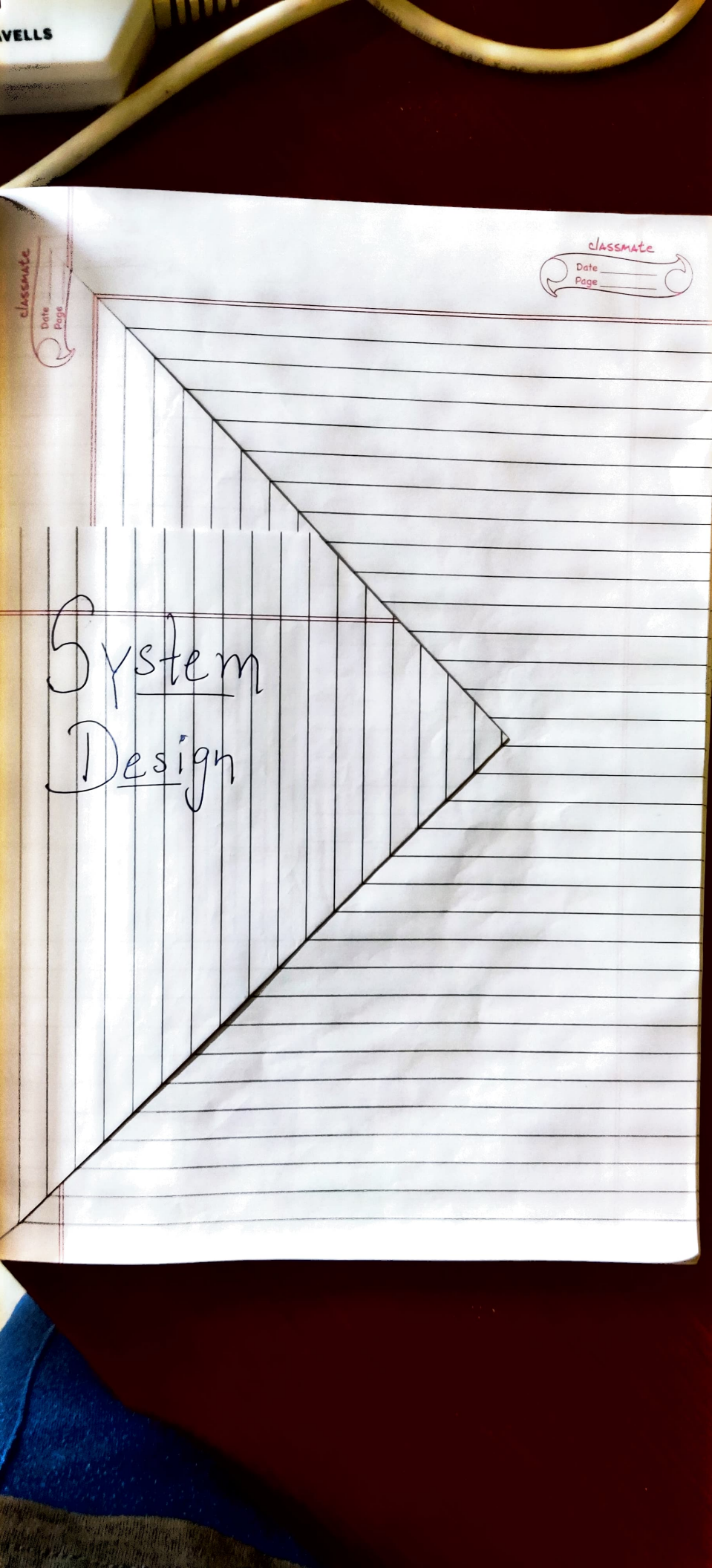
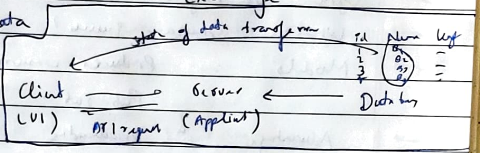

System
Design

REST API:

→ Representational State Transfer (REST)   REST != HTTP

→ guidelines ~~for~~ between client & server to exchange representational status of data



* Guidelines:

i) Client Server

ii) Cacheable

iii) Layered (component knows about the immediate component)

iv) Stateless (server does not know who is requesting)

v) Uniform Interface

vi) Code on Demand

eg REST API implemented over HTTP:

HTTP method — Uniform Resource Identifier (URI) — (Response Data)

i) GET https://api.books.com/mystore/books   (server domain: https://api.books.com)

```
{
  { name: --
    price: --
  }
  { name —
    price —
  }.
}
```

(JSON) — Representation status of data (JSON)

ii) POST ″

Payload:

```
{ id: B5
  name: —
  author —
  price —
}
```

Response:

```
{ status: 200
  id : B5
}
```

iii) PUT <server_domain>/mystore/books/B1

```
{ cost = 90;
}
```

(payload) — Response

iv) DELETE ″ — ″

* Path Parameters : <server_domain> / mystore / book / id

Query Parameter : ″ / ″ / ″ ? limit=20&offset=0

(20 books displayed at a time)

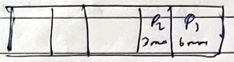
* Security, Authorization & Error Handling :

Message Queues

→ Sync vs Async Communication
→ What are message queues?
→ Models : Producer consumer
Pub-Sub
→ Advantages & Disadvantages
→ Use Cases

Asynchronous : Message Queue : decouple the components (scalability)

* Message Queue (asynchronous) → Producer → Consumer (can be multiple & decouple the load)



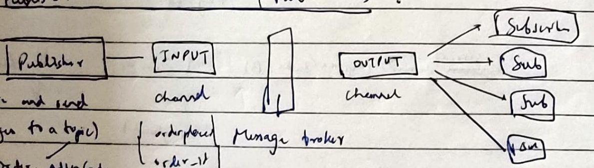
→ messages
→ order (eg FIFO) ~~FIFO~~ (can be ordered or out of order)
→ consumption (one to one) (one message is processed only once)

eg: Invoice generation → order doesn't matter

* Asynchronous Communication : Info is shared independent of time (response not necessary)

Synchronous → instant response (real time) ~~exchange~~

Publisher - Subscriber (Pub - Sub) : (Scalability, decoupling)



Publisher (create and send messages to a topic)

eg: Order, AddToCart, logs

{orderplaced, order_id} Message broker

SMS
EMAIL
Logging
Order processing

Subscriber (create a subscription to a topic to receive messages)

subscribe: SMS, email, order processing

HAVELLS

classmate
Date
Page

Use Cases of Pubsub:

i) Async. Work flow
ii) Decoupling
iii) load balancing (increase the no. of subscribers)
iv) Deferred Processing (processing message at a later point of time (off peak time))
v) Data streaming

Not to be used when: small no. of requests per day
requirement of real time processing

Performance metrics:

i) Throughput : Work done / time taken (No. of API calls served per unit time)
ii) Bandwidth : High resource 'And' High Bandwidth ⇒ High throughput
iii) Response time : Time taken to serve the request
iv) Latency

Performance Metrics of Components:

i) Application: API response time, throughput of API, error occurring, bug / Lag / defect in code
ii) Databases : time to execute queries, No. of queries exec/time (throughput)
iii) Cache : Latency of writing to cache, No. of cache eviction & invalidation, memory of cache instance
iv) Message Queues : production/ consumption rate, fraction of stale/unprocessed messages
v) Workers : time to complete job, resources used in processing
vi) Server instances : Memory / RAM, CPU

Performance metrics tools: dashboard of metrics (GCP, Azure, AWS)

Fault and Failure :

Cause → Effect

→ The oven ~~fire~~ crashed so cake couldn't be baked

fault → failure

→ Faults : Transient fault (short time, hard to detect) Permanent fault (system won't work until fixed, easy to detect)

in distributed systems

Solution : Replication

→ Faults eg : software bug (write tests to detect), network error, memory exceeds, high CPU utilization, etc.

→ Even in case of faults, system must fail gracefully.

Scaling : eg : 1K request / minute → 10K request per minute

→ Vertical scaling : Increasing capacity of resource (eg : adding more chairs to table)

→ Horizontal scaling : " no. of resources (eg : adding multiple tables)

Properties :

→ handle increased load

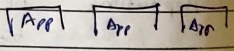
→ not complex

→ performance should not take a hit

Vertical

1K requ / min → 10K requ/m

Horizontal :



30K request / m.

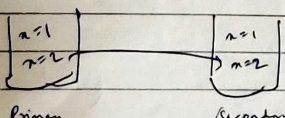
(load balancer required in case of horizontal)

Database Replication :

→ handle faults / failure

→ reduces latency (multiple geographical locations)

→ application performance ↑ (eg multiple reads)



At $t = t_1$, $x = 2$ in Primary

Replication lag : e

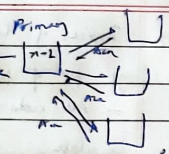
At $t = t_2$, read is performed

if $e < (t_2 - t_1)$ ✓ else inconsistency

HAVELLS

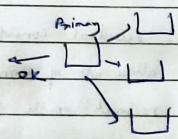
classmate
Date
Page

* Synchronous replication :
→ All replicas updated before host is acknowledged
→ consistent results but takes more time for write.
→ Problem : one of the replica fails ⇒ no acknowledgement



(Read after all write)

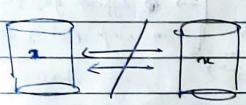
* Asynchronous replication :
→ Host is acknowledged after primary DB is updated.
→ better performance but may have inconsistency (faster writes)



(Semi-synchronous → wait for one acknowledgement for any backups (or some K))

* Snapshot ⇒ store the state of data and roll back in case of failure

CAP (Consistency, Availability, Partitioning) :



(if broken ⇒ partition)

Consistency → both databases agree on same value when the communication link fails, and the system still runs :
Either we can achieve consistency (stop receiving requests on B)
or availability (receive request on both A and B) (not consistent)

→ If we tolerate a partition (system can still run and ensure consistency/availability) → then it is called partition tolerant system

CAP Theorem :
(Brewer's theorem)
→ Having All 3 properties is impossible
→ one has to be sacrificed.
→ Partition is inevitable hence we must ensure partition tolerance
→ Consistency / Availability needs to be chosen according to requirement

Consistency : Any read after a latest write, all nodes should return the latest value
Availability : Every available node should respond in a non-error format to any read request without guarantee of returning latest write
Partition Tolerance : system will respond to all read & write even if communication channel between nodes breaks

→ Degrees of Consistency & Availability :

eg1: one of the nodes stop serving reads, but keeps serving writes.
⇒ All reads will give consistent data and also the system is more available.

eg2: eventual consistency:
allow both of them to read, but after sometime when link comes back, reconcile on the latest value.

— Use cases : Consistency → ATM / flight booking / banking
Availability → youtube likes / analytics data

→ Backup network connections to avoid partition (expensive)

Database Sharding :

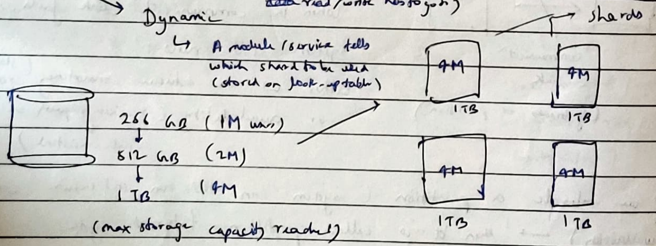
Advantages

- → when maxm storage capacity reached
- → query optimization, reduced latency
- → different geographic location
- → no single point of failure

SHARDS → logical
SHARDS → Physical

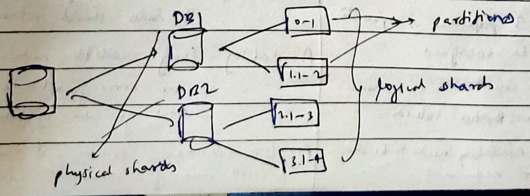
Sharding → Algorithmic → (App: using some function, returns which shard the ~~data~~ read/write has to go) ← say user-id is parameter

Sharding → Dynamic
↳ A module / service tells which shard to be used (stored on look-up table)



Vertical Partitioning → store a set of columns on one DB (partition on basis of columns)

Sharding → Horizontal Partitioning → store a set of rows (all columns)
(same)



classmate
Date
Page

* Key Based Sharding:

→ Based on some shard key (may or may not be the primary key)
Hash (shard key) ⟶ which shard to use
(Algorithmic sharding)

→ Advantages : Equal distribution of data
Disadvantage → when data increases / decreases, no. of shards need to change ⇒ hash funct^n has to be changed
migration of data b/w shards is difficult

→ Shard key should be static (not changing frequently)
(multiple columns can also be chosen)

* Range Based sharding :

user-id	event	date
1	login	18-02-2022
2	add to cart	14-04-2024
3	order	

→ Sharding based on some range
eg: Jan - Feb^t, Mar - Apr^t, - - - -   (6 shards)

→ Algorithmic sharding (in the application code.)

→ Advantage : no hash funct^n so no much migration of data when data/ans grows.

Disadvantage : more data in one shard (eg Jan-Feb)
Hotspot → uneven distribution of data leading to more data in one shard.

→ Use cases → range based queries (<= , >= , etc)

Directory Based Sharding : (dynamic)

User Id	Zone
101	A
102	B
103	C
104	D

Look Up Table

Zone	Shard
A	1
B	2
C	3
D	4

App ⟶ Look Up Table ⟶ DB / Shard
(extra hop required for every R/W)

Advantage :
→ easy to add / remove more shards

Disadvantage :
→ extra lookup
→ fails if look-up table crashes (replicas)

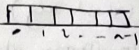
SURGE PROTECTED

classmate
Date
Page

Hashing:

→ String / Objects / Nos ⟶ Integers (~~fixed~~ eg 32 bits)

→ Hash function eg: $f(n) = n \% m$



Hashed value can be used as array index for quick access

→ eg use case → mapping users to servers (UI → Obj)

→ Disadvantages of Normal hashing: (eg $n \% n$)

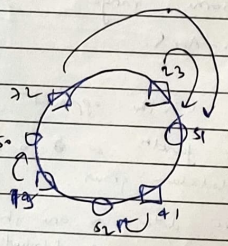
If no. of servers increase / decreases, the hash function has to be changed, and too many keys have to be redistributed.

Sol^n → consistent hashing

Ankur, Amit, Ram, Kumar → hash function → 23, 41, 72, 19 (keys) (0-100) (hash output range)

Key % 4 ⟶ Server (S0, S1, S2, S3) (Normal hashing)

Consistent hashing:



Angle → [0, 360)

Every value to sit on the next server in clockwise / anticlockwise (choose any one) direction

map output value of hash func^n → points on circle

Also map the hash values of server (0-100) to points on circle

Problem: removing a server, adds too much load on the immediate next server

Sol^n : create replicas of servers and map each of them to the circle

HAVELLS

Classmate
Date
Page

Functional & Non-functional requirements

Functional requirements: functionalities provided by system to fulfill user requirements

- eg: Twitter : post tweet, delete tweet, favourite a tweet, follow user
- Outcome: system / components of system
- distribute the responsibilities of functional requirements to APIs, workers, events, messaging (System design diagram, Component & Architecture diagram)

Non functional:

- Latency of posting tweet, availability of system
- Traffic, Users, Availability, throughput, Latency
- Choice of resources → Capacity Estimation
  No. of resource →
- Tech choices, resource utilization, data storages, services or hardwares