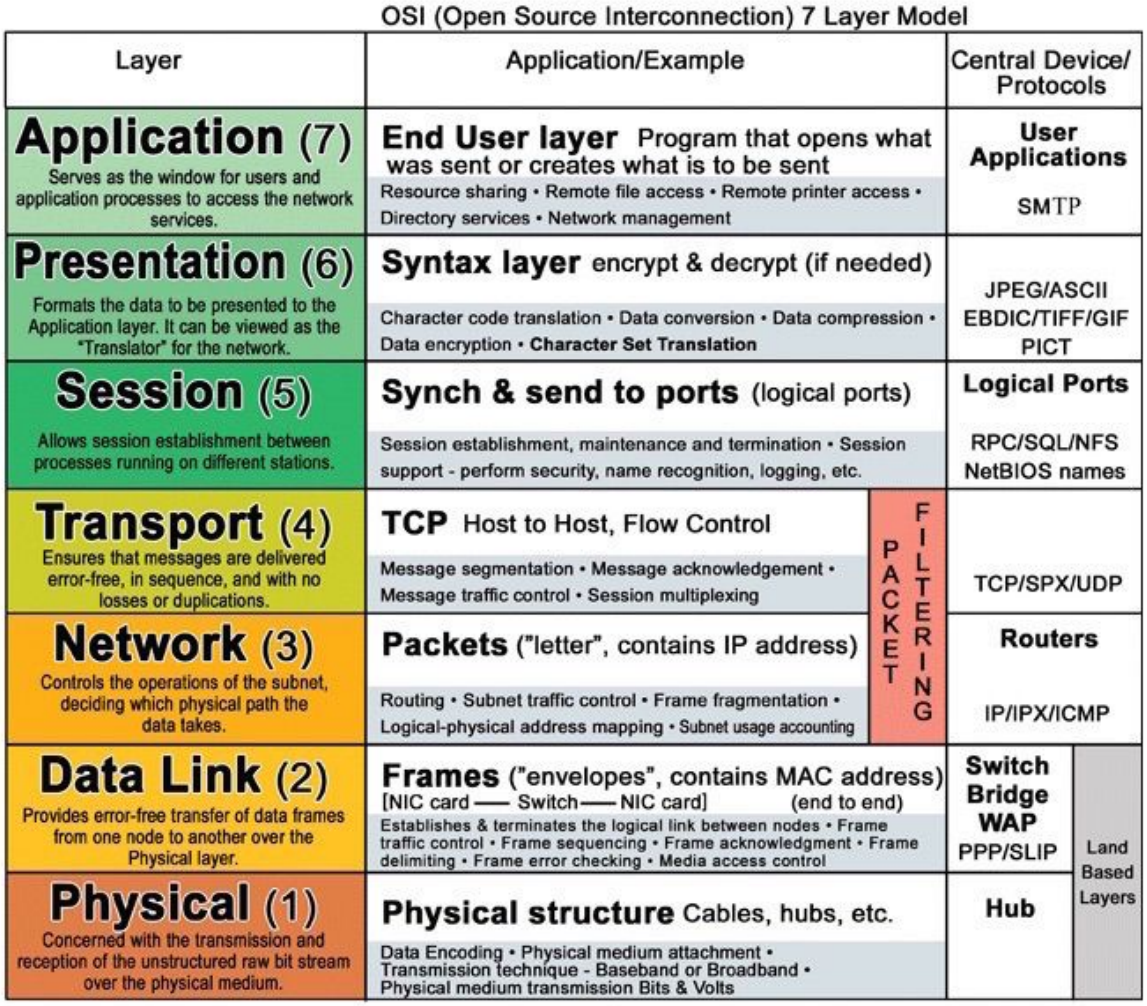
CDN (Content Delivery Networks)

Why CDN?

- CDN can improve performance in two ways
    - User get data from data center close to them.
    - CDN can help your servers answer some requests.

Generally, static files are served from CDN, some CDNs also support dynamic data.

Communication - HTTP, TCP, UDP and RPC



HTTP Request

A basic HTTP request consist of:

- A verb (or method)
- A endpoint (or resource)

Verb	Meaning	Idempotent	Safe	Cacheable
GET	Reads a resource	Yes	Yes	Yes

POST	Creates a resource or triggers a data-handling process	No	No	Only cacheable if response contains explicit freshness information
PUT	Fully updates (replaces) an existing resource or create a resource	Yes	No	No
PATCH	Partially updates a resources	No	No	Only cacheable if response contains explicit freshness information
DELETE	Deletes a resource	Yes	No	No

HTTP vs TCP

TCP works in the Transport layer while HTTP works in Application layer of TCP/IP model. This just means that HTTP works on top of TCP. TCP is in charge of setting up a reliable connection between two machines and HTTP uses this connection to transfer data between the server and the client. HTTP is used for transferring data while TCP is in charge of setting up a connection which should be used by HTTP in the communication process. Without TCP, HTTP cannot function.

TCP vs UDP

- TCP is connection-oriented, UDP is connectionless.
- TCP, the data is ordered; UDP, data is unordered.
- TCP is heavyweight, UDP is lightweight.

RPC vs REST

Both RPC and REST use HTTP protocol which is request/response protocol.

- REST is stateless.

Server

Load balance (See details in Load Balance topic)

Why Load balance?

- Preventing requests go to unhealth servers.
- Preventing overloading issue in server side.
- Providing **availability**. (Avoid single point failure)
- Providing horizontal **scalability**
    - Horizontal scaling vs Vertical scaling (see scalability session)
- Decrypt incoming requests && encrypt responses from server.

Load balance type

- Hardware (Good performance but expensive)
- Software (ELB)

How LB routes traffic (5 common ways)

- Round robin
- Weighted round robin
- Least connection
- Weighted least connection
- Random

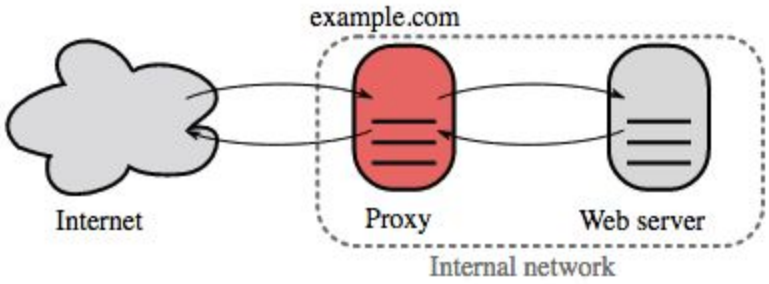
Load balancer models

(See details in Availability patterns session), it's common to set up multiple load balancers to protect against failure.

- Active-passive
- Active-active

- Disadvantage of load balance
    - Could be a performance bottleneck if doesn't have enough resource.
    - A single load balancer is a single point of failure, multiple load balancers increase complexity.
- Caches

Reverse proxy



What is reverse proxy?

- A reverse proxy is a web server that centralized internal servers and provides unified interface to public.

Why reverse proxy?

- Increased security - hide backend servers and limit number of connections per client.
- Increased **scalability** and flexibility - client only see the reverse proxy's IP
- SSL termination - Decrypt incoming requests && encrypt server responses(Notes, Load balance also offer this feature).
- Cache, it's also a good place to cache responses.

Reverse proxy vs Load balance

- Load balance is applied when you have multiple servers and these servers serving same function.
- Reverse proxy could be useful even with single server.

Disadvantage of reverse proxy.

- Increased complexity, same as load balance.

Database

Relational database

ACID properties:

- Atomicity - each transaction is all or nothing.
- Consistency
- Isolation

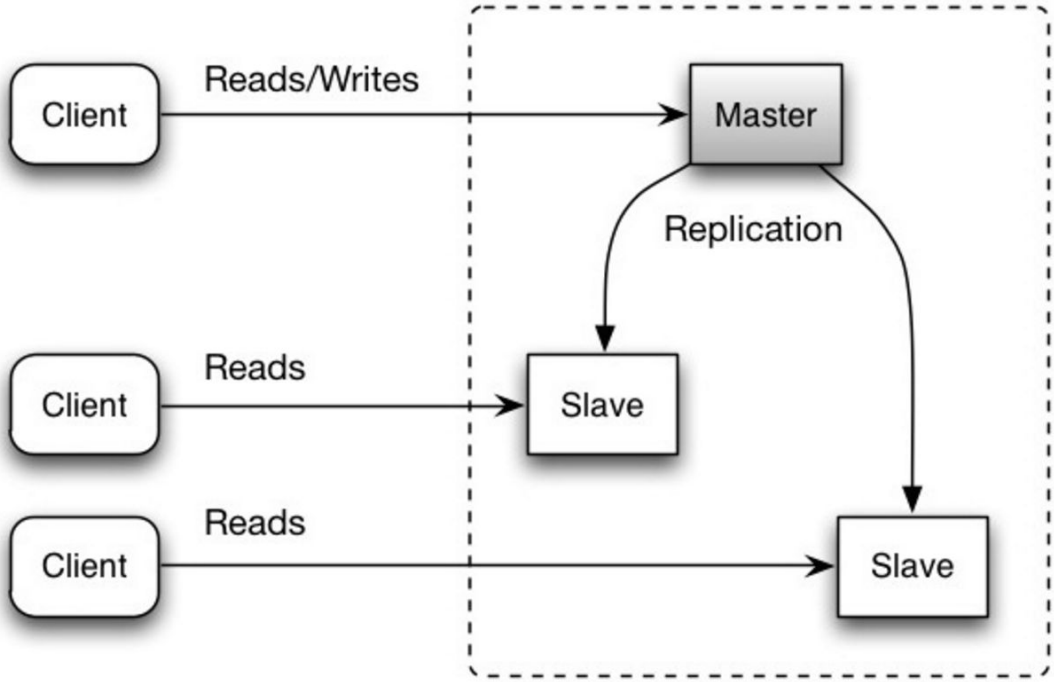
- Durability

BASE properties(For NoSQL)

- Basically available.
- Soft state.
- Eventual Consistency

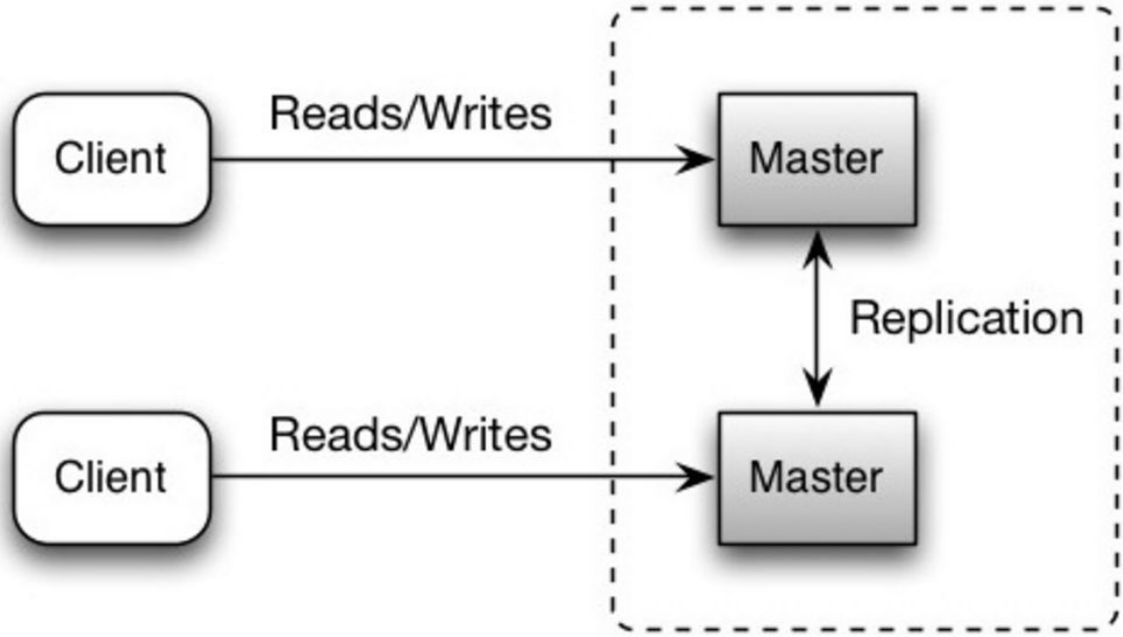
How to scala relational database (Replications)

Master-slave replication



- Master servers reads && writes
- Slaves server reads only
- If master down, system provide read only util a slave is promoted to master.
- Disadvantage:
    - Additional logic is required to promote slave to master.

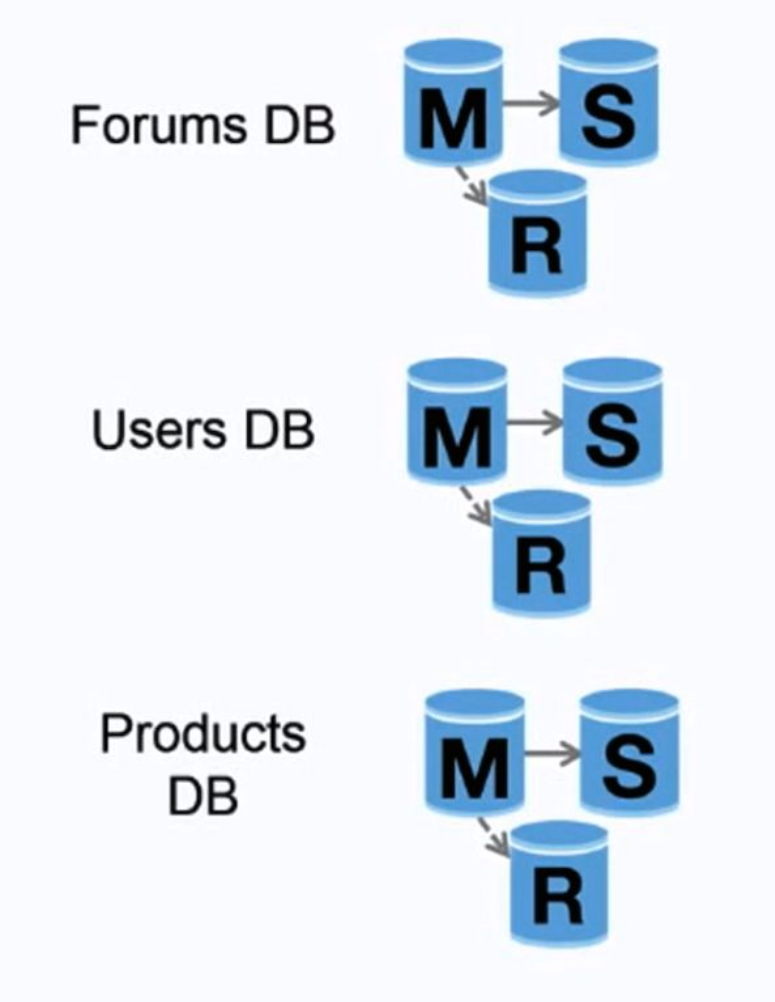
Master-master replication



  - Both masters can provide reads && writes.
  - Even one master goes down, the system can still provide both reads && writes.
  - Disadvantages:
    - A load balancer is required.
    - Loosely consistency
- Disadvantages of replication
  - Potential for loss of data.

Federation

  - Federation (or functional partitioning) splits up database by function.



Database Concurrency Control

Collisions

The collisions means there are multiple users try to modify same data at simultaneously, there are three ways that users can interfere with others.

- Dirty read.
- Non-repeatable read.
- Phantom read.

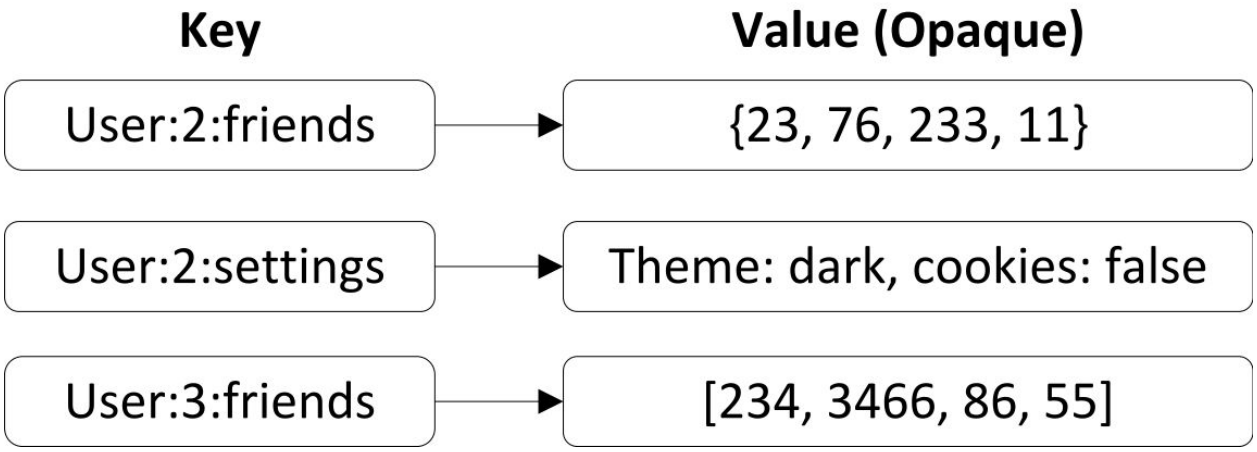
Locking strategies

- Pessimistic Locking
    - Place a lock until the transaction is finished.
    - A write lock preventing other users from read and write.

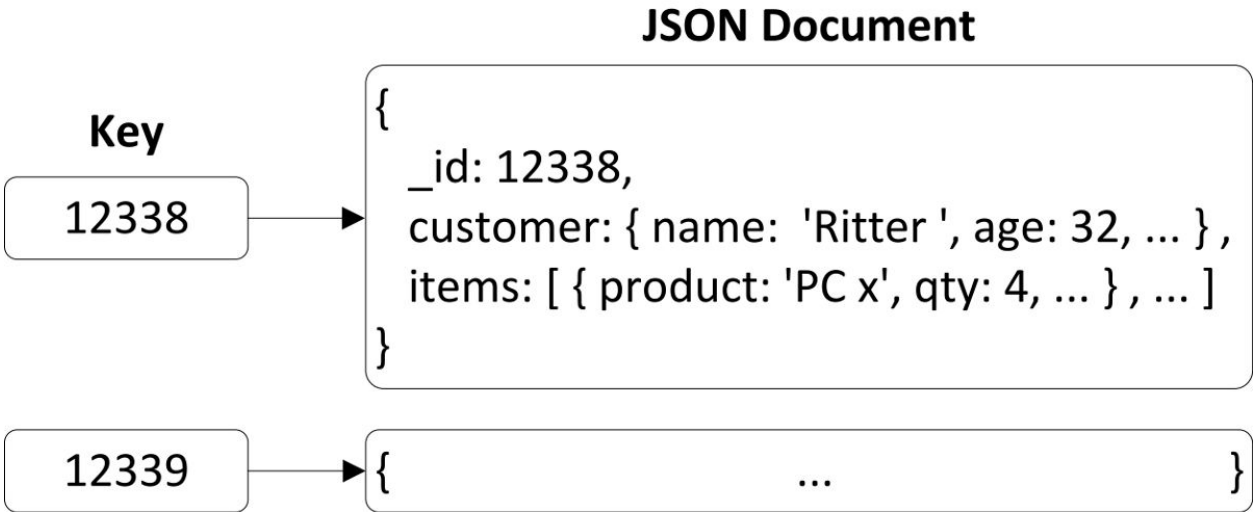
- Optimistic Locking
- Overly Optimistic Locking

NoSQL

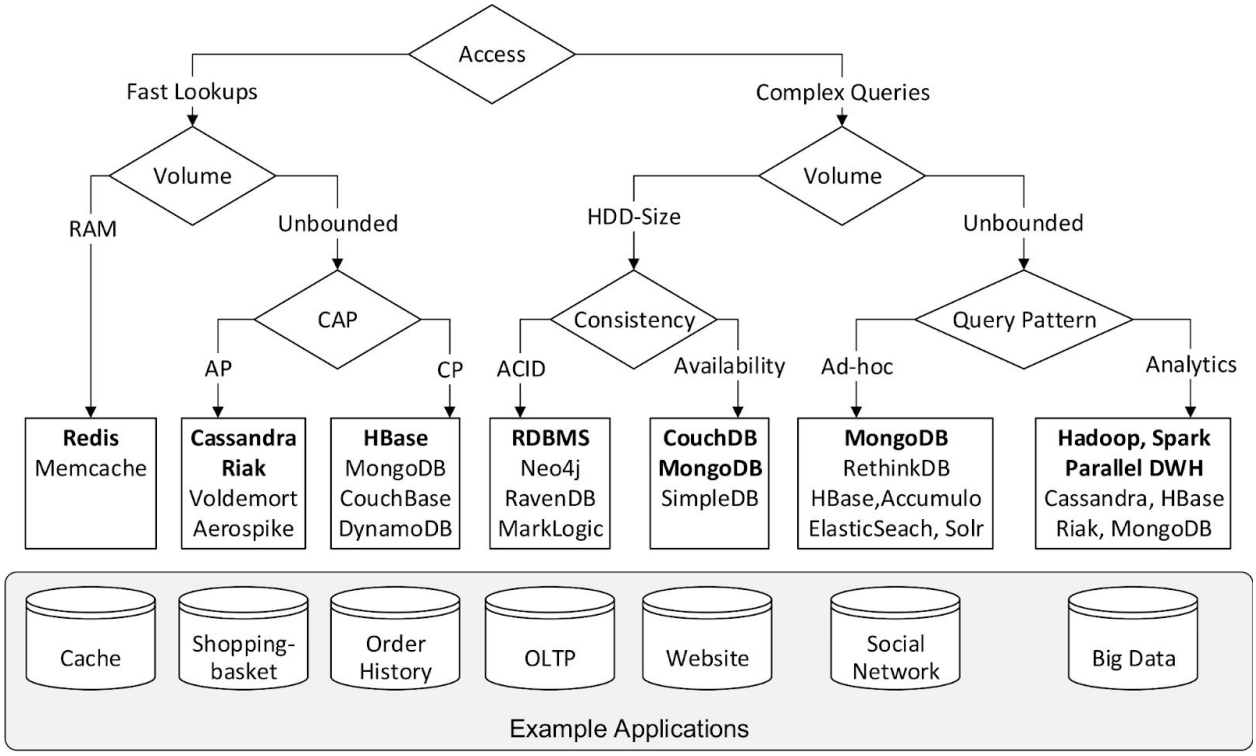
SQL family: MySQL, Oracle, Postgres
NoSQL family:



Key-Value Database



Wide-Column Database



- Key-Value
  - Radis, MemcacheDB
  - Advantages:
    - Simplicity
    - Easy to partition and query
    - Good for low latency and high throughput
  - Disadvantages:
    - Not good for complex query
- Wide-Column-Oriented
  - Canssandra, HBase
  - Search is fast, but insert and update are generally slower
- Document-Oriented
  - MongoDB, Couchbase
- Graph
  - Neo4J, OrientDB

NoSQL vs SQL

SQL

- Pre-defined schema
- Vertical scale (Scala-up)
- Replicable, easy to be replicated across multiple nodes.
- Shading (Can't be done on most SQL database, MySQL is able to do that)

- Good for complex query, transactions
- Lookup by index is very fast

NoSQL

- Dynamic schema, flexibility.
- Horizontal scala (Scala-out)
- Store many data(TB or PB)
- High throughput for IOPS

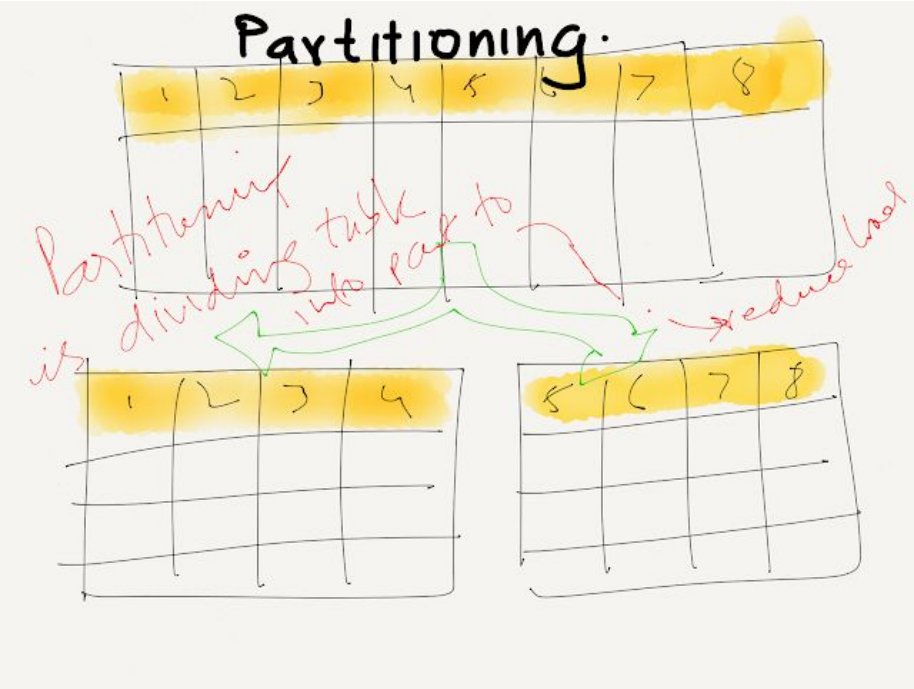
Benchmarks (QPS)

- MySQL/PostGres ~1K
- MongDB/Cassandra ~10K
- Redis/Memcached ~1M

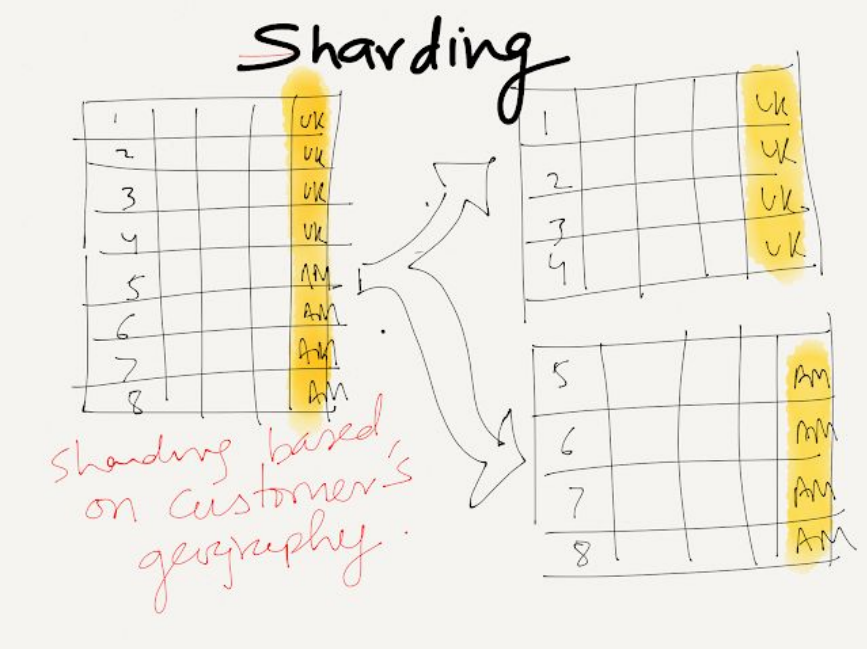
Sharding, Partitioning, Replication

Sharding vs Partitioning

- Partitioning, divide the logic entities into different physical entities for performance or availability.
    - There’re foreign keys.



- Sharding, is a horizontal partitioning, divide the data based on a shard key for scalability.
    - Shared-Nothing



Sharding

- Range-sharding
- Hash-sharding
    - Consistent hashing
- Entity-Group-sharding

Replication

- Simply copy all of the data to another location.

Shared-Nothing Architecture (SNA)

- It's a pattern used in distributed system.
- Each node shares no resource with other nodes.
- All information is available on at least two nodes.
- Example, MySQL custer, ~200M QPS

CAP Theorem

- Consistency
    - A read is guaranteed to return the most recent write
- Availability
    - A non-failing node will return a reasonable result
- Partition-Tolerance
    - The system will continue to work when network partition occur

In distributed system, you have two options here:

- CP

        - MongoDB
        - If you need atomic reads and writes, CP is applied.
        - Disadvantage:
            - The system may result timeout error.
- AP
    - Cassandra
    - Disadvantage:
        - Stale data

Schema Design

- SQL database focus and effort is around description of the entities, the queries and indexes are designed later.
- NoSQL have a "**query-first**" schema design, like HBase.

Key Points

Scalability

- Horizontal scaling(scaling out) vs Vertical scaling(scaling up)
- Horizontal scaling
    - Advantage:
        - More cost efficient (lower cost)
        - Higher availability
    - Disadvantage:
        - Servers should be stateless, shouldn't contain any user-related data likes sessions. (How to fix this issue? Sessions can be stored in a centralized database or **persistent cache**)
        - Downstream servers such as cache or database have to handle more simultaneous connections.(compare with scala up)

Availability

There are two patterns to support availability, fail-over and replication

Fail-Over

- Active-Passive
    - The heartbeats are sent between active and passive, once the heartbeat is interrupted, passive will take over the active and resume service.
    - Like master-slave
- Active-Active
    - Like master-master

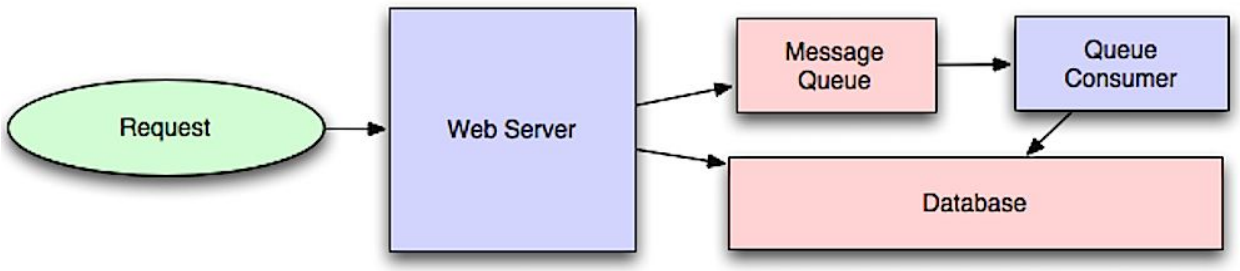
Cache

- Client cache (OS, browser, etc)
- CDN
- Server side
- Database

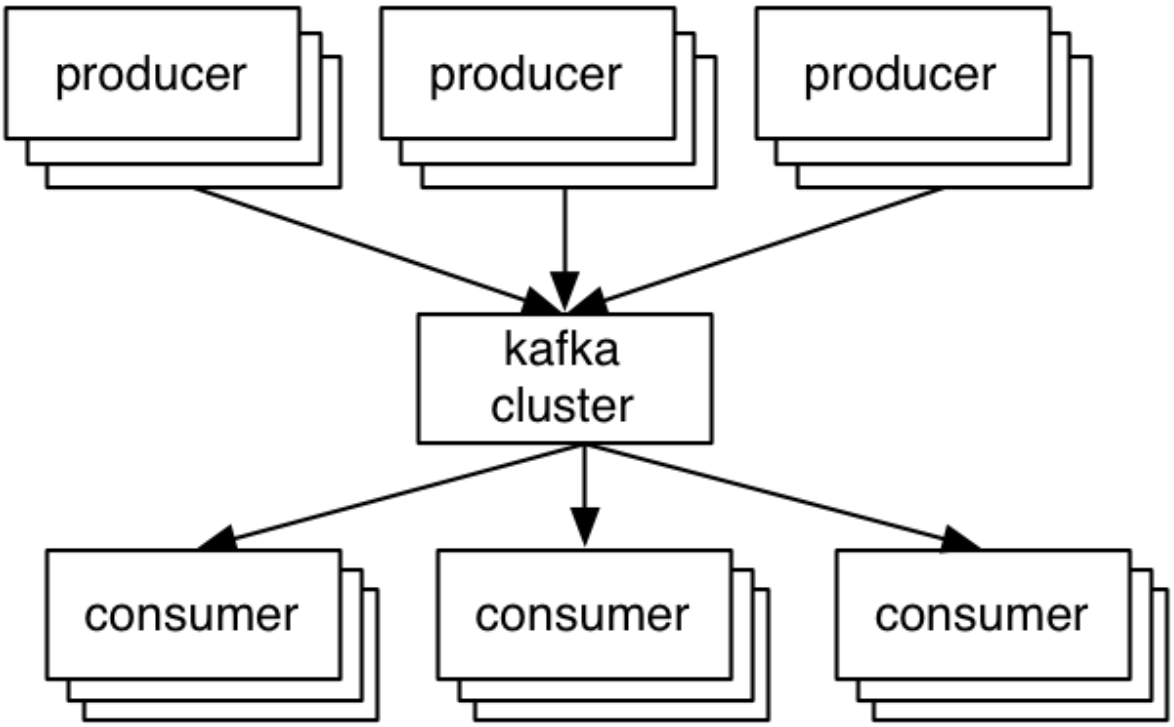
Cache System Design Pattern

- Write through cache
    - Flow, request -> write(cache) -> write(database) -> both two writes succeed -> return confirmation to client.
    - Advantages:
        - Good for write and re-read in quickly.
    - Disadvantages:
        - High write latency.
- Write around cache
    - Flow, request -> write (database) -> return confirmation to client.
    - Advantages:
        - Low write latency.
    - Disadvantages:
        - High re-read latency.
- Write back cache
    - Flow, request -> write(cache) -> return confirmation to client.
      -> Async(MQ) to database.
    - Advantages:
        - Low quick re-read latency
        - Low write latency.
    - Disadvantages:
        - Data lost (consistency).

Asynchronism



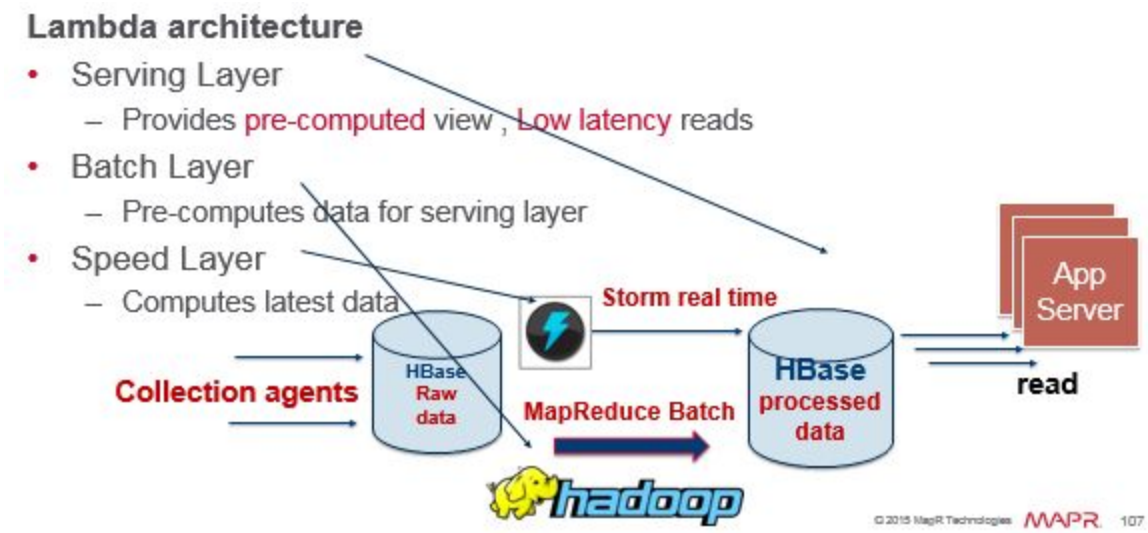
Message Queue



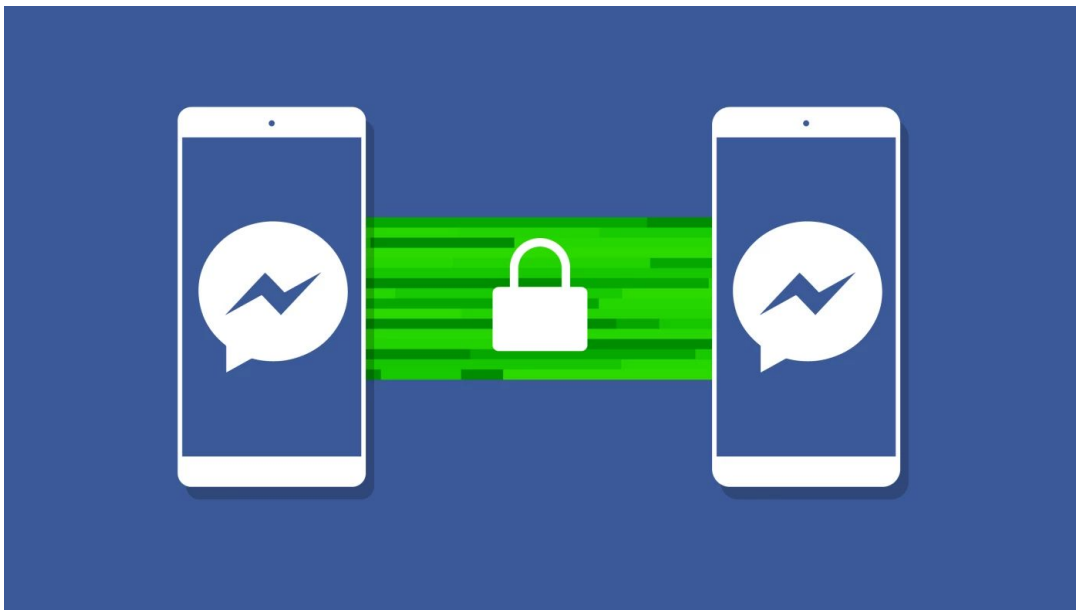
- RabbitMQ, Kfaka
- Why MQ?

Lambda Architecture

The Lambda architecture solves the problem of computing **arbitrary functions** on **arbitrary data** in real time by decomposing the problem into three layers: the batch layer, the serving layer, and the speed layer.



Security - Message Encryption (end-to-end encryption)

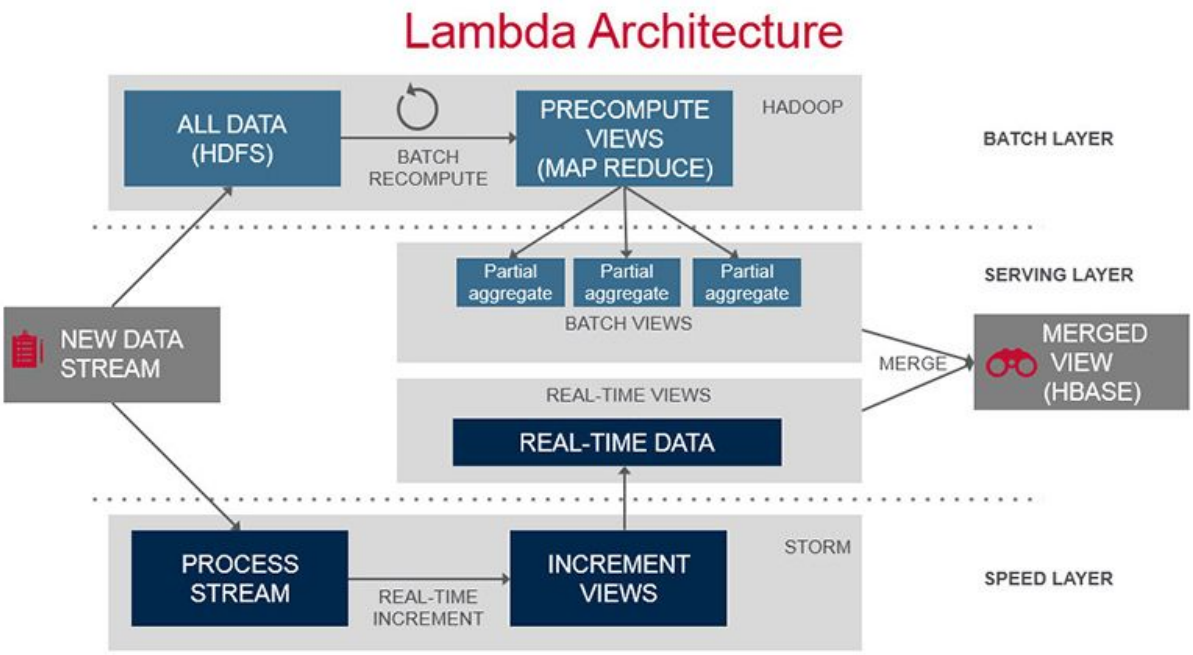


- RSA
- Twofish
- AES

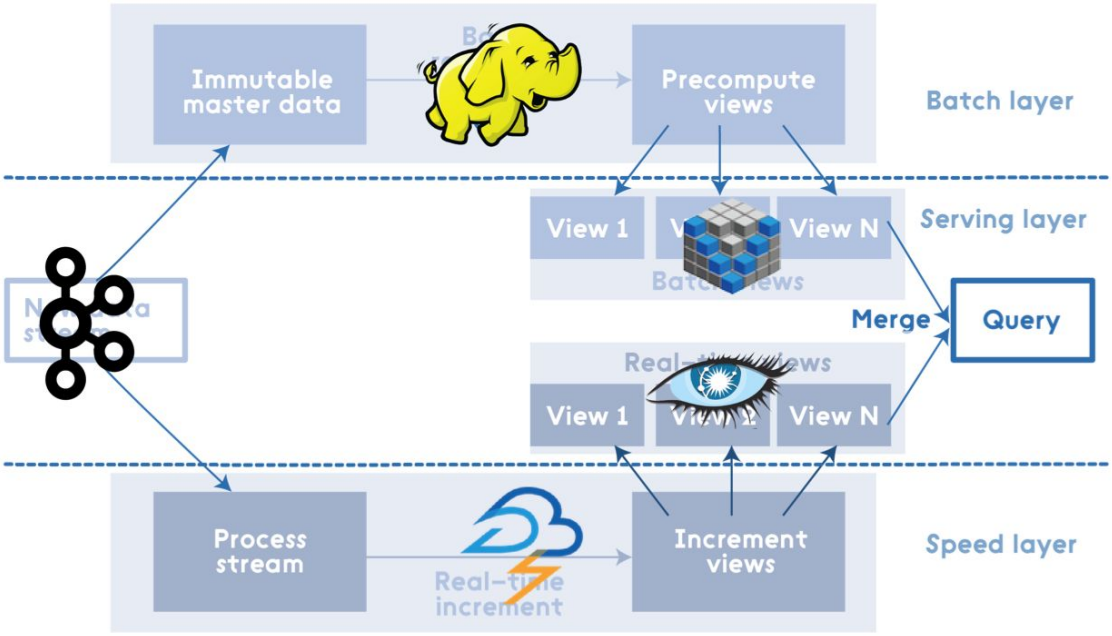
Real-time Data Architectures

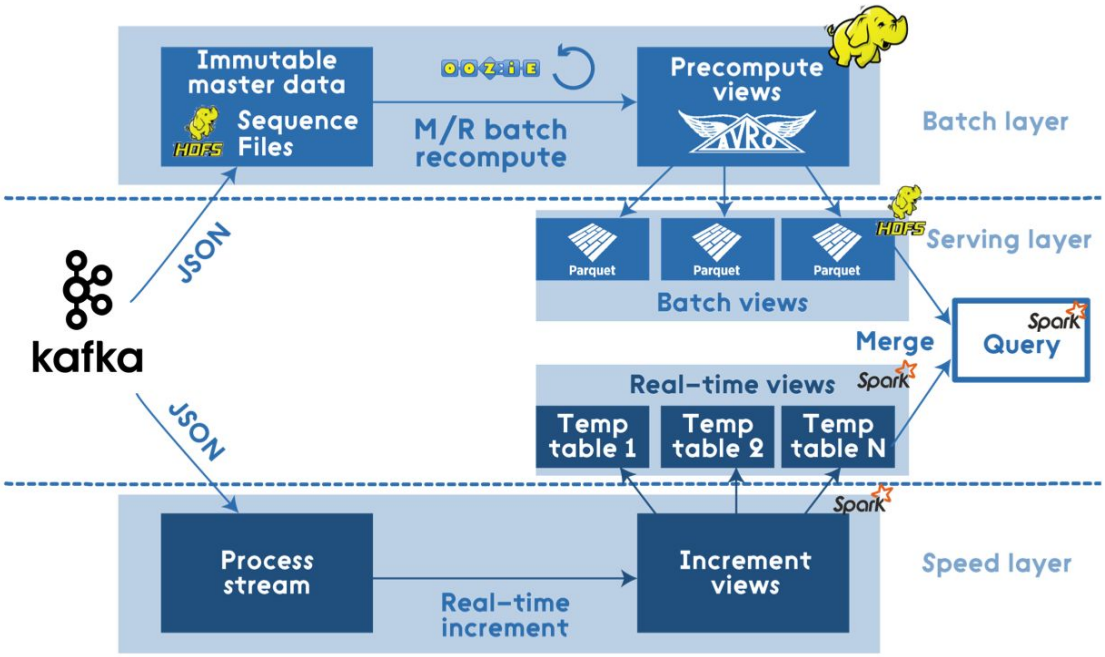
- Lambda Architecture
- Kappa Architecture

Lambda Architecture vs Kappa Architecture



Lambda

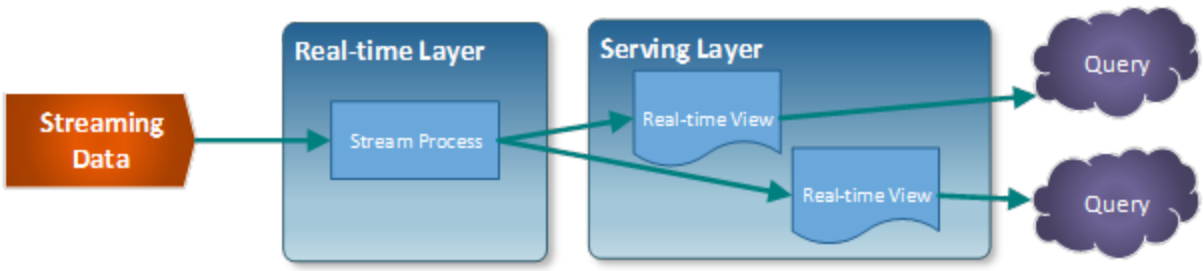


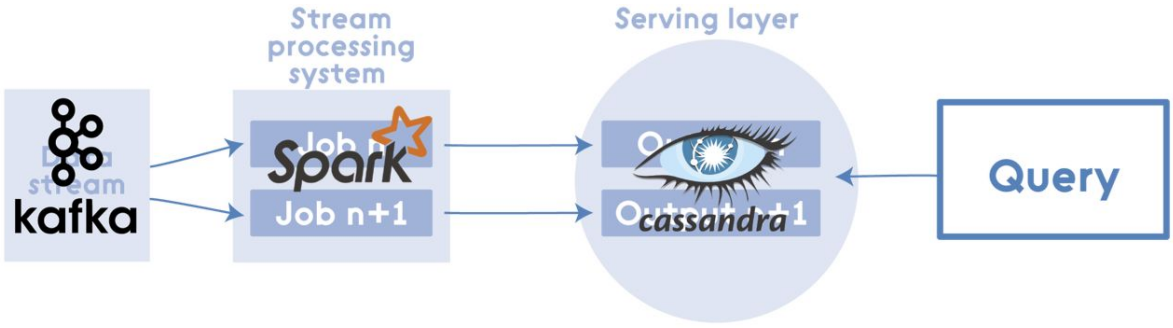


Three Layers

- Batch layer
  - Store all raw data with fault-tolerant distributed storage.
  - Batch view - More complex and expensive rules applied.
- Speed layer
  - Has low latency since it deals with real-time data ONLY, incremental update.
  - No complete data is stored.
- Serving layer
  - Both batch view and real-time view are forward to serving layer.

- Pros
- Cons
  - Batch view and real-time view have duplicated logics.

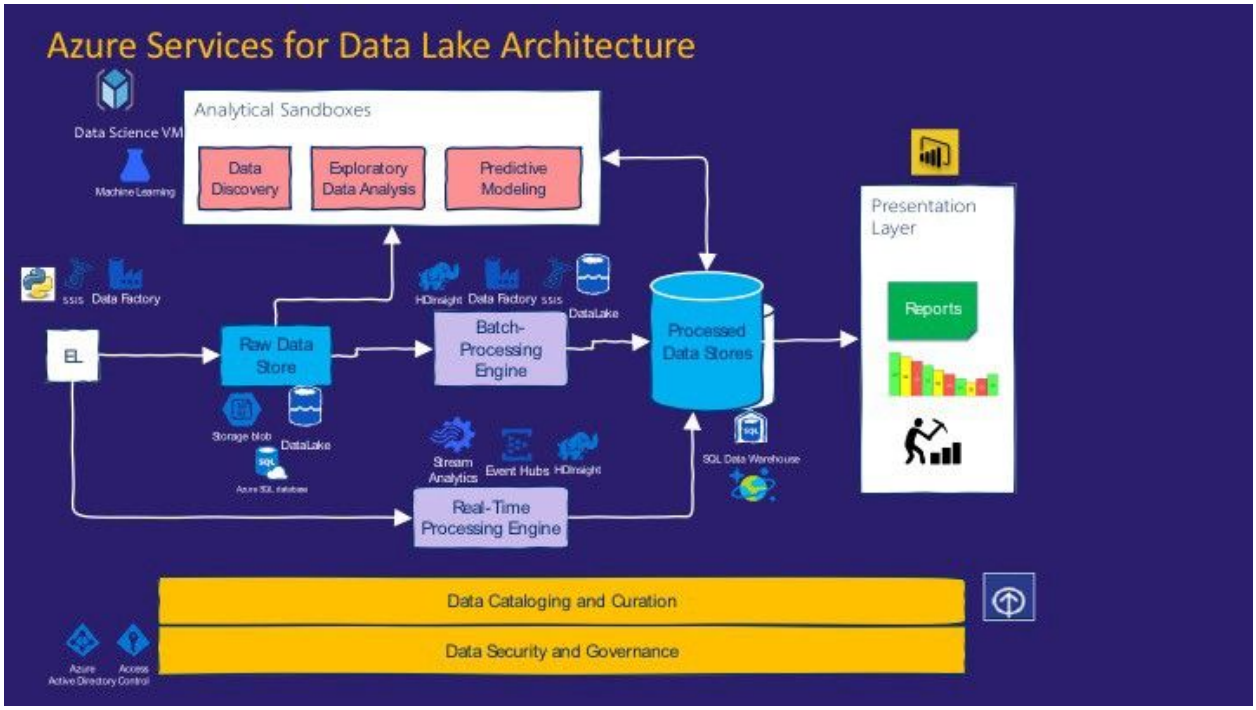




Kappa

More details:
https://dzone.com/articles/lambda-architecture-with-apache-spark
https://towardsdatascience.com/a-brief-introduction-to-two-data-processing-architectures-lambda-and-kappa-for-big-data-4f35c28005bb
https://www.talend.com/blog/2017/08/28/lambda-kappa-real-time-big-data-architectures/

Data Lake Architecture



https://medium.com/@rpradeepmenon/demystifying-data-lake-architecture-30cf4ac8aa07

General Steps

Functional Requirements

- Should we need consider "**Mobile First**"?
    - Limited data && Limited bandwidth
    - Sending less data and reduce HTTPs fetches
    - Encoding data
        - Json
        - Others, likes Thrift?
-

Are we heading in the right direction?

Non-Functional Requirements (Or Goal)

- Acceptable latency? High latency or Low latency?
- Prioritize availability and consistency.
- Read && Write QPS, read heavy or write heavy
-

Are we heading in the right direction?

EST(Estimation)

- 1B bytes ~= 1 GB
- 1M kb ~= 1 GB
- 1 day ~= 86k sec

Are we heading in the right direction?

Skeleton Of Design

Asynchronous or Synchronous?

- If asynchronous way works, think about MQ

Are we heading in the right direction?

Scaling, Partitioning, Sharding

Appendix

Handy metrics:

- Read sequentially from disk at 30 MB/s
- Read sequentially from 1 Gbps Ethernet at 100 MB/s
- Read sequentially from SSD at 1 GB/s
- Read sequentially from main memory at 4 GB/s
- 6-7 world-wide round trips per second
- 2,000 round trips per second within a data center

References

Building mobile first infrastructure for messenger

https://code.facebook.com/posts/820258981365363/building-mobile-first-infrastructure-for-messenger/