URL Shortener
https://nlogn.in/designing-a-realtime-scalable-url-shortening-service-like-tiny-url/
Educative - grokking system design interview-3.pdf

1. **Requirement**

****
Funtional

1. Long to short, globally unique across customer
2. Support of customisation - brand domain based, subdomain based short url
3. Short url expiration policy
4. Analytics
    1. Recent click stats available with a 5 min lag
    2. Long look-back stats can be computed daily
5. Prevent abuse of service,
    1. access based on userid/api-dev key
    2. rate limit amount of new url's created and existing accessed in a given time
    3. Avoid visiting restricted domains (maintain blacklist domains, bloom filter based scaling)

Non functional

1. Highly available
2. Real time shorten url generation
3. Shorten url are not guessable/predictable

2. **Why/Benefits**

****
1. Saves space when shared, rendered across the internet
2. better suited for social media, email campaigns

3. **Interface/API**

****
1. For given long url, generate short url
    1. **create_shortURL(** long_url, custom_url(optional), expiry_date(optional)**)**
2. for a given short url, fetch long url to redirect ...HTTP 302 redirect with location header, if not found —HTTP 404
    1. redirectURL(short_url)

4. **Scale - Read, Write, Storage**

****
1. **Read** (hits per month/sec) & **Write** (new data created per month/sec)
    1. **assumption -** write - **300M writes in 30 days** new unique URL shortened per month**/115 per sec(Write)**
    2. 100:1 Read versus Write. Read **- 30B reads per month/11.5K per**

sec(Read - multiple same URL access across the day)

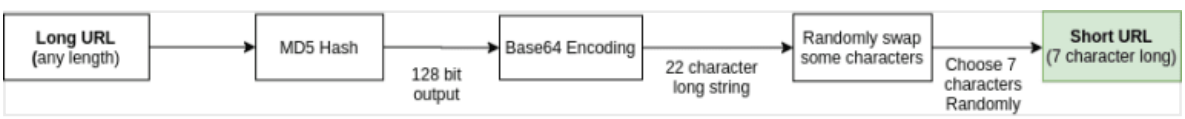
2. **Storage** (write per sec * per object size)
    1. **assumption** - store for 5 years
    2. Long url(upto 2048 chars(as per RFC) - 2KB)
    3. Short url (7 chars - 7 byte) + space for protocol, domain name
    4. created_at - timestamp type —4bytes
    5. expiration - date time — 8 bytes
    6. 500M * 5*12 = 30000M = 30B, with roughly 2k bytes each, 30B*2k = **60TB**
3. **Bandwidth**
    1. Incoming/store new - 115 writes per sec *2KB = 230 KB/sec
    2. outgoing/fetch existing - 11.5K reads per sec * 2KB= 23,000 KB = 23MB/sec
4. **Cache - 20% of daily unique reads**
    1. **assumption** - 20% of url's generate 80% of traffic and therefore are hot
    2. Storage across day = 11.5K reads*24*3600 * 20% * 2KB = **993.6 GB (actual will be less due to multiple duplicate access in 20K)**
5. **Short url length**
    1. URL with length 7, will give 64^7 = ~5 Trillion URLs

5. **Shortening Algo**

****

APPROACH 1 - "Md5 + Base64 + Random pick 7 chars" approach

1. **Variable len input URL string to fixed size intermediate o/p**
    1. MD5 HASH = 16 byte/128bit
    2. MD5 is fast algo and generates shorter fixed o/p
2. **Hashing alternative if collisions are to be avoided**(Since every hash also has possibility of collisions(though very low 10 pow -29) and re-generate and keep trying until you don't collide
        1. Maintain **unused pre generated unique set of url's** and consume as needed. In case of distributed setup, nodes request for url via some coordinator.
3. **Ensure short url chars are safe for transfer(**non-HTTP chars to HTTP compatible chars**) -** Md5 to base 64 encoding(I/p 16byte, base 64 uses 6 bit per character) (16*8/)6 —> 22 characters
4. **Reduce chance of collision -** shuffle
5. **and limit to 7 char short url** — pick 7 characters
6. **Ensure globally across customer unique url - append** the **userid/API key** to a the long URL and then do the shortening process (assuming user is logged in)



CONS:

1. random 7 chars could lead to collisions

2. handling multiple request in parallel and issuing new unique url's without duplication will be challenge
3. Single point of failure and performance bottleneck

APPROACH 2 - Generating/sharing unique url's in distributed way

1. cluster of id generators that reserve chunks of the id space from a central coordinator (e.g. ZooKeeper)
2. independently allocate IDs from their chunk

5. **Click Analytics Layer**

****

1. For stats to be updated once a day at eod, store in hdfs and run map reduce jobs
2. For near real time stats, buffer clickstream data in messaging layer -> aggregation layer consumes and generates metrics -> writes to persistent store

7. **Data Model -** NOSQL can be used due to high scale and no joins based query pattern across tables

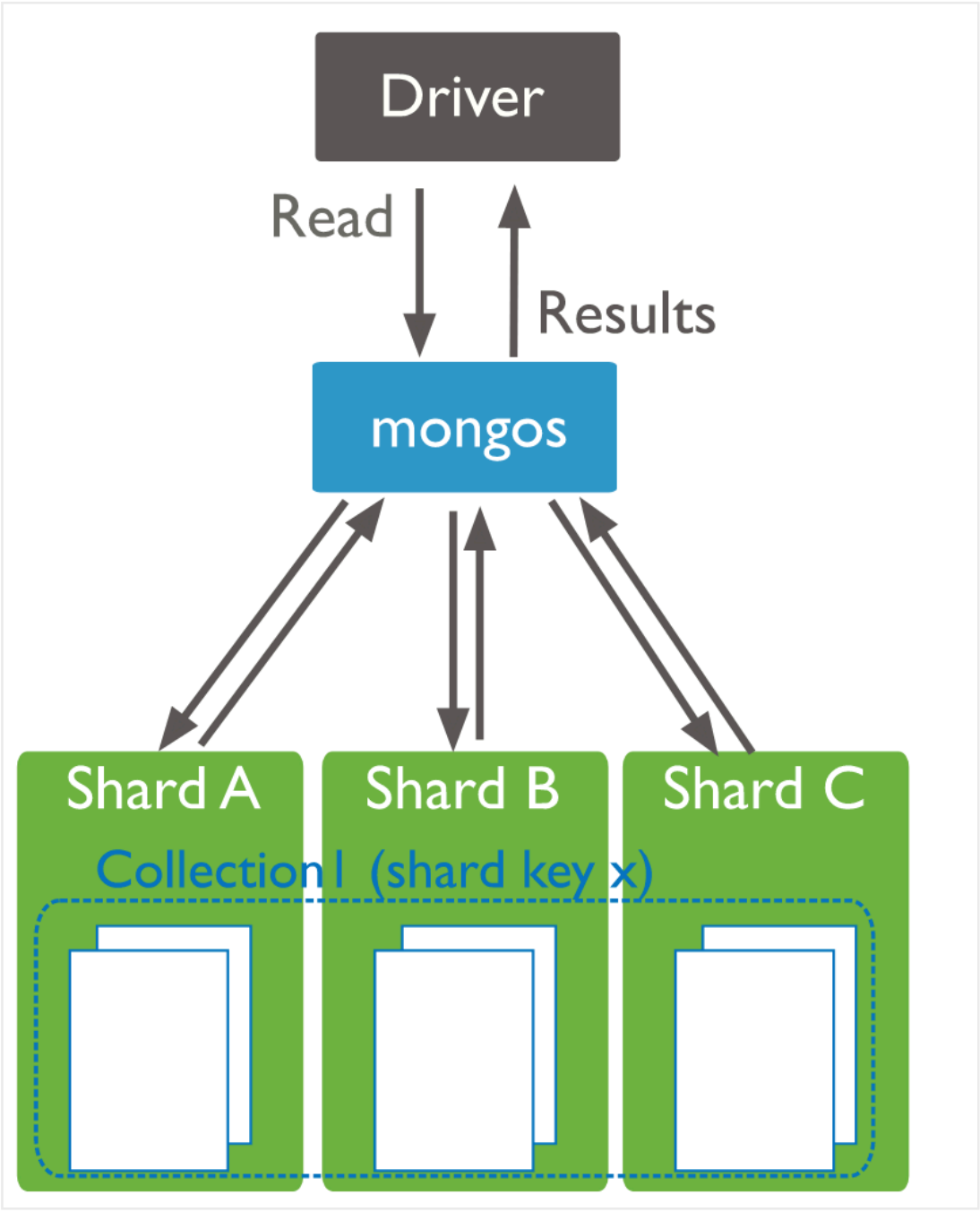
****

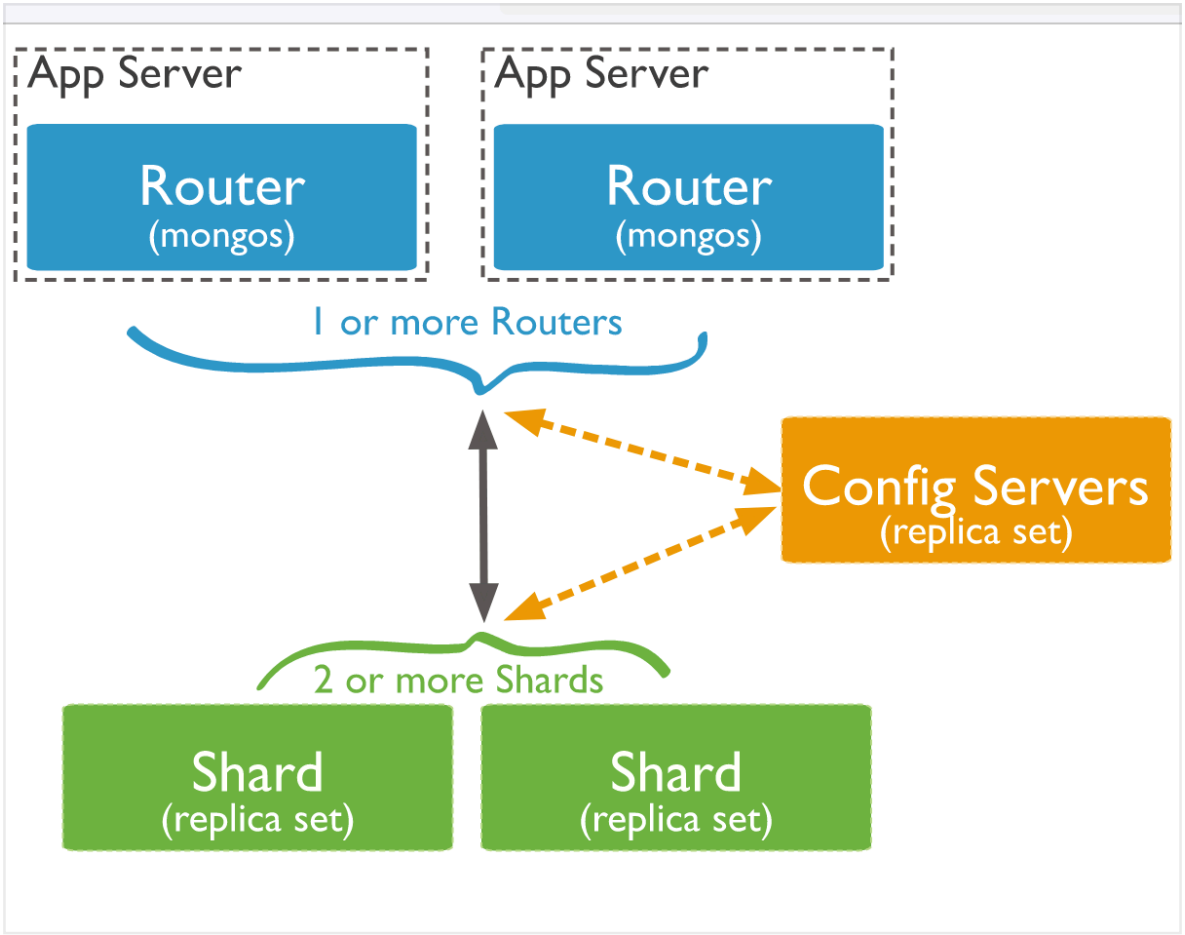
1. **user table**
    1. userId/api key — int, auto increment NOT NULL, PK
    2. Name — varchar(100)
    3. Email — varchar(255)
    4. phone — varchar(10)
    5. created — timestamp
    6. updated — timestamp
2. **Short url table**
    1. Short url - varchar(7)
    2. Orig url - varchar(2048)
    3. userId/api key —- int, FK
    4. expiry ——DateTime
    5. created — timestamp
    6. updated — timestamp
    7. status - active/expired

7. **DB Sharding -**

****

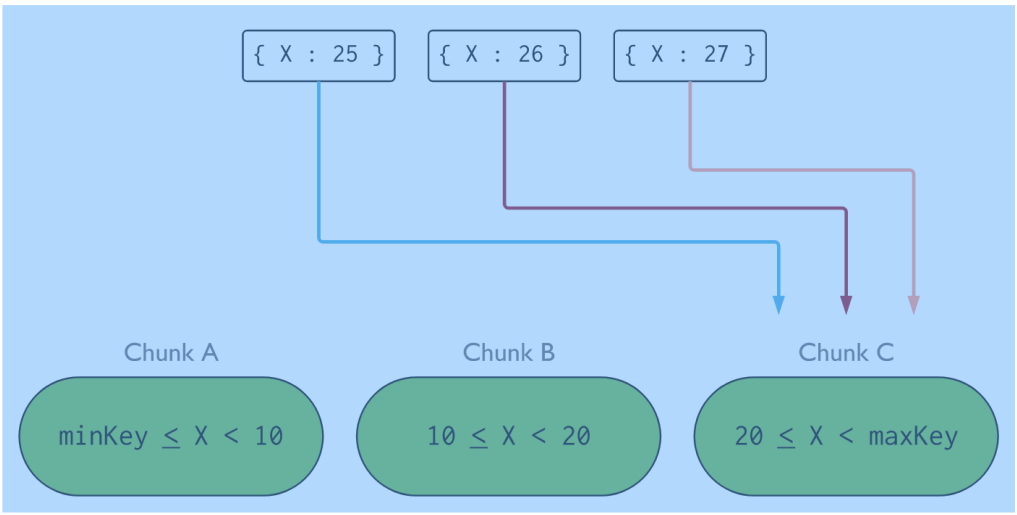
Mongo DB
https://docs.mongodb.com/manual/sharding/
https://docs.mongodb.com/manual/core/hashed-sharding/


Driver
Read
Results
mongos
Shard A
Shard B
Shard C
Collection1 (shard key x)



Hashed vs Ranged Sharding

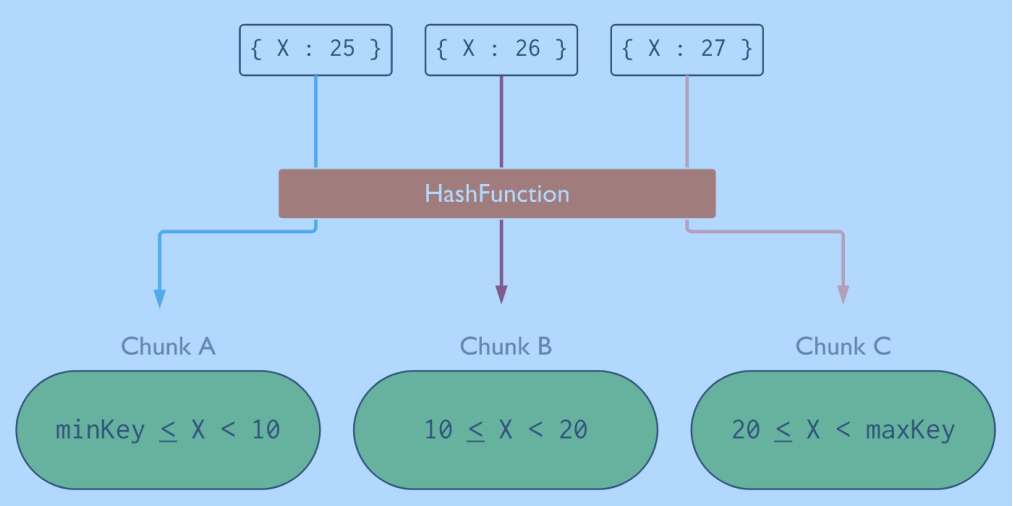
Given a collection using a monotonically increasing value `X` as the shard key, using ranged sharding results in a distribution of incoming inserts similar to the following:



click to enlarge

Since the value of `X` is always increasing, the chunk with an upper bound of maxKey receives the majority

By using a hashed index on `X`, the distribution of inserts is similar to the following:



click to enlarge

Since the data is now distributed more evenly, inserts are efficiently distributed throughout the cluster.