Basics

30 March 2024 02:07 PM

SHARDING → dividing data into smaller fragments + storing them on different distributed machine

TYPES

- → Range Based / Dynamic
  - → Each shard holds a range of values
  - → Ex:- date ranges, time ranges
  - → when ordering + ranges are there in data
- → Hash-Based / Algorithmic / key based
  - → use a hash func^n on data to get which shard
  - → used when there is no natural order (ids)
- → Directory Based
  - → you manually maintain a data → shard mapping
- → Geo-Based
  - → shard based on geo locations or names to locations

Partitioning → same logic but logically within same system

PROS

- → Faster and efficient queries (given you pick good sharding strategy)
- → eliminate single pt of failure

CONS

- → outage possible as it is distributed
  - sol^n: master-slave design
    - handles writes
    - handles reads
- → Fixed number of shards · Need rebalancing / resharding
  - sol^n: consistent hashing
- → Skewed shards — Better sharding f^n approach
- → joins across shards are costly

IMPROVEMENTS

→ SHARD + INDEXING

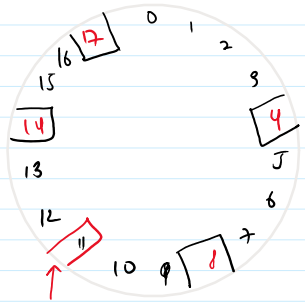
Ex :- shard on CITY + index on AGE

→ here first shard on city + Internally each shard is indexed based on age

Query - give me all persons in DELHI with age between 20-40

CONSISTENT HASHING

* Normal hashing - lets say you have (N) servers
  - (h(data) %. N)
    - h: hash f^n
    - N: no of servers
  - → you cant change this
  - if done needs rehashing everything
* Consistent Hashing
  - → Consider your slots are in circular form
  - → x → hash f(n) → (0 - 360°) example
  - → Each result of hash fn will lie on this circle
  - → some of these slots will be servers also



HOW IT WORKS

- → data → h f^n ( ) → slot-no = (3)
- → Move clockwise and see next server to (3)
- → (3) gets stored in (4) ie (4)

New server gets added

- → (11) is now made a server
- → New indexing doesnt change (9,10 will go to (11) instead of (14))
- Rebalancing is minimal
  - → only (9) (10) which was prev sharded to (14) need to be rendered to (11) NOW

* LOAD Factor = 1/N (ideal case) equal load to each server
  - → But this isnt possible when you have sparse

sol^n: Virtual servers

- → Add more server slots which brings load factor to 1/N
- → Internally a single sever handles (2+ slots)
  - Ex:- (8, 11) might be handled by a single physical server

ACID

→ Maintained in Relational DBs
- → Good at vert scale (Bad at vertical)
- → Fixed schema
- → Generally not distributed

↳ Maintained in **Relational DBs** → Good at vert scale (Bad at vertical)
→ Fixed schema
→ Generally not distributed

Atomicity

→ A txn can either happen completely or not happen at all

Ex:- Bank amt transfer

① Deduct frm ur acc → ② Add to the desc acnt

Now this shud happen completely. Ex lets say it got deducted from accnt but then failed

→ In such case revert back everything

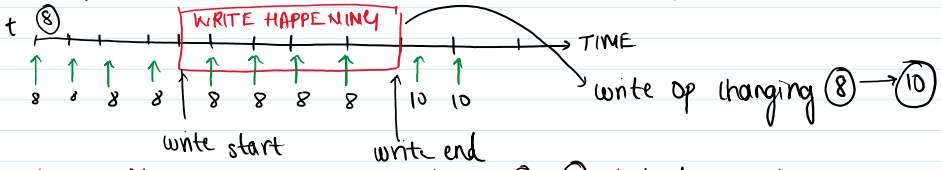
everything or nothing approach

consistency → multiple reads at same time should give same results

Why easy? since its not distributed no issue of out-of-sync replicas

Isolation

→ Every transactn happens independently without caring abt other parallel transactions



→ Here write op^n was changing val frm 8 → 10 but during the process any read doesn't know or care abt this and still reads as 8

→ Only after write operation is completed and data is updated, Read returns 10

Durability Every txn is logged and data is being persisted

BASE

↳ shown in NOSQL Databases → No fixed schema
→ vertically scalable
→ Distributed (data stored across machines)

Basically Available

→ Since data is distributed across servers it can handle outages

→ if one server crashes another replica/machine can give us the data

Soft states

↳ States/values can auto change without txn or user queries

How? SYNC across replicas

↳ since data is stored across machines/replicas it may happen that data copies keep getting updated

dirty data NOT IN SYNC → Latest in-sync data

Eventually Consistent

↳ Not immediately consistent → ACID ones are immediate consistent

→ 2 simultaneous reads may give diffent Results sometimes

why? sync might have been happening b/w 2 reads

∴ Txns are not always consistent due to distributed nature

BUT will become consistent after some time (once sync/updates are completed)

Rate Limiters

06 June 2024 22:10

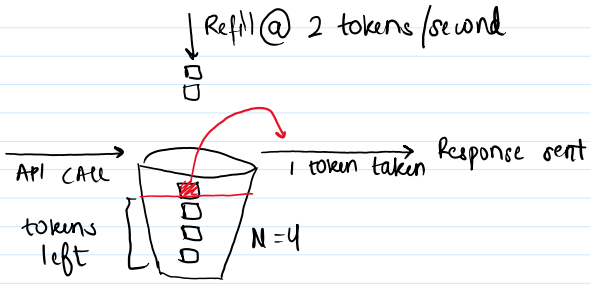
RATE LIMITERS → server side
RATE LIMITERS → client side

TYPES →
- Leaky bucket
- Token Bucket
- Fixed window
- Sliding window
- Counting sliding window

TOKEN BUCKET

→ You have (N) tokens in a Bucket (consider this as a currency)

→ whenever a request comes
① Check if bucket has token
- (NO) → send (429) response code
- (Yes) → Take one token from the bucket



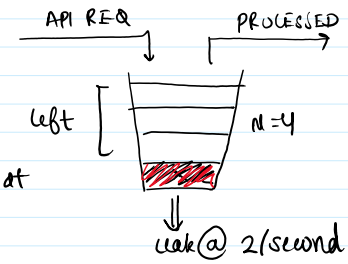
⚠ Refiller :→ Refill the tokens in the bucket at certain rate

refill_rate ≡ TPS/speed of your api

* Configurations → ① N = Bucket size
② Refill rate

LEAKY BUCKET

→ You have a bucket of size (N)
→ A request comes in, if bucket
- (Full) → 429 code
- (else) → Add 1 token to the bucket

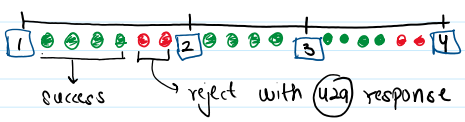


→ (Leak rate) → You keep on removing tokens at this given rate
Similar to refill rate in token bucket

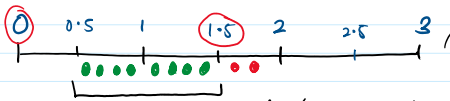
* IDEAL CASE → Rate of API hits ≤ LEAK RATE

FIXED WINDOW

→ You maintain certain windows (INTERVALS) and each window can only contain (N) items
→ Ex:- Window of 1 min each → N = 4



CON → Doesnt always guarantee rate limit
Ex:- N = 4/sec



→ valid since our windows are 0 – 1 – 2 – 3
But truly not limiting at Boundaries

SLIDING LOG WINDOW

→ Instead of fixed window you use timestamps to dynamically calculate it
→ Rate = 4 req/sec

Ex
→ Request comes at 1.5, you count no of req blw 0.5 ↔ 1.5
if num < 4

Ex
→ Request comes at 1:5, you count no of rq blw 0:5 ↔ 1:5
if num < 4
→ Insert this log (timestamp + req id)

* Ensures proper dynamic rate limiting

CONS
→ you store each request as one entry
∴ Space needed = (N)
→ lets say your Rate is 1000 rps ⇒ you will store 1000 log entries

SOLUTION

SLIDING COUNTER WINDOW

→ Instead of pushing each request as log item maintain a counter
→ Rate = 1000 rps

ts	counter
0:1	1
0:2	100
0:3	200
⋮	⋮
0:9	400

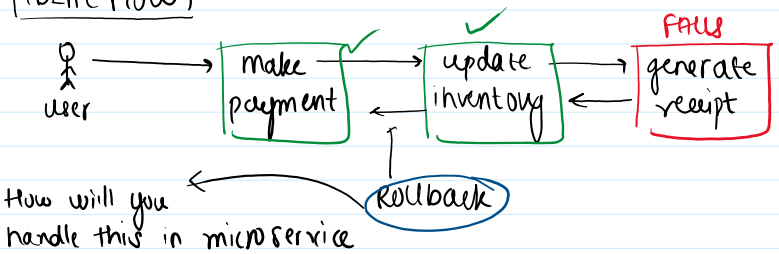
→ Here instead of 1000 logs you will only have 10 logs
→ space = 1000 → 10
→ No of ts steps

DISTRIBUTED TRANSACTIONS

PREMISE → You have 3 services

- PAYMENT → make payment
- INVENTORY → decrease purchased items
- RECEIPTS → generate receipt

IDEAL FLOW



How will you handle this in microservice

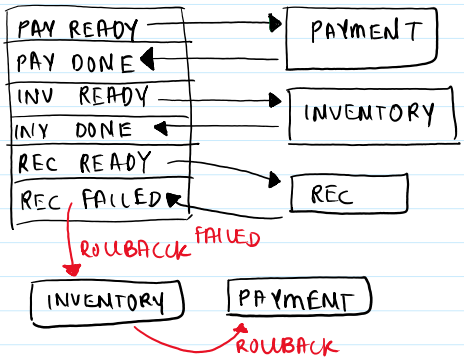
→ ① 2PC [2/3 phase commits] BLOCKING
② 3PC
③ SAGA (ASYNC)

SAGA → Sequence of Local transactions

* Asynchronous and non-blocking but complex to implement

→ Queues are used for Inter Process Communication

STEPS

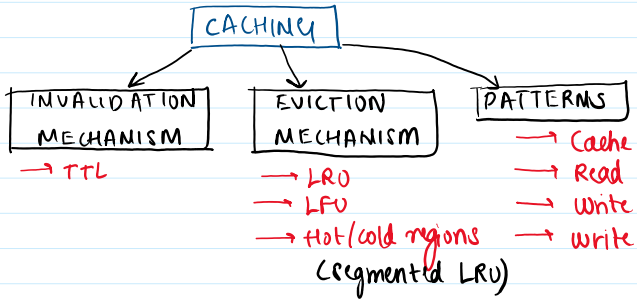


Implementation

→ 3 microservices
→ (3*2) Queues [1 success, 1 failure] one b/w each μservice
→ 2 Queues
① Handles all success
② Handles all failures
(all μservice listen to this OR listen to a specified partition)

Caching

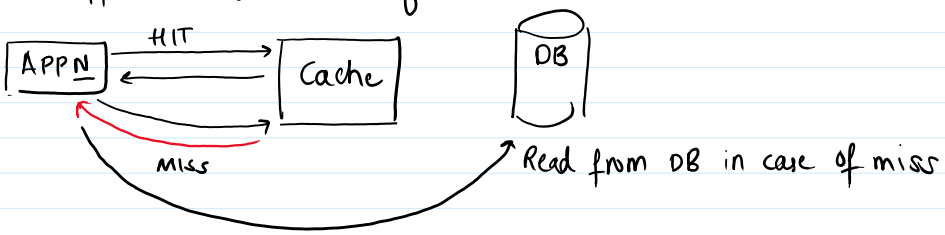
07 June 2024 23:04



PATTERNS

CACHE ASIDE

→ Cache cant communicate directly with DB
Application does that for it



→ Initially data is loaded into the cache by appn
On every cache miss, the appn directly fetches data from DB
→ On new data, it will always be MISS
↳ until you update cache

PROS

* Good for HEAVY-READ appns like dashboards
* If cache fails → you can directly hit the DB each time down
* Since data stored in cache and DB is different, you can have diferent schemas

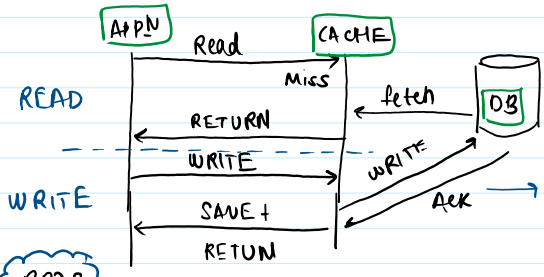
CONS

* New data will be Miss
* Inconsistency in data if it gets updated in DB

READ THROUGH

→ Cache can communicate directly with DB Appn only accesses Cache
→ READ MISS : Cache itself updates value from DB and then returns
↳ New val, update

→ Writing new data :- you will write to cache + cache will write to DB
→ if this was update invalidate cache which will be updated on next miss



* DB schema ≡ Cache schema

WRITE are SYNCHRONOUS

PROS

→ Great for heavy reads why? (On write cache writes to DB, waits for Ack saves it in cache and then sends it back)
→ You dont worry abt Miss and fetch from DB

CONS

→ Cache cant fail
→ Cache miss for new data (PRE-HEATING can solve)
→ Cache schema ≡ DB schema

→ Cache miss for new data (PRE-HEATING can solve)
→ Cache schema ≡ DB schema
→ Data inconsistency

WRITE THROUGH → used with READ THROUGH (100% CONSISTENT)

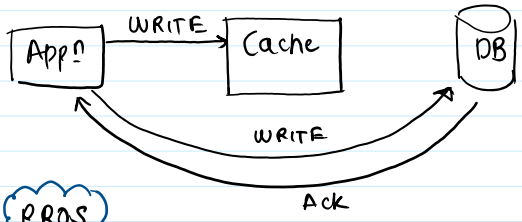
→ Here for writes you directly write to DB and invalidate cache (if updated)
→ Next hit will be cache miss and Cache will fetch from DB
(saves latency of app ⟷ cache ⟷ DB for write operations)

WRITE AROUND → You ONLY WRITE TO DB and forget
* when you do a get next time; CACHE miss and cache will fetch from DB

WRITE BACK → SYNCHRONOUS — same as write through (BUT cache updates DB → Not app itself)
→ ASYNCHRONOUS — NOT 100% consistent

SYNCHRONOUS

→ App itself writes to both DB and cache whenever a write hapens



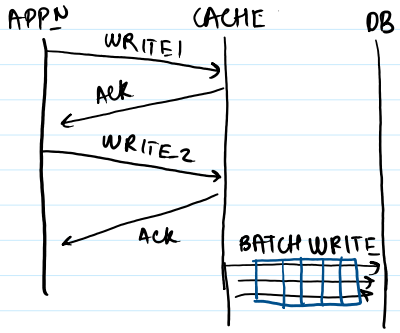
① write to DB
② wait for ACK → use 2PC
③ write to DB

PROS
→ Very high hits and low misses
→ Very rare chance of inconsistency
→ you always write to (DB + Cache) each time No lag

CONS
→ High latency due to 2 writes and ACK wait
→ Need 2PC to make sure DB update is ACID

ASYNCHRONOUS

→ App only writes to cache and then forgets
→ Cache then asynchronously writes data into the database independently
→ done via QUEUES
→ you can publish 1 event for each write
OR Batch (n) writes and at once write to DB



PROS
→ WRITE HEAVY applications
→ Lesser write latency and lower inconsistency
→ Better performance

CONS
→ Sync issues if you immediately read again before DB write is done

Best with READ THROUGH CACHE
why? → when write hapens u just write to cache and forget
→ Now you can directly read that data from cache (even though its still not in DB)

GLOBALLY DISTRIBUTED CACHE

U: → Now you can directly read that data from cache (even though its still not in DB)

GLOBALLY DISTRIBUTED CACHE

① There can be (N) different cache servers distrubed across clusters
↳ A cache gateway redirects requests to appropriate servers
ⓐ Default uses **consistent hashing** for allocation
ⓑ Geographically based - route/store in nearest store
ⓒ Identifier based – Ex: Based on country_id group and store in same server
or multiple countries INTO 1 continent server

REDUNDANCY

* use QUORUM u need consensous for both READ + WRITES by setting values of (R), (W)
→ you need values read from atleast (R) copies
→ you need to write data to atleast (W) copies/machines

(R) → Read consensus frm (R) nodes
(W) → write consensus frm (W) node
(N) → Replication factor

R + W ≤ N
EVENTUALY CONSISTENT

R + W > N
STRONGLY CONSISTENT

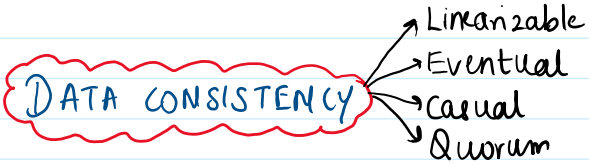
W+H?
→ R+W ≤ N : assume N = 3
* R = 1, W = 2
WRITE : u write to 2 copies and leave third
READ : needs only 1 copy
↳ if it reads from 3rd then inconsistent
* R = 2 : W = 1
→ WRITE : to any 1 assume (C1)
→ READ : (C1)(C2) valid
(C2)(C3) INCONSISTENT
(C3)(C1) valid

R + W > N
Ensures that there is atleast 1 overlap blw write and read consensus

READ (R) C1 C2 C3 WRITE (W)

Isolation Levels

08 June 2024 14:54



LINEARIZABLE

→ single thread sequential execution of Queries
→ No chance of any mismatches
→ PERFECT consistency

EVENTUAL

→ DB might not be always consistent BUT eventually consistent
→ multiple Queries/read/writes can happen of multiple threads/servers
  ↳ parallel process so is faster
* But sync takes some time after transactions

CASUAL

→ Allows parallel transactions BUT operations are linear/serialize in single transaction
→ Consistent within transaction
→ Better than casual but not as good as Linear

CONS: Aggregates/joins are inconsistent

QUORUM

→ Configurable consistency vs performance

R + W > N	Highly consistent
R + W ≤ N	Eventually consistent

TRANSACTION ISOLATION LEVELS

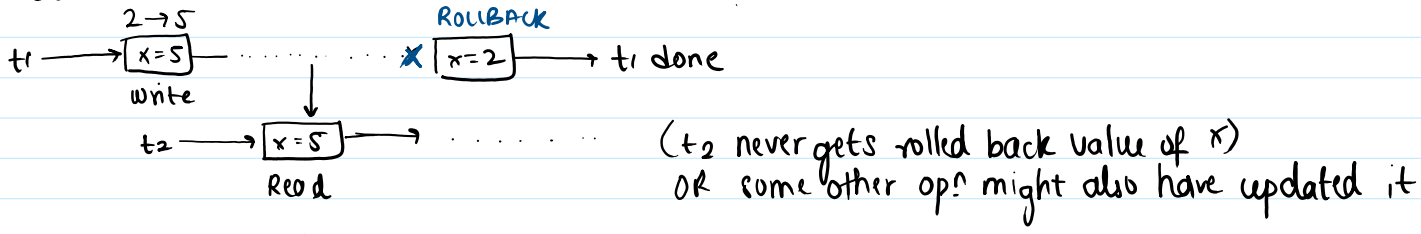
A transaction has three operations

BEGIN TXN → COMMIT TXN → ROLLBACK

- BEGIN TXN: Start ops
- COMMIT TXN: txn done and update res to DB
- ROLLBACK: Rollback commit in case of errors/failure

READ UNCOMMITED

→ Each txn directly writes to DB ≡ No concept of COMMITS (No data replication)
→ Causes DIRTY READS



READ COMMITTED

read a ... or some other op: might also have updated it

READ COMMITTED

→ Txn can only read values which are committed by other txns (↳ it wont roll back later)
→ slower since you have commit step and dont directly update DB

* in both read commited/uncomited you cant ensure REPEATABLE READS
→ if you have 2 reads of same var in the txn you can get 2 different values (someone else must have updated it)

REPEATABLE READS = SNAPSHOT ISOLATION

→ when any row is read by a transaction its LOCKED until that transaction ends
→ other txns which update this locked value can locally commit and keep it as new version and update in DB once lock is released

SERIALIZABLE

→ All operations in txn are executed sequentially in order
→ if 2 txns dont have any common ops then they can run in parallel
→ use serialized locks

common ops: Both writes and reads

least isolation most efficient — Least efficiency most isolation

READ UNCOMMITTED	READ COMMITTED	REPEATABLE READS	SERIALIZABLE

← EFFICIENCY →

→ ISOLATION →