

# TinyUrl-Pastebin

Problem requirements-

1. We 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

- 1) Median URL has 10k clicks, most popular URL in millions of clicks
- 2) At a time may have to support up to 1 trillion short URLs
- 3) For certain pastes, text sizes can be in the gigabytes, average size in kilobytes,  $1 \text{ trillion} \times 1 \text{ kB} = 1 \text{ PB}$

Reads >> Writes so we have to optimize it

Link Generation

Idea: We probably want to hash (longURL, userId, createTimeStamp)  
 $\rightarrow h("pornhub.com", 69, 01/06/2021) = \text{tinyurl.com/32f21ca8}$

Number of characters?  $\rightarrow$  possibilities =  $36^n$  if we use 0-9, a-z  
 $10 + 26 = 36$   
 $\rightarrow$  If  $n=8$  we have around 2 trillion combinations

Hash collisions?



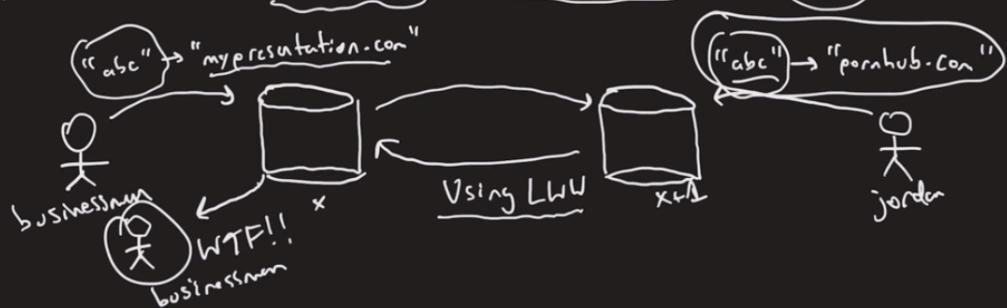
$\rightarrow$  If 32f21ca8 is taken try 32f21ca9 and so on ...

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

Replication

We want to try to maximize our write throughput where possible!

→ Can we use multi-leader / leaderless replication? **No!**



→ Telling people a few minutes later is too late!!

→ Rules out Cassandra, Scylla, Riak, etc... Need to use **SINGLE LEADER**

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 .

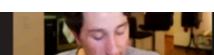
So we need to use **SINGLE LEADER**.

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

### Assigning URLs - Partitioning



Not only is partitioning very important if we have a lot of data, it can help us speed up our reads and writes by reducing load on every node!



Can partition by range of short URLs, they're already hashed so should be relatively even

→ Allows probing to another hash to stay local to one DB → ~~x~~ → x+1

→ Keep track of consistent hashing ring on coordination service, minimize reshuffling on cluster size change



Can use consistent hashing .

## Predicate Queries

Predicate Query:

Select \* From URLs WHERE ShortURL = "dK45a721"

→ Index on ShortURL makes predicate query much faster! ( $O(\log(n))$  vs.  $O(n)$ )

→ Using a stored procedure can reduce network calls in event of hash collision!

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

## DB Engine Implementation

We don't need range queries so a hash index would be super fast, but we're storing 1pb of data so probably too expensive for in memory PB

LSM Tree + SSTable

- Reads, + Writes

B-Tree → better for us!

+ Reads, - Writes

## LSM TREES VS B-TREES TRADEOFFS

LSM TREES- WRITES ARE FASTER , READS ARE SLOWER AS WHEN WE READ WE HAVE TO GO THOROUGH 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

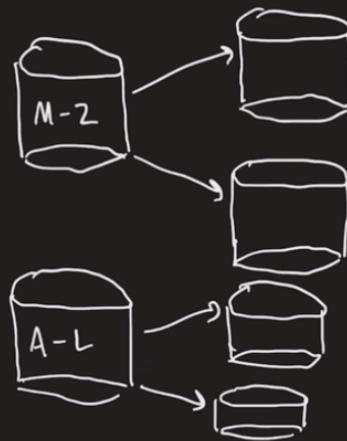
B-TREES - WE WILL ALWAYS BE WRITING TO DISK,WRITES ARE MORE EXPENSIVE AS WE HAVE TO TRAVERSE THE B-TREE AND ALSO END UP RESTRUCTURING THE B TREE WHICH IS EXPENSIVE WHEREAS IN READING IS VERY SIMPLE- IT IS JUST TRAVERSAL OF B-TREE

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

DB CHOICE

So far we have:

- Single leader replication
- Partitioned
- B-tree based



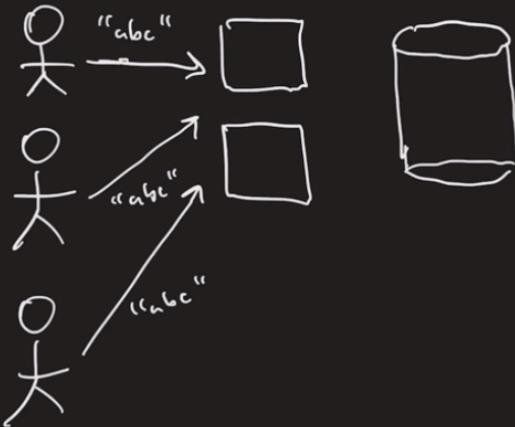
Seems easy to just use a relational DB, we're not really making any distributed joins anyways, MySQL?

→ Could make the argument for Mongo DB but I don't see the point

## Hot Links - links which are very active

Some links are "hot", get much more traffic than others

→ A caching layer can help mitigate a lot of the load!



Caching layer can be scaled independently of DB!

Partitioning the cache by shortURL should lead to fewer cache misses.

## Populating the Cache-Strategies

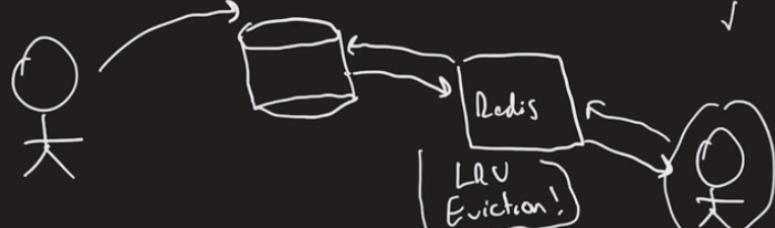
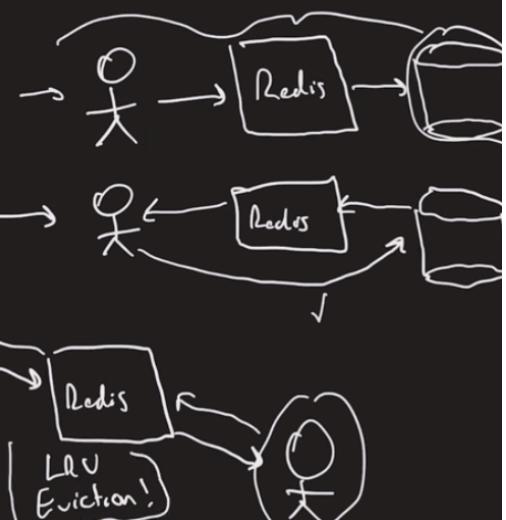
Note: We can't "push" links to the cache in advance, we don't know what will be popular! So how should we populate it?

- Write back, can have write conflicts

- Write through, slows down write speeds

- Write around, causes initial cache miss

best option for us



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  $O \rightarrow O \rightarrow O \rightarrow$

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



We can dump our click information to Kafka!

Options:

- ~~HDFS + Spark~~

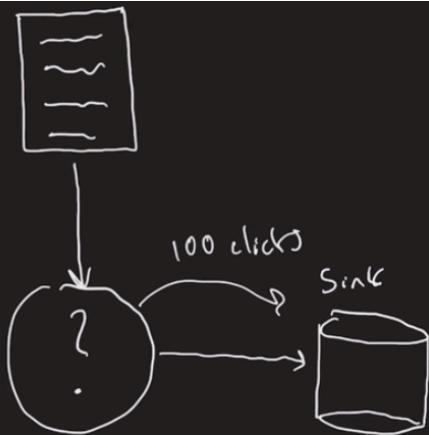
→ Batch job to aggregate  
Clicks, may be too infrequent

- ~~Flink~~

→ Processes each event individually,  
may send too many writes to the  
database depending on implementation

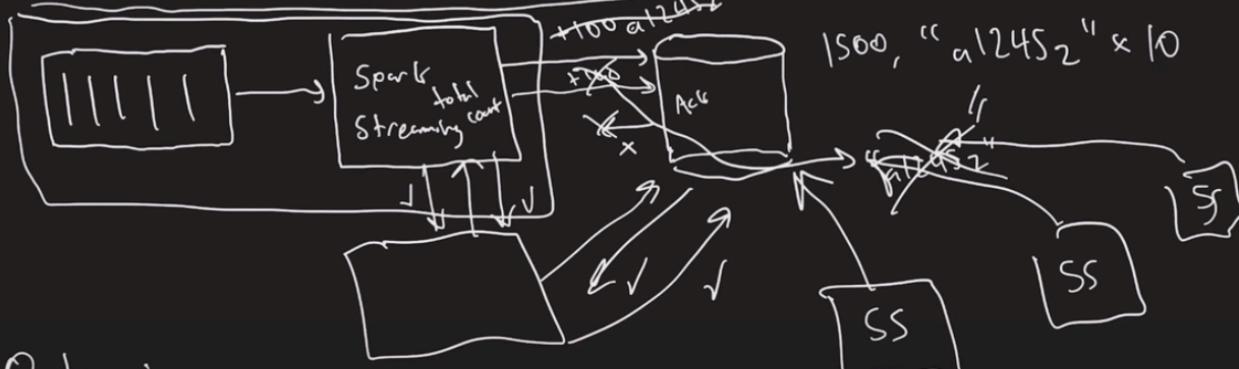
- ~~Spark Streaming~~

→ Processes events in configurable  
mini-batch size



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

Events are only processed exactly once internally



Options:

- Two phase commit (super slow)
- Idempotency Key, scales poorly if many publishers for same row

6 / 38:30

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

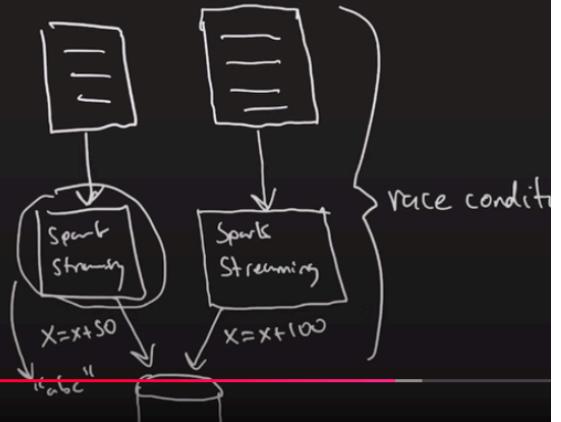
solution: we can use an idempotency key for each update , if the next update has the same key then we can ignore it , however lets say we have multiple spark publishers for the same row then this would be a problem as now we have to populate the idempotency key for all the publishers

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

By partitioning our Kafka queues and Spark Streaming consumers by short URL  
We can ensure that only one consumer is publishing clicks for a  
Short ID at a time.

Benefits:

- Fewer idempotence keys to store
- No need to grab locks on publish step



4 / 38:30

Deleting expired links

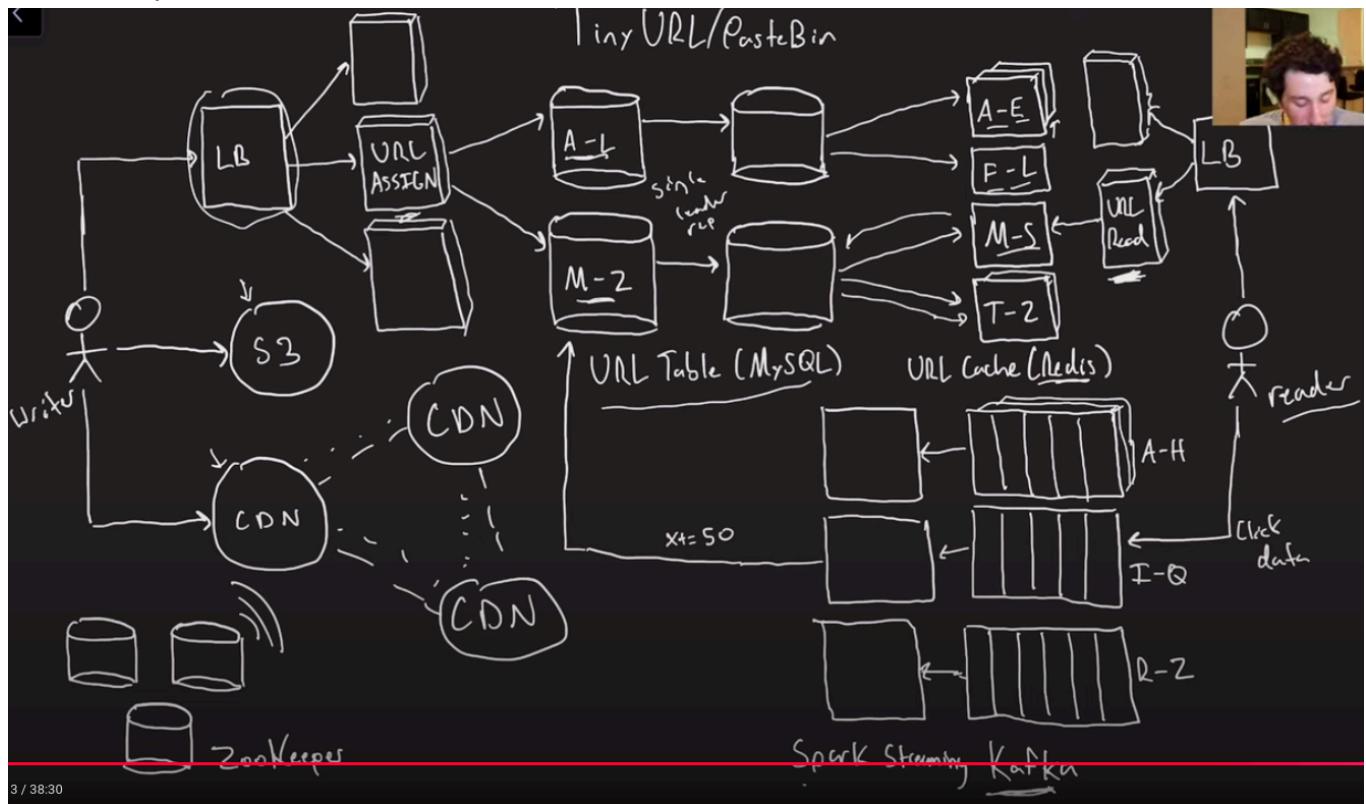
Can run a relatively inexpensive batch job every x hours to check for  
expired links

→ Only has to grab lock of row currently being read

if (currentTime > row.expiredTime):  
 row.delete() / row.clear();

4 / 38:30

## 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.