**Scalability:**

David malan- Youtube :https://youtu.be/-W9F\_\_D3oY4

Vertical scalling: - not a full solution of scalling

- increase RAM anfd disk space -> that increases costing

- sas drives 1500 speed /s - high speed data drives

Horizontal scalling:

- get bunch of slower or cheaper machines

- load balancer to distribute traffic across backend servers

- session saved under tmp folder as serialised files

- if we store session on one server and that server goes down all systel goes down.

- RAID - is a best solution for session sharing and reducing app downtime

RAID 0,1,2,3,4,5,6,10 has its own features

-It is used to increase the performance of the system

Sticky session;

fiber channel

Caching:

- mysql cache - set \_cache=1 in config

- memcahche- is good way of caching to store data and cleaup un used data to store new data.

facebook uses memcahche to load profile

-

In read heavy case-> facebook uses

send read requests to slave db and write request to master DB (master master setup to avoid write downtime)

read call willl be redirected to multiple dbs using load balancer

**High level Design:**

Divide into 2 logical parts:

1. Logic-> which will handle all incoming request
2. Data storage- store all data that need to be persisted

Design a simple UI or draw it and explain to interviewer to check both are on same page.

* Handling User Request:
* System should handle at least thousand request /seconds
* Solution are concurrent and use less memory than others.
* Scale machine parallelly to handle multiple request and distribute load- use loadbalancer
* Storing the data
* We should use some caching solution that could save us lot od reads directly from the database. Memcache
* Memcache stores data in memory so it has faster access.
* We need to add appropriate index to execute quickly joining queries.

**Low level Issue:**

* **Database design**
* If you are using relational DB, create a database design and show relationship between tables.
* Create appropriate indexes to execute queries fast.
* Draw it on paper and explain
* **Building RESTful API**
* The API endpoints will likely be built around the data entities that we have and the needs of the user-facing part of the application.
* Put authentication

Increased Number of requests:

1. The bottleneck would be our DATABASE.
   * We should use replication so read request will be distributed. Or
   * shard database if more write calls
2. Web application slow down
   * Scale machine horizontally (add more machines)
   * Auto scaling machine – aws feature

**Performance vs scalability**

* If you have a **performance** problem, your system is **slow for a single user**.
* If you have a **scalability** problem, your system is **fast** for a **single** user but **slow under heavy load**.

**Latency vs throughput**

* **Latency** is the time to perform some action or to produce some result.
* **Throughput** is the number of such actions or results per unit of time.

Generally, you should aim for **maximal throughput** with **acceptable latency**.

**Availability vs consistency**

In a distributed computer system, you can only support two of the following guarantees:

* **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 failure

A partition is when the network fails to deliver some messages to one or more nodes by losing them.

CP - consistency and partition tolerance

* 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/synchronized reads and writes**.

AP - availability and partition tolerance

* 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.

**Content Delivery Network:**

A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user.

* Pull CDN:

- Updates content when new changes occur and rewrites url to point to CDN.

- can configure expire time and when it can be updated.

- Minimize traffic, increases storage.

Work well with- less updates and less amount of traffic.

* Push CDN:

- Load content on users first request and rewrite URLs to CDN, it slows down the request.

- Minimize storage but create redundant traffic if file expires.

- Work with heavy traffic

Disadvantages:

-Costly depends on traffic, URL changes needed

**Load Balancer:**

­­ It distributes the incoming traffic to application servers and databases. Returns the response to clients.

LB effective at:

* + Preventing requests from going to unhealthy servers
  + Preventing overloading resources
  + Helping eliminate single points of failure.

Additional Benefits:

* SSL termination:

Decrypt incoming request and encrypt response so backend servers do not have to perform these operations

* Session Persistence:

Issue cookies and route traffic to same server.

Load balancers can route traffic based on various metrics

1. Random
2. Least loaded
   * Check the active connection to the server and route the traffic
3. Sticky Session/cookies
   * A load balancer that keeps sticky sessions will create a unique session object for each client. For each request from the same client, the load balancer processes the request to the same web server each time
4. Round robin
   * Request are routed to servers on cyclical basis/ sequentially.
   * It only works well with identical server configuration (computing capabilities and storage).
5. Weighted round robin
   * Admin assign weights to each server based on criteria like traffic handling capacity so with higher weights receive higher client requests (if servers are with different configuration).
6. Layer 4
   * Read transport layer information source, destination IP addresses and ports from the heard and decides how to distribute the request performing network address translation.
   * It doesn’t read the content of the packet.
   * It requires less time for routing.
7. Layer 7
   * Look at the application layer data like header, messages and cookies and route the traffic.
   * E.g video traffic to server that host videos
   * Server or Active cookie persistence
   * Server or Active cookie persistence with source IP

**Reverse Proxy:**

* A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public

Additional Benefits:

* Increased Security

-hide information about backend servers.

* Increase scalability and flexibility

-Only reverse proxies IP exposed to client so easy to add new servers

* SSL termination

-Auto encrypt and decrypt request and response

* Compression

-Compress server responses

* Caching

-Return response for cached requests

* Static content

-serve static content directly , HTML,video,photos

Use case:

Load Balancer: when you have multiple servers

Reverse proxy: only one web or application server.

Disadvantages:

-Single point of failure and multiple proxies increases complexity.

**Database:**

**ACID** is a set of properties of relational database [transactions](https://en.wikipedia.org/wiki/Database_transaction).

* Atomicity - Each transaction is all or nothing
* Consistency - Any transaction will bring the database from one valid state to another
* Isolation - Executing transactions concurrently has the same results as if the transactions were executed serially
* Durability - Once a transaction has been committed, it will remain so

**Database scaling technique**:

* + **Master-Slave Replication**

Master- > only write data Slaves-> only Read calls

* + - Master replicates data to all slave databases.
    - If master fails system continue to operate in read only mode until slave is promoted as master.

Disadvantages:

Additional logic needed to promote slave as master.

* + **Master-Master Replication**

Both masters serve reads and writes and coordinate with each other on writes

Disadvantages:

* Need load balancer or you need to write logic to determine where to write
* Either loosely consistent (violating ACID)
* Write calls increase latency bcoz of synchronization

Replication Disadvantages:

* Loss of data if master fails before replication
* More writes slow down the read calls and cannot do more read calls
* More read slave ->more replicated-> lead to replication gap
* replication write supports only sequentially
  + **Federation**
* Splits database by functions like user, forum, product
* Less read and write traffic to each DB and less replication lag
* Increase throughput by parallel writes.

Disadvantages:

* Not effective if more functions and tables.
* Need application logic to determine which db to read and write
* Joining data from two db complex
* Add more hardware and complexity
  + **Sharding**

Sharding distributes data across different databases such that each database can only manage a subset of the data

**-**sharding results in less read and write traffic, less replication, and more cache hits.

**-** Index size is also reduced, which generally improves performance with faster queries.

Disadvantages:

* + - Update application logic to work with shards ->result in complex sql queries.
    - Data distribution become lopsided -> set of power user -> increase load to that shard
    - Joining data complex
    - More hardware and complexity
  + **Denormalization:**
* Improve read performance at the expense of write.
* Redundant copies of data write in tables to avoid expensive joins
* Oracle and postgres support materialized view to store redundant and consistent copies.

Disadvantages

* Data duplication
* Constraints help to sync data-> complexity of db design
* Heavy write load performs worse
  + **SQL Tuning:**
* Benchmark - Simulate high-load situations with tools such as [ab](http://httpd.apache.org/docs/2.2/programs/ab.html).
* Profile - Enable tools such as the [slow query log](http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html) to help track performance issues.

Tighten up the schema

* MySQL dumps to disk in contiguous blocks for fast access.
* Use CHAR instead of VARCHAR for fixed-length fields.
* CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
* Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
* Use INT for larger numbers up to 2^32 or 4 billion.
* Use DECIMAL for currency to avoid floating point representation errors.
* Avoid storing large BLOBS, store the location of where to get the object instead.
* VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
* Set the NOT NULL constraint where applicable to [improve search performance](http://stackoverflow.com/questions/1017239/how-do-null-values-affect-performance-in-a-database-search).

Use good indices

* Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
* Indices are usually represented as self-balancing [B-tree](https://en.wikipedia.org/wiki/B-tree) that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
* Placing an index can keep the data in memory, requiring more space.
* Writes could also be slower since the index also needs to be updated.
* When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

