课前鸡汤:

If there is one take away, It definitely should be “The major barrier to skill acquisition isn't intellectual, it's emotional." It just like exercise, the mental barrier is much harder to overcome than the physical obstacle. Let's face the fear and get it done folks ! **-- It only take 20 hours**

Key point during SD interview

- Heuristic problem solving skill (Key idea: Big problem → break down to Smaller problem) → **Keep going , Never Stop**.
- Get yourself and the interviewer **engaged**. (Never idle, as if you were speaker of **TED talk**)
- **Don’t be scared**, Be a **Technical Leader**.
- **Make as much progress as you can**, don’t back down from the problem.
- [Dig Deep !!! → All about **trade off**]

Questions need know how to answer after classes:

Concurrency.
Do you understand **threads, deadlock, and starvation** ?
Do you know how to **parallelize** algorithms ?
Do you understand **consistency** and **coherence** ?
A:**Coherence** defines the behavior of reads and writes to the same memory location; **Consistency** defines the behavior of reads and writes with respect to accesses to other locations.

Networking.
Do you roughly understand **IPC(Inter-Process Communication)** and **TCP/IP**?
Do you know the difference between **throughput** and **latency**, and when each is the relevant factor?

Abstraction.
You should understand the systems you’re building upon.
Do you know roughly how an **OS, file system, and database** work?
Do you know about the **various levels of caching** in a modern OS?

Real-World Performance.
You should be familiar with the **speed of everything** your computer can do, including the relative performance of **RAM, disk, SSD and your Network**.

Estimation. Estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few **prototypes** or **micro-benchmarks** to write.

- L1 cache reference 0.5 ns
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Mutex lock/unlock 100 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

Availability and **Reliability**.
Are you thinking about **how things can fail**, especially in a distributed environment?
Do you know how to design a system to **cope with network failures**?
Do you understand **durability**?

How do you ensure that the data remains correct and complete, even when things go wrong internally? **Qurum (R+W>N)**
How do you provide consistently good performance to clients, even when parts of your system are degraded? **Replication / Redundancy;**
How do you scale to handle an increase in load? **Replication/Partition/Sharding**
What does a good API for the service look like?

- **Good API Examples (Rate limit)**
  - /product**s**
  - /product**s**?name='ABC'
  - /v1/products
  - /v2/products
  - /products?limit=25&offset=50
- API Design

What **type of database** would you use and why?
What **caching solutions** are out there? Which **would you choose and why**?
A: **Memechae** for pure performance, **Redis** if for other(general) purposes.
What **frameworks** can we use as infrastructure in your ecosystem of choice?

High Level SD Step by Step:

- **Step 1**: Ask **questions** ! Gathering **requirements**/**features**, do **calculation** and **estimation** (基于UC/Feature 来推断各种基础数据, 再基于数据来做设计) - GOALs

    - 记住,永远从用户量开始！(e.g. QPS)
    - E.g.(举栗) Design Instagram: 300m **MAU/30 --> DAU** - **10 millions** -> 10m/24/60/60 → ?? QPS
        - **80/20: 20% post daily, 80% view --> Read/Write ratio**;
            - will decide which DBs to use. --> NoSQL
            - Image Storage → AWS S3.
        - 2 millions post per day --> datatype (Pic --> 5Mb/photo)
            - Photo: 2m * 5M/image --> 10,000,000 M/PerDay --> 10TB/PerDay (24*60*60) --> **5G/s** (AWS S3 → File Server)
            - How much data to store ?
                - 10 * 365 = 3650 T → 4PB * 5 yrs → 20PB (S3 Glacier)
            - User Data / MetaData / Views / Likes etc. (DBs)
        - 10 millions views per day.
            - ???
    - Other part **API Design / Database Design / High-Availability**
- **Step 2**: Create a **high level abstract design** - Design Goals (**high-level trade-offs**: Keep in mind that everything is a trade-off.)
    - write down all the necessary **components**;
    - DB Choice: go with NoSQL (MongoDB, DynamoDB);
    - …
- **Step 3**: **Design Core components**
    - Application Server / API Design
        - API - Endpoint
    - DBs -- DB Design
        - Scale
    - Class Diagram (OOD)
    - Algorithm (e.g. Limit Rater)
- **Step 4**: **Scale** the design → find bottleneck(s)
    - DAU 10m in US --> 100m worldwide

In depth SD Step by Step:

Step 1: Gathering **requirements/features**, do calculation and **estimation** (基于UCs/Features 来推断各种基础数据, 再基于数据来做设计) - GOALs (10 minutes)

Question : (**it all about how to use (best) utilize resources[Hardware, software, People/Developer] to provide services.**)

Two Aspects: (user requirements, specs, functionalities)

- **Functional requirements(Basic)**: (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways),
    a. What does the **system do** ?
    b. What are the **inputs** and **outputs** of the system ?
    c. **Who** is going to use it ? Where are there (**Global** vs. **NA**) ?
        - Rate Limiter: who? Other developer, API consumer;
        - Instagram/Twitter: who ? normal consumer. Everyone.
    d. **How** are they going to use it ? Use Cases;
    e. How **many users** are there ? MAU/DAU.
- **Non-functional requirements**: (general properties like **security**, **reliability**, **compliance**, **scalability**, **compatibility**, and **maintainability**)
    a. **How much data** do we expect to handle ?
    b. How many **requests per second(QPS)** do we expect? What is the **request/query size** in byte ? 5G/s;
    c. What are the **(peak)throughput(**水管**)/latency(**延迟**)** requirements ? How many concurrent requests should we expect ? **6pm - 10pm** (90%)
    d. What is the expected **read to write ratio** ? (10:1 or 100:1 or **80:20** ?)
    e. Is **Storage heavy(Instagram)** or **Computing heavy(Google Maps)** ?
        - Storage heavy: lower CPU core(4c)
        - Computing heavy: higher CPU core(10c)
    f. How big/small is the data size ? 1KB vs. 1MB vs. 1GB ?
    g. What's the relationship between the data ? SQL vs. NoSQL(**MongoDB**, **DynamoDB**, **Cassandra**, **Riak**)
    h. What's the average expected **response time** ? 100ms → 10ms
    i. What's the average expected **reliability time** ? (99.9% or 99.999%)
    j. What **clients** do we want to support (UI: mobile, web, etc) ?
    k. Do we require authentication? **Analytics**? Integrating with existing systems ?
    l. What's the limit of the data we allow users to provide ?
    m. Do we want to discuss the **end-to-end** experience(Use-Case) or just the API ?

Assumptions/Estimation/Constraints -- Scope:

E.g. Assume **500M MAU** -> **70% DAU 350M DAU** → 70% 245 M DAU at peak Hour → **245M/60/60 ⇒ 68k concurrent users** at Peak Throughput.
(Will a single **EC2** can support ? if **1k per server**, we can handle it with **68 servers**)

Capacity estimation:

Assert what is reasonable ? (From stand-point of the user, be **empathetic** to the user)
What is **the minimum(Upper-bound) we need** to achieve the goal we want to achieve(at the same time provide the ability to scale) ? (You can doesn't mean you should)

Handy conversion guide:
2.5 million seconds per month
1 request per second = 2.5 million requests per month
40 requests per second = 100 million requests per month
400 requests per second = 1 billion requests per month

- **Traffic Estimate**:
  - How many requests per sec/day/week/month (**QPS=Queries Per second**)?
- **Storage Estimates**: (usually for 5 years) -->Data Driven DB design
  - Request/Month * 5/yr * 12/mo * Request size ⇒ Total size
    - 10^3 bytes=1,000 = 1 KB
    - 10^6 = 1,000,000 = 1 MB
    - 10^9 = 1,000,000,000 = 1GB (1 billion bytes)
    - 10^12 = 1,000,000,000,000 = 1TB (1 Trillion bytes)
- **Bandwidth Estimates**:
  - QPS * Request size -- need consider both write and read
- **Memory Estimates**: (Cache , **80/20 rule**, 20% common request driven 80% of the traffic)
  - **Cache** 20% of the common request (Total Request per day * Request size * 20%) < reality it should be less than that;
- **High Level Estimates**:
  - What's the **availability** ?
    - **SLA(Service Level Agreement)** --> Uptime → three 9s or four 9s (series or parallel) ?
    - 99.9999% **uptime** → 365*24*0.0001(downtime) = 0.876 hr/year

Step 2: Create a **high level** abstract design - Design Goals (5-10 minutes)

high-level trade-offs: Keep in mind that everything is a trade-off.

- **Performance** vs. **Scalability**
  - If you have a **performance** problem, your system is **slow for a single user**.(N → N^2 → N^3)
  - If you have a **scalability** problem, your system is **fast for a single user** but **slow under heavy load**. (LogN → N)
- **Latency(流的速度)** vs. **Throughput(水管流量)**
  - Aim for **maximal throughput** with **acceptable latency**.
- **Availability** vs **Consistency (DBs)**

    - **CAP** Theorem: (**Consistency**, **Availability**, **Partition-Tolerance*****)
        - 3 choice 2;
    - **Partition Tolerance** - The system continues to operate despite arbitrary partitioning due to **network failures** (This is **NOT** a choice, it **unavoidable**)
    - **CP** vs. **AP**
        - **CP** is a good choice if your business requires **strong consistency** with **atomic reads and writes**. (e.g. **Bank system**)
        - **AP** is a good choice if the business needs allow for **eventual consistency** or when the system needs to continue working despite external errors. (e.g. **Social Network Site**)
- **Consistency** patterns:
    - Weak(e.g. **Memcached**,**VoIP**) vs.
    - **Eventual**(e.g. DNS,Email, Likes) vs.
    - Strong(e.g. RDBMSes, system that need transactions, **banking system**)
- **Availability** Patterns:
    - **Replication**
        - Master-Slave(Maybe READ-Only)
        - Master-Master(READ/WRITE)
        - Lead - Follower
    - **Fail Over**:
        - Active-Passive(master failover to next slave)
        - Active-Active
        - **Disadvantage(s)**:
            - Fail-over adds more hardware and **additional complexity**.
            - There is a **potential for loss of data** if the active system fails before any newly written data can be replicated to the passive.

Step 3: Design Core components(25-35 minutes)

Design Principles for distributed system:

1. **Availability**: redundancy, rapid-recovery;
2. **Performance**:
    - Algorithm optimization;
3. **Reliability**: (fault tolerance → replication)
4. **Scalability**: → support more users.
    - DB optimization;
5. **Manageability**/**Simplicity**:
6. **Cost**: “you can does not mean you should”.

API and object-oriented design:

1. Clarification
2. Object/Class
3. (Class)Relationship

    a. Design DB Schema/relationship;
4. Actions(Methods)

Step 4: **Scale** the design (5-10 min)

Basic

- **Services**: aka Platform layer, SOA/SOD
    - Service Discovery: (e.g. **Consul**, **Etcd**, and **Zookeeper, gossip**) need to tell which node have what data/service (e.g. routing tier)
- **Redundancy**: shared nothing architecture, reduce single-point-of-failure;
- **Partitions/Sharding**: keeping consistency is the key, secondary indexes could be messy to handle. (Document-based or Term-based)
- **Handling failure**:

Building Blocks of **Fast** and **Scalable** Data Access (App & DB server)

- **Goal**: scalability of storage & fast access of data;
- **Caches**: (e.g. **Redis**, **Memcached**)
    - Global caches:
    - Distributed caches: also with Consistent Hashing (avoid hot-spot)
        - 1,hashing “server”,
        - 2, has”key”,
        - 3,multiple hashing.
- **Proxies**: (**Nginx**, **HAProxy** [High Availability Proxy], ActiveProxy)
    - Collapsed forwarding (coaggregate same request to speed things up), by content(data) or by data location.
- **Indexes**: (Secondary Index, B-Tree 4k)
    - **table of content** (e.g. SST(String Sorted Table) LSM-Tree(Log-Structured Merge Tree))
    - **Inverted Index**: e.g. if there are 10 billion books in the library, how do you search ? create an index by “word” instead of by “book”. (because there are much more book than words)
        - Similar to sorting 10 billion 10 bit integers.
- **Load Balancers**: (**Nginx, AWS ELB, F5(Hardware)**)
    - DNS (AWS Route 53, CloudFlare)
- **Queues**: (**Kafka, RabbitMQ**, AWS **SQS**, ActiveMQ, **Zookeeper**, **Redis**)
    - Scheduling: Celery/Cron

Additional consideration

- **CDN(Content Delivery Network)**: (**CloudFlare**, AWS **CloudFront**)
- **Monitoring**: (**Zookeeper**, AppDynamic,**Splunk**,DataDog, AWS **CloudWatch**)
- **DNS**: **CloudFlare** and **AWS Route 53**; (Avoid single point of failure)

E.g. Step by Step Scale from 1 to 10+ millions user

- Users > 10
  - single host (E.g. 1 EC2 instance)
  - One host for the web site.(Tomcat/SpringBoot)
  - One host for the database.(MySQL) (E.g. 1 RDS DB- 20G storage)
- Users > 100
  - Separate host for the web tier.(Multiple application servers) (E.g. 2 EC2 instances)
- Users > 1000
  - Slave DB (master - slave) (E.g. 2 RDS DB - 2 * 20G)
  - Load Balancer (multiple web app servers) (E.g. 3 EC2 + AWS ELB)
- Users > 10,000s - 100,000s
  - more web-tier instance
  - more DB replicas
  - CDN / CloudFront (CSS/JS files)
  - AWS S3 object base store (Photos/Video)
  - **AWS ElastiCache**/DynamoDB (Session/Cache/**Redis**)
- Users > 500,000+
  - **CloudWatch**/CloudTrail
  - **Splunk** → analyze logs
  - Auto scaling ~ (up to 1000s) instances (Kubernete/Docker)
  - Automation (**Elastic Beanstalk**, OpsWorks,CloudFormation)
  - **CodeDeploy** (Chef/Puppet, **CI/CD** Pipeline)
  - Decouple Infrastructure(SOA/**Microservice-DB**)
  - **SQS**/**Lambda**(Serverless,**kafka**, no-state)
- Users > 1,000,000+
  - Multi-AZ(Available Zone, Geolocation, **US-EAST**, US-WEST, Asian, Europe)
  - Caching for DB (Redish for DB write/read)
- Users > 10,000,000+
  - DB (**Sharding**, Migrate DB, Federation)
- Users > 11 Million
  - Multi-AZ to multi-region.

Key Concepts Deep Dive:

Concepts we need to know

- **Reliability(Foundation)**
  - **Redundancy**
- **Scalability**
  - **Stateless**
- **Maintainability**
  - **KISS (Keep It Simple & Stupid)**
- "**Don't count the server, but make the server count**" → scale up by **optimizing** code ! (Monitor, Analyze, Optimize)
  - Do-less (push calculation to the client side if possible, reduce computing) ;
  - Reduce code (reduce memory);
  - Async IO (reduce latency, mitigate server starvation);

DNS(Domain Name Server/System) Component:

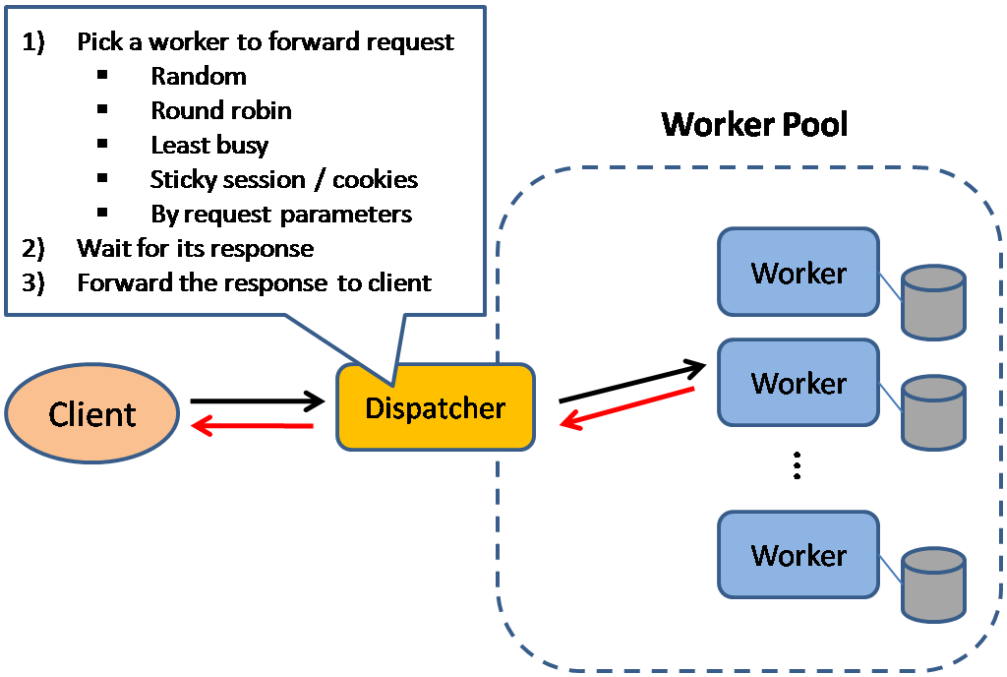
- Domain map to IP. (巨型的HashMap<Domain, IP>)
  - PING www.google.com → (IP: **142.250.72.196**)
    - <www.google.com, 142.250.217.196>
- Services such as **CloudFlare** and **AWS Route 53**
  - www.google.com → 142.250.217.196
  - Bard.google.com → 142.250.217.197
  - XXX.google.com → 142.250.217.198
- **Route methods**
  - Weighted round robin; (e.g. A/B testing, Server maintenance etc.)
  - **Latency**-based; (lower latency the better)
  - **Geolocation**-based; (Closer to the user the better)
- **Disadvantage**(s): DNS
  - **Added Delay**: Accessing a DNS server introduces a slight delay, although mitigated by caching described above.
  - **Added Complexity**: DNS server management could be complex and is generally managed by governments, ISPs, and large companies.
  - **Added Security Risk**: DNS services have recently come under DDoS attack, preventing users from accessing websites such as Twitter without knowing Twitter's IP address(es).

CDN(Content Delivery Network):

- Amazon's **CloudFront** supports dynamic content. **CloudFlare** etc.
- Data Route Method:
  - **Push**: receive new content whenever changes occur on your server.
    - **Con**: may push lots of unnecessary content to waste CDN Space, Sites with a small amount of traffic or Content don't change too often.
  - **Pull**: grab new content from your server when the first user requests the content.
    - **Con**: First request is slow, works well for sites with heavy traffic.
- **Time-To-Live (TTL)** - 24hr/6hr/1 week. determines how long content is cached.

    - Cache invalidation. (e.g. LRU cache)
- **Advantages**:
    - Users receive content at data centers close to them;
    - Your servers do not have to serve requests that the CDN fulfills;
- **Disadvantages**:
    - **Added Costs**: CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
    - **Added Delayed**: Content might be stale if it is updated before the TTL expires.
    - **Added Layer**: CDNs require changing URLs for static content to point to the CDN.

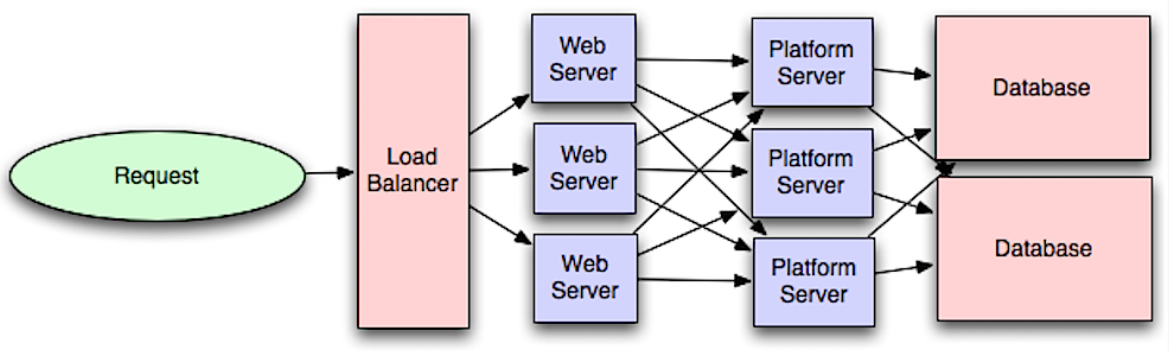
Load Balancer: (HAProxy, Nginx)



- **Advantages**:
    - Preventing requests from going to unhealthy servers;
    - Preventing overloading resources;
    - Helping eliminate single points of failure;
    - SSL termination; (just need one certificate)
        - Removes the need to install X.509 certificates on each server;
    - Session persistence (sticky session by issue cookie);
- **Disadvantages**:
    - The load balancer can become a **performance bottleneck** if it does not have enough resources or if it is not configured properly.
    - Introducing a load balancer to help eliminate single points of failure results in **increased complexity**.
    - A **single load balancer is a single point of failure**, configuring multiple load balancers further increases complexity.
- **Route Method (Pros vs. Cons)**

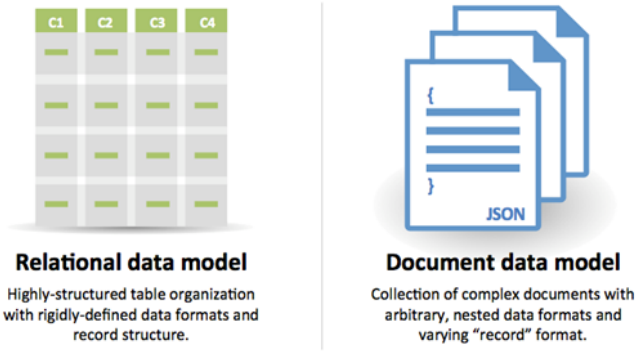
    - Random
    - Least loaded
    - Stick-Session/cookies:
        - User1 Request 1 → Server A(Session/Cookie)
        - User1 Request 2 → Server A
    - Round robin or Weighted round robin: 轮流来
    - **Layer 4**: (F5 - physical LB)
        - look at info at the **transport** layer(**source, destination IP addresses, and ports in the header**) to decide how to distribute requests. performing Network Address Translation (NAT).
    - **Layer 7**:
        - look at the **application** layer(**header, message, and cookies**) to decide how to distribute requests.
    - **L4 vs. L7**:
        - At the cost of flexibility, layer 4 load balancing requires **less** time and computing resources than Layer 7.
- Requirement for **Horizontal scaling vs. Vertical Scaling**
    - **Disadvantage**(s): horizontal scaling
        - Scaling horizontally introduces complexity and involves cloning servers:
            - Servers should be **Stateless(without Session)/Lambda**: they should not contain any user-related data like sessions or profile pictures;
            - **Sessions(User temp. data)** can be stored in a centralized data store such as a database (SQL, NoSQL) or a persistent cache (Redis, Memcached);
        - Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out;
- **Load balancer** vs. **Reverse proxy**
    - Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.
    - **Reverse proxies** can be useful even with just one web server or application server, opening up the benefits described in the previous section.
    - Solutions such as **NGINX** and **HAProxy** can support both layer 7 reverse proxying and load balancing.

Application/Platform Layer: (aka. Microservice, SOA/SOD)



- Separating out the **Web layer** from the **Application layer (also known as Platform layer)** allows you to scale and configure both layers independently.
- **Microservices**: a suite of independently deployable, small, modular services.
    - **E.g.** Pinterest, could have the following microservices: user profile, follower, feed, search, photo upload, etc.
- **Service Discovery**: (e.g. **Consul**, **Etcd**, and **Zookeeper**)
    - Can help services find each other by keeping track of registered names, addresses, and ports.
    - Health checks help verify service integrity and are often done using an HTTP endpoint.
    - Both **Consul** and **Etcd** have a built-in key-value store that can be useful for storing config values and other shared data.

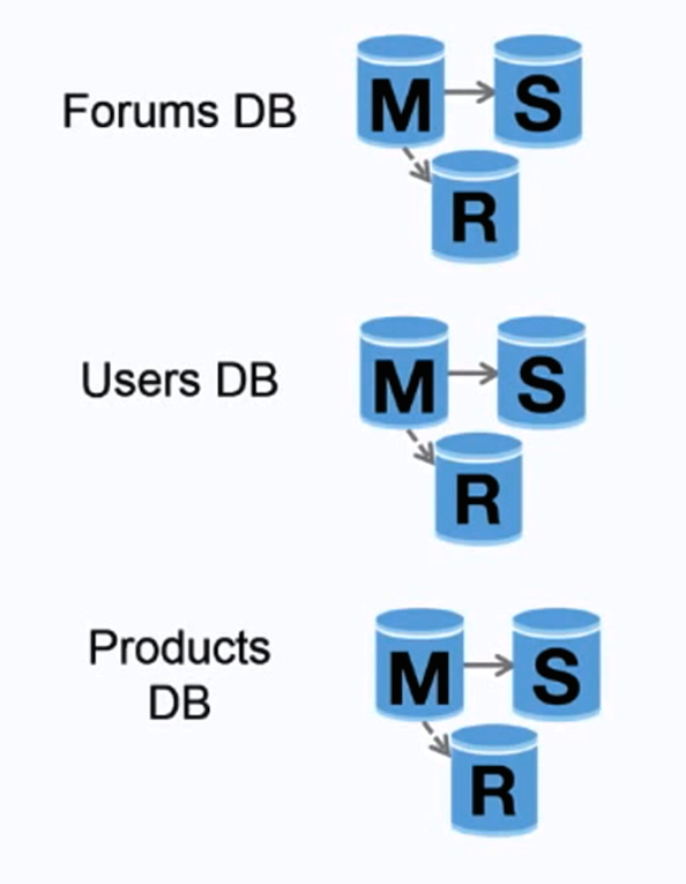
Database*(重点)



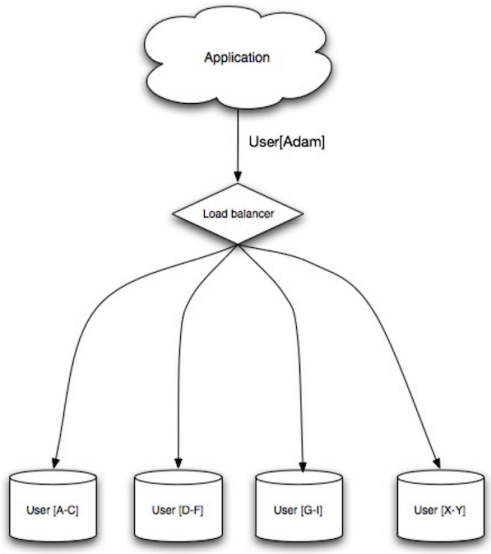
SQL - RDBs:

- There are many **techniques** to scale a relational database:
    - **master[write/read]-follower(s)[read] replication**,
        - **Con**: Additional logic is needed to promote a slave to a master.
    - **master-A[R/W]-master-B[R/W] replication**,
        - **Con**:
            - Either loosely consistent (violating **ACID**) or have increased write latency due to synchronization;
            - Write Conflict handling;
    - **Replication Con**(s):
        - **Potential Loss of data**: There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
        - **Lower Read speed**: Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
        - **Replication lag**: The more read slaves, the more you have to replicate, which leads to greater replication lag.

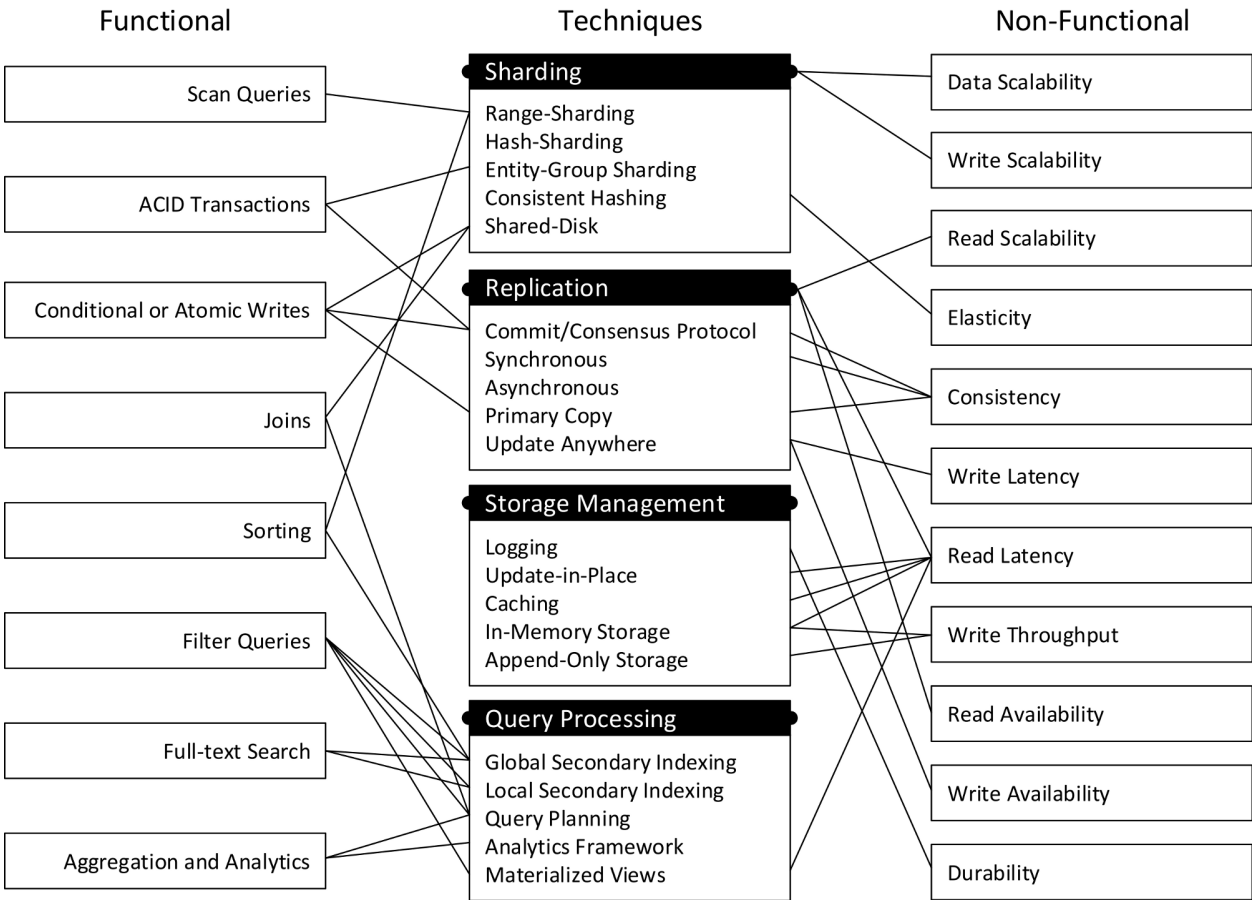
        - **Single Thread Write**: On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread;
        - **Cost & Complexity**: Replication adds more hardware and additional complexity.
    - **Federation**: Federation (or **functional partitioning**) splits up databases by function. (C: Not good for relational data)
        - E.g. use three databases: **forums**(MySQL1), **users**(MySQL2), and **products**(MySQL4), **posts**(MySQL3)
        - **Con(s)**:
            - Federation is not effective if your schema requires huge functions or tables.
            - You'll need to update your application logic to determine which database to read and write.
            - **Joining** data from two databases is more complex with a server link.



- ***Sharding/Partitioning(分区)**, (e.g. UserDetail last-name DB → A~ F, G ~ P, W~Z, e.g lastName **Li**)



  - **Con(s)**:
    - You'll need to update your application logic to work with shards, which could result in complex SQL queries.
    - Data distribution can become lopsided in a shard. (E.g. A set of power users[e.g Justin Biber] on a shard could result in increased load to that shard compared to others;)
      - **Rebalancing** adds additional complexity;
      - A sharding function based on **Consistent hashing** can reduce the amount of transferred data.
    - Joining data from multiple shards is more complex.
- **Denormalization(平铺数据)**, (**User** join **UserAddress** → **UserDetail**)
  - **Trade off**:
    - Time complexity & Space complexity.
  - Denormalization attempts to **improve read** performance at the expense of some write performance.
  - Redundant copies of the data are written in multiple tables to **avoid expensive joins**.
    - e.g. Alibaba avoid more than 3 table joins;
- In most systems, reads can heavily outnumber writes 100:1 or even 1000:1.
  - A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.
- **Other SQL tuning**.
  - Tighten up the schema;
    - Avoid storing large **BLOBS(binary → static files)**, store the location of where to get the object instead. (e.g S3)
  - Use good **Indices(e.g. a HashMap within DB)**;
    - **Pro**: increase lookup speed;
    - **Con**: (slightly) decrease write speed; (Need also update the HashMap on every write)



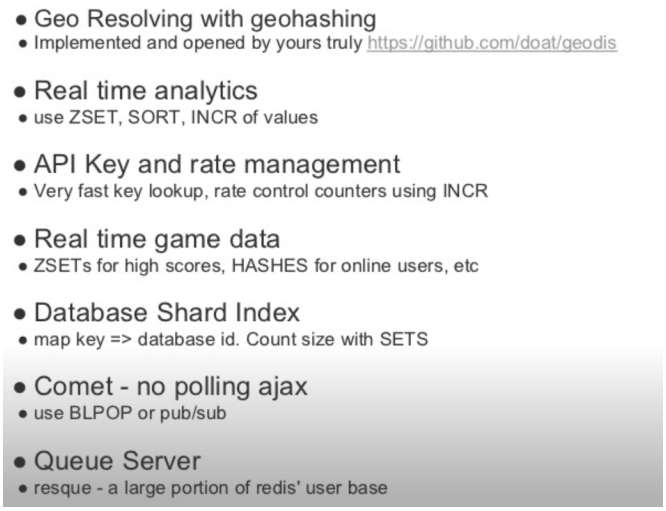
	Funct. Req.								Non-Funct. Req.									Techniques																					
	Scan Queries	ACID Transactions	Conditional Writes	Joins	Sorting	Filter Queries	Full-Text Search	Analytics	Data Scalability	Write Scalability	Read Scalability	Elasticity	Consistency	Write Latency	Read Latency	Write Throughput	Read Availability	Write Availability	Durability	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared-Disk	Transaction Protocol	Sync. Replication	Async. Replication	Primary Copy	Update Anywhere	Logging	Update-in-Place	Caching	In-Memory Storage	Append-Only Storage	Global Indexing	Local Indexing	Query Planning	Analytics Framework	Materialized Views
MongoDB	x		x		x	x	x	x	x	x	x		x	x	x	x	x		x	x	x					x	x		x			x	x			x	x	x	
Redis	x	x	x								x		x	x	x	x	x		x							x	x		x		x								
HBase	x		x		x			x	x	x	x	x	x	x		x			x	x					x		x		x		x			x				
Riak							x	x	x	x	x	x		x	x	x	x	x	x		x		x				x		x	x	x	x			x	x		x	
Cassandra	x		x		x		x	x	x	x	x	x		x		x	x	x	x		x		x				x		x	x		x		x	x	x		x
MySQL	x	x	x	x	x	x	x	x			x		x						x					x	x		x	x	x						x	x		

Table 1: A direct comparison of functional requirements, non-functional requirements and techniques among MongoDB, Redis, HBase, Riak, Cassandra and MySQL according to our NoSQL Toolbox.

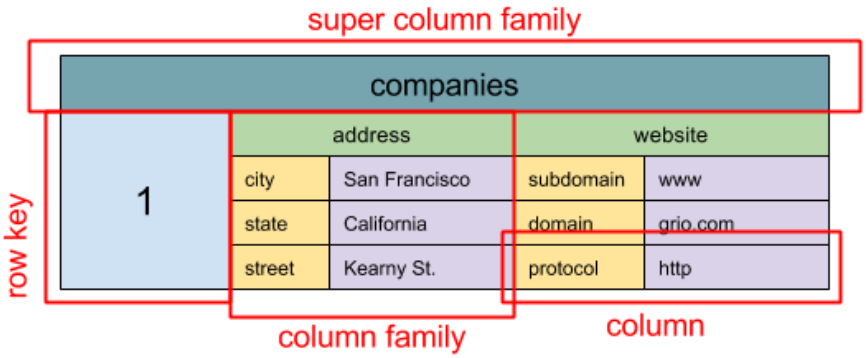
NoSQL - DBs: (Aggregation-Oriented)

- **4 Type of NoSQL DBs**: key-value stores, document stores, wide column stores, and graph databases;

- **Key-value stores**: (e.g. **Memecahed**, **Redis**, **DynamoDB**)
  - **Abstraction: hash table**;
  - **O(1) reads and writes** and is often backed by memory or SSD;
    - With some “Metadata” e.g. for index/secondary index;
    - Value == Aggregate
  - **Used cases**: basis for more complex systems. (e.g. Cache)



- **Document stores**: (e.g. **MongoDB**, CouchDB, **DynamoDB**)
  - **Abstraction: key-value store with documents stored as values**;
    - A document store is centered around documents (XML, **JSON**, binary, etc), where a document stores all information for a given object.
    - Document == Aggregate
  - **Used cases**: Document stores provide **high flexibility** and are often used for working with occasionally changing data. (schema)
- **Wide column stores(?)**: (e.g. Bigtable → HBase[Hadoop], **Cassandra**)



  - **Abstraction: nested map ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>**;
  - A wide column store's basic unit of data is a column (name/value pair).
  - A column can be grouped in column families (analogous to a SQL table).
  - Super column families further group column families.

    - You can access each column independently with a row key, and columns with the same row key form a row.
    - Column family == Aggregate
    - **Used cases**: Wide column stores offer **high availability** and **high scalability**. They are often used for very large data sets.
- **DynamoDB**:
    - Dynamo has a multi-master design(with Global Tables)? (Single Leader in each Region ?) requires the client to resolve version conflicts and
    - DynamoDB uses synchronous replication across multiple data centers for high durability and availability.
- **1st three NoSQL types are Aggregation-Oriented**:
    - It is great if you don't need to change the "Aggregate", but if you do, then it will be little more complicated and you will need to do extra to slice/dice the data (e.g. Use MapReduce);
- **Graph databases**: (e.g. **Neo4j**, FlockDB etc.) [**ACID** Properties]
    - **Abstraction: graph**;
        - In a graph database, each node is a record and each arc is a relationship between two nodes.
    - **Used cases**: Graphs databases offer high performance for data models with complex relationships, such as a social network.
- **Aggregation-oriented** vs. **Transaction-oriented**
    - **AO**: also known as analytical databases, are a type of database that is designed to handle large volumes of data and provide quick access to aggregated data.
        - E.g. Amazon Redshift, Google BigQuery, and Apache Cassandra.
        - Use Cases: Business Intelligence, data warehouse;
    - **TO**: optimized for rapid transaction processing.

NoSQL vs SQL

More SQL vs NoSQL differences

Reasons for SQL:

- **Structured** data
- **Strict schema(格式)**
- **Relational data(JOINs)**
- Need for complex joins
- **Transactions**
- Clear patterns for scaling (?)
- More established: developers, community, code, tools, etc
- Lookups by index are very fast;

Reasons for NoSQL: (easier for Scale)

- Semi-structured data(格式不那么严格)
- **Dynamic or flexible schema(implicit schema)**

- **Non-relational data**
- No need for complex joins
- **Store many TB (or PB) of data**
- Very data intensive workload
- **Very high throughput for IOPS (QPS)**

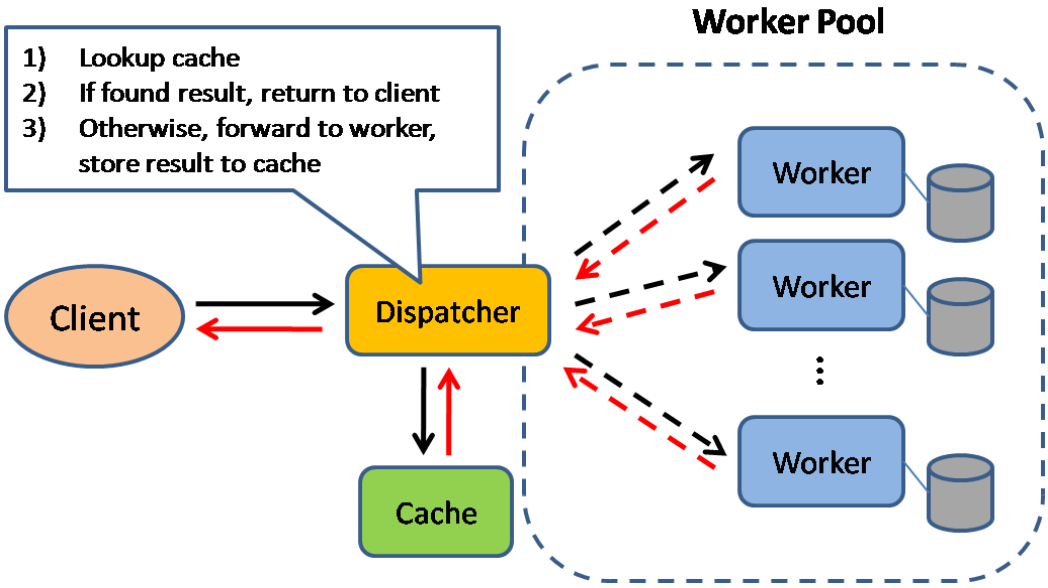
Sample data well-suited for NoSQL:

- Rapid ingest of clickstream and log data;
- Leaderboard or scoring data;
- Temporary data, such as a shopping cart;
- Frequently accessed ('hot') tables;
- Metadata/lookup tables;

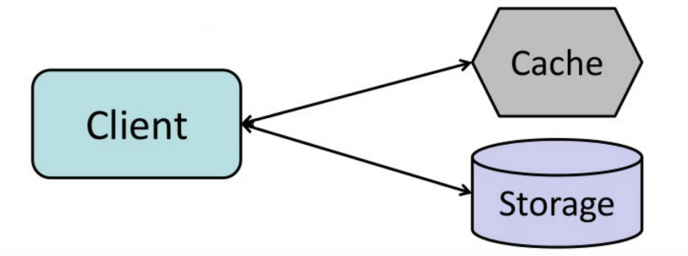
Asynchronism

- **Message queues**:
  - **Kafka** - for real-time data streaming and processing, designed to handle large amounts of data with high performance and fault tolerance.
    - It can handle millions of messages per second and can store petabytes of data.
    - Help building real-time data pipelines and streaming applications.
  - Redis is useful as a simple message broker but not as stable.
  - RabbitMQ is popular but requires you to adapt to the 'AMQP' protocol and manage your own nodes.
  - Amazon SQS is hosted but can have high latency and has the possibility of messages being delivered twice.
- Task Queues
  - **Celery** has support for scheduling and primarily has python support.
- **Prevent Queue overflow** - Back pressure;
  - E.g. Rate limiting, flow control, and throttling;
- **Cons**:
  - Use cases such as inexpensive calculations and real time workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.

Cache*(万精油)

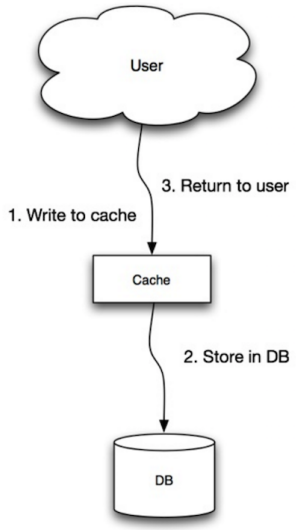


- Caching improves page load times and can reduce the load on your servers and databases.
- Putting a cache in front of a database can help **absorb uneven loads** and spikes in traffic.
- **Type of Cache**: Client, CDN(为S3做Cache), Web Server, Database, Application;
- **Cache invalidation** (Great topic) **TTL**
  - 8 hrs / 24 hrs / 3 days / 1 week ; (e.g. YouTube vs. Netflix)
  - E.g. based view count within > 1 million (1 week);
- **Objects to cache**:
  - User sessions (never use the database!)
  - Fully rendered blog articles (static content)
  - Activity streams
  - User<->friend relationships
- **Cache Strategies(*When/How to update cache)**:
  - **Cache-aside** is also referred to as **lazy loading**.



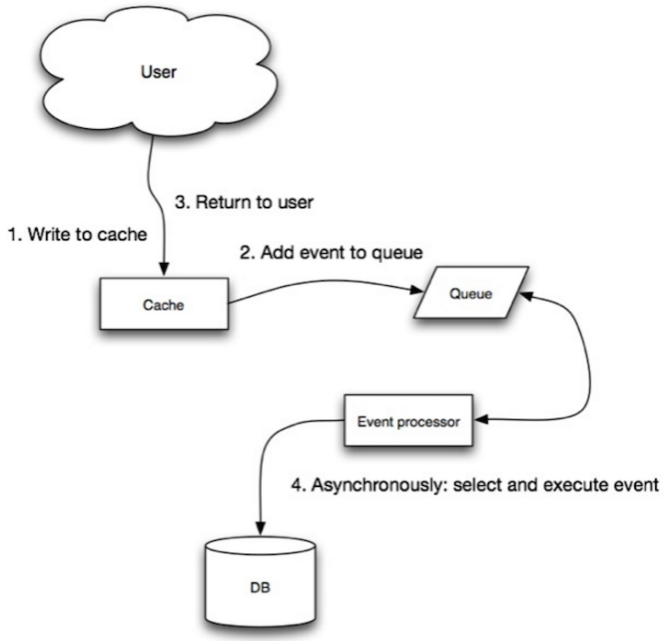
  - Only requested data is cached, which avoids filling up the cache with data that isn't requested.
  - Application logic to handle **both** cache & DB**;**
  - **Cons**:

        - Each cache miss results in three trips, which can cause a noticeable delay.
        - Data can become stale if it is updated in the database. (Use TTL)
        - When a node fails, it is replaced by a new, empty node, increasing latency.
    - **Write-through**: write to both Cache and DB at the same time (Higher Latency)



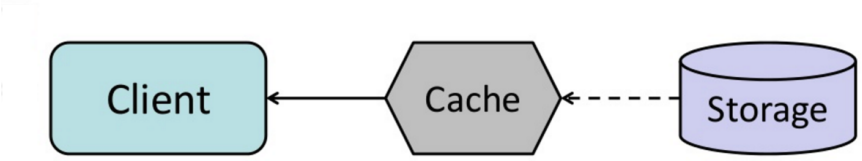
        - Cache handle DB;
        - Write-through is a slow overall operation due to the write operation.
        - **Cons**:
            - When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. (Cache-aside in conjunction with write through can mitigate this issue).
            - Most data written might never be read, (Solved with a TTL)
    - **Write-behind (write-back)**: Add/update entry in cache, **Asynchronously** write entry to the **data store(DB)**, improving write performance. (faster read/write with

lower latency but risk of loss data)



- **Cons**:
  - There could be data loss if the cache goes down prior to its contents hitting the data store.
  - It is more complex to implement write-behind than it is to implement cache-aside or write-through.

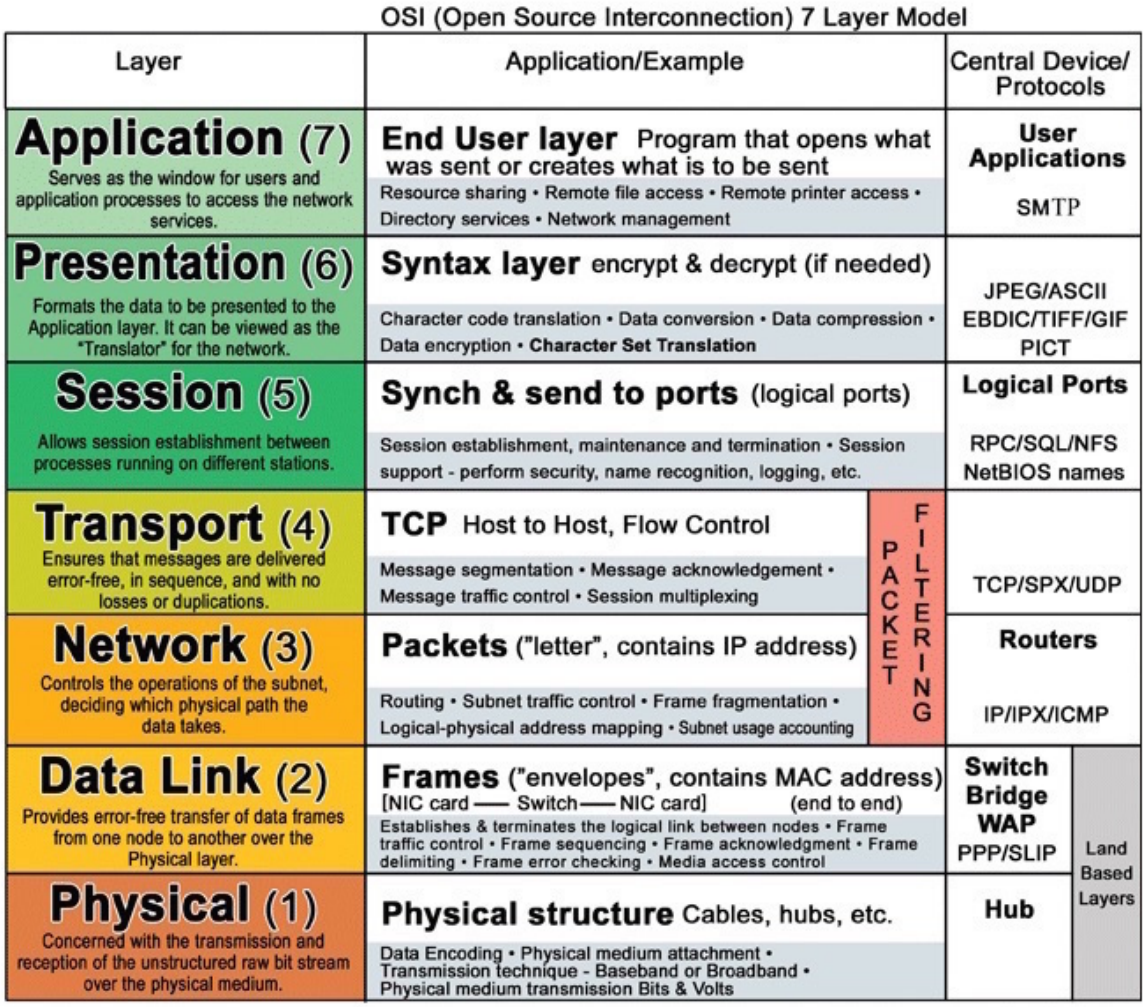
- **Write-Around**: write to **DB(replication)** first and **then cache** read from the DB(faster write, slower read)
  - higher requirements on **consistency**; (e.g. Bank System)
- **Refresh-ahead**: automatically refresh any recently accessed cache entry prior to its expiration.



  - Refresh-ahead can result in reduced latency vs. read-through if the cache can accurately predict which items are likely to be needed in the future.

- **Cache Disadvantage(s)**:
  - Need to maintain consistency/sync between caches and the source of truth(DBs) through cache invalidation.
  - TTL → Cache **invalidation** is a difficult problem, there is additional complexity associated with when to update the cache.
  - Need to make **application** changes such as adding **Redis** or **memcached**.

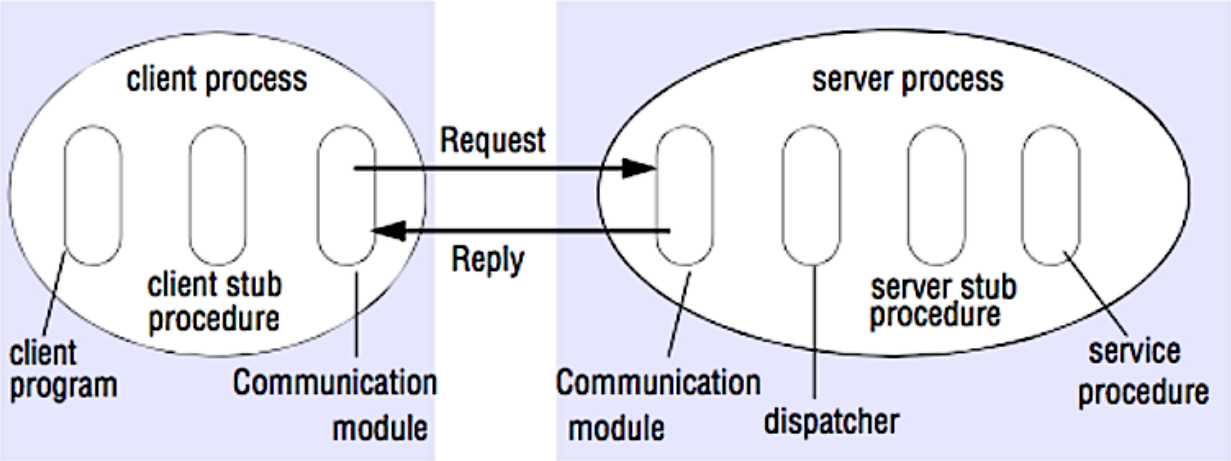
Communication



- **PUT vs. PATCH**
  - **PUT = replace** the ENTIRE RESOURCE (e.g. User address) with the new representation provided (no mention of related resources in the spec from what i can see)
  - **PATCH = replace** parts of the source(e.g. Apartment get updated) resource with the values provided AND|OR other parts of the resource are updated that you haven't provided (timestamps) AND|OR **updating** the resource effects other resources (relationships)
- ***TCP(Transmission control protocol) vs UDP(User datagram protocol)**
  - TCP: TCP is a connection-oriented protocol over an IP network
    - TCP is useful for applications that require **high reliability but are less time critical**. (e.g. web servers, database info, SMTP, FTP, and SSH.)
    - You need all of the data to arrive intact

        - You want to automatically make a best estimate use of the network throughput
    - UDP: UDP is **connectionless**. (send it and forget it)
        - UDP is **less reliable but works well in real time use cases** (e.g. VoIP, video chat, streaming, and real time multiplayer games.)
        - You need the lowest latency
        - **Late data is worse than loss of data**
        - You want to implement your own error correction

Remote procedure call (RPC)



- RPC frameworks include **Protobuf(**Google**)**, **Thrift**, and **Avro**.
- RPC is focused on exposing behaviors. RPCs are often used for performance reasons with internal communications, as you can hand-craft native calls to better fit your use cases.
- **Disadvantage(s): RPC**
    - RPC clients become tightly coupled to the service implementation.
    - A new API must be defined for every new operation or use case.
    - It can be difficult to debug RPC.
    - You might not be able to leverage existing technologies out of the box.

Representational state transfer (REST)

- All communication must be **stateless** and **cacheable**.
- Being stateless, REST is great for horizontal scaling and partitioning.
- REST vs. RPC

Security

- Checklist of the most important security countermeasures when designing, testing, and releasing your API. API security check list

Additional Resources:

- Great Summary
  - https://www.1point3acres.com/bbs/thread-698113-1-1.html
  - https://www.1point3acres.com/bbs/thread-706795-1-1.html
  - https://www.1point3acres.com/bbs/thread-726130-1-1.html
- https://www.educative.io/courses/grokking-the-system-design-interview?aff=K7qB
- Tech dummies 和 Gaurav sen，两位的sys design视频看完基本能cover大部分主流框架了
- https://www.tutorialspoint.com/software_engineering/index.htm
- 九章讲的系统设计

Github

- ☐ https://github.com/donnemartin/system-design-primer
- ☐ https://github.com/checkcheckzz/system-design-interview#tips

Book:

- ☐ <Designing Data Intensive Applications>
- ☐ https://tianpan.co/notes/2016-02-13-crack-the-system-design-interview (Educative)

Tools:

- Online whiteboard https://awwapp.com/
- Design tool https://Draw.io
- Good Design Pattern site https://refactoring.guru/design-patterns
- Algorithm Visualization https://visualgo.net/en

Official Blog:

- ☐ https://aws.amazon.com/blogs/architecture/
- ☐ https://github.com/donnemartin/awesome-aws

Reading:

- ☐ Back of the Envelope calculation
- ☐ https://www.1point3acres.com/bbs/thread-208829-1-1.html
- ☐ https://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=559285
  - ☐ http://highscalability.com/blog/2016/1/11/a-beginners-guide-to-scaling-to-11-million-users-on-amazons.html
  - ☐ http://www.aosabook.org/en/distsys.html
- ☐ Great article
  - ☐ https://hackernoon.com/how-not-to-design-netflix-in-your-45-minute-system-design-interview-64953391a054
- ☐ https://www.quora.com/Where-can-I-practice-for-system-design-interview-questions

Examples

- ☐ http://ninefu.github.io/blog/System_Design_Reading_List/
- ☐ https://www.evernote.com/shard/s576/client/snv?noteGuid=7e58b450-1abe-43a8-bf82-fbf07f1db13c&noteKey=049802174415b418a2e65f75b744ab72&sn=https%3A%2F%2Fwww.evernote.com%2Fshard%2Fs576%2Fsh%2F7e58b450-1abe-43a8-bf82-fbf07f1db13c%2F049802174415b418a2e65f75b744ab72&title=Interview%2BPreparation

Practice Tool

- [ ] Grokking the System Design Interview
- [ ] https://www.interviewbit.com/courses/system-design/
- [ ] https://www.javatpoint.com/design-patterns-in-java