**Sharding in System Design**

**Example: Instagram Scaling with Sharding**

**📌 Scenario:**

Imagine **Instagram** in its early days:

* A **single SQL database** stores user data, posts, likes, and comments.
* Initially, the database **handles traffic well** because the user base is small.
* As **millions of users** join, the **database becomes a bottleneck**, slowing down queries.

**🔴 Problem: Why a Single Database Fails?**

1️⃣ **Increased Read/Write Load** – Millions of users posting images, liking, and commenting.  
2️⃣ **Storage Limitations** – A single server’s disk space and performance are **limited**.  
3️⃣ **Slow Queries** – Searching for posts from a single, large table takes too long.  
4️⃣ **Downtime Risks** – If the **single database crashes**, the entire app **goes down**.

**🟢 Solution: Sharding**

🔹 **Sharding** = Splitting a large database into **smaller, faster, more manageable** parts (**shards**).  
🔹 Each shard is **independent** and stores a **subset of data**.

**🔹 Step-by-Step Implementation at Instagram**

**Step 1: Choosing a Sharding Key**

Instagram has multiple ways to **split data**:  
✅ **User-based Sharding** → Store all posts of a user on the same shard.  
✅ **Post ID-based Sharding** → Store posts across multiple shards by post ID.  
✅ **Geographical Sharding** → Store posts based on the user's location.

🔸 Instagram **chose** **User-based Sharding** because:  
✔️ Ensures all a user’s data is in one place.  
✔️ Reduces cross-shard queries for user profiles.

**Step 2: Distributing Data Across Shards**

* Users are **evenly distributed** across shards based on **User ID**.
* Example:
  + **Shard 1 → Users 1-1M**
  + **Shard 2 → Users 1M-2M**
  + **Shard 3 → Users 2M-3M**

When **User 150,000** posts a picture:  
➡️ Query directly goes to **Shard 1** instead of searching all data.

**Step 3: Routing Queries Efficiently**

Since data is split across **multiple databases**, how does Instagram know **where to find a user’s data?**  
✅ **Shard Map (Metadata Store)** – A lookup table that tells the system which shard stores which user.  
✅ **Load Balancer** – Ensures queries go to the **correct** shard, preventing overload.

**🟢 Optimizing for Scalability**

✅ **Replication** → Each shard has a **read replica** to **distribute read queries** (faster loading).  
✅ **Consistent Hashing** → Avoids the problem of unbalanced shards when **new shards** are added.  
✅ **Auto-sharding** → When a shard reaches **80% capacity**, a new one is created, and users are redistributed.

**🔹 Trade-offs of Sharding**

| **Approach** | **Pros** | **Cons** | **When to Use?** |
| --- | --- | --- | --- |
| **Single Database (No Sharding)** | Simple, easy to manage | Doesn’t scale beyond a limit | Small applications with <100K users |
| **Horizontal Sharding** | High scalability, better load distribution | Complex routing, cross-shard queries | Large applications with high traffic (Instagram, Twitter, YouTube) |
| **Vertical Scaling** | Easy to implement (upgrade RAM/CPU) | Limited by server capacity | Useful for **small to medium** applications before sharding |

**📌 Final Scalable System at Instagram**

✔️ **User-based sharding** ensures fast queries and better distribution.  
✔️ **Load Balancers** handle traffic to the correct shard.  
✔️ **Replication** ensures backups and fast reads.  
✔️ **Auto-sharding** helps Instagram handle **millions of users** smoothly.

**🛠️ Real-World Examples of Sharding**

* **Instagram** → User-based sharding for posts & profiles.
* **YouTube** → Video metadata sharded by **video ID**.
* **Twitter** → Tweets are sharded based on **Tweet ID** for fast retrieval.
* **MongoDB Atlas** → Auto-sharded NoSQL database.