1. Database Design

Types

- **RDBMS** (*MySQL, PostgreSQL, Amazon Aurora (Online Transactional Processing, OLTP, backend), Redshift (Online Analytic Processing, viz.& BI)*) – tables, rows + joins using SQL
- Non-relational or **NoSQL** databases – generally, no join operations:
  - **key-value** store (dicts or hash maps; *DynamoDB* - serverless, scalable, replicated, low-latency key-value store with in-memory caching for high-perf. apps; can also store docs)
  - **graph** store (*Neo4j, Amazon Neptune*),
  - **column** store (*Amazon Keyspaces*; *Cassandra* - distributed, NoSQL, high-availability, wide-column store (WCS): no single point of failure, asynch. masterless replication => low latency. WCS: tables, rows, cols, but names/format of cols can vary from row to row in one table = 2D key-value store)
  - **document** store (*Amazon DocumentDB, CouchDB* stores "documents" of JSON-like data, no schema,).

    Good when:
  - need super-low latency OR store massive amounts of data,
  - unstructured data OR no relational data,
  - you only serialize / deserialize data (JSON, XML, YAML, etc.).

HBase - distributed DB, runs on top of HDFS (Hadoop Distributed File System) providing Google's Bigtable-like capabilities for Hadoop - fault-tolerant storing of large quantities of sparse data.
Redis (Amazon ElastiCache, A. MemoryDB for Redis) - distributed in-memory key–value store, cache / message broker; data: str, list, map, set, streams, etc.; used w/real-time stream solutions (Apache Kafka, Amazon Kinesis) w/sub-ms latency; real-time analytics - social media, ad targeting, personalization, IoT.
Amazon Timestream, Amazon Ledger

Scaling (not only DBs)

- **Vertical scaling**, "scale up" - adding **more power** (CPU, RAM, etc.) to server (good for low traffic as a simple solution, BUT hardware limitations + no failover / redundancy – single point of failure)
- **Horizontal scaling**, "scale-out" - adding **more servers**. (good for large scale apps)

Replication

Original or **master DB** - write operations (insert, delete, or update). Copies or **slave DBs** – read operations (gets copies of data from master). Most apps require many more reads than writes => need a lot more slaves.
Advantages: a) better perform. – distrib. slave nodes process more queries in parallel, b) reliability as data replicated at multiple locations, c) high availability - website still operational even if one DB is offline.
If slave offline, reads directed to healthy slaves OR to master if only one slave. If master offline, a slave promoted as new master. Challenge in prod. systems: slave ~ updated => recovery scripts, multi-masters, circular replication etc.

Sharding

Large DB into smaller parts - easier managed, same schema, different data. Hash f(x) to find corresponding shard (e.g. user_id % 4). Sharding / partition key – 1+ cols that determines how data is distributed (e.g. "user_id"), allows to retrieve, modify data efficiently by routing DB queries to correct DB. Challenges: resharding (due to over max limit, uneven requests, etc.), celebrity problem (info on 5 celebs in one shard – uneven requests), etc.

2. Load / Response Time

Consideration to **cache frequently used data**: a) when data is read frequently but modified infrequently, b) need an expiration policy - once cached data is expired, remove it from cache, c) need consistency: keeping the data store and the cache in sync, d) multiple cache servers across different data centers are recommended to avoid single point of failure, e) over-provide memory as buffer in case of sudden memory demand increases, f) eviction policy when cache is full: least-recently-used (LRU), Least Frequently Used (LFU) or First in First Out (FIFO).
CDN – content delivery network

3. Scaling Web Tier Horizontally

Stateful server – remembers client / session data (state) in each request => every request from the same client must be routed to the same server – possible, but adds overhead + changing servers and handling server failures is challenging.
Stateless server - keeps no state info => simpler, more robust, and scalable. Store session data in a persistent storage: RBD or NoSQL.

4. Message Queue

Scalable and reliable: producer can post a message to the queue when the consumer is unavailable and vice versa. Example: photo-processing tasks.
Kafka, RabbitMQ, AmazonMQ, ActiveMQ

5. Map Reduce and Platform Layer

- Adhoc SQL queries may not scale up trivially once the quantity of data stored or write-load requires sharding. Adding a **map-reduce layer** makes it possible to **perform data and/or processing intensive operations in a reasonable amount of time**. You might use it for calculating *suggested users in a social graph*, or for *generating analytics reports*. e.g. Hadoop, and maybe Hive or HBase.
- *Platform Layer (Services)*: **separating platform and web app** allows **independent scaling** - you can add platform servers for a new API w/out adding unnecessary capacity for your web app tier. Adding a platform layer => **reusing infra for multiple products** (a web app, API, iPhone app, etc) w/out writing too much redundant boilerplate code.

Numbers Every Programmer Must Know

Power of 2

ASCII char = 1 byte (8 bits)

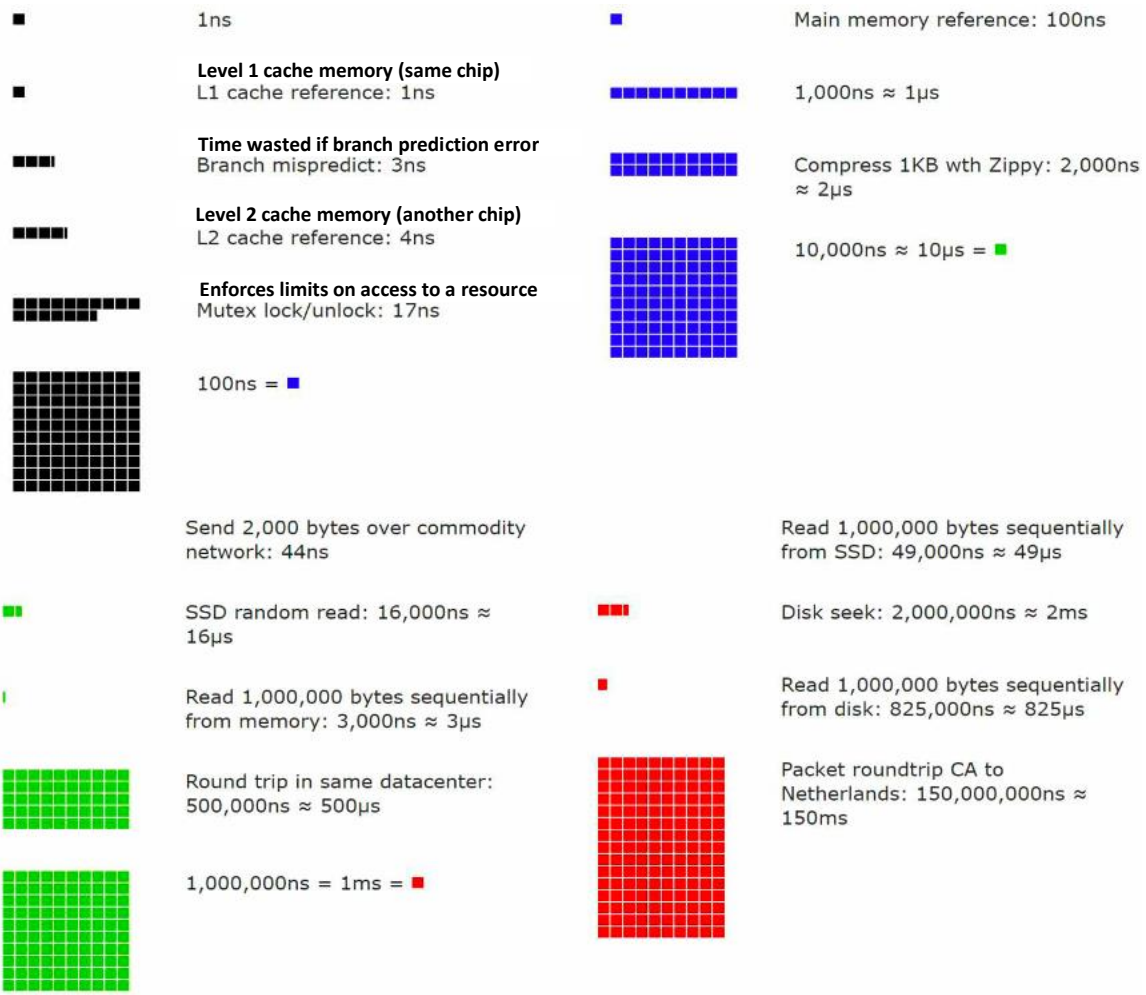
Power of 2	Approximate value	Size
2**10	Thousand	1 KB
2**20	Million	1 MB
2**30	Billion	1 GB
2**40	Trillion	1 TB
2**50	Quadrillion	1 PB

Availability

Availability %	Downtime per day	Downtime per year
99%	14.40 minutes	3.65 days
99.9%	1.44 minutes	8.77 hours
99.99%	8.64 seconds	52.60 minutes
99.999%	864.00 milliseconds	5.26 minutes
99.9999%	86.40 milliseconds	31.56 seconds

Table 2-3

Latency



Conclusions

- Memory fast, disk slow => avoid disk seeks
- Simple compression algos fast => compress data before sending over internet
- Data centers - in different regions => takes time to send data between them

Tips

- Round and approximate
- Write down assumptions (reference later)
- Label units
- Frequently asked estimations: QPS (queries per sec), storage, cache, # servers

System Design Interview Framework

Syst. des. Interview demos: ask **good questions**, **collaborate**, work **under pressure**, resolve **ambiguity**.
Red flags: a) **over-engineering**! by attempting design purity and ignoring tradeoffs (adding a load balancer or cache without thinking and explaining why)=> big additional costs, b) narrow mindedness, c) stubbornness

- **Part 1. Problem space exploration (3-5 mins)** - sign of senior eng. Do not jump right into solution right away (sign of junior eng)! Write down: **business goals**, **use cases** (what should the system do, scope), **constraints**. Clarify **requirements**, **assumptions**, **ambiguities** (due to open-ended questions) – what features, # users, how fast scales, company's stack, use existing service? Eliminate subsystems which interviewer is **not interested in**.
  What part of system to focus on (if a big system), display timeline?, what services, what content (a/v files, text), what to display in app, search?, trending topics, push notifications, etc. – **Impress here to prove your seniority**!
  Questions to ask:
  - What's the **business context** for the problem?
  - What are the **success metrics**?
  - **How many users** should the design handle? For example, 1 million or 100 million?
  - **How many entities** should the design handle? For example, 10Mor 100M rentals?
  - What kind of **latency** is the product specifying?
  - Any **legal** or **privacy related issues** to be aware of?

- Back-of-the-envelope **estimation of scale**
  # users, # messages, # requests, # queries per sec/month, storage (regular + audio/video files), network bandwidth => how manage traffic / balance loads

- **Part 2. High-level design** (work w/interviewer as a teammate, get their feedback) ~ 10 min.
  - **Block diagram** - 5-6 boxes w/core components
  - **Bottlenecks** based on above estimates
  - Should we include **DB schema**? Ask (depends on the case)
  - This should be a **simple working solution**. Most common use cases. Don't overeng (unnecessary components)! Will edge cases change yr design significantly – mention to the interviewer.
  - Follow the **data flow** and note how each component modifies the data (if not sure how – u don't need this component?).

- **Part 3. Deep dive** ~ 20 min
  - No single answer => different **approaches, pros** and **cons, tradeoffs** keeping system constraints in mind.
  - **Data flow** between components, **sharding** (logic), order (e.g. if need latest), **storage**, zipping, transport, encryption. Which DB? What block storage for media files? **DB schema**
    Data model:
    User: *UserID, Name, Email, DoB, CreationData, LastLogin, etc.*
    Tweet: *TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.*
  - **Scaling**: DNS, CDN, web/mobile app?, load balancer, autoscaling servers, DB replication / sharding, caching, message queue, stateless servers, logging / metrics / monitoring / automation
  - Define **APIs**
    ```
    post_tweet(user_id, content, tweet_location, user_location, timestamp)
    ```

  - Be prepared to implement algos in pseudocode, not just mention off-the-shelf components. If you have an open source component e.g. to handle the data processing – be ready to give a detailed description of how it works.
  - Mindset to remember - your interview design can handed to engineers so that they can implement it with no further instructions – and they must know algos to use in a particular component.
  - Interviewer may ask for quantitative analysis (requires envelope math). E.g. estimate # storage DBs. ❌ A poor answer: Maybe three DB instances based on my experience.
    ✅ A good answer: we are storing 100 M objects, each approx. 100 bytes => total 10^2*10^6 objects * 10^2 bytes / object = 10^10 bytes = 10 GB – one hard drive = one DB instance will suffice + backup instance for fault tolerance.

- **Final bottlenecks**
  - Single points of failure?
  - Enough data replicas or copies of services – a few failures not cause shutdown?
  - How performance monitored? Alerts of critical failures or bad performance?
- **Tradeoffs** (to avoid over-engineering)
- Wrap up

Important to demonstrate excellent **technical communication** - ability to communicate your reasoning in a logical and structured manner + the technical language you use in areas of expertise.

Common Mistakes

- ❌ **Not defining the problem** at the beginning. Identify the correct problem(s), the more senior the role, the more you are expected to dedicate time to this.
- ❌ **Overdesign** - adding a load balancer, cache, queue, etc. without thinking why it's needed.
- ❌ **Using "feelings"** instead of back of the envelope calculations - "I feel like 10 DBs are enough." Instead "each server handles approx. 5000 QPS, we calculated we're expecting 50,000 QPS => we need approx. 10 servers. To account for spikes in traffic, double that to 20 servers!"
- ❌ **Not rounding numbers** for quantitative analysis (rough estimations) - round to multiples of 5 or 10 to simplify the math.

References

- https://www.trybackprop.com/blog/system_design_interview (incorporated here)
- System Design Summary: https://gist.github.com/vasanthk/485d1c25737e8e72759f
- Hired In Tech course: https://www.hiredintech.com/classrooms/system-design/lesson/59
- Blog post about ML design by the author of "ML Design Interview": https://mlengineer.io/machine-learning-design-interview-d08be9f44260

Recommended reading: System **Design Interview - An Insider's Guide: Volume 1&2** Alex Xu if no time OR **Designing Data-Intensive Applications** Martin Kleppmann if you have 2 months or so.

REST vs. SOAP

- SOAP - strict standard / rules. REST – no strict standard, but constraints.
- SOAP uses only XML. REST uses XML, JSON, Plain-text, SOAP (but SOAP cannot use REST)
- SOAP - difficult to implement (heavy XML format), needs more bandwidth. REST - easy to implement, light weight, fast, used in smartphones (GET, POST, PUT, DELETE).
- SOAP - language, platform, and transport independent (REST requires use of HTTP)
- SOAP - ACID compliance, better security.

From https://gist.github.com/vasanthk/485d1c25737e8e72759f#key-topics-for-designing-a-system

Key topics for designing a system

1. **Concurrency**
- Do you understand threads, deadlock, and starvation? Do you know how to parallelize algorithms? Do you understand consistency and coherence?
2. **Networking**
- Do you roughly understand IPC and TCP/IP? Do you know the difference between throughput and latency, and when each is the relevant factor?
3. **Abstraction**
- You should understand the systems you're building upon. Do you know roughly how an OS, file system, and database work? Do you know about the various levels of caching in a modern OS?
4. **Real-World Performance**
- You should be familiar with the speed of everything your computer can do, including the relative performance of RAM, disk, SSD and your network.
5. **Estimation**
- Estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few prototypes or micro-benchmarks to write.
6. **Availability & Reliability**
- Are you thinking about how things can fail, especially in a distributed environment? Do know how to design a system to cope with network failures? Do you understand durability?

Web App System design considerations:

- Security (CORS)
- Using CDN
    - A content delivery network (CDN) is a system of distributed servers (network) that deliver webpages and other Web content to a user based on the geographic locations of the user, the origin of the webpage and a content delivery server.
    - This service is effective in speeding the delivery of content of websites with high traffic and websites that have global reach. The closer the CDN server is to the user geographically, the faster the content will be delivered to the user.
    - CDNs also provide protection from large surges in traffic.
- Full Text Search
    - Using Sphinx/Lucene/Solr - which achieve fast search responses because, instead of searching the text directly, it searches an index instead.
- Offline support/Progressive enhancement
    - Service Workers
- Web Workers

- Server Side rendering
- Asynchronous loading of assets (Lazy load items)
- Minimizing network requests (Http2 + bundling/sprites etc)
- Developer productivity/Tooling
- Accessibility
- Internationalization
- Responsive design
- Browser compatibility

Working Components of Front-end Architecture

- Code
    - HTML5/WAI-ARIA
    - CSS/Sass Code standards and organization
    - Object-Oriented approach (how do objects break down and get put together)
    - JS frameworks/organization/performance optimization techniques
    - Asset Delivery - Front-end Ops
- Documentation
    - Onboarding Docs
    - Styleguide/Pattern Library
    - Architecture Diagrams (code flow, tool chain)
- Testing
    - Performance Testing
    - Visual Regression
    - Unit Testing
    - End-to-End Testing
- Process
    - Git Workflow
    - Dependency Management (npm, Bundler, Bower)
    - Build Systems (Grunt/Gulp)
    - Deploy Process
    - Continuous Integration (Travis CI, Jenkins)