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.

1. **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

1. **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.

1. **Message Queue**

A picture containing icon

Description automatically generated

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

**Numbers Every Programmer Must Know**

|  |  |
| --- | --- |
| **Power of 2** | **Availability** |

**Latency**

Graphical user interface, application, table

Description automatically generated

**Enforces limits on access to a resource**

**Time wasted if branch prediction error**

**Level 2 cache memory (another chip)**

**Level 1 cache memory (same chip)**

|  |  |
| --- | --- |
| **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 => big additional costs, b) narrow mindedness, c) stubbornness

* Do not jump right in to give a solution! Write down: **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?  
  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. ~ 5 min.
* 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
* **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)
* **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)**
* **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

**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.