System Design Notes

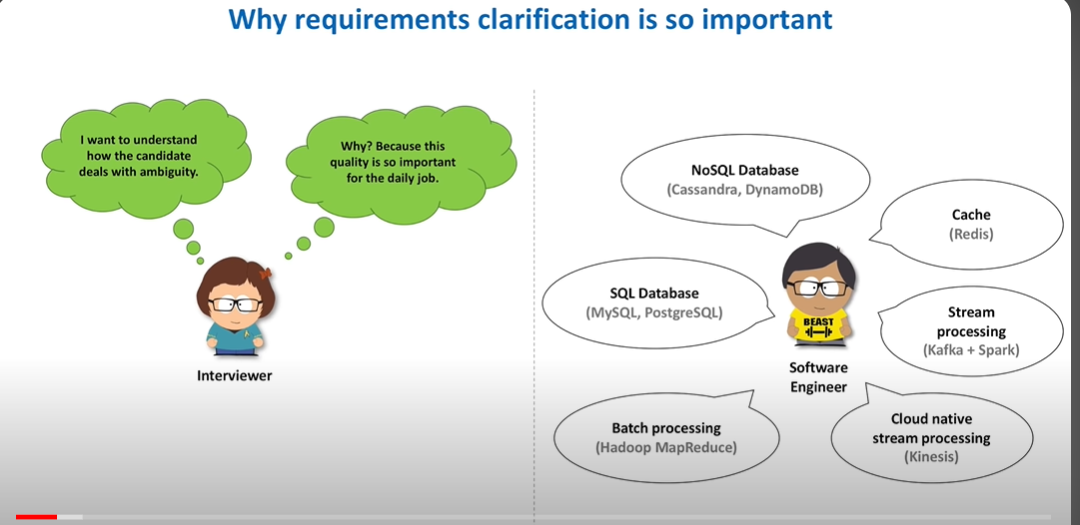
Interviewer will ask a question by giving a problem statement.

Always ask interviewer questions about the given problem to get more information.

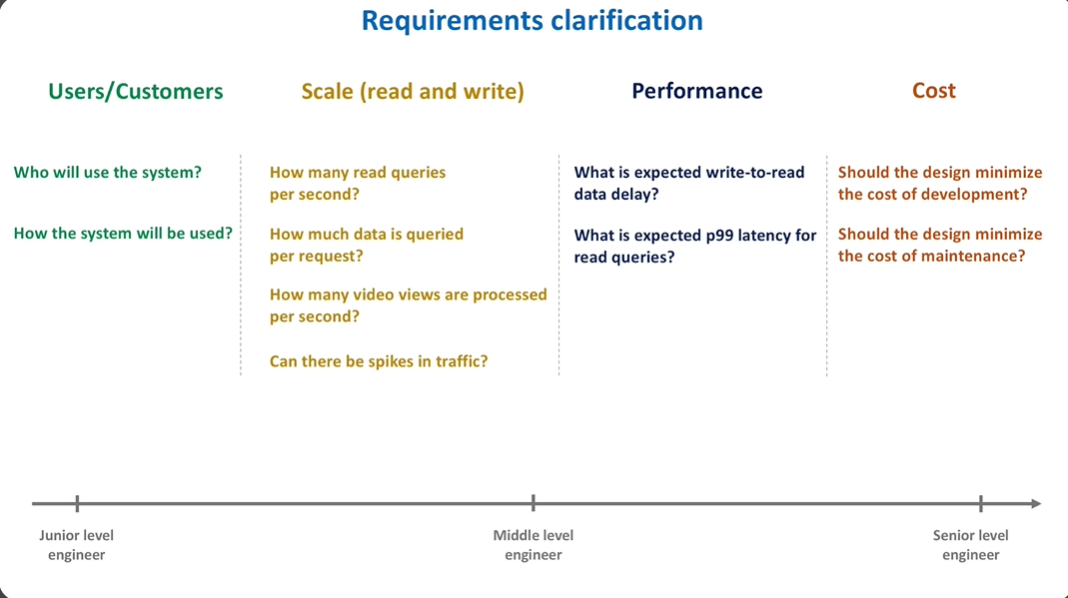
We need to know which pieces of system we must focus on.

**Problem: Count views of a YouTube video:**

Different engineers will solve this problem in different ways:

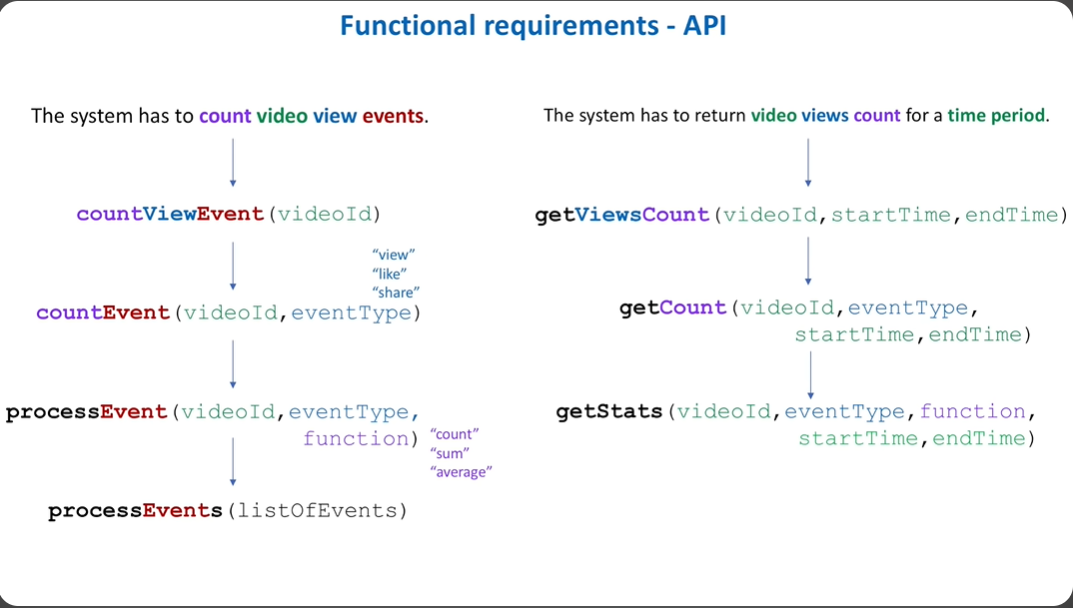


We need requirement clarification for the below things:



Function Requirements: What should the system do? What functions the system is supposed to perform.

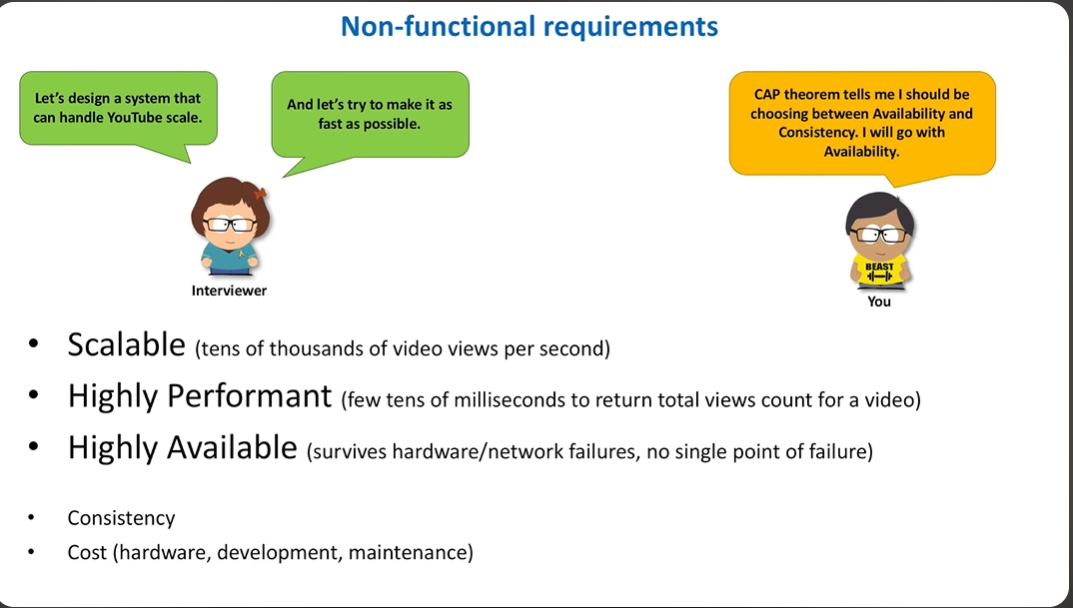
Basically, what APIs we need to have in the system.



Find the how many APIs we need, name of APIs, what will be the input and output of these APIs.

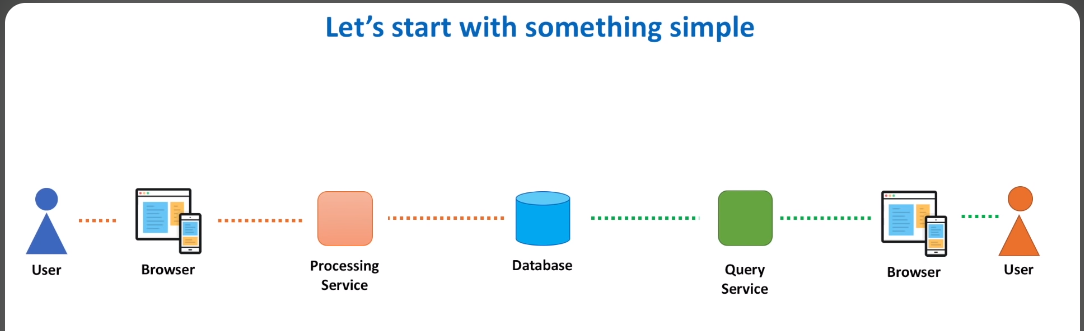
Non-Functional Requirements:

Making the system fast, scalable, reliable etc.



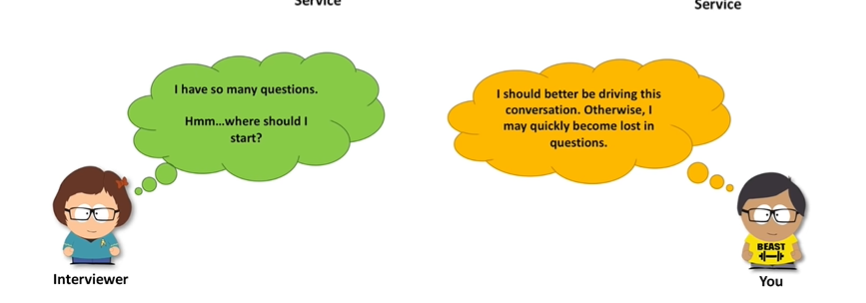
We can now start with designing the system:

Very high-level design:

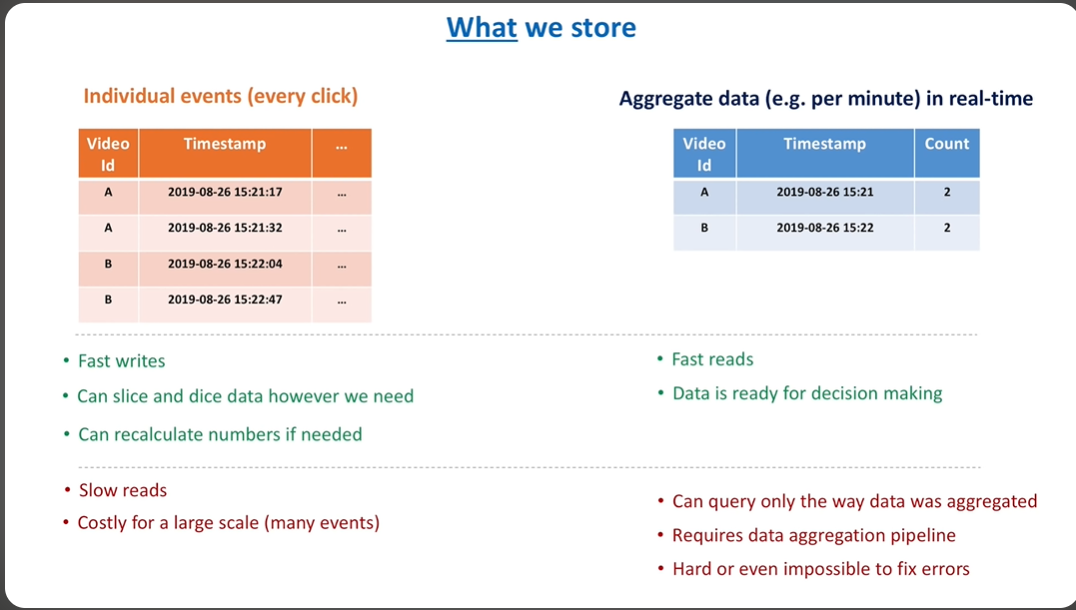


Processing Service: Will process the data from user/browser and store it in the DB.

Query Service: Will query the DB and return the view count to the user.



We can start from the DB:

**What we store:** we need to know what we store. We can store data in many ways. We can store each individual event, or we can store events combined which happened in a period of time, eg 1 min. Both of these approaches have their own goods and bads. 

Ask the interview question to choose which one of the above should be chosen. You can ask questions like what is expected delay, do we need this data immediately for data analysis etc.

For now, we will take both cases. We will store individual events as well as aggregate data. This will give us a lot of flexibility, but it will incur more cost. There is a tradeoff.

**Where we store:** Evaluate your database based on three non-functional requirements:

**Available, Scalability, Performance.**

A screenshot of a questionnaire

Description automatically generated

**SQL DBs:**

Things are simple when we can store all the data in a single machine.

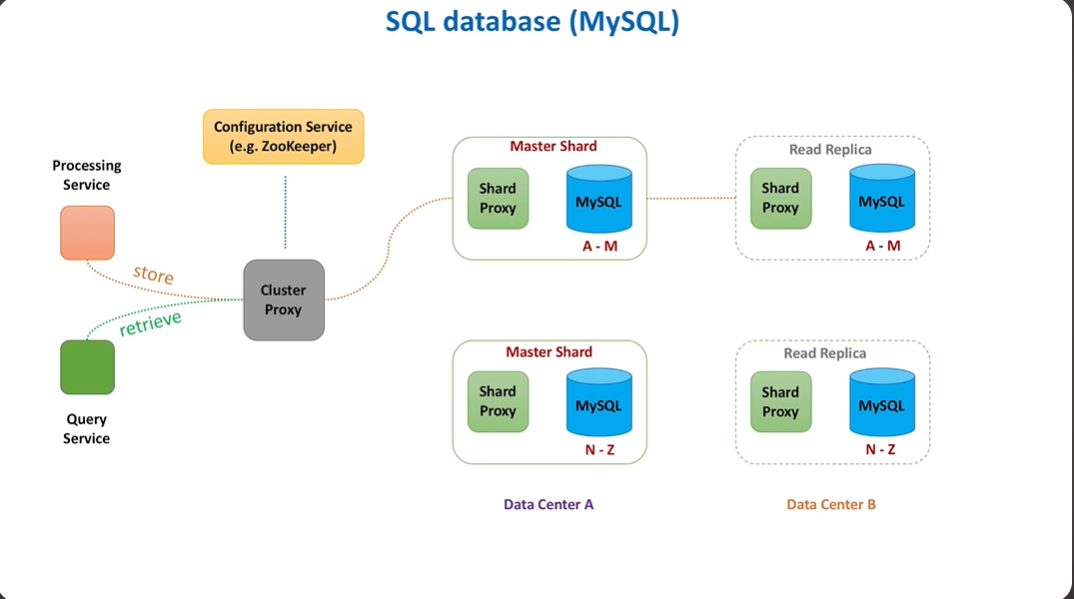
If we can’t store data on a single machine, we need to introduce more machines, this process is called **Sharding or horizontal portioning.** Each shard stores subset of the data. Now that we have multiple machines, services that we use to connect and get data from the DB need to know how many machines are there and which one to connect and get/store specific data.

We already have processing services (to store data in DB) and retrieving services (to get data from the DB), each of these services will have to go to all the machines which is more work. A better solution is to introduce a **proxy server** which knows about all the machines and routes are services to the correct shard or machine. Now, the services don’t need to know about the shards anymore but the proxy need to know about the shards. Proxy needs to know if new shard is added, a shard dies etc. To get this info to the proxy, we introduce a **configuration service**.

Instead of now directly querying the DB from proxy, we introduce a new proxy called **shard proxy.** Proxy will send query to shard proxy which will then run it on the DB. Shard proxy can cache DB results, monitor DB health, publish metrics, terminate queries which take longer than usual etc.

Now we need to make sure **availability** of the system. To do so, we need to replicate data.

Each DB + its shard proxy is called a master shard. Each master shard will have a read replica which will only be used to read data. Data will be written to master shard and then synchronously or asynchronously.



**This is what YT is using**.

Let’s now try NoSQL DB:

**Cassendsra DB:**

Here we don’t need master and replica shards, instead we will have nodes which are equal. We don’t need configuration services to monitor the health of each shard. We will allow nodes to talk to each other and exchange information. This is called **Gossip protocol.**

When client sends a request to a node, the node will decide whether to forward this req to another node or process it.

Initially which node will be selected can be decided based on robin round scheduling or whichever node is closest to the client and it will be called coordinator node.

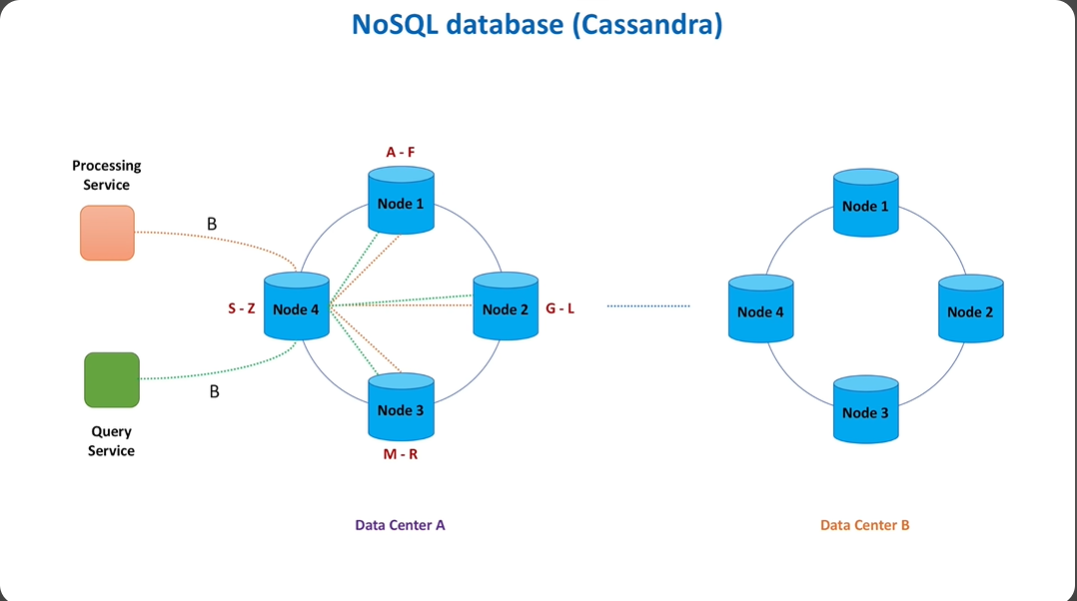
Processing service who wants to write data will call coordinator node which will then call one or more nodes to write data. This is called **quorum writes.**

A write quorum is a group of nodes in a distributed system that must all agree on a write action for it to be valid.

Quorum-based consistency uses a voting mechanism to determine the consistency of data operations. A quorum is the minimum number of votes that a distributed transaction must obtain to perform an operation. This number is usually a majority of the nodes.

For example, a read quorum, r, is such that at least r replicas must be accessed by a read operation. A write quorum, w, is such that at least w replicas must be accessed by a write operation. Given n replicas, r + w > n. We have to do this because some nodes in the system may not have been updated. So, we access many nodes for read and write and then do the operation based on majority.

Quorum consistency is used in systems where consistency is more important than availability.



**Stale Data:** Stale data is data that is out-of-date, obsolete, or no longer accurate. In the context of databases, software applications, and computing systems, stale data often occurs when an update or refresh operation fails, is delayed, or is not performed regularly.

We have two main non-functional requirements, availability or consistency.

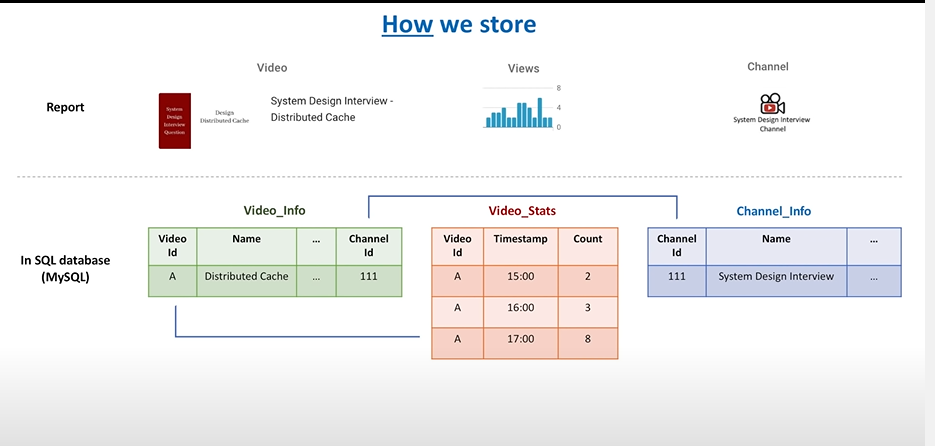
We can choose one over the other based on the kind of system that we are trying to create.

We usually update the data async, as a result some replicas have stale data. Which is why some users may see different views for the same video. This inconsistency is temporary. Eventually all the replicas will have the same data. So, consistency will be achieved later. This is called **Eventual Consistency.**

We have discussed **what we store and where.**

Let’s discuss **How we store data i.e., data modelling.**

To define models, we will find verbs in the system and then we will convert these verbs into models or tables. Let’s see how we will store YT info in relational/SQL DB:

 A screenshot of a video menu

Description automatically generated

In SQL DB, we wil create tables and we will store data in them. We will link tables using foreign keys and wen will retrive data from multiple tables using foreign keys.

Four types of No-SQL DB:

1. Column
2. Document
3. Key/Value
4. Graph

We have used Cassandra because it is:

1. Scalable
2. Fault tolerant
3. Supports multi data center replication
4. Works well with time series data

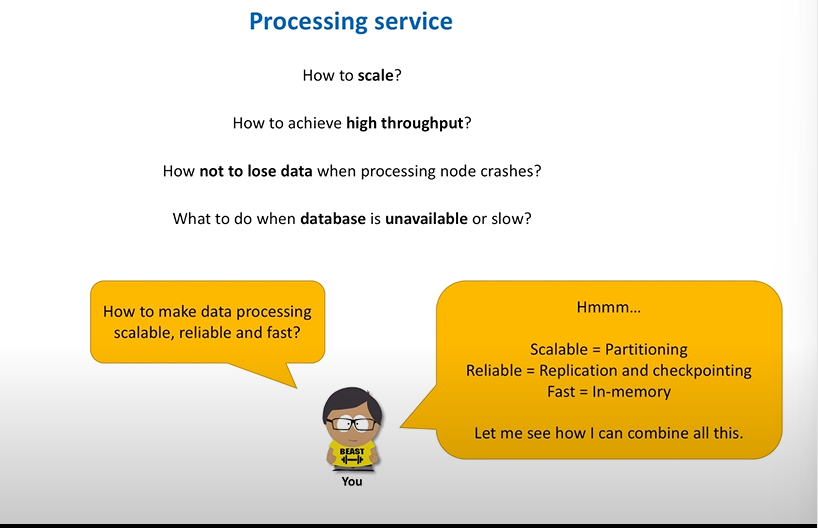
Not all non-relational DBs don’t have the same architecture.

Let’s now talk about **Processing services.**

It involves:

1. When YT user opens the video, we want to display the total view count immediately. We need to calculate total views of the video on the fly.
2. When video owner opens the video, we want to show hours count.

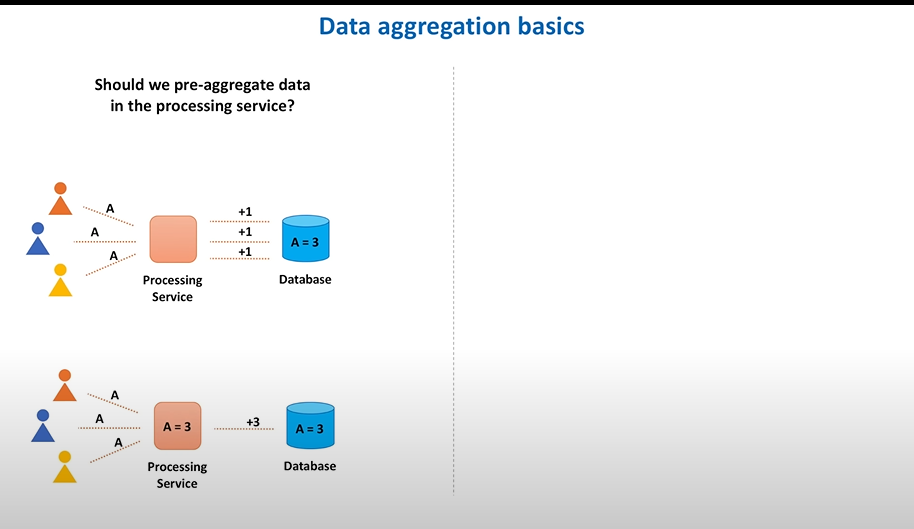
So, we increase many counters whenever we process data. Whenever we get an event, we want to do some processing. Also, when events increase, we need the system to be scalable.



How should we update counter:

For every event or for a bunch of events, this is called data aggregation.

Whenever we receive an event, we increase counter. Or we wait for a period of time and after that period we increase the counters at then same time.



For larger scale systems, waiting for a period of time and then increasing the counter is better.

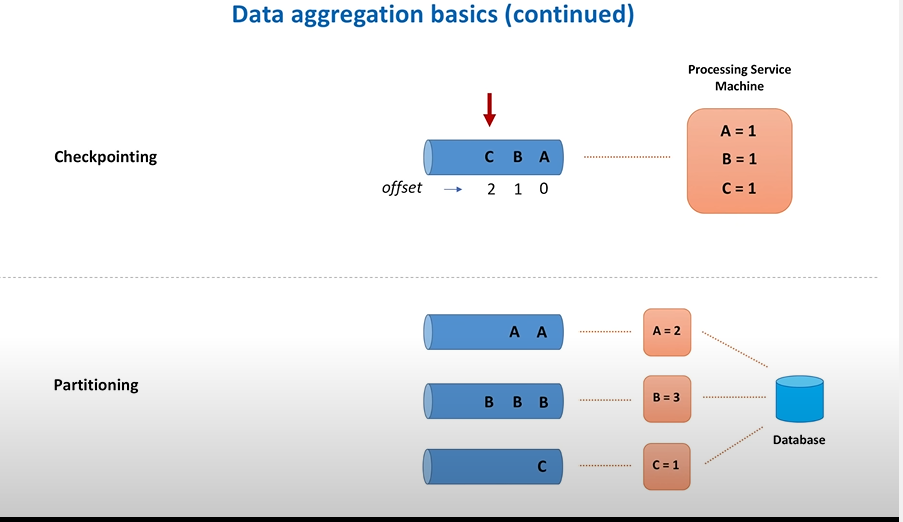
Now, instead of processing the data directly from the user, we first store the data in a temporary called a queue, and then process data from the queue. This makes it **Fault Tolerant.** If while processing the machine fails data is lost, this is not the case if we use processing queues where we have the data in processing queue and we can process that data once the machine is back up. Also, it is easier to scale.

A diagram of data aggregates

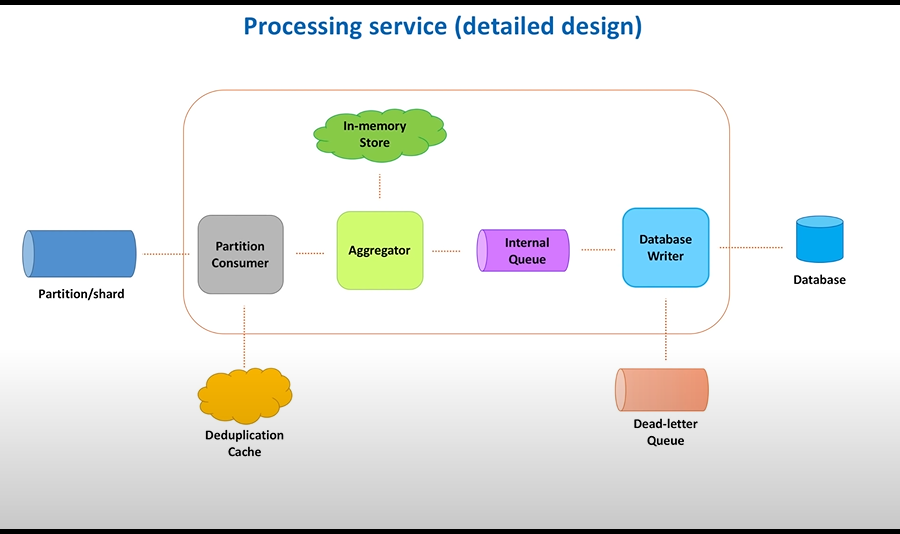
Description automatically generated

**Checkpointing:** Events are processed in FIFO order. We write checkpoints to persistent storage after processing several evets. This is helpful in streamline data processing.

**Partitioning:** Instead of a single queue, we have serval queues now. We put events into the machine by using hashing.



So, processing service will receive events from a processing queue and write/read from the DB.



Partition Consumer: Will consume data from the partition/queue. It will also move duplicate reqs to the duplication cache.

Aggregator: Like a hash table and accumulates data for some time. Data from this is then sent to the internal queue.

Internal Queue: It will help us to process our events on multiple threads instead of just one thread. This will make processing faster.

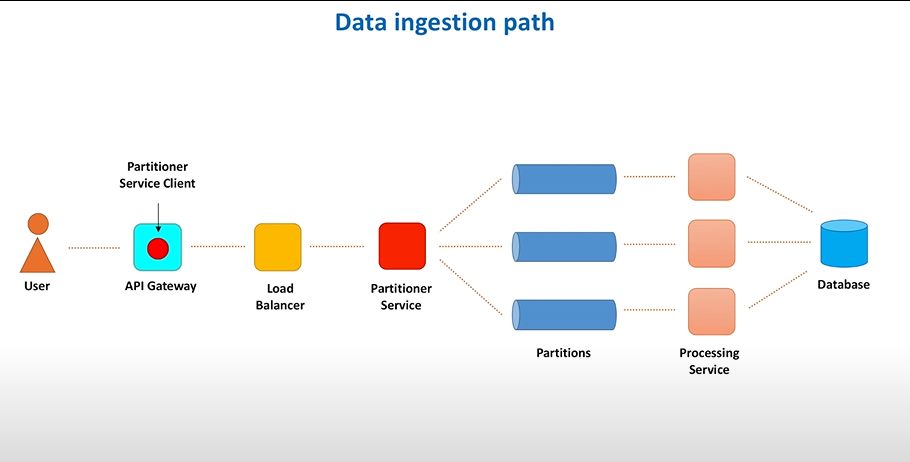
Database Writer: Will write the aggregated data from the aggregator to the database.

Dead-letter Queue: Messages are sent here, if events can’t be routed to their correct destination. This is done for performance and availability issues. There is a separate process that reads messages from this queue and then sends them to the DB.

**Data Ingestion Path:** We have multiple partitions/queues which has the events, and we process data from these queues through processing services. We need a component which will distribute data across partitions, let’s call it **partitioner service**.

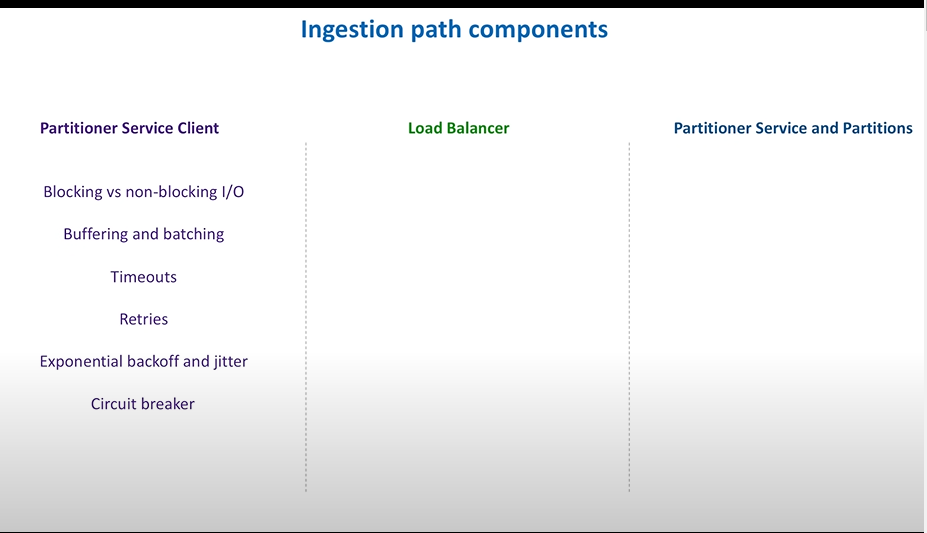
Let’s also put a **load balancer** in front of the partitioner service to evenly distribute events to partioner servicer.

User will create a request and that request will go through an **API Gateway**. API Gateway will route the user requests to backend services.



There are three components in **Ingestion Path Component:**

1. **Partitioner Service Client**
2. **Load Balancer**
3. **Partitioner Service**

****

**Partitioner Service Client** can have **Blocking or non-blocking I/O.**

**Blocking I/O:** When client sends a request to a server. Server will process the request and send back a response. Client initiates the connection using a socket. When client makes a request, the socket that handles that request connection is blocked. This happens in case of single execution thread. So, the thread that handles that request is blocked as well. When other clients send the same req, we have to create another thread to process that req. This is blocking IO works. Create one thread per request/connection.

If we get hundreds of threads at the same time, server slows down as no of requests and threads increases.

**Non-Blocking I/O:** Use single thread on the server side to process multiple concurrent connections. Server will just queue the req and the actual IO is processed at some later point of time. These systems have better throughput and are more efficient but have higher complexity.

Blocking systems are easier to debug/fix. We can easily track the progress of a thread since there is just one.

**Buffering and batching:** Instead of continuously sending the request we will create batches of request and then send them together. We wait for several seconds for the buffer or batch to fill up and then send it to the partitioner service.

**Timeouts:** How much time client is willing to wait for the response from the server. We expect one percent of the total requests to timeout. The timeout time is chosen by analyzing the latency.

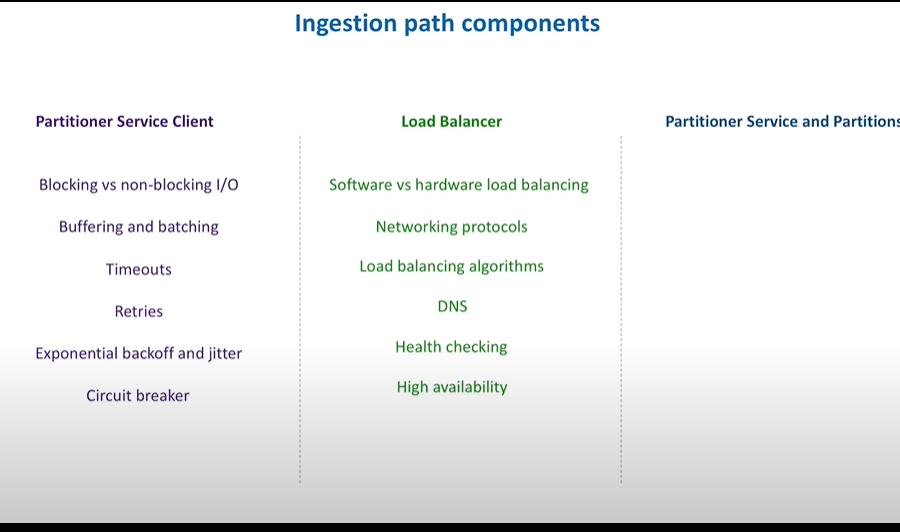
What to do with the timed out reqs: Lets retry them. But we have to make sure we don’t retry too many times otherwise we may create too many retries and block the queues.

To avoid this, we will use **Exponential Backoff and Jitter Algorithms** to set timeout times. We retry request many times but wait longer between each retry. Jitter will add some randomness to the time.

**Circuit Breaker:** What happens in case of partitioner services failure. We will get too many retries because the service has failed. We will calculate how many reqs have failed in recent times. If error threshold is exceeded, we stop calling the downstream service. Sometime later we will only send few reqs to the downstream services. If these reqs are successful, we resume the operations.

A drawback of circuit breaker is that it makes hard to test the system.

Everything in distrusted systems has tradeoffs.



**Load Balancer:** They distribute data traffic between multiple servers. There are two types of load balancers: **hardware, software. Hardware Load Balancers** are network devices we buy from known organizations. These are powerful machines having many CPU cores, memory and optimized to handle very high throughput. **Software Load Balancers** are just software we need to route traffic. Many of them are open source. AWS Elastic Load Balancing (ELB) is a software load balancer.

Another way of categorizing then is based on **Networking Protocol.** They are TCP and HTTP load balancers.

**TCP load balancers** route the TCP packets without inspecting the content of the packets.

**HTTP load balancers** will look into the message and then make a routing decision. E.g., based on cookie information or header info.

Load balancers will also use **Load balancing algorithms** to make a decision. **Round Robin Algorithm** wil distribute requests in order across the list of servers. **Least connection algorithm** will send the request to the server which has the least connections at the moment. **Least response time** algorithm sends request to the server with fastest response time. **Hash based algorithms** will send the request based on a key we define which can be the client IP address or the client request URL.

In our system design problem, we need to answer many questions.

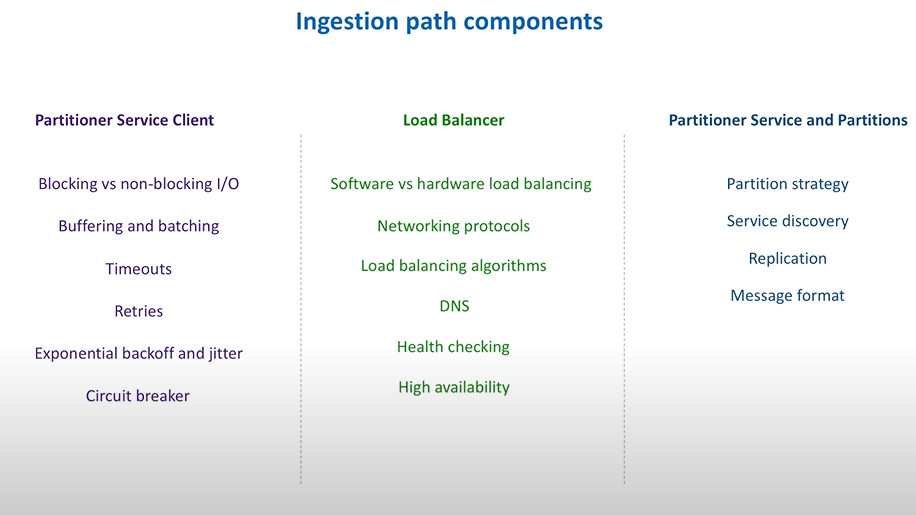
How partitioner service client knows about knows about the load balancers?

How load balancer knows about partitioner service machines?

How does load balancers guarantee high availability?

**DNS:** Maintains domain names next to their respective IP addresses. We register our partitioner service in DNS. We specify the domain name and associate it with the IP address of the load balancer device. When clients hit domain name request goes to load balancer. We will also explicitly tell the IP addresses of the partitioner services to the load balancer. Both hardware and software load balancers provide APIs to register the servers. Load balancers need to know which servers are available and which are down. Load balancer will ping each server periodically to check the health of each server.

To provide **high availability**, load balancers use a concept called primary and secondary nodes. **Primary load balancer** accepts connection and serves request. **Secondary load balancer** will check the health of the primary load balancer. If primary goes down, secondary takes over. They also live in different data centers in case if one data center goes down.

****

**Partitioner Service: Portioner Service** is a web service that gets request from client, looks inside each request and then retrieves individual view counts. It routes each event to some partition.

**Partitions:** It is also a web service. It gets messages and it stores them on disk in the form of the append only log file.

But which partition gets which message? This is answered through **Partition Strategy**. A simple strategy is to calculate a hash function based on some key, which can be a video identifier. Based on this we will decide a machine. This strategy does not work on large scale because it might create **hot partitions.** If we have a very popular video, then the events related to them might go to the same machine which causes some machines to get too many requests.

Partitioner Service wants to send messages to partitions, in order to do that partitioner service needs to know about every partition. This is where we use concept of **Service Discovery.**

In Microservices, we have two service discovery patterns: **Server-Side Discovery and Client-Side Discovery.** We saw server-side discovery, when we looked at load balancers. Client knows about load balancers; load balancers know about server-side instances.

Partitioner service works like a load balancer because it gets messages and then it distributes them between partitions. This is client-side discovery pattern. Here, every server instance will register themselves in a common place, named **service registry.** It is also a very highly available web service which can perform health checks on each registered server. Clients query service registry and obtain a list of available servers. Eg. **Zookeeper.** Partitions register themselves with Zookeeper and partitioner services query them to get the partitions.

We don’t want to lose events when they are stored in partitions. So, we perform **replication.** Three types: **Single Leader Replication, Multi Leader Replication, Leaderless replication.**

**Single leader replication** was used while replicating SQL DB. **Leaderless replication** was used during the Cassandra DB. **Multi leader replication** is used when there are multiple data centers.

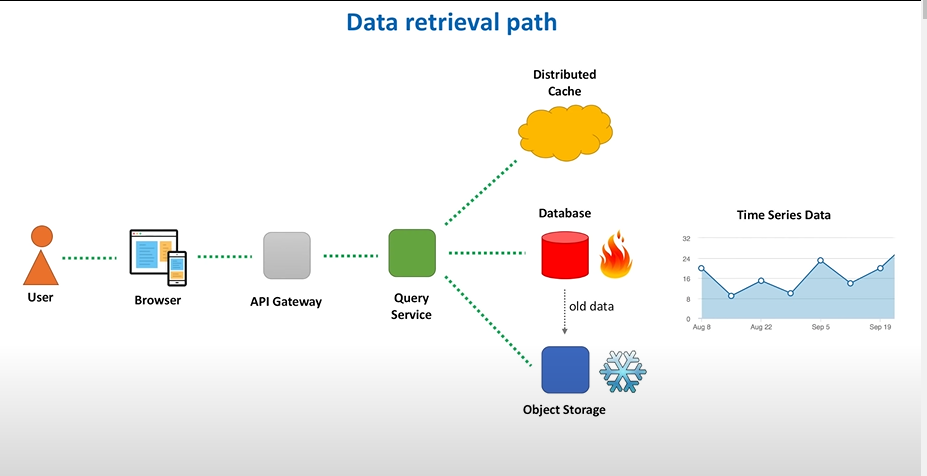
**Single leader replication:** Each partition has a leader and several followers. We read and write events in the leader only. When leader is alive, all followers copy events from the leader. If leader dies, we choose a new leader from the followers. Leader keeps track of its followers; whether followers are alive, or any of the followers are too far behind. If a follower dies, gets stuck or is too far behind, leader removes it from its followers.

When we have to write to the leader, we write it to many replicas and once we have many copies of the data on many replicas then we send the response back that write was successful. This improves **durability** but **latency** increases. Also, if required number of replicas are not available at the moment, availability will suffer. This is the tradeoff.

**Message Formats:** We have textual formats such as **XML, CSV, JSON** etc. We also have binary formats such as **thrift, protocol buffers, avro** etc. For large-scale real-time processing systems binary formats provide much more benefits. These messages are more compact and faster to parse.

Messages will contain several attributes such as video identifiers, timestamps and user related information. In JSON, we will use many fields for these which will increase the total size.

**Binary formats** are smarter in this respect. **Apache Thrift and Protocol Buffers** use field tags instead of field names. Tags are just numbers, and they act as aliases for the fields. Tags occupy less space when encoded. Schemas are crucial for binary formats. Client needs to know the schema to serialize the data. Processing services need to know the schema to de-serialize the data. **Apache Avro** is a good choice for our case.

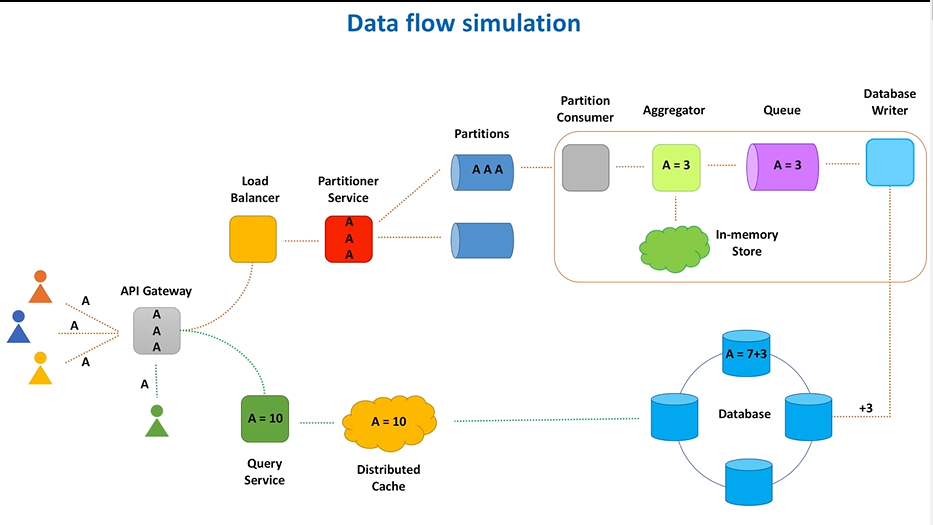


**Data Retrieval Path:** When a user clicks the video, we need to show total view counts for the video. To build a video web page we call several services. A web service that retrieves information about the video, a web service that retrieves information about the comments, and another one for recommendation. Among them there is our query web service which is responsible to video stats.

All these web services are typically hidden behind an **API Gateway Service,** a single-entry point. API gateway routes client requests to backend services.

When we need time series data i.e., stats for specific time i.e., a minute, an hour etc. If we store this data directly, we need huge storage. To avoid this, we use rollup. We store per minute data for few days. After a week, per minute data is aggregated into per hour data, after a month we store per hour data and so on. We store old data, let’s say older than three months, is stored with 1 day granularity, ie data aggregated for one day, store in **Object Storage (AWS S3)**.

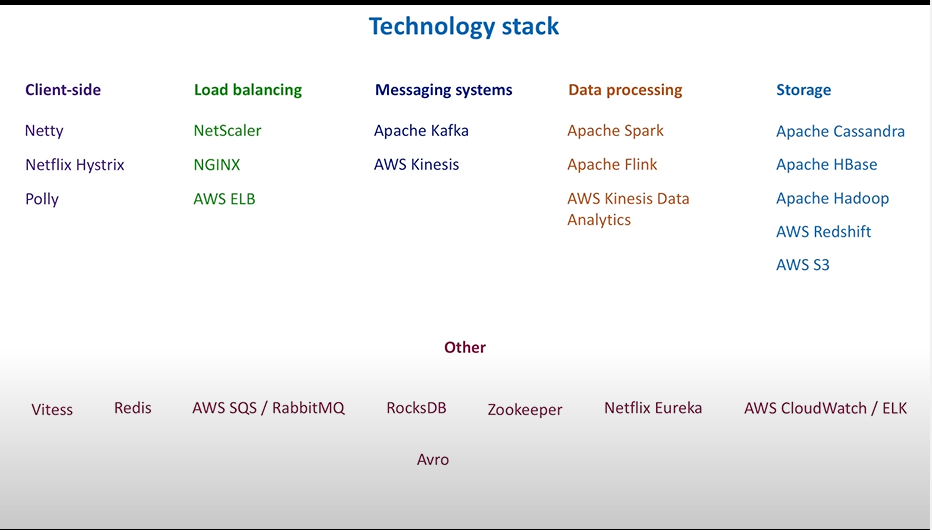
**Hot storage** refers to frequently used data that must be accessed fast. **Cold storage** has archive data, not needed to be accessed fast. Query service will retrieve recent stats from hot storage and older stats from cold storage.

****

Above is the final flow of the data. To **write the data**, let’s say we get three views/three events, first reaches the API gateway which sends all of the three events to the load balancer. Which picks the partitioner service, and which then sends it to the correct partition. All three events end up in the same partition, as we partition data based on the video identifier.

Now, in the partition services, first the partition consumer consumes all the three events one by one and sends them to the aggregator. Aggregator counts messages for a one-minute period and flushes calculated values to the internal queue at the end of that minute. DB write picks count from the internal queue and sends it to the database. In the DB, we store count per hour and the total number of views for each video. Total count was seven prior to this, we add 3 for the current minute.

During **reading the data** when user opens video A, API gateway sends request to the query service. Which checks the cache, if data is not in cache, we call the DB and get the count.



Another important aspect of system design is the **technology stack.** Which tech stack should be used for our system?

**Netty** is a high-performance non-blocking IO framework for developing network applications.

**Hystrix from Netflix and Polly** simplify the implementation of many client-side concepts such as timeouts, retries, circuit breaker pattern.

**Citrix NetScaler** is a famous hardware load balancer. **NGINX** is a popular software load balancer. In cloud, **Elastic Load Balancer** is a good pick.

For **partitioner service and partition,** we can use **Apache Kafka** or its cloud counterpart **AWS Kinesis.**

To process events and aggregate them in memory, we can use stream-processing frameworks such as **Apache Spark or Flink.** We can also use cloud-based solutions such as **Kinesis Data Analytics.**

For storage, we can use **Apache Cassandra or Apache HBase.** We can store raw events in **Apache Hadoop or AWS Redshift (cloud-based warehouse)**. When we roll up the data and need to archive it, AWS S3 is a natural choice.

Other that these techs, we also use **Vitess,** which is used for scaling and managing large clusters of MySQL instances. Vitess has been serving all YT DB traffic since 2011.

We also rely on distributed cache in several places, for message deduplication and to scale read data queries, **Redis** is a good choice.

For dead-letter queue mechanism, when we need to temporarily queue undelivered messages, we can use an open-source message-broker such as **RabbitMQ** or public cloud alternative such as **Amazon SQS.**

For data enrichment, when we store video and channel related info locally on the machine and inject this info in real-time, we may use **RocksDB,** a high-performance embedded DB for key value data.

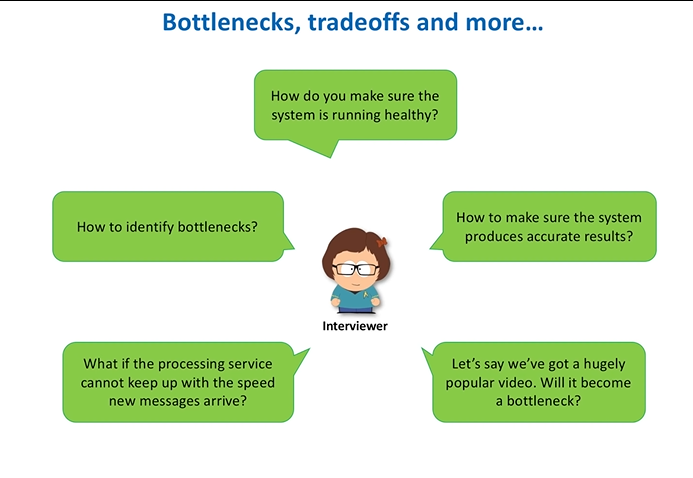
To do leader election for partitions and to manage service discovery, we may rely on **Apache Zookeeper.** We also have an alternative called **Eureka Web Service** from Netflix.

To monitor each of our system design components, we may rely on monitoring solutions provided by Amazon such as **AWS CloudWatch** or popular stack of open-source frameworks like **Elasticsearch Logstash Kibana, ELK for short.**

Binary message format is preferred for our system, we can use **Thrift, Protobuf and Avro.**

For partitioner service to partition the data, we need to use a good hashing function. We can use **MurmurHash.**

We should know about the tradeoffs about each component.



**Bottlenecks:** To need to test the system under heavy loads to identify bottlenecks. This can be done by doing **performance testing.**

**Load Testing:** Checking the behavior of the system under specific expected load.

**Stress Testing:** Test beyond normal operational capacity, often to a breaking point. We check which components start to suffer first. Memory, CPU, Network Disk IO.

**Soak Testing:** Test system with a typical production load for an extended period of time. Check is system is scalable. We try to find leaks in resources e.g., memory leaks.

Tools like **JMeter** can be used to generate desired loads.

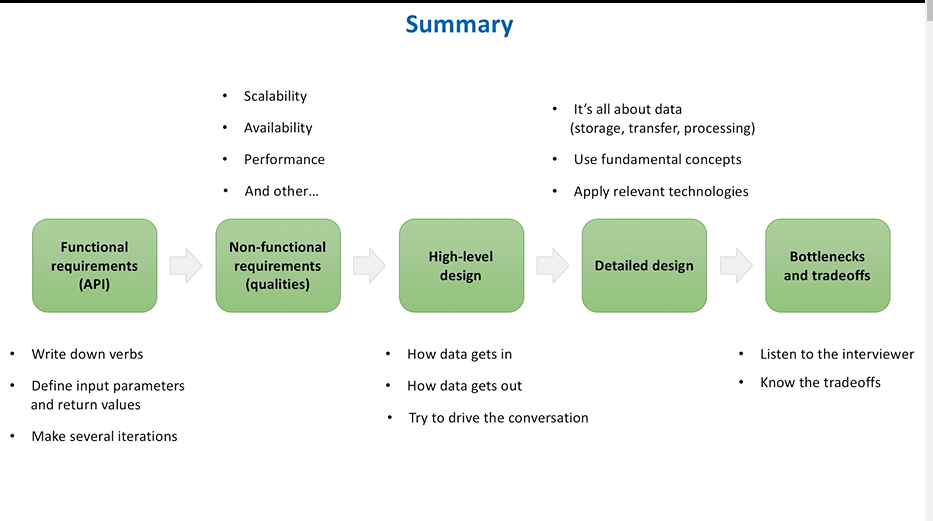
**Health Monitoring:** We need metrics (error count or processing time etc.), dashboards, alerts to check health of each component. Four golden signals of monitoring are **Latency, Traffic, Errors and Saturation.**

**We also need to know the accuracy of the system.**

We need to not just count the video views accurately but also ad views, because we want to charge ad owners and pay video owners. This can be handled by building an **Audit System.** They can be weak or strong.

For popular videos, we need to spread their events evenly across all partitions. So, that a partition is not overburdened due to too many reads and writes.

**Summary:**

****