Round Robin Method
since no key and partition is sent

m1 — 0
1
2

hash % (no. of partition)
same like modulo operation in DB sharding

Producer

Consumer
keeps polling if there are any new reque
(may be short polling)
(ie. pull based technique)

Key - (optional)
Value - (actual msg)
Topic - topic0
partition : (optional)
decides which message to be processed in which queue

no.of topics and partition queues
in broker is introduced by the user

topic0

Broker

Dead letter Queue

retry : 03
Dead letter Queue — can be used in SAGA pattern as well

dropped after fixed no.of tries

producer — consumer

kafka is used where bulk requests are processed
ola, uber, telegram

Discord Server
Ola/uber
server --> request

9 out of 10 fortune companies use kafka

msg : Bug / cannot be resolved

Rabbit MQ — similar to kafka

like partition queues
queue0
key : 123qwe
IMP
producer — Exchange
queue1
consumer1
key : 123rty
key : 456qwe
queue2
consumer2

msg :
key : 123qwe

Exchange : Rules that decide, msg will go in which Queue
Fan-out : Broadcast — i.e. to multiple consumers
Direct Exchange : match MK to RK.
Topic Exchange : wildcards are allowed. (*)

Keys :
Message key : key that comes with the msg
Routing key : exhange key

Topic Exchange :
MK : 123qty
if RK (starting portion till *)string matches MK
RK : 123q*

Re-Queue mechanism — if requeue mechanism fails then it goes to dead letter queue

difference
kafka : Pull based technique
RabbitMq : Push based technique

New Topic — Caching

Expensive : 1GB
2000 TB
server — cache — DB
key1 : 200ms

Cache :
Temporary storage
TTL (Time to Live)

DB : 200 ms
Cache : 20 ms

Types of Cache:
CDN --> caching use (Static resource) — and dynamic but difficult
Load Balancer (web pages)
Server side caching (Redis)
Proxies also use caching.

Distributed caching --> (uses) Consistent hashing
since servers are distributed

c1 c2 c3

Cache Data POST/GET

1. Cache Aside technique (Retrieve)

server → Cache (cache hit), server → DB

1. Chech cache
2. if found (cache hit) return the data
3. if not, (cache miss) fetch the data from DB, put it into the cache and then return thr data

Pros :
Simple to implement
DB doc structure and cache can be different
(i.e userId data type may be int / i.e userId data type may be string)

Cons:
For every new data there will be always be cache miss.

Pre-heating the cache — like data of famous people on insta need to preheat

2. Read through cache: (Data retreival)

server ⇄ Cache (cache hit); Cache ⇄ DB (cache miss)
cache takes data from DB

1. Read through cache
2. if found, cache hit, return the data
3. Cache miss, cache internal library will automatically fetch the data from DB, and put it in cache and return to the server

pros :
Client don't need to bother about DB logic.

Cons :
Cache doc structure should be same as DB.
Starting --> cache miss.
Pre-heat the cache. (solution)

Write through cache : (To POST data)

server → Cache, server → DB — Transaction

1. Write to cache and DB simulaneously.

pros : Consistency will be high.
COns:
Slow.
uses 2 phase commit (overhead) — slow
locking mechanism to lock distributed servers

2. Write back/behind cache:

server → Cache, server → DB

Write into cache and async write to DB
Pros: Fast

Disadvantage :
Inconsitency can come.

3. Write Around technique

Cache — key = 0 (DIRTY) — not updated
server — key = 1 → DB — key = 1

Directly write Data to DB
Do not write data to cache, just invalidate the data in cache. (mark it as dirty)

Advantage :
without using any 2PL, it can maintain consistency.
(2 phase)

Cache Eviction Policies

| v1 | v2 | v3 | v4 |

v5 — new entry in cache

Limited storage

TTL

How to and what to remove data from cache.

LRU Cache (Least Recently used)
MRU (most recently used)
LFU (Least Frequently used)
FIFO(First in first out)
Random

LRU (Least Recenlty used)
when the cache is full,
LRU evict the item that hasnt been accessed in the longest time.

G → | G | E | F | B | — D C (B: LRU)
i.e. maintains using timestamp removes data with high timestamp value

Pros : simplicity
cons : Assume past record patterns will repeat in future. — evicts B assuming B will not be used in future

Most recently used (MRU) — removes most recently used

| E | C | B | A | (E: MRU)
i.e. maintains using timestamp removes data with least timestamp
reverse of LRU

Disadvantage :
Do not bother about past records.

Least Frequently used (LFU)

| A | E | D | C |
3 2 1 1
maintains the count of the requests
LFU (if frequency is same then LRU is used)

FIFO (First in- first out)

E → | E | D | C | B | — A
FIFO (bank queue)

pros : simplicity

Do not need to maintain any timestamp or frequencey

Cons :
Cache miss will be higher. — cache is almost same as LRU or LFU but FIFO is better

practically : FIFO

Assumptions --> Algorithms — in HLD

Random cache eviction

| E | F | C | B | — evicts randomly

Redis --> TTL

key : userName
value : Aditya
ttl : cache specific --> Redis

auth token
user Id
session id
→ time bounded

caching is used everywhere
Load balancers
CDN --> dynamic caching also supported.
API gateway
Proxies
Server side caching

client side caching --> cookies

Authenticaton & auhtorization HLD