System design interviews:

1, Requirements clarifications:

Always clarify requirements at the beginning (functional/non-functional/extended)
Find the exact **scope/purpose** of the system***, end goal (what to focus on)***
How to define success? Functional, nonfunctional (scalability, availability, performant)

2, Capacity estimation and constraints (back-of-the-envelope estimation):

Design considerations, estimate the **scale** (number of), availability vs latency
Daily **storage,** 5 years, traffic volume (outgoing/read, incoming/write)
Read vs write, storage size, traffic **bandwidth**, QPS, memory/cache
80-20 rule (meaning 20% of hot objects generate 80% of traffic)

3, System interface definition:

System APIs, functional interface
Explicitly state what is expected from the system

4, Defining data model:

Database design, primary data object
Clarify how data will flow between different components
Guide data partitioning and management, storage (distributed file storage P/Video)

5, High-level design:

Diagram with core components
Multiple application servers
Read vs Write heavy, SQL vs NoSQL

6, Detailed design:

Delve deeper into major components, focus on **data, storage, transfer, processing**
Present different approaches, pros/cons, why prefer one to the other, **tradeoffs**
Data partition, hot users

7, Identifying and resolving bottlenecks:

Single point of failure (**SPF**), backup/replica
Scaling, partitioning, load balancing and caching
Log, monitor, alert on key metric: QPS, latency, availability (SLI), cache/CDN hit rate
Collect **metrics** based on SLI, define thresholds of metrics based on SLO alerting, monitor the threshold so that does not break SLA

系统设计考察角度：

1，交流沟通和理解能力，充分交流理解所设计系统的目标，方便做 tradeoff：
   a. 询问系统商业目的，解决什么问题，服务受众，商业/个人用户
2，资源需求，PQS，latency，需要多少机器，CPU，内存，硬盘，带宽资源分配：
   a. 询问系统功能和技术需求，DAU/QPS/Latency/availability，哪些子功能
   b. 如何定义成功，大概估算，写下来并跟面试官确认
3，整体设计和架构能力
4，扩展性，容错性，延迟要求， 跟 operation 相关的要求，今后的扩展/maintain

Web server (handle user sessions), app server (handle complex business logic, ticket management, store data in DB, work with cache)
Reverse proxy: gatekeeper, screener, load-balancer; VPN (forward proxy), LB (reverse proxy)
LB: client->server, server->DB, server->cache, keeps the mapping
Partition issues: hot users, unbalanced servers
Data to users: pull, push, **hybrid**; fanout to online friends; push to notify/pull for serving
Client keep open connection with server: long polling, WebSocket, push notification, establishing an early connection with the server
Round robin load balancer:
    simple to implement, no overhead, automatically skip dead server
    doesn't consider server load, (use intelligent LB, periodically query server load, adjust traffic)
Client-side monitoring: average latency, delivery stats, daily peak
Search: indexing, ranking, and retrieval
Ranking improvement: constantly evaluating with user stickiness, retention, ads revenue
Replica: for load balancing, and fault tolerance; HA, disconnected ops, latency, scalability
Consistent hashing: fault tolerance, replication, remove dead host
Rate limit algorithm: token bucket, leaky bucket
    fixed window ({key: {count, timestamp}}), remote Redis/memcache
    rolling window ({key: [ts1, ts2, ts3, …, tsn]})
    sliding window with counter ({key: {ts1: count1, ts2: count2, ts3: count3}}), Redis hash
Key characteristics of distributed systems:
    scalability, reliability, availability, efficiency, durability, serviceability/manageability
Cache eviction policies: FIFO, LIFO, LRU, MRU, LFU, RR
Data partition:
    horizontal (row), vertical (col), directory (mapping)
    range, hash, list, round-robin, composite (consistent hashing)
    joins and de-normalize, referential integrity (foreign key), rebalancing (not uniform)

ACID: Atomicity, Consistency, Isolation, and Durability
CAP Theorem for distributed system: consistency, availability, and partition tolerance

NoSQL:
    key-value store (Redis, dynamo),
    document (mongodb, couchdb, orientdb),
    wide column store (hbase, **Cassandra**, amazon simpledb),
    graph (neo4j)
Storage options:
    Database: SQL (mysql, postgres), NoSQL (cassandra, dynamodb)
    In-memory Storage: Redis (persistence), memcache
    Message Queues: Kafka/SQS;
    Batch processing: Hadoop MapReduce, offline job, higher latency
    Stream processing: kafka/spark, kinesis (cloud); time sensitive

1, requirement clarification discussion, Deal with ambiguity
User/customers (who is using the system/how the system will be used),
Scale (read/write), how many read per second (QPS), how much data per request (bandwidth), how many video views per second, can there be spikes in traffic
Performance, what's the expected write/read data delay (batch processing, stream processing)
Cost (minimize development, open source), maintenance (public cloud)

2, Functional: APIs (Name of API, parameters), what systems should do
Non-functional: system quality, how system supposed to be, scale vs performant vs available
Write down verbs, define input parameters and return values
Scaling = Partitioning
Reliable = Replication and checkpointing
Fast = In-memory/cache

3, Data model:
Individual events (fast write, easy slice/dice, recalculate); slow read, costly for large scale aggregated data (read fasts, data is ready for decision making), require aggregation pipeline, hard to fix bugs
Where we store: how to scale write/read/make both write&read fast, not lost data, strong consistency, recover data, ensure data security, make it extensible for change
How we store: mysql vs nosql

4, High level design (drive conversation):
How data gets in
How data gets out
Where to store

5, Detailed design
Focus on data, storage, transfer, processing
Use fundamental concepts
Apply relevant technologies

6, Bottlenecks/tradeoffs (listen to interviewer, knows the tradeoff):
4 golden singles: latency, traffic, errors, saturation
Quality system: weak (continues query), strong (raw events, Hadoop)
Top k most frequent items: single system
Batch processing vs stream processing

7, Processing service:
How to scale, high throughput, not to lose data, data is unavailable/slow
Push vs pull, pull has more advantages (fault tolerance)

8, Client:
Checkpointing, persistent storage
Data partitioning (multiple queues)

Buffering/batching, blocking vs non-blocking IO,
Timeout (connection 10ms, request processing/latency percentile)
Retries with exponential back-off, jitter add random delay
Circuit breaker, threshold,

9, Load balance:
Software, hardware LB, network (TCP/fast, http/look inside msg)
LB algorithm: round robin, least conn, least response, hash based
DNS, health checking, high availability, primary/secondary, different data centers

10, Service:
Partition strategy, video id, event, consistent hashing
Service discovery, (client side, server side), service registrar (zookeeper)
Casandra (gossip to each other, leaderless)
Replication (single leader, leaderless, multi leader)
Message format, text/json/xml, binary format (schema: thrift/protobuff)

Technology stack
Client: netty, Netflix hystrix, polly
LB: netscaler, nginx, ELB,
Messaging systems: apache Kafka, kinesis,
Data processing: spark, flink, kinesis data
Storage: HBase, Cassandra, Raw: Hadoop/redshift/s3
Vitess: large scale MySQL
Message deduplication: Redis, SQS, RocksDB (key/value)
Zookeeper:

Storage (master-slave, distributed), Computation (MapReduce, spark), Messaging (Kafka)

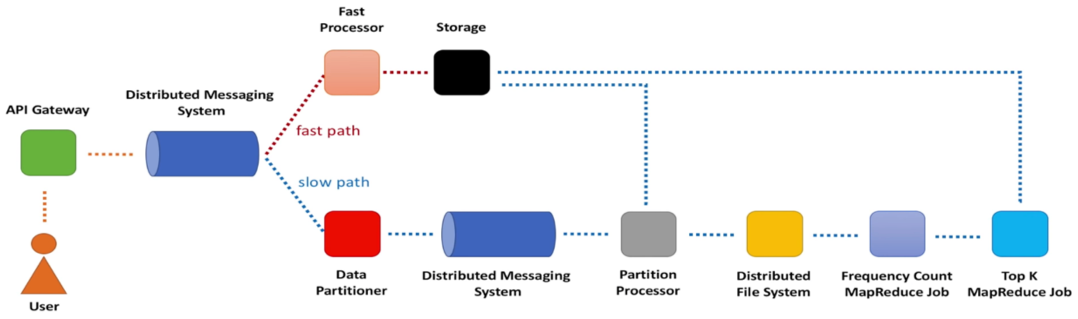
Four distributed system architectural patterns:
Modern three-tier (Presentation/business/Data)
Sharded (MySQL): complexity, no comprehensive view of data, oversized shards
Lambda architecture, assumes unbounded (short term), immutable data (long),
Messaging Streaming

Production ready: alerts, canary analysis, autoscaling, chaos, consistent naming, ELB, healthcheck, immutable machine images, squeeze testing, staged, red/black deployment, timeouts, retries, fallbacks

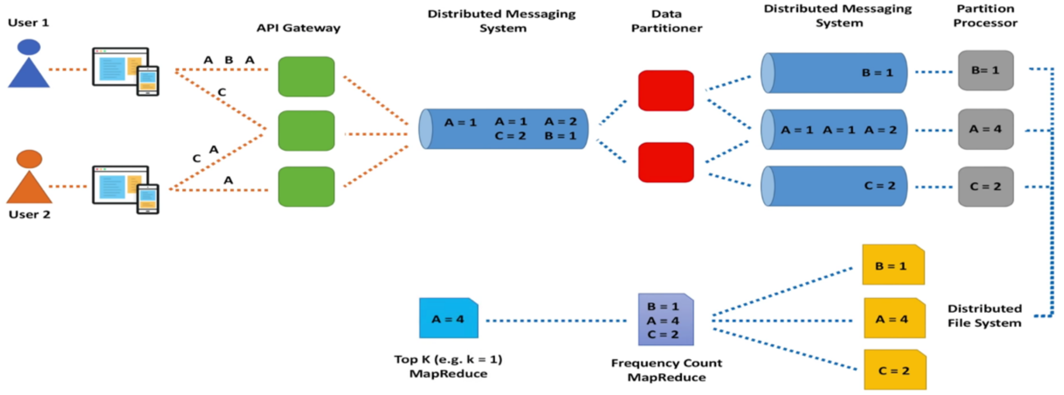
Example-1: Top K Problem, heavy hitters
Trends, popular products, most active/frequent, ddos attack (ip heavy hitter)
a), Functional: topK(k, startTime, endTime)
Non-functional: scalable (deal with increasing volume), highly available (survives failure, no single point of failure), highly performant (tens of milliseconds to return results), Accurate
b), Topk problem simple solution (sacrifice of accuracy)
Count-min sketch (height, width)

How to retrieve data: return minimum of all counters for the element
How do we choose width and height? accuracy and probability
Count-min sketch returns frequency count estimate for a single item; use CMS to count and use heap to store the topk list, heap is always present/needed
Where is topk list in the picture? Replace hash table with fixed size storage
CMS is to save memory, still need nlogk to get top k, nlogk for heap + klogk for sorting
Compare with hashtable, scale well with different scenarios, many different topK lists, need new sketch to count different event types
c), High level architecture
User -> Gateway -> distributed message system (kafka)
    -> fast path (short time) -> fast processor -> storage (sql/nosql)
    -> slow path
        -> data partitioner -> distributed messaging -> partition processor
        -> map reduce (s3/hdfs) -> frequency count -> topk job
Data partitioner: hot partitions (id + time window)
API gateway cons: Frontend, CPU/mem constraints, parsing/aggregation on separated cluster
Alternative to count-min sketch, countering based
How big is the k? several thousands, network bandwidth, storage
Drawback of this architecture? **Lambda** architecture, to build stream processing application on top of map reduce and stream processing engines, batch/stream processing, complexity
Depends on data volume: kafka+spark/heap merge, in memory aggregation

High-level architecture

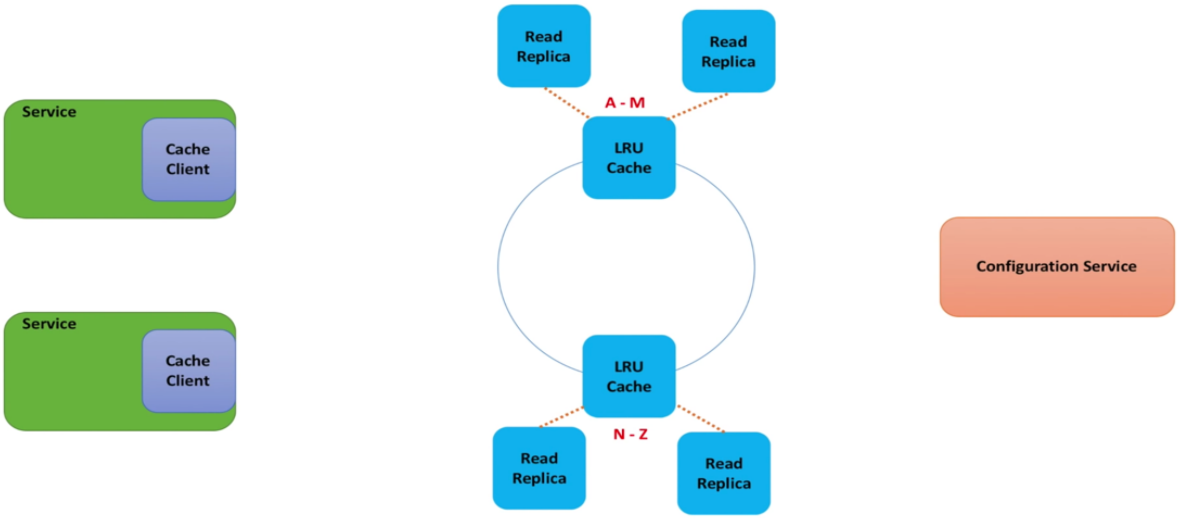


Data flow, slow path



Example-2: Distributed cache
Functional: put(key, value), get(key)
Non-functional: scalable (scale out), highly available (survives failures, consistency for CAP), high performant (fast), durability (data persistence)
Start from simple -> evolve with complicated (hash table/local cache), LRU cache
Dedicated cache cluster: isolation of resources between service and cache, can be used by multiple services, flexibility in choose hardware
Co-located cache: no extra hardware and operational cost, scales together with service
Consistent hashing (host shard)
Cache client, know all servers, have same list of services, store list of servers in sorted order, binary search, cache miss if network failure
Configure management: config file in s3/daemon pull, configuration service (zookeeper)
Achieving High availability: leader -> read replica (different data centers)
Available/hot shard/replication: probability protocols (gossip), eventual consistency; consistency protocol (2 phase commit), strong consistency;
Leader selection (configuration service, zookeeper)
Consistency: replicate data (asynchronously replication, different list of servers)
Data expiration: (time to live, actively expire, passively expire, probability algorithm)
Local (LRU, guava) and remote cache
Security: not exposed to internet, firewall to restrict access, ensure list of approved clients, encrypt on storage, decrypt when reading
Monitoring and logging: latency, network io, CPU, cache miss rate; (who/when/key/status)
Cache client: maintain server list, [handling shard and routing], handle failure, emit matrix, proxy
Consistent hashing: (may have domino effect – Add multiple server on cycle multiple times)

Summary



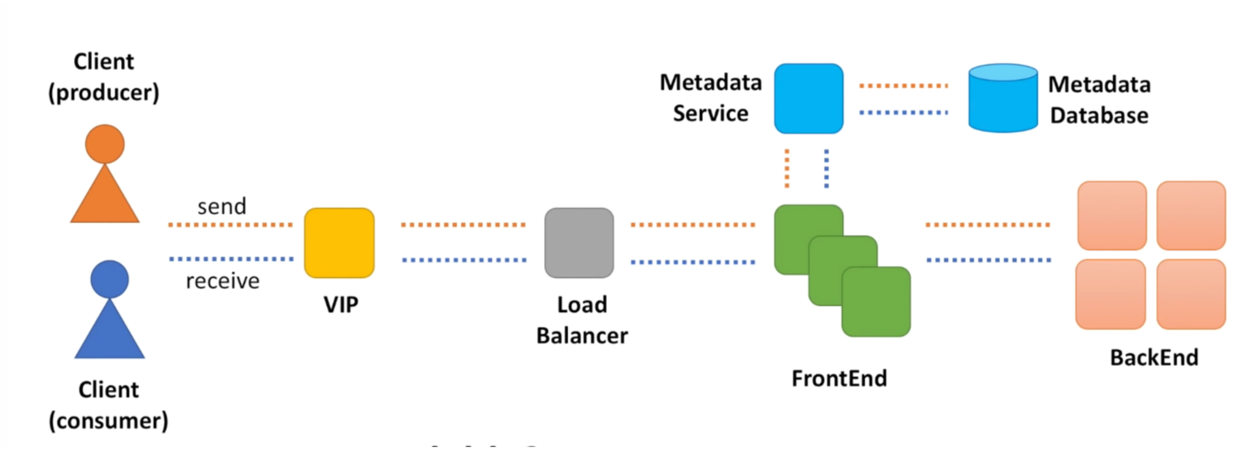
Example-3: Distributed message queue:
Functional: sendMessage(messageBody), receiveMessage()
Non-functional: scalable, highly available, high performant, durability, throughput
VIP and LB: High availability concern, primary serve the requests, the secondary monitor the primary; for scalability concern, across multiple VIPs/LBs in multiple DCs (also improve availability and performance)
Frontend service: light web service, stateless service deployed across several data centers.
Actions supported:

request validation (ensure parameters, data falls within acceptable range),
authentication/authorization (validating identity), user session management
IP Whitelisting
TLS termination (privacy and data integrity, refers to decrypting request and passing on un unencrypted request to the back-end service, SSL on LB is expensive, usually handled by **separated TLS http proxy** on the same host),
server-side encryption (stored/encrypted as soon as Frontend receives, only decrypts when they are sent back to consumer),
caching (copies of source data, reduce load to backend, increase throughput and availability, decrease latency, store metadata info, store user identity information to save on calls to auth services),
rate limiting (throttling, protect from overwhelmed, leaky bucket algorithm),
request dispatching (bulkhead pattern helps to isolate elements of application into pools so that if one fails, others can continue to function, circuit breaker pattern),
request deduplication (issue for at least once semantics, use cache),
usage data collection (for audit and billing)

Metadata service: caching layer between frontend and persistent storage
Backend service:
Where and how do we store message? High throughput requirement, **RAM and local disk of backend host**
How do we replicate data? Replicate with in a group of hosts
How frontend select backend host? Metadata service

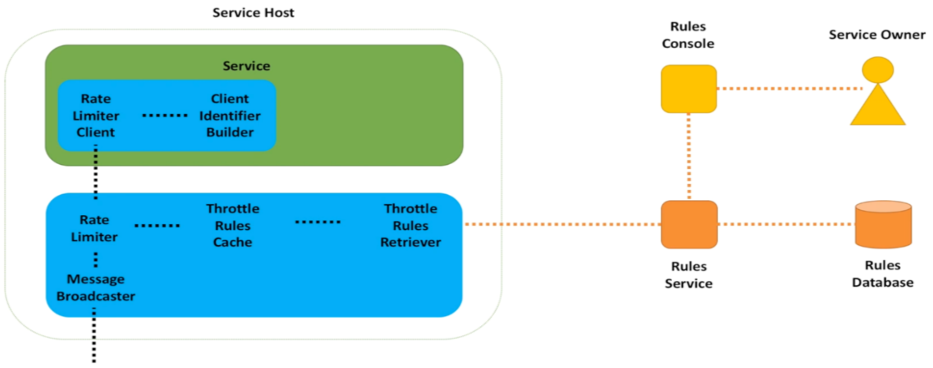
1) Leader – follower relationship (in-cluster manager, zookeeper)
2) Small cluster of independent hosts (out-cluster manager, between clusters, Cassandra)

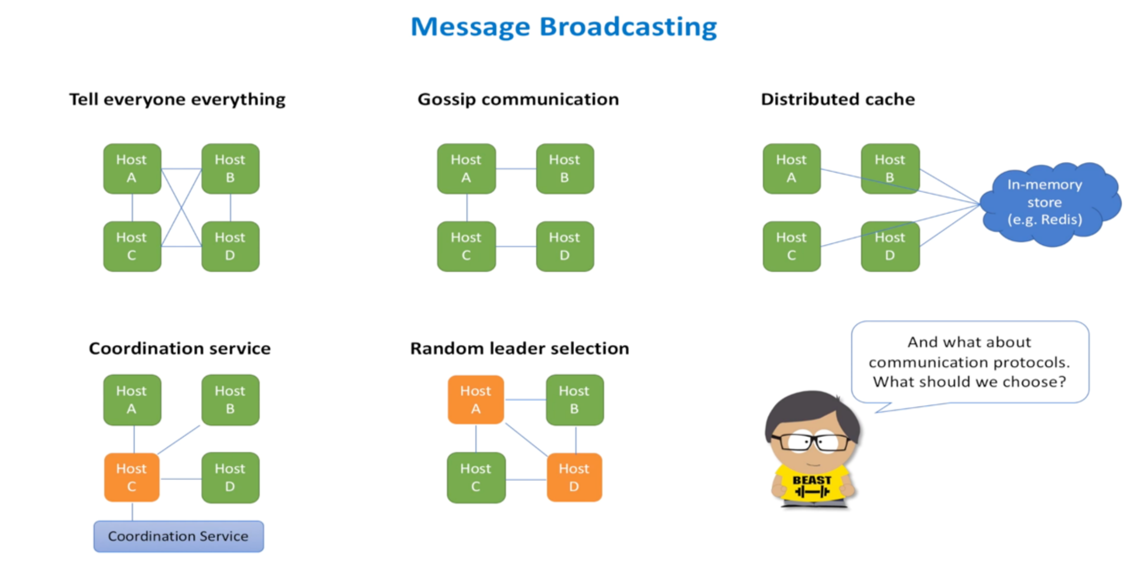
Queue creation and deletion, through command utility
Message deletion, order of msg, kafka (consumer responsible for deletion, track position/offset in log) /SQS (message marked as invisible, consumer call delete API)
Message replication (sync – high durability with high latency, async-performant but may fail)
Message delivery semantics: at most once, **at least once**, exactly once
Push (notified with new msg) vs pull (constantly async, easier to implement)
FIFO, hard to maintain order, does not guarantee order, or has throughput limit (maybe slow)
Security, encryption, SSL over http; encrypted on storage
Monitoring: health of system components, visibility into customer experiences (state of queues); write logs



Example – 4, Rate limiting
Functional: allowRequest(request)
Non-functional: fast, accurate, scalable, easy integration
Start with simple solution (single server); Token bucket algorithm; Object-oriented design of the solution
Message broadcasting: tell everyone everything (not scalable), Gossip communication (random peer selection, sharing data), distributed cache cluster (in-memory store, redis); coordination service (leader selection); random leader selection;
How to integration with service: as part of the service, library integrate with service (faster, resilient to inter-process call failures); rate limiter client with separated rate limiter process daemon (programming language agnostic, use its own memory space, makes it easier to deal with service teams paranoia, not impacting service itself, auto-discovery)
Bucket sizes, bucket can be removed if no recent requests
Failure scenarios: daemon failure (not throttled)
Config rule management: introduce self-service service, service owner manages
Synchronization: token bucket, thread safety, token bucket cache (concurrent hashmap)
Client options: resent, retry (exponential backoff) with jitter (random delay)
Depends: number of hosts/rules/rates, separated cluster (with distributed cache)

One final look





Example 5, Notification service

Functional: createTopic(topicName), publish(topicName, message), subscribe(topicName, endpoint)

Non-functional: scalable, available, performant, durable

Common pattern: LB -> Frontend -> Metadata Service -> Metadata Database

Frontend service host:

Frontend service, local cache

Local disk (service logs agent, metrics agent, audit logs agent

request -> reverse proxy (TLS termination, compression, handle errors 503)

Log data processing (agent, data aggregation, post processing)

Metadata service:

Distributed caching layering, between frontend and persistent storage

Many reads, little writes

Service discovery: Frontend -> Configuration Service (Zookeeper) central registry; Gossip protocol (random peer selection)

Temporary storage: fast, available, scalable web service, store several days (persistent)

Databases: ACID, easily scale; SQL vs NoSQL; column, key-value (Casandra/Dynamo)

In-memory storage: need persistent storage support (redis)

Message queues: apache kafka, Amazon SQS

Stream processing platforms: apache kafka, kinesis

Sender:

Message retriever from temp storage (track ideal threads, use counting semaphore to restrict the number of read message threads)

MS client -> Metadata service

Task creator -> Task executor; Split into tasks; tasks (call http endpoints, send email, send SMS, mobile push)

Spammers: register, agree, verify with email

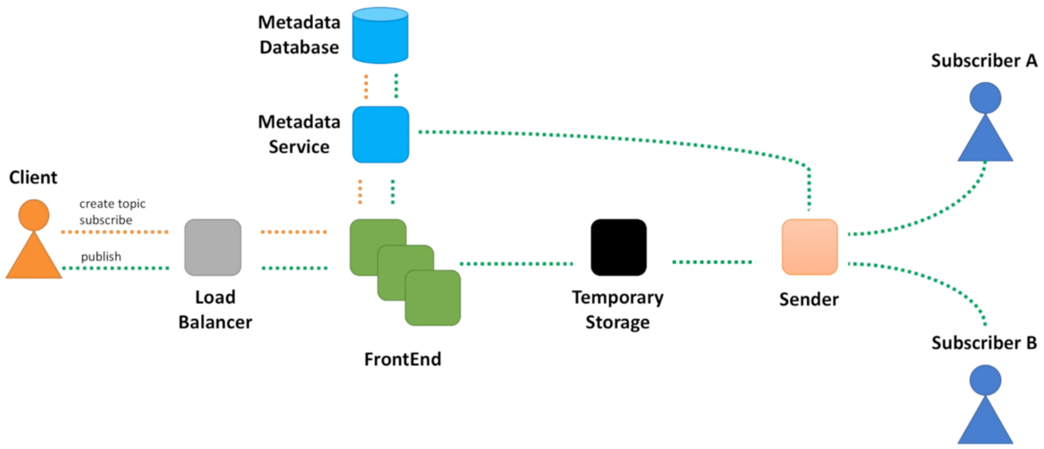
Duplicate message: sender/subscriber

Retry of delivery attempt: max retry, monitor with undelivered

Message order: FIFO, delivery failure, retry

Security: authenticated, only registered pub/sub, encryption during transfer/storage
Monitoring: number of messages

High-level Architecture



Common Distributed Systems:

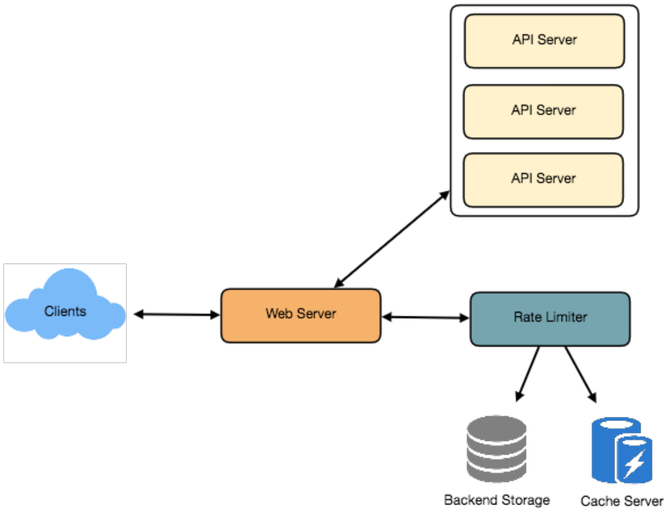
GFS: SSTable, index, chunk server
BigTable: NoSQL DB column based on GFS, write ahead log, memTable in mem
    master server (GFS sharding, tablet server check)
    lock server (client communication, chubby/zookeeper, metadata to find key server)
    bloom filter: check key existence in SSTable index

Cassandra: NoSQL, P2P, column key, consistent hashing (save on 3 following virtual nodes for data replication); CAP Theorem, Gossip protocol:

Write: logging data in commit log, writing data in memtable, flushing data from memtable, storing data on disk in SSTable

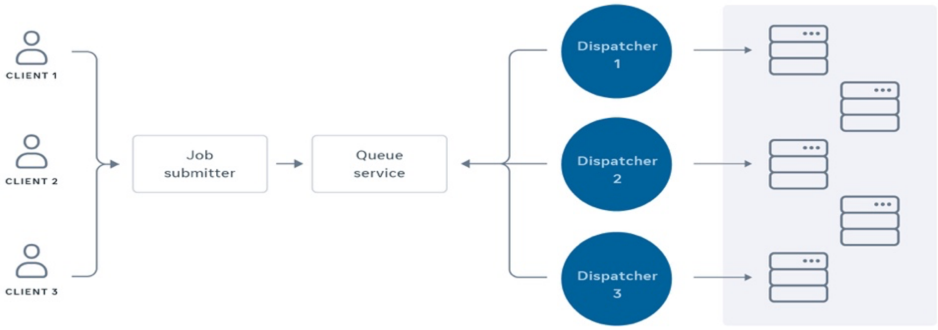
Read: check memtable, check row cache (if enabled), check bloom filter, check partition key cache (if enabled), partition summary -> partition index -> compression offset, locate data on disk using compression offset map, fetch data from SSTable on disk

API Rate Limiter:
Bucket token, leaky bucket, fixed window, sliding window, sliding window count



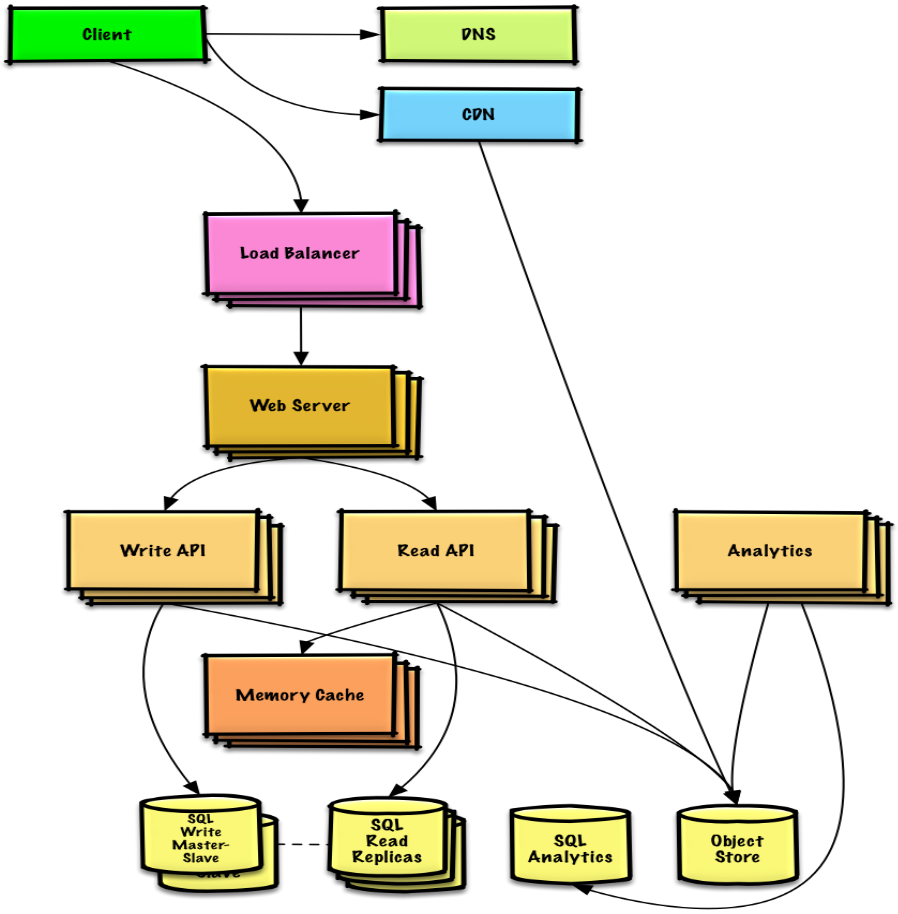
Atomicity problem: use Redis **lock** during read-update operation (could add latency)
Sliding window: use Redis sorted set, takes a lot of memory
Sliding window count: use Redis Hash, each request increments a counter in the hash, also sets the hash to expire an hour later
Caching recent active users, write-back cache by updating all counters and timestamps in cache only, the write to permanent storage can be done at fixed intervals (later)
Rate limit by IP (shared IP/gateway for many users) or user (works after authentication, not applicable for login API, DOS attack against user); Hybrid approach, requires more memory and storage

Job scheduler (Async):

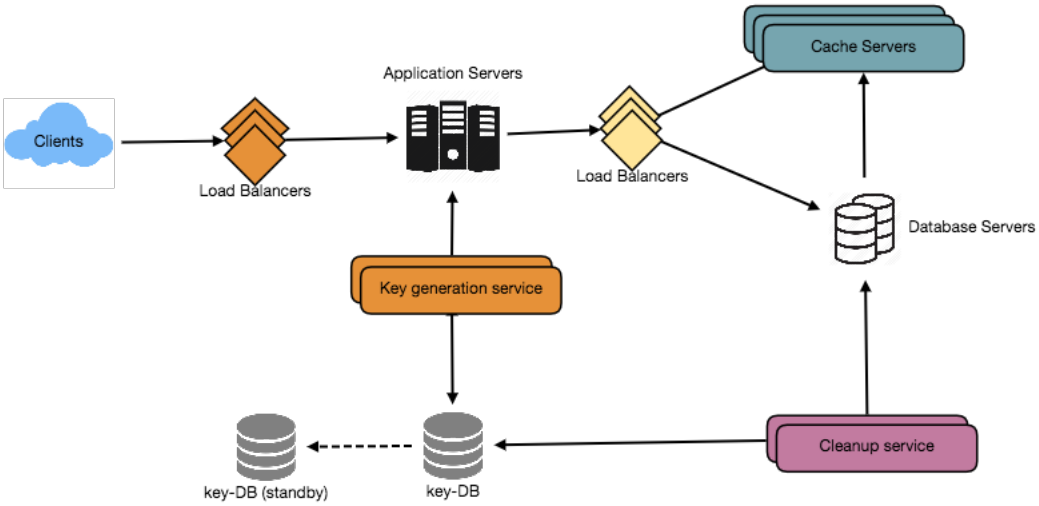


Major challenges: Prioritization (don’t block other jobs), Capacity optimization (daily predicable traffic, major events traffic, short-lived incident response traffic), Capacity Regulation
Delay tolerance: infer priority by understanding how long each request can be delayed
Optimize capacity: queuing (multiple queues), time shifting (predictive/deferred compute), batching (accumulate multiple jobs into mega job, efficient with cache reuse and code warmup)
Capacity policy: quota and rate limiting, allow incident response requests for short periods

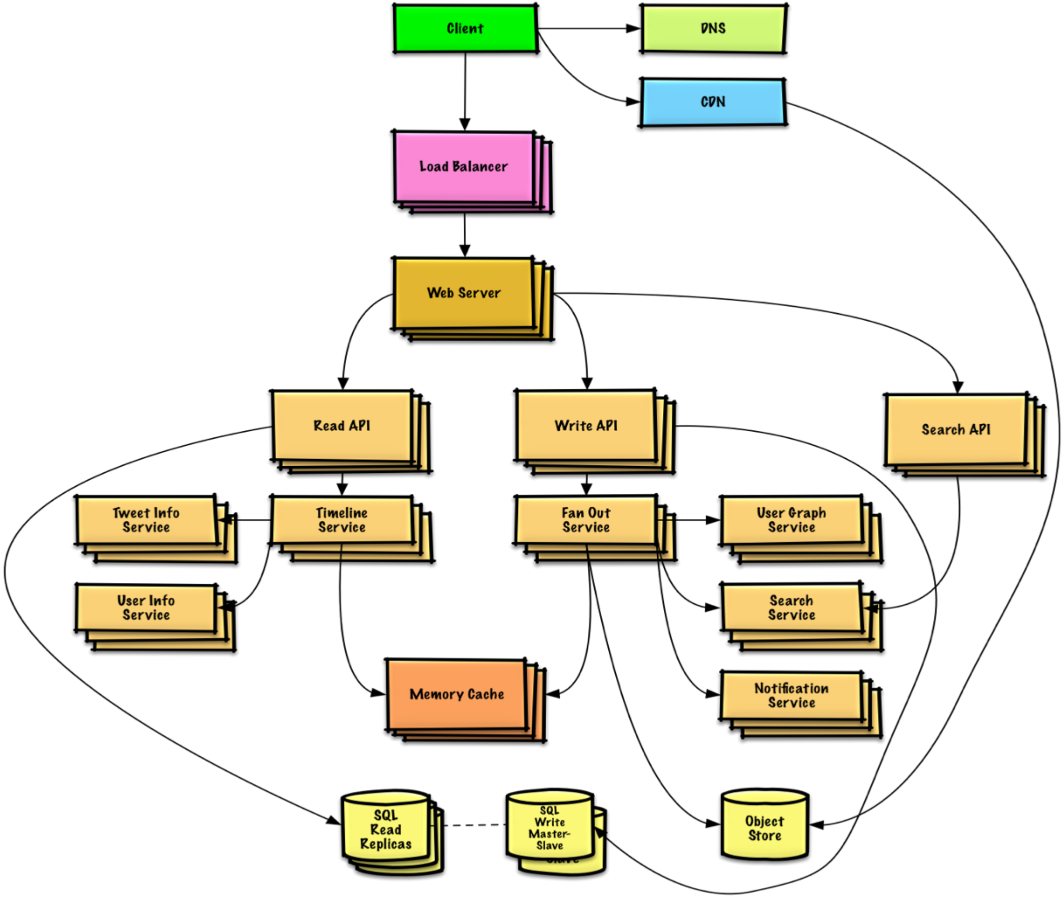
Pastebin:



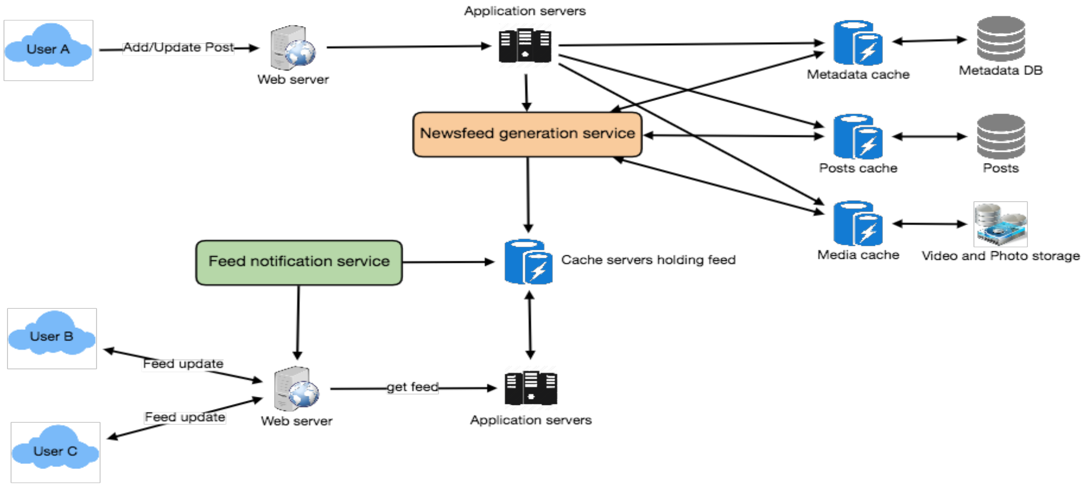
Separated key generation service
Async cleanup service to remove expired data
Object storage vs metadata database
Load balancer between client/server, server/database



Facebook news feed and search, Tweeter, Instagram:



User, Entity/Organization/Page/Group, UserFollow, FeedItem, FeedMedia, Media, Post
Sharding based on timestamp and postID: (epoch seconds, incrementing sequence), quickly find the latest posts/tweets, low write latency (secondary index on creation time), low read latency
Intelligent cache: LRU, 80-20 rule, try to cache 20% of daily read volume from each shard



Feed generation: adjust number of feed items according to pattern, use LRU cache to remove users from cache, only generate feed for recently active users

Push/fanout mode for normal posts, async/offline feed generation

Pull mode for celebrity/popular posts

Feed publishing:

push, fanout on write, maintain long poll request; http long poll, WebSocket

pull, fanout on read, latency and frequency issue;

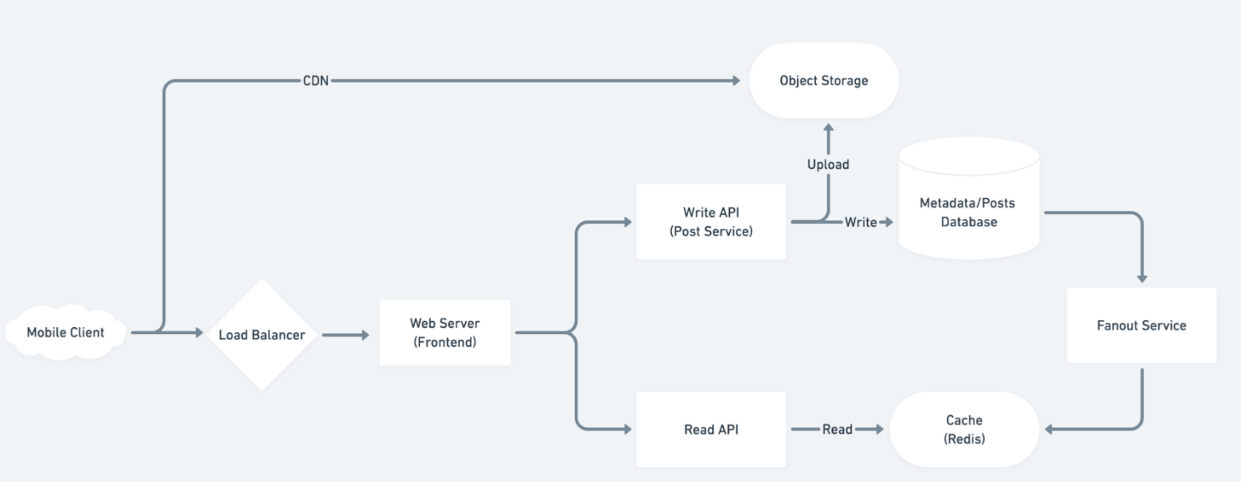
Hybrid, pull for celebrity, push for online friends; push to notify, pull for serving

For mobile user, let users pull to refresh to get new posts

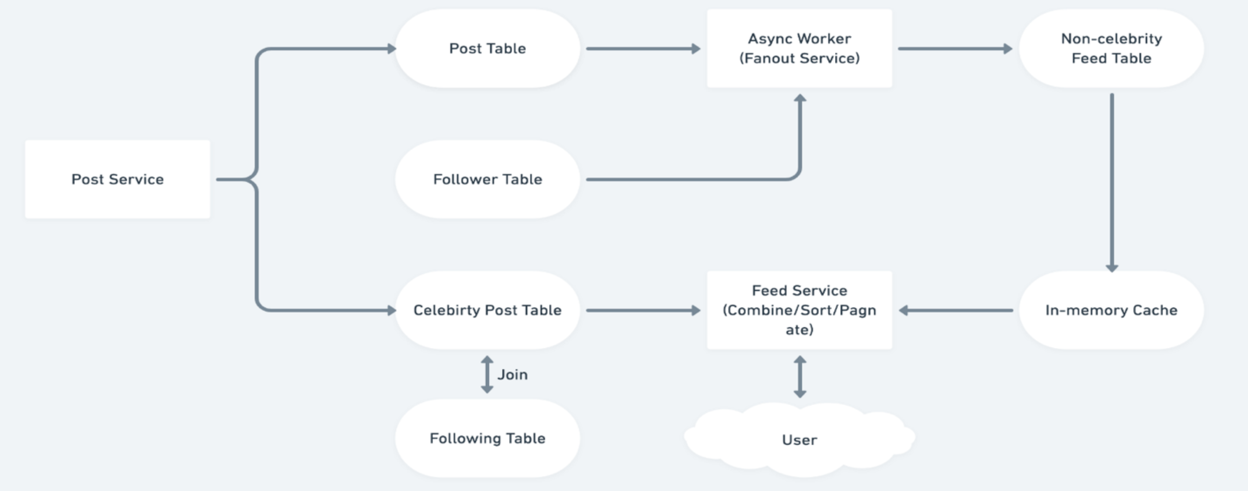
Feed ranking: by creation time, signals of importance (number of likes, comments, shares, time, video/image); constantly evaluating based on user stickiness, retention, ads revenue

Data partitioning: Shard original posts and metadata, by timestamp and PostId, can't range query, need aggregation server, can easily fetch latest posts; Shard aggregated feed data, based on UserID, storing data of a user on one server, store in memory

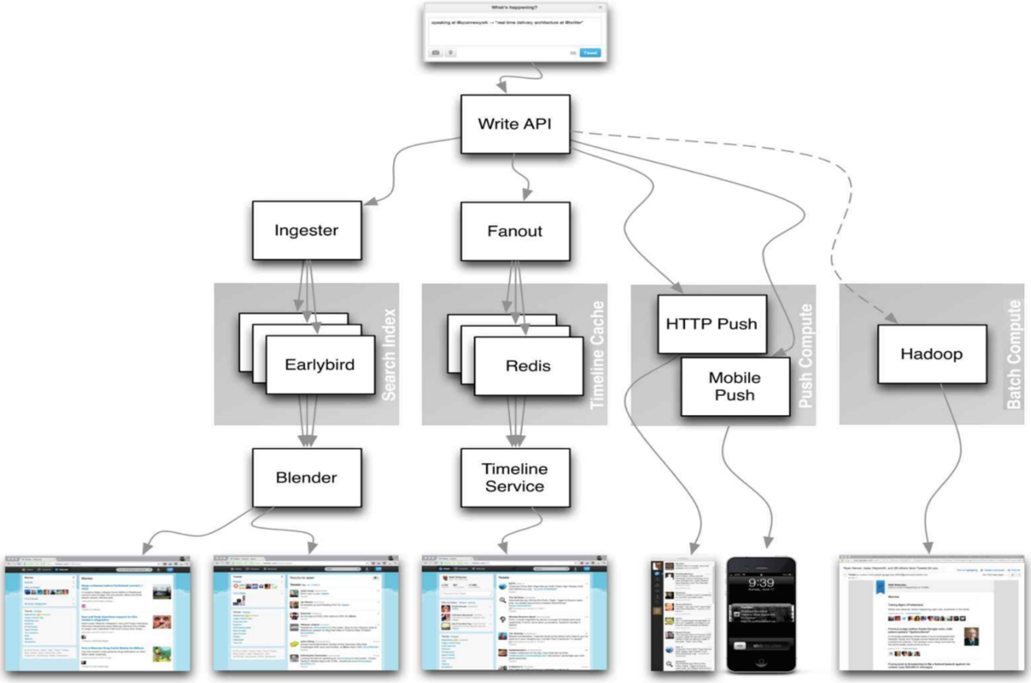
Instagram major components:



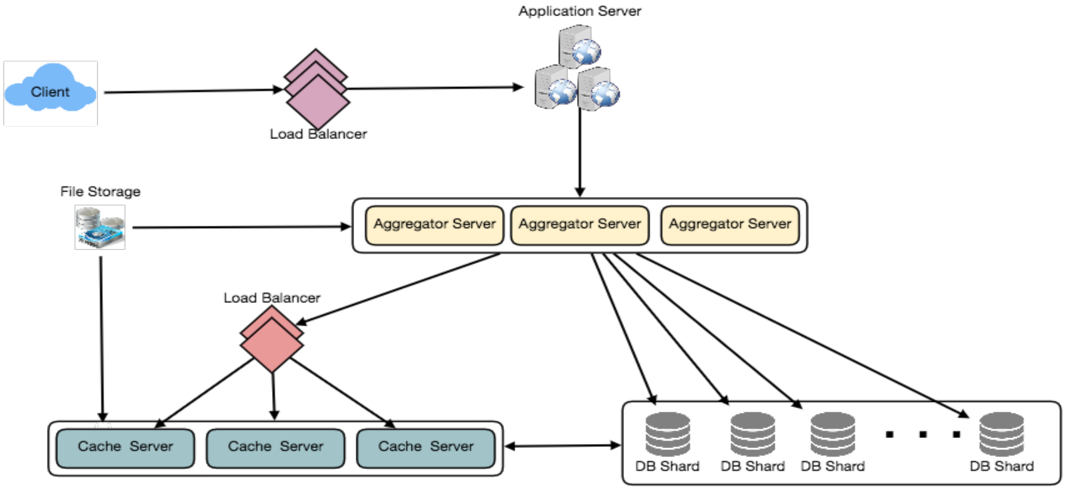
Uploading can be slow and can consume all available connections. This means reads cannot be served if the system gets busy with all the write requests. To handle this bottleneck, we can split reads and writes to ensure that uploads don't hog the system. Separating photos' read and write requests will also allow us to scale and optimize each of those operations independently.



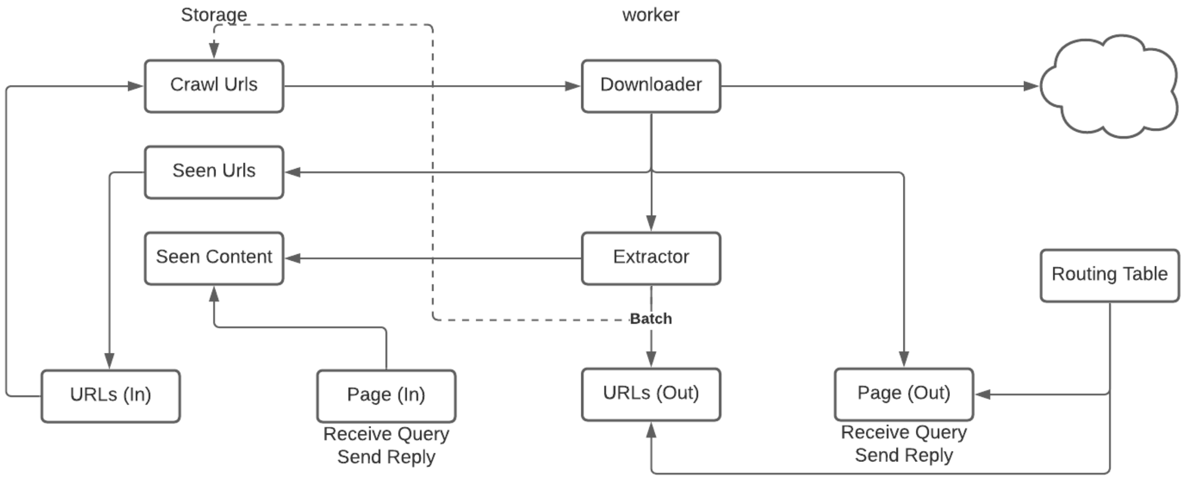
Twitter:



Shard by UserID: hot user
Database tables: Tweet, User, UserFollow, Favorite
Shard by TweetID: high latencies for user's tweets
Combine **tweet ID and creation time**, include epoch time in TweetID: lower latency for read (epoch time included in key) and write (no secondary index on creation time)
Cache: hot tweets and users, most recent data (last 3 days)
Fanout (push) vs Pull (for celebrity)

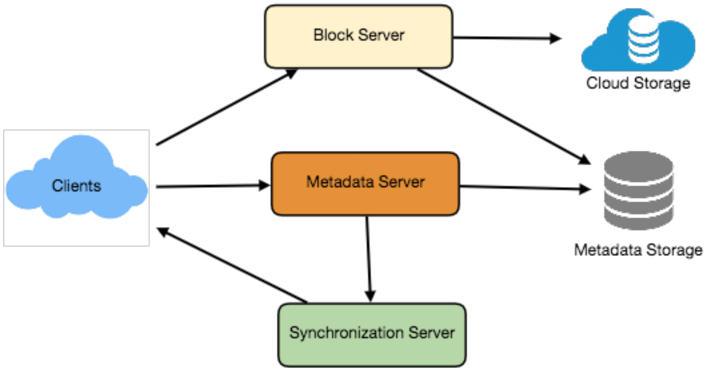


Web crawler (Centralized):
BFS (for different hostnames) or DFS (within the same website)
Crawler can have distinct FIFO sub-queues on each server, one worker downloads from a given web server, not overload target with the queue
URL frontier with separate buffers for enqueue/dequeue, store URLs on disk
Document Input stream, cache downloaded document locally with abstraction, in memory or on disk; Use md5/sha checksum to remove duplicated documents
Check point to guard against failure, storing FIFO queues on disks, dump snapshot of all data



Decentralized crawler (10k hacked/botnet machines):
Main components: Downloader, URL Extractor, Duplicate Tester
Data structures: Crawl-jobs, Seen-URLs, Seen-Content
DHT: a data structure to store data and routing information without centralized server, each node is both a client and a server; store the mapping between particular key and value in a distributed manner, by maintaining a small routing table at each node. To achieve this, each peer is given an identifier and is responsible for a set of keys. Assignment is achieved by normalizing the key and peer identifier to a common space (consistent hashing); Ping, Store, Find_node, Find_value
Lookup (URL->peer) hash the URL, look up the value, return IP of peer responsible
New peer join/leave: new peer split workload; map key (URL/content hash) to multiple nodes
Peer membership: contact existing peer (entry node, bootstrapping entry node, DNS->bootstrapping nodes, maintaining currently alive nodes), periodic (UDP) message exchange between neighboring peers to identify if peer joined/left
Bloom filters: false positive rate with of 0.001, using 5 hash functions with m/n = 10
1billion URLs: 1*10^9*(10bit): 1G; per machine: 1*10^9/10^4 = 10^5Bytes = 100kB
Hashset: 10^9/10^4*100B (ave url len): 10MB
Consensus protocol (Raft, Paxos) to indicate finish, pass around counter, gossip protocol
Master node, find node to continue the failed nodes: heartbeat to worker, save progress on master
Paxos (proposers, acceptors, learners); Raft (leader election, log replication),
Different P2P algorithms (**routing table, join, leave, copy, find closest**): Chord [dis(a,b)=(b-a) mod 2**N], CAN, Pastry (prefix based routing table), Kademlia (logN lookup time,160-bit IDs; [dis(a,b)=a xor b]); Tapestry (routing table match with prefix, 128-bit IDs)

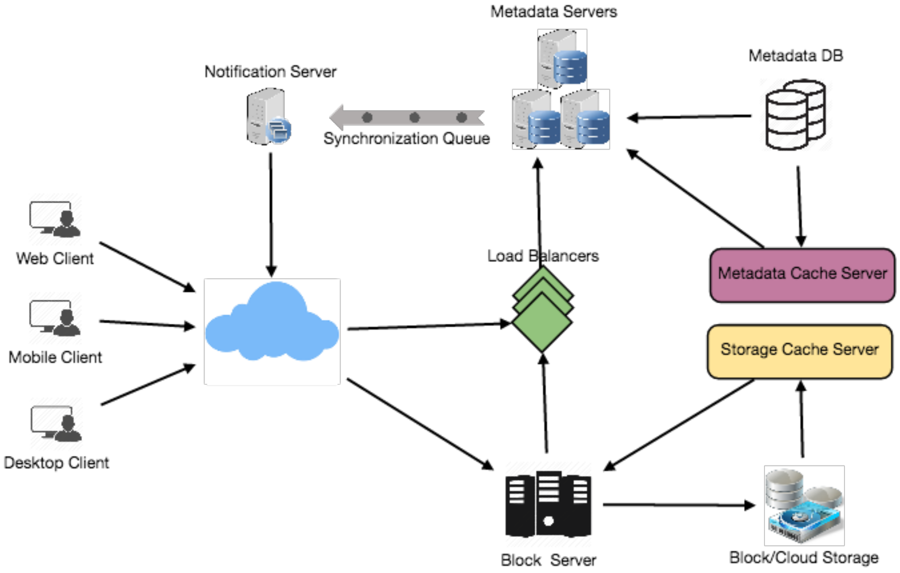
Dropbox:



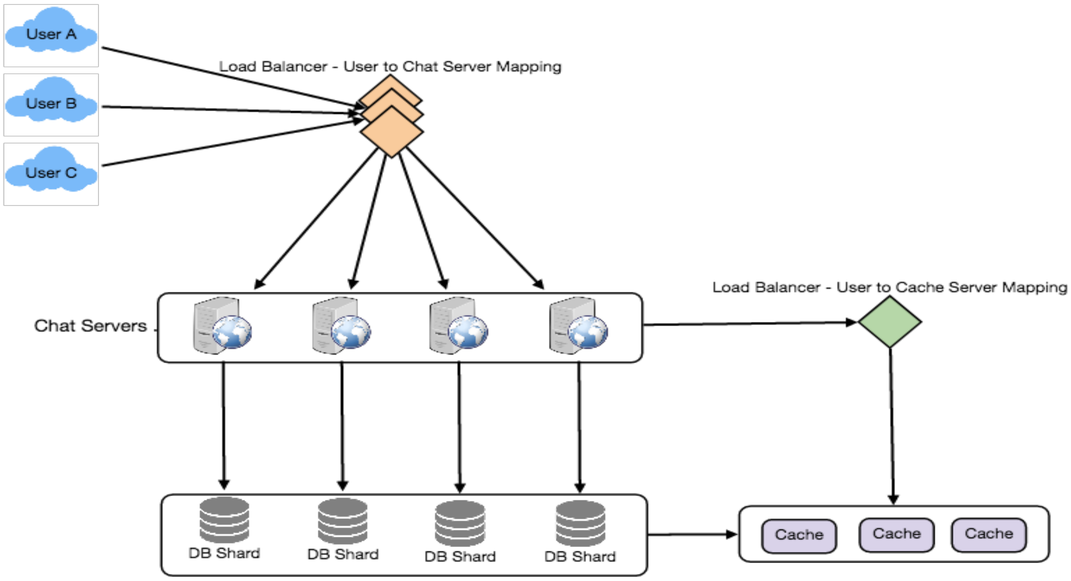
Metadata DB: User, devices, workspace, files, chunks; ACID
Synchronization service: check metadata DB for consistency then process file updates by client, apply changes to other clients, synchronize client changes to server metadata DB
Message queue service: separate read/response queue (client pull) and write/request/update queue (client push)
Inline deduplication: save network and storage usage, apply early
Metadata partitioning:

- Vertical partitioning (user/chunks), slow for aggregation,
- Range based partitioning (A/B/C), unbalanced server with hot partitions,
- Hash based partitioning, consistent hashing, preferred option

Caching: metadata cache, block storage cache, LRU eviction
LB: client -> block server; client -> metadata server,
Security: privacy and security of their data, storing permissions of each file in metadata DB
Logging and monitoring:

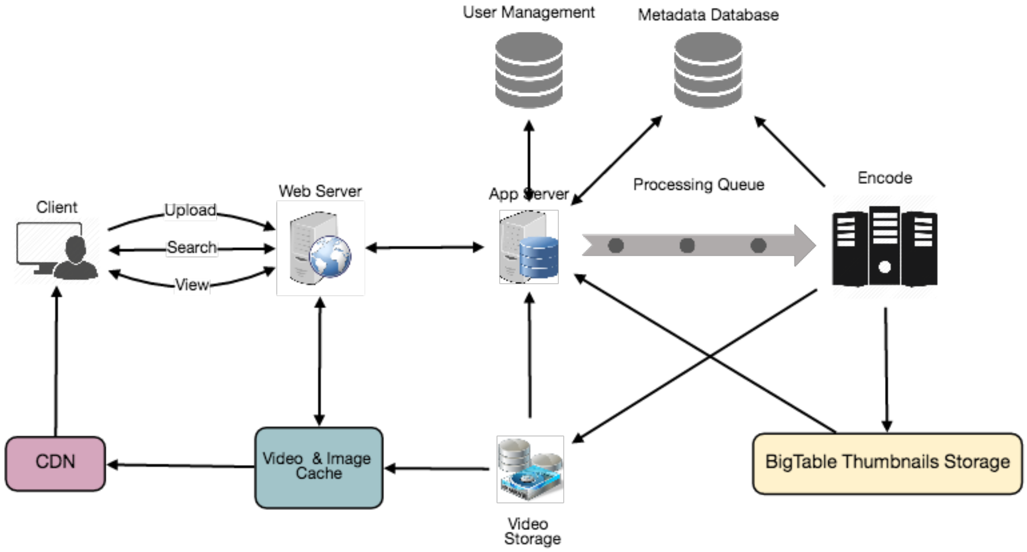


Facebook Messenger:
Pull vs Push model: keep a connection open with server (Long Polling, WebSocket)
Status update: pull/broadcast on startup, change on failure, pull status for new chat
Message storage: high rate of small updates, fetch range of records quickly (Cassandra/MySQL)
Sequence number: logical clock to identify operations, typically using counters that are incremented for every operation, the (single) leader increment a counter for each operation and assign a monotonically increasing sequence number to each operation in the replication log.
Single-leader replication: The replication log defines a total order of write operations.
No single leader: a) each node generates its own sequence number; b) attach physical clock timestamp (if high resolution); c) preallocate blocks of sequence number
Lamport timestamps: ensure total ordering of messages, format (count, node ID), every node and client keeps track of maximum counter seen so far, and includes the maximum on every requests, when a node receives a request or response with maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.
Total order broadcast: the idea of knowing when your total order is finalized. The log ensures that all consumers see messages in the same order, a stream process consumes all messages in log partition sequentially on a single thread, unambiguously and deterministically decide which one of several conflicting operation came first


User A
User B
User C
Load Balancer - User to Chat Server Mapping
Chat Servers
Load Balancer - User to Cache Server Mapping
DB Shard
DB Shard
DB Shard
DB Shard
Cache
Cache
Cache

Online status: distributed memory cache to store status (last login, heartbeat timestamp); pub/sub model, publish status change to user specific queue, push status change to all subscribed users
Use last in-chat timestamp, to record read progress of the group message, use batch-update if messages continuing to flow in the chat
MySQL tables: User (uid), Group (gid), Message (mid)**;** Storage: HBase/MySQL (rocksdb, storage/write optimized)
Append only in message table, treat like/delete as new messages, client only looks forward
Purge history data asynchronously to cleanup old/deleted messages
Offline clients: rely on mobile notification service
WebSocket/long poll persistent TCP connection: server checks receive online status (in cache) and find the server hostname maintaining connection, send async notification for offline user
Always commit data into database (source of truth) first

Youtube/Netflix:



Functional: upload/view video, search video title, records stats, add/view comments

Database tables: User, Video, Comment

Separate Read and Write traffic

Metadata replication: primary for write, secondary for read

Object Storage for video, thumbnails

Video encoding: read from processing queue, encode into multiple formats, asynchronously notify users when finishing

Video dedup early: save data storage, caching, network usage, energy consumption

Use CDN to distribute videos to geographically closer location, save less popular videos locally

Cache: for hot metadata rows, LRU eviction strategy, intelligent cache with 80-20 rule, predictive model based on usage pattern, videos from stars/celebrities

Typeahead Suggestions: Trie

Store the count of searches that terminated at each node

Store the top suggestions at each node, only store reference of phrase

Use offline pipeline to calculate the frequencies

Two copies on each server, switch once update finishes

Exponential moving average, give more weight to latest/new data

Remove term, remove during regular/lazy update, filter layer before return

Other suggestions: trending, top, history, freshness, location, language

Store it in a file (serialization), recalculate top suggestions and merge results

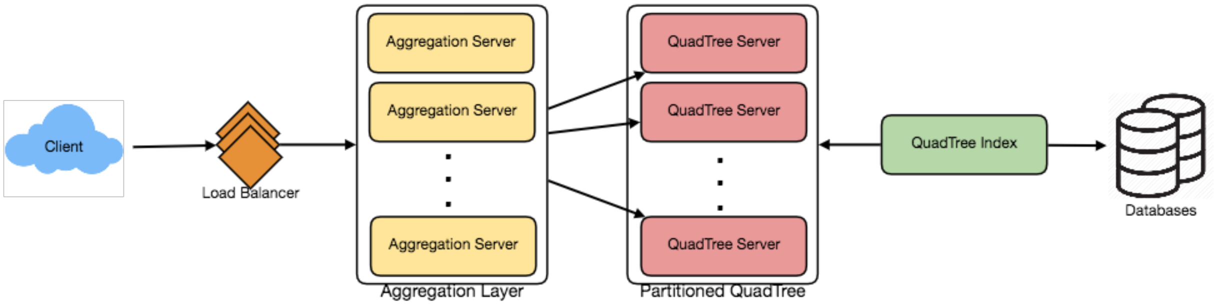
Partition dynamically based on maximum capacity of the server

Primary, secondary take over after failover, primary rebuild based on last snapshot

Establishing an early connection

Store personal history of user separately on server and cache on the client

Yelp:



Geohash (with MySQL): geographical hash based on latitude and longitude
Grids: Based on a given location/radius, find neighboring grids, then query these grids to find nearby location IDs, then query backend database to find details about those places
Dynamically adjust grid size: QuadTree, tree structure in which each node can have 4 children
Leaf node contain locations, connect leaf nodes with doubly linked list, or through parent nodes
Dynamic size grids (QuadTree): dynamic break down into four grids of equal size and distribute places among them, store list of places in leaf nodes
Search/find grid for a given location: start with root node, search downward to find target node/grid, check if current node has children
Find neighboring grids of given grid: only leaf nodes contain list of locations,

1) we can connect all leaf nodes with a **doubly linked list**, iterate forward and backward among neighboring leaf nodes to find out design locations;

2) through **parent nodes**, keep a point in each node to access to its parent, each parent has pointers to all of its children. locationID -> backend DB

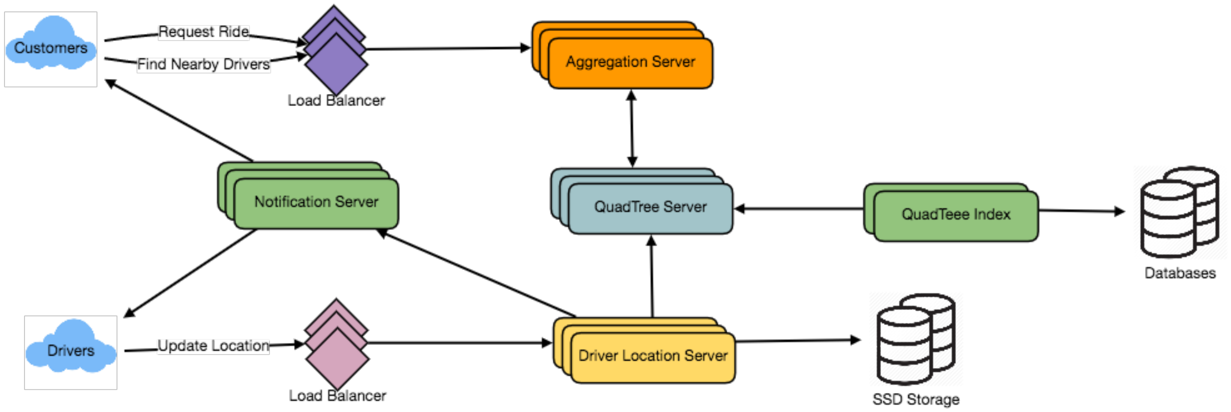
Search workflow: 1) find user location; 2) find neighbors; 3) continue expending to neighboring nodes if not enough designed places
Maximum memory for storing QuadTree, 500M location*24B, 1M grid
Insert new location: add into location database, and update QuadTree; find grid server of the new place and then add it there
Sharding: based on regions, locationID, find set of nearby places then aggregate and return to user, each server contains a world map but with subset of locations in each grid
Replication and Fault Tolerance: replicate vs partitioning to distribute read traffic, take control in case of failover:

If both primary/secondary dies, we need to rebuild QuadTree. To efficiently retrieve a mapping places and QuadTree server, build reverse index that map all the places to their QuadTree server.

Using hashmap on (QuadTree Index Server), key is the QuadTree server number, value is the hashset containing all places being kept on the server, have replica of QuadTree index server
Cache: Adjust cache server based on client usage pattern, LRU eviction policy

Solution for “Hot” objects: 1) Adding cache in front of database; 2) Using alternative sharding mechanism to evenly distribute the load (need aggregation server to process search query)

Uber:



Procedure: 1) Drive regularly notify server about their current location, 2) Passenger see all nearby drivers, 3) Customer request ride, driver get notified, 4) Driver accept ride, constantly update current location until trip finishes, 5) Driver marks journey complete for new availability
Differences between Yelp: need to frequently update QuadTree, **the purpose of QuadTree is to find nearby drivers/places efficiently**.
DriverLocationHT: Keep driver location in hashtable and update QuadTree less frequently
For scalability, performance, and fault tolerance, we should distribute DriverLocationHT
Driver location server:

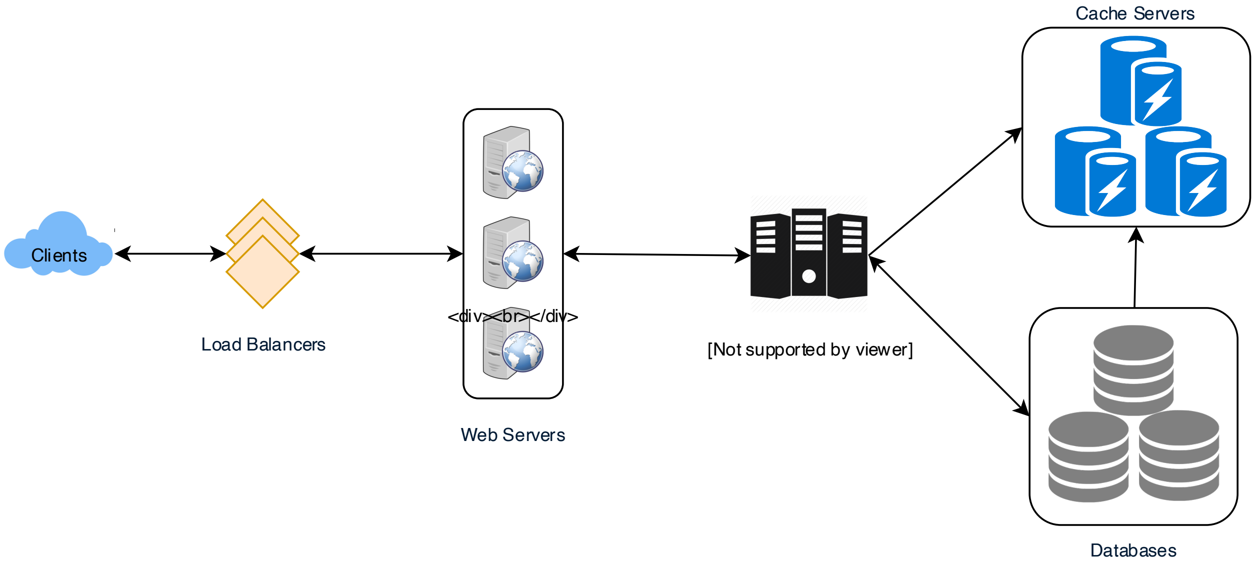
Driver submit new location, Driver location server push mode with notification service,
Broadcast current location of drivers to all interested customers,
Notify QuadTree server to refresh driver location

Publisher (driver)/subscriber(customer) model:

Client open app, pull nearby driver
Return list of nearby drivers to client
Subscribe customer for all the updates from nearby drivers
Maintain list of subscribe for driver, broadcast current location to subscribed customers

Notification service: HTTP long polling or push notifications
Client pull every five seconds to get updated driver list for **new drivers**
Grid can grow/shrink an extra 10% before partition/merging, reduce load on high traffic grids
Store in persistent storage for easier recovering
Request ride procedure: 1) customer send ride request, 2) Aggregator server take the request and ask QuadTree server to return nearby drivers, 3) Aggregator server collect and sort/return the list of drivers, 4) Aggregator server send notification to top drivers, whoever accept first will be assign the ride, other drivers receive cancellation requests; Aggregator request ride to next top drivers if no one accepts in first batch, 5) notify customer if driver accept request
Fault tolerance and replication: Primary/secondary servers for driver location server and notification servers, and store data in persistent storage
Ranking: store ratings of each driver in database and QuadTree

Ticketmaster:



Application server to handle ticket management:
Movie, Show, Booking, User, Cinema, City, Cinema_Hall, Cinema_Seat, Show_Seat, Payment
Daemon services: ActiveReservationService, keep track of all the active reservations that haven't been booked yet, and expiry of active reservation; Waiting UserService, keep track of all waiting customers
Concurrency control: use transaction, within a transaction, if we read rows, we get a write lock on them so that they can't be updated by anyone else
Design strategy, Classical pattern, message queues, database, hashtable

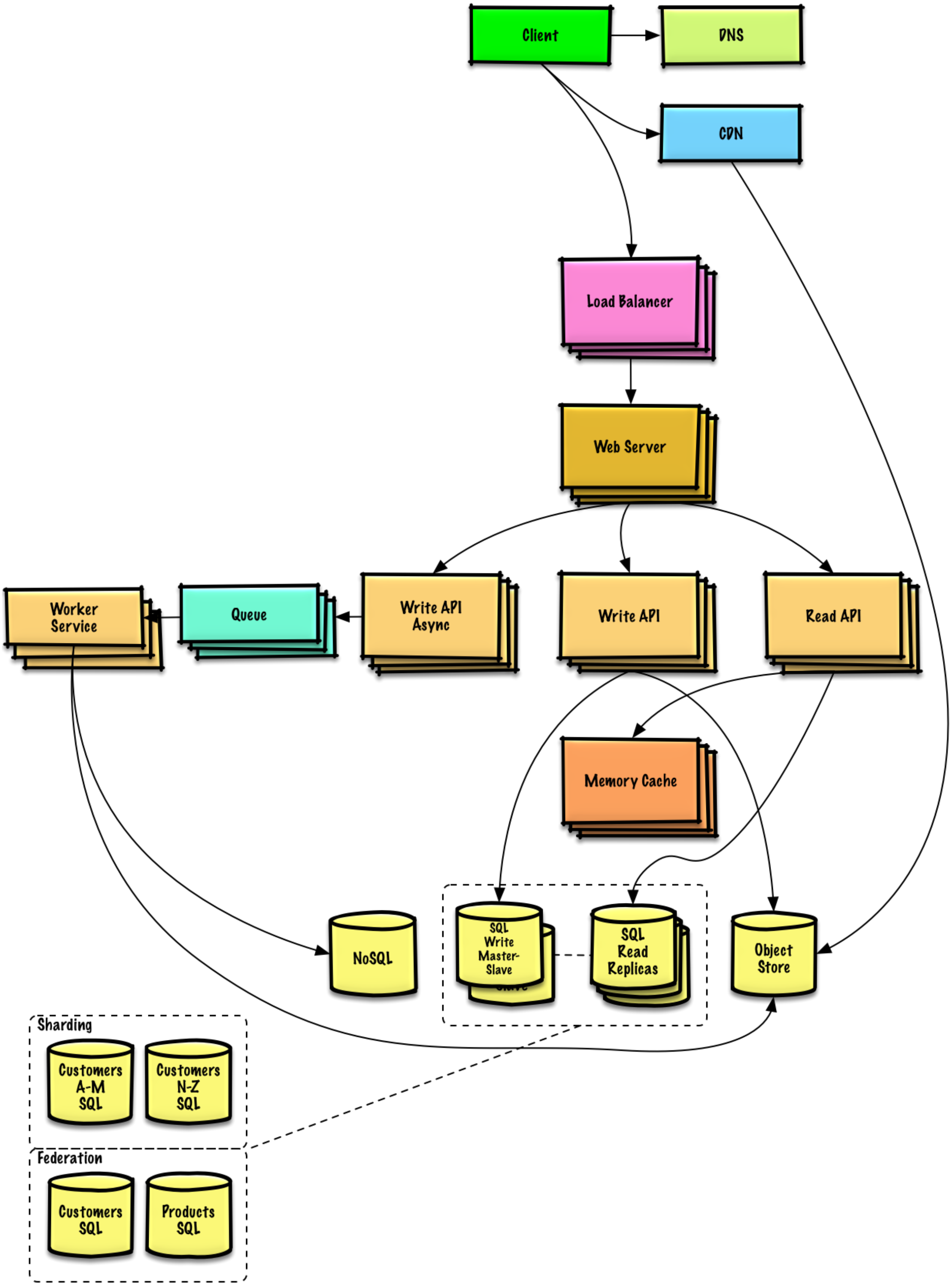
Design craigslist
User, product catalog, post metadata, index/search, metadata database/object storage

Design log collection and analytic system
App servers - > Kafka - > Cloud storage (s3)
High availability, minimum data loss, horizontal scalability, low latency, minimum ops overhead

Design ad click count (similar to topK/rate limitor)
Use time, counts array, timestamp % window = index
Click -> Front End -> action events tracking endpoint -> logging pipeline (Kafka) -> data store and counting service

Facebook Live:
Broadcast client –(BTMPS)→ POP → Encoding Server (Data Center) → FB EDGE FANOUT (CDN) → Playback client
POP and Data Center: Proxygen hosts, BigCache Hosts

Website on AWS with millions of users:



1, problem exploration

Find the exact **requirement** and **scope** of the system, end goal (what to focus on)
Business motivation, Number of users, data, functional features (how to define success)
Do not add new features in the middle, speak more but write less
Non-functional requirement: scalability, availability, performant, durability (persistent)

2, quantitative analysis

Estimation: **QPS, bandwidth**, memory/disk **storage**, concurrent connections

3, completeness of the design

High level design, start from simple, gradually enrich
All the components required

4, ability to reason tradeoff

Talk about pros and cons, bottlenecks
How to shard data (user/tweet/group/post)
Data management, store, structure, define data entities, sql vs nosql
Storage options:

- Database: SQL vs NoSQL, dynamodb, cassandra;
- In-memory Storage: Redis (persistence);
- Message Queues: Kafka/SQS;
- Stream process platforms: Kafka/Kinesis

Counting semaphore: to restrict the number of read message threads
Cache, cache eviction policy: LRU, LFU, TTL
Load Balancing
Replication and Fault Tolerance: secondary databases
Fault Tolerance, both primary/secondary dies, build reverse index map/server

5, deep dive,

Sharding, Multi-level sharding
Security: not exposed to internet, firewall to restrict access, ensure list of approved clients, encrypt on storage, decrypt when reading, sanitize input, least privilege, Throttle based on allocated quota
Monitoring and Logging: QPS, latency, success rate, CPU/load, cache (cdn/lb) miss rate
Continue benchmarking, profiling and monitoring the system to address bottlenecks as they come up,