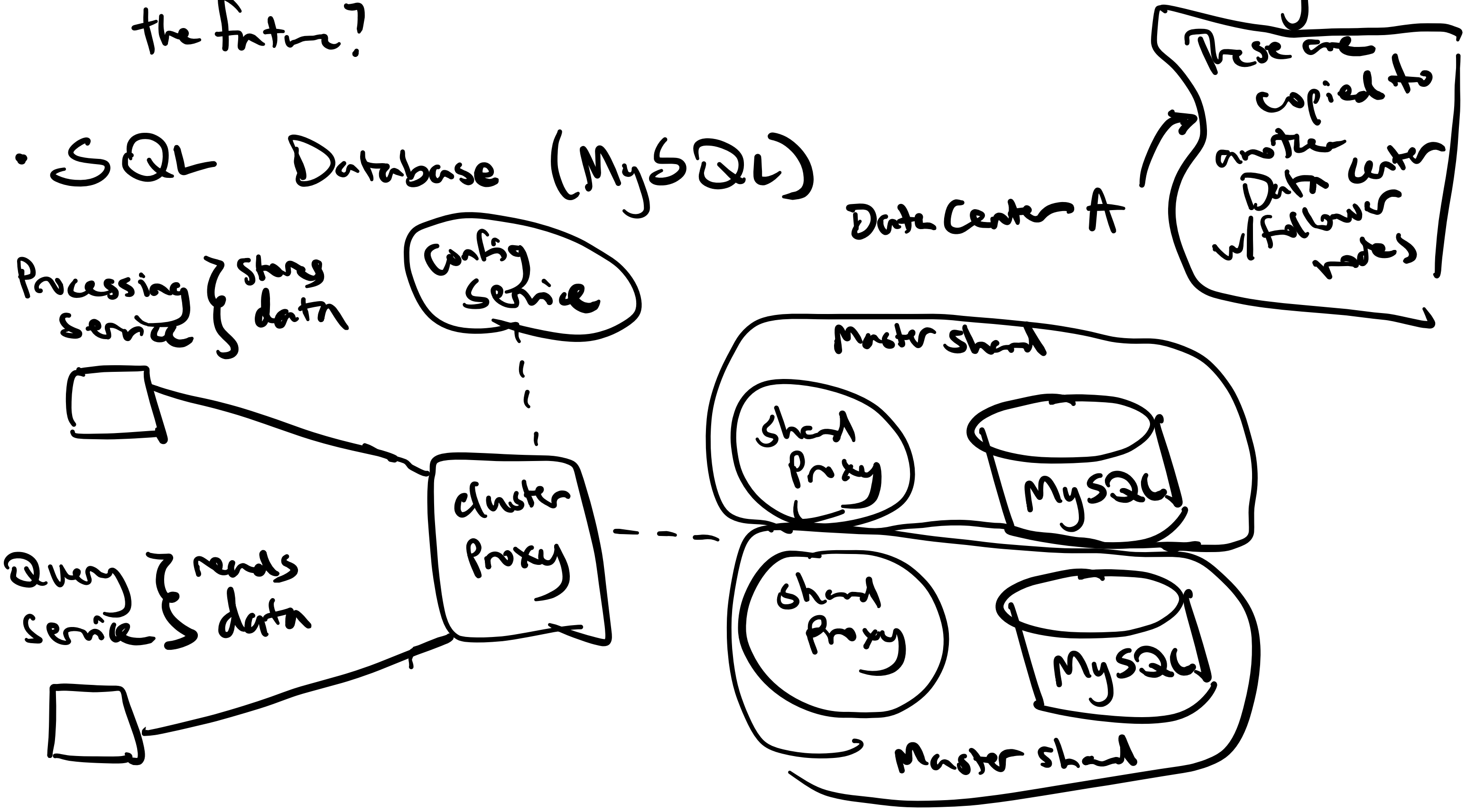
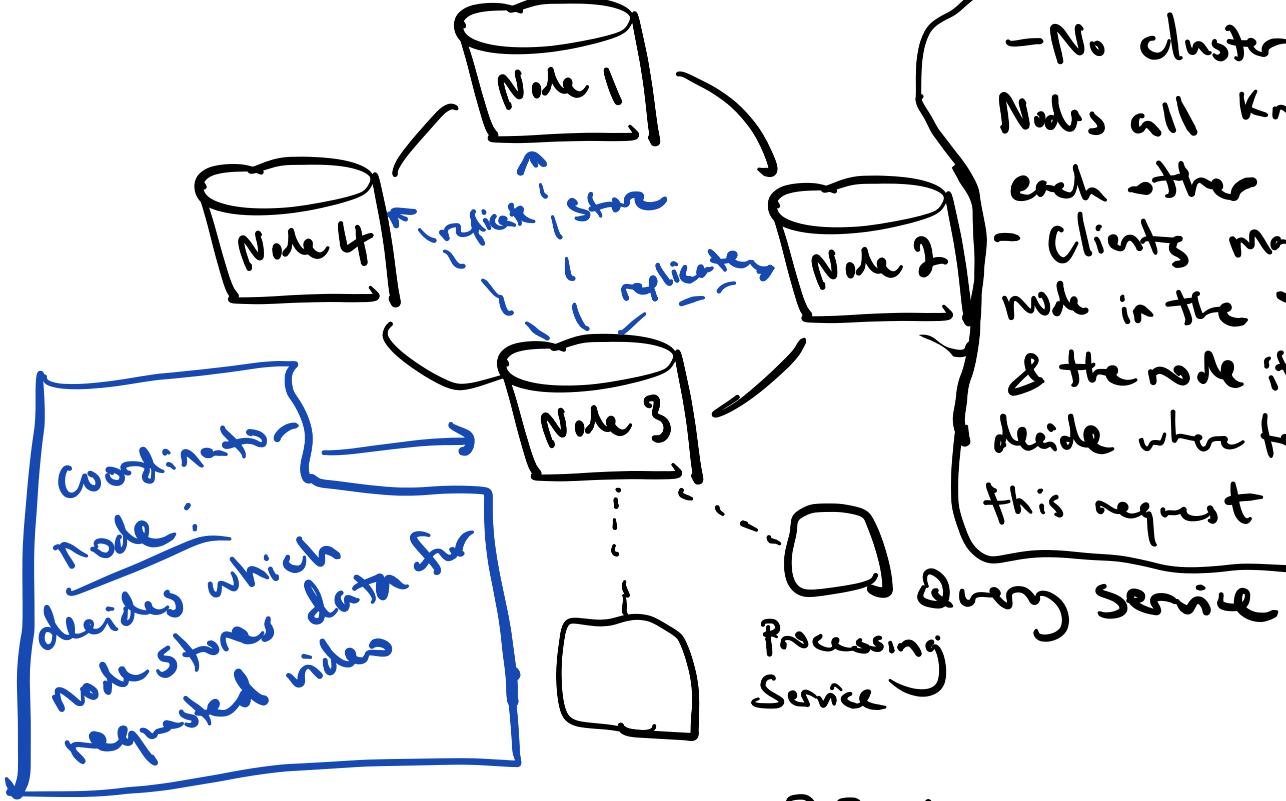
System Design Interview Guide (Part II)

- Let's think about where we store data.
We're using the same example as Part I: "Count YT vid views"
Evaluate both SQL & NoSQL databases, since both can scale & perform well.
  - ↳ How to scale writes?
  - ↳ How to scale reads?
  - ↳ How to make both reads (r) & writes (w) fast?
  - ↳ How to not lose data in case of hardware faults & network partitions?
  - ↳ How to achieve strong consistency? What are the tradeoffs?
  - ↳ How to recover data in case of an outage?
  - ↳ How to ensure data security?
  - ↳ How to make it extensible for data model changes in the future?

- SQL Database (MySQL)


These are copied to another Data center w/ follower nodes
Data Center A
Processing service } stores data
Config service
Master shard
Shard Proxy
MySQL
Cluster Proxy
Query service } reads data
Shard Proxy
MySQL
Master shard

- NoSQL Database (Cassandra)



coordinator node: decides which node stores data for requested video

- No cluster Proxy: Nodes all know about each other
- Clients may call any node in the cluster & the node itself will decide where to forward this request further

- we can use a simple RR algo to choose the initial node, or we may be smarter & choose a "closest to client" node.
- Node 3 writes to Node 1. Can also copy to other nodes simultaneously.
  ↳ waiting for responses may be slow, so use a quorum: coordinating node returns successful if a certain number of nodes return successfully

use consistent hashing to decide which node handles the operation.

- Similar concept for reads: Some nodes may have stale data, a read quorum defines a minimum # of nodes that have to agree upon the response
- Cassandra uses version num to determine staleness of data

Similar to SQL DBs, we copy data to other data centers for high availability. So this setup could be one of many.

- In Part I, we chose availability over consistency: Stale data better than no data.
  - If we replicate data synchronously it will be slow but consistent. We usually replicate it asynchronously.

Example: Leader-Follower Replication

↳ some read replicas may be behind their master

↳ some users will see different view counts

- eventually the nodes they're reading from will have correct info: eventual consistency

- <u>How</u> do we store Data?
  - Define Norms

↳ Build a report that shows: info about the video, # of total views per hour for last several hours & info about the channel this video belongs to

↳ can store all of the data in either relational or non relational DBs

- While we don't need to know the details of how every DB works, it's important to know the tradeoffs

↳ example: Cassandra is a wide-column DB that supports asynchronous masterless replication.

↳ example: MongoDB is a document-oriented DB & uses leader-based replication

↳ example: HBase is column based like Cassandra but supports leader-based replication

- Processing Service

↳ How to scale?

↳ How to achieve high throughput?

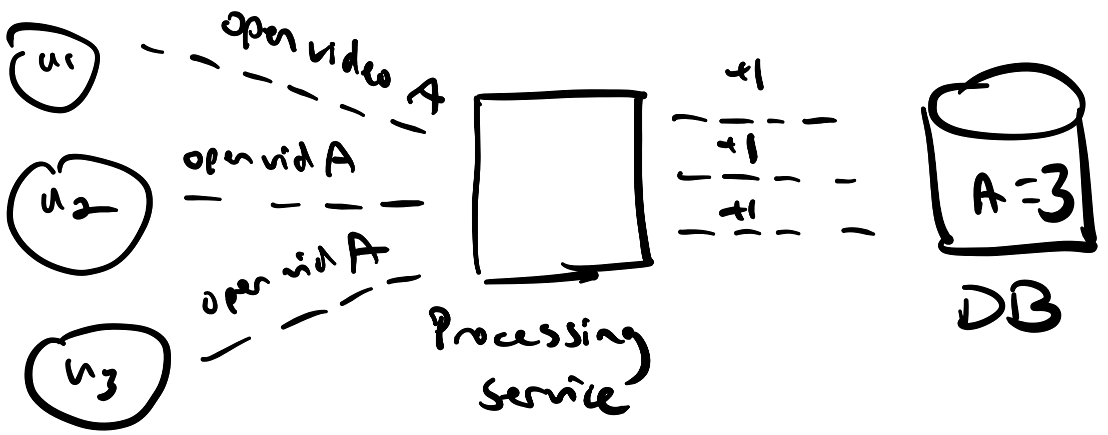
↳ How to not lose data when processing node crashes?

↳ What to do when DB is unavailable or slow?

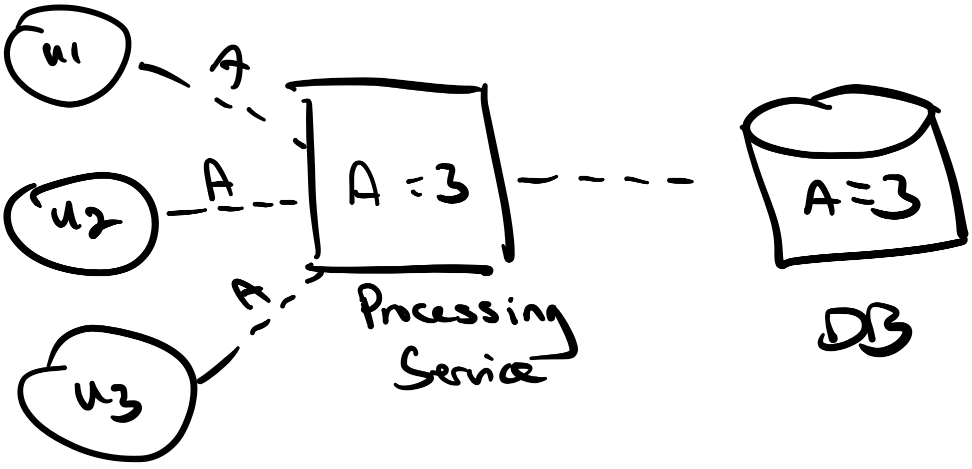
How to make data processing scalable, reliable, & fast?
↳ scalable = Partitioning
↳ Reliable = Replication & checkpointing
↳ Fast = In-Memory

. Data Aggregation Basics
- Should we pre-aggregate data in the processing service?

What does that look like: (Push events synchronously to Processing service)

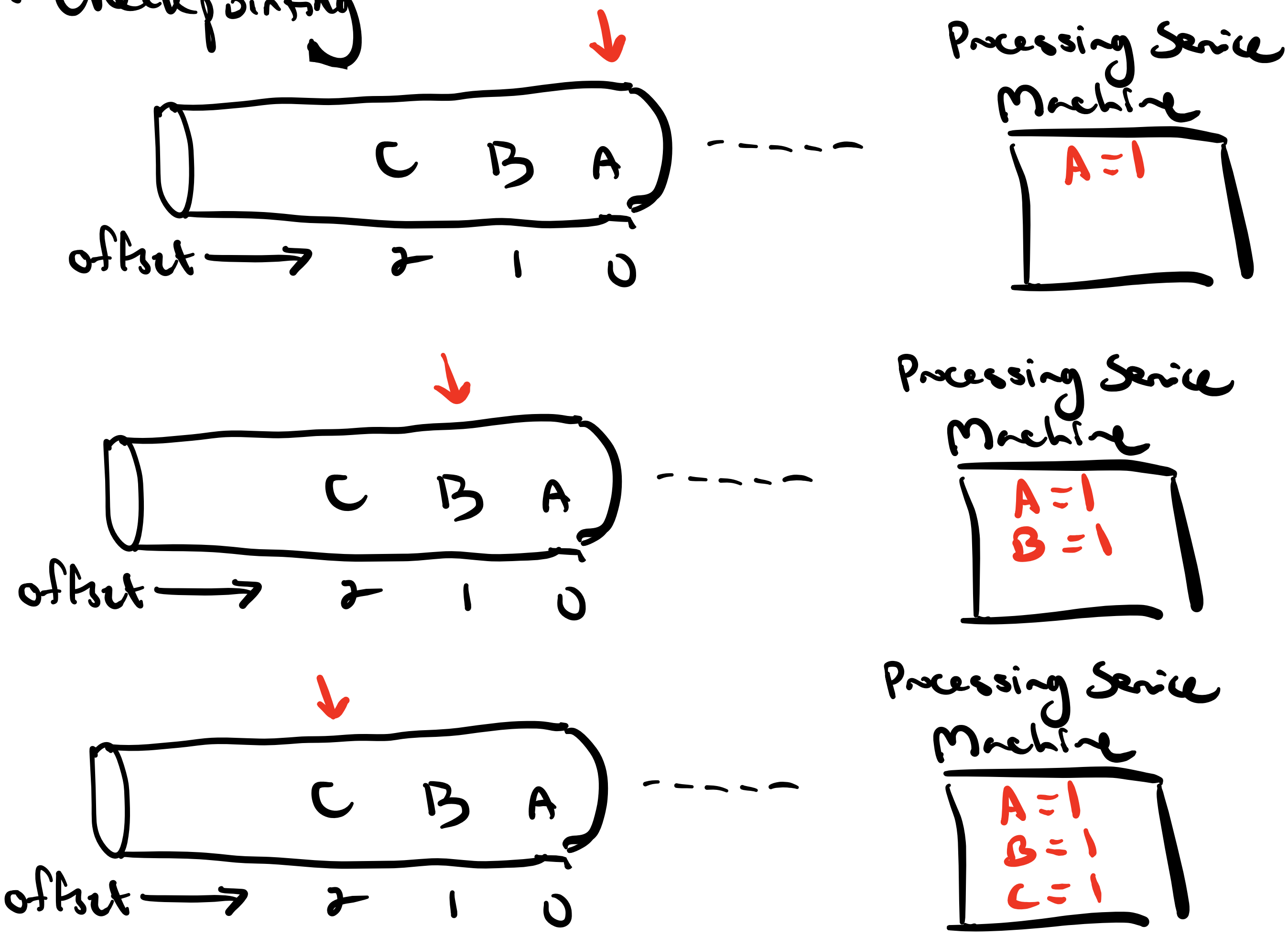


or better (for large systems): (Pull from temp storage)



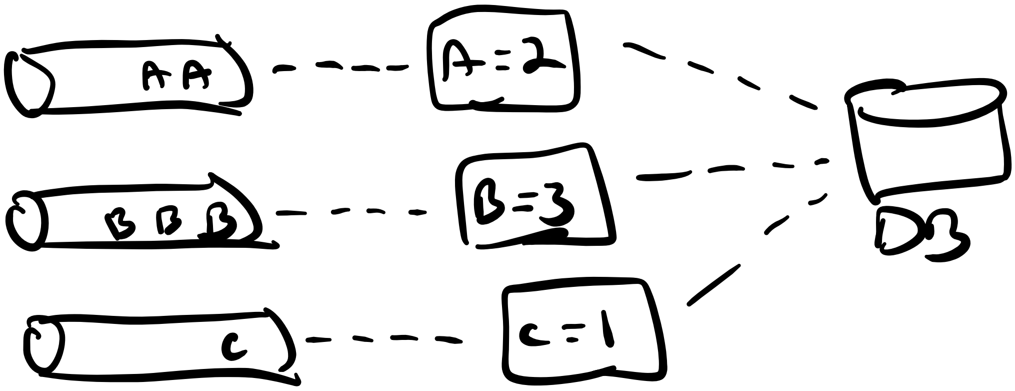
Both options are valid, but Pull has more fault tolerance because data is pulled from temporary storage by the processing service. If the processing service fails, the temporary storage will persist the data.

- Checkpointing


Processing Service Machine
A=1
C B A
offset →
2 1 0
Processing Service Machine
A=1
B=1
C B A
offset →
2 1 0
Processing Service Machine
A=1
B=1
C=1
C B A
offset →
2 1 0

- Every time an event is read from storage, current offset moves forward.
- After several events are processed & stored in the DB, we write the checkpoint to some persistent storage
- If processing machine for service fails, it will be replaced; resumes processing where prev left off

- Partitioning



- Compute Hash & based on this hash we pick a queue
- As we can see, partitioning helps us parallelize events

processing.

- More events, more partitions are made

See Part III for more detail on the processing service Design.