HLD - System Design

Type	Year-long goals
Planned for	Q1, Q2
Status	In progress
Tags	Career

Resources:

- System Design Roadmap (whimsical.com)
- system-design-primer/README.md at master · donnemartin/system-design-primer (github.com)
- Udemy→ Mastering the system design → Offline
- Grokking system design pdf

Aa Name	Tags	Date
long polling vs websockets vs server-sent events	networking	
TinyURL System Design	sd-interview-problems	
Design Strategies	design-interview-strategies	
Define Requirements	design-interview-strategies	
Design Interview → Working Backwards	design-interview-strategies	
Cloud Computing	working-with-big-data	
Apache Spark	data-analytics working-with-big-data	
Message Queues	working-with-big-data	
DSA Review	DSA	
Distributed Storage Solution	designing-system-that-scale	
Resiliency	designing-system-that-scale	
Caching and CDN	designing-system-that-scale	
ACID Compliance and CAP Theorem	designing-system-that-scale	
Data Lakes	designing-system-that-scale	
Scalability	designing-system-that-scale	
Failover Strategies	designing-system-that-scale	
Database Sharding	designing-system-that-scale	

Few Terms:

Website: Read only

Web Application: Read and write

Stateless server:

Subsequent requests should not depend on something stored on that server from a previous request. DB can store stuff but the server itself shouldn't because we have no idea where the previous request got routed to!

Stateful:

Can have info about previous requests on a given server.

Latency: Time taken by a server to process the request i.e. network delay + computational delay.

sol: 1. Caching

2. CDN
3. Upgrade Systems

Throughput: amount of data transmitted or information flowing through a system per second. Unit: bps(bits per sec)

Low thp reason:

1. Latency
2. Protocol Overhead
3. Congestion(large number of requests)

Sol:

1. CDN
2. Caching
3. DS (Distributed System)
4. Use LB
5. Upgrade Resources

Availability(replication vs redundancy):

Fault Tolerance ↔ HIgh Availability

how to increase HA:

1. Replication: includes redundancy but synchronization of nodes (in db). Types: Active-Active(read-write both), Active-Passive(Master-Slave) (read-write by master only)
2. DS
3. Redundancy: duplication of nodes in case some of them are failing. Types: Active(which is currently providing service) and Passive(at stale).
4. Active -Passive Replication**:**
    a. Master-Slave
    b. Master-Master

Consistency(strong vs eventual): If all the requests are getting the same output regarding of whom accessing it, is called consistency.

Factors to improve c:

1. Improve network bandwidth
2. Stop the dirty read operation
3. Replication based on distance aware strategies(keep the servers close)

Types of consistency:

1. Strong C: when system doesn't allow read operation until all the nodes with replicated data are updated. ex: train/flight seats.
2. Eventual C: We don't stop read operation when nodes are getting replicated. Some user will get old data and some new but eventually all data will be updated to latest after some time. ex: social media

3. Weak C: No need to update

Lamport Logical Clock: To overcome the different timezone clock and find out the sequence of events

doc: Lamport's logical clock - GeeksforGeeks

File Based Storage System:

Where data is stored in form of file.

Challenges:

1. Data Redundancy: update/delete anamoly
2. Poor Security
3. Slow

RDBMS:

A software for RDB which stores data in tables.

1. No data redundancy and inconsistency
2. Data Searching (built in using queries)
3. Data Concurrency (locking system to prevent abnormalities)
4. Data Integrity(add some constraints)

Prob:

1. Rigid Schema
2. High Cost
3. Scalability Issue (sharding is very difficult)

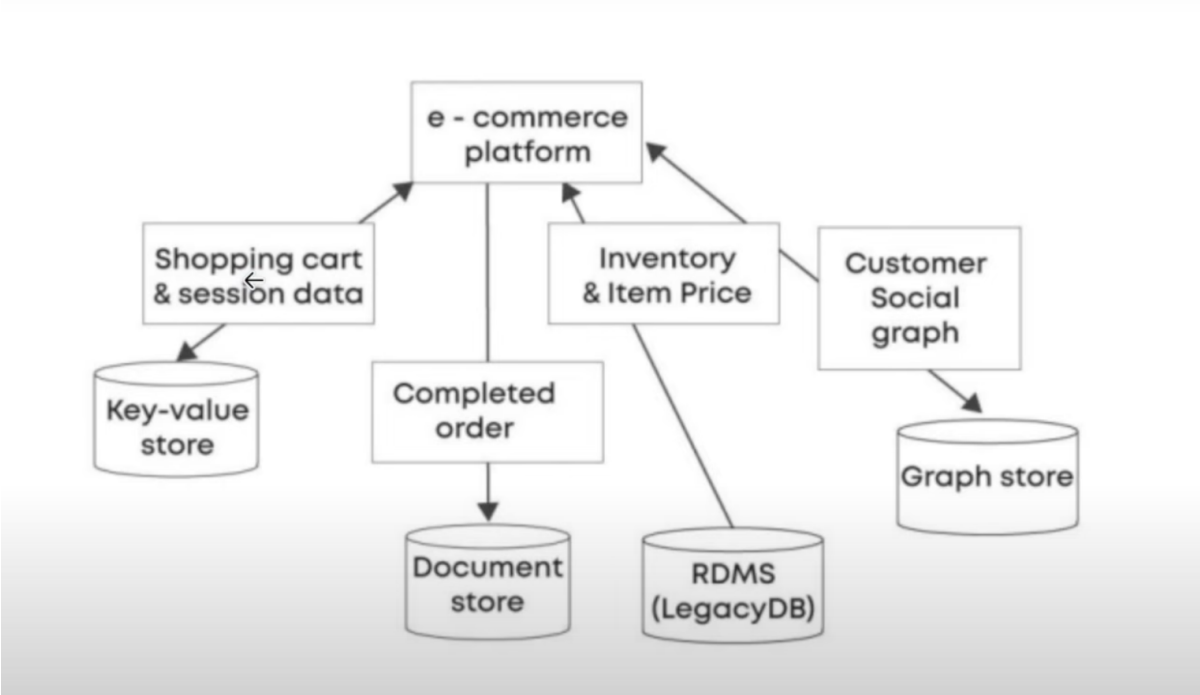
NO-SQL

1. Key Value db : stores in the form of key and value. ex: cache db i.e Redis db. [Use when everything is simple]
2. Document db: best of both RDBMS and No-SQL. relationship concept from RDB and dynamic schema and scaling from NoSql. ex: MongoDB [Use when nested items]
3. Columnar db: columns are stored together instead of rows. Aggregation is fast in such db. used for data analytics, machine learning. ex: cassandra [use when machine learning]
4. Graph db: tores entities and relationship in the form of graph. ex: social network. [use when social graph]

ex: shopping cart: Key -value(Redis), Map→ Graph, Player-score: key value, machine learning→ Columnar db
Payemnt: need high consistency→ use RDBMS

Polyglot Persistence:

When a single db is not able to fullfill application requirenment and we use multiple db called Polyglot Persistence/System



Normalization: Splitting the data into multiple tables to avoid redundancy.

DeN: Keeping all the data into single place.

Indexing:

Organizing the searching / sorting the things i.e implement binary search in tables. We can reduce TC from O(N) to O(logN)

Def → Indexing creates the lookup table with the column and the pointer to the memory location of the row, containing the column.

DS: B-Tree data structure is used for indexing as it is multi-level format f the tree based indexing which has balanced binary search tree.
Use case:

1. Read intensive db only.
2. Not use with write intensive: 2 times entry→ main table + entry in indexing table + reshuffling/sort indexing table.

Communication:

1. Synchronous
2. Asynchronous

Message Based Communication:

1. P2P model: peer-to-peer like sending an email
2. Pub-Sub model: One consumer but multiple subscriber

Tools: Kafka, RabbitMQ

Communication Protocol:

1. Push: s2c, After Long: It is like I subscribed and server will send response when new data available. It is server to client conn.
   The client open the connection with server and keeps it always active.
   The server pushes new events to the the client. This method reduces the server load.

   Sent notification even if the website is not open.
   Prob: what if server sends response at a time when i don't need it.
2. Pull/Polling ⇒ c2s, Client request, Server Respond (ask and get). Prob: asked 100 req but fulfill only 5, then comes long polling
3. Long Polling ⇒ c2s
   Client request, Server keep it open until it gives response. ⇒ ask 100, gives 5 now and keep open channel and provides once available.
   Prob:
   a. Ordering issue
   b. Server will always keep running
4. Socket ⇒ When we need continuous and frequent connection. 2-way channel conn. ex: chat application

5. Server sent events(SSE or WebSockets) ⇒ Ex: Cricbuzz
   Client subscribes to the server ‘stream’, and the server will send message (’stream f events’) to the client until the server or the client closes the stream.

   Works as long as User is using the website.

   a. One way connection

   b. Long Lived Connection

Authentication vs Authorization:

Authentication: who are you?

1. Basic Auth: For log-in, send user-name and password each time to the server in clear text
2. Token Based Auth: First time generate a token with username and password and with every subsequent request use token.
3. OAuth: ex: ex: sign up with google/Facebook on an app.

Authorization: What can you do?

Proxies:

A Proxy server is a hardware or a software hat is placed between the client and the application to provide intermediate service in the communication.

The Proxy server provides a gateway between the user and the internet.

1. Forward Proxy: Client → proxy server → Real Server
   FP hides the Real server identity. proxy server act as a client for Real server.
2. Reverse Proxy: Load balancer connect with multiple server but we send request for LB ip only. Here server is hiding. Ex: API Gateway, Load

Balancer.
RP hides the server.

Scalability:

Two types:

1. Horizontal Scaling
2. Vertical Scaling

Monolithic Architecture:

- Single Server Design
  - Single point of failure
  - Good for small website
- Single Server Design with Database scale-out
  - Database load splitter
  - Scale independently
  - Little better Resiliency
  - Still a Single Point of failure

Distributed System:

Vertical Scaling:

- Bigger Server (More Hardware)
- VS has its limit, it can’t extend infinite
- Still a Single point of failure

Horizontal Scaling:

- Multiple Server
- Load Balancer distributing the load on each server
- Not a Single point of failure
- Can scale infinitely
- More stuff to maintain
- This is easier if servers are stateless

DNS Server:

1. Load Balancing with BIND
2. Round Robin
3. Cause *Celebrity Problem*
4. Browser cache dns info
5. cache can cause CP

Good Solution: Let LB decide whom to send request

LB Problem:

1. Session problem → might need to be logged in on all servers
2. store session somewhere → new file server (but single point of fail)

LB types:

1. Round Robin(static→predefined)
2. Weighted Round Robin(static)
3. IP Hash Algo(static)
4. Source IP Hash(static)

5. Least connection algo: least active conn. (dynamic→ runtime)
6. Least Response time (dynamic→ runtime)

Failover Strategies

Servers:

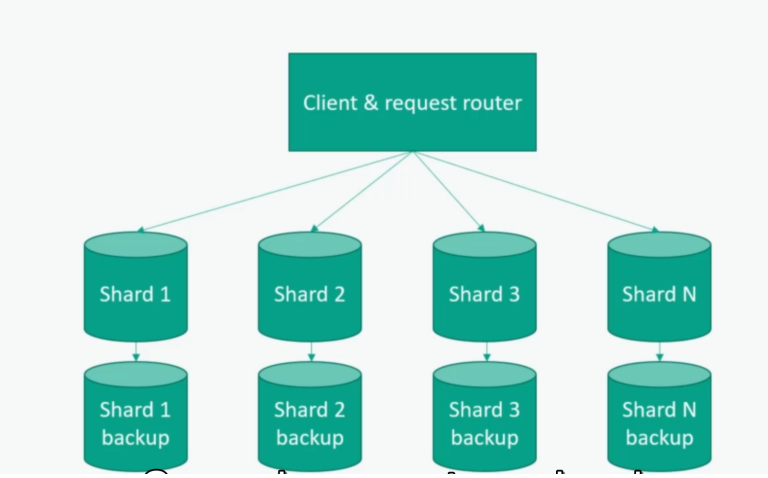
1. On Premise Data Center
2. Cloud Provider ex:EC2
3. Serverless ex: Lambda, Athena

Database:

Vertical Scaling

1. Cold Standby: A standby database ready to take the place of failed db, but needs to backup data into the new server which we already stored at some place periodically.
   manually intervention is needed.
2. Warm Standby: Constantly backing up the data into the new server. Instantly ready to take the place of failure db. **Replication**.
3. Hot Standby: Write data simultaneously into all backup servers. No replication is needed.

Horizontal Scaling:



Shards: Horizontal Partitioning of db

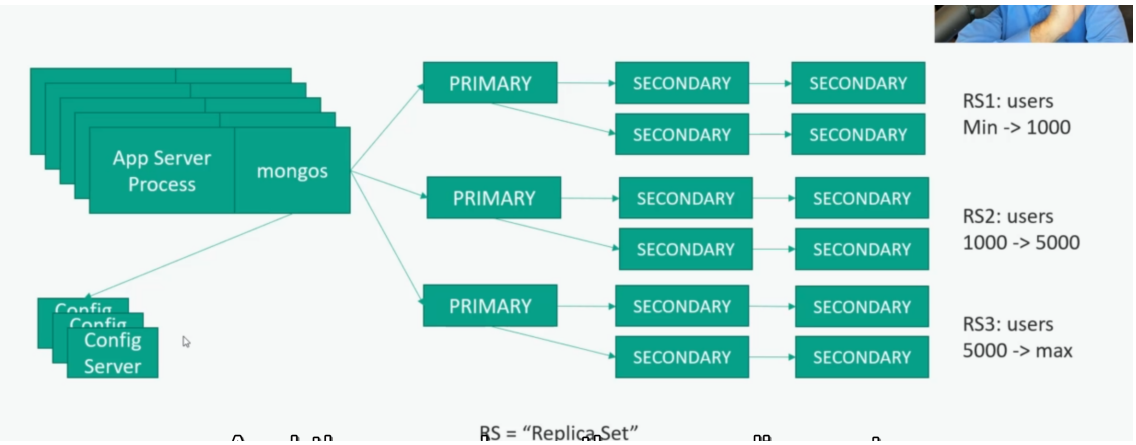
At the top, we have a router that sends requests to each shard.

Each shard can have a backup DB: like a hot standby

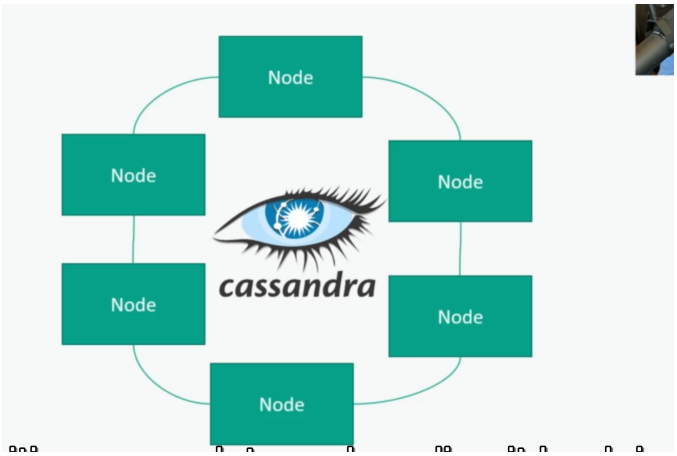
Having a good hash fn can reduce joins and complex SQL queries.

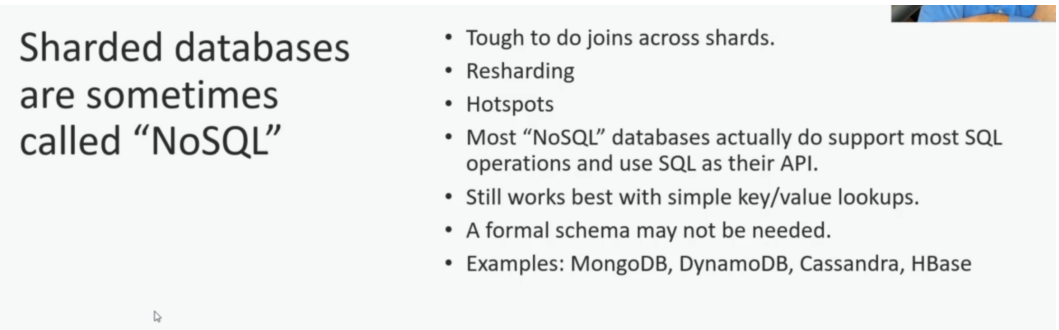
Examples:

1. MongoDB:
    - No-SQL DB
    - It is a highly scalable database.
    - No single point of failure, and Multiple config servers too.
    - It can have multiple replication backup db across multiple regions or data centre



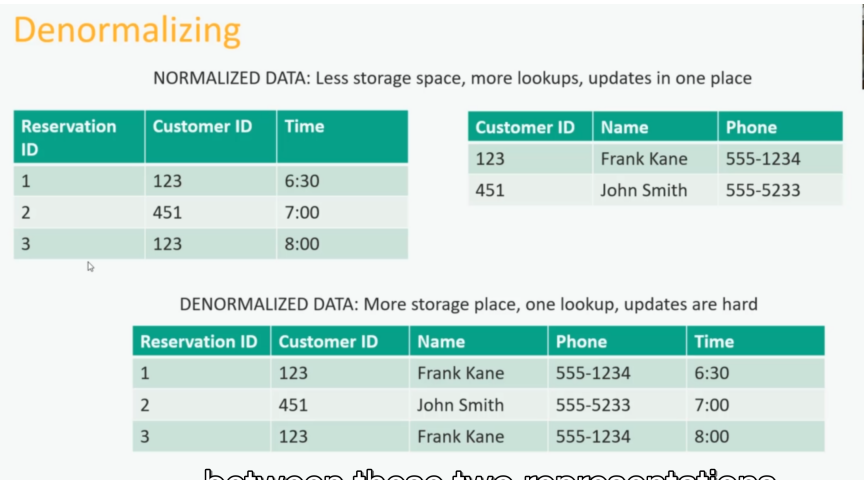
2. Cassandra
    - No Single point of failure
    - Any node can serve as the primary node
    - Data might not be consistent because data needs to be written to all nodes and that could take time. So if we can have some time gap b/w reading and writing, then it is a good choice.



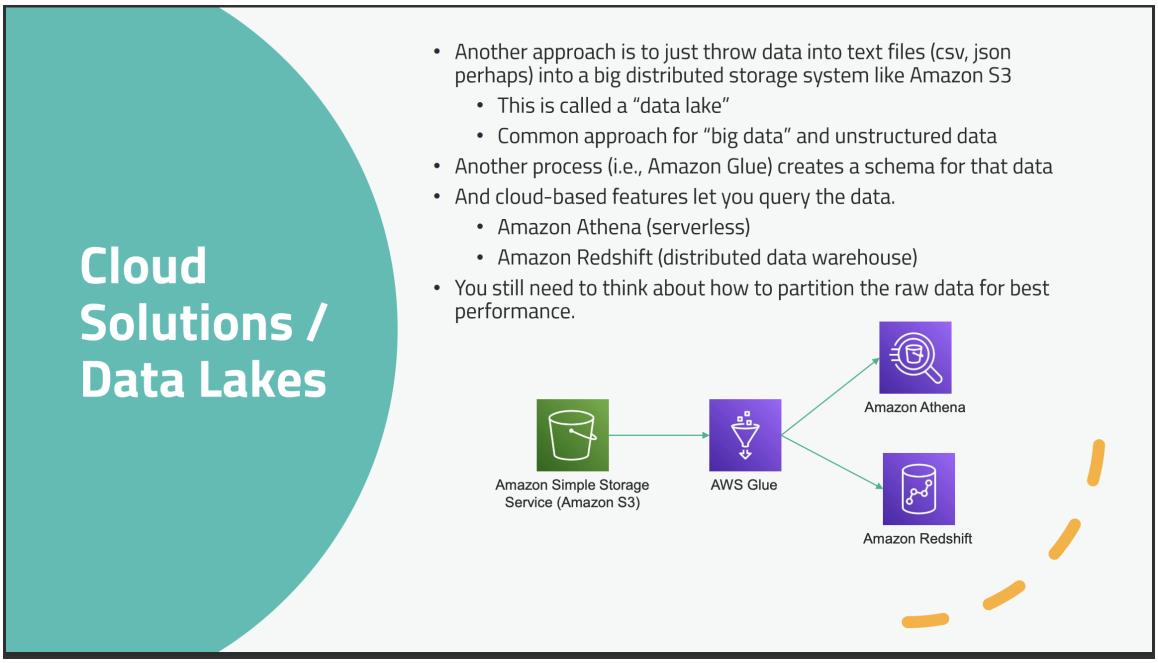


1. Sharding Databases are sometimes called “NoSQL”.
2. Resharding: Once we scale up, we need to redistribute data to other DB, on what basics and how much etc.
3. Hotspot: *Celebrity Problem* → what if some of the data is getting hit harder than other sections, how to redistribute db on actual traffic?

Most Large-scale databases normally use *denormalised* data(All data in one place, no partition of tables like relational databases). One single hit is required. Otherwise, two or more in relational db to join the table and all.
Finally, it depends on customer experience whether we require more updates of the database or more hits.



Data Lakes



Redshift has a thing called *redshift spectrum* which can directly query S3.

Here is a simple system design solution:

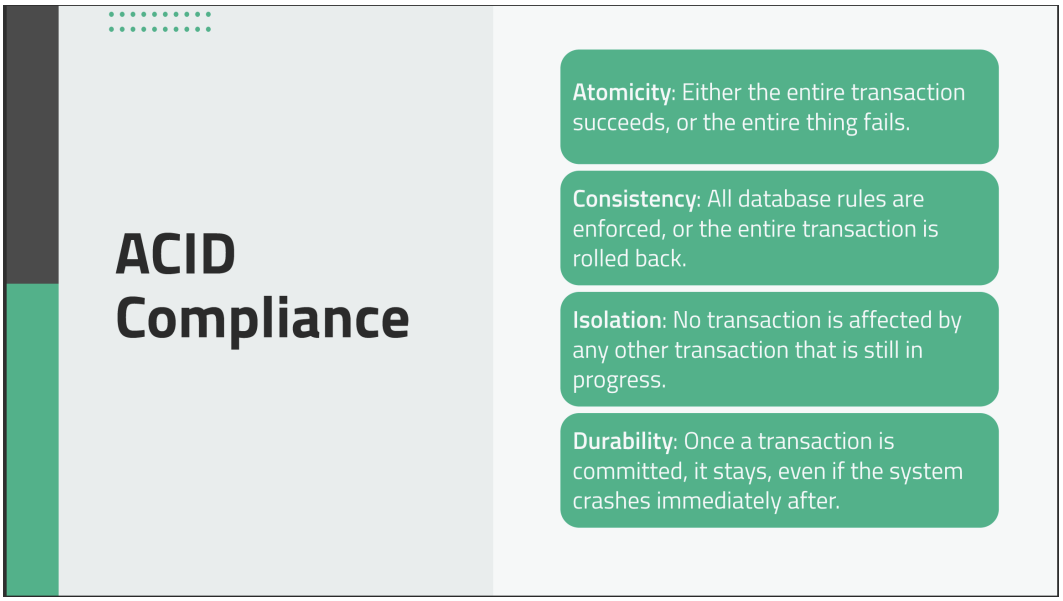
You have unstructured data in a data lake like S3.

You can organize it using **amazon glue** which creates schema out of data lake.

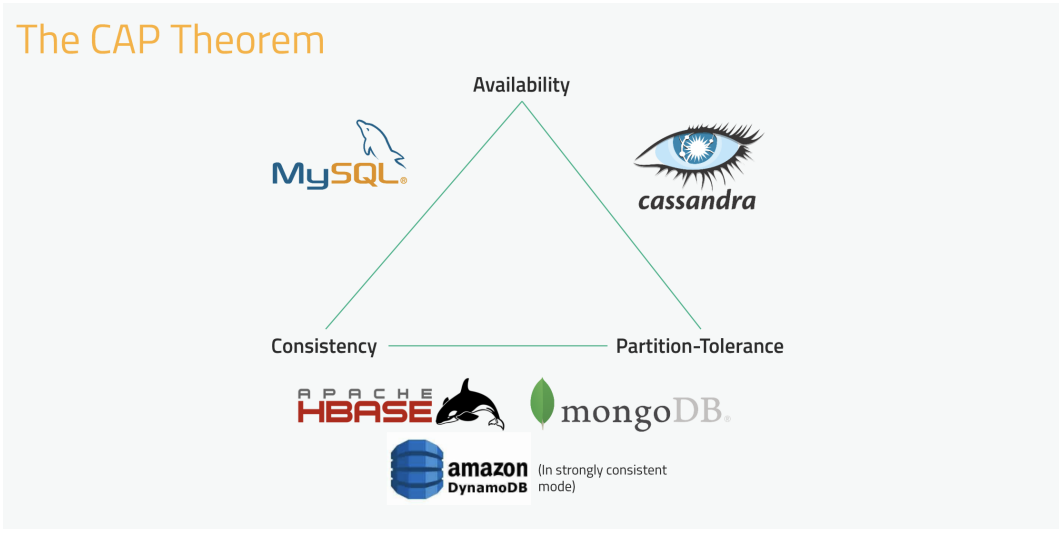
We can query database created by glue using amazon Athena or Amazon Redshit.

ACID Compliance and CAP Theorem

ACID Properties of database:



CAP Theorem:



You can only have 2, not three.

Consistency: Do I get the data right away from what I just wrote?

Availability: Is my system a single point of failure, even for a very short period?

Partition-tolerance: A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition

tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.
Now, if we want system to work in partition tolerance situation, we need to choose either A or C. If we choose C, we need to stop dirty reads hence we are negotiating A but if we choose A, then C won't be there between nodes.

Most modern databases provide the best of all three. But it would help if you thought of trade while deciding the db for your system.

P is the most important in CAP theorem. Now, all decision comes to choose between A and C.

note:
If you want A and C, that system will always be centralized.

Choosing DB using CAP:

Mongo DB: trading off A → single master, single point of failure

Cassandra: trading off C → No single master, No single point of failure but it takes time to replicate data across all nodes.

MySQL: trading off P → not easy to Scale traditional DBS

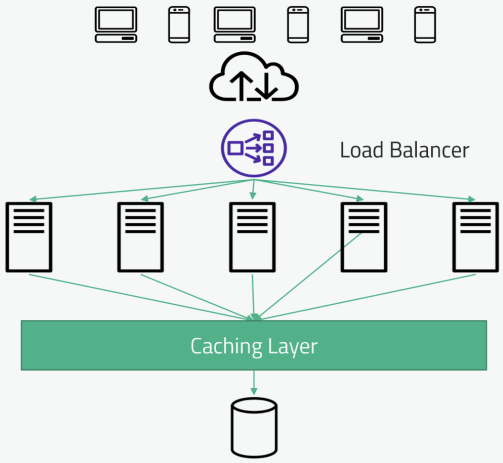
Caching and CDN

Servers can have their cache memory but that might not be enough every time.
Use: when website is read intensive or static content.

2 types:

1. In memory/Local Cache
2. Distributed Cache

Caching Layers

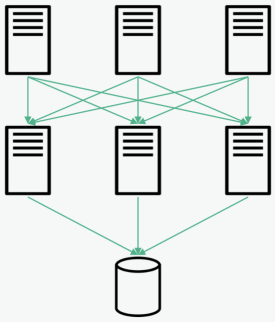


A Few Caching Technologies

Memcached	Redis	Ncache	Ehcache	ElastiCache
▪ In-memory key/value store ▪ Open source.	▪ Adds more features ▪ Snapshots, replication, transactions, pub/sub ▪ Advanced data structures ▪ More complex in general	▪ Made for .NET, Java, Node.js	▪ Java ▪ Just a distributed Map really	▪ Amazon Web Services (AWS) solution ▪ Fully-managed Redis or Memcached

How Caches Work

- Horizontally scaled servers
- Clients *hash* requests to a given server
- In-memory (fast)
- Appropriate for applications with more reads than writes
- The *expiration policy* dictates how long data is cached. Too long and your data may go stale; too short and the cache won't do much good.
- *Hotspots* can be a problem (the "celebrity problem")
- Cold-start is also a problem. How do you initially warm up the cache without bringing down whatever you are caching?



BY FRANK KANE

Sundog Education

SLIDE 29

Eviction Strategies for Caching

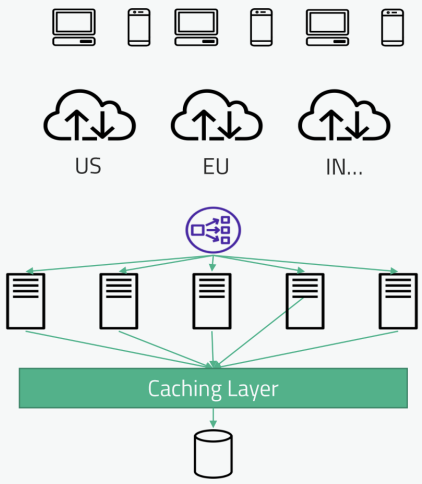
LRU: Least Recently Used
LFU: least Frequently used
FIFO: First In First Out
MRU: Most Recently Used
MFU: Most Frequently used
LIFO: Last In First Out
RR: Random Replacement

- [ ] TODO: Look for the LRU Cache code

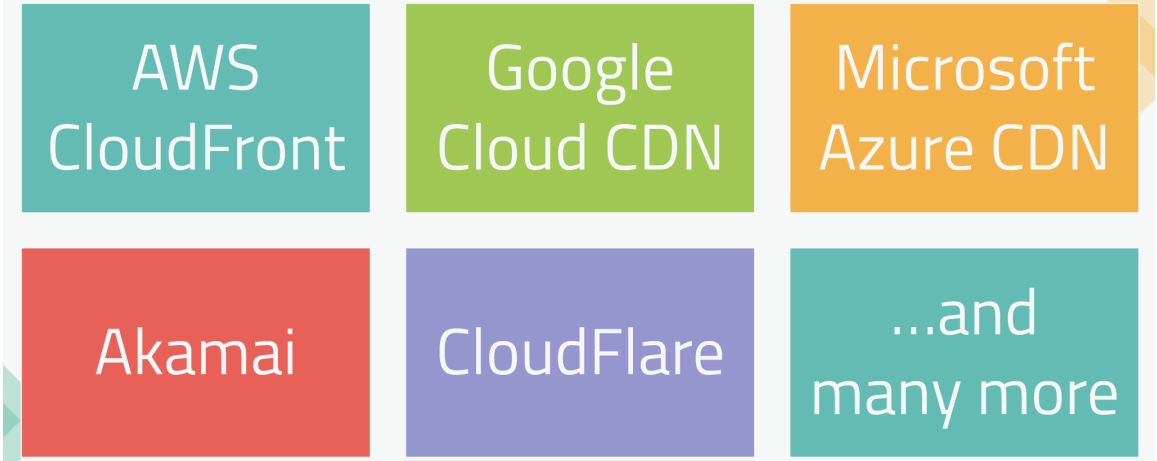
CDN: static content

Content Delivery Networks (CDNs)

- Geographically distributed
- Local hosting of
  - HTML
  - CSS
  - Javascript
  - Images
- Some limited computation may be available as well
- Mainly useful for static content, such as images or static web pages.
  - You probably won't be asked to design a static web page, though!



CDN Providers



Resiliency

Resilience to Failure → How to handle failure on small or large scale

Data centre → Availability Zones

A Server, an AZ or an entire region could go down.

Be smart about distributing your servers

- Secondaries should be spread across multiple racks, availability zones, and regions
- Make sure your system has enough capacity to survive a failure at any reasonable scale
  - This means overprovisioning
- You may need to balance budget vs. availability. Not every system warrants this.
  - Provisioning a new server from an offsite backup might be good enough.
- Again, ask questions!

Overprovisioning → have excess capacity to handle any kind of failure

Distributed Storage Solution

Distributed storage solutions

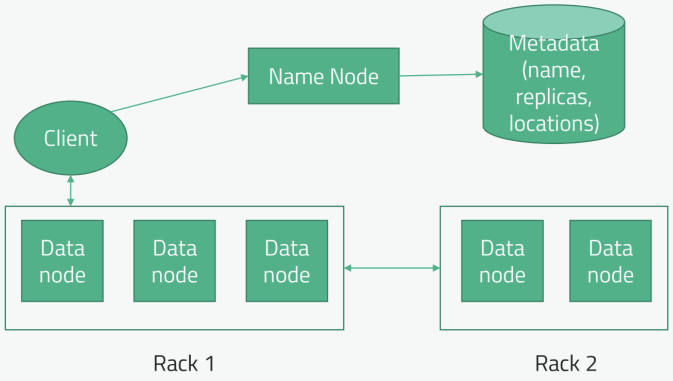
- Services for scalable, available, secure, fast object storage
- Use cases: “data lakes”, websites, backups, “big data”
- Highly durable:
- Amazon S3 offers 99.999999999% durability!

- What do we mean by 99.999999999% durability?
  - This is a *percentile*
  - This one is “11 9’s” of durability
  - Meaning: there is a 0.000000001% chance of losing your data with S3.
- This can also be applied to *latency*, or how quickly a service responds to a request.
  - For example: you can say your “3 nines” latency is 100ms, meaning that 99.9% of requests come back within 100ms.
- Availability SLA’s can be deceiving…
  - 99% availability would still result in 3.65 DAYS of downtime in a year
  - Whereas 99.9999% (6 nines) would result in about 30 seconds of downtime

Distributed storage solutions

- Amazon S3
  - Pay as you go
  - Different tiers, ie Glacier for archiving is cheaper, but harder to read from. You can also choose the amount of redundancy you need to save money.
  - Hot / cool / cold storage
- Google Cloud Storage
- Microsoft Azure
- Hadoop HDFS
  - Typically self-hosted
- Then there are all the consumer-oriented storage solutions
  - Dropbox, Box, Google Drive, iCloud, OneDrive, etc.
  - Generally not relevant to system design

Example: HDFS architecture



- Files are broken up into "blocks" replicated across your cluster
- Replication is rack-aware
- A master "name node" coordinates all operations
- Clients try to read from nearest replica
- Writes get replicated across different racks
- For high availability, there may be 3 or more name nodes to fall back on, and a highly available data store for metadata

Message Queues

single consumer → Aws SQS

pub/sub → Aws SNS

Message Queues as a scaling tool

Publishers / Producers

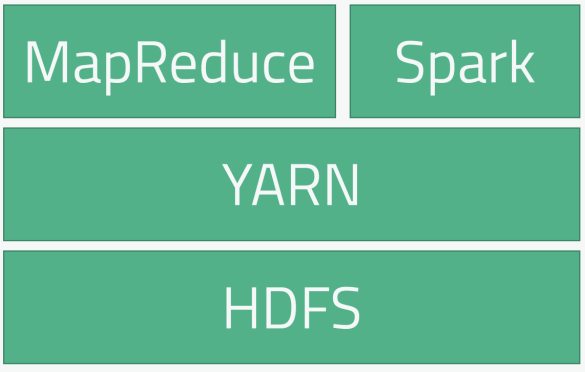
Subscribed consumers

- Decouples producers & consumers
- So if the consumers get backed up, that's OK.
- Example: Amazon SQS service
  - Single-consumer vs. pub/sub
- This is different from *streaming* data (generally real-time, massive data)

Apache Spark

> What is Spark? - Introduction to Apache Spark and Analytics - AWS (amazon.com)
> Transform or Analyse Large Amount of Data → Apache Spark

Apache Spark

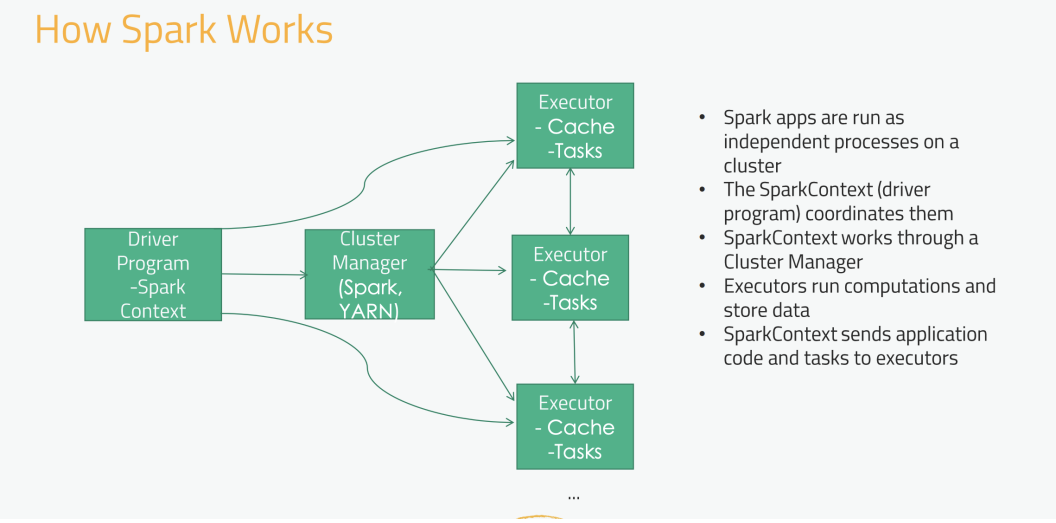


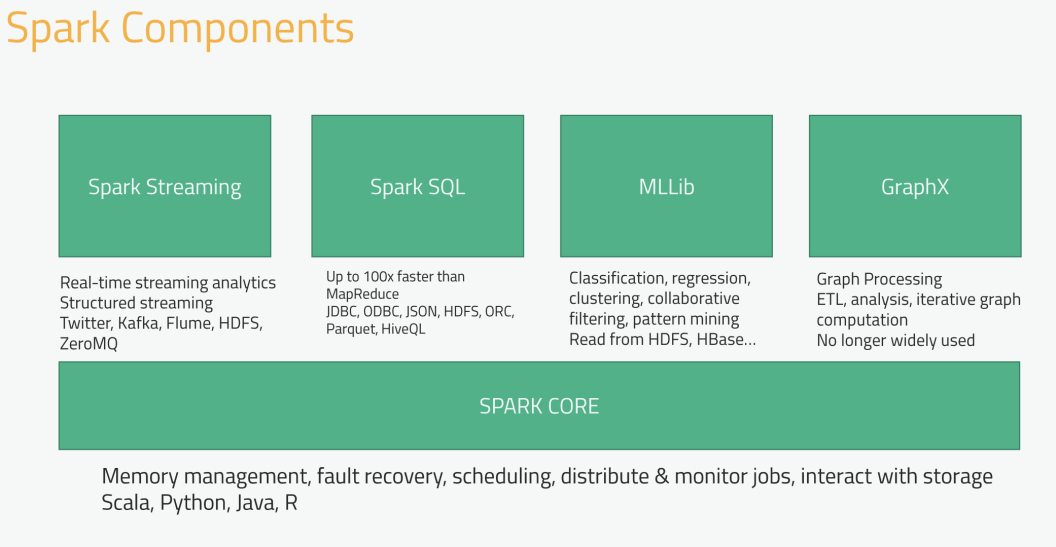
- Distributed processing framework for big data
- In-memory caching, optimized query execution
- Supports Java, Scala, Python, and R
- Supports code reuse across
  - Batch processing
  - Interactive Queries
    - Spark SQL
  - Real-time Analytics
  - Machine Learning
    - MLLib
  - Graph Processing
- Spark Streaming
  - Integrated with Kinesis, Kafka, on EMR
- Spark is NOT meant for OLTP

Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters1. It's a unified engine for large-scale data analytics1.
Here are some key features of Apache Spark:

- **Batch/streaming data**: Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java, or R1.
- **SQL analytics**: Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting1.
- **Data science at scale**: Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling1.
- **Machine learning**: Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines1.

Apache Spark is used by thousands of companies, including 80% of the Fortune 500, and has over 2,000 contributors to the open source project from industry and academia1. It integrates with your favorite frameworks, helping to scale them to thousands of machines1.





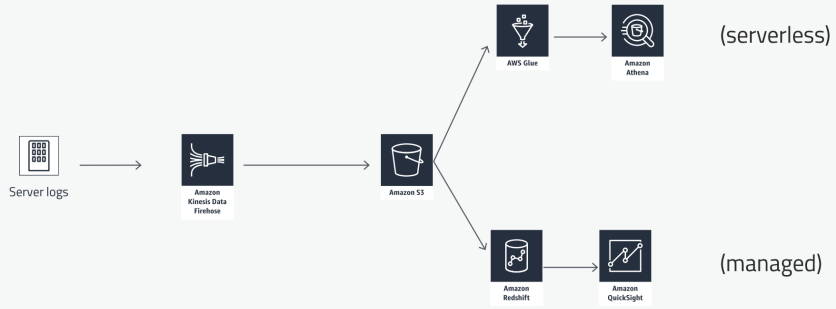
Cloud Computing

A Brief Review:

	Amazon Web Services (AWS)	Google Cloud	Microsoft Azure
Storage	S3	Cloud Storage	Disk, Blob, or Data Lake Storage
Compute	EC2	Compute Engine	Virtual Machines
NoSQL	DynamoDB	Bigtable	CosmosDB / Table Storage
Containers	Kubernetes / ECR / ECS	Kubernetes	Kubernetes
Data streams	Kinesis	DataFlow	Stream Analytics
Spark / Hadoop	EMR	Dataproc	Databricks
Data warehouse	Redshift	BigQuery	Azure SQL / Database
Caching	ElastiCache (Redis)	Memorystore (Redis or Memcached)	Redis

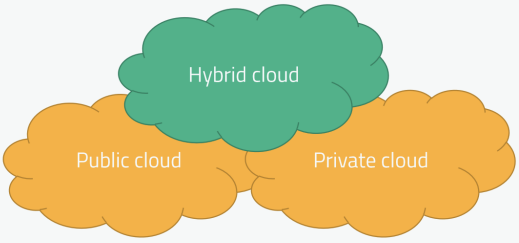
My solution in Amazon:

Example: Design a Data Warehouse for Log Data with AWS



Hybrid Cloud

- Combine your own data centers ("on-premises" or "private cloud") with a public cloud (AWS, Google, Azure, etc.)
- Allows easy scaling of on-premises systems
- Allows for regulations that require certain data to be on-premises
- Requires bridges between your data center and the cloud
  - The specifics vary by cloud provider
- "Multi-Cloud" – more than one public cloud provider



Design Interview → Working Backwards

Steps:

Q: Design Youtube.

1. Repeat the question → so that you are on same page
2. Break problem into pieces → like what part of youtube you want me to design, recommendation, feed, and whatnot.

Ask about requirements, latency, availability, storage solution, cost.

3. Ask lots of questions.
4. Think Out loud!

Working Backward

- Start from the customer experience to define your requirements
  - (This will gain MAJOR POINTS at Amazon, but works in general.)
- YouTube Example:
  - How will users discover videos? Do we need to think about building a search engine? A recommender engine? An advertising engine?
  - Use this to limit the scope of what you're being asked to do.
  - Understand the *customer experience* you are being asked to deliver.

Say, Okay I am gonna working backward from customer experience here.

- Identify WHO are the customers
- WHAT are their use cases
- WHICH use cases do you need to concern yourself with
  - You're not going to design all of YouTube in 20 minutes.
- Your initial task is to CLARIFY THE REQUIREMENTS of what you are designing.
- Your interviewer wants to see that you can think about problems from a business perspective and not a purely technical one.

Define Requirements

Scaling Req:

Defining scaling requirements

- Nail down the scale of the system. Is it hundreds of users? Millions?
  - This will inform you on the need for horizontal partitioning
  - How often are users coming? What *transaction rate* do you need to support?
- Also define the scale of the data.
  - Hundreds of videos? Millions?
- YouTube example: millions of users, millions of videos.
  - You will need every trick in the book for horizontally scaled servers and data storage.
- Some internal tool might not need this level of complexity, however.
  - Always prefer the simplest solution that will work.
  - Vertical scaling still has its place.

Latency Req:

Defining latency requirements

- How fast is fast enough?
  - This informs the need for caching and CDN usage
  - (Caching is also a tool for scaling, however – it reduces load on services & data stores)
  - Try to express this in SLA language (i.e., 100ms at three-nines for a given operation)
- YouTube example:
  - Caching video recommendations
  - Caching video metadata, descriptions, etc.

Availability Req

Defining availability requirements

- How much downtime can you tolerate?
  - Is being down a threat to the business? Or just an inconvenience?
  - If the former, you need to design for high availability
    - Opt for redundancy across many regions / racks / data centers rather for simplicity or frugality

If Interviewer doesn't tell anything→ think like a customer and Make a guess

They might not tell you.

- Work backwards from the customer to estimate what sorts of requirements make sense from their standpoint.
- "Back of the envelope" calculations may be needed. (How many users and videos *does* YouTube have? You can make an educated guess.)
- Get buy-in from the interviewer before proceeding to design the system.

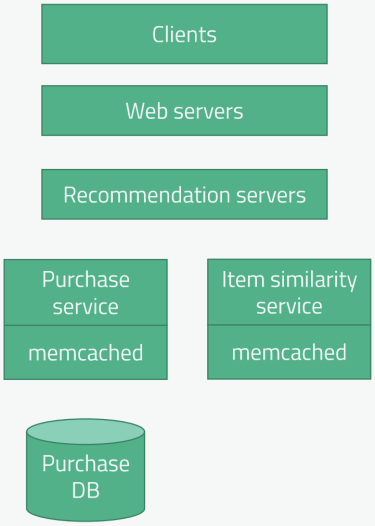
Design Strategies

1. THINK OUT LOUD

2. Sketch your design

Sketching Out Your Design

- Start with high-level components
- Work backwards if you can (especially at Amazon)
- Then flesh out each component as time permits
  - How do they scale?
  - How are they distributed for availability?
- Let the interviewer talk, listen to them. They may be trying to steer you in the right direction.
- Identify bottlenecks, maintenance, costs concerns as you go – show that you understand the tradeoffs of the choices you are making
- Notation and format generally doesn't matter much, as long as you can communicate what it means.



3. Be Hones about Knowledge

Be Honest

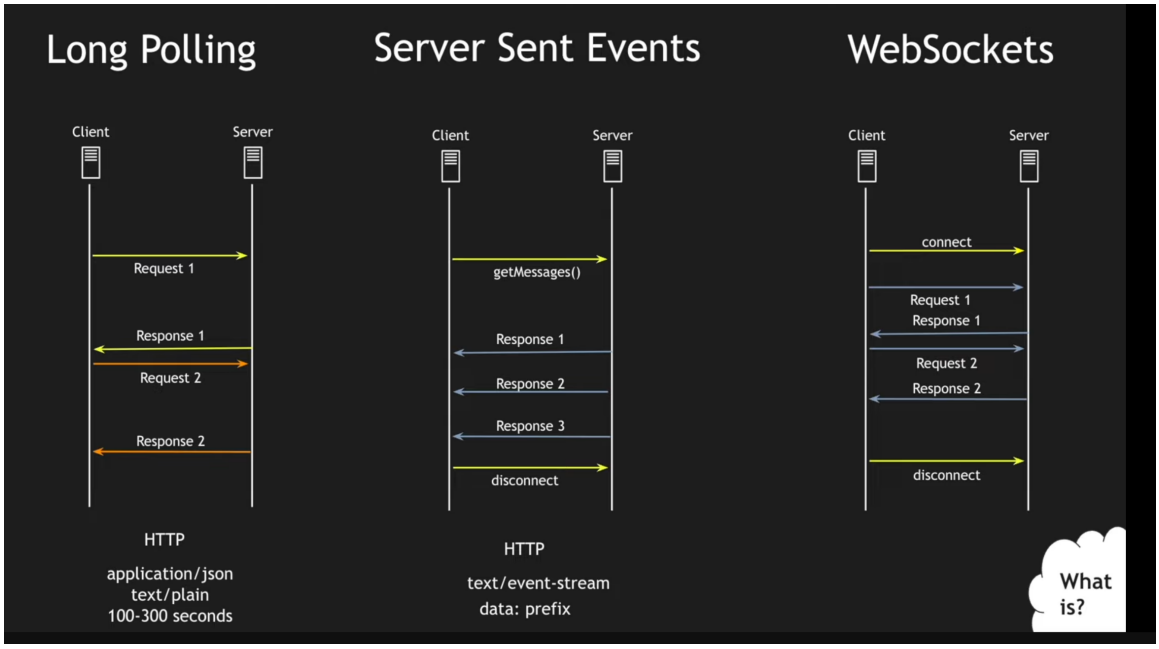
- Don't pretend to know stuff you don't know. That won't end well.
- If you're steered into a direction you're unfamiliar with, say so.
- But don't just give up! Try to think through it, working with the interviewer to come up with a solution collaboratively.
- This is an opportunity to demonstrate grit, perseverance, and the ability to work with others - which is more important than anything.

4. Defend Your Design

Defending Your Design

- The interviewer will try to poke holes in your design.
- What happens if X fails?
- What happens if we get a sudden surge of traffic / data?
- Did you meet the scaling & availability requirements you defined?
- Does your system meet all of the use cases discussed?
- How would you make it better?
- How would you optimize or simplify it?
- What is its operational burden? How will you monitor it?
- DON'T GET DEFENSIVE – take feedback constructively

long polling vs websockets vs server-sent events



Summary-

This video discusses different techniques for developing web applications that interact with the client-server model.- The three techniques covered are HTTP long polling, server-sent events (SSE), and websockets.- The video explores when to use each technique by overlaying them with examples from Xeroda and a gaming application.- Pros and cons of each technique are also discussed.

Highlights-

🌐 HTTP long polling is a technique where the client continuously polls the server for updates, ensuring no information is missed.-

📩 Server-sent events (SSE) allow for unidirectional data flow, where the server pushes data to the client without requiring additional requests.-

🤝 Websockets enable bidirectional communication between the client and server, supporting asynchronous messaging and binary data transmission.-

💡 HTTP long polling is easy to implement but adds latency and lacks reliability.- 💡

SSE is lightweight and based on HTTP, but it is unidirectional and has limited support for binary data transmission.- 💡

Websockets offer bidirectional and asynchronous communication, but lack automatic recovery of data and have limited support in older browsers.-

💡 Each technique has its pros and cons, and the choice of technique depends on the specific use case and requirements.

TinyURL System Design

https://www.youtube.com/watch?v=AVztRY77xxA&ab_channel=codeKarle

Summary:

https://www.codekarle.com/system-design/TinyUrl-system-design.html

Diagram