senoritadeveloper.medium.com

System Design Interview — Study Notes I - Nil Seri - Medium

Nil Seri

15-19 minutes

Scalability

- **Horizontal Scaling (Scaling Out)** = adding additional nodes or machines to your infrastructure to cope with new demands.
- **Vertical Scaling (Scaling Up)** = adding additional resources (CPU, memory, storage, etc.) to a system so that it meets demand (upgrade limitations likely to occur).
- EC2 (Amazon)
- data centers, zones => redundancy

Load Balancer

- distribute incoming client requests to computing resources such as application servers and databases.
- Preventing requests from going to *unhealthy* servers
- Preventing *overloading* resources
- Helping to eliminate a *single point of failure*
- *Distribution* => random, least loaded, session /cookies, round-

robin, weighted round-robin, layer 4, layer 7

- *Layer 4 Load Balancing* = decision info (source, destination IP, ports in the header) at transport layer.
  Layer 4 load balancers forward network packets to and from the upstream server, performing Network Address Translation (NAT).
  NAT = mapping an IP address space into another by modifying network address information in the IP header of packets
- *Layer 7 Load Balancing* = decision info (header, message, cookies) at application layer.

Fail-over

- **active-passive** = heartbeats are sent between the active and the passive server on standby (decision of passive server taking over the active's IP address and resuming service)
- **active-active** = both are actively serving, spreading the load between them. if public, DNS needs to know both of their IPs. if internal, app logic needs to know about them both.
- prone to loss of data, additional complexity added because we have more servers to handle

Replication

Quorum

- Replication problem => how to make sure that all replicas have the latest copy of the data and that all clients see the same view of the data?
- Quorum = minimum number of servers on which a distributed operation needs to be performed successfully before declaring the

operation's overall success.

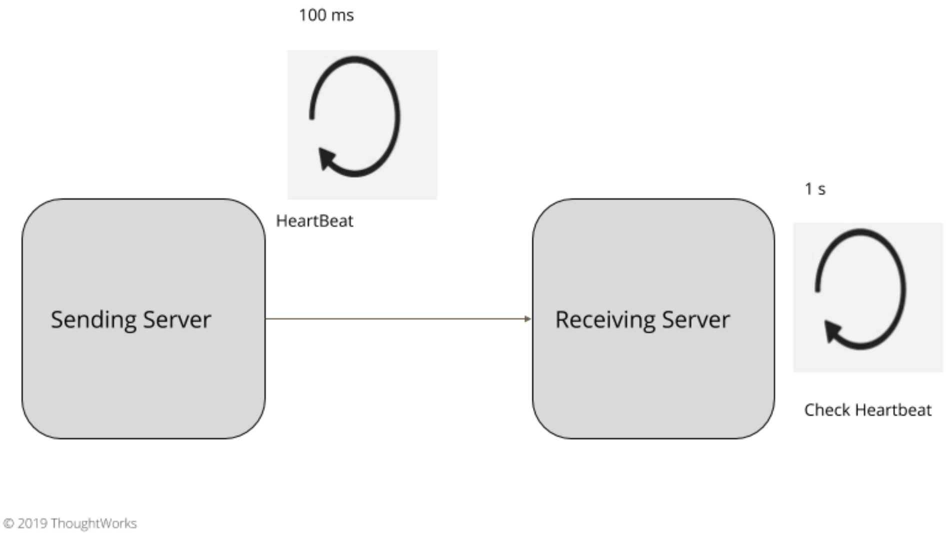
- Choosing the value = (*N*/2+1) where N is *total number of nodes.*
- To ensure every read will see at least one copy of the latest value written: *R+W>N* where W is minimum *Write* nodes, minimum *Read* nodes.

Leader / Follower

- to overcome problems (during failure) arisen using quorum
- Select one server amongst the cluster as leader.
- The leader is responsible for taking decisions on behalf of the entire cluster and propagating the decisions to all the other servers.

Heartbeat

- detection of server failures
- show a server is available by periodically sending a message (periodically) to all the other servers.



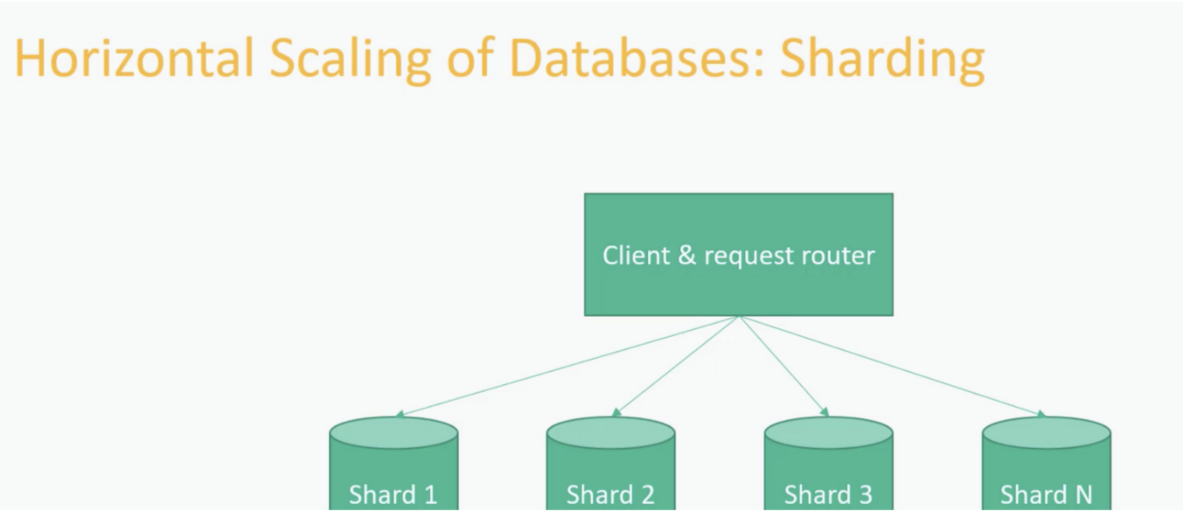
© 2019 ThoughtWorks

Heartbeat — https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html

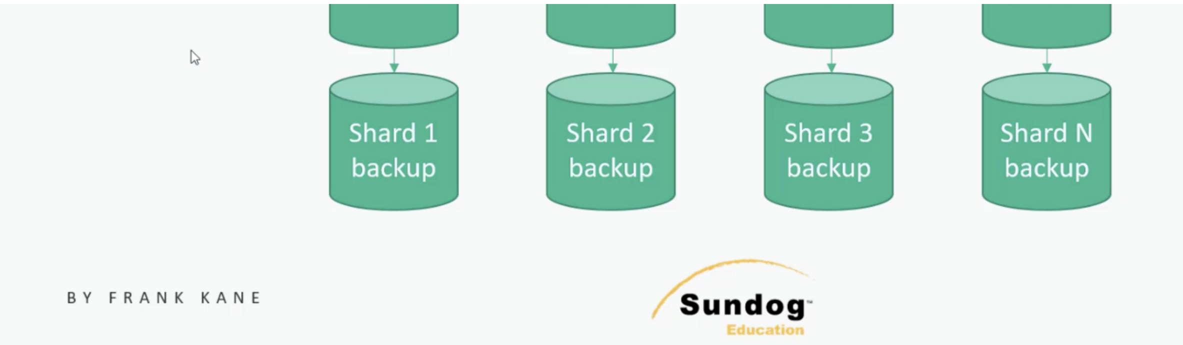
Horizontal Scaling for Databases

Failover servers:

- **cold standby** = periodic backup, restore
- **warm standby** = replication
- **hot standby** = for example, our frontend web host writes to both (not replication) simultaneously (so that data is in sync for read operations)
- **sharding = horizontal partitioning of your database, resiliency** ⬆️
  - hashing system required between router and shards
  - replication will be performed between a shard and its backup.
  - replica set = shard
  - data partitioned among shards
  - If you want to join data across different shards, it will be better to make sure the hash result is the same for a specific customer's data, meaning all its data should be kept in the same shard.
  - You should think of it as a key-value data stores and build up your data structure accordingly.

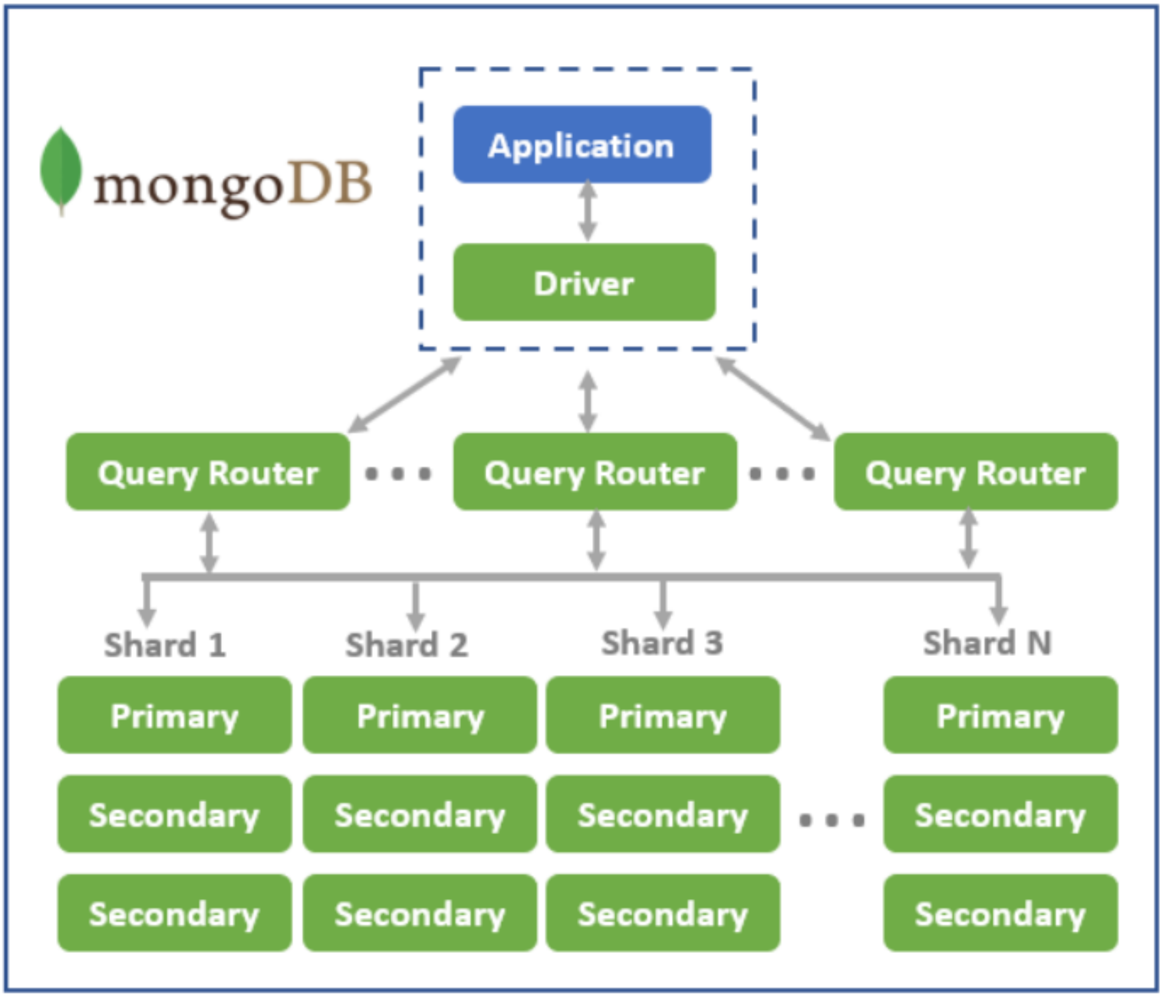


Horizontal Scaling of Databases: Sharding



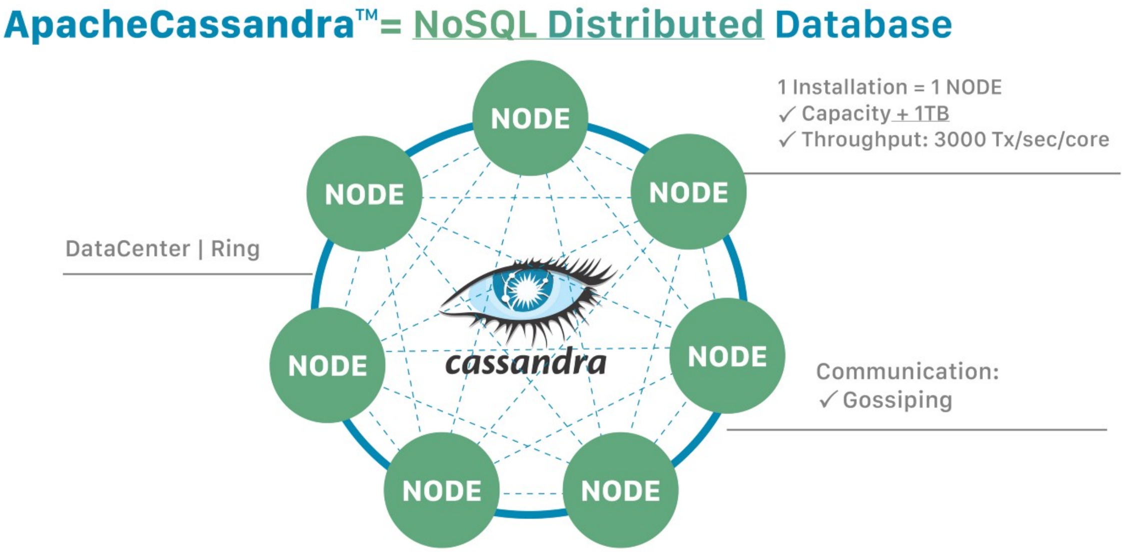
Sharding — https://www.udemy.com/course/system-design-interview-prep

- **MongoDB** = There should be at least 3 servers. In case of a failure of the current primary server, a primary is selected among secondaries. The primary server also acts as a router within that specific shard. Config server (Query Router) -> knows the mapping, holds communication info and routes the traffic.



MongoDB — https://laptrinhx.com/the-battle-of-the-nosql-databases-comparing-mongodb-and-couchdb-176648253/

- **Cassandra** = nodes (circular), ring system. Replicates data on other servers as well. Any of the nodes can be a primary server and serve an incoming request (write to one node, read from another node). Eventual Consistency. If the difference between a read and write will not cause a problem in your system, you can use Cassandra as your database.



Cassandra — https://cassandra.apache.org/_/cassandra-basics.html

- Sharded databases are sometimes called as “NoSQL” (not only SQL) databases.
  - tough to join across shards.
  - resharding (consider the resharding process as you add new servers to the current system)
  - hot spots => **celebrity problem**
  - most NoSQL dbs support most SQL operations and use SQL as their API.
  - still works best with simple key-value lookups.

- a formal schema may not be needed.
- **MongoDB, Cassandra, DynamoDB (serverless, sharding etc. operations are in the provider's care), HBase (Hadoop)**
- no strict shema = data store

- **Denormalizing**
  normalized data => normalized data: less storage space, more lookups, updates in one place (*relational database)*
  denormalized data => more storage place, one lookup, updates are hard. actual calls that will be made to db for most common operations. bottleneck if more than one db hit.

  Tips & Tricks: Start with normalized table structure, then according to the requirements will convert to denormalized. This way, you will be showing the interviewers that you know the trade-offs.

  Indexes

  Be careful about the indexes you will add in your table design since indexes decrease write performance.

  https://use-the-index-luke.com/sql/dml/insert

  > The number of indexes on a table is the most dominant factor for *insert* performance. The more indexes a table has, the slower the execution becomes. The *insert* statement is the only operation that cannot directly benefit from indexing because it has no *where* clause.

Data Lakes

- for "big data" solution
- cloud solutions

- throw data to text files (*as raw format to log files, csv or json*) into a big distributed storage system like **Amazon S3** => data lake, into a big bucket. dump unstructured data
- common approach for big data and unstructured data.
- **Amazon Glue** => for crawling, creates a schema for that data.
- to query the data (SQL interface) => **Amazon Athena**— serverless
  Amazon Redshift — distributed data warehouse, **Redshift Spectrum**
- partitioning (logs -> big data) dependent on the patterns of the queries.

ACID Compliance

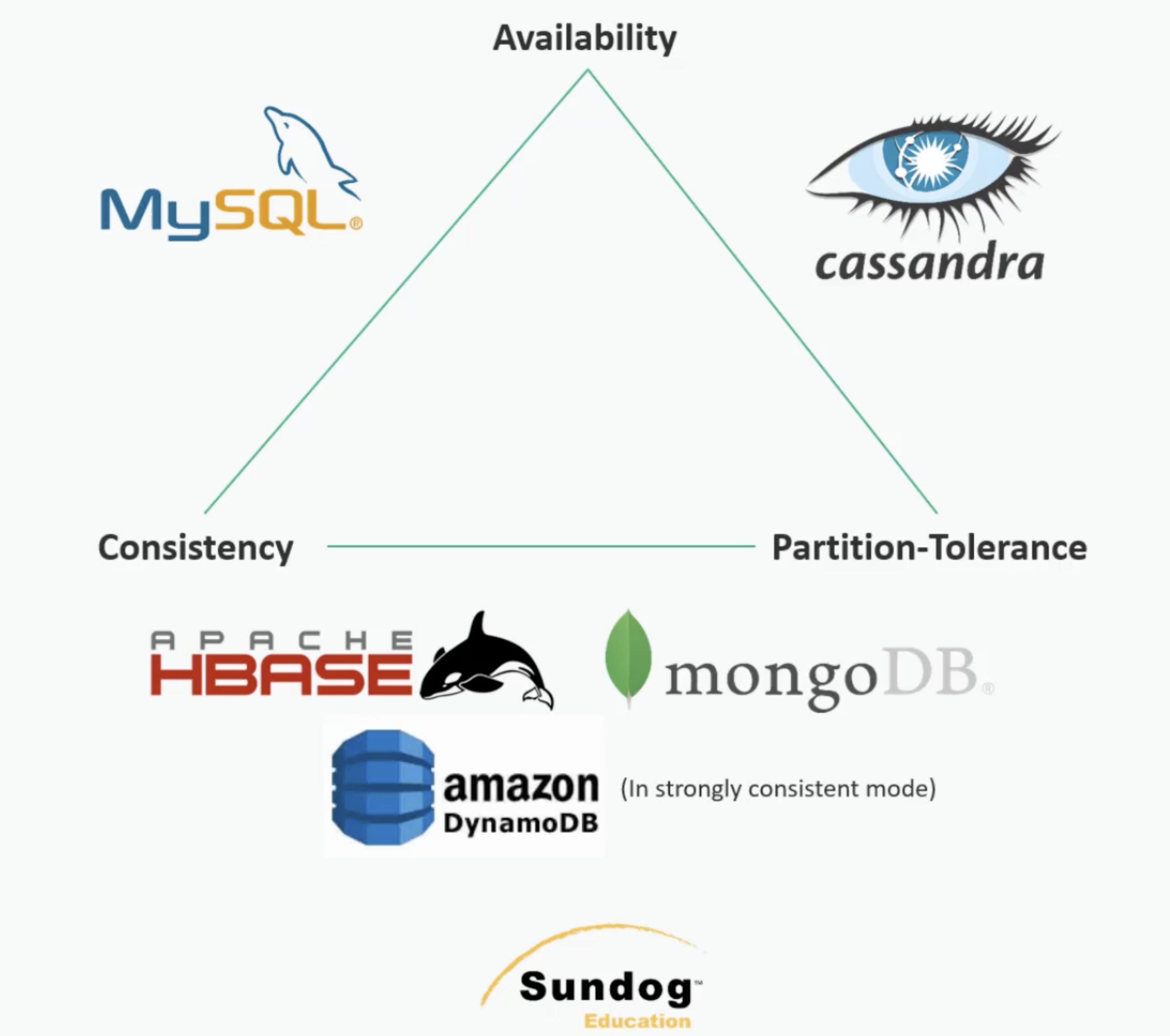
- **Atomicity**: either the entire transaction succeeds, or the entire fails. everything has to be performed at once (*atomic operation*)
- **Consistency**: all database rules are enforced, or the entire transaction is rolled back (here means consistent because of rules, not the consistency in CAP theorem. In CAP theorem, it is related to replication — read back what we wrote)
- **Isolation**: no transaction is affected by any other transaction that is still in progress (read and write operations should not break each other).
- **Durability**: once a transaction is committed, it stays even if the system crashes immediately after (durable storage)
- Oracle => full ACID compliance
- you have to give up scalability for ACID traits

Write-Ahead Log (RDBMS)

- providing atomicity and durability
- The changes are first recorded in the **log**, *which must be written to stable storage*, before the changes are written to the **database**.

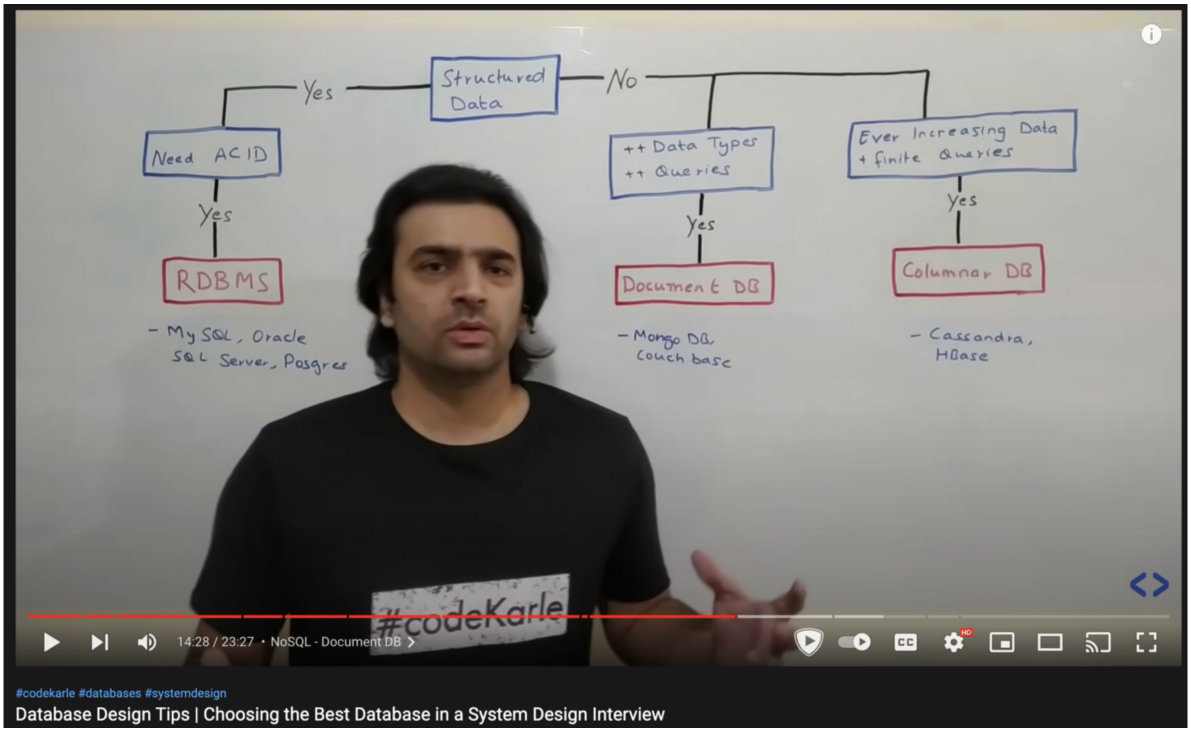
CAP Theorem

- **Consistency**: Can I get back what I just wrote right away?
- **Availability**: point of failures
- **Partition Tolerance**: something like durability -> horizontal scaling



The CAP Theorem — https://www.udemy.com/course/system-design-interview-prep

- **Oracle, MySQL** -> suitable for banking systems
- **HBase, MongoDB, DynamoDB** -> primary router host goes down, some downtime to switch => misses availability
- Partition Tolerance is more favored these days.

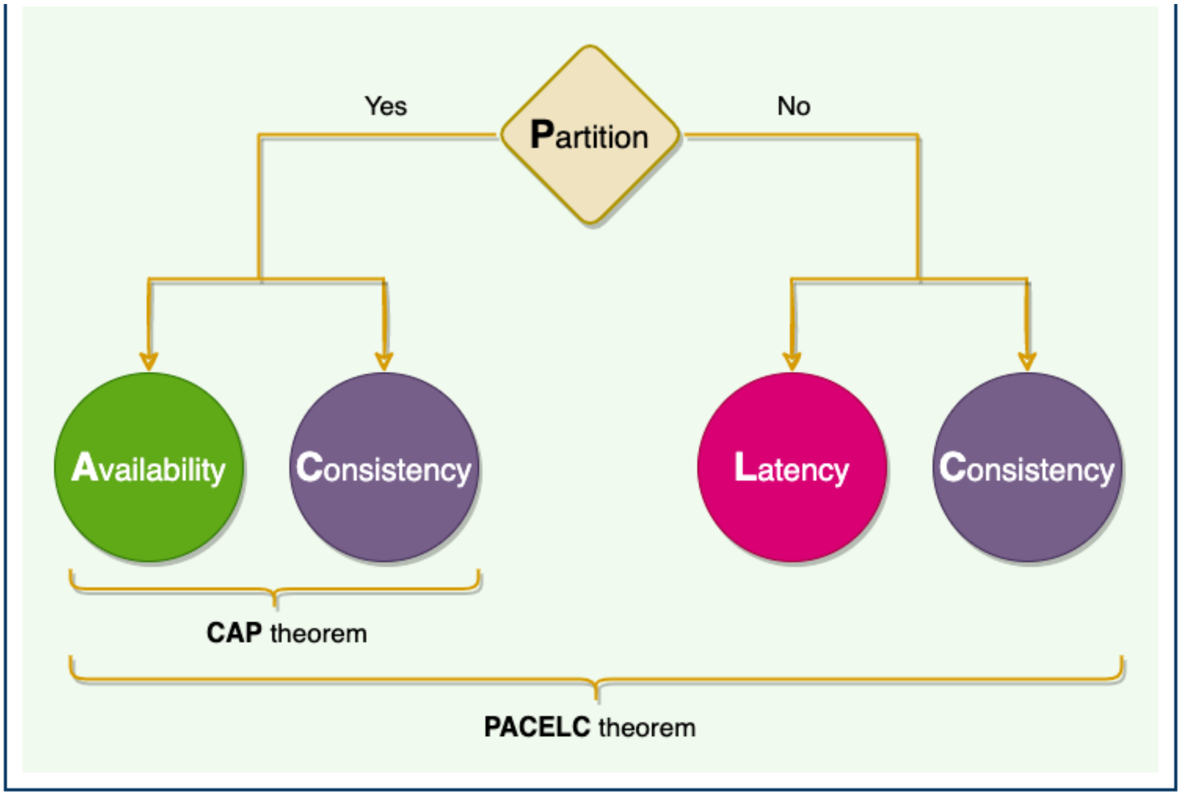

Structured Data
Yes
No
Need ACID
Yes
RDBMS
- MySQL, Oracle SQL Server, Posgres
++ Data Types ++ Queries
Yes
Document DB
- Mongo DB, Couch base
Ever Increasing Data + finite Queries
Yes
Columnar DB
- Cassandra, HBase
#codeKarle
14:28 / 23:27 • NoSQL - Document DB
#codekarle #databases #systemdesign
Database Design Tips | Choosing the Best Database in a System Design Interview

Database Design Tips — Choosing the Best Database — https://www.youtube.com/watch?v=cODCpXtPHbQ

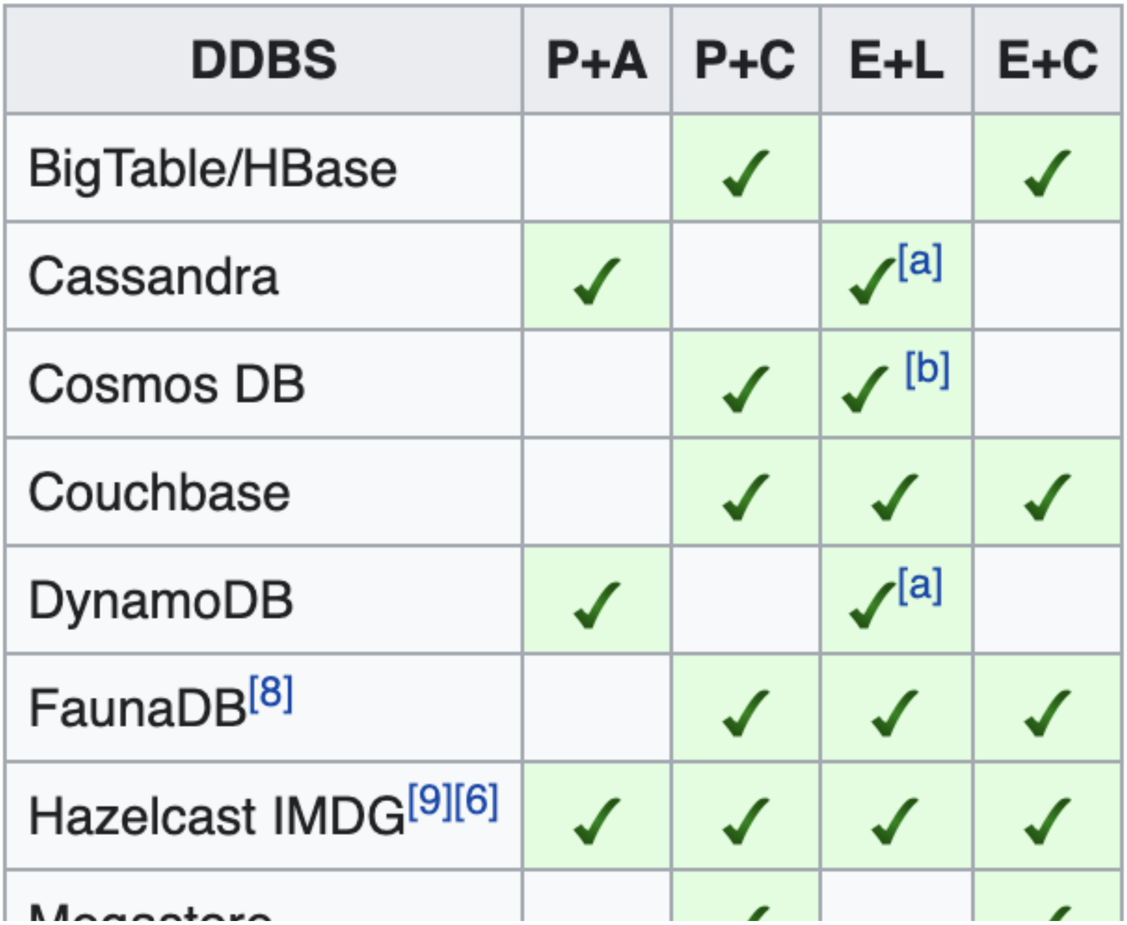
You can also follow the above chart according to your needs to find the most suitable database you can use in your design.

PACELC Theorem

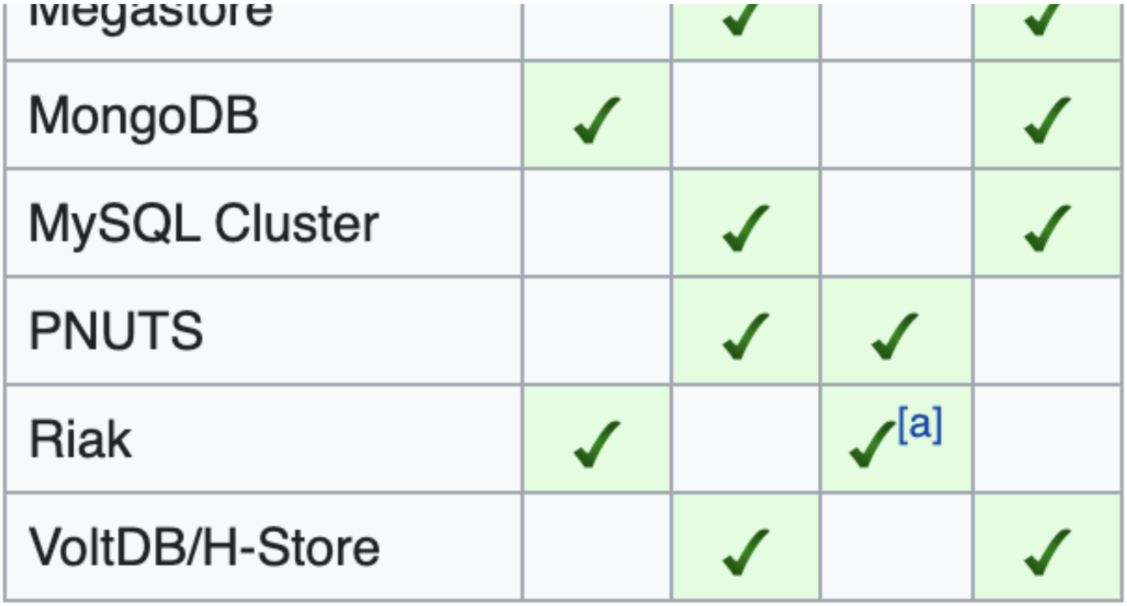
> It is an extension to the CAP theorem. It states that in case of network Partitioning in a distributed computer system, one has to choose between **A**vailability and **C**onsistency (as per the CAP theorem), but **E**lse, even when the system is running normally in the absence of partitions, one has to choose between **L**atency and **C**onsistency.



PACELC Theorem — https://medium.com/interviewnoodle/system-design-interview-basics-cap-vs-pacelc-cf7c5eebc313



DDBS	P+A	P+C	E+L	E+C
BigTable/HBase		✓		✓
Cassandra	✓		✓[a]	
Cosmos DB		✓	✓ [b]	
Couchbase		✓	✓	✓
DynamoDB	✓		✓[a]	
FaunaDB[8]		✓	✓	✓
Hazelcast IMDG[9][6]	✓	✓	✓	✓
Megastore		✓		✓



Megastore		✓		✓
MongoDB	✓			✓
MySQL Cluster		✓		✓
PNUTS		✓	✓	
Riak	✓		✓[a]	
VoltDB/H-Store		✓		✓

Database PACELC Ratings — https://en.wikipedia.org/wiki/PACELC_theorem

Caching

- SLA (service level agreements) specifies that requests to your underlying services respond in a matter of milliseconds almost all of the time. => caching (*avoid disk seeks, long trips across the internet*)
- Cache database info in memory inside a cache
- Databases also have a caching mechanism in themself but it's not enough. It would be better to keep the data close to app hosts in distributed systems since db servers are in different data centers.
- Alongside your application servers, a fleet of caching servers should be placed (caching layer). in-memory copies of data going to that database. most recent hits or most popular ones.

How caches work?

- horizontally scaled servers (horizontally distributed)

- clients **hash** requests to a given server
- in-memory (fast)
- appropriate for apps with reads >> writes
- the **expiration policy** dictates how long data is cached.
too long => your data may go stale
too short => won't do much good
- **hotspot**s can be a problem => "the celebrity" problem
not hashing only the key itself, dedicate a server or distribute this data among multiple servers. => Vnode (virtual nodes)
- **cold-start** is also a problem.
- how do you initially warm up the cache without bringing down whatever you are caching?
- initial burst
- before exposing to the entire world, warm-up procedure
- artificially simulated requests for caching layer or playing previous day backlogs.
- for systems like Wikipedia, e-commerce (products) => read >> writes
not suitable for banking systems

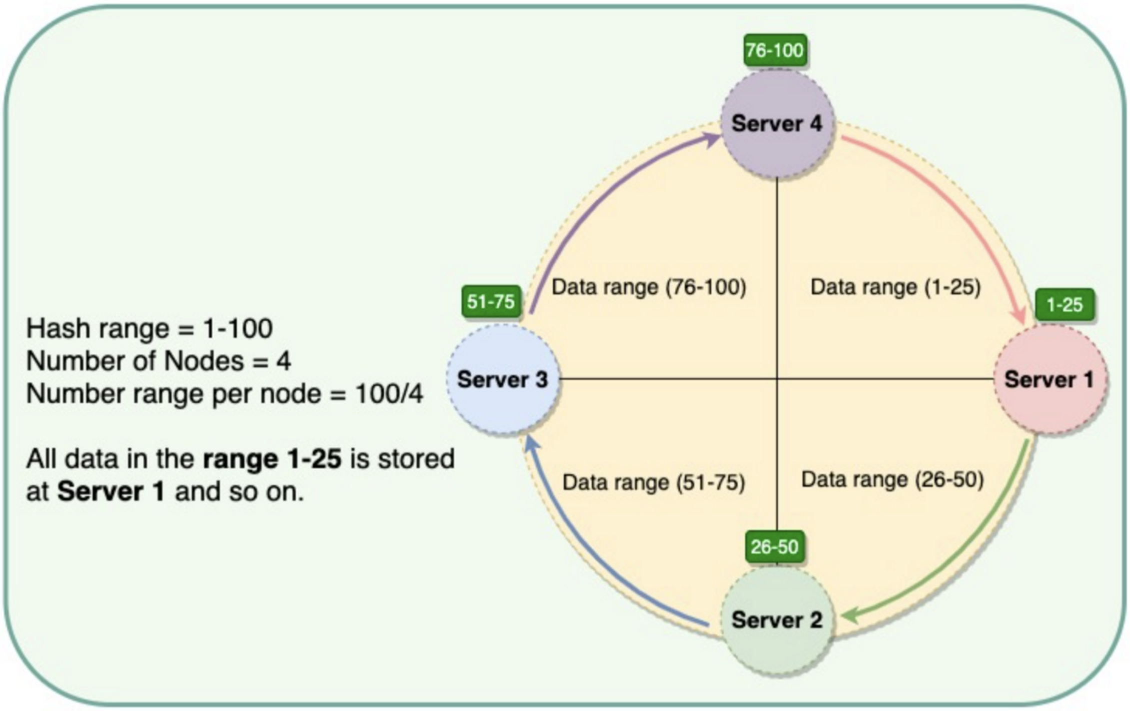
Tips & Tricks: Ask your interviewer about the expiration policy requirements.

Consistent Hashing

- Adding / removing nodes may require re-hashing.
- Distributed systems can use Consistent Hashing to distribute data across nodes.
- Ensures that only a small set of keys move when servers are

added or removed.

- Stores the data managed by a distributed system in a ring. Each node has a specific range of data.
- Amazon's **DynamoDB** and Apache **Cassandra** use Consistent Hashing to distribute and replicate data across nodes.



Consistent Hashing ring

Consistent Hashing — Ring — https://www.educative.io/courses/grokking-the-system-design-interview/B81vnyp0GpY

- **Bloom Filters:** (used in hashing) space-efficient probabilistic data structure (using a bit vector) that is used to test whether an element is a member of a set (either *definitely is not* in the set or *may be* in the set).

Eviction Strategies for Caching

Eviction Policies:

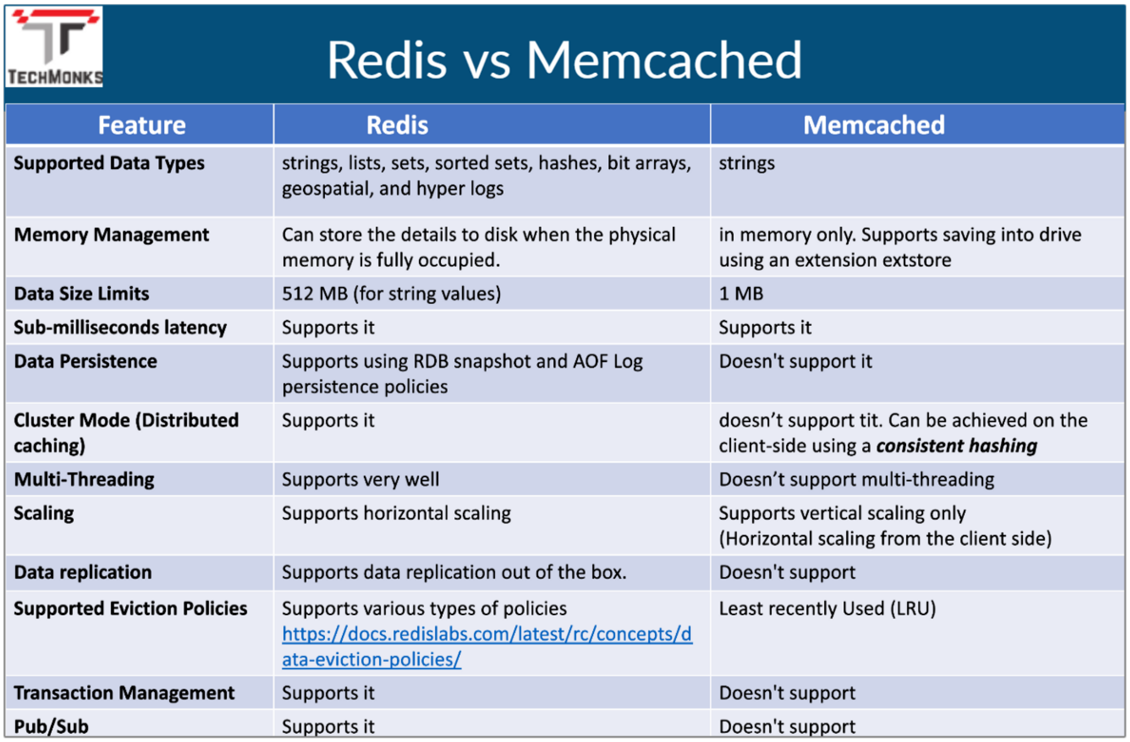
- **LRU** (Least Recently Used)
- **LFU** (Least Frequently Used)
- **FIFO** (First In First Out)

- cache miss
- most commonly used => LRU, consists of HashMap and Doubly LinkedList; a head pointer and a tail pointer. access =>using head pointer
  new => add to the front of the list.

  Caching Technologies:

- **memcached**
  - in-memory key/value store
  - open-source
- **redis**
  - adds more features
  - snapshots, replication, transactions, pub/sub (change in data)
  - advanced data structures
  - more complex in general
- **ncache**
  - made for .NET, Java, NodeJS
- **Ehcache**
  - Java
  - just a distributed map
- **ElasticCache**
  - cloud-based
  - Amazon Web Services (AWS) solution
  - fully-managed Redis or memcached
  - servers will be within the same data centers as your other app

components (if you are using AWS products; ElasticCache on top of DynamoDB, in front of your app running on Lambda or EC2)



Redis vs Memcached

Feature	Redis	Memcached
Supported Data Types	strings, lists, sets, sorted sets, hashes, bit arrays, geospatial, and hyper logs	strings
Memory Management	Can store the details to disk when the physical memory is fully occupied.	in memory only. Supports saving into drive using an extension extstore
Data Size Limits	512 MB (for string values)	1 MB
Sub-milliseconds latency	Supports it	Supports it
Data Persistence	Supports using RDB snapshot and AOF Log persistence policies	Doesn't support it
Cluster Mode (Distributed caching)	Supports it	doesn't support tit. Can be achieved on the client-side using a ***consistent hashing***
Multi-Threading	Supports very well	Doesn't support multi-threading
Scaling	Supports horizontal scaling	Supports vertical scaling only (Horizontal scaling from the client side)
Data replication	Supports data replication out of the box.	Doesn't support
Supported Eviction Policies	Supports various types of policies https://docs.redislabs.com/latest/rc/concepts/data-eviction-policies/	Least recently Used (LRU)
Transaction Management	Supports it	Doesn't support
Pub/Sub	Supports it	Doesn't support

Redis vs. Memcached — https://dzone.com/articles/redis-vs-memcached-which-one-to-pick

Content Distribution Networks (CDNs)

- regions
- latencies — clients at edge locations
  allows me to distribute the serving of some of this data
  go to request it from the closest
- expensive solution
- geographically distributed
- local hosting of
  - HTML
  - CSS

- Javascript
- Images
- Some limited computations may be available as well

- mainly useful for static content, such as images or static web pages. (YouTube, serving videos, etc.)
- CORS (Cross-Origin Resource Sharing) definitions required.
- **pull CDN** => pulls new content from server when the first user requests the content (using TTL)
  push CDN => receive new content whenever changes occur on server

  CDN Providers

- **AWS CloudFront** (you can use independently, you do not have to use with AWS Stack)
- **Google Cloud CDN**
- **Microsoft Azure CDN**
- Akamai
- **Cloudflare**

Resiliency

- resilient to failure
- the more servers you have, the more possibility of failure
- what if a rack of servers / a data center / a data region goes down?
- backup hosts (replicating)
- entire region? => data center = availability zone

Geo-Routing

- Using DNS tricks to figure out where to send a request locally.
- European traffic routed to European data centers.
  North American traffic routed to North American data centers.
  …
- For redundant capacity, overscale (as an extra region) to make sure of resiliency.
  If Europe is down, you will be routing to North America.
- 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 => over-provisioning
- You may need to balance budget vs. availability, not every system warrants this two.
  - provisioning a new server from an offsite backup might be good enough

Tips & Tricks: Ask your interviewer if this is required for your system.

Scaling Your Data

- scale data as well as traffic (storage & scale)

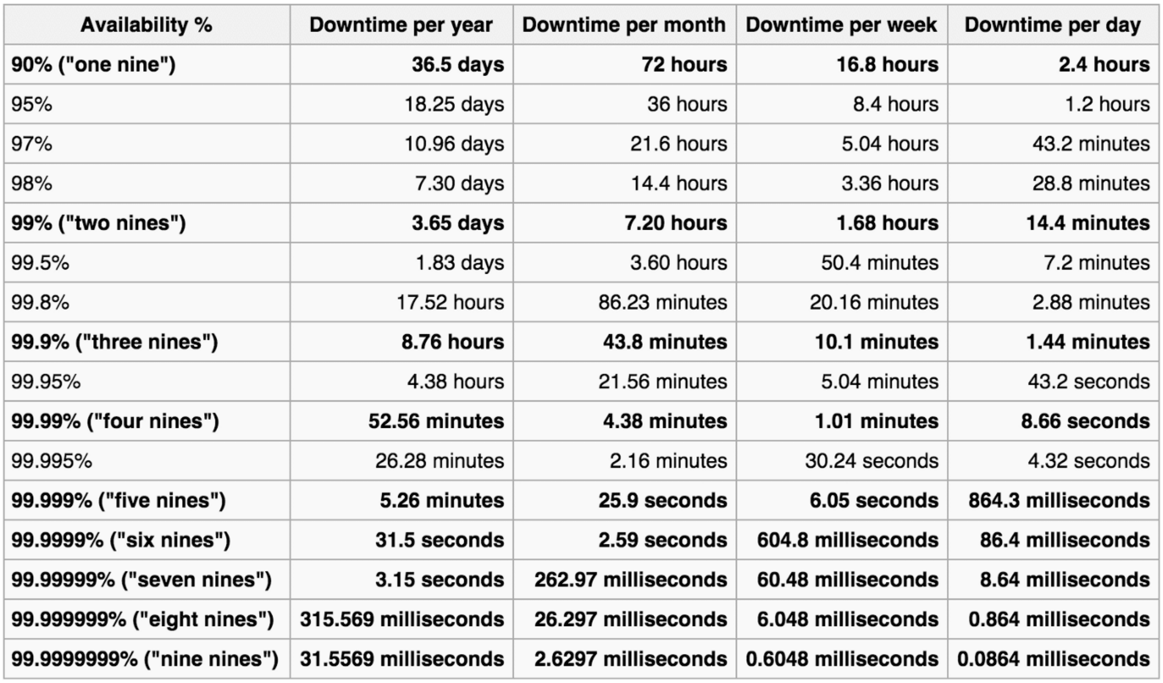
Distributed Storage Solutions

- services for scalable, available, secure, fast object storage
- use-cases: “data lakes”, websites, backups, “big data”

- highly durable (Amazon S3)

  SLA

- 99.9999999% durability (11 nines)
- percentile (of promise)
- latency => how quickly a service responds to a request.
  “3 nines” latency = 100 ms = 99.9% of requests come back within 100 ms.
- “availability” SLAs can be deceiving:
  %99 = 3.65 days of downtime in a year
  %99.9999 = “6 nines” = 30 seconds of downtime



Availability %	Downtime per year	Downtime per month	Downtime per week	Downtime per day
90% ("one nine")	**36.5 days**	**72 hours**	**16.8 hours**	**2.4 hours**
95%	18.25 days	36 hours	8.4 hours	1.2 hours
97%	10.96 days	21.6 hours	5.04 hours	43.2 minutes
98%	7.30 days	14.4 hours	3.36 hours	28.8 minutes
99% ("two nines")	**3.65 days**	**7.20 hours**	**1.68 hours**	**14.4 minutes**
99.5%	1.83 days	3.60 hours	50.4 minutes	7.2 minutes
99.8%	17.52 hours	86.23 minutes	20.16 minutes	2.88 minutes
99.9% ("three nines")	**8.76 hours**	**43.8 minutes**	**10.1 minutes**	**1.44 minutes**
99.95%	4.38 hours	21.56 minutes	5.04 minutes	43.2 seconds
99.99% ("four nines")	**52.56 minutes**	**4.38 minutes**	**1.01 minutes**	**8.66 seconds**
99.995%	26.28 minutes	2.16 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	**5.26 minutes**	**25.9 seconds**	**6.05 seconds**	**864.3 milliseconds**
99.9999% ("six nines")	**31.5 seconds**	**2.59 seconds**	**604.8 milliseconds**	**86.4 milliseconds**
99.99999% ("seven nines")	**3.15 seconds**	**262.97 milliseconds**	**60.48 milliseconds**	**8.64 milliseconds**
99.999999% ("eight nines")	**315.569 milliseconds**	**26.297 milliseconds**	**6.048 milliseconds**	**0.864 milliseconds**
99.9999999% ("nine nines")	**31.5569 milliseconds**	**2.6297 milliseconds**	**0.6048 milliseconds**	**0.0864 milliseconds**

Breaking Down the Nines — https://discoposse.com/2015/11/24/two-of-your-nines-dont-need-five-nines/

- There is hot-cool-cold storage in **S3**
  for cold storage, **Glacier** : cheap for archiving but late for read operations
- **Amazon S3**

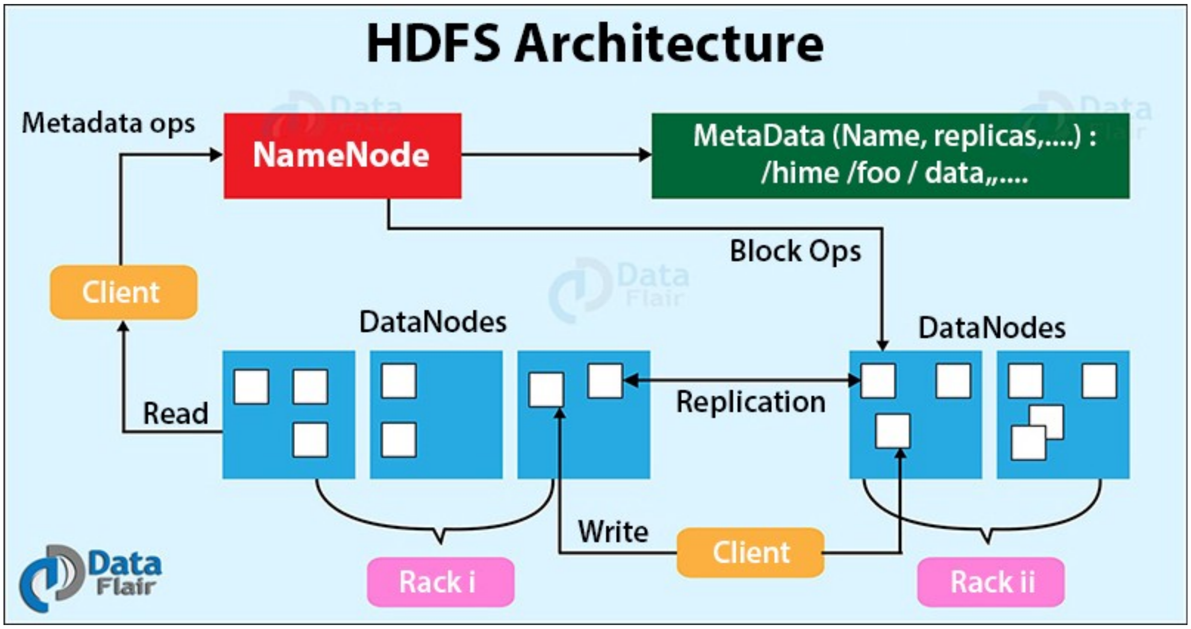
Google Cloud Storage

Microsoft Azure

Hadoop HDFS

…

HDFS Architecture



HDFS Architecture — https://data-flair.training/blogs/hadoop-hdfs-architecture/

- Files are broken 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.
- **Hadoop** => distributed storage / file system

Long-Polling, WebSockets, Server-Sent Events (SSE)

Ajax Polling

- mainly used by AJAX apps.
- repeatedly poll/request a server for data to get updates
- a lot of empty responses, HTTP overhead

HTTP Long-Polling (Hanging GET)

- client requests, server may not respond immediately
- pushes information to a client whenever the data is available
- after full response is sent, client re-requests
- has a time-out so periodical reconnect is required

WebSockets

- full duplex (and persistent) communication over TCP connection (while keeping the connection open)
  with TCP : All packets sent are guaranteed to reach the destination in the original order and without corruption (using checksum and ack)
- at first => WebSocket handshake
- lower overheads, facilitating real-time data transfer from and to the server

Server-Sent Events (SSE)

- persistent and long-term connection where server sends data to client
- not bi-directional like WebSockets

Other Tips & Tricks

- Optimize for performance and reliability over cost — like endless money source (but also inform about this to your interviewers).
- Nice to add reporting and analytics during / at the end of your design
- Mention about security (authentication / authorization, throttling)
- Add performance monitoring tools and alerting to your design