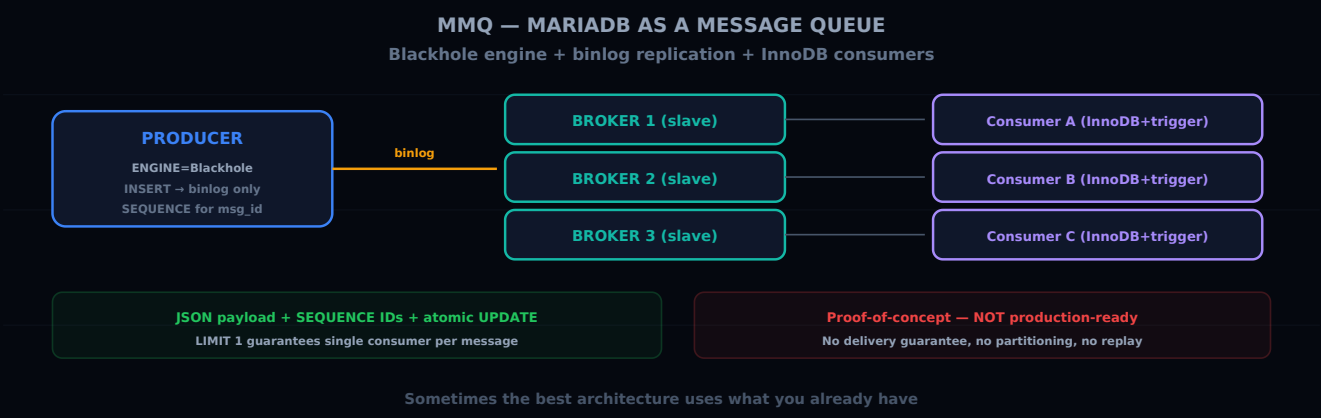
MMQ: MariaDB as a Message Queue

Sylvain ARBAUDIE · November 5, 2024

MARIADB MESSAGE-QUEUE BLACKHOLE REPLICATION



The Crazy Idea

What if we built a message queue using only MariaDB? Not Kafka, not RabbitMQ, not Redis Streams. Just MariaDB, its storage engines, and binlog replication.

This is a thought experiment, a proof-of-concept. The goal is not to replace proven messaging solutions, but to demonstrate the flexibility of MariaDB / MySQL's architecture and explore lesser-known patterns.

The MMQ Architecture

MMQ (MariaDB Message Queue) relies on three native components:

1. **Blackhole engine**: a storage engine that accepts INSERTs but stores nothing. Data "disappears" — except it is recorded in the binlog.
2. **Binlog replication**: MariaDB's native replication mechanism that propagates events from one server to another.
3. **InnoDB tables + triggers**: for message consumption and tracking.

The Producer (Publisher)

The producer server has a Blackhole table serving as the entry point:

```
CREATE TABLE message_queue (
    msg_id BIGINT NOT NULL,
    topic VARCHAR(255) NOT NULL,
    payload JSON NOT NULL,
    created_at DATETIME(6) DEFAULT NOW(6)
) ENGINE=Blackhole;
```

When an application publishes a message:

```
INSERT INTO message_queue (msg_id, topic, payload)
VALUES (
    NEXT VALUE FOR msg_sequence,
    'order.created',
    '{"order_id": 12345, "customer": "acme", "total": 99.99}'
);
```

The Blackhole engine writes nothing to disk. But the INSERT is recorded in the server's binlog. That is the magic of Blackhole: it participates in the binlog without consuming storage.

Sequences for Identifiers

MariaDB supports sequences (since version 10.3), offering unique identifiers without the cost of an AUTO_INCREMENT with locking:

```
CREATE SEQUENCE msg_sequence
    START WITH 1
    INCREMENT BY 1
    CACHE 1000;
```

`CACHE 1000` pre-allocates 1,000 values in memory, reducing disk access and locks.

The Broker (Relay)

The broker is a MariaDB server configured as a slave of the producer. It receives binlog events and replicates them. This is the distribution mechanism.

For fanout (one message to multiple consumers), you can have multiple slaves of the same master — each slave receives an independent copy of all messages.

```
Producer (Blackhole) → binlog → Broker 1 (slave)
                              → Broker 2 (slave)
                              → Broker 3 (slave)
```

The Consumer

Each consumer has an InnoDB table storing received messages and a consumption tracking mechanism:

```
CREATE TABLE consumed_messages (
    msg_id BIGINT PRIMARY KEY,
    topic VARCHAR(255),
    payload JSON,
    created_at DATETIME(6),
    consumed_at DATETIME(6) DEFAULT NULL,
    consumer_id VARCHAR(100) DEFAULT NULL
) ENGINE=InnoDB;
```

A trigger transforms replicated INSERTs into usable rows:

```
CREATE TRIGGER trg_message_arrived
BEFORE INSERT ON message_queue
FOR EACH ROW
BEGIN
    INSERT INTO consumed_messages (msg_id, topic, payload, created_at)
    VALUES (NEW.msg_id, NEW.topic, NEW.payload, NEW.created_at);
END;
```

Consumption is done via an atomic query:

```
UPDATE consumed_messages
SET consumed_at = NOW(6),
    consumer_id = 'worker-01'
WHERE consumed_at IS NULL
  AND topic = 'order.created'
ORDER BY msg_id ASC
LIMIT 1;
```

`LIMIT 1` combined with the atomic `UPDATE` ensures only one consumer processes each message (no double consumption).

JSON Messages

MariaDB's native JSON format (since 10.2) allows structuring messages with rich payloads:

```
INSERT INTO message_queue (msg_id, topic, payload) VALUES (
    NEXT VALUE FOR msg_sequence,
    'user.profile.updated',
    JSON_OBJECT(
        'user_id', 42,
        'changes', JSON_ARRAY(
            JSON_OBJECT('field', 'email', 'old', 'old@mail.com', 'new', 'new@mail.com'),
            JSON_OBJECT('field', 'name', 'old', 'John', 'new', 'Jonathan')
        ),
        'timestamp', NOW(6)
    )
);
```

The Limitations (And There Are Many)

Let us be clear: MMQ is a concept, not a production-ready solution.

No reliable delivery guarantee. If replication breaks, messages are lost (or delayed). No native retry mechanism.

No partitioning. All messages flow through a single binlog. No per-topic distribution like Kafka.

No replay. Once consumed, a message cannot be easily replayed (unless you keep binlogs on the producer).

Replication latency. Replication latency adds delay between publication and message availability. Acceptable for async, not for real-time.

No distributed acknowledgement. The producer does not know if the consumer processed the message.

Why It Is Interesting Anyway

Despite its limitations, this pattern demonstrates important concepts:

1. **The binlog as an event stream.** The MariaDB / MySQL binlog is an ordered, durable, replicable event stream. Conceptually close to a Kafka log.
2. **The Blackhole engine as an adapter.** Blackhole allows "publishing" without storing, using the binlog as a transport channel.
3. **Replication as a distribution mechanism.** Multi-slave replication provides native fanout with no extra configuration.
4. **The database as versatile infrastructure.** If you already have MariaDB in production, you already have the infrastructure for simple messaging.

For simple use cases — internal notifications between services, event auditing, event replication between sites — MMQ may be sufficient without adding an extra infrastructure component.

Conclusion

MariaDB as a message queue: a crazy idea, a fun proof-of-concept, and a demonstration of Blackhole engine + binlog replication flexibility. Do not use it in production for critical messaging. But keep the concept in mind — sometimes the best architecture is one that uses what you already have.

This article was originally published on Medium.