System Design

Resources:

1. hiredintech
2. successintech
3. interviewbit
4. lecloud-scalability
5. consistent hashing
6. system design 1
7. Tim Berglund's Distributed System in One Lesson
8. coder's journey
9. Julia Evans
10. architecting systems for scale

System design process:

1. constraints
    1. clarify system constraints. Never assume things that were not explicitly stated.
    2. identify use cases
    3. scope of the system
    4. amount of traffic
    5. amount of data

2. abstract design (components)
3. bottleneck
4. scalability

Systems design steps:

1. clarify requirements
2. system interface definition / APIs
3. estimations (scale, traffic(read/write), storage, bandwidth, memory)
4. defining data model (db schema, dbs → relational → mysql, postgres, NoSQL → cassandra, mongodb, Hbase, HDFS, BigTable)
5. component design
6. identify bottlenecks (scaling, SPOF, concurrency)
7. resolving bottlenecks (LB, db sharding, Replication, fault tolerance, monitoring)
8. security

Characteristics of Distributed Systems:

1. Scalability
2. reliability (partition tolerance) → replication, redundancy → eliminate single point of failure
3. availability
4. efficiency → response time / latency, throughput / bandwidth (#items delivers in given unit time)
5. serviceability / manageability

Scalability Principles:

1. Vertical scaling
    1. CPU: cores, L2 cache
    2. Disk: PATA, SATA, SAS, SSD, RAID
    3. RAM
2. Horizontal scaling - clones

3. Load balancing
    1. High availability
        1. active-active LBs
        2. active-passive LBs
    2. Algorithms
        1. Round robin
        2. weighted round robin
        3. route to server with least load
        4. route to server with least connections to clients
        5. fastest response time server
        6. IP hash of client IP address
        7. url hash → requests for a given object will always go to just one backend cache
        8. consistent hashing
    3. ssl termination
    4. geography based load balancing - at DNS level / global LB
    5. elastic scalability
4. Proxy server
    1. filter requests
    2. log requests
    3. transform requests (by adding/removing headers, encrypting/decrypting, or compressing a resource)
    4. cache
    5. reverse proxy
5. Caching
    1. query cache in db
    2. memcached - indexes in mem (influxdb)
    3. LRU, MRU(Most), FIFO, LIFO, LFU(least frequent), RR(Random replacement)
    4. bloom filters (bit vector)
    5. application caching / database caching

6. cache invalidation
    1. write-through cache (write→db→cache)
    2. read-through cache / write-around cache (write→db→return , read→cache miss→write to cache)
    3. write-back cache (write→cache→return, async write to db)
7. locality of reference principle: recently requested data is likely to be requested again
8. 80-20 rule → 20% of daily read volume is generating 80% of traffic
9. distributed hash table
10. Content distribution network (CDN) for static media
    1. CDN → overlay n/w → IP n/w (IP addr) is an overlay on LAN (MAC addr) → CDN (Node ID) is an overlay on TCP/IP (IP addr)
    2. coral key-based routing

6. Sessions-cookies → multiple servers don't store state in their hard drive but in a global shared storage(fs,db) or in Load balancer → but then this shared storage or load balancer becomes a bottleneck, SPOF → storing server hash in cookie for load balancer to send the req of a particular session to the same server → RAID → sharding, replication
7. Platform layer b/w application and db. Platform server is more I/O intensive (needs SSD) and application server is more cpu intensive for compute. Product agnostic platform interface.
8. DB
    1. out of space → archive data
    2. denormalization - no joins in db queries - app does the dataset-joins
    3. sql tuning
    4. sharding
    5. db indexing
    6. object storage (S3, HDFS)
    7. bigtable/hbase
        1. combines multiple files into one block to store on the disk
        2. is very efficient in reading a small amount of data
    8. NoSQL
        1. K-V → Redis, Voldemort, Dynamo

        2. document → mongodb, couchdb
        3. wide column → cassandra, HBase (twitter, fb, instagram - followers, tweets)
        4. graph → Neo4j, InfiniteGraph
9. Data deduplication
10. Replication
    1. master-slave
    2. master-master
    3. read from slaves, write to masters
    4. read replication
        1. consistency
        2. single write master is the bottleneck
    5. problems:
        1. master can be SPOF → be redundant
        2. all write to master must be replicated to slaves
        3. replication lag causes slave lag
11. Sharding / Partitioning
    1. high availability
    2. faster queries
    3. reduces contention in data store
    4. more write bandwidth, parallel writes in shards
    5. you can do more work at backend, handle more load due to || writes
    6. data are denormalized
    7. data are parallelized across multiple parallel instances
    8. smaller sets of data - easy to cache, backup, restore, manage
    9. data are more highly available
        1. replication within a shard
        2. master-slave / master-master within a shard
    10. sharding doesn't use replication
    11. types
        1. horizontal sharding (range based)

        2. vertical sharding
        3. key/hash based sharding
        4. consistent hashing - solves elastic scaling problem [Good]
        5. directory based sharding - lookup service - solves elastic scaling problem, but migration takes time
    12. problems:
        1. Rebalancing data → your data references can be invalidated so the underlying data can be moved while you are using it.
        2. joining data from multiple shards
        3. no referential integrity (foreign keys)
        4. how to partition data in different shards
        5. scatter-gather
    13. sharding based on ID(user/entity), creation time, (creation time epoch + auto inc ID)
    14. locations across the world → location QuadTree
12. Reed-Solomon encoding to distribute and replicate data.
13. Map-Reduce / Scatter-Gather → hadoop [batch processing, HDFS, Hive] / Spark [RDD, batch/stream processing] / Apache Storm [streaming data processing] / lambda architecture [batch/stream processing] / flink
14. Geographically distributed data centres - availability zones
15. High availability
    1. Do a multi-availability zones deployment
    2. automatic db snapshots
16. API rate limiting / throttling
    1. hard/soft throttling
    2. elastic/dynamic throttling
    3. fixed window algorithm
    4. rolling window algorithm
17. Security
    1. tcp 80/443 → LB → SSL termination
    2. tcp 80 → server

3. tcp db-port → db
4. IP blacklisting and whitelisting

18. Being asynchronous → loosely coupling subsystems
    1. precomputing
    2. signal/event/callback
    3. messaging
        1. message queues
19. Kafka
    1. distributed message queue
    2. message - immutable array of bytes
    3. topic - queue
        1. topic partitioning
        2. ordering is only within a partition, no global ordering across all the partitions of a topic
    4. producers
    5. consumers
        1. can request messages by index
    6. message brokers
    7. zookeeper
        1. producers use it to find partitions and replication information
        2. consumers use it to track current index, zookeeper keeps per consumer per partition current index
20. Zookeeper
    1. distributed in-memory db
    2. leader-followers
    3. eventual consistency
    4. distributed locks
    5. used for consensus
    6. storm, kafka
21. Long polling, web sockets, server-sent events, pub-sub model, push notification

22. pagination
23. Consistent hashing
    1. elastic scaling for cache servers (dynamic adding/removing of servers based on usage load), storage nodes (NoSQL dbs)
    2. every time we scale up or down, we do not have to re-arrange all the keys or touch all the database servers
    3. when a server goes down, use reverse index/hash → index/hash builder → {server_id: hashset(data_ids)}, server data snapshot
    4. Consistent Hashing has successfully solved the problem of non-uniform data distribution (hot spots) across our database server cluster → As the number of replicas or virtual nodes in the hash ring increase, the key distribution becomes more and more uniform
    5. Facilitates Replication and partitioning of data across servers
    6. relieves hot spots
24. Eventual consistency - distributed computing
    1. gives high availability → all nodes are always available to be read but some nodes may have stale data at a particular point of time
    2. lower latency
    3. read scalability
    4. data propagation to all the replicated nodes takes time
    5. examples:
        1. Photo sharing system like Flicker
        2. Message timeline for a social app like Facebook or Twitter
        3. DNS (Domain Name System)
    6. x strict/strong/immediate consistency → all readers are blocked until replication of the new data to all the nodes are complete
25. Strong Consistency
    1. R+W>N
    2. R = #nodes which should agree when data is read
    3. W = #nodes which should acknowledge the update when data changes
    4. N = #nodes to know the data(#replicas)
26. CAP theorem
    1. CA - Relational DBs - mysql, oracle

2. CP - Google's Big table, mongodb, Hbase, memcached, redis
3. AP - Couchdb, cassandra, dynamodb, voldemort, riak, simpledb

27. Distributed transactions - ACID
    1. Atomic - either all the changes of the transaction work or none
    2. Consistent - after transaction completion, the database is left in a valid state
    3. Isolated - concurrent transaction leaves the database in the same state that would have been obtained if the transactions were executed sequentially
    4. durable - on tx commit the changes persist
    5. responses to failure
        1. write-off / rollback
        2. retry
        3. compensating action

28. Distributed consensus
    1. Paxos → quorum (read) → prepare, promise, accept request, acceptance (write, proposer-acceptor)
    2. Raft
    3. blockchain (can handle liars)