

Step 1: Sharding Concepts Review

Horizontal vs. Vertical Partitioning

- **Horizontal Partitioning (Sharding):** Splitting a table by **rows**. You move different rows into different physical databases.
 - *Appropriate when:* The table has too many rows to fit on one disk or handle one server's CPU load.
 - *Scalability:* Offers near-infinite linear scaling.
- **Vertical Partitioning:** Splitting a table by **columns**. You move specific columns (e.g., "User Bio") into a separate table or database from "User Login."
 - *Appropriate when:* You have very "wide" tables where only a few columns are frequently accessed.

Sharding vs. Indexing

- **Indexing** is a pointer system that helps the database find data faster within a single server. It is usually sufficient until the hardware (CPU/RAM/IO) hits a ceiling.
 - **Sharding** is necessary when you exceed the physical limits of a single machine.
 - **Together:** You always use both. Every shard is internally indexed to ensure that once a query reaches the correct shard, it executes instantly.
-

Step 2: Shard Key Selection

Dataset 1: User Accounts

- **Proposed Shard Key:** `user_id`
- **Strategy:** Hash-based
- **Justification:** `user_id` has the highest cardinality. Using a hash ensures that users from the same country or those who joined on the same day are spread randomly across all shards, preventing "hotspots."
- **Trade-offs:** Searching by `email` or `username` will require a cross-shard "scatter-gather" query unless you maintain a global lookup index.

Dataset 2: Transaction Records

- **Proposed Shard Key:** `user_id`
- **Strategy:** Hash-based (or Composite with Date)

- **Justification:** Most queries ask "Show me all transactions for User X." By sharding on `user_id`, a single user's history lives on one shard, making common queries extremely fast.
- **Trade-offs:** A massive merchant (like Amazon) might have billions of transactions, creating a "hot shard" if you shard by `merchant_id` instead.

Dataset 3: IoT Sensor Data

- **Proposed Shard Key:** `device_id`
- **Strategy: Composite (`device_id + month`)**
- **Justification:** High write volume is spread across shards by `device_id`. Including a time element (month) allows for easier "TTL" (Time To Live) management where old data shards can be archived.
- **Trade-offs:** Queries for "Average temperature across ALL devices" will hit every shard.

Dataset 4: Orders by Region

- **Proposed Shard Key:** `region`
 - **Strategy: Range/Directory-based (Geographic)**
 - **Justification:** Since queries are often region-specific, keeping European orders in a "EU-West" data center reduces latency and helps with data residency laws (GDPR).
 - **Trade-offs:** If the "North America" region grows 10x faster than others, that shard becomes a bottleneck.
-

Step 3: Risk Analysis

Dataset	Hot Shards Risk	Query Routing Risk	Data Skew Risk	Mitigation Strategies
User Accounts	Low (Hash spread)	High (Email search)	Low	Use a secondary index for username lookups.

Transactions	Medium (Power users)	Low (User-centric)	High	Split "Power User" data or use salt in hash.
IoT Data	Medium (Active devices)	Medium (Global stats)	Medium	Use sub-partitioning by timestamp.
Orders	High (Large regions)	Low (Regional)	High	Implement "Shard Splitting" for large regions.

Step 4: Final Design Summary

When is Sharding Necessary?

Sharding is a "last resort." You should only shard when:

- Write Throughput:** A single leader cannot handle the volume of incoming **INSERTs**.
- Dataset Size:** The database exceeds 2-5 TB, making backups and restores take days.
- Geography:** You must store data in specific countries for legal reasons.

Decision Matrix

Dataset	Shard Key	Strategy	Primary Benefit	Main Risk
Users	<code>user_id</code>	Hash	Perfect write balance	Non-ID lookups

Transactions	user_id	Hash	Fast user history	Large user skew
IoT	device_id	Composite	Balanced ingestion	Global analytics
Orders	region	Directory	Low latency/Legal	Region growth