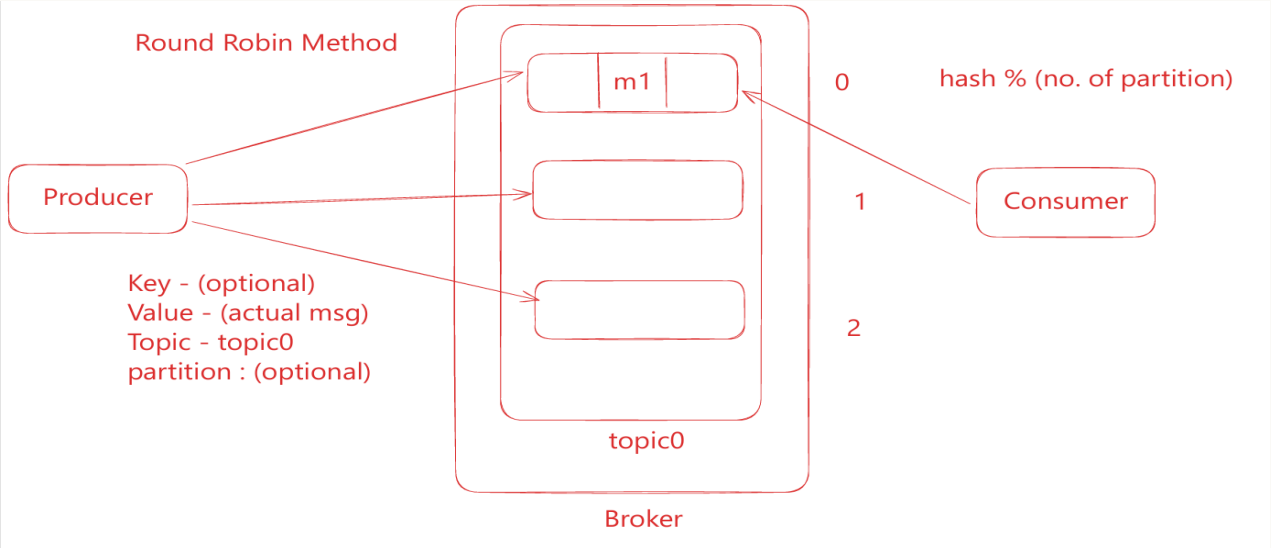
📨 LECTURE 09 : KAFKA PART -2 | CACHING TECHNIQUES | CACHE EVICTION POLICIES 🌟

🔹 PART 1: MESSAGING QUEUES

📌 KAFKA - COMPLETE ARCHITECTURE

Basic Components:



Message Structure:

```
{
  "Key": "optional (for partitioning)",
  "Value": "actual message content",
  "Topic": "topic0",
  "Partition": "optional"
}
```

Partitioning Methods:

- **Hash Method:** `hash(key) % number_of_partitions`
- **Round Robin Method:** Equal distribution in cycle ⚡

❓ Question & Answer 💡

Question1️⃣:
Jab producer partition ko message karta hai, to kaise decide karta hai ki kaunse topic ke liye partition me message send karna hai? 🤔

Answer1️⃣:

- Producer message send karta hai **saath me Key, Value, Topic, Partition** bhi bhejta hai ✉️
- **Value Topic** hamesha **mandatory** hota hai ✅

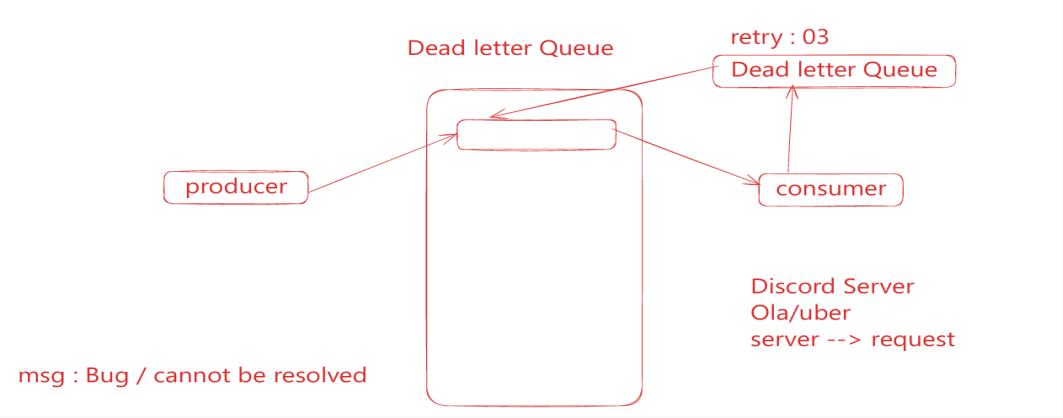
Question2️⃣:
Jab humare paas **na Key ho na Partition ki value**, to kaise decide karenge? 🤷‍♂️

Answer2️⃣:

- **Round Robin Method** ka use karenge 🔄
- Ek-ek karke saare partitions choose karte rahenge 🏃‍♂️💨

⚠️ DEAD LETTER QUEUE (DLQ)

Concept Flow:



💡 Explanation

- Dead Letter Queue me **wo saare messages store hote hain** jo **consumer consume nahi kar paata** ya **resolve nahi kar paata** ⚠️
- Reason: Message me **bug** tha ya **error** aaya jo automatically fix nahi ho sakta ❌

🔹 Retry Mechanism

- Messages ko **3 baar** try karte hain consume karne ke liye 🔄
- Agar **3 attempts ke baad bhi fail ho jaye**, to message **DLQ me move** ho jata hai 🚨

Example:

- Key messages with **bugs** that cannot be resolved ❌

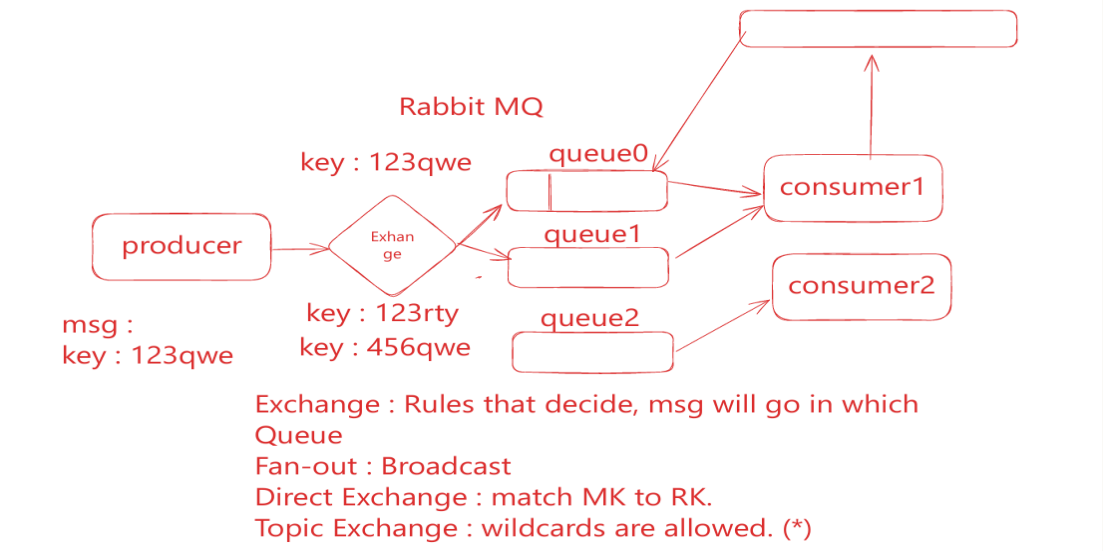
🔹 Real-World Use Cases 🌐

- **Discord Server**: Real-time messaging requests 📨
- **OLA / Uber**: Ride matching system 🚗
- **High-throughput servers**: Multiple requests at the same time ⚡

> ⚡ **Note:** Kafka ka **9 out of 10 use cases** me use hota hai for high throughput & reliable message delivery ✅

🐇 RABBITMQ - COMPLETE FLOW

Basic Architecture:



Exchange Types:

1. **Fan-out Exchange** 📢 → Broadcast to ALL queues
2. **Direct Exchange** 🎯 → Exact matching (Message Key = Routing Key)
3. **Topic Exchange** 🌟 → Wildcard matching (*)

Key Terminology:

- **Message Key (MK):** Key that comes with the message
- **Routing Key (RK):** Exchange key

Topic Exchange Example:

```
Msg:

key: 123qwe

key: 123qwe

key: 123rty

key: 456qwe


Topic Exchange:

MK: 123qty

RK: 123q*
```

🔄 Re-Queue Mechanism Comparison

- **Kafka**: Pull-based ⬇️
    - Consumer **pulls messages** when ready 🏃
    - ✅ High throughput, consumer controls rate
    - ❌ Complex setup
- **RabbitMQ**: Push-based ⬆️
    - Producer **pushes messages** automatically ⚡
    - ✅ Real-time delivery, simple
    - ❌ Consumer overload possible

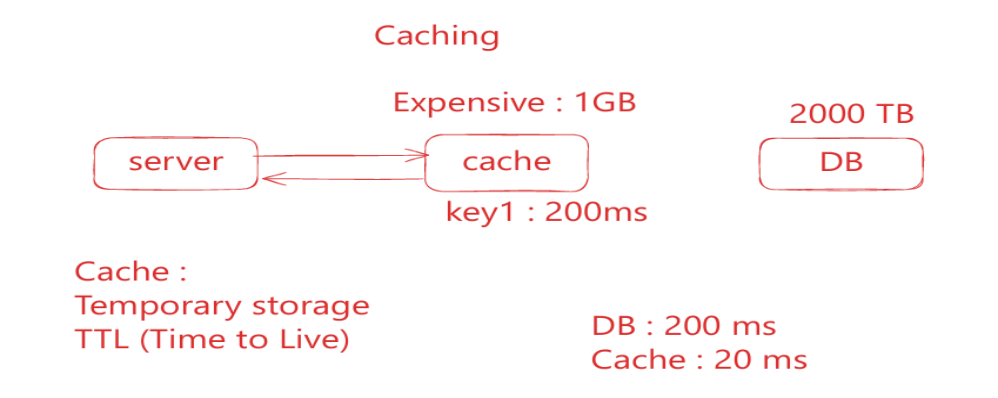
💡 **Tip**: Pull = consumer control, Push = producer control 🧠

🌟 Real-world Examples

- **Discord** → Real-time messaging 💬
- **Ola/Uber** → Ride matching system 🚖
- **Server Requests** → API communication 🌐

💡 **Note**: Ye sab systems me **messaging queues** ka use hota hai for **scalability & reliability** ⚡

💾 PART 2: CACHING SYSTEM



Performance Comparison:

```
Database Access: 200 ms ⏳

Cache Access:    20 ms ⚡

10x faster!
```

💾 Cache Characteristics

- **Temporary Storage** 🗄️ — Permanent nahi hai
- **TTL (Time to Live)** ⏱️ — Automatic expiration
- **Cost Effective** 💰 — 1GB Cache vs 2000TB Database
- **Access Time** ⚡ — key1: 200ms (DB) vs 20ms (Cache)

💡 **Note**: Cache use karne se **performance 10x faster** ho jata hai! 🚀

🎯 CACHE TYPES - COMPLETE LIST

Different Levels of Caching:

1. **CDN Caching** 🌐 → Static resources (images, CSS)
2. **Load Balancer** ⚖️ → Web pages caching
3. **Server-side** 💻 → Redis, Memcached
4. **Proxies** 🔁 → Intermediate caching
5. **Distributed** 🏗️ → Consistent hashing
6. **Client-side** 🖥️ → Cookies, local storage

🌐 Distributed Cache Architecture

- Jab **HLD me cache** discuss karte hain, **Distributed Caching** internally **Consistent Hashing** ka use karta hai 🔄

- Example Architecture:

```
┌──────────┐    ┌──────────┐    ┌──────────┐
| Cache 1 |    | Cache 2 |    | Cache 3 |
|    c1    |    |    c2    |    |    c3    |
└──────────┘    └──────────┘    └──────────┘
```

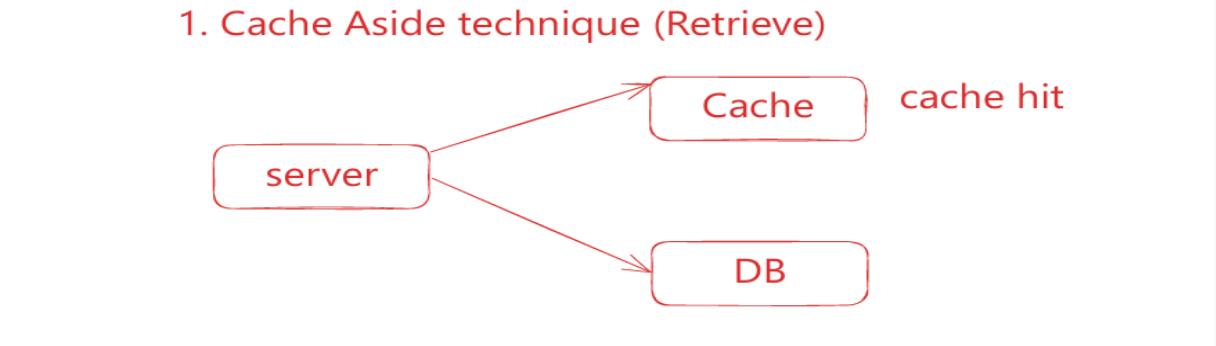
💡 **Tip**: Consistent Hashing se **load evenly distribute** hota hai aur **scalability** improve hoti hai 🚀

🔹 PART 3: CACHE STRATEGIES - STEP BY STEP

📊 READ STRATEGIES

1. CACHE-ASIDE TECHNIQUE 🗂️

`Server → Cache → [Hit/Miss] → DB`



- **Check Cache**
- **If Cache Hit** ✅ → Return data immediately
- **If Cache Miss** ❌ → Fetch from DB → Store in Cache → Return data

Pros: ✅

- Simple to implement
- DB & Cache can have different structures

Cons: ❌

Every new data → always a cache miss

Solution: 🔥

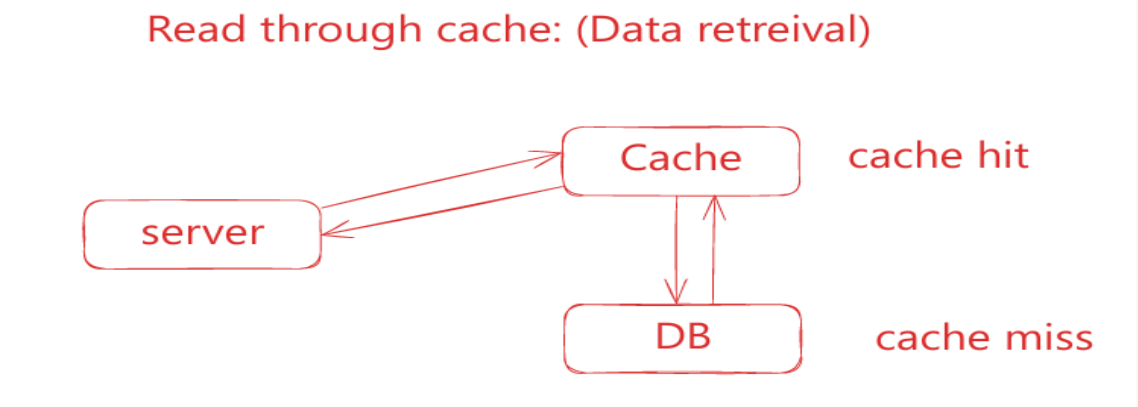
- Pre-heat the cache to avoid initial misses and improve performance

---

2. READ-THROUGH CACHE 🔁

```
Cache hit → return data

Cache miss → Cache library auto fetches from DB
```



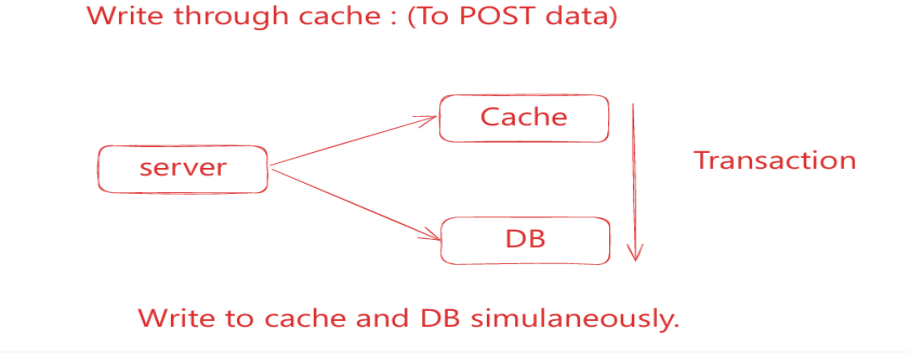
Workflow:

- **Cache Hit** ✅ → Return data
- **Cache Miss** ❌ → Cache auto-fetches from DB → Return data

Pros: ✅ Client doesn't handle DB logic
Cons: ❌ Cache structure must match DB, initial cache miss
Solution: 🔥 Pre-heat cache for smooth performance

📝 WRITE STRATEGIES

3. WRITE-THROUGH CACHE ✍️

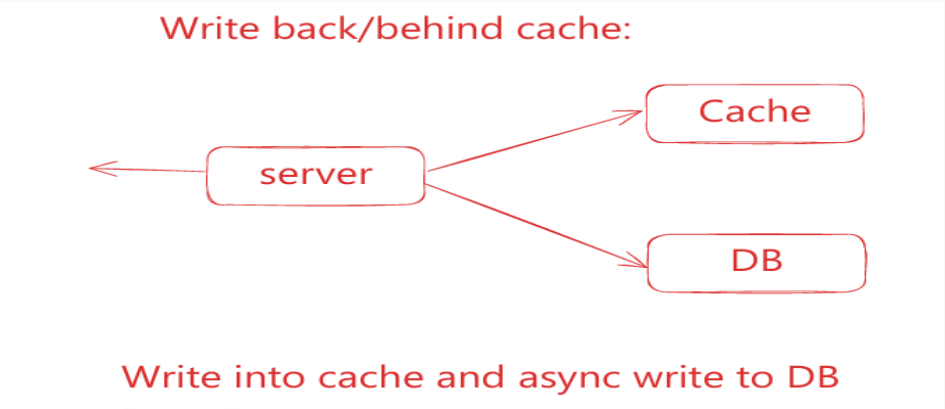


Workflow:

- Write → Cache → Simultaneously write to DB

Pros: ✅ High consistency
Cons: ⏳ Slow, uses 2-phase commit (overhead)

4. WRITE-BACK CACHE ⏩

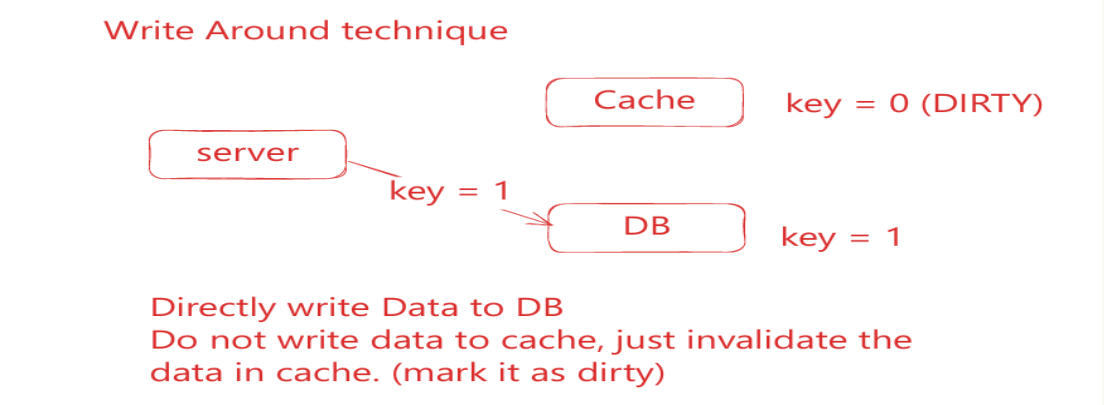


Workflow:

- Write → Cache → Async write to DB

Pros: ⚡ Very fast
Cons: ⚠️ Possible inconsistency

5. WRITE-AROUND CACHE ↪️



Workflow:

- Write → DB → Mark Cache as DIRTY
- **Example:** key=1 → DB write, Cache: key=0 (DIRTY)

Pros: ✅ Maintains consistency without complex protocols

🗑️ PART 4: CACHE EVICTION POLICIES

Why Needed:

- Limited cache storage 💾
- TTL not enough ⏱️
- Need to remove old data when cache full

🔄 COMPLETE EVICTION POLICIES (FULL FORM + FIGURE + DEFINITION + PROS/CONS + USE CASE)

---

1. LRU - Least Recently Used

🔹 Concept:

LRU cache hamesha **sabse purana accessed item** ko remove karta hai jab cache full ho jaata hai aur naya item add karna ho.

- **Cache Size**: Suppose 5 items (A, B, C, D, E)
- **Rule**: Jo item sabse **lambe time se access nahi hua**, wahi remove hoga.

🔹 Step-by-Step Example:

Initial Cache (full):

```
[A] [B] [C] [D] [E]
```

1. **Access B** → B ab recently used ho gaya

```
[A] [C] [D] [E] [B]
```

2. **Access D** → D ab recently used ho gaya

```
[A] [C] [E] [B] [D]
```

3. **Add new item F** → Cache full, remove **least recently used** (A)

```
[C] [E] [B] [D] [F]
```

✅ **Key Point:** LRU hamesha "sabse purana access" item remove karta hai.

🔹 Pros & Cons:

- **Pros**: Simple, intuitive, widely used
- **Cons**: Agar access pattern repeat na ho → kabhi-kabhi inefficient ho sakta hai

🔹 Real-life Example:

- Browser cache: Last recently visited website ka data remove karna
- Redis cache: Frequently used session data ko retain karna

---

2. MRU - Most Recently Used 🔄

Definition: Remove the **most recently accessed** item when cache is full.

Working Figure & Explanation:

```
Initial Cache: [A] [B] [C] [D] [E]   ← Most Recent on Right


Step 1: Access C

  → C becomes Most Recently Used (moves to right)

  Cache: [A] [B] [D] [E] [C]


Step 2: Add New Item F

  → Cache is full, remove the Most Recently Used (rightmost item, C)

  → Add F at the end

  Cache: [A] [B] [D] [E] [F]
```

Step-by-Step Summary:

1. Track the most recently used item in cache.
2. When new item comes and cache is full → remove **most recently used item**.
3. Add the new item in its place.

Pros: ✅ Works for special cases
Cons: ❌ Removes hot items, ignores history

Use Case: Special scenarios like **stack caching**

---

3. LFU - Least Frequently Used 📊

Definition: Remove the item that has been **used the least number of times** when cache is full.

Working Figure & Explanation:

```
Initial Cache with usage counts:

A(3) B(2) C(1) D(1) E(4)   ← Number in () = frequency


Step 1: Access C

  → Increase frequency of C by 1

  Cache: A(3) B(2) C(2) D(1) E(4)


Step 2: Add New Item F

  → Cache full, remove the **least frequently used item**

     → D(1) and C(2) → D has lowest frequency → remove D

  → Add F with frequency 1

  Cache: A(3) B(2) C(2) E(4) F(1)
```

Step-by-Step Summary:

1. Track frequency of each item in cache.
2. When cache is full → remove item with **lowest frequency**.
3. Add the new item and set its frequency to 1.

Pros: ✅ Frequency-based removal ensures popular items stay
Cons: ❌ Needs counting of accesses → overhead
Use Case: Popular content caching (e.g., trending articles, videos)

---

4. FIFO - First In First Out ⏳

Definition: Remove the item that **entered first** in the cache.

Figure & Working:

```
Initial Cache: [E] [A] [B] [C] [D]   ← E is the first inserted

New Item F comes in:

Step 1: Cache is full → Remove first item (E)

Step 2: Add F at the end

Final Cache: [A] [B] [C] [D] [F]
```

Pros: ✅ Simple, easy to implement
Cons: ❌ May remove frequently used items, can increase cache miss
Use Case: Simple queue-based caching

---

5. RANDOM - Random Eviction 🎲

Definition: Remove a **random item** from the cache.

Figure & Working:

```
Initial Cache: [A] [B] [C] [D] [E]


Step 1: Cache is full → Randomly remove one item (C)

Cache after removal: [A] [B] [D] [E]


Step 2: New Item F comes in → Add at end

Final Cache: [A] [B] [D] [E] [F]
```

Pros: ✅ Easy to implement
Cons: ❌ Unpredictable behavior
Use Case: Equal access probability caching

🔹 PART 5: REDIS & PRACTICAL IMPLEMENTATIONS

Redis Features:

- TTL support ⏱️
- Key-Value storage 🔑
- Fast in-memory access ⚡

Use Cases:

```
{
  "auth_token": "user_session_data",
  "user_id": "user_profile_info",
  "session_id": "login_session_data"
}
```

- Authentication tokens → 1 hr expiry 🔐
- Session data → 30 min expiry 🕒
- Temporary user data 🗃️

Applications:

- Load balancers ⚖️
- CDN 🌐
- API gateway 🔌
- Proxies 🔁
- Server-side caching 💻
- Client-side caching → cookies 🍪

🚀 PERFORMANCE COMPARISON

Database vs Cache:

```
Database Access: 200 ms | Storage: 2000 TB | Permanent Storage
Cache Access:    20 ms | Storage: 1 GB   | Temporary, Fast Access
```

🎯 MESSAGING QUEUE COMPARISON

- **Kafka:** Pull-based, Partitioning: Hash %, Round Robin, Topic-based routing, High Throughput ⚡
- **RabbitMQ:** Push-based, Exchange-based routing (Direct, Fan-out, Topic), Flexible Routing 🔁

🎉 ✅ FINAL ENGAGING & COMPLETE NOTES READY!

- Har word, diagram, aur figure included ✅
- Eviction policies ab **full form + figure + definition + pros/cons + use case** ke saath ✅
- Perfect for **revision, interviews, practical implementation**