https://www.youtube.com/watch?app=desktop&v=bUHFg8CZFws&ab_channel=SystemDesignInterview

Prompt: Design a system to count youtube views.

How do you deal with ambiguity?

- ask questions -- clear the requirements and figure out the tradeoffs specific to the problem/task at hand
- there are very many solutions but you need to find the best one specific to your task!

I want to learn this. The way to find this not boring/a slog is to get GOOD. If you are GOOD you will enjoy the process. So get good!

Running thoughts

- Always start with the requirements. Then work backwards.

Running questions

- time scale?
  - as far as I understand this, the lower bound on the memory we need is the amount of memory needed to serve all the requests concurrently
  - but what is the time unit of analysis here?
  - we calculate number of requests per second * data per request = data per second
  - so shuold be just design systems to all the data in one second? then flush it out/forget about the data?
- Need to re read the stream processing chapter (Chapter 11)
  - partitioning
  - fault-tolerance and persistence
  - how to handle consumers failing to consume an event?
- Need to re read partitioning chapter (Chapter 6)
  - consistent hashing

Phase 1: Requirements clarification

- Users/Customersdi
  - To help us understand: WHAT data is coming to the system and WHAT we should STORE
  - Who will use the system?
    - eg youtube viewers? or some engineers trying to build ML models? or for per hour statistics?
  - How will the system be used?
    - often?

        * real-time?
- Scale (read and write)
    - To help us understand: HOW MUCH data is coming to the system and how much retrieced-- HOW to store the data
    - How many read queries per second? x how mach data is queried per request?
    - How many write queries? x how much data written per request?
    - How many video views are processed per second?
    - Can there be spikes in traffic?
- Performance
    - To help us understand: what SYSTEM DESIGN do we need, based on WHAT TECHNICAL CONSTRAINTS do we need to hit?
    - What is the expected write-to-read delay?
        * eg if it is several hours -- you can run a batch processing system or a stream processing system
        * but if it needs to be on the fly then batch data processing is not an option
    - What is the expected p99 latency for read queries?
        * if it is supposed to be minimal, then it is a hint that we need to perform counting when we write data itself -- not when we read the data
            · Data must already be aggregated ideally
            · Q: What numbers do you expect to hear here? I.e what number is low latency vs high latency? How much time does a typical join take for example?
- Cost
    - To help us understand: what is the TECHNOLOGY STACK should we use?
    - Should the design minimize the cost of development?
        * if yes, can just use open source frameworks
    - Should the design minimize the cost of maintenance?
        * if yes, can just use public cloud services

The goal here is to get closer to defining **functional and non-functional requirements**

1.1 What should our system do?
Functional behaviour

- this is what our system will do -- set of operations the system supports
- APIs, interfaces, etc

Q: How to figure out functional behaviour?
Write a statement describing the function we want to see.
then work out what the APIs will be.

Writes/Processing service

1. The system has to count video view events.
→ countViewEvent(videoId) → generalize to all events
→ countEvent(videoId, eventType={view, like, share}) → generalize to all functions
→ processEvent(videoId, eventType, function={count, sum, average}) → generalize to list of events, not just videos
→ processEvents(listOfEvents)

Similarly,

Reads/Query service
2. The system has to return video views count for a time period.
→ getViewsCount(videoId, startTime, endTime)
→ getCount(videoId, eventType, startTime, endTime)
→ getStats(videoId, eventType, function, startTime, endTime)

Use this technique to

- identify names of APIs
- input params
- make iterations to generalize APIs

1.2 How should our system do it?

Non-functional behaviour

- system qualities -- how it is supposed to be -- fast, fault tolerant, etc

Q: How to figure out non-functional behaviour?

- usually interviewer wont say this explicitly -- you need to disover the tradeoffs
- 'let's make it handle huge scale and make it as fast as possible'
- Scalability, Performance, Availability
  - Reliable? Maintainable?

Write down the attributes you want to see.

- Scalable: 10000s video views per second
- Highly performant: Few 10s of ms to return total views count for a video
- Highly available: survives hardware/network failures, no single point of failure

Worth also mentioning

- Consistency
  - though by CAP you implictly prefered Availability over COnsistency here -- you prefer to show stale data instead of no data at all
- Cost? maintenance?

Phase 2: High level architecture

- how does data get in the system?
- how does it get stored?
- how does it get out of the system?

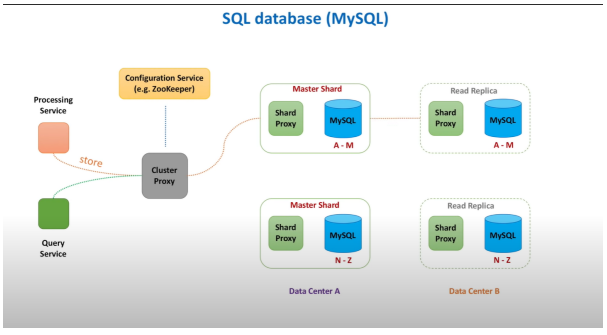
2.1 What data do we store? Schema!!!

Options

- maintain individual events db and calculate aggregates every time
  - this is not a bad option if there can be a lag between write and read of aggregate, order of hours -- batch processing
  - can throw out data after days/weeks
- maintain aggregate data
  - this is an option if we need lower latency -- stream processing
- do both!
  - most flexible
  - drawback is the cost and complexity of the system

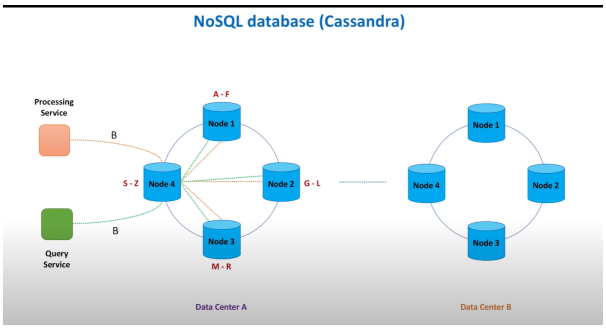
This example follows the aggregate data (2) approach.

2.2 Where do we store the data? I.e. what db/storage mechanism?



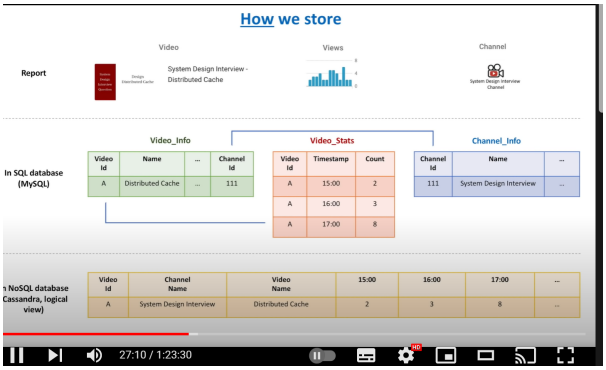
- Processing service for writes, Query service for reads

- Cluster proxy and configutation service: to figure out which node a read or write request must be routed to
- Leader/follower -- either synchronous or asynchronous replication
  - usually async since sync is slow -- prefer to offer eventual consistency



- USes read and write quorums + gossip protocol to sync data
  - Dynamo like?
- Cassandra is a wide column (column oriented) db that supports asynchronous masterless replication

2.3 How do we store the data? What is the data model?



2.4 How do we process the data? What is the processing service?

the processing service needs to

- scale -⟶ partition
- achieve high throughput -⟶ in memory?
- not lose data -- available -⟶ replication
- what do do when db is unavailable?

2.4.1 Should you pre aggredate data in the processing service?

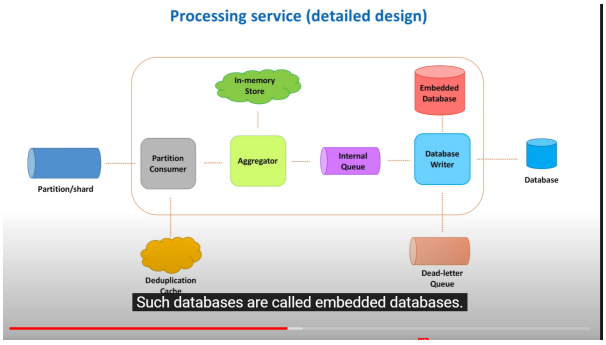
- Remember that this example is using a db that stores aggregated counts

- either you increment a counter in the db every time there is a new client view -- i.e. one write req per client view
- OR the processing service batches view count increments in memory, then sends write one request to the db -- i.e. one write req per n seconds/ms
- For now, we will use the second option (2)
    - it is likely better for large-scale systems

2.4.2 Push or pull?

- Should clients push increments to the processing service?
    - downside: if this is an in-memory store, we will lose data if the machine fails
- Or should the processing service *pull* new updates from some storage/stream?
- Pull better

Processing service



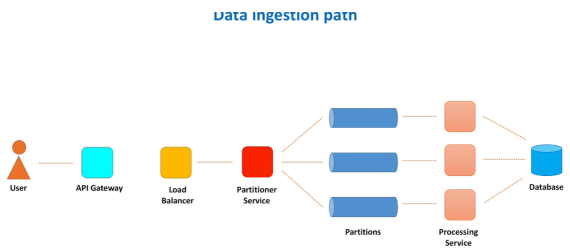
Decouple consumption and processing!

- ideally you want to allow multi-threaded or multi-core processing if possible, even if consumption is single-threaded
    - with events in a stream, single-threaded consumption is helpful since it can preserve orderingy

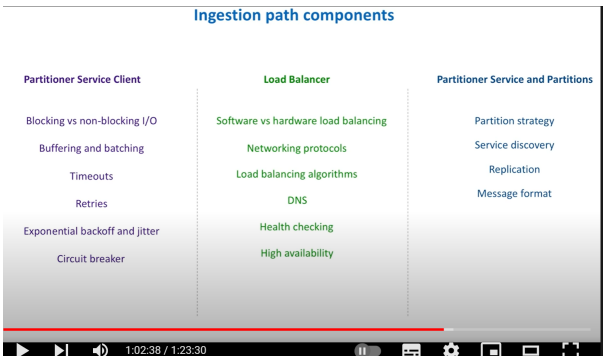
Phase 3: Details

3. Data Ingestion path

- how does the write data get to where it needs to be?



3.1 Ingestion path components



Partitioner Service client (sits on API gateway)

- Blocking vs non/blocking IO
  - blocking systems: one thread per request
    * when a client makes a request to a server:
      - server processes the request and sends back a response
    * the client initiates the connection by using sockets
    * when the client initiates the connection, the socket that handles the request on the server side is blocked
      - A socket uniquely identifies the endpoint of a communication link between two application ports.
      - A port represents an application process on a TCP/IP host, but the port number itself does not indicate the protocol being used: TCP, UDP, or IP. The application process might use the same port number for TCP or UDP protocols. To uniquely identify the destination of an IP packet arriving over the network, you have to extend the port principle with information about the protocol used and the IP address of the network interface; this information is called a socket. A socket has three parts: protocol, local-address, local-port.
      - https://www.ibm.com/docs/en/zos/2.2.0?topic=concepts-

understanding-sockets
        * this happens within a single execution thread -- so the thread that handles the connection is blocked as well
        * when another clientt sends another request, another thread needs to be crated to process that request
        * modern machines can handle hundreds of concurrent connections -- but you can hit trouble when there are many threads already open serving active connections
        * this is why you need rate limiting
        * pros
            · easy to debug -- just look at thread stack for a given request
            · can use thread-local variables
    – non-blocking
        * use a single thread server side to handle multiple concurrent connections
        * server just queues requests
        * actual I/O happens later
        * cons
            · complex: hard to debug
- Buffering and batching
    – if we want to use a blocking system, the API gateway cluster will need to comprise of many machines
        * thousands of youtube view events happening every second
    – but if we pass each individual event to partitioner serivce, that needs to big too
    – so let's batch requests
        * combine events together and send together to partitioner service
        * cons: increased complexity, esp when handling failure
            · if some events in the batch fail but some suceed - should you retry the whole batch or just the failed evnts?
            · presumably just the failed events...?
- Timeouts
    – connection timeout
        * time a client will wait for a connection to establish
        * tens of ms
    – request timeout
        * request processing -- beyond this client is not willing to wait
        * what should you do it you hit a timout?
- Retry
    – but need to be smart about retries -- dont want ot overload the server with retry requests and create a retry storm event
- Exponential backoff for retries + jitter to reduce number of retries
- Circuit breaker

    - if a large number of requests fail, circuit breaker stops passing them through
    - eventually lets a small amount through
    - can make system harder to debug but useful nonetheless

Load Balancer

- software vs hardware load balancing
    - hardware:
        * poweful machines that have many vcpu cores and memory
        * optimied for millions of requests per second
    - software
        * dont need fancy machines
        * ELB from AWS
- networking protocol
    - TCP
        * simply forward packets without inspecting the contents
    - HTTP
        * looks at the content of a packet to make the load balancing decision
        * eg: look at a cookie
- load balancing algs
    - hash based
    - round robin
    - least busy
    - lowers request time
- Question: how does the partition service client know about the load balancer? and vice vers?
- DNS
    - we register the partitioner service in DNS with a specific domain name
        * eg partitionerservice.domain.com
    - tell the load balancer ther IP address for each partitioner service client
    - load balancer needs to know which servers on the registered list are healthy and which are unavailable
- High availablity
    - primary and secondary

Partitioner Service

- Partition strategy: how do you partition new requests from clients?
    - The partition service receives requests from clients, looks inside individual video view events, anr routes these events/messages to partitions
    - each partition appends these events to an append only log file
    - but how do you decide which partition to send a request?
        * hash partitioning: hash(key) may not scale well -- results in hot partitions

            - eg if the key is video id but some videos are just really popular
            - can include some variance into the key by for eg including timestamp
        * consistent hashing
        * can a priori identify certain partitions for very popular keys
- to send messages to partitions, the partitioner service needs to know know about every partition
    - service discovery: comes in two patterns in the world of microservices
        * server side
            - as with load balancer -- the clients know about the load balancer, the lb knows about server side servers
            - but the partitioner service itself is a load balancer!
        * client side
            - every server registers itself in a *service registry* eg zookeeper
            - clients query the registry and gets a list of available partitions
- Replication
    - don't lose events when stored in a partition
        * single leader -- my SQL
        * multi leader
        * leaderless -- Cassandra
- Message format
    - binary
        * protobuf
        * avro
        * faster and easier to parse
    - json/xml

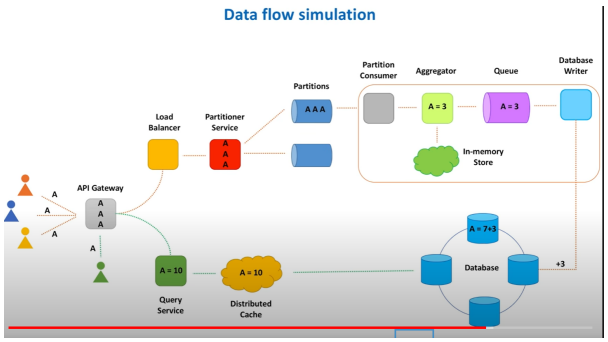
3.2 Data Retrieval path

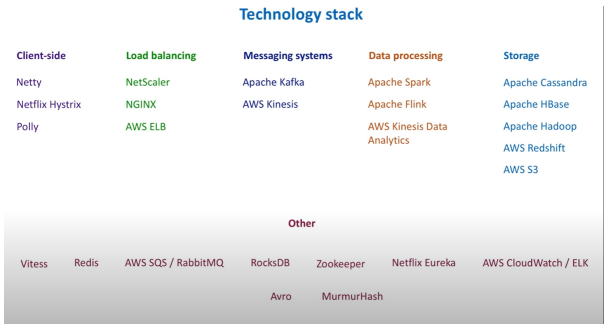
- how does a client read the data it needs to?

Problem: at scale, storing highly granular data is difficult -- too much data

- solution: roll up data after intervals (eg after a week) from per-minute to per-hour, and then even less granular after another interview (eg a month)
- store older data in object storage like s3: cold storage
    - vs hot storage: fast storage
- store query results in a distributed cache

3.3 Final data flow

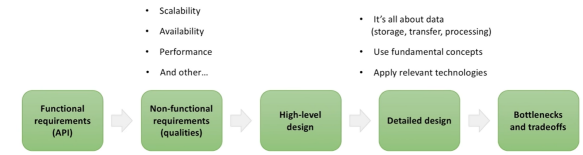




4. Performance testing

- how to id bottlnecks?
  - load testing - measure behaviour under specfic expected load
  - stress testing - beyond expected load to a breaking point - find a breaking point, what component will suffer first and what the componnent is: CPU, memory, etc
  - soak testing - test system with typical production load for an extended period of time -understand that the system is indeed scalable - find leaks in resources eg memory
- metrics
  - latency
  - traffic
  - errors
  - saturation
- how to maintain accuracy?
  - audit systems
  - weak:
    * running end to end test continuously
    * not 100% reliable -- may not test rare scnarios
  - strong
    * build a second system
    * compare results across them both

Summary



- Write down verbs
- Define input parameters and return values
- Make several iterations

- How data gets in
- How data gets out
- Try to drive the conversation

- Listen to the interviewer
- Know the tradeoffs