High-Throughput Kafka-to-MySQL Processor with Redis Backup

1. Introduction

This document proposes the design of a **Delivery Tracking Microservice** to efficiently handle and persist a high volume of Kafka messages while minimizing the load on the MySQL database.

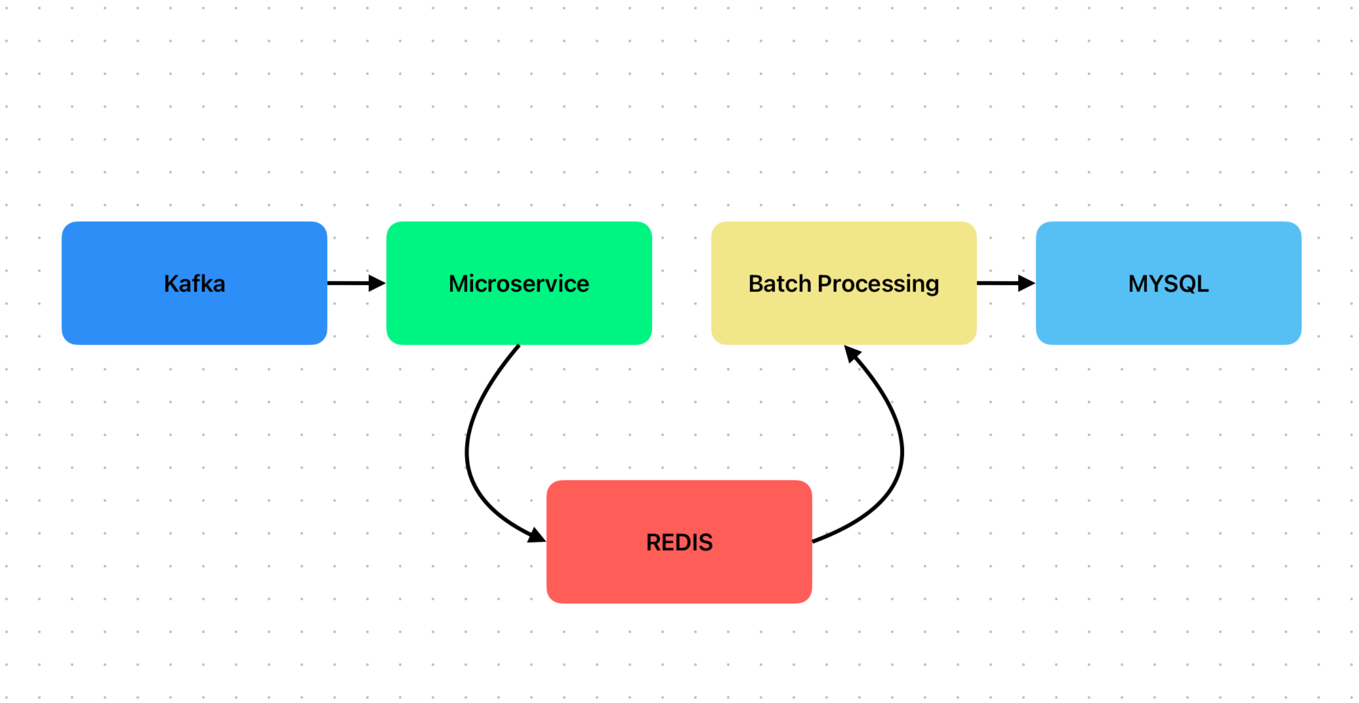
The core objective is to **reduce direct database hits** by introducing a buffering mechanism using **in-memory structures and Redis**, followed by **batch inserts** into MySQL at fixed intervals (approximately every 30 seconds).

1.1 Scope & Assumption

- Handle ~463 messages/sec across distributed instances (40 Million messages per day)
- Minimize DB hits using Redis-backed batching

2. System Architecture

2.1 High-Level Diagram



1. **Kafka (Message Ingestion Layer)**
    - Kafka serves as the primary ingestion point, receiving messages related to delivery tracking in real time.
    - The microservice subscribes to relevant Kafka topics to consume these messages.
2. **Microservice (Processing Layer)**
    - A Spring Boot-based microservice that processes incoming Kafka messages.
    - Messages are buffered **In-memory** and **Redis** for short-term storage.
    - Ensures fault tolerance by persisting unprocessed messages in Redis in case of failures.
3. **Redis (Temporary Buffer Storage)**
    - Acts as a distributed caching layer to hold messages temporarily.
    - Ensures resilience and prevents data loss during unexpected failures.
    - Supports quick retrieval for batch processing.
4. **Batch Processing (Database Interaction Layer)**
    - Periodically (every 30 seconds), the batch processor fetches messages from **Redis and In-Memory buffers**.
    - Performs batch inserts into MySQL, reducing individual database hits.
    - Ensures efficient write operations, minimizing transaction overhead.

5. **MySQL (Persistent Storage Layer)**
    - Stores the final processed delivery tracking data.
    - Designed to support optimized bulk inserts for high-throughput scenarios.

3. Design Highlights

3.1 In-Memory Buffering

- Holds up to `1000` messages per pod
- Triggers batch insert either:
    - When buffer size hits 1000 or every 30 seconds

3.2 Redis Backup

- Key format: `msg:buffer:<instance-id>:<uuid>`

3.3 MySQL Batch Insert

- JDBC batch insert via `JdbcTemplate`
- Batches of up to 1000 rows

4. Crash Recovery

Failure	Solution
Pod crash (OOM etc.)	Restore from Redis on startup
MySQL outage	Retry with exponential backoff
Redis failure	Fallback to in-memory (data loss risk)
Message duplication	Use idempotent inserts

5. Redis & Memory Estimations

5.1 Message Size Calculation

- Each message: 500 characters

- Java String overhead: ~40 bytes per string
- Redis storage overhead: ~100 bytes per key-value pair
- Estimated size per message: ~700 bytes

5.2 Memory Requirements

5.2.1 For 40 million messages/day:

- Messages per second: ~463 messages/second (40M / 86400)
- Batch size: 1000 messages
- Batches per second: ~0.46 (every ~2 seconds)

5.2.2 In-Memory Buffer:

- Max buffer size: 1000 messages
- Memory needed: 1000 * 700 bytes = ~0.7 MB per instance

5.2.3 Redis Requirements:

- Worst case (all messages in Redis): 40M * 700 bytes = ~28GB
- Practical case (buffer overflow only):
    - If we have 10 consumer instances
    - Each might have up to 1000 messages in Redis
    - Total: 10 * 1000 * 700 bytes = ~7MB

5.3 Recommended Redis Sizing

- Start with 1GB Redis instance
- Enable Redis persistence for crash recovery
- Consider Redis Cluster if high availability is critical

6. Conclusion

✅ High Throughput (40M messages/day)
✅ Fault Tolerance with Redis
✅ Optimized MySQL inserts
✅ Kubernetes Ready
✅ Resilient and Observable