Design a Stock Trading Platform like Robinhood

Robinfoud:- Robinfoud is a brokerage and interfaces with external entities that actually manage order filling/cancellation.

Capability: 1) Order Processing = Synchronously places orders & Cancels orders via request/response API.

2) Trade Feed: offers subscribing to a trade feed for symbols. "pushes" data to the client every time a trade occurs, including the symbol, Price per share, number of shares & the orderId.

Background: Financial Markets :-

a) Symbol :- META, AAPL (Uniquely identify a stock)

b) Order :- buy or sell a stock. Can be market order or limit order.

c) Marketorder :- immediate Purchase or sale of stock at Current market Price

d) Limit order :- Purchase or sell a stock at a specified Price.

Functional Req :-

(1) See live Price of stocks.
(2) Manage orders for stock.
(market/limit order, create/cancel order)

out of scope -
- Users can trade outside of market hours.
- User can see orderbook in real time.
- Trade ETFs, options, crypto.

Non Functional Req :-

(1) High Consistency for orders.
(2) Scalable (lots of trades, lots of users)
(3) Low latency (Price updates, orders)
(4) Minimize exchange connections

out of scope -
- System connects to multiple exchanges for stock trading.
- System manages trading fees / calculations.
- System enforces daily limit on trading behaviour.
- System protects against bot usage.

Core Entity :-

(1) User
(2) Symbol
(3) Order

The APIs :-

(1) GET /symbol/:name
Response : Symbol

(2) Post /order
Request : {
position: "buy",
Symbol: "META",
PriceInCents: 52210,
numShares: 10
}
Response : Order.

(3) DELETE /order/:id
Response : {
ok: true
}

(4) GET /orders
Response : Order[]
(Paginated)

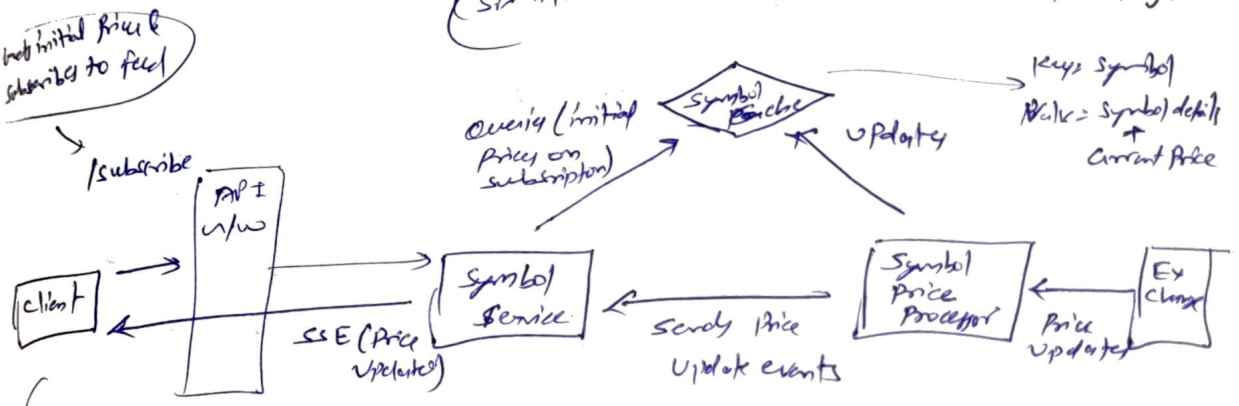
#HLD

(1) User can see live price of stocks.

⇒ Bad soln :- Polling Exchange Directly

⇒ Good soln :- Polling Internal Cache

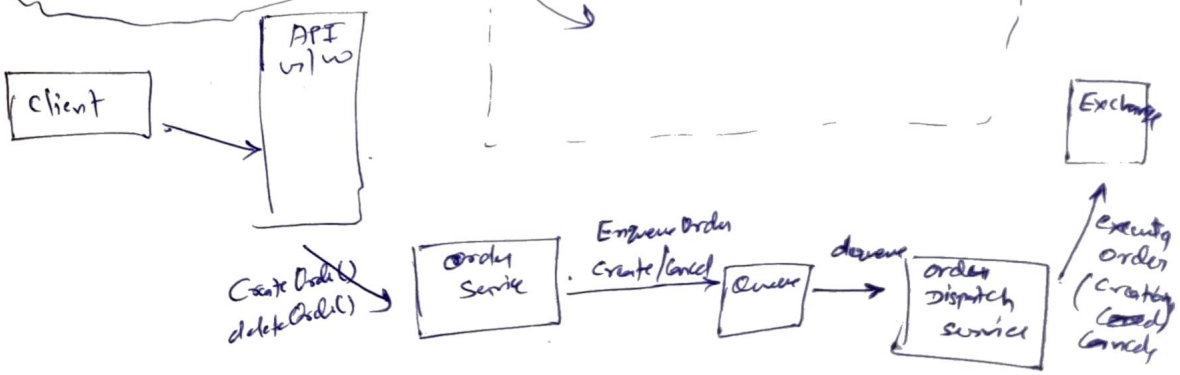
⇒ Great soln :- Server Sent Events (SSE) Price Updates.
(similar to our Facebook Live Comment write up).



(2) Users can manage orders for stocks.

⇒ Bad soln - Send order Directly to exchange.

⇒ Good soln - Send orders to Dispatch Service via Queue.



⇒ Create
* we want SLA (~200ms)
* during very high trading traffic & before the order dispatch service has scaled up, order might take a while to be dispached to exchange, which could violate our goal SLA.

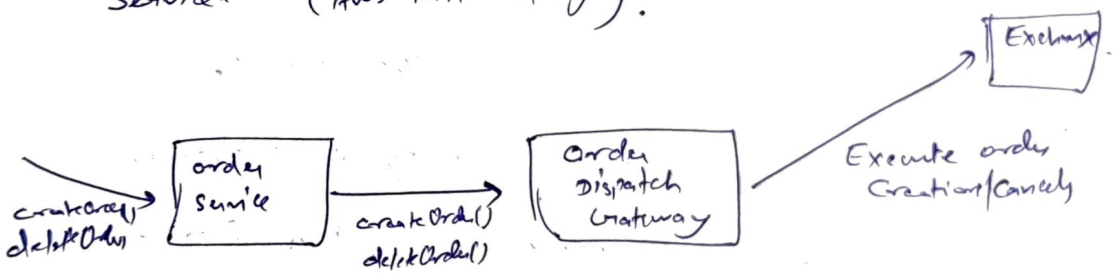
↓
example ##

Imagine a user want to quickly order stocks or quickly cancel an outstanding order. It would be unacceptable for them to be left waiting for our dispatcher to eventually handle their order, or for our service to start more machine up to scale up given increased queue load.

what other option do we have?

⇒ Great solution - Order Gateway

↳ This approach involves sending our orders directly from the order service to an order dispatch gateway. The gateway would enable external internet communication with the exchange via the order service. ("AWS NAT Gateway").



⇒ In order to track orders, we can stand up an order database that is updated when orders are created or cancelled.

Potential Deep Dive

1) How can the system scale up live price updates?

- If many users are subscribing to price updates from several shards, the system will need to ensure that price updates are successfully propagated to users via SSE.

- The main problem we need to solve is :- "How do we route symbol price updates to the symbol service servers connected to users who care about those symbol updates.

⇒ To enable this functionality in a scalable way, we can leverage Redis Pub/Sub to publish/subscribe to price updates. Users can subscribe to price updates via the symbol service & the symbol service can ensure it is subscribed to symbol price updates via Redis for all the symbols the users care about. 🧠

2) How does the system track order updates?

When digging into the order dispatch flow, it's worth clarifying how we'll ensure our order DB is updated as orders are updated on exchange.

* Problem - The trade processor cannot efficiently update the order DB based on a trade, since it can't look up orders using just "externalOrderId", due to the order table being partitioned by "userId".

* Ans - Introduces a key-value store (e.g. RocksDB) mapping "externalOrderId" → (orderId, userId).

* Who populates it & when -

↳ The Order Service will populate this store synchronously after submitting an order to the exchange.

* Why this helps :-

↳ It allows the trade Processor to quickly determine if a trade involves an internal order and efficiently update the correct row in the sharded order table.

(3) How does the system manage order Consistency?

orders are stored in a horizontally partitioned relational database (eg. Postgres) by userId, ensuring that all reads & writes happen on the same shard for a given user.

(A) Order workflow :-

when an order is created, the system:-

- Stores the order with status Pending to ensure the client intent is recorded before external submission.
- Submits the order synchronously to the exchange & receives an externalOrderId.
- writes this externalOrderId into a key value store & updates the order's status to Submitted.
- Reports to the client confirming the order

This ensures that if a system failure occurs at any step, we always have a recoverable trace (a) If submission fails, the order is marked as failed.

(b) if failure occurs after submission but before status update, a background clean-up job can use the clientOrderId to query the exchange, determine the true state of the order, & reconcile it (record the externalOrderId or mark it as failed)

(B) Cancel Workflow –

when an order is canceled :-

- status is first updated to pending Cancel.
- Cancellation is submitted to the exchange.
- The cancellation is recorded in the database.
- The client is notified of success.

Failure at any of these steps is also mitigated through the clean-up job. Forexample, if Cancellation fails or Cancellation via the exchange API, updating local state according.

Key Takeaways –

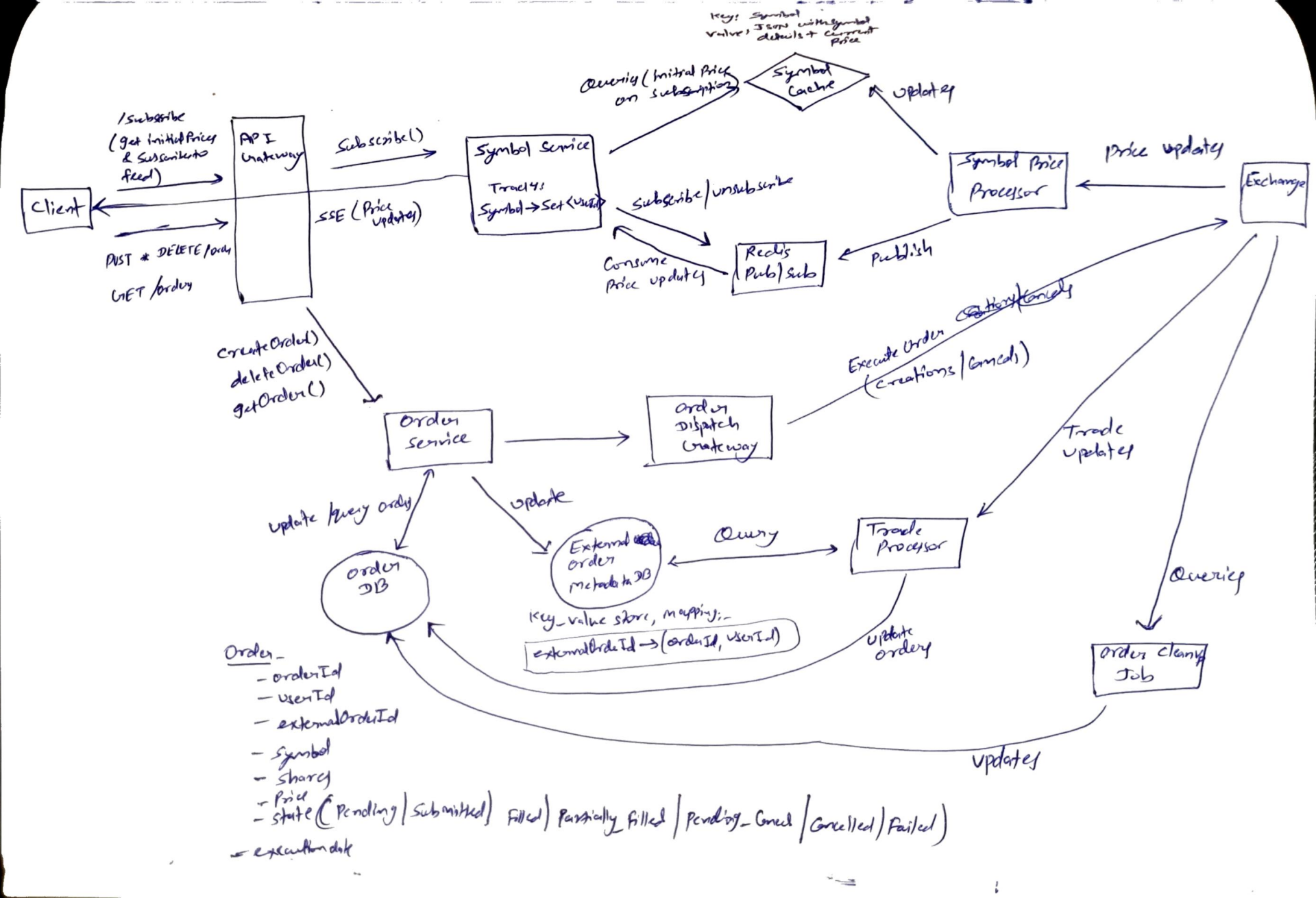
* Issue – Partial failures during order/Cancel flows may cause inconsistencies.
* Solution: write-ahead storage of order intent, use of a key-value store for mapping externalOrderId, and a clean-up job for reconciliation.
* who populates :– The order service populates the key-value store & updates the database during normal execution.

* Benifit — Ensures strong fault-tolerance & eventual Consistency, allowing the system to recover from mid-flight failures without losing track of client activity or exchange state.

HLD design on next page

⇒ some additional deep dives You might Consider

① Excess Price updates
② Limiting exchange correspondence
③ Live order updates
④ Historical price/ Portfolio value data


Key: Symbol
Value: JSON with symbol details + current price
Symbol Cache
Querying (Initial Price on subscription)
Updates
/subscribe (get initial Price & Subscribe to feed)
API Gateway
Subscribe()
Symbol Service
Track: Symbol → Set<UserId>
Client
SSE (Price updates)
POST * DELETE /order
GET /order
Subscribe/unsubscribe
Consume Price updates
Redis Pub/Sub
Publish
Symbol Price Processor
Price updates
Exchange
createOrder()
deleteOrder()
getOrder()
Order Service
Order Dispatch Gateway
Execute Order (creations/Cancel)
Trade updates
Update/query order
Update
Order DB
External order metadata DB
Query
Trade Processor
Queries
key_value store, mapping:-
externalOrderId → (orderId, userId)
Update order
Order cleanup Job
Updates
Order-
- orderId
- userId
- externalOrderId
- Symbol
- Share
- Price
- state (Pending/Submitted/Filled/Partially_filled/Pending_Cancel/Cancelled/Failed)
- execution date

Design an Ad click Aggregator

Functional Req –

1. Users click on ad & get redirected to the advertiser's website.
2. Advertisers can query click metrics over time of 1 minute min granularity.

out of scope

- Ad targeting & serving
- Cross device tracking
- Integration with offline channels

Non functional Req –

(clicks per second)

1. Scalable to support peak 10K CPS.
2. Low latency analytics queries < 1 sec.
3. Fault tolerance
4. Data integrity
5. As realtime as possible
6. Idempotency of ad clicks.

out of scope

- Spam Detection
- Demographic Profiling
- Conversion tracking

Scale

1. 10M ads at any given time
2. 10 K ad clicks per second

HLD :-

1. User can click on ads & be redirected to the target.

⇒ Good soln – Client side redirect

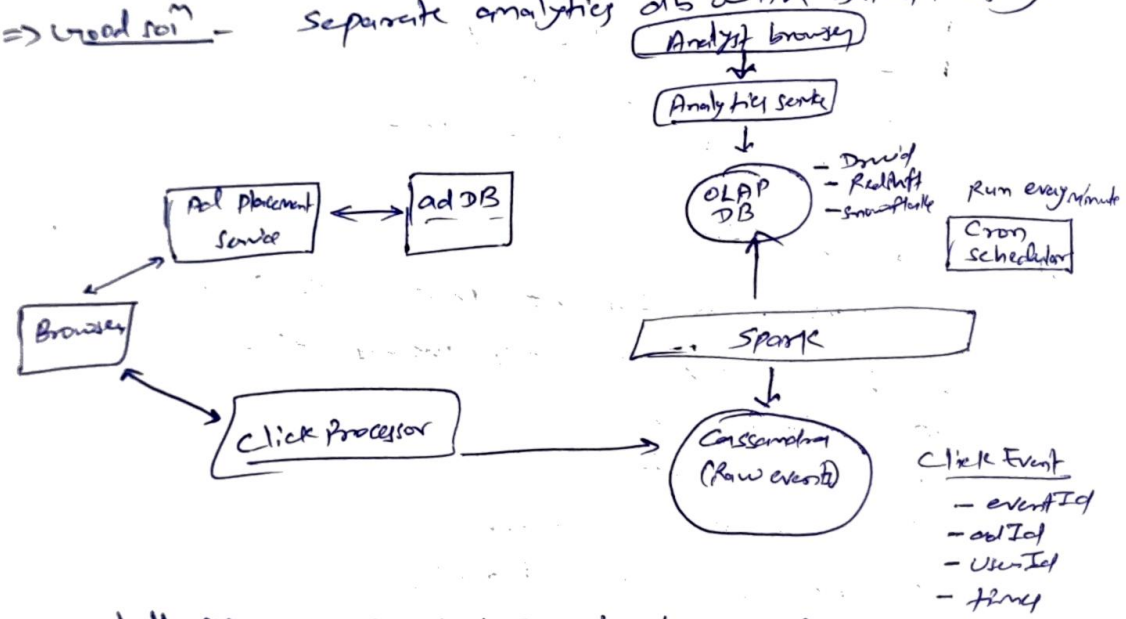
⇒ Great soln – Server side redirect

(2) Advertisers can query ad click metrics over time at 1 minute intervals.

We'll expand on the click processor path that are introduced above by braking some options for how a click is processed & stored.

(a) => Bad soln - store & query from the same Database.

(b) => Good soln - Separate analytics db with batch processing



challenges → * batched processing job introduce a significant delay in the data pipeline.

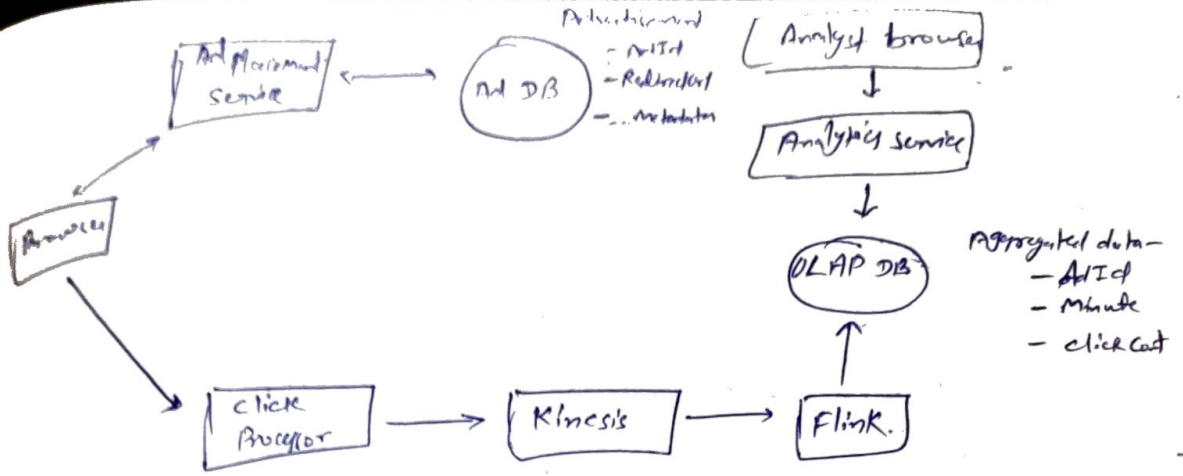
* not able to handle load, if we have a sudden spikes in clicks.

↓

Introduce queue b/w click processor & event store would help with scalability issue, it would not solve the latency problem.

(c) Great soln - Real time Analytics with Stream processing

↓



Now when a click comes in:-

- The click processor service write events to a stream like Kafka or Kinesis.
- A stream processor like Flink or spark Streaming reads the events from the stream & aggregates them in real-time.
- The aggregated data is stored in our OLAP database for querying.
- Advertisers can query the OLAP database to get metrics on their ads in near realtime.

⇒ It's much more realistic for us to decrease the Flink aggregation window than it is for us to increase the spark job frequency given the overhead of running spark jobs.

This means we could aggregate clicks every couple of seconds with this streaming approach to decrease latency, while we'd be hard-pressed to run spark jobs every couple of seconds.

Even better, Flink has flush intervals that can be configured, so we can aggregate on a minute boundary but flush the results every couple of seconds. This way, we get the best of both worlds.

Potential Deep Dives

(1) How can we scale to support 10k clicks per second?

(a) Click Processor service –

* Bottleneck → Processing incoming clicks.
* Sol^{n} → Scale horizontally with multiple instances behind a load balancer, use cloud auto-scaling.

(b) Stream (e.g. Kafka / Kinesis) –

* Bottleneck : Limited throughput per shard (e.g. Kinesis : 1MB/s or 1000 records/s per shard).
* Sol^{n} – Add more shards & shard by AdId to allow parallel processing

(c) Stream Processor (e.g. Apache Flink) –

* Bottleneck : Limited Parallel Processing capability.
* Sol^{n} – Scale Flink horizontally with more tasks; assign one job per shard for aggregations
  we'll have a separate Flink job reading from each shard doing the aggregation for the AdIds in that shard.

(d) OLAP Database –

* Bottleneck : Query performance at scale.
* Solution : Scale horizontally, shard by AdvertiserId for efficient querying across all their ads

(e) Hot shards (skewed traffic Problem) –

* Problem :- Popular ads (e.g. Nike x Lebron) overload specific shards
* Solution :- Add randomness to Partition Keys for hot ads, e.g. Use `AdId:0-N` to distribute load across multiple shards.

(2) How can we ensure that we don't lose any click data?

* we are already using stream like Kafka / Kinesis
  ↓
  distributed, fault-tolerant, highly available

* We can configure a retention period of 7 day, for example, so that if, for some reason, our stream processor goes down, it will come back up & can read the data that it lost from the stream again.

* Skip checkpointing for small windows:- For small aggregation windows (eg, 1min), Flink can reprocess from stream without needing checkpoint recovery.

* Periodic Reconciliation -
  * Store raw events in S3.
  * Run batch jobs to re-aggregate & compare with stream results for accuracy.

⇒ Combining real-time + batch Processing for reliability & Correctness

(3) How can we prevent abuse from users clicking on ads multiple times?

⇒ Bad solⁿ - Add UserId to click event payload. (Con - All user need to login.)

⇒ Great solⁿ - Generate a unique impression id.

(4) How can we ensure that advertisers can query metrics at low latency?

~~In Kafka Processing~~

- In real time stream Processing, data is already aggregated & stored in the OLAP database making the query fast.

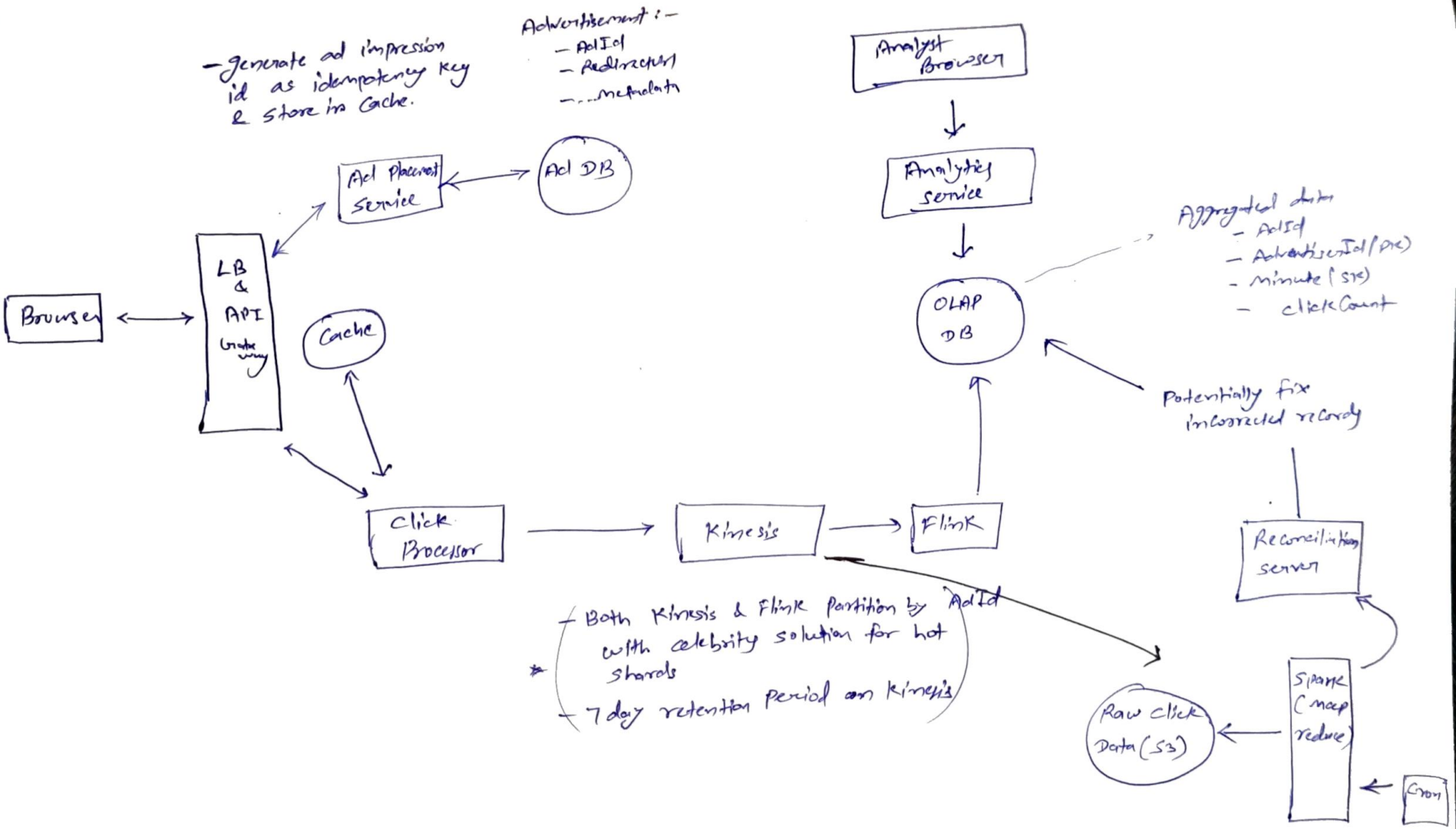
- where this query can still be slow is when we are aggregating over larger time windows, like a days, weeks or even years. In this case, we can pre-aggregate the data in the OLAP database.

This can be done by creating a new table that stores the aggregated data at a higher level of granularity, like daily or weekly.

↓

This can be done via a nightly cronjob that runs a query to aggregate the data & store it in the new table.

when an advertiser queries the data, they can query the pre-aggregated table for higher level of granularity & then drill down to the lower level of granularity if needed.


- generate ad impression id as idempotency key & store in Cache.
Advertisement :-
- AdId
- RedirectUrl
- ...metadata
Analyst Browser
Ad Placement Service
Ad DB
Analytics service
Aggregated data
- AdId
- AdvertiserId (PK)
- minute (SK)
- clickCount
Browser
LB & API Gateway
Cache
OLAP DB
Potentially fix incorrected records
Click Processor
Kinesis
Flink
Reconciliation server
* Both Kinesis & Flink partition by AdId with celebrity solution for hot shards
- 7 day retention period on Kinesis
Raw click Data (S3)
Spark (map reduce)
Cron

Design a Collaborative Document Editor like Google Docs

Functional Req

1. Users can create new documents.
2. Users can edit documents.
3. Users can get real-time updates
4. Users can see each other's cursors

Out of scope

- Assuming simple text editor (no sophisticated document structure)
- Permissions & collaboration levels (eg who has access to document)
- Document history & versioning.

Non-functional Req

1. Documents are eventually consistent.
2. Updates are low latency (100 ms)
3. Millions of users, billions of documents
4. No more than 100 concurrent users
5. Durable documents

Core Entities –

1. Editor : A user editing a document
2. Document : A collection of text managed by an editor.
3. Edit : A change made to the document by an editor.
4. Cursor : The position of editor's cursor in the document.

APIs :-

1. POST /docs
   { title: string } → { docId }

② WS /docs/{docId}

```
SEND {
    type: "insert"
    ...
}

SEND {
    type: "updateCursor",
    position: ....
}

SEND {
    "type": "delete",
    ....
}

RECV {
    type: "update"
    ....
}
```

#HLD#

① Users should be able to create new documents.

API → create new document, take a title & return documentId.

② Multiple Users should be able to edit the same document concurrently.

we have consistency & scaling issues regarding this

↓

will see in deep-dive

Initial problem – Sending Snapshots (wrong approach)

↳ on every edit, entire document is sent to a central document service & stored (eg. in S3).

issues ⟵ Inefficient
↳ Inconsistent → if two user update simultaneously, final document depends on which updates arrive last – causing loss of changes from the other user

Improvement – Sending Edits (Better, but not enough)

↳ How it works ⇒ send only specific edit (INSERT, DELETE)
↳ Benifit ⇒ solve inefficiency
↳ New problem ⇒ order of operations matter

⇒ Collaborative Editing Approaches –

(a) Operational Transformation (OT) –

↳ An approach where each user's edit is transformed based on prior edits to maintain consistency in collaborative editing. For example, if one user inserts text before another's deletion, the delete position is adjusted accordingly.

– OT is fast & memory-efficient but requires a central server for ordering operations & is complex to implement correctly, limiting scalability.

(b) Conflict-Free Replicated Data Types (CRDT) –

↳ CRDTs allow edits to be applied in any order & still reach the same final result.

– They use real numbers to represent text positions & keep deleted characters as hidden "tombstones". This

this avoids the need for a central server & works well offline or in peer-to-peer setups.

- However, CRDTs can use more memory & may be slower due to storing all edits & handling conflicts

~~Our approach~~

⇒ * we'll use OT for our design, like google docs, leveraging a central server for low-memory, text friendly collaboration.

* For large-scale or peer-to-peer scenarios, CRDTs could be an alternative, though this design focuses on OT.

* Edits are sent as operations to the server, transformed in order, & then saved.

* A "Cassandra-backed, append only document operations DB" ensures fast writes & durability, partitioned by documentId & ordered by timestamp.

(3) Users should be able to view each other's changes in real-time:

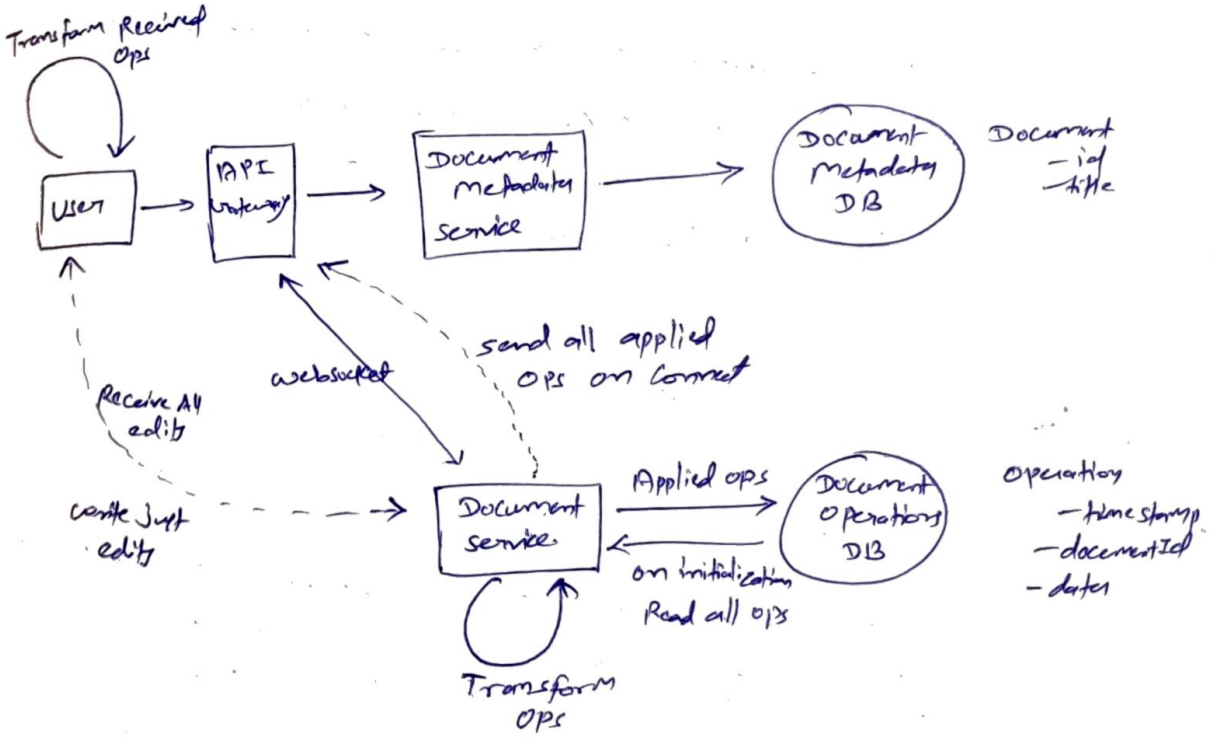
To allow users to see each other's changes in real-time, you need to manage two key aspects of reading data:-

(a) Initial Document loading

↳ when a user first connects or reconnects to a document, the system pushes all prior operations from the database to their "Websocket Connection". This ensures everyone starts with the latest document version.

(6) Real Time Updates & Consistency –

- when a collaborator makes an edit, the server records the operation in the database & then immediately sends it to all other connected clients for that document.
- client also perform operational transform (OT) on incoming updates. This is crucial because users expect their own changes to appear instantly...
- OT handles out-of-order edits, ensuring that no matter the sequence in which operations are received by different clients or the server, everyone's document ultimately becomes consistent.
- This means even if User A's edit arrives before User B's on the server, & User B's local edit was applied first on their end, OT guarantees that both users will see the exact same final document state.



(4) Users should be able to see the cursor position & presence of other users.

↳ This feature is crucial for avoiding simultaneous edits in the same area & prevented duplicated work.

Given that cursor position & user presence are ephemeral (only relevant in the moment & while connected), this data isn't stored within document itself. Instead:-

- Users report changes to their cursor position to Document Service.
- Document service stores this info 'in-memory' & immediately broadcasts it to all other connected users via the same websocket connection.
- when a new user connects, Document service retrieves current cursor positions & presence of other users from memory & send them to the new client.
- The Document service also monitors disconnections; when a user leaves, their info is removed from memory & a broadcast is sent to remaining users to reflect their departure.

Potential Deep Dives

(1) How do we scale to millions of websocket connections?

- we need to scale number of 'Document Service' servers to the number of concurrent connections. when I connect to 'DocumentService', I either (a) need to be co-located with all other users connected to same document or (b) know where all the other users are

connected. (see whatsapp for a similar problem).

=> To ensure million of concurrent users & ensure high availability, the core strategy is to "horizontally scale the Document service", which handles real-time websocket connections.

Key points includes:-

* Websocket Edge Termination — websockets are managed at the Document service "edge" to maintain statelessness in other backend services.

* Consistent hashing for Distribution :- A consistent hash ring, coordinated by Apache zookeeper, is used to distribute document connections. This ensures all collaborators on a single document are routed to and co-located on the "same Document service server".

* client connection flow — Clients connect to an initial server, are redirected if necessary to the correct server based on the hash ring & then establish a websocket connection. This co-location enables efficient "broadcasting of real-time updates" directly to all connected users for that document.

* Scalability challenges — Significant "state migration (connection, document operations)" is required when servers are added/removed.

↳ This necessitates robust handling of server failure, client reconnection logic & careful capacity planning.

(2) How do we keep storage under control ?

To manage storage effectively with potentially billions of documents (assume each document is 50KB ⇒ total 50TB) & countless operations, & to keep active document memory usage low, periodic snapshotting/compaction of operations is crucial. This reduces stored operations & data transferred to clients.

two approaches –

(a) Good approach – offline snapshot/compact (with compaction service)

* **Approach** → A dedicated compaction service periodically reads operations from the Document operations DB, compacts them (eg into a single insert) & writes them to a new documentVersionId in the DB.

* **Atomicity** → To ensure consistency for live documents, the compaction service tells the Document service to "flip" the documentVersionId. If the document is currently active/loaded in the Document service, this flip command is deferred to avoid corrupting in-flight operations.

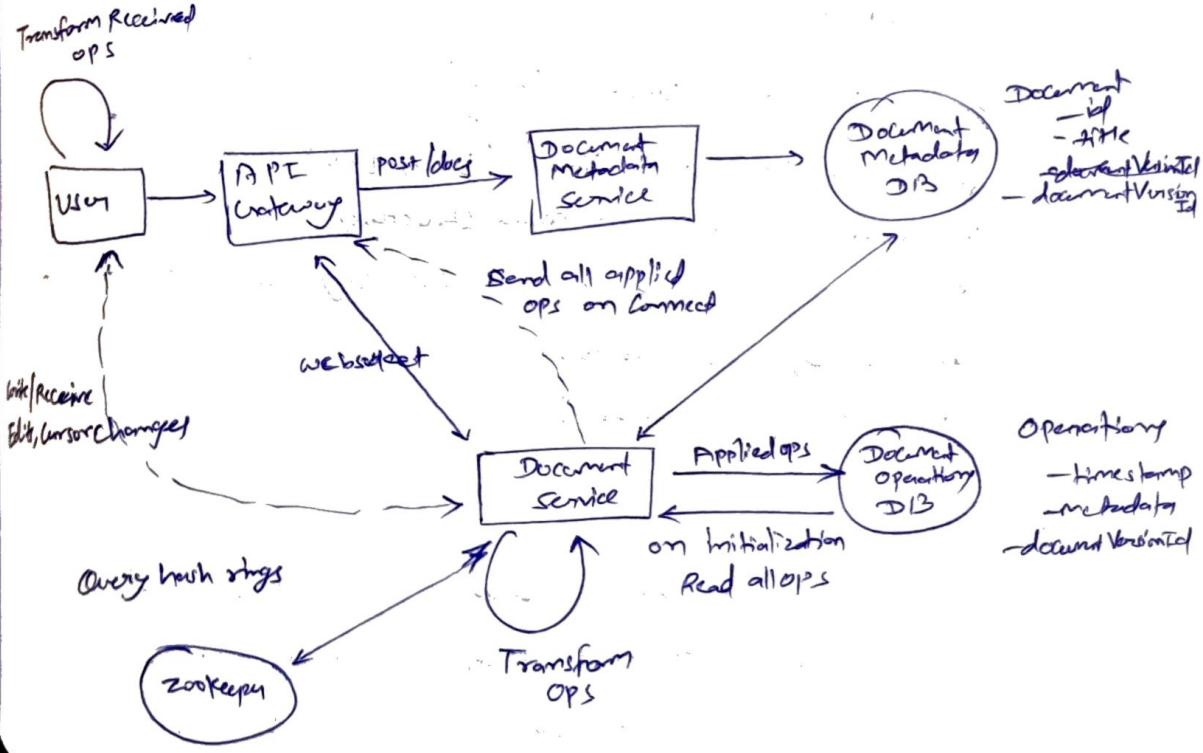
* **Challenges** → This involves complex distributed transaction, especially with databases like Cassandra that have limited transaction scopes (row-level only). Coordinating b/w services to ensure atomicity is tricky.

(b) current approach — online snapshot/Compact (within Document service)

* Approach → The Document service itself periodically performs Compaction. This is ideal when a document become idle (no active connection), as the Document service already has all operations in memory & has exclusive ownership.

* Process → when the last client disconnects from a document, the Document service offloads compaction to a separate, low-priority CPU process. The results are written to the DB under new documentVersionId, which is then flipped in the Document Metadata DB

* Challenges → The main risk is increasing latency for live document operations if Compaction tasks hog CPU & resources. Running Compaction in a separate, lower-priority process helps mitigate this.



Additional Deep Dive -

1. Read-only mode
2. Versioning
3. memory (memory usage of Document Service Can be a bottleneck) How can we further optimize it?
4. offline mode.

(1) Read only mode -

To scale read only access significantly, it bypassed the complex OT system. This involves: "separate access path", "serving static snapshots via CDN's", "using read replicas", and for "live" read-only, employing a dedicated Read-only stream service that receives pre-computed updates from the main Document service & broadcasts them without client-side OT.

(SSE over websockets)

(2) Memory optimization (Document Service)

* Smart Document Eviction (LRU, LFU)
* Efficient in-memory data structures
* Offload & asynchronous processing (for Compaction Process)
* Language/Runtime tuning (C++)
* Sharding (inherent)

(3) Versioning : -

To implement versioning, each compacted document snapshot is preserved as a distinct, timestamped historical version. The version metadata (e.g ID, Timestamp, author, Parent) is stored

in a dedicated version History DB.

past document states are reconstructed by loading the relevant snapshots & then applying any subsequent granular operations up to the desired point.

The user interface allow Browse, Previewing & restoring these historical versions as the current live document.

Apache Cassandra

(1) Introduction –

Type :- Open source, distributed NoSQL DB
Origin :- Facebook (inspired by Dynamo + Bigtable)
Strengths :- Scalable, fault-tolerant, highly available, schema-flexible
Use cases :- Netflix, Discord, Apple, Bloomberg

(2) Data Model –

* Keyspace – Like RDBMS database; Configure tables, define replication.
* Table – store rows.
* Row – Identified by a "Primary Key".
* Column – Flexible schema; supports UDTs, timestamps & sparse data.

– (Think of Cassandra like a JSON of tables with flexible columns per row.)

(3) Primary Key –

* PrimaryKey = (Partition Key) + [clustering key(s)]
* Partition Key = Decide data distribution
* clustering key = Decide row sorting within a partition.

– Use clustering keys for ranges scans & sorting.

(4) Partitioning –

* Uses Consistent Hashing on a ring.
* Data stored on first Vnode clockwise from the hash.
* Vnodes (virtual nodes) :- Distribute load evenly across physical nodes.

– Adds/removes nodes with minimal data reshuffling.

(5) Replication —

(a) SimpleStrategy → for dev/test. clockwise replica placement.
(b) NetworkTopologyStrategy → Prod use. Data center + rack aware

```
// simple
REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 3}

// Network aware
REPLICATION = {'class': 'NetworkTopologyStrategy',
               'dc1': 3, 'dc2': 2}
```

(6) Consistency —

* Tunable Consistency: ONE, TWO, QUORUM, ALL
* QUORUM = $\lceil (N/2) + 1 \rceil$ → overlap in reads/writes
  = read-your-write guarantee
* Eventual Consistency is default behaviour.
* No full ACID : only atomic + isolated row-level writes.

(7) Query Routing :—

* Any node can be a coordinator.
* Coordinator :—
  - knows all nodes (via gossip)
  - Routes requests to appropriate replicas using hash rings & replication info.

(8) Storage Model —

* Uses LSM Tree (not B-tree) :—
  - CommitLog (WAL) → Memtable (RAM) → SSTable (disk, immutable)
    (write ahead log) (Log structured Merge Tree)
  - "Tombstones" mark deletions

* Read flow :
  - Memtable → Bloom filter → SSTables (new → old)

⇒ Compaction : merges SSTables, cleans tombstones.
⇒ SSTable Indexing : Maps keys to byte offsets for fast disk access.

(9) Gossip Protocol –

* Peer-to-peer info sharing (live nodes, schema etc)
* Seed nodes :- Anchor gossip to prevent split cluster.
* Vector clocks :- Track node state freshness.

(10) Fault Tolerance –

* Failure Detection – Phi Accural Detector (Probabilistic)
* Hinted Handoff :-
  - stores temporary "hints", if a node is down,
  - Hints sent when node is back.
* Node is only "dead" after manual decommission,

(11) Data Modelling Strategy –

Cassandra = Query-first design (Not entity-relationship driven)

* Partition key :- Drives distribution
* Clustering key :- Drives sort/order
* Denormalize data to support access patterns.
* Avoid large partitions (Performance bottleneck)

(12) Example : Discord Messages –

Final schema :-

```
CREATE TABLE messages (
    channel_id bigint,
    bucket int,
    message_id bigint,
    ---
    PRIMARY KEY ((channel_id, bucket), message_id)
) WITH CLUSTERING ORDER BY (message_id DESC)
```

- Snowflake ids → Avoid timestamp collisions
- Bucket :- 10-day window avoids unbounded partitions.
- Partition = (channel_id, bucket)

PostgreSQL

(1) Why PostgreSQL?

* ACID-Compliant with strong consistency.
* Ideal for systems needing complex relationships & structured + semi-structured (JSONB) data.
* Rich built-in features: Full-text search, geospatial features (PostGIS), JSONB querying.
* Scales horizontally via replication, Partitioning & sharding.
* well-supported ecosystem & strong community.

(2) Read Performance & Indexing

* Default: B-tree indexes (equality, range, sort)
* Custom: Create indexes for foreign keys, filter conditions.
* Covering Indexes: INCLUDE all needed columns to avoid table lookups.
* Partial Index: Smaller + Faster indexes for filtered data.

```
CREATE INDEX idx_user_email ON users(email);
CREATE INDEX idx_post_user_include ON posts(userid) INCLUDE (title, created_at);
CREATE INDEX idx_active_user ON users(email) WHERE status = 'active';
```

(3) Full-text & JSONB search

* Full-text (tsvector + GIN) - Fast word search, relevance ranking, stemming.
* JSONB + GIN - Store flexible metadata (mentions, tags etc)

5) Geospatial with PostGIS –

* Add extension : CREATE EXTENSION Postgis ;
* Use GIST Indexes for fast Proximity queries.

6) Query optimization –

* Keep working set in RAM : Huge boost in speed.
* Avoid over-indexing : Slow writes, increases space usage.
* Use Covering + Partial indexes for critical query paths.

6) Write architecture –

* WAL (write-Ahead Log) : Durable disk write.
* Buffer cache : In-memory data updates.
* Background writer : Flushes dirty pages to disk.
* Indexes : Every write updates relevant indexes (costly with many indexes)

7) Write optimization Techniques –

(a) Vertical Scaling – (Upgrade hardware : NVMe disks, more RAM, better CPUs)

(b) Batch Processing –

* Instead of 1000 separate inserts :-

INSERT INTO likes (post_id, user_id) VALUES (1,101),(1,102), ... ;

(c) Write offloading – Async write via Kafka or queues for logs/analytics/metrics

(d) Table Partitioning

(Create main table)

CREATE TABLE posts(...) PARTITION BY RANGE (created_at);

CREATE TABLE post_2024_01 PARTITION OF posts FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

(Create one Partition)

These two statements together define one partition of posts table.

(8) sharding :-

- shard by user_id, avoid cross-shard queries.
- use tools like Citus for easier sharding

(9) Replication :- (Scaling Reads + High availability)

* Asynchronous : Faster, risks of lag.
* Synchronous : Slower, Stronger Consistency
* Used read replicas to offload queries
* Auto-failover in managed solutions (AWS RDS, CNPG).

(9) Data Consistency & Transactions -

ACID:

* Atomicity — All or nothing writes
* Consistency — DB remains valid state
* Isolation — Each transaction as if alone
* Durability — Committed = won't be lost

Isolation levels: Read Committed, Repeatable Read, Serializable

(10) When to Consider alternatives —

Need	Alternative
(a) > 1M writes/sec.	Cassandra, DynamoDB
(b) Global writes (multi-region)	CockroachDB
(c) Simple KV access	Redis, DynamoDB

Interview tips :-

(a) Start with PostgreSQL, unless requirements demand otherwise.

(b) Justify with ACID, indexing, write/read scaling, search, & partitioning.

(c) Don't just say "PostgreSQL is ACID" – show how you use it for consistency.

(d) Mention trade offs : eg- (write throughput vs number of indexes).
, (read replicas vs lag).