SYSTEM DESIGN

Standard Interview Questions
Design Patterns
Low Level ,High Level, & Class Based Design Questions

"System design interview questions have become a standard part of the software engineering hiring process. Performance in these interviews reflects upon your ability to work with complex systems and translates into the position and salary the interviewing company offers you. Most engineers struggle with system design interview (SDI), partly because of their lack of experience in developing large scale systems, and partly because of the unstructured nature of SDIs. Even engineers, who've some experience building such systems, aren't comfortable with these interviews, mainly due to the open-ended nature of design problems that don't have a standard answer."

STEP BY STEP APPROACH –

1) Requirements Clarification E.g., Text Chat, Group Chat, Attachments etc. for deigning a Messenger. Questions about scale, size, budget, team required for design.
2) System Interface Definitions E.g., Send Message (from_user, to_user, message)
3) Back of the Envelope Estimation (How much memory or disk space required?)
4) Defining Data Model (E.g., TblSchema, and their relationships)
5) High Level Design (E.g., Flow diagrams between servers, storage, cloud and client/users) (Top-down / Bottom-up Approach)
6) Component Design (E.g., Cache, load balancers, Servers, Storage, Partition, Cloud)
7) Identify and Resolve BottleNecks

Below are most frequently asked design questions in interviews which generally involves building large-scale systems and handling bottlenecks. This

is almost always the last round of the interview process which decides your final package. Learn how to tackle such problems.

Q1. A Word About System Architecture and Scalability.

Ans. Scalability cannot just be optimized. It has to be planned. It has to be part of the architecture design at the early stages. If our System Architecture isn't scalable, it is often an indication that developers are running into situations where the architecture of their system limits their ability to grow their service.

A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Alternatively, adding resources should not result in loss of performance.

Scalability requires applications and platforms to be designed with scaling in mind, such that adding resources actually results in improving the performance or that if redundancy is introduced the system performance is not adversely affected. Many algorithms that perform reasonably well under low load and small datasets can explode in cost. If either requests rates increase, the dataset grows or the number of nodes in the distributed system increases. if we architect and engineer our systems to take scalability into account, we can deliver good scalability.

The simplest advice is that scaling in the large is not adding resources to an architecture designed to scale in the small. an architect dealing with very large distributed systems has to decide whether to relax requirements for consistency or availability since network partitions are a given. It's a trade off one needs to make while dealing with larger and larger audience. For example, if a developer decides to

relax consistency requirements, he needs to decide how to handle a situation where a write to the system is not immediately reflected in a corresponding read.

Q2. A word About Gossip Protocol and its role in Scalability.

Ans.

- **The Basic Essence -** A gossip protocol is simple in concept. Each nodes sends out some data to a set of other nodes. Data propagates through the system node by node like a virus. Eventually data propagates to every node in the system. It's a way for nodes to build a global map from limited local interactions.
- gossip protocols can be used to detect node failures and load balance based on data sampled from other nodes.
- Gossip protocols have proven to be effective means by which failures can be detected in large, distributed systems in an asynchronous manner without the limitations associated with reliable multicasting for group communications.

- Example, GEMS or Gossip Enabled Monitoring System is a highly responsive and scalable resource monitoring service, to monitor health and performance information in heterogeneous distributed systems. It comes with features such as detection of network partitions and dynamic insertion of new nodes into the service and distribution of system & application specific data in relatively fast response time and low resource utilization.

- **Failure Detection -** By exchanging and combining the reachability data from a lot of different nodes you can quickly determine when a node is down. You need atleast two nodes to reach the consensus that a node is down. When a node is down, there's no need to attempt to write to that node, saving queue space, CPU, and bandwidth.

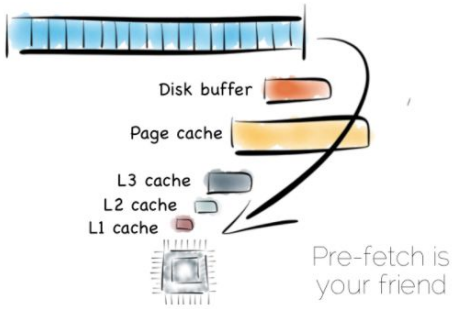
- **Gossip for Messaging -** Gossip offers an effective approach to transmit node properties to other nodes. Stats like load average, free memory, etc. would allow a local node to decide where to send work, for example. If a node is idle send it work. This local decision making angle is the key to scale. There's no

centralized controller. Local nodes make local decisions based on local data. This can scale as far as the gossip protocol can scale.

- For example, queue depth for a module could be sent out so other modules could gauge the work load. Alarm information could be sent out so other entities know the status of modules they are dependent on. Key information like configuration changes can be passed on. Even requests and response can be sent through this mechanism. Another advantage of this approach is that data could flow directly into your monitoring system rather than having a completely separate monitoring subsystem bolted on.

Q3. Explain the concept of Locality in performance.

Ans.



01*. Locality in CPU; Faster Performance during Sequential Access*

Q4. What are Deep Neural nets?

Ans. The "**deep**" in Deep Learning refers to the **number of layers in the neural network**. Neural nets learn a really complicated function from data. Inputs from one space are transformed into outputs in another space. The function these neurons compute is the weighted sum of their inputs times the weights applied through some nonlinear function. The nonlinear function used today is a rectified linear unit (max(o,x)) instead of sigmoid or tan h functions. Since it has the nice property of

giving true zeros when the neuron doesn't fire as opposed to values which are close to zero which can help you when optimizing the system.

For example, if a neuron as three inputs X1, X1, X3

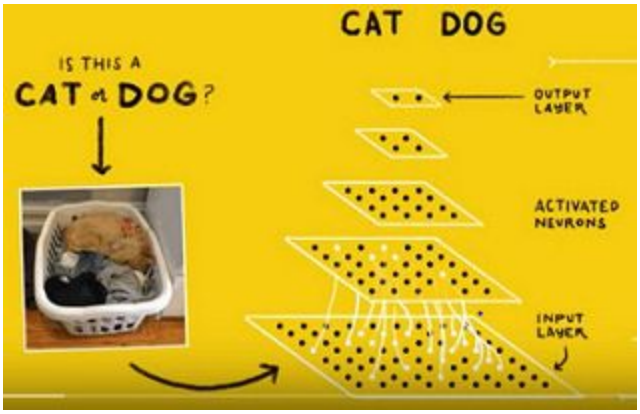
with the weights -0.21, 0.3, and 0.7,

the calculation would be:

$$y = \max(0, -.0.21*x1 + 0.3*x2 + 0.7*x3)$$

- In determining if an image is a cat or dog the image will be put through a series of layers. Neurons will fire or not depending on their inputs.

- In determining if an image is a cat or dog the image will be put through a series of layers. Some of the neurons will fire or not based on their inputs.



- The lowest layer neurons will look at little patches of pixels. The higher level neurons will look at the output of the neurons below and decide to fire or not. The model will work its way up through the layers.

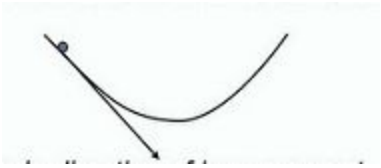
- The **goal of a neural net, is to make little adjustments to the weights on all the edges** throughout the model to make it more likely to get the example right. This is done in aggregate across all the examples so that in aggregate you get most of the examples right.

TYPES OF DATA THAT CAN BE USED IN NEURAL NETWORKS –

- Text: Languages, Words, Sentences, Grammar and Translations
- Visual Data: Billions of images and videos.
- Audio: Hours of Speech
- User Activity: Search Queries, Application usage, Spam filtering in e-mails, Social networking data.
- Knowledge Graph: Lots of labelled relation triples.

STEP OF THE THE LEARNING ALGORITHM -

- Pick a random training example "(input, label)".
- Run the neural network on "input" and see what it produces.
- Adjust weights on edges to make output closer to "label".
- Use Backpropagation i.e., taking partial derivatives along the path with neural nets.
- Follow the gradient of error in the direction of the arrow and adjust weights accordingly in an iterative manner.
- To keep getting better results throw more data at the problem and make the model bigger.
- Lower the error rate as much as possible.



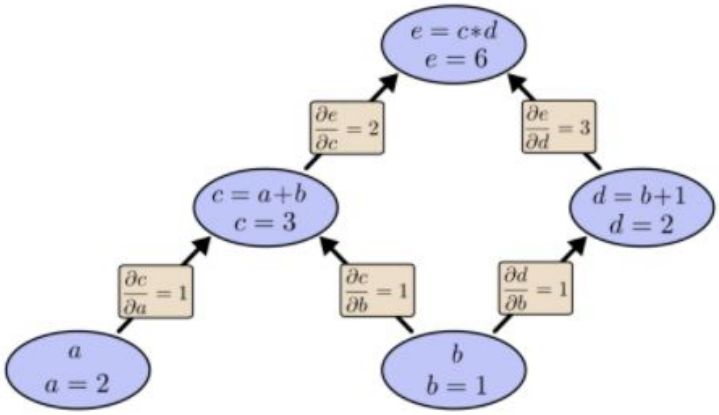
2. Gradient Descent

BACKPROPOGATION -

Backpropagation or 'reverse mode differentiation' is the key algorithm that makes training deep models computationally tractable. It's a technique for calculating derivatives quickly. If variable *a* directly affects variable *c*, then we want to know how it affects *c*. If *a* changes a little bit, how does *c* change? We call this **the partial derivative** of *c* with respect to *a*.

For modern neural networks, it can make training with gradient descent as much as ten million times faster, relative to a naive implementation. That's the difference between a model taking a week to train and taking 200,000 years.

Below, the graph has the derivative on each edge labeled.



The general rule is to sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path together. For example, to get the derivative of *e* with respect to *b* we get:

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

This accounts for how *b* affects *e* through *c* and also how it affects it through *d*. Reverse-mode differentiation gives us the derivative of *e* with respect to every node, that is derivative of output w.r.t each input parameter.

When training neural networks, we think of the cost (a value describing how bad a neural network performs) as a function of the parameters (numbers describing how the network behaves). *To calculate the derivatives of the cost with respect to all the parameters, for use in gradient descent, backprop is much better option.* Now, there's often millions, or even tens of millions of parameters in a neural network. So, reverse-mode differentiation, called backpropagation in the context of neural networks, gives us a massive speed up!

$$E^p = \frac{1}{2}(D^p - M(Z^p, W))^2, \qquad E_{train} = \frac{1}{P}\sum_{p=1} E^p$$

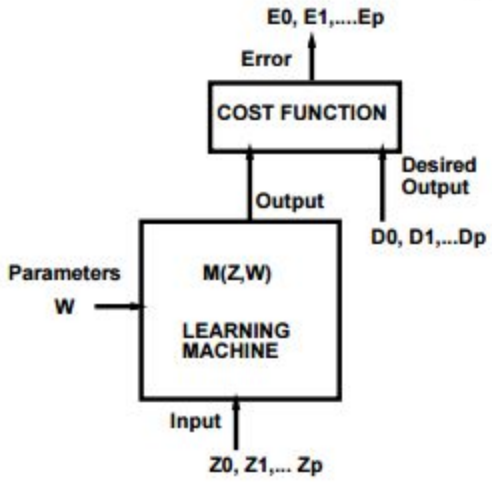


Fig. 1. Gradient-based learning machine.

Fig(i). The learning problem consists of finding cost unction by calculating the difference between output produced by the system and desired output. And Minimizing Average Error by adjusting W values

Q5. How Dropbox dealt with Scaling and other Technological issues faced by them initially?

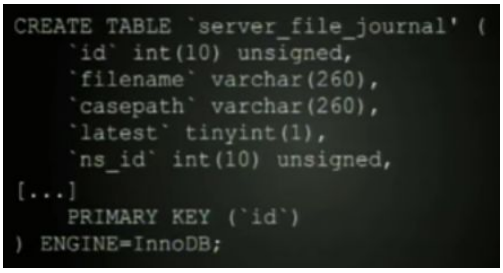
Ans. 65% CONSUMER BASE OUTSIDE USA

Constraints:

- ☐ ACID PROPERTY.
- ☐ HIGH CONSISTENTCY REQUIREMENT
- ☐ MASSIVE WRITE UPDATES THAN READ
- ☐ DATA STORED AS FILE BLOCKS
- ☐ SHARED FOLDERS ARE TRICKY TO SHARD

Techniques:

- ☐ DE-DUPLICATION
- ☐ PARTITIONING FILE INTO 4K BYTES OF SUB BLOCKS
- ☐ HASHING SUB BLOCKS WITH SAME HASH VALUES AMONG THE SAME FILE.
- ☐ RECONNNECTING SUBBLOCKS WITH SAME HASH VALUES INTO A SINGLE ORIGINAL FILE.
- ☐ ACTUAL FILES ON AMAZON EC2 SERVER (WEST COAST) REST ON LOCAL SERVER (EAST COAST).
- ☐ REPLICATION HANDLED BY AMAZON.
- ☐ LOG CHANGES TO FILES IN DATABASE AND METADATA ABOUT WHAT IS IN USERS' DROPBOX



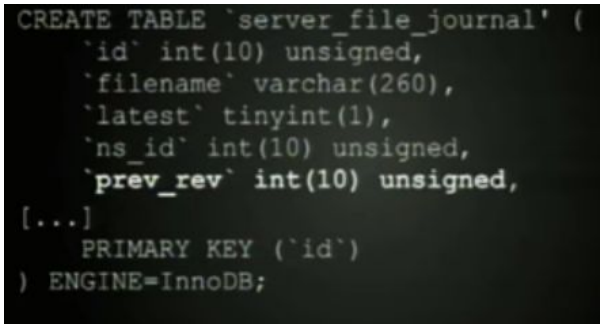
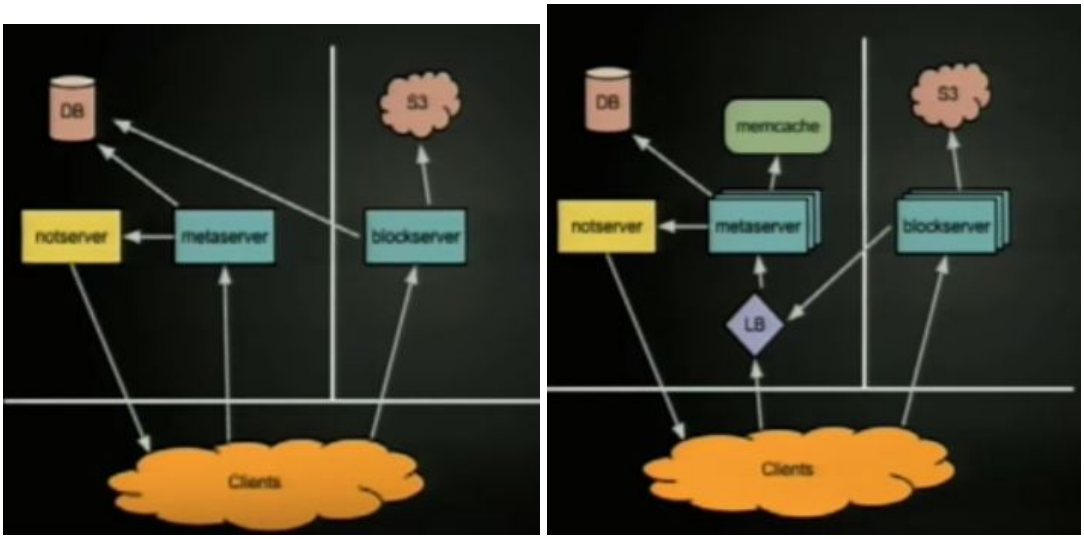


TABLE SCHEMA (original and modified later)



Initial High Level Architectures (Multiple blockserver for write updates, and Multiple metadata for metaserver) Multiple Memcache and Load Balancers added

Q6. How do Netflix Recommendation Engine works?

Ans.

- **ANALYZING METADATA ABOUT USER VIEW PATTERNS**

By looking at the metadata, you can find all kinds of similarities between shows. Look at user behavior—browsing, playing, searching.

- Were they created at roughly the same time?
- Do they tend to get the same ratings?
- Are they from similar Genre?
- Are they from same Director?
- Do they hold similar cinematography pattern?
- Do they provide similar movie experience? For example, Spielberg may have built movies from different genres but the art of storytelling remains the same through all of them. Or some other, writer or producer.

- **KEEPING REGULAR TRACK OF USER BEHAVIOUR**
  - Tracking what user played, searched, or rated.
  - Tracking viewing time, date, and device.
  - Even tracking user interactions such as browsing or scrolling behavior.
  - Tracking Placement. The closer to the first position on a row a title is, the more likely it will get played. The higher up on the page a row is, the more likely it is to generate a play.
  - Tracking whether Recommendation feature is more commonly used than search feature.
  - All that data is fed into several algorithms, each optimized for a different purpose.

- **PREPPING ALGORITHMS BASED ON AVAILABLE DATA FOR OPTIMIZING VIEWING EXPERIENCE**
  - Algorithms are based on the assumption that similar viewing patterns represent similar user tastes.
  - So, similar users are matched and the behavior of similar users are used to infer your preferences.
  - Use Contextual data to improve Recommendation engine.
  - Contextual recommendation data suggests there is different viewing behavior depending on the day of the week, the time of day, the device, and sometimes even the location.

Q7. Overview about SCALABILITY, AVAILABILITY and STABILITY patterns?

Ans. SCALABILITY – If your system is fast for a single user but slow under heavy load.

PARTITIONING --------SHARDING----CACHING—CONCURRENCY

In a scalable architecture, resource usage should increase linearly (or better) with load, where load may be measured in user traffic, data volume, etc. Where performance is about the resource usage associated with a single unit of work, scalability is about how resource usage changes as units of work grow in number or size.

- **For Centralized Systems, Partition Tolerance is not Required. So we can pick both Consistency and Availability.**

- **For Distributed Systems, we have only two option CP or AP.**

ACID Vs BASE

DROP IT: Atomicity, Consistency, Isolation and Durability.

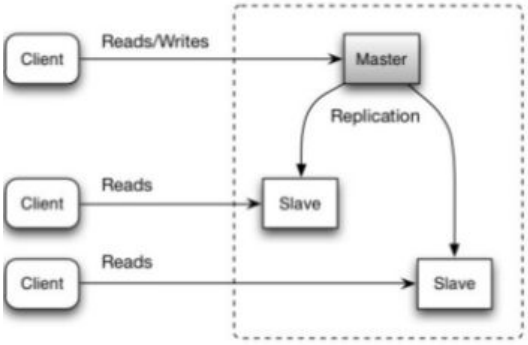
BASE: BASICALLY AVAILABLE, SOFT STATE, EVENTULLY CONSISTENT

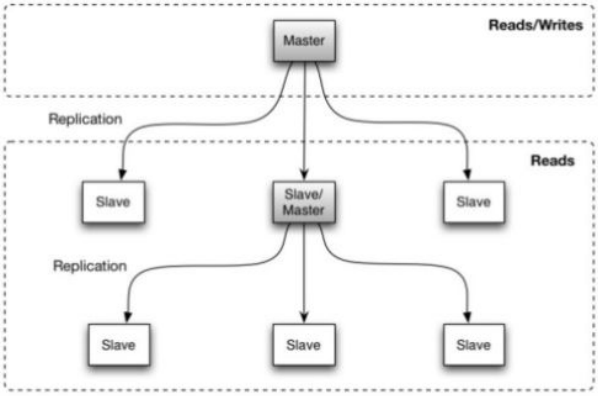
AVAILABILITY PATTERNS:

Availability is the net up-time of server. OR Total Duration Minus Downtime.

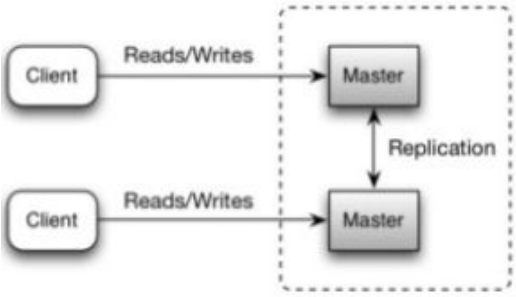
- **Fail Over**
  - A procedure by which a system automatically transfers control to a duplicate system when it detects a fault or failure.
- **Replication**
  - Master-Slave
    - A device known as master unidirectional controls one or more other devices known as slaves. It allows you to easily maintain multiple copies of data which is automatically copied from master database to slave database.
  - Tree Replication
    - Replication using a Tree Structure/Model
  - Master-Master
    - Allows data to be stored by a group of computers, and updated by any member of the group. All members are responsive to client data queries.
  - Buddy Replication
    - Buddy Replication allows you to suppress replicating your data to all instances in a cluster. Instead, each instance picks one or more 'buddies' in the cluster, and only replicates to these specific buddies.

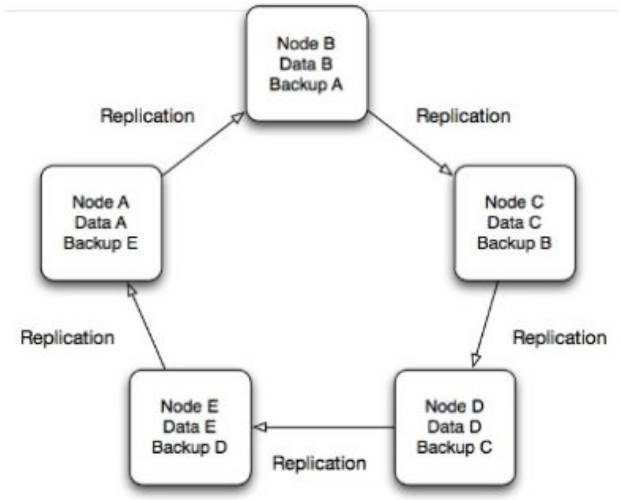
Different Data Different Needs





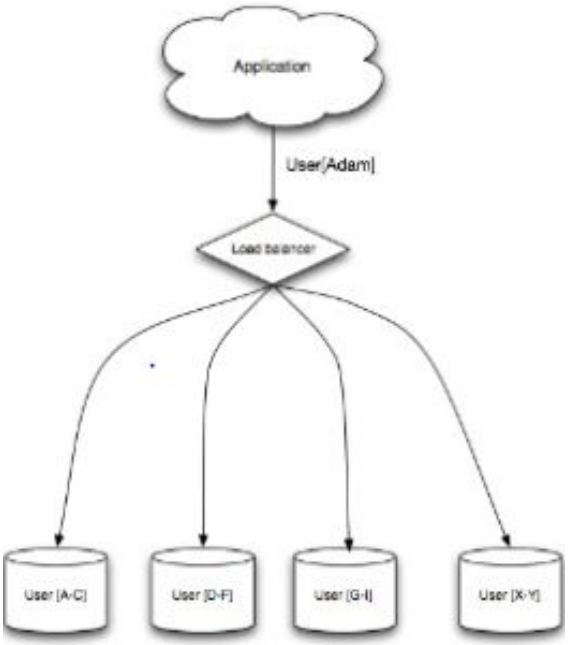
Master Slave Replication *Tree Replication*

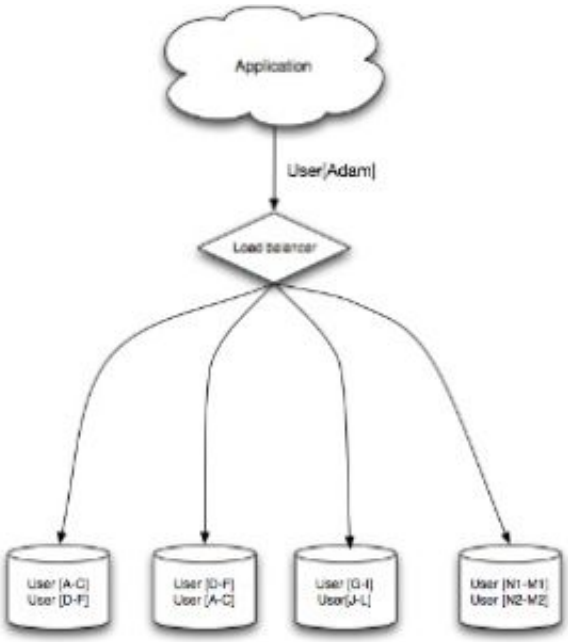




Master-Master Replication *Buddy Replication*

How to scale Databases?





1. **SHARDING/PARTITION**

2. REPLICATION

NOT ONLY SQL or NOSQL is easier to scale than relational databases. Examples:-

- Google: Bigtable
- Amazon: Dynamo
- Amazon: SimpleDB
- Yahoo: HBase
- Facebook: Cassandra
- LinkedIn: Voldemort

CONCURRENCY

- SHARED STATE CONCURRENCY
- MESSAGE PASSING CONCURRENCY
- DATAFLOW CONCURRENCY
- SOFTWARE TRANSACTIONAL CONCURRENCY

ISSUES

- Race Condition
- Deadlocks
- Starvation
- Live Locks

MEMCACHED IS A DISTRIBUTED CACHING MECHANISM WITH FEATURES LIKE

- Eviction Policies
- Replication
- Cache Invalidation
- Peer to Peer
- Key Value (String to Binary)

Caches are considered efficient when cache accesses are more than cache misses.

- storage cost
- retrieval cost (network load / algorithm load)
- invalidation (keeping data up to date / removing irrelevant data)
- replacement policy (FIFO/LFU/LRU/MRU/RANDOM vs. Belady's algorithm)
- cold cache / warm cache

Features of memcached

What to Cache?

- **Database Access**
- **File system Access**
- **API Calls**
- **Heavy Computation**
- **Operations that work with slower resources**
- **Expensive operations**

Q8. How to keep up with Rapidly Growing Systems?

Ans. Following are some best practices for Scalability used by EBay:

1. **SPLIT BY FUNCTIONALITY: functional decomposition**
   - **At the Application tier,**
     - EBay segments different functions into separate application pools.
     - Selling functionality is served by one set of application servers, bidding by another, search by yet another.
     - In total, EBay organizes its roughly 16,000 application servers into 220 different pools.

- This allows them to scale each pool independently of one another, according to the demands and resource consumption of its function.

- **At the Database Tier,**
  - There is no single monolithic database at eBay.
  - There is a set of database hosts for user data, a set for item data, a set for purchase data, etc.
  - In total, 1000 logical databases partitioned functionally on 400 physical hosts.
  - This approach allows us to scale the database infrastructure for each type of data independently of the others.

❷ SPLIT HORIZONTALLY: break the workload of the functional units

- **At the Application Tier,**
  - Use a standard load-balancer to route incoming traffic to any application server, since all serve the same functionality.
  - Add additional servers for more processing power.
- **At the Database Tier,**
  - Shard the data horizontally along its primary access path.
  - Different use cases use different schemes for partitioning the data.
  - As numbers of users grow, and as the data stored for each user grows, more hosts are added, and the users are subdivided further among hosts.

- The general idea is that an infrastructure which supports partitioning and repartitioning of data will be far more scalable than one which does not.

❸ **AVOID DISTRIBUTED TRANSACTIONS:** Guaranteeing Commit across all resources or not at all, incurs ever-increasing cost of coordination as the clients and dependent resources grow in number which worsens Scaling, performance, and latency & even availability. Thus, one needs to Relax Transactional Requirements across unrelated systems.

- The CAP Theorem states that of three highly desirable properties of distributed systems - Consistency (C), Availability (A), and Partition-tolerance (P) - one can only choose two at any one time.
- For a high-traffic web site, one has to choose partition-tolerance, since it is fundamental to scaling.
- For a distributed system, *partition tolerance* means the system will continue to work unless there is a total network failure. If few nodes fail, the system keeps going.
- For a 24x7 web site, one has to choose availability.
- So immediate consistency has to give way. Guaranteeing immediate consistency across multiple systems or partitions is typically neither required nor possible.
- Consistency should not be viewed as an all or nothing proposition. Most real-world use cases simply do not require immediate consistency.
- Tailoring the appropriate level of consistency guarantees to the requirements depending upon the operation is what is required.
- One must employ various techniques to help the system reach eventual consistency such as (1) careful ordering of database operations, (2) asynchronous recovery events, and (3) reconciliation or settlement batches.

❼ DECOUPLE FUNCTIONS ASYNCHRONOUSLY: aggressive use of asynchrony is the key element to scaling

- If component A calls component B synchronously, A and B are tightly coupled, and that coupled system has a single scalability characteristic -- to scale A, you must also scale B.
- if A and B integrate asynchronously, whether through a queue, multicast messaging, a batch process, or some other means, each can be scaled independently of the other.
- Avoid synchronous coupling as much as possible between the components.
- At every level, decomposing the processing into stages or phases, and connecting them up asynchronously, is critical to scaling.
- Reduce physical dependencies and improve deployment flexibilities.

❺ MOVE PROCESSING TO ASYNCHRONOUS FLOWS: aggressive use of asynchrony is the key element to scaling

- Performing operations synchronously forces you to scale your infrastructure for the peak load as it needs to handle the worst second of the worst day at that exact second.
- Moving expensive processing to asynchronous flows, though, allows you to scale your infrastructure for the average load instead of the peak.
- Instead of needing to process all requests immediately, the queue spreads the processing over time, and thereby dampens the peaks.

❼ **VIRTUALIZE AT ALL LEVELS:** As we scale our infrastructure through partitioning by function and data, an additional level of virtualization of those partitions becomes critical.

- The operating system abstracts the hardware. The virtual machine in many modern languages abstracts the operating system. Object-relational mapping layers abstract the database. Load-balancers and virtual IPs abstract network endpoints.
- **Virtualizing the Database:** Applications interact with a logical representation of a database, which is then mapped onto a particular physical machine and instance through configuration.
- **Virtualizing the Search Engine:** To retrieve search results, an aggregator component parallelizes queries over multiple partitions, and makes a highly partitioned search grid appear to clients as one logical index.
- Virtualization makes scaling the infrastructure possible because it makes scaling manageable.
- With judicious use of virtualization, higher levels of the infrastructure are unaware of the changes, and you are therefore free to make them.
- Operational flexibility ensures seamless experience when Hardware and software systems fail, and requests need to be re-routed. Or, Components, machines, and partitions are added, moved, or removed.

❼ **CACHE APPROPRIATELY:** the goal of an efficient caching system to maximize your cache hit ratio within your storage constraints, your requirements for availability, and your tolerance for staleness.

- Reducing repeated requests for the same data can make a substantial impact. The most commonly cached data includes slow-changing, read-mostly data such as metadata, configuration, and static data.

- The more memory you allocate for caching, the less you have available to service individual requests.
- A good caching system can bend your scaling curve below linear - subsequent requests retrieve data cheaply from cache rather than the relatively more expensive primary store.
- On the other hand, caching done poorly introduces substantial additional overhead and availability challenges.
- Some companies like E-bay take the approach where they do not cache shared business objects, like item or user data, in the application layer explicitly trading off the potential benefits of caching this data against availability and correctness.
- Caching strategies and approaches should be devised as appropriate for your situation.

Q9. How performance can be improved by Design?

Ans. PERFORMANCE IS A DESIGN DECISION. ISSUES WITH IT: -

High latency due to sequential resource access

Memory constrained

There are certain design principles if followed can keep many development issues, scalability issues, and performance issues at bay. These are:-

Design Principals

-: SIMPLIFY LAYERS: -
DECOMPOSE ARCHITECTURE
DEFINE SLAs

CONTINOUS PERFROMANCE TESTING
UTILIZE CACHING OR DATA GRIDS
USE A CDN
USE COMPRESSION AND MINIFICATION
-: APPLY BEST PRACTICE UI PERFORMANCE TECHNIQUES: -

Performance Testing Can be done by: -

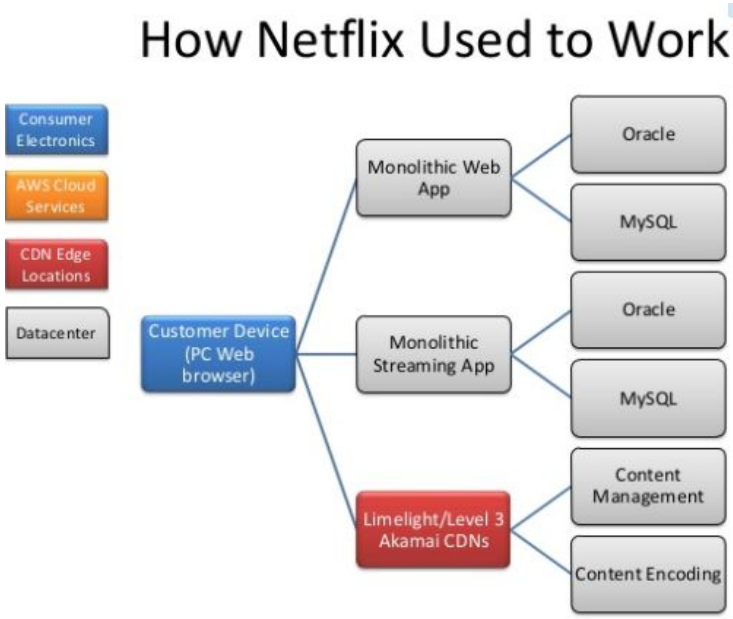
1. Response Time by Number of Concurrent Users
2. Request per second by Number of concurrent users
3. Server loss scenarios
3. Page Loads

A CDN can help –

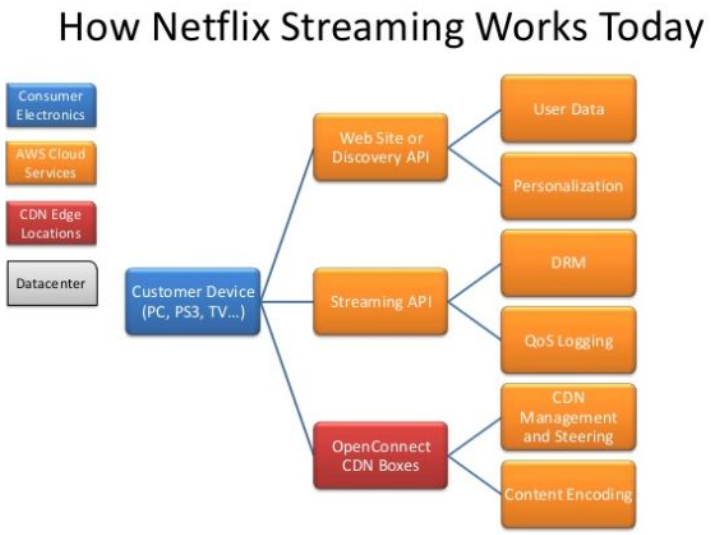
1. Move Your Content Closer to end users
2. Reduce Latency
3. Increase offloads

Q9. Design Architecture of Netflix?

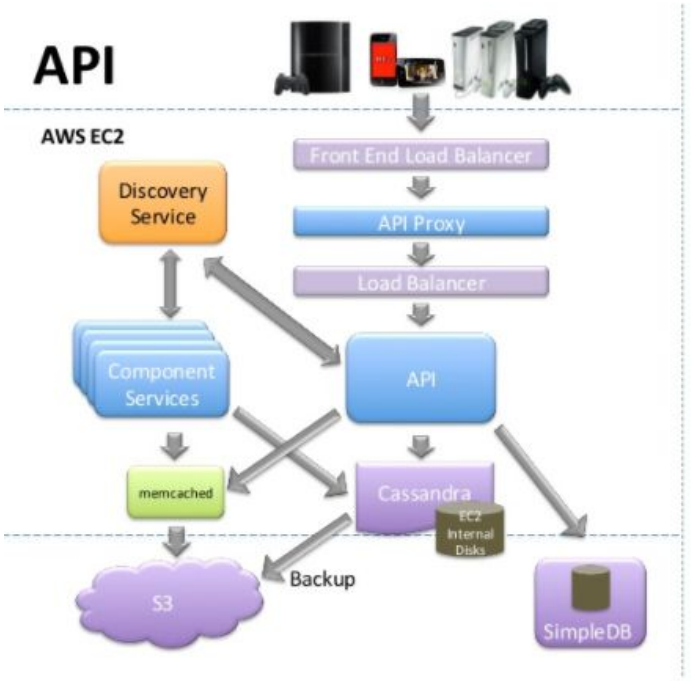
Ans. Netflix has built a large scale global Platform as a Service (Paas). It utilizes Amazon's cloud services extensively to meet its daily user needs. Cloud is fast, cheap and Scalable. You pay as you go. You get when and as much you need. Availability is guaranteed at peak or burst loads. Netflix runs on top of Cassandra (nearly few milliseconds of network latency). Runs Monkey Testing regularly.



Netflix Initial Operations



Netflix Current Operations

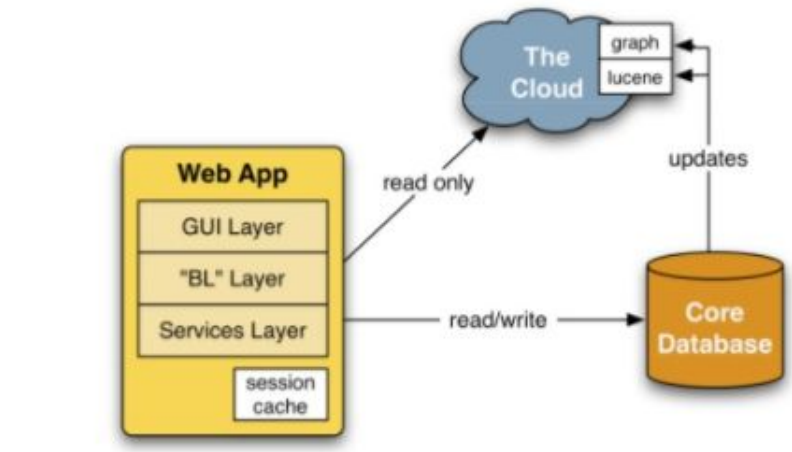


Netflix Current Architecture Design

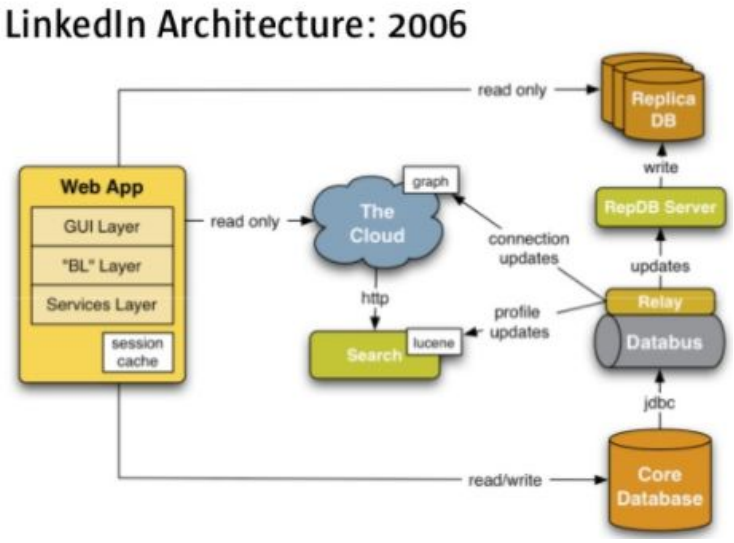
Q10. EXPLAIN WITH EXAMPLE HOW SYSTEM ARCHITECTURE EVOLVES AS NETWORK TRAFFIC GROWS TO MEET SPECIFIC REQUIREMNTS OF THE CLIENT?

Ans.

LinkedIn started in 2003 and evolved through many years as network traffic grew for operations such as profile searches, invitations, connections and messaging among professionals on LinkedIn grew.

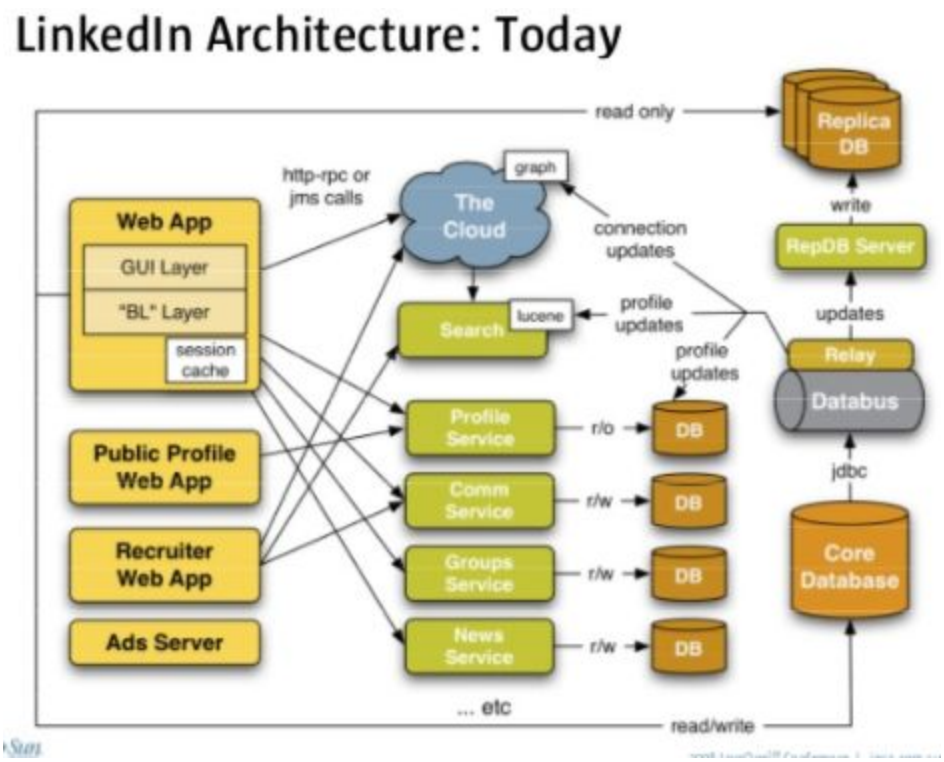


Simple Architecture



LinkedIn Architecture in 2006 (First Profit Year)

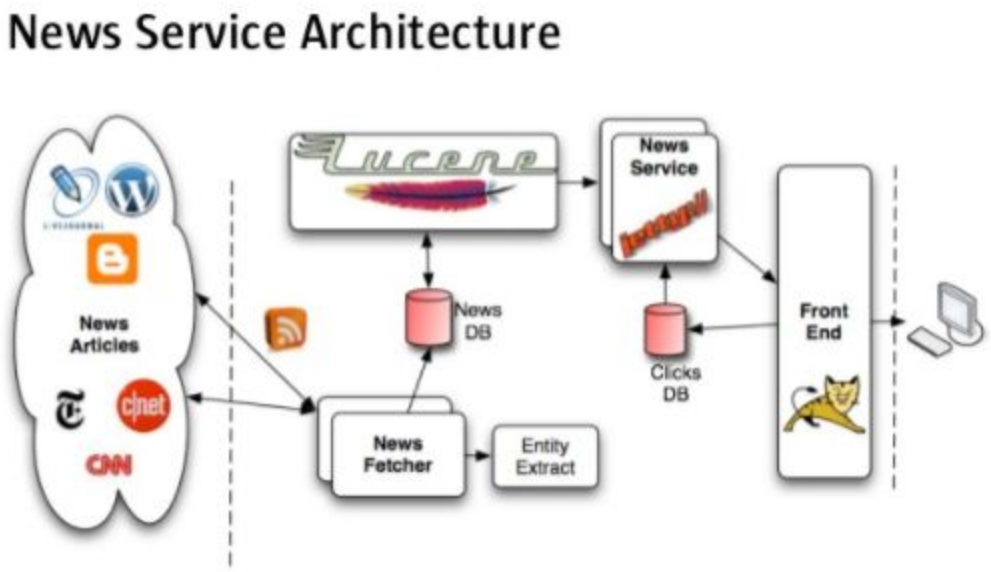
- Graph operations:
  - `findRoute(m1, m2)`
  - `visit(visitor, deg)`
  - `visit(visitor, deg, since)`



Linkedin's Current Service Oriented Architecture -22M users. (Not applicable for small or simple sites)

Benefits of SOA

- ☐ **Each Service can scale independently**
- ☐ **Reduces dependencies**
- ☐ **Encourages Decoupling**
- ☐ **Graceful Degradation of Functionality**
- ☐ **Cloud Service acts as Backend Server Caching the Entire Network**



LinkedIn News Service Architecture – First crawls web for News, Provides Indexing and Searching to serve right news to right people

Q11. Common Mistakes and success measures for Building a Scalable Communication Platform/Network?

Ans.

- 22M members
- 130M connections
- 2M email messages per day
- 250K invitations per day

Daily Stats of a typical large scale communication platform

STEPS INVOLVED: -

I. FIRST LAY THE SERVICES TO BE OFFERED

> The Communication Service
- Permanent message storage
- InBox messages
- Emails
- Batching, delayed delivery
- Bounce, cancellation
- Actionable content
- Rich email content

> The network updates service
- Short-lived notifications (events)
- Distribution across various affiliations and groups
- Time decay
- Events grouping and prioritization

II. DISTRIBUTE THE COMMUNICATION LAYER INTO MESSAGE CREATION LAYER AND MESSAGE DELIVERY LAYER

Message Creation
- Clients post messages via asynchronous Java Communications API using JMS
- Messages then are routed via routing service to the appropriate mailbox or directly for email processing

- Message content is processed through the JavaServer Page™ (JSP™) technology for pretty formatting
- The scheduler can take into account the time, delivery preferences, system load
- Bounced messages are processed and redelivered if needed

message delivery

III. IDENTIFY FAILURES

- **Messages can Bounce**
- **Messages can get lost**
  - **Database Problems**
  - **Bugs in the Code**
  - **Bugs in the content processing of email**
  - **Unavailable Service**

IV. SCALE COMMUNICATION SERVICE

- ⌘ Functional Partitioning
  - ☞ Sent
  - ☞ Received
  - ☞ Archived
- ⌘ Class Partitioning
  - ☞ Member Mailboxes
  - ☞ Guest Mailboxes
  - ☞ Corporate Mailboxes
- ⌘ Range Partitioning
  - ☞ Member ID Range
  - ☞ Email lexicographical Range
- ⌘ Asynchronous Flows

V. TAKE AWAYS

Q12. How Facebook manages a massive ever-increasing growth of 2Billion users, 5Billion API Calls, 20Billion photos and services across the planet?

Ans. FACEBOOK INFRASTRUCTURE: SCALING INTERCONNECTED DATA

Challenges

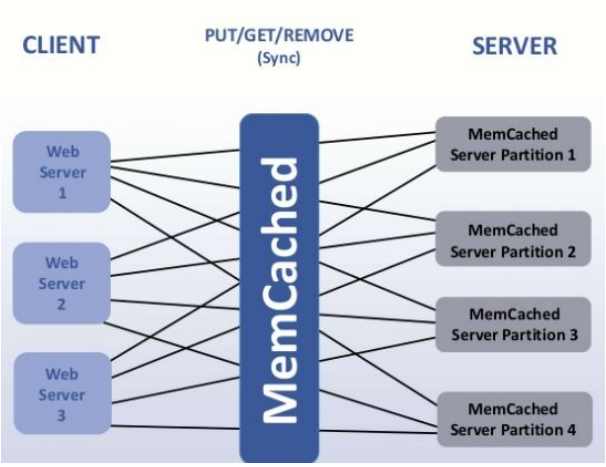
- ❖ **High Concurrency**
- ❖ **High Data Volume**
- ❖ **Multilevel Hierarchical Data**
- ❖ **PHP downside - High Memory Consumption and CPU Usage**
- ❖ **LAMP difficult to Scale**

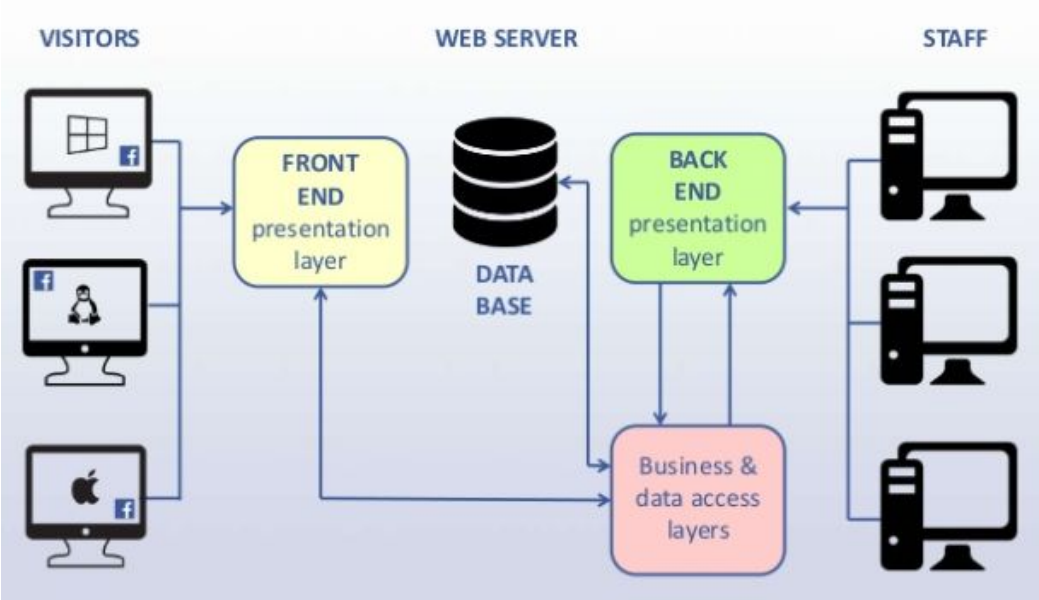
- **Data Management**

User ID	Friends with	User Name	Age	Bio	Interests
1	2,3,4	XYZ	..	..	..
2	1	..	..	..	..

Approach

- **Cache Hot Data**
- **Everything is a hash lookup.**
- **In-memory Distributed Hash Table**
- **Hip Hop Source Code Transformer from PHP to C++**
- **Asynchronous AJAX Pages**
- **MYSQL for Speed and Reliability**
- **LAMP (Linux-Apache-MySQL-PHP) for High Availability, Heavy Duty Dynamic Website**
- **Use of Memcached Caching System to Relieve the Burden of Database and Reduce Reading Time**
- **Hadoop and Hive Ecosystem for Processing, Structuring and Analyzing Large Data Sets**
- **Thrift Protocol for Cross language services development.**
- **Scribe to maintain server log.**
- **HTTP Proxying**
- **Object Storage System Haystack Photo Storage Solution at less cost and high throughput.**
- **Proper use of cookies like for Alerts in case of Security issues, or for improving Facebook experience or showing relevant ads, features and products.**
- **In house project BigPipe**





FRONT END AND BACK END: ***FACEBOOK***

Q13. Describe the 7 stages of Scaling Web Apps?

Ans. STAGE 1: SIMPLE ARCHITECTURE (STARTUPS)

- **Firewall and Load Balancers**
- **Pair of Web Servers**
- **Database Server**
- **Internal Storage**

- No Redundancy
- Low Complexity
- Low Operational Costs

STAGE 1: RELATIVELY SIMPLE ARCHITECTURE (PROFITABLE STARTUPS)

- Low Risk Tolerance
- Add Redundant Firewall incase of failure
- Add Redundant Load Balancers
- Add more web servers for performance
- Optimize the database using indexing techniques
- Add Database Redundancy

STAGE 3: GROWTH MANAGING ARCHITECTURE (FAMOUS STARTUPS)

- Reverse Proxy (Squid, Varnish)
- High End Load Balancer
- Cache Static Content
- Add even more webservers to manage content
- Single Database
- Start Database Replication. Add Master Slave Databases. All writes go to single Master with read only slaves.
- Scale thru Database Replication according to the requirement, if application is read intensive or write intensive.
- Refactor the code if and wherever required.
- Start Maintaining Logs and Improve Coding Standards.

STAGE 4: LARGE SCALE ARCHITECTURE (Multi National STARTUPS)

- Caching with Memcached
- Database Partitioning – Features get their own database
- Choosing between CP and AP as Application demands (CAP theorem)
- Shared Storage for Content
- Re-architecting the App and Database to support load.

STAGE 6: GLOBAL ARCHITECTURE (International STARTUPS)

- Architecting each component for Scale.
- More Database Partitioning based on Geography, Last Name, User Id.
- Creating User Clusters aggregating similar users.
- Scaling Cluster independently. All Features available on each cluster.
- Using hashing scheme or master Database to map user to their cluster.
- Rethinking Entire Application Model / Database Model
- Maintain Source Control Versions.

STAGE 6: MASSIVE ARCHITECTURE (Government, Social Networking)

- Scalable application and database architecture
- Acceptable Performance
- Starting to Add New Features
- Manageable Traffic
- Keep Optimizing the code
- Keep profits higher than consumption.
- Use CDN and Cloud Services.

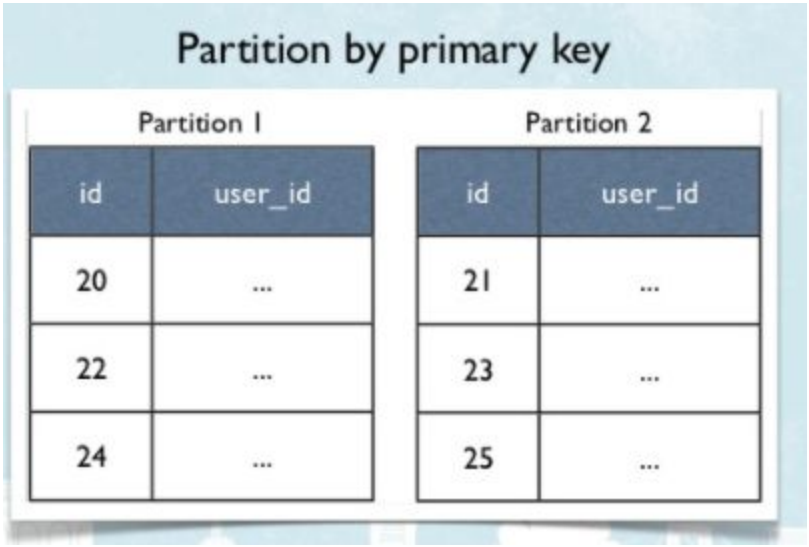
STAGE 7: PLANET WIDE ARCHITECTURE (Banking, Search Engines)

- More Power, More Space, More Bandwidth
- If Hosting Provide Good Enough? Or Build own Multiple Data Centers and Data Grids for faster reach.
- Check for bottlenecks caused by Firewall and Load Balancers.
- Address Bottlenecks related with Storage, People or Processes.
- Any limits incurred by technology.
- Balance Replication data and load balancing geographically.
- Isolate Services. Don't change lots of things at once.
- Think Horizontal Not Vertical. (Not How fast. But How many.)
- Scale Appropriately using Load Testing.

Q14. HOW TWITTER HANDLES BIG DATA IN REAL TIME?

Partitioning, Indexing and Replication.

id	user_id	text	created_at
20	12	just setting up my twttr	2006-03-21 20:50:14
29	12	inviting coworkers	2006-03-21 21:02:56
34	16	Oh shit, I just twittered a little.	2006-03-21 21:08:09



	PK Lookup
Memcached	1ms
MySQL	<10ms*

```
SELECT * FROM tweets
WHERE user_id IN
  (SELECT source_id
   FROM followers
   WHERE destination_id = ?)
ORDER BY created_at DESC
LIMIT 20
```

Original Implementation

Q15. What is Memcache? Why do we need it? What are its benefits and caveats?

Ans. Memcache is an in-memory Key-Value Pairs data store. Used widely for caching.

- **Set Value to a Key**
- **Get Value using Key**
- **Read Frequently and Write Rarely is most suitable to combine with Memcache.**
- **Benefits:-**
  - **Reduce Application Operational Cost**
  - **Increase Application Performance**
  - **Minimizes Request and Response Time (~10X faster)**
  - **Can Cache and handle 90% of requests**
  - **Reduces Load on Datastore**
- **Coordinating Read Usage**

- Check if Memcached value exists
- If yes, display/use it directly
- If No, Fetch the value from Datastore and store in Memcache

✷ **Coordinating Write Usage**

- Invalidate the old value in Memcached
- Write the value to datastore
- Update the Memcached Entry.

✷ **Improve Memcache Performance**

- Use Batch Operations like getAll() or PutAll() for multiple memcache entries.
- Use Atomic Operations to update a value consistently by concurrent requests. Helps Manage Data Consistency in concurrent/ multi-instance environment
- Asynchronous Operation Calls

✷ **Caveats**

- Memcache is Volatile: Entries can be evicted anytime if entry reaches expiration, or Memcache is full or its server failed.
- Memcache is not transactional.
- Cache misses needs to be handled gracefully.
- Too much items in Memcache can make it less efficient.
- Redis another alternative to Memcached offers data to be stored structurally in Set, sorted set, hash table etc.

Q16. HOW TO DESIGN A COMPLEX SYSTEM IN AN INTERVIEW?

Ans.

Steps:

❶ **Infrastructure Requirements Clarification**

- Near Real Time Communication?
- Scale to Process Millions of user requests in a Second?

- ☐ Other Feature Requirements.

❷ Design Requirements Imposed

- ☐ Heavy Read Load?
- ☐ Any Write Load?
- ☐ Data is Transient or Persistent?
- ☐ Need Data Centers to be geographically distributed around the world?
- ☐ Enough Databases to support the Load?
- ☐ Enough webservers to meet fine performance?
- ☐ Data Sharding across the databases?
- ☐ More Memcache servers to reach read capacity?
- ☐ Geographically distributed replica databases for Fault tolerance
- ☐ How to handle write updates - to local replica or cross country master db. Again depends on application being read intensive or write intensive.
- ☐ Level of Decoupling to Allow independent scaling.
- ☐ Keep Persistent Store and Cache Separate to scale persistent data and transactional data independently.
- ☐ Scheme of Cache Invalidation being used.
- ☐ Race Conditions due to Writes or Updates.
- ☐ How to minimize cache misses?
- ☐ How a Cache Miss is handled gracefully?

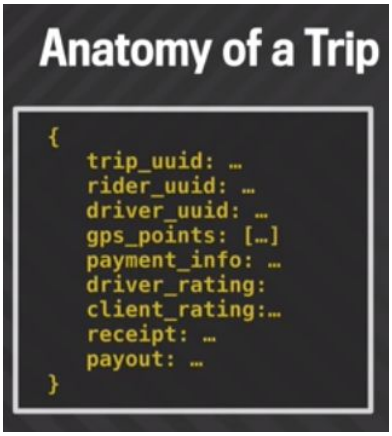
❸ Resource Management

- ☐ Caching Photo/Photo Storage
- ☐ Audio/Video Storage
- ☐ Text Content
- ☐ Graph Connections
- ☐ Messaging
- ☐ Location Data Coordinates etc
- ☐ Key-Value Pairs
- ☐ Clustering/ Replication

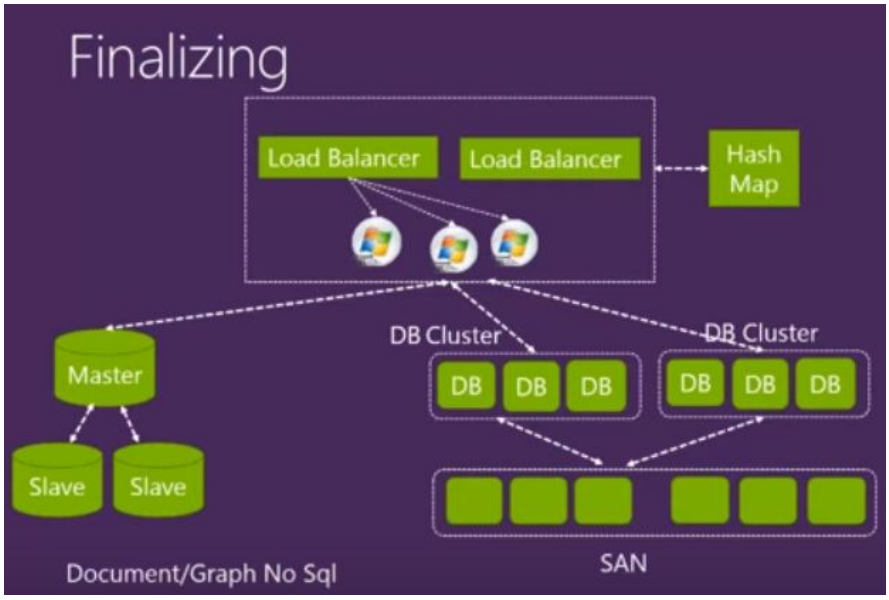
- ☐ **Indexing**
- ☐ **Cloud Justification**

Q17. DESCRIBE THE ANATOMY OF CAB SERVICE APPS?

Ans.



Fare Calculation – Base + Miles + Surcharge
Payment Methods Cash, Card, Wallets
Cab Sharing – Single Traveler, Multiple Traveler
Destination – Single or Multiple
Cab Driver Details – Driver ID, Full name, Cab ID, Miles travelelled, phone
User Details – name, source, destination, phone
Ratings / Comments

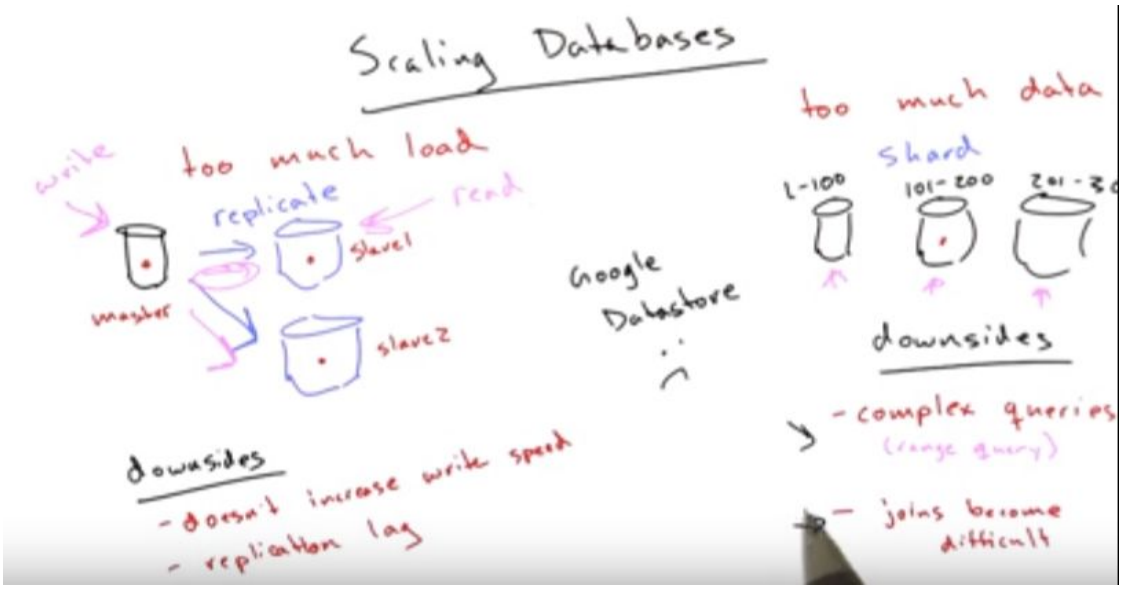


Scaled Infrastructure

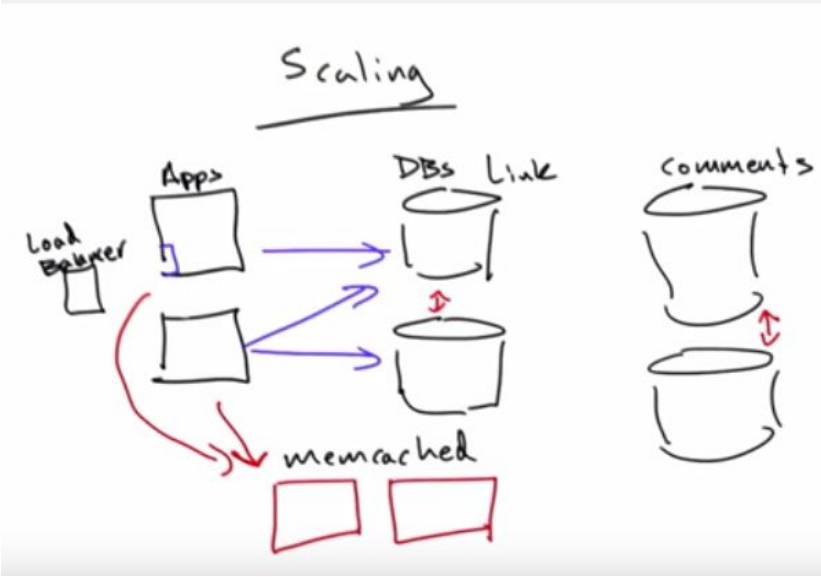
Q18. Example cases of Handling Scaling Databases, Application Servers or Web Application?

Ans.

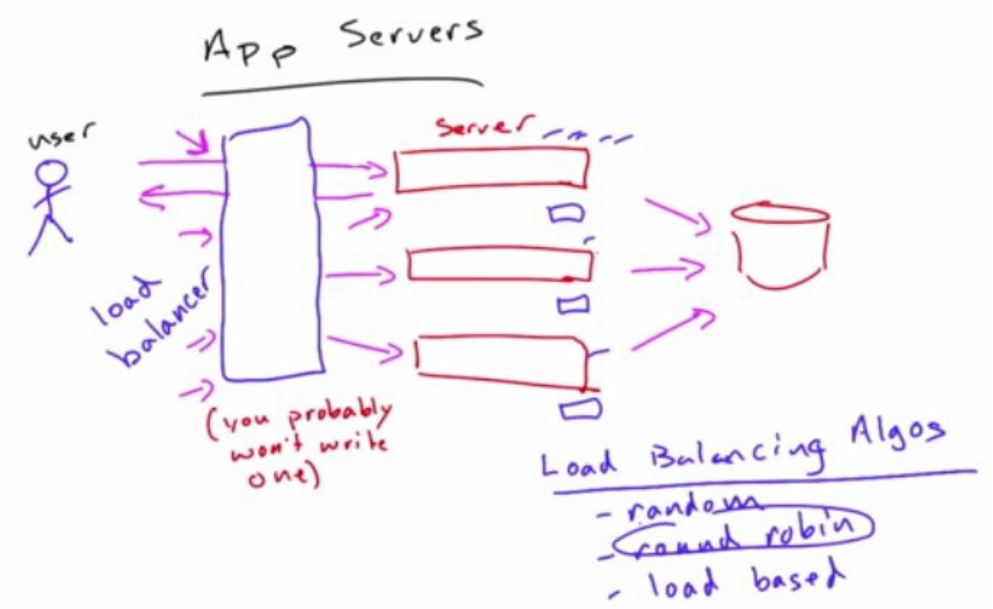
SCALING DATABASES BY SHARDING



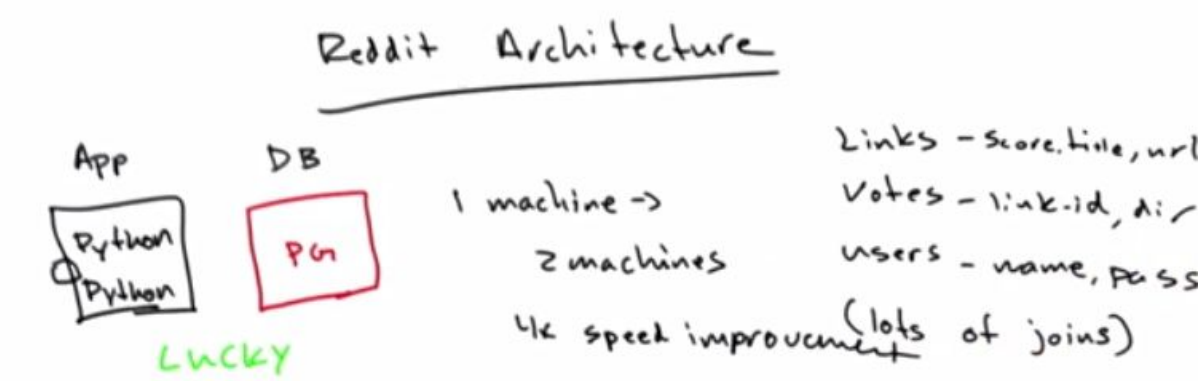
SCALING DATABASES BY FEATURE



APP SERVER SCALING TECHNIQUES



EXAMPLE: REDDIT ARCHITECTURE



INTERVIEW SPECIFIC QUESTIONS

Q19. How to approach a System Design Problem in a specific way?

Ans. System Design Questions check your knowledge and ability to build large scale distributed System step by step. Do you know the constraints? What kind of inputs does your system need to handle? You have to get a sense for the scope of the problem before you start exploring the space of possible solutions. Focus on the architecture and the tradeoffs behind each decision.

Step 1: Requirement Gathering

- **Resolve any ambiguities regarding the system while Clearly Defining the End goals of the system.**
- **Example Queries Related to Twitter**
  - **Who Posts the Tweets (user, logged in user)**

- ? Who reads the Tweets (followers, registered users, publicly available to anyone)
- ? What is the content of the Tweet (video, text, photos, fixed or flexible size constraint)
- ? Who can follow the user?
- ? Can users like tweet or upvote tweet?
- ? Are we storing metadata about the tweet?
- ? What gets included in the user feed? (trending tweets or tweets from everyone whom user is following)
- ? What is order of tweets in feed? (Chronological, trending)
- ? Are tweets searchable?
- ? How many total users are there?
- ? How many daily active users r there?
- ? Are we designing interaction between client and server or the backend architecture as well?

Step 2: System interface definition

- Based on the Requirements gathered, define what APIs are expected from the system
- Example Services from Twitter

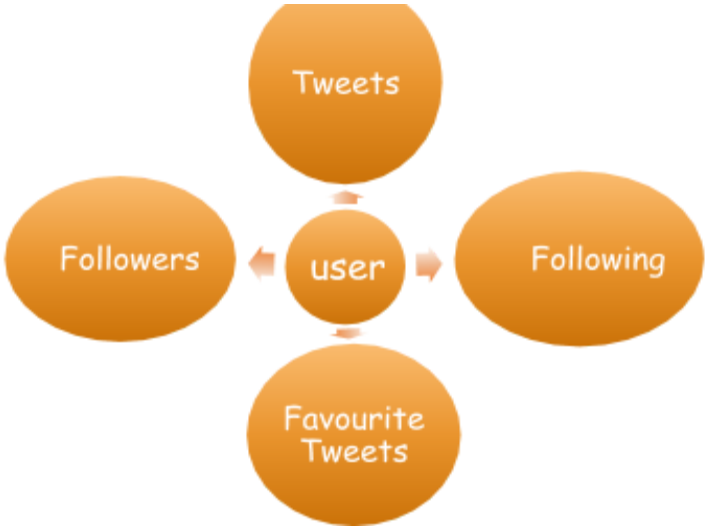
- ***postTweet*** *(user_id, tweet_text, image_url, user_location, timestamp, ...)*
- ***generateTimeline*** *(user_id, current_time)*
- ***recordUserTweetLike*** *(user_id, tweet_id, timestamp, ...)*

Step 3: Back-of-the-envelope capacity estimation

- Estimate the scale of the system to be designed. This information helps during scaling, partitioning, load balancing, caching and meeting read/write network traffic.
- Examples of Gathering System Scale
  - number of new tweets posted per sec
  - number of tweet views per sec
  - how many timeline generations per sec?
  - How much storage would we need per tweet? Video tweets take larger space, than photo tweets or a text tweet. (3MB-30MB)

Step 4: Defining the data model

- Info related to the data flow of the system helps in partitioning later. Identify various entities and how they interact with each other. Also think about storage, encryption, transfer.
- What type of database is best? SQL? or NO-SQL?
- Blob storage for photos and videos where Data is provided in form of large files that are made available in a file system-like fashion that provides elasticity.
- Example of Twitter entities



- **User:** User_ID, Name, Email, DoB, CreationData, LastLogin, etc.

- **Tweet:** Tweet_id, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.

- **UserFollows:** User_ID (FK), Followers_ID, Following_ID

- **FavoriteTweets:** UserID, TweetID, TweetReaction, TimeStamp

Step 5: High Level Design

- Draw a block diagram representing enough core components to solve the system problem from end to end.
- **Examples of client server interaction**
  - **At high level, Twitter would need a few application servers and a load balancer to distribute the traffic load**
  - **If read traffic is a lot more than write traffic, Twitter would need separate application servers for serving read and write requests.**

- **Examples of back end architecture**
  - **At high level, Twitter would need an efficient database that can store all the tweets.**
  - **A Mater slave distributed storage system where Slave databases can be used for handling read requests. And Master DB would handle writes as well as updates on slave database.**
  - **Storing images out of the database is best from scalability point of view. It's a lot easier to manage and manipulate (crop, resize etc.) images on the file system. Separate space for BLOBs also ensure a smaller Database to deal with. However, check for low tolerance to security. Database would be better otherwise.** Reading or writing large files is

faster than accessing large database BLOBS.

- If files are large, more than 1 MB, then a distributed file storage system gives better read/write throughput for photos and videos. For smaller files less than 1 MB, Databases give a better performance.
- Ensure Synchronizing metadata (obj_name, replca_loc) in the database with the file objects in the file system/ file server. And indexing their path in database.

- Lucene or some other in-memory Search to enable searching tweets.

Step 6: Detailed Design of Core Components

- Keeping in mind the system constraints, discuss the tradeoffs while reasoning thru the pros and cons of each approach to implement a component in detail.
- **Examples –**
  - **Think thru the CAP theorem. Brief about Partition Tolerance as a must for a reliable network. And move on to make a tradeoff between CP or AP depending upon the application being read or write intensive or both. While making a move to ACID or BASE approach.**
  - **Decide how the huge data will be distributed thru partitioning – say functionally or by range or geographically. Make sure to mention what issues can be caused if all the data is kept in the same place, while understanding that too much replication can cause lags too.**
  - **Keep Moving to the next challenge and address it through a reasonable approach. For example, How should high traffic users like celebrities should be handled? Say, maintaining a dedicated**

cache repository for such reads using redis or Memcache. Take note that this would serve the purpose by |1| reducing response time for the frequently accessed data and |2| by significantly offloading the database server.

- **Keeping in mind another system constraint of providing recent and trending tweets on the user's feed. Query optimizations should be done while collecting, storing and fetching latest tweets (say, sequential access or random access)**
- **Identify what components need better load balancing. And does any provision exist for maintaining data on cloud or CDN.**

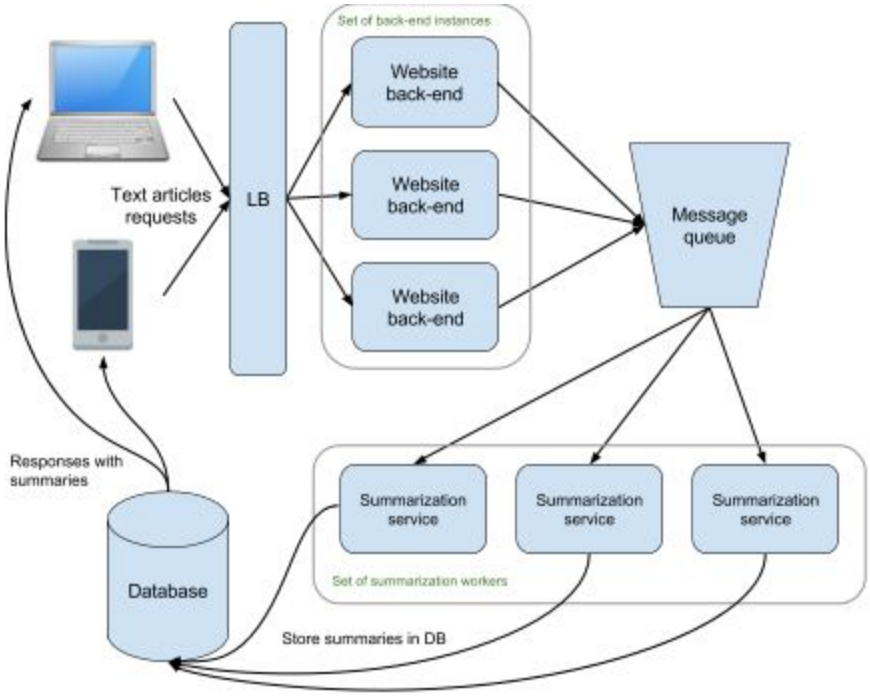
Step 7: Identifying and Resolving Bottlenecks

- Finally, make sure to discuss a few bottlenecks and suggest approaches to mitigate them.
- **Examples –**
- **Identify any single point of failures and mitigate them.**

- How are we handling fault tolerance? Is partition ensuring a fully functional site in case of few server instance failures.
- Do we have backups or replication to ensure data management?
- Does system degrade gracefully in case of major failures.
- Do we issue any alerts in case of a shutdown or degradation?
- How are we logging and monitoring the performance of the services?

Q20. Draw a sample architectural design depicting a working system for a text summarizing app?

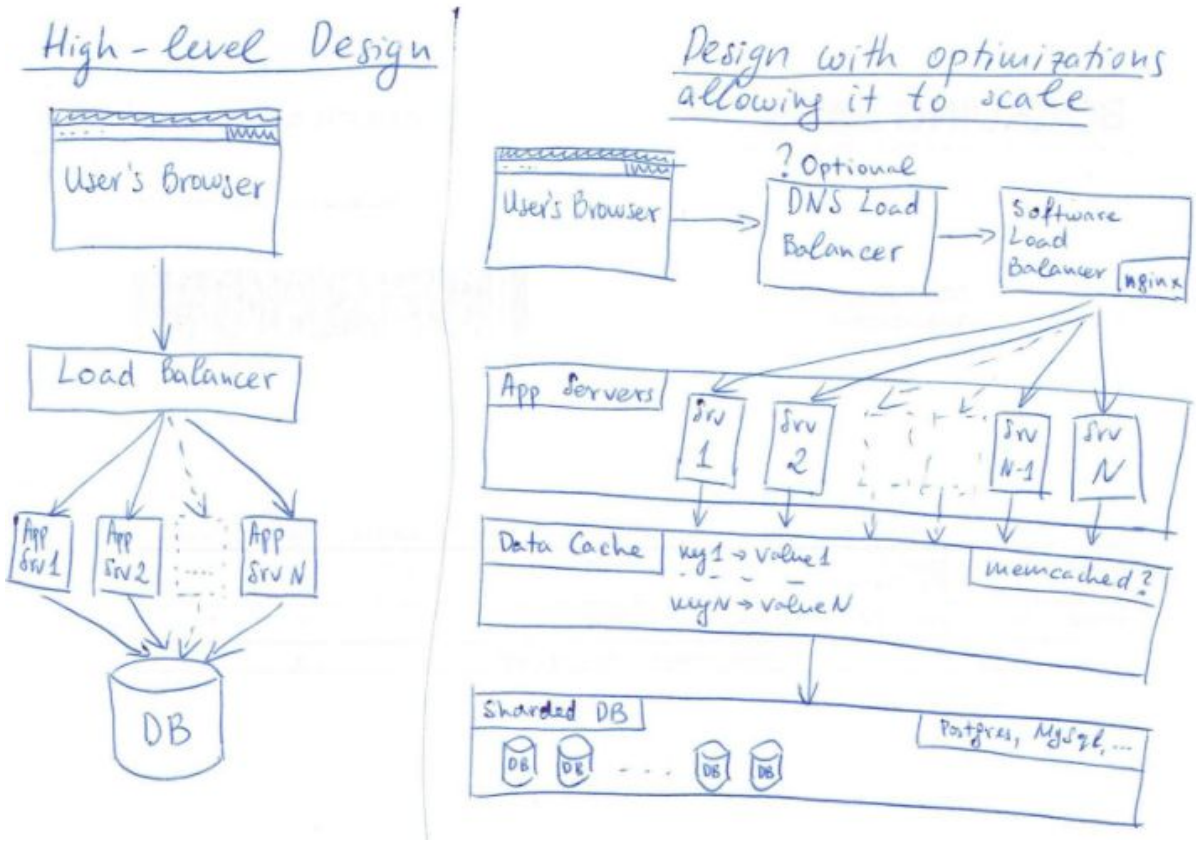
Ans. The upper bound of the size of each text resource is set to >1 MB.



A mobile/web app that takes text articles and uses a summarization algorithm and responds with a brief summary of the article.

Q21. ***Design a simplified version of Twitter where people can post tweets, follow other people and mark favorite tweets.?***

Ans.



Ref. *https://www.hiredintech.com/classrooms/system-design/lesson/72*

ENDS HERE