How to Mitigate Hot Partitions in a Kafka Topic

Hot partitions in Kafka topics occur when the partitioning strategy causes one or a few partitions to receive a disproportionately high volume of messages, leading to imbalance and performance bottlenecks.

1.0 Strategies to Mitigate Hot Partitions

1.1. Review Partition Key Strategy

- **Root cause**: Often, the key used for partitioning has low cardinality or skewed distribution.
- **Mitigations**:
  - Choose a key with higher cardinality (e.g., `user_id` instead of `country`).
  - Hash or salt the key to increase distribution randomness.
  - Use **composite keys** (e.g., `user_id + timestamp`) to spread records more evenly.

1.1.1 What is meant by cardinality?

- High cardinality → many unique values.
  - Example: user_id in a consumer app with millions of users.
  - Each partition key gets spread across many users, so distribution is usually balanced.
- Low cardinality → few unique values, repeated often.
  - Example: country_code in a global app → maybe only ~200 possible values.
  - If you have 12 partitions but only 5 country codes dominate (like US, IN, BR, CN, UK), those partitions get overloaded while others stay underutilized → hot partitions.

1.1.2 What is meant by Hashing or Salting?

What is meant by Hashing or Salting? Is *changing the key slightly* so it distributes more evenly across partitions. Let's break down both:

1.1.2.1 Hashing the Key

- Kafka's **default partitioner** already hashes the key (using `hash(key) % partitionCount`).

- The **problem** comes when the key has **low cardinality** (few unique values) → the hash result is still limited.
- Example:
  - Key = `country_code` (only `US`, `IN`, `BR`).
  - `hash("US") % 12` → always maps to the same partition.
  - So "US" traffic may overload a single partition → hot partition.

👉 Hashing works **best when keys are high-cardinality** (lots of unique values).

1.1.2.2 Salting the Key

- **Salting** = adding some **extra random (or controlled) value** to the key before partitioning.
- This increases the number of unique key values, spreading load more evenly.
- Example:
  - Instead of using just `user_id = 12345` as the key:
  - Add a small random number (or hash prefix/suffix):
    - `12345#1`, `12345#2`, `12345#3`
  - Now Kafka's partitioner sees more unique keys → spreads them across partitions.

⚖️ Trade-Off

- **Pro:** Balances load and prevents hot partitions.
- **Con:** Ordering per original key is broken because now `user_id 12345` messages may land in different partitions.
  - If **ordering matters per user_id**, you can use a **controlled salt**:
    - Example: `salt = hash(user_id) % N`
    - Key = `user_id + salt`
    - Guarantees the same user_id always hashes to the same set of salted keys.

1.2 Use Kafka's Default Partitioner or Custom Partitioner

- The **default partitioner** hashes the key, which usually balances data if the key distribution is good.
- If your use case requires custom routing, implement a **Custom Partitioner** that:
  - Adds randomness when skew is detected.
  - Ensures related records still land in the same partition when needed.

1.3 Increase Partition Count (With Caution)

- More partitions can reduce per-partition load.

- **Caution**:
  - Increasing partitions affects ordering guarantees (ordering is per-partition).
  - Repartitioning an existing topic does not rebalance old data automatically.
  - Consumers may need to handle more partitions.

1.4 Leverage Kafka Streams or Flink for Repartitioning

- Use **Kafka Streams** `repartition()` or **Flink SQL** to redistribute data more evenly across a new topic.
- This is useful when you can't control the producer's partitioning.

1.5 Apply a Load-Balancing Technique

- **Key salting**: Append a small random number or hash suffix to the key (`user_id#1`, `user_id#2`).
- **Round-robin partitioning**: If key-based affinity isn't required, allow records to be distributed randomly across partitions.

1.6 Monitor and Tune

- Use metrics from **Confluent Metrics API** or **JMX metrics** to detect hot partitions.
- Automate alerts when lag or throughput imbalance is detected.
- Regularly revisit partitioning as data characteristics change.

✅ **Rule of Thumb**: If **ordering per key matters**, use salting or composite keys carefully (preserving grouping logic). If **ordering doesn't matter**, random or round-robin distribution is usually the simplest fix.