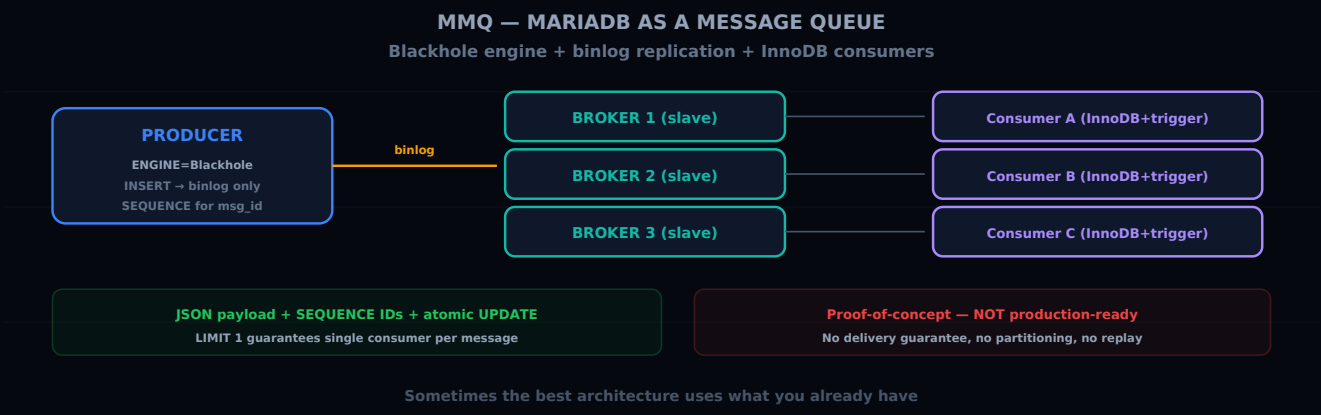
MMQ : MariaDB comme file de messages

Sylvain ARBAUDIE · 5 novembre 2024

MARIADB MESSAGE-QUEUE BLACKHOLE REPLICATION



L'idée folle

Et si on construisait une file de messages (message queue) en utilisant uniquement MariaDB ? Pas Kafka, pas RabbitMQ, pas Redis Streams. Juste MariaDB, ses moteurs de stockage, et la réplication binlog.

C'est un exercice de pensée, un proof-of-concept. L'objectif n'est pas de remplacer des solutions de messaging éprouvées, mais de démontrer la flexibilité de l'architecture de MariaDB / MySQL et d'explorer des patterns méconnus.

L'architecture MMQ

MMQ (MariaDB Message Queue) repose sur trois composants natifs :

1. **Blackhole engine** : un moteur de stockage qui accepte les INSERT mais ne stocke rien. Les données "disparaissent" — sauf qu'elles sont enregistrées dans le binlog.
2. **Réplication binlog** : le mécanisme natif de réplication de MariaDB qui propage les événements d'un serveur à l'autre.
3. **Tables InnoDB + triggers** : pour la consommation et le suivi des messages.

Le producteur (Publisher)

Le serveur producteur possède une table Blackhole qui sert de point d'entrée :

```
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;
```

Quand une application publie un 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}'
);
```

Le moteur Blackhole n'écrit rien sur disque. Mais l'INSERT est enregistré dans le binlog du serveur. C'est la magie du Blackhole : il participe au binlog sans consommer de stockage.

Les séquences pour les identifiants

MariaDB supporte les séquences (depuis la version 10.3), qui offrent des identifiants uniques sans le coût d'un AUTO_INCREMENT avec verrouillage :

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

Le `CACHE 1000` pré-alloue 1 000 valeurs en mémoire, réduisant les accès disque et les verrous.

Le broker (Relay)

Le broker est un serveur MariaDB configuré comme slave du producteur. Il reçoit les événements binlog et les réplique. C'est le mécanisme de distribution.

Pour un fanout (un message vers plusieurs consommateurs), on peut avoir plusieurs slaves du même master — chaque slave reçoit une copie indépendante de tous les messages.

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

Le consommateur (Consumer)

Chaque consommateur a une table InnoDB qui stocke les messages reçus et un mécanisme de suivi de la consommation :

```
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;
```

Un trigger transforme les INSERT répliqués en lignes exploitables :

```
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;
```

La consommation se fait par une requête atomique :

```
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;
```

Le `LIMIT 1` combiné avec l'`UPDATE` atomique garantit qu'un seul consommateur traite chaque message (pas de double consommation).

Les messages en JSON

Le format JSON natif de MariaDB (depuis 10.2) permet de structurer les messages avec des payloads riches :

```
-- Publier un événement complexe
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)
    )
);
```

Les limites (et elles sont nombreuses)

Soyons clairs : MMQ est un concept, pas une solution production-ready.

Pas de garantie de livraison fiable. Si la réplication tombe, les messages sont perdus (ou retardés). Pas de mécanisme de retry natif.

Pas de partitionnement. Tous les messages passent par un seul binlog. Pas de distribution par topic comme Kafka.

Pas de replay. Une fois consommé, un message ne peut pas être rejoué facilement (sauf à conserver les binlogs sur le producteur).

Latence de réplication. La latence de la réplication ajoute un délai entre la publication et la disponibilité du message. C'est acceptable pour de l'asynchrone, mais pas pour du temps réel.

Pas d'acknowledgement distribué. Le producteur ne sait pas si le consommateur a traité le message.

Pourquoi c'est intéressant quand même

Malgré ses limites, ce pattern démontre des concepts importants :

1. **Le binlog comme event stream.** Le binlog de MariaDB / MySQL est un flux d'événements ordonné, durable et réplicable. C'est conceptuellement proche d'un log Kafka.
2. **Le moteur Blackhole comme adaptateur.** Le Blackhole permet de "publier" sans stocker, utilisant le binlog comme canal de transport.
3. **La réplication comme mécanisme de distribution.** La réplication multi-slave offre un fanout natif sans configuration supplémentaire.
4. **La base de données comme infrastructure polyvalente.** Si vous avez déjà MariaDB en production, vous avez déjà l'infrastructure pour un messaging simple.

Pour les cas d'usage simples — notifications internes entre services, audit événementiel, réplication d'événements entre sites — MMQ peut être suffisant sans ajouter un composant d'infrastructure supplémentaire.

Conclusion

MariaDB comme file de messages : une idée folle, un proof-of-concept amusant, et une démonstration de la flexibilité du moteur Blackhole + réplication binlog. Ne l'utilisez pas en production pour du messaging critique. Mais gardez le concept en tête — parfois, la meilleure architecture est celle qui utilise ce que vous avez déjà.

Cet article a été initialement publié sur Medium.