Agenda

SQL databases ✓
Normalization ✓
ACID ✓
Sharding
SQL vs NoSQL
Key-Value
Document
Column
When to choose what

more to next class

NoSQL ⇒ Say NO to SQL → bad ✗

Not Structured Query Language ✗

Not only SQL ✓

↳ if SQL does not satisfy your needs

↳ data is just too large for SQL to handle

The good parts of SQL

↳ Relational databases

→ tables
- ↳ columns → schema
- ↳ rows → actual data
- form relationships
  - ↳ foreign key constraints

} perform query across tables (join)

→ Normalization → possible & easy

↳ removing redundancy

↳ achieves data consistency across tables

Addresses

user_id	pin-code	street-address	city	state
1	1000	———	Bombay Mumbai	Maharashtra
2	7000	———	Bangalore	Karnataka.
3	1009	———	Bombay Mumbai	Maharashtra
4	1009	Powai	Mumbai	Maharashtra.

duplicated data ?

↳ wasted space

↳ anomalies (insert, update, delete)

↳ either update all values in case of change

↳ or deal with inconsistent data

↳ deleting a row → unintended / side effects

Scaler

student_id	student_name	course_id	course_name	course_length
1	Mana	5	HLD	1.5 months
2	Shaw	5	HLD	1.5
3	Rajeev	6	Data Engg.	2 months

Rajeev → only student enrolled in Data Engg.

normalize

normalization → process of de-duplicating data

students

id	name
1	Manav
2	Shaan
3	Rajeev

courses

id	name	duration
5	HLD	1.5 mo
6	DotA	2 mo

student_courses / enrollments

FK students.id FK courses.id

student_id	course_id
1	5
2	5
3	6

→ ACID guarantees → out of the box in SQL

- Atomicity → transactions are atomic (un-dividable) — either all or none → no in-between
- Consistency → invariants are maintained before & after transaction
- Isolation → transactions should NOT interfere — no dirty reads
- Durability → completed transactions are persisted even in case of failures (reboot/crash)

Rohit

~~1000Rs~~
100Rs

withdraw 900Rs → Send → Neeraj

(1) check if rohit.balance >= 900
(2) rohit.balance -= 900
(3) neeraj.balance += 900
(4) neeraj.notify()

rollback ↑

fail

Before transaction

Rohit = 1000
Neeraj = 1000

Total = 2000

magically vanished ↓

Total = 1100

After (failed) transaction

Rohit = 100
Neeraj = 1000

in-consistent

Rohit → Neeraj
900Rs

Rohit >= 900 ✓ 9:41
Rohit -= 900 9:42
Neeraj += 900 9:43
Notify

Audible ← charge 500Rs ← Rohit

dirty ← Rohit >= 500 ✓ 9:41
Rohit -= 500 9:42

Rohit = 1000Rs

1 → Neeraj 90

2 → Aadith 500

→ **Joins / Complex Queries**

∵ tables are related → slice & dice & join data

→ **Strongly defined / rigid schema**

each column has a datatype

each row has a fixed set of columns

Shortcomings of SQL databases

→ rigid schema → if data is highly flexible

E-commerce → 100,000+ product types

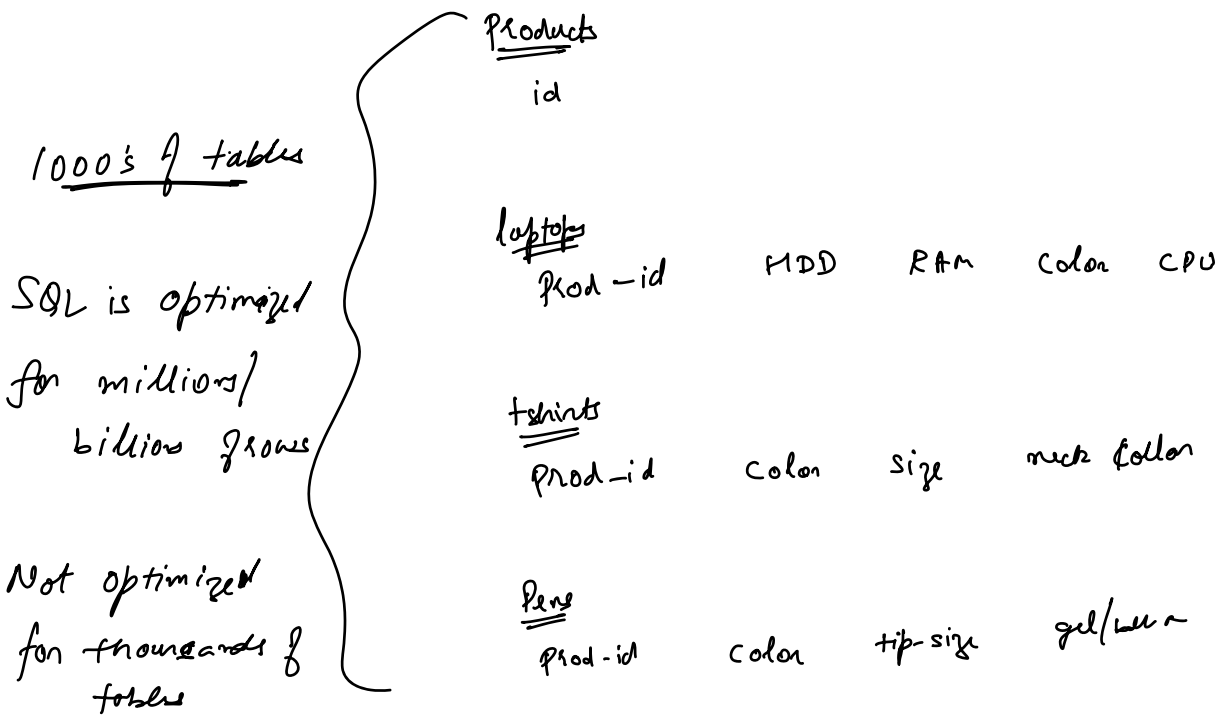
Products

id	name	RAM	HDD	brand	Size	neck collar	color

each product has a different set of attributes.

① add all the possible types to the column list
↳ each row will have thousands of null values
↳ Space
↳ ambiguity
↳ no data?
↳ is data but we don't know it?
↳ comparisons

② for each product type, I can have a table



If there is a lot of data → single machine cannot handle the load

- ↳ amount of data
- ↳ # reads → multiple replicas
- ↳ # writes

Sharding → only solution

Sharding nullifies almost all SQL advantages

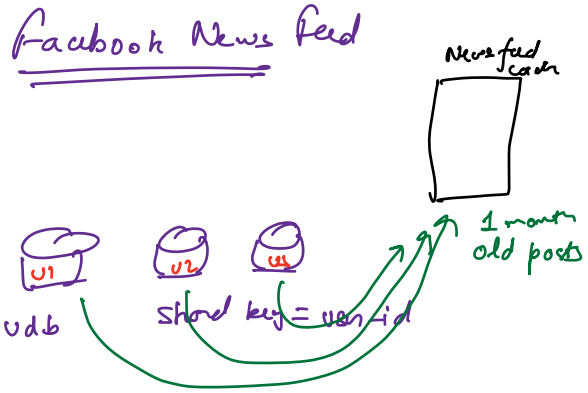
- ↳ within shard → all good
- ↳ cross-shard
  - ↳ joins → difficult, expensive
  - ↳ relations → not supported
  - ↳ ACID → no longer true across shards

Most popular Relational DBs

Postgres / SQLite

No official support for sharding.

Facebook News feed



NoSQL databases

- key-value
- document
- Column database
- graph → much smaller market share
  - emerging technology

Survival DB

---

choosing a good sharding key (NoSQL)

↳ includes SQL

chat/messenger

user_id → all message of a user goes a shard

Omkar → Anurag : Hello (m1)

Q: In which shard should we store (m1)

Context: Freq Queries → given a user/conversation fetch recent messages

① sender_id — M1 → shard for Omkar

fetching recent messages for Anurag → difficult

all sent by me ✓

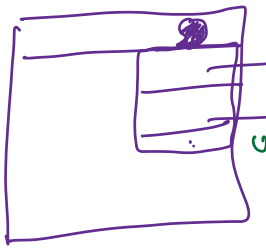
all sent to me ✗

(2) reciever_id → symmetrically same issue

all sent by me ✗
all sent to me ✓

(3) conversation_id (sender_id, reciever_id)

fetch all messages of this conversation ✓
find all I sent ✗
find all I recieved ✗

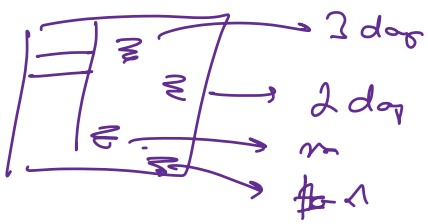


: Hello
: Hi

→ all messages that I have recieved recently

→ all messages associated with conversation b/w Anurag & me

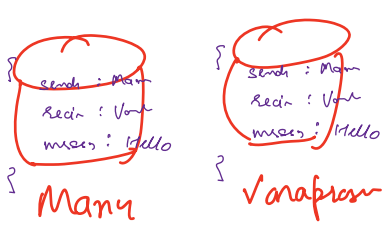
(4) timestamp



(5) geo-location

Solution shard data still by user_id

but now any message that is sent is stored in both
- ↳ sender's shard
- ↳ reciever's shard



{
sender : Mam
Recir : Vom
mseg : Hello
}

---

① Banking System

Context
① users are from multiple cities
② Most frequent queries
- → Balance Query (account-id)
- → Fetch the transaction history (account-id)
- → List all my accounts (user-id)
- → Make transaction (sending-account-id, reciem-id, amount)

① location → Make transaction query is difficult
✗ bad (not really ∵ hit two shards)

→ users can change cities
↳ copy transaction

→ some cities will have significantly (1000x) more activity than other cities

② user_id → all queries except MakeTransaction
good Only need 1 shard.

→ MakeTransaction
↳ save data in both sender & receiver shard.

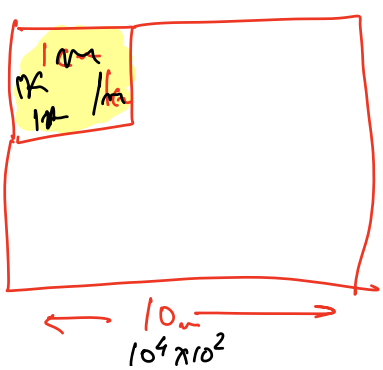
Bangalore → 50,000 rides

③ Ride sharing app

Context → find all nearby cabs which are available
(driver_id, location, available)

driver_id → nearby drivers will all belong to different shards

city_id → find nearby cabs
↳ drivers can change location
Delhi → Ghaziabad
move fast



≈ 2×10^2 cabs

↳ search for intra-city rides ✗
✗ drivers must be in source shard

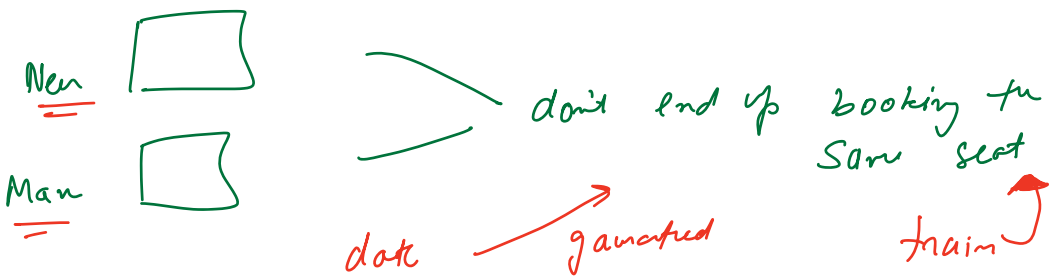
(8) IRCTC

Context → Tatkal ticket booking

(user_id, train_id, date, source, dest, class)

→ 27,000 bookings per minute

≈ 400 per second.



date → shard for today / tomorrow → overloaded

train_id → solves our issue completely.

1 million $

shoes only ✗ not needed

addition Constant

Summary of choosing sharding key

→ load should be evenly distributed across shards

→ Most frequent operation / queries / usecases should be efficient

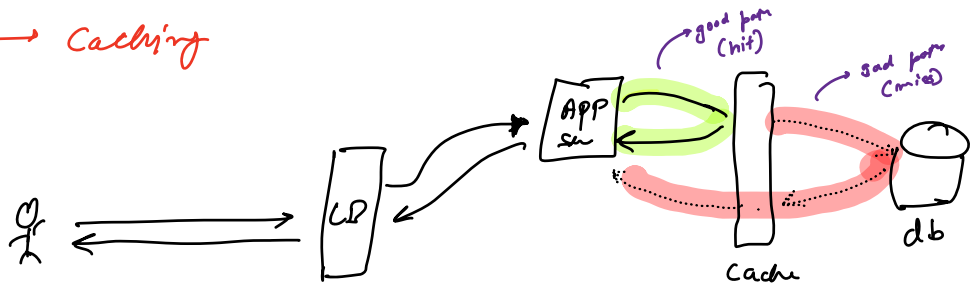
→ Most freq writes → update least amount of shards

→ better data consistency

→ Minimize (can't always perfectly avoid) the data duplicacy.

Doubts

Caching → Caching



+ve caching
-ve caching
} → response

pragy Solns.com.

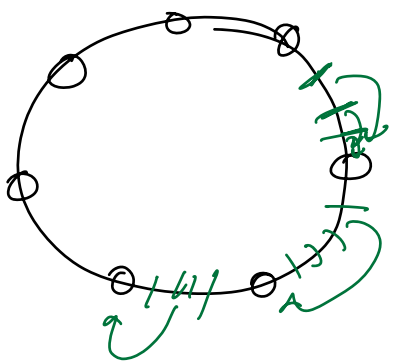
→ is email rohan@gmail.com registered?

Rohan was regist → +ve result from db ✓ definetly cache it

Rohan was not registered → -ve data

Consistent Hashing

train_id (11,000 trains)



4MB cache of local machine

↳ reference / catalog data → mostly static

IRCTC

Users

id name age ——

↳ 1 m users

Stations

id name #platform

Sched ——

Trains

id name #seats Sch——

11,000

7000

→ bookings

id usr_id train_id dot

src dest class seat_no

~ 10m per day

no sharding

Cache layer

group chats → Slack channel 100,000 users

msg → sender + receiver

100,000 copies of msg? X

users

id name avatar - -

user-groups

id group-

user-user-chat (DMs)

sender-id receiver-id msg timestamp

→ shards by user-id

shards by user id

msg → copied → sender receiver

group-chat

id sender-id group-id msg timestamp

shard on group-id

group-chat-read

group-id msg-id user id

10 200 1

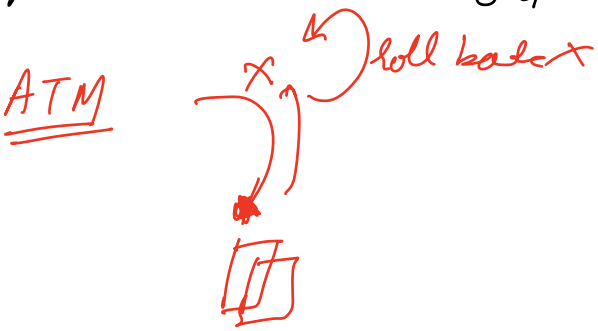
websockets

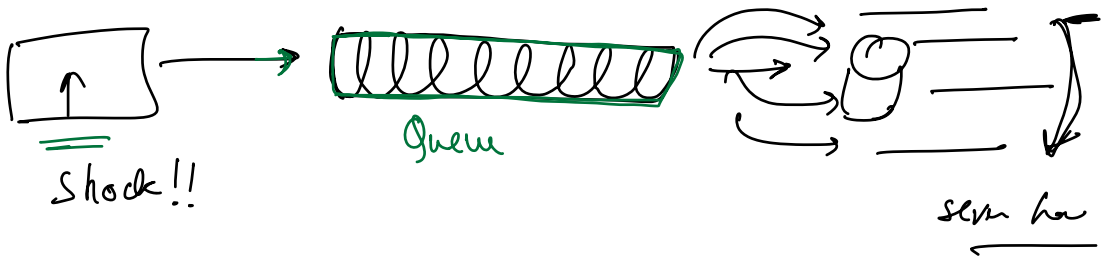
CRDTs →
Blockchain →

Some non banking use cases

Online

ATM





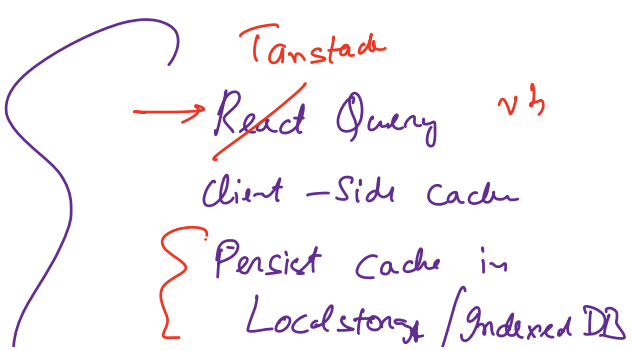
---

users = 1,000 → 1500 → split → water

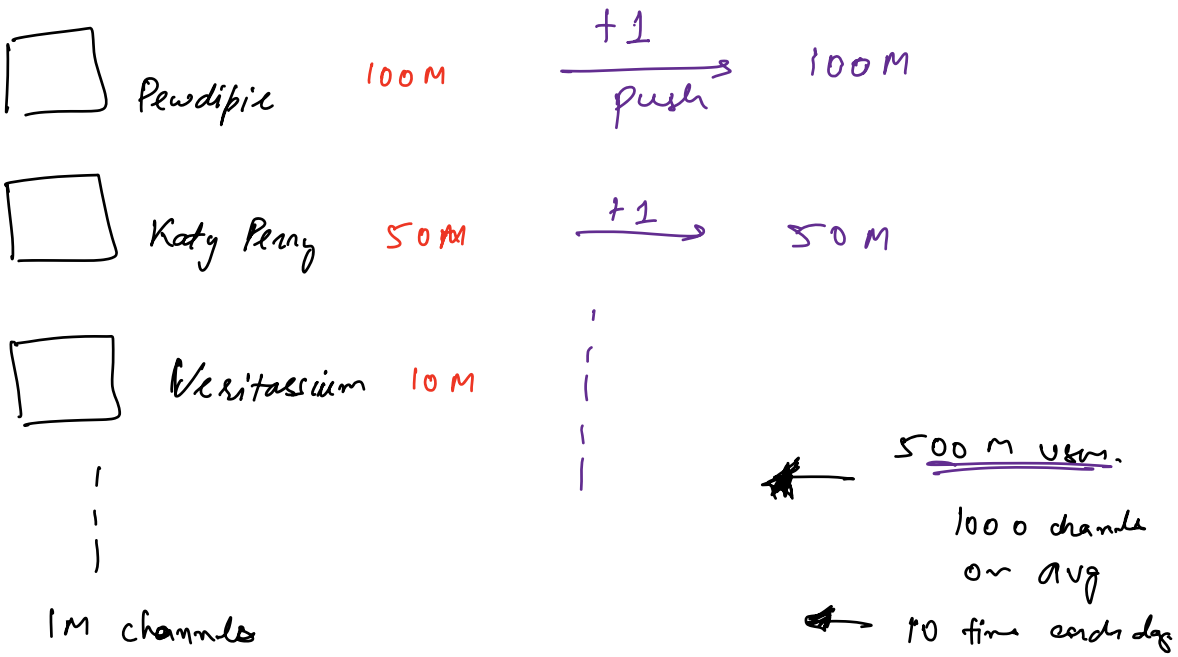
Udemy's Scale / Unacademy

100,000 →  300,000 burst

10M users

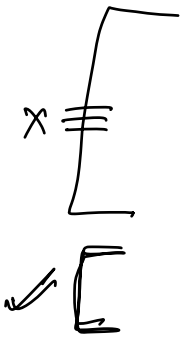


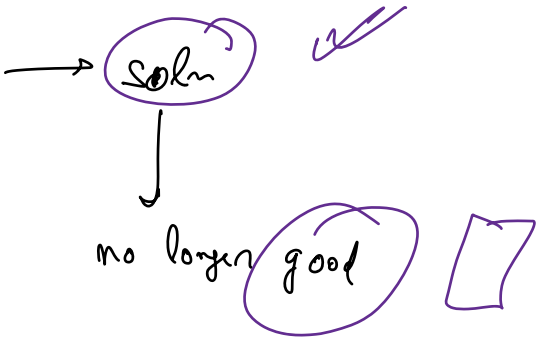
→ SWR
→ normalize each in GraphQL



events in day = (# videos per day * # users subscribed to that channel)

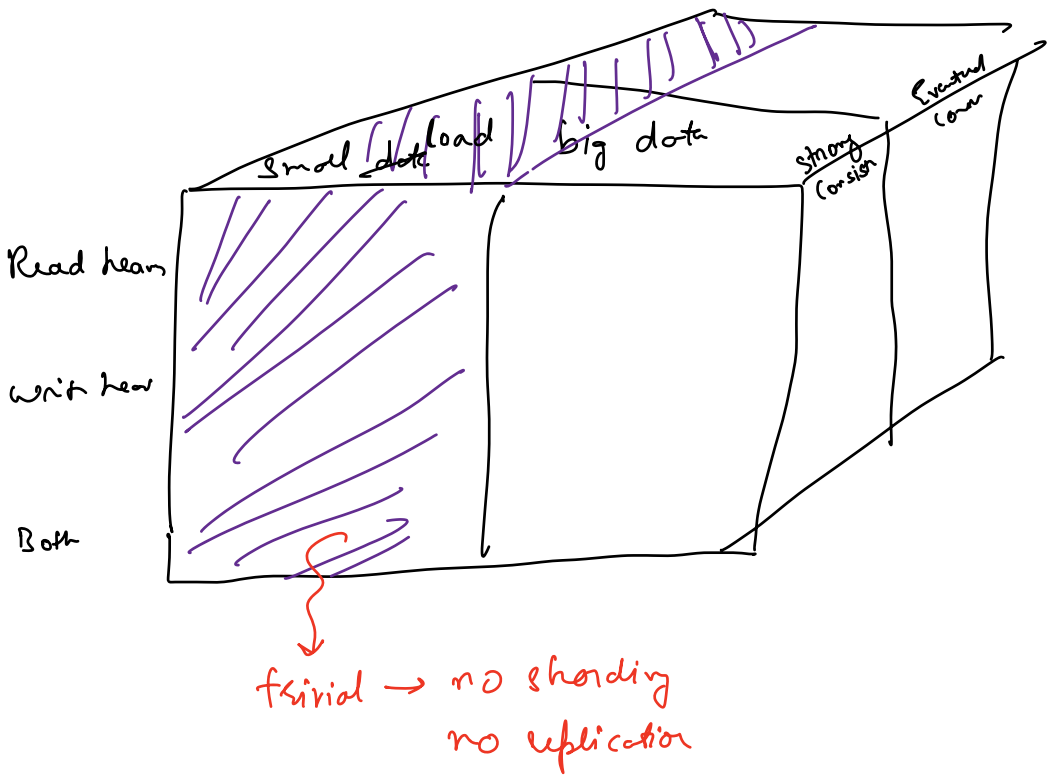
events per day = 500 M * 1000 * 10 >>





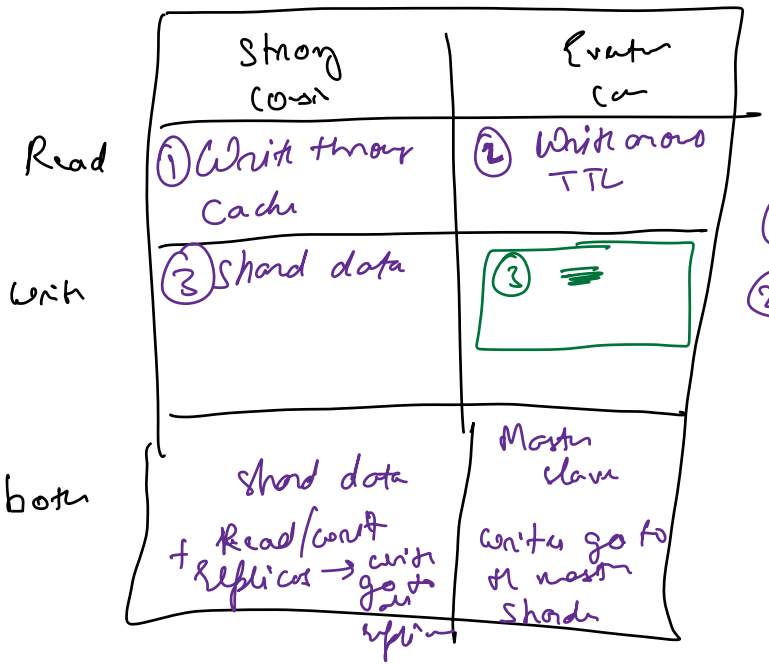
① anticipate & pre-emptively plan for it

② deal with it → salvage as much as possible



trivial → no sharding
no replication

Big Load / data



① → ② → cache with replicas
any write goes to DB + all cache replicas
No DB replication
No DB sharding.

(?) Ensure that any write queries that require strong consistency fall in a single shard

---

Replica → only helps optimize reads & availability

Sharding → helps optimize both reads & writes but not availability.

→ bad :: leads to inconsistency if not managed

Duplicacy → Redundancy → food ∴ It prevents loss & gives food roads

Redundancy ← duplicacy.