DESIGN FB NEWS FEED

FR

- Create post
- follow users
- View Feed
- Page through Feed

Bonus

- Like and comment
- Privacy settings

NFR

- Eventual consistency (lil bit of lag is OK) (CAP)
- Say latency we want ≈ 500ms for posting and viewing
- 2b DAU

Core Entities

- User
- Post
- Follow

} we can add on as we go

APIs

POST /posts

{

content

}

→ say returns posted.

B's id

PUT /users/[id]/followers { } → A wants to follow B, then this API.

DELETE /users/[id]/followers } → to unfollow.

→ why PUT ? why not POST ?

POST → create a new resource.

PUT → updates an existing resource primarily.

So,

POST → not idempotent (calling it twice, creates 2 resources)

PUT → idempotent (calling it twice = same result as calling once)

Now to paginate through the feed

GET /feed?pageSize={size}&cursor={cursor}

(or)

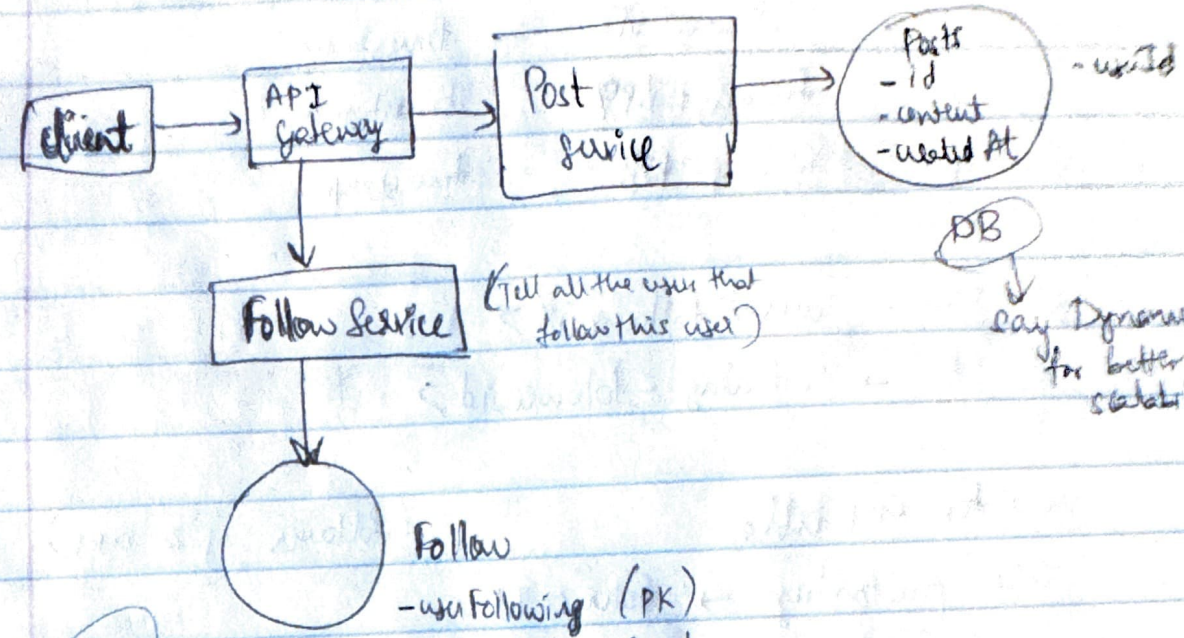
GET /feed?cursor=post456&limit=20

{

posts = Post[]

nextCursor = " "

}



enables efficient reverse lookups

PK → partition key → determines how our data is physically distributed across partitions (storage nodes)

say partition key = user_123, then all data belonging to that user will go to the same partition.

SK → Sort Key → DynamoDB treats

~~follow~~

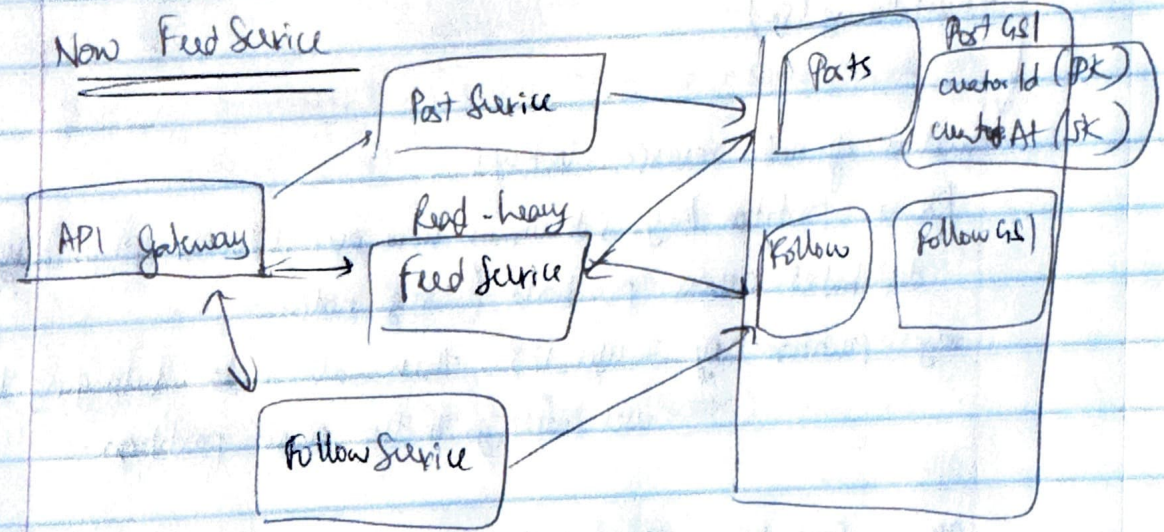
PK	SK	Other attributes
usr#123	following#456	timestamp
usr#123	following#789	timestamp
usr#456	following#123	timestamp

PK → user <follower_id>
FK → following <followee_id>

(followers of a user).

and for GSI table
GSI partition key → FolloweeId
GSI SortKey → FollowerId

Dynamo DB



Why do we need Post GSI?

So firstly our post table looks like this →

PostId	UserId	CreatedAt	Content

Here primary key is postId ~~So now for~~

Before understanding this - Let's try to understand how feed service is trying to view the feed.

① Get all users U that user X is following:

② For all those users U, get all their posts (older than timestamp T if paging).

So, since the primary key is PostId. We cannot do it efficiently as the fast query is on the PostId and there is no index on ~~createdAt (or)~~ creatorId.

So Posts GSI:

PK → creatorId

sortKey → createdAt

So this would help us design our first naive design where we -

① First get N users that a person follows

② Then want to get posts for those N users efficiently

Problems with this design?

→ User X might follow hundreds (or) 1000s of users
→ Each user might have millions of posts
→ We need to fetch posts from many partitions (creatorId) & then globally sort them ourselves, which could result in fetching huge datasets just to return 20 items on the paginated feed.

So, how can we avoid this?

PreComputed Feed

- Instead of building feed on demand
- ~~We will~~ whenever a user creates a post, we will store in this precomputed feed cache.

Precomputed Feed table

- userId : PK
- CreatedAt : SK
- postIds[ ]

So, when a userId posts anything, we immediately insert postId into all of their followers feed.

When UserB open their feed:
We just read from their precomputed feed.

Advantage:
Read latency & complexity is dropped dramatically

① System writes the post to the Posts table
② The system fans out this postId to the precomputed Feed for every follower of user A

More problems to this design
④ How do we handle users with a large number of followers?
When a user with millions of followers creates a POST, writing the post to millions of Precomputed Feed record all at once may lead to
→ High write amplification
→ Load imbalance
→ Throughput spikes
→ single post service getting overwhelmed

⑤ How do we handle uneven reads (hot keys) on Posts table?
some posts will receive huge read traffic
hot key → single partition/node overloaded

The above are for Staff roles in-depth.