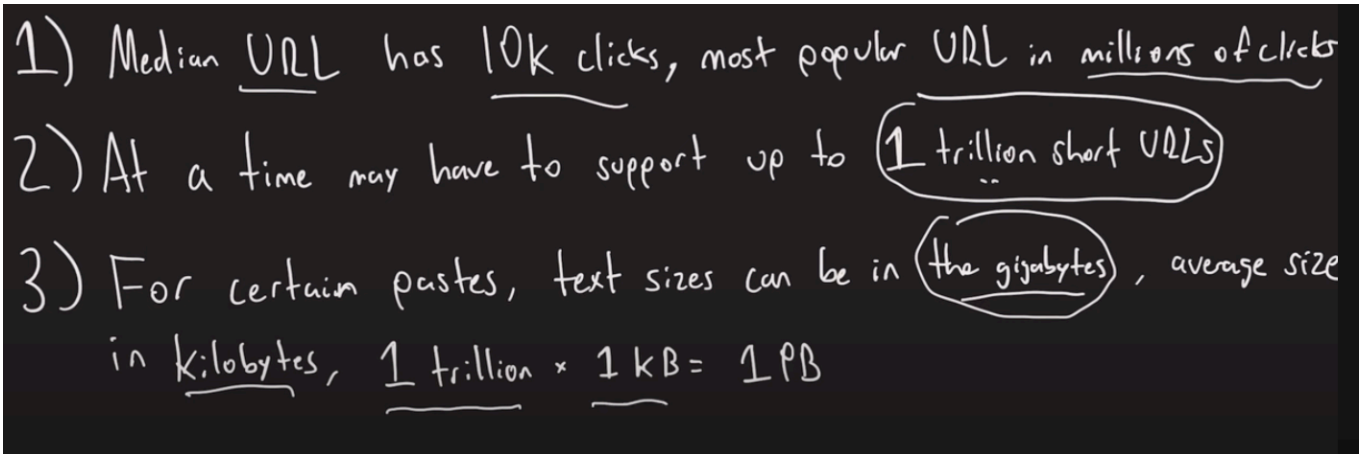
TinyUrl-Pastebin

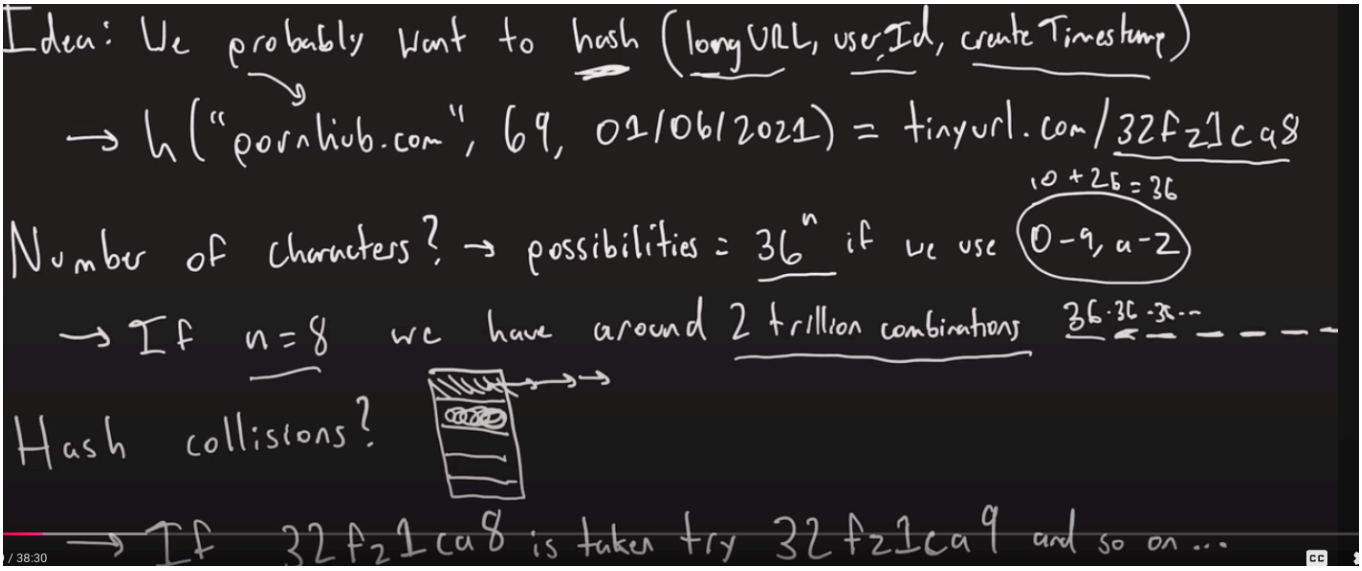
Problem requirements-
1We need to generate a unique shortUrl per paste
2.Per generated url we need to allow the original maker to see how many people have clicked it

Performance Considerations



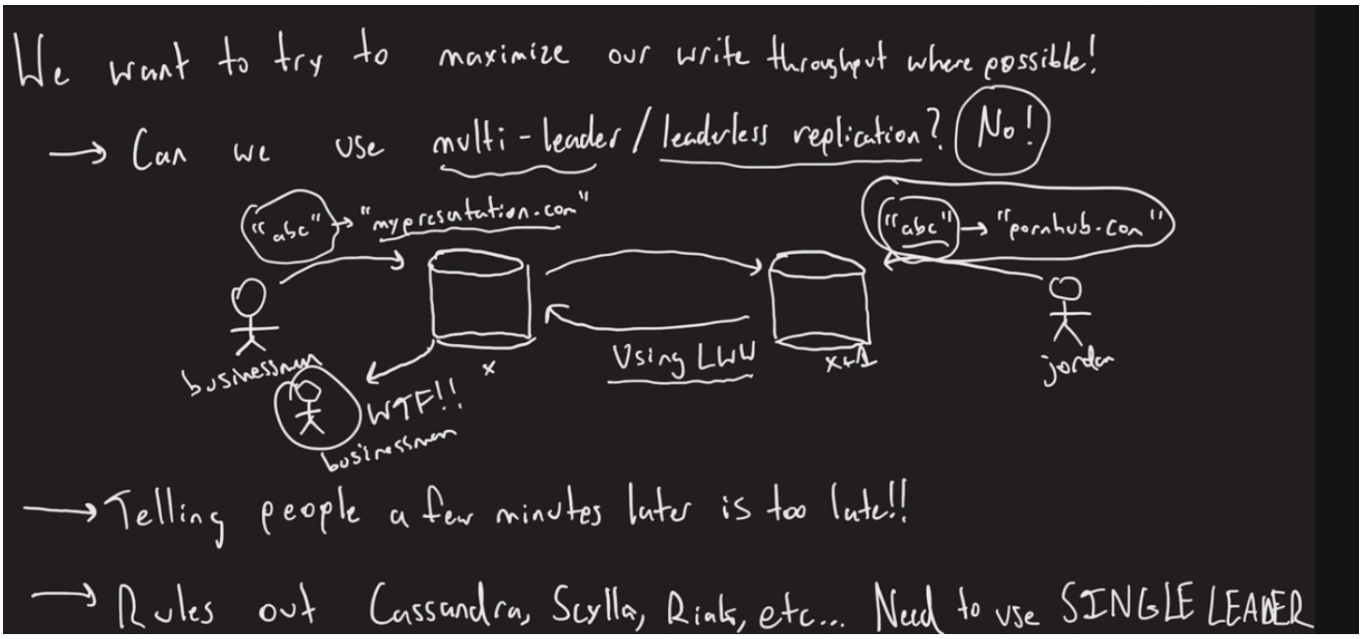
Reads>>Writes so we have to optimize it

Link Generation



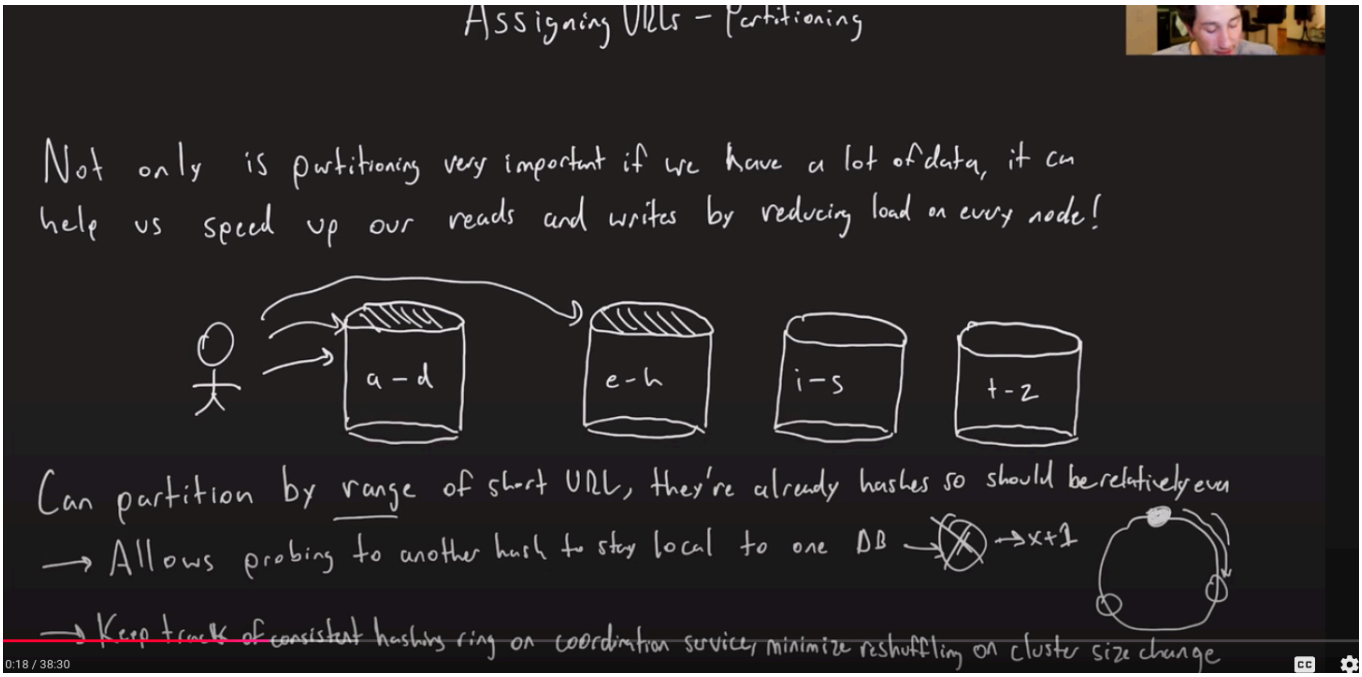
We use linear probing to find the next link and so on

Replication



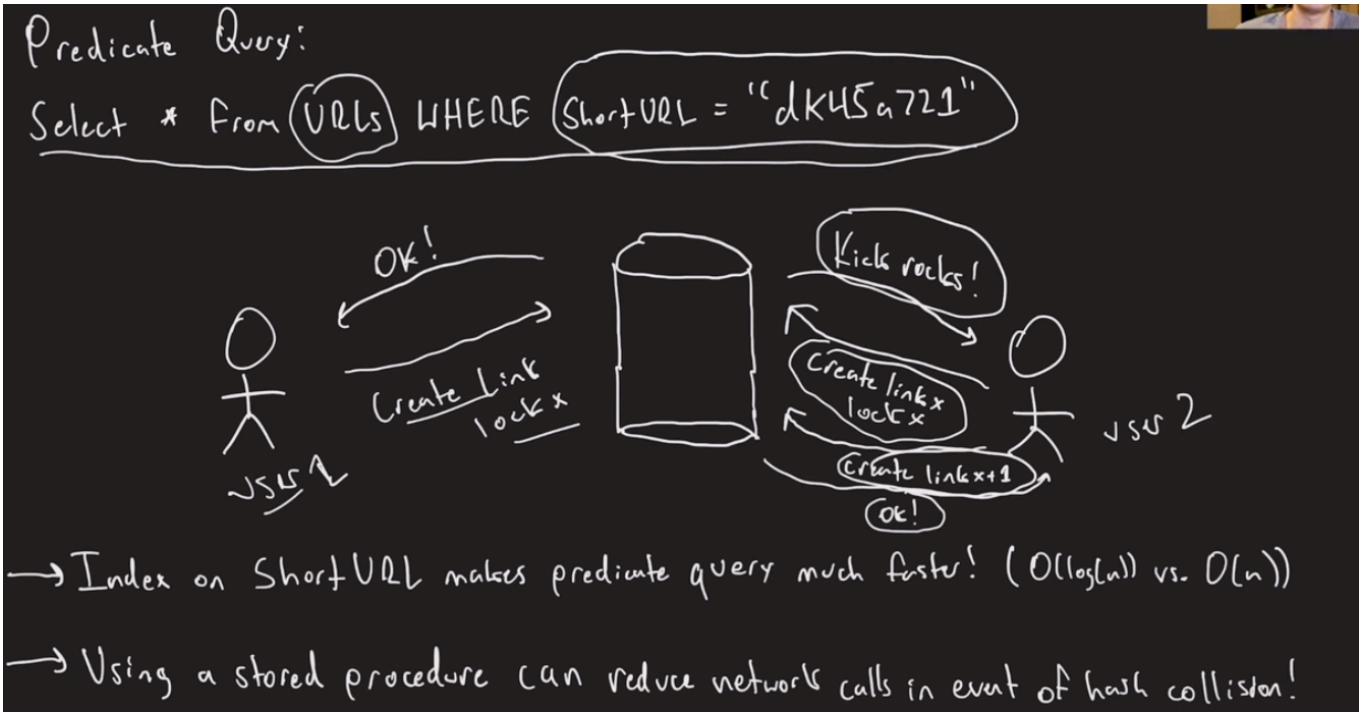
There could be collisions in the links generated if we use a multi-leader replication and even if we alert the person whose link is invalid a few seconds late then it is already too late .
So we need to use SINGLE LEADER.
If we apply last write wins strategy then one of the generated links will be invalid , which will cause problems.
LWW is a mechanism that resolves data conflicts by considering the most recent update as the definitive version. This approach is commonly employed in distributed databases and systems where data replication and concurrent modifications are prevalent. The fundamental principle of LWW is straightforward: when a conflict arises, the system favors the latest write based on a timestamp or a similar time-based marker.

Partitioning



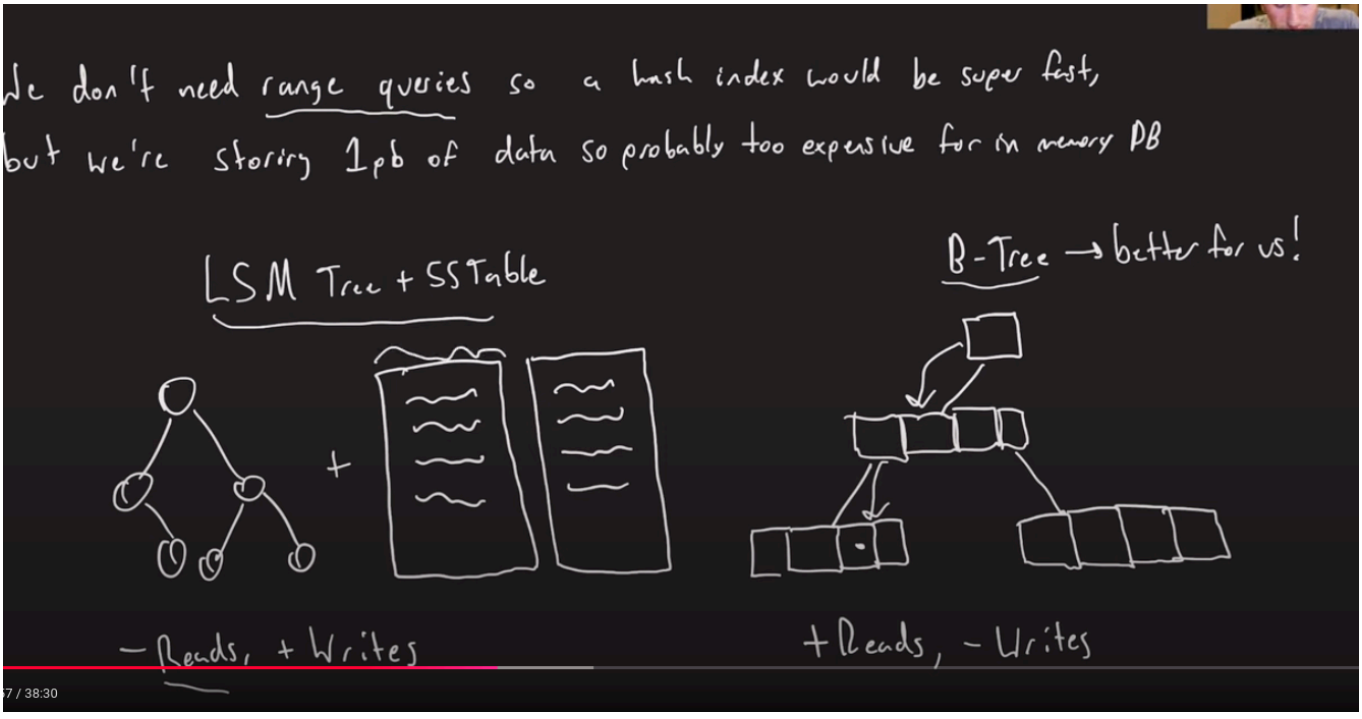
Can use consistent hashing .

Predicate Queries



Predicate Locks -https://www.linkedin.com/pulse/predicate-locks-database-yeshwanth-n/

DB Engine Implementation



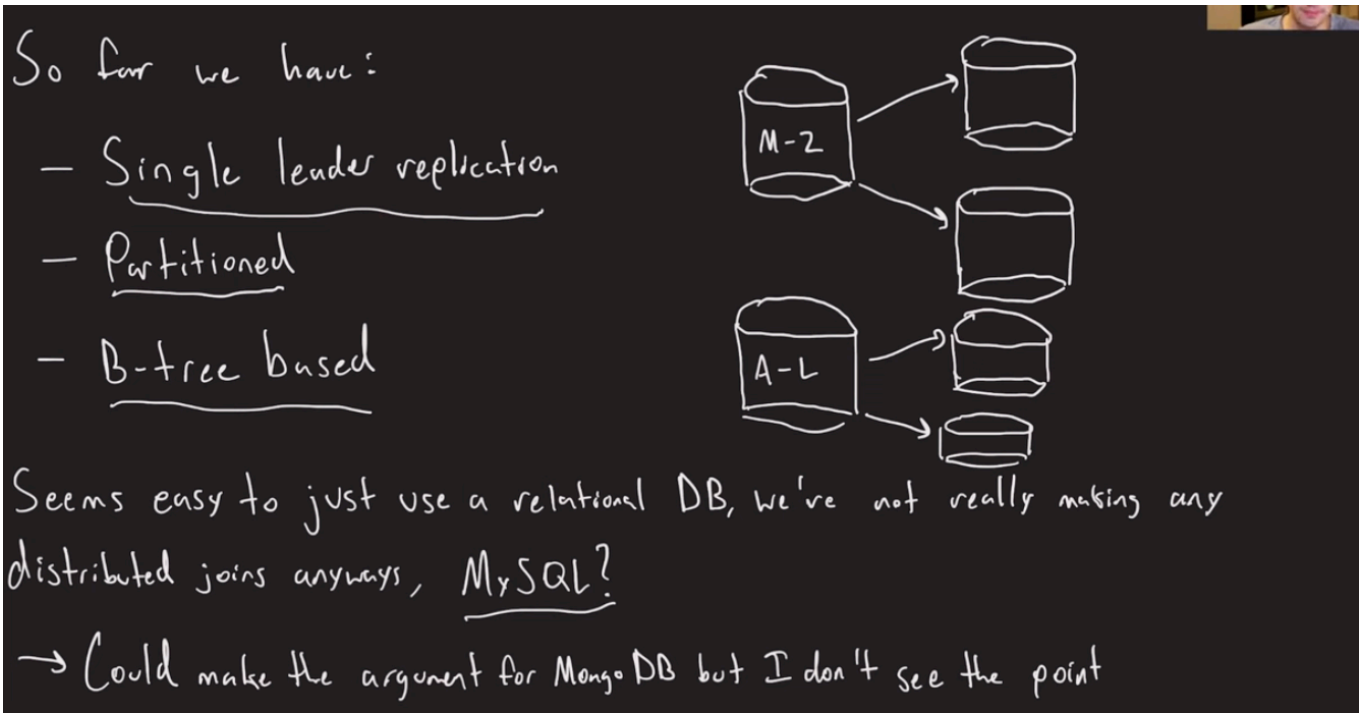
LSM TREES VS B-TREES TRADEOFFS

LSM TREES- WRITES ARE FASTER , READS ARE SLOWER AS WHEN WE READ WE HAVE TO GO THORUGH MULTIPLE SS-TABLES TO READ WHEREAS DURING WRITES ALL WE HAVE TO DO IS WRITE TO THE LSM TREES WHICH ARE FLUSHED TO DISK LATER

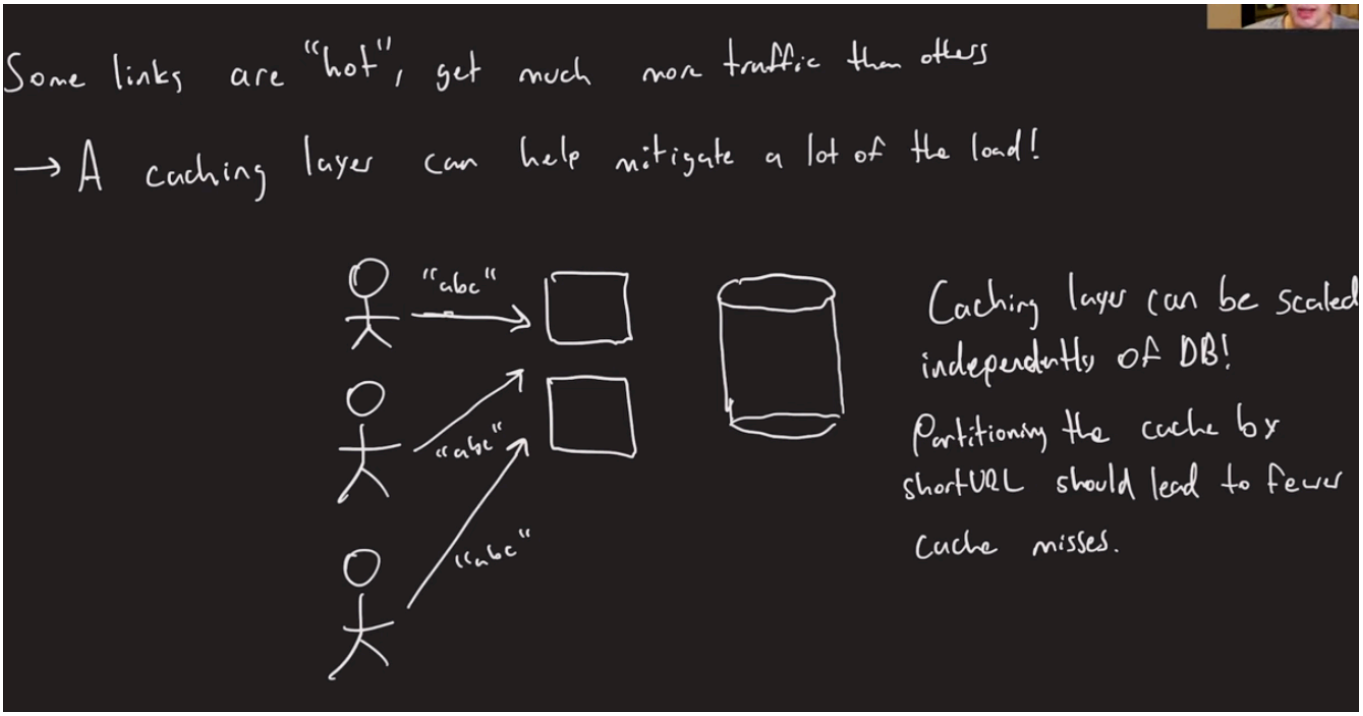
B-TREES - WE WILL ALWAYS BE WRITING TO DISK,WRITES ARE MORE EXPENSIVE AS WE HAVE TO TRAVERSE THE B-TREE AND ALSO END UP RESTRUCTURING THE B TREE WHICH IS EXPENSIVE WHEREAS IN READING IS VERY SIMPLE- IT IS JUST TRAVERSAL OF B-TREE

SINCE WE PREFER FASTER READS WE WILL GO FOR B-TREES

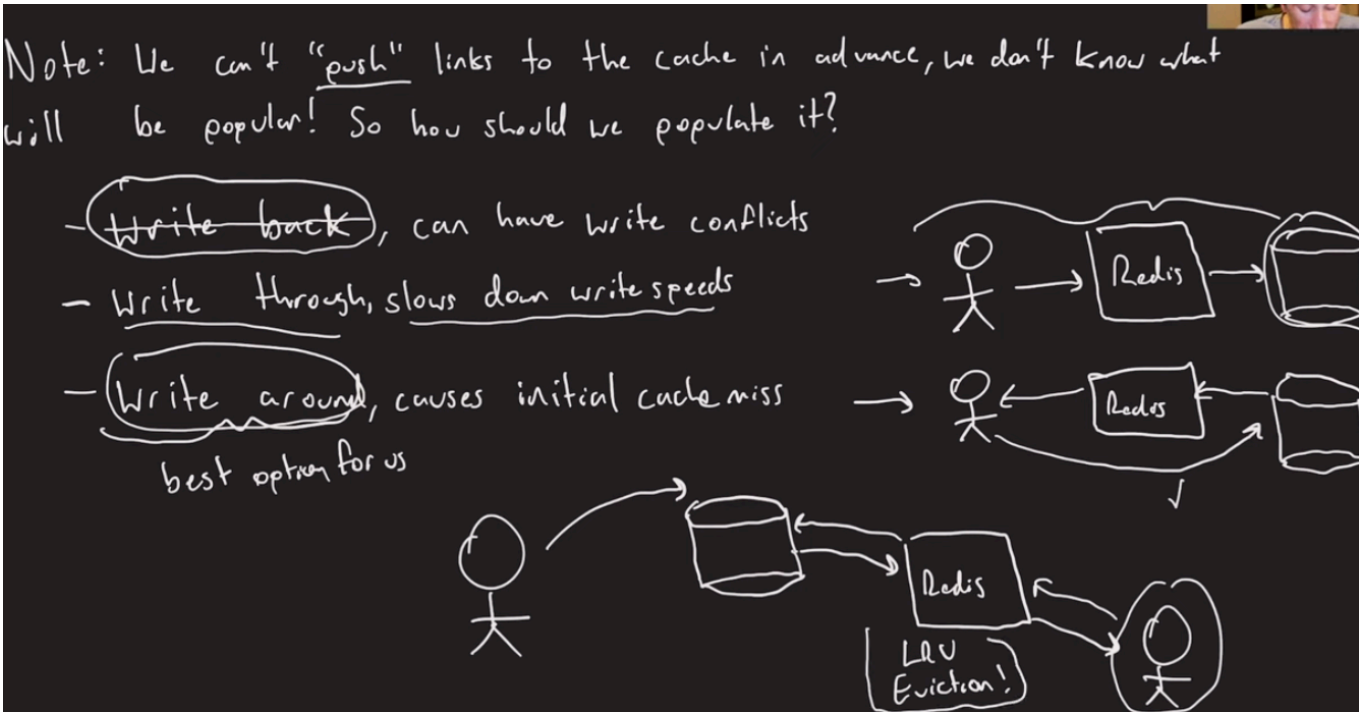
DB CHOICE



Hot Links - links which are very active



Populating the Cache-Strategies



Since we anyways need the most active "hot" links write around cache

ANALYTICS

If we just did a normal x=x+1 there would be a race condition.

so to prevent this we could have locks / use atomic increment , however this would be slow af for the hot links

Idea: We can place individual data somewhere that doesn't require grabbing locks, and then aggregate them later!

Options:

~~Database~~ → Relatively slow

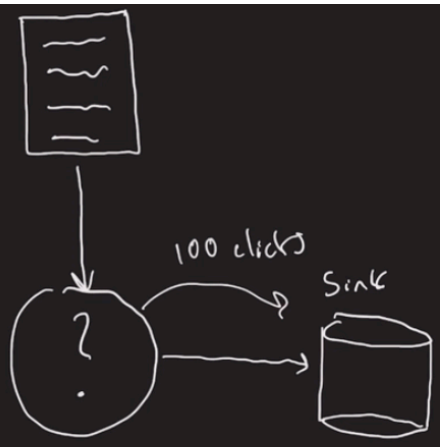
In memory message broker → super fast, not durable 

Log Based message broker → Basically writing to write ahead log, durable 

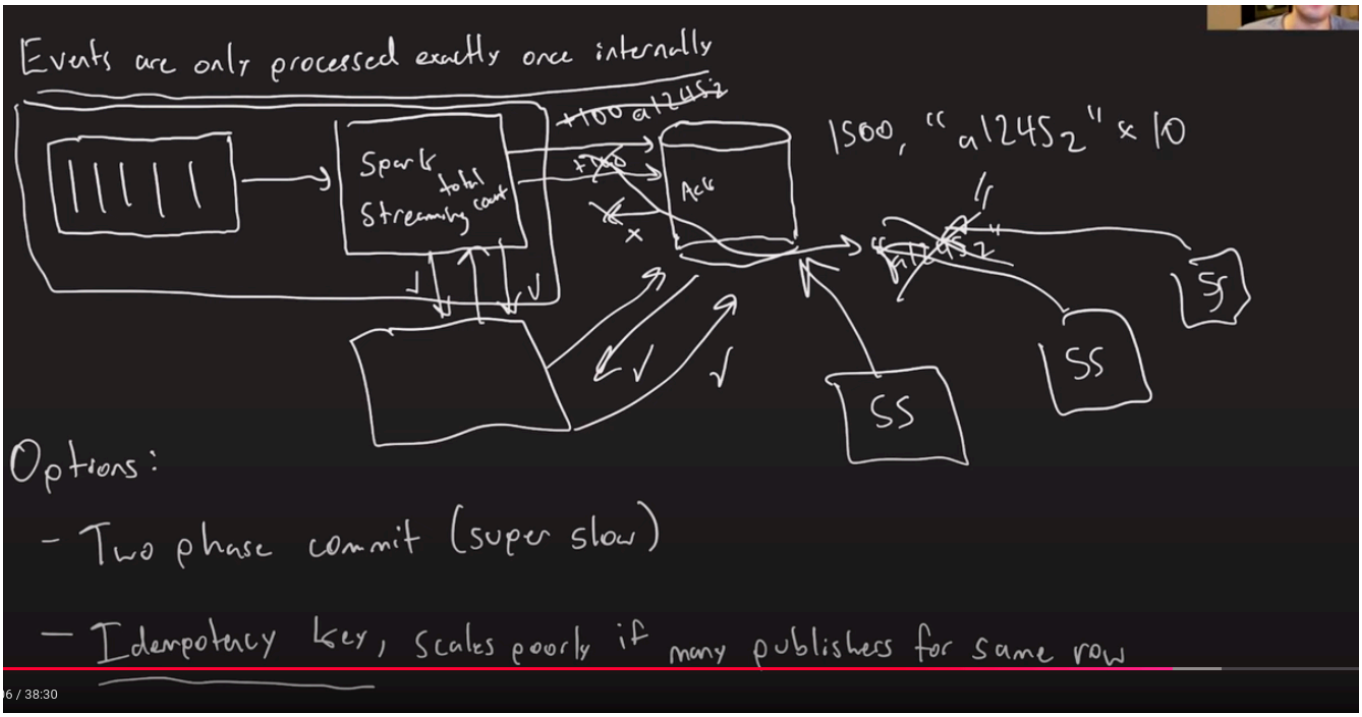
We can dump our click information to Kafka!

Options:

- ~~HDFS + Spark~~
  → Batch job to aggregate clicks, may be too infrequent
- Flink
  → Processes each event individually, may send too many writes to the database depending on implementation
- Spark Streaming
  → Processes events in configurable mini-batch size



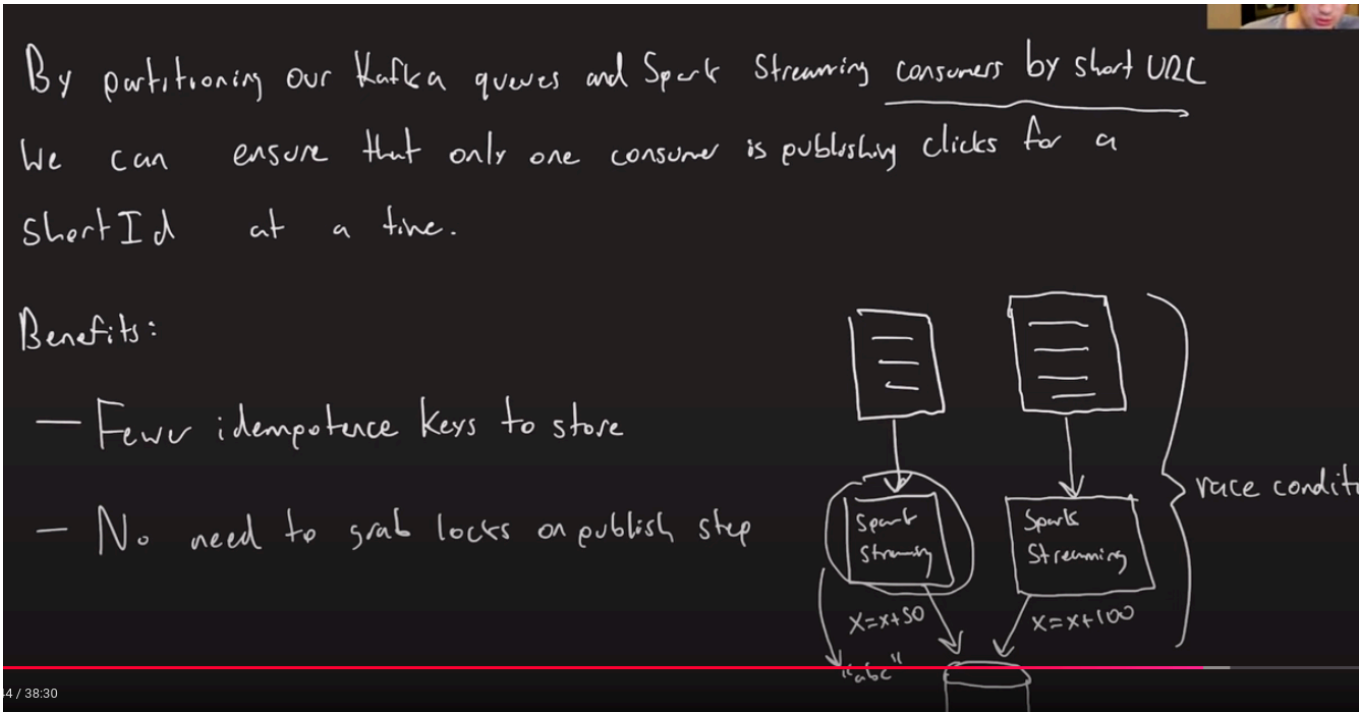
Stream Consumer frameworks enable us to ensure exactly once processing of events via checkpointing / queue offsets!



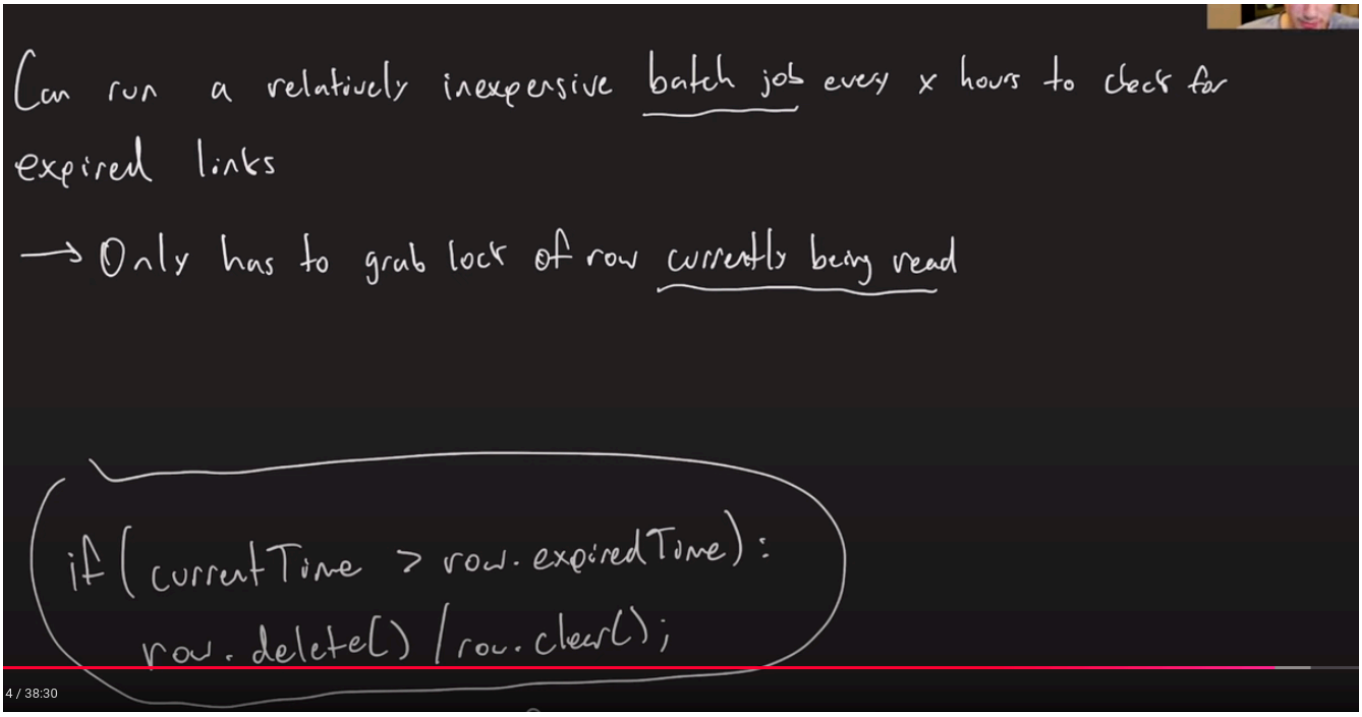
are we sure events are published only once
between the user and the spark consumer we have only once processing internally
however between the spark publisher and the database this may not be the case
lets say the publisher tells the database to update a row , before the database replies back to the publisher it fails and when it comes back up and publisher again tells the database to update the same row becuz it doesnt know if the previous update was acknowledged or not

solution: we can use an idempotency key for each update , if the next update has the same key then we can ignore it , however lets say we have multiple spark publishers for the same row then this would be a problem as now we have to populate the idempotency key for all the publishers

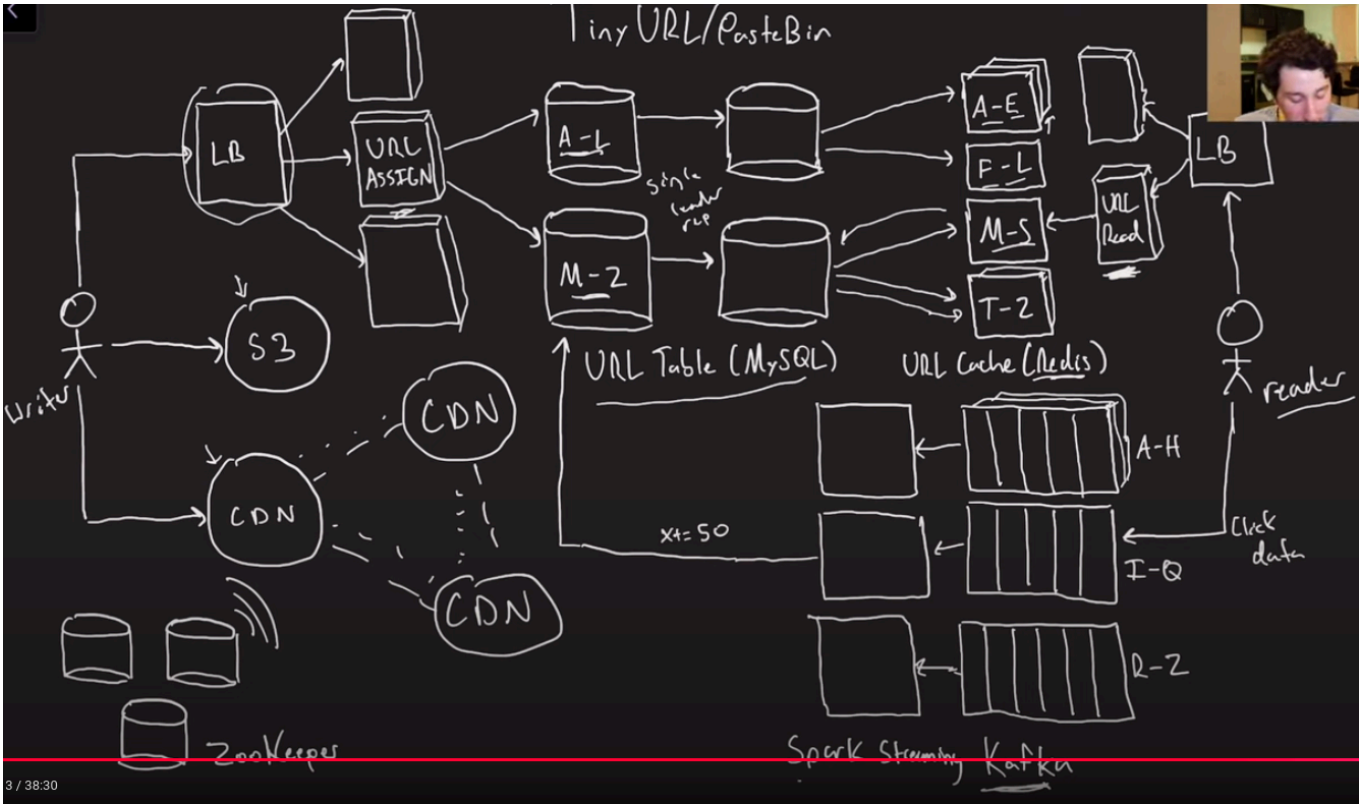
so for this we acn instead make it one publisher for each row/link



Deleting expired links



FInal Setup



Zookeeper is used for keeping all the partitioning info , all the load balancing info .etc
it is strongly consistent and is esentially a key-value store.