System Design Day 5: News Feed System

📲 News Feed System Design (Facebook-style)

1. 🧾 Requirements

Functional Requirements

- Create a post.
- Follow a user (unidirectional).
- View a paginated news feed.
- [Bonus] Like/comment, post privacy, unfollow.

Non-Functional Requirements

- **Availability:** Eventual consistency is acceptable for post propagation.
- **Latency:** <500ms for post/feed access.
- **Scale:** ~2 billion users.
- **Feed freshness:** A new post should appear in ~1 minute.

---

2. 📦 Core Entities

- **User**
- **Post**: ID, content, timestamp, creator.
- **Follow**: follower → followee (unidirectional).

---

3. 📡 APIs Design

```
POST /posts
  → Creates a new post
  → Returns 201 + post ID
```

```
PUT /users/{id}/followers
  → Adds a follow relationship

GET /feed?pageSize=25&cursor=timestamp
  → Returns paginated list of posts
  → Includes `nextCursor` for deep scroll
```

---

4. 🧱 High-Level Design

a. Post Creation Flow

- Client → API Gateway → Post Service
- Data stored in **DynamoDB** (or Cassandra).
- Uses `Spring Data` repositories for persistence.

b. Follow Service

- Stores follows in a **Follow Table** with:
  - `Partition Key`: Follower ID
  - `Sort Key`: Followee ID
- Add **GSI** for reverse lookup (i.e., who follows a given user).

c. Feed Service

- Read-heavy service.
- Fetches posts of followed users.
- Merges, sorts by timestamp, and paginates.

---

5. ⚙️ Feed Generation Strategy

Strategy	Description	Pros	Cons
Pull Model (On-demand)	At request time, aggregate posts from followed users.	Fresh data, no storage overhead.	High latency for large follows.
Push Model (Precomputed)	On post creation, fanout post ID to	Fast reads.	High write cost for viral accounts.

Strategy	Description	Pros	Cons
	follower feeds.		

🔁 **Hybrid Model**:

- Use push for normal users.
- Use pull for high-follower accounts (e.g., >100k).
- Mark in the `follow` table: `isPrecomputed=true/false`.

---

6. ⚡ Real-time Updates Handling

- Use **async worker pool** (e.g., RabbitMQ or Kafka).
- When a post is created:
  - Add a task to a queue.
  - Workers fetch followers and update their feed entries asynchronously.

---

7. 🧠 Caching Strategy

- Use **Spring Cache** (backed by Redis or Caffeine).
- Cache recent or popular feed responses per user.
- Use TTL (e.g., 10-30 seconds) and LFU (Least Frequently Used) policy.
- Also cache **hot posts** to reduce DB load (avoid hot shard issue in DynamoDB).

Hotkey Solution:

- Instead of one cache node → **Multiple cache replicas**.
- Requests are randomly routed across instances → reduced pressure per node.

---

8. 🔪 Scalability Techniques

- **Sharding**: Split users across shards using consistent hashing.
- **Load Balancing**: Use AWS ALB/NLB or Nginx.
- **Pagination with cursors**: Scalable feed traversal without offset overhead.
- **Rate Limits**: Enforced for posting or following limits (prevent abuse).

---

9. 👨‍💻 Java/Spring Integration

Feature	Implementation
Persistence	Spring Data + NoSQL (e.g., DynamoDB, Cassandra) repositories.
Feed Storage	Use Spring Cache abstraction + Redis for precomputed feeds.
Async Updates	Use `@Async`, Spring Batch, or Kafka consumers for feed fanout.
API Layer	Spring Boot REST endpoints for post, follow, and feed ops.

10. ✅ Final Thoughts

- Real-world systems like Facebook and Twitter use a **combination of precomputed feeds + on-demand merges**.
- System design is about trade-offs:
  - Speed vs. freshness.
  - Write cost vs. read cost.
  - Simplicity vs. performance.
- Interviewers care more about **your process** than a perfect answer.

🧮 11. Storage Estimates

- Precomputed Feed:
  - 200 post IDs/user ≈ 2KB.
  - 2B users = ~4TB (manageable).
- Hot post cache: 10K posts * 1KB = ~10MB.

🧩 12. Failure & Recovery Considerations

- **Retry mechanisms** in feed workers.
- **Dead letter queues (DLQ)** for failed post fan-outs.
- **Graceful degradation** (fallback to pull if feed is stale).