Why normal hashing fails

সাধারণ হ্যাশিং (যেমন: $server = hash(key) \pmod n$) কেন বড় ডিস্ট্রিবিউটেড সিস্টেমে ফেইল করে বা কাজ করে না, তার প্রধান কারণ হলো এর **অস্থিরতা (Instability)**।

নিচে বিস্তারিত কারণগুলো তুলে ধরা হলো:

১. সার্ভারের সংখ্যা পরিবর্তন (Scaling Issue)[1]

সাধারণ হ্যাশিং-এর সবচেয়ে বড় সীমাবদ্ধতা হলো এটি সার্ভারের মোট সংখ্যার (n) ওপর সরাসরি নির্ভরশীল।

- **সার্ভার যোগ করলে ($n+1$):** ধরুন আপনার ৩টি সার্ভার আছে। এখন ট্রাফিক বাড়ার কারণে আপনি ৪র্থ একটি সার্ভার যোগ করলেন। যেহেতু এখন মোডুলাস ভ্যালু ৩ থেকে ৪ হয়ে গেছে, তাই প্রায় **৭৫% থেকে ১০০%** ডাটার নতুন পজিশন বা ক্যালকুলেশন বদলে যাবে।
- **সার্ভার কমে গেলে ($n-1$):** একইভাবে একটি সার্ভার ক্র্যাশ করলে আপনার পুরো হ্যাশ টেবিল এলোমেলো হয়ে যাবে।

২. ক্যাশ মিস (Massive Cache Miss)

সাধারণত হ্যাশিং ব্যবহার করা হয় যাতে নির্দিষ্ট ডাটা সবসময় নির্দিষ্ট সার্ভারেই পাওয়া যায়।

- যখন সার্ভার সংখ্যা পরিবর্তনের কারণে ডাটার লোকেশন বদলে যায়, তখন রিকোয়েস্টটি ভুল সার্ভারে যায়।
- সেখানে ডাটা না পেয়ে সিস্টেমকে আবার মেইন ডাটাবেস থেকে তথ্য আনতে হয়। একে বলে **Massive Cache Miss**। এর ফলে ডাটাবেসের ওপর হঠাৎ করে প্রচণ্ড চাপ পড়ে, যা পুরো সিস্টেমকে ডাউন করে দিতে পারে।

৩. ডাটা ডিস্ট্রিবিউশনে অসমতা (Uneven Data Distribution)

সাধারণ হ্যাশিং সবসময় গ্যারান্টি দিতে পারে না যে প্রতিটি সার্ভারে সমান পরিমাণ ডাটা থাকবে।

- কোনো একটি সার্ভারে হয়তো অনেক বেশি ডাটা চলে গেল (Hotspot), আবার কোনোটি প্রায় খালি পড়ে থাকল।
- এর ফলে রিসোর্সের সঠিক ব্যবহার হয় না এবং কোনো একটি নির্দিষ্ট সার্ভার চাপে ভেঙে পড়তে পারে।

৪. রি-হ্যাশিং এর খরচ (Re-hashing Cost)

যখনই আপনি একটি নতুন নোড বা সার্ভার যোগ করবেন, আপনাকে প্রায় সব ডাটা পুরনো সার্ভার থেকে সরিয়ে নতুন নির্ধারিত সার্ভারে ট্রান্সফার করতে হবে।

- কোটি কোটি ডাটা মুভ করা অত্যন্ত সময়সাপেক্ষ এবং এটি নেটওয়ার্ক ব্যান্ডউইথ শেষ করে ফেলে।

একটি সহজ উদাহরণ দিয়ে বুঝে নিই:

মনে করুন, আপনার ৩টি ঝুড়ি আছে এবং আপনি ৯টি আম ভাগ করে রাখছেন ($9 \pmod 3 = 0$)। অর্থাৎ ৯ নম্বর আমটি ০ নম্বর ঝুড়িতে যাবে।

এখন হঠাৎ একটি ঝুড়ি ভেঙে গেল, আপনার হাতে আছে ২টি ঝুড়ি। এখন হিসেব হবে $9 \pmod 2 = 1$।

আগে আমটি ছিল ০ নম্বর ঝুড়িতে, এখন তাকে সরাতে হবে ১ নম্বর ঝুড়িতে। এভাবে প্রায় প্রতিটি আমের জায়গা বদলে যাবে। এটাই সাধারণ হ্যাশিং-এর বড় ব্যর্থতা।

সমাধান কী?

এই সমস্যা সমাধানের জন্যই **Consistent Hashing** ব্যবহার করা হয়। সেখানে ডাটা মুভমেন্ট হয় খুবই সামান্য (শুধুমাত্র $1/n$ পরিমাণ ডাটা সরাতে হয়), যা সিস্টেমকে অনেক বেশি স্টেবল এবং স্কেলেবল করে তোলে।

🧠 Normal Hashing কী?

👉 Traditional sharding formula:

shard = hash(key) % N

Where:

- key = userId / objectId
- N = number of shards

❌ Where It Fails (CORE PROBLEM)

1. Shard Count Change = Disaster 😵

ধরি:

N = 4 shards

User 101:

hash(101) % 4 = 1 → Shard-1

Add one shard

N = 5

hash(101) % 5 = 3 → Shard-3

🚨 Same key → different shard

🚨 Almost **ALL keys move**

📌 Interview line:

“Normal hashing causes massive data reshuffling when shard count changes.”

2. Re-sharding Cost (Production Nightmare)

When shards change:

- TBs of data move
- Heavy network traffic
- System slowdown
- Possible downtime

❌ Not acceptable at scale

3. Cache Invalidation Explosion

In caching systems:

- Key → new shard
- Cache miss everywhere
- Cold start storm

📌 Example:
Redis cluster resize → hit ratio drops

4. Load Balancer Instability

Normal hashing used in:

Request routing → server = hash(clientId) % N

When server added/removed:

- Users bounce to different servers
- Session loss
- Sticky session breaks

5. Failure Handling is Bad

If one shard fails:

N decreases

→ ALL keys re-mapped

❌ Cascading failure risk

📌 Interview line:

“A single node failure can cause system-wide rehashing.”

⚖️ **Why This Is Fundamentally Wrong?**

Normal hashing assumes:

- Fixed number of nodes
- No failures
- No scaling

📌 Reality:

Distributed systems are dynamic.

🧠 Key Insight (INTERVIEW GOLD)

Hashing should depend on key, not total node count

Normal hashing violates this.

✅ How Consistent Hashing Fixes It (Preview)

- Nodes on hash ring
- Key maps to nearest node
- Adding/removing node → only local keys move

📌 Data movement ≈ 1/N

🏆 FAANG Interview Perfect Answer 💎

Say this 👇

“Normal hashing fails in distributed systems because changing the number of nodes causes nearly all keys to be remapped, leading to massive data movement, cache invalidation, and instability. Consistent hashing solves this by minimizing key redistribution.”

🧠 One-Line Memory Hook

Normal hashing breaks when N changes