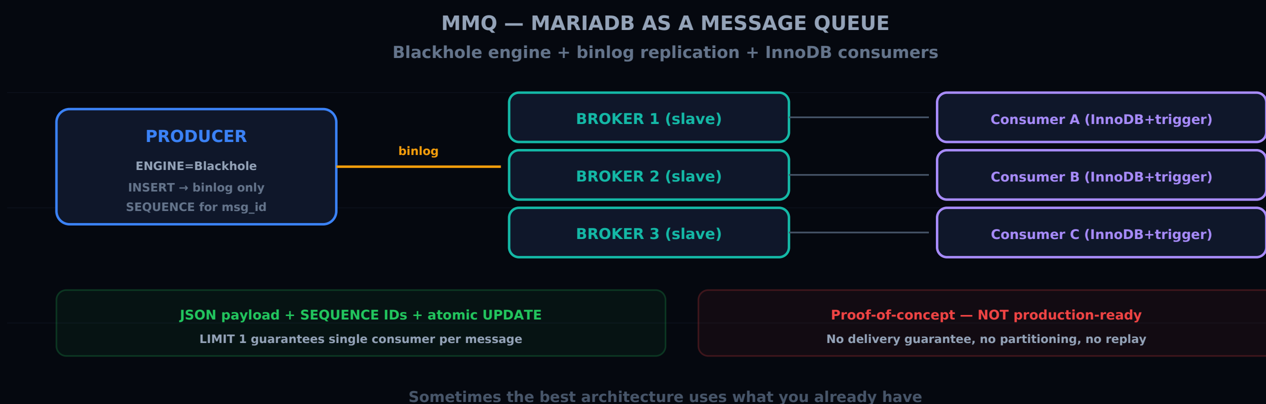
MMQ: MariaDB jako kolejka wiadomości

Sylvain ARBAUDIE · November 5, 2024

MARIADB MESSAGE-QUEUE BLACKHOLE REPLICATION



Szalony pomysł

A gdyby zbudować kolejkę wiadomości (message queue) używając wyłącznie MariaDB? Nie Kafka, nie RabbitMQ, nie Redis Streams. Tylko MariaDB, jej silniki przechowywania i replikacja binlog.

To ćwiczenie myślowe, proof-of-concept. Celem nie jest zastąpienie sprawdzonych rozwiązań messagingu, lecz demonstracja elastyczności architektury MariaDB / MySQL i eksploracja mało znanych wzorców.

Architektura MMQ

MMQ (MariaDB Message Queue) opiera się na trzech natywnych komponentach:

1. **Silnik Blackhole**: silnik przechowywania, który akceptuje INSERT, ale nic nie przechowuje. Dane „znikają" — z wyjątkiem tego, że są rejestrowane w binlogu.
2. **Replikacja binlog**: natywny mechanizm replikacji MariaDB propagujący zdarzenia z jednego serwera na drugi.
3. **Tabele InnoDB + triggery**: do konsumpcji i śledzenia wiadomości.

Producent (Publisher)

Serwer producenta posiada tabelę Blackhole służącą jako punkt wejścia:

```
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;
```

Gdy aplikacja publikuje wiadomość:

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

Silnik Blackhole nie zapisuje niczego na dysku. Ale INSERT jest rejestrowany w binlogu serwera. To magia Blackhole: uczestniczy w binlogu bez zużywania pamięci masowej.

Sekwencje dla identyfikatorów

MariaDB obsługuje sekwencje (od wersji 10.3), które zapewniają unikalne identyfikatory bez kosztu AUTO_INCREMENT z blokowaniem:

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

`CACHE 1000` prealokuje 1000 wartości w pamięci, redukując dostępy do dysku i blokady.

Broker (Relay)

Broker to serwer MariaDB skonfigurowany jako slave producenta. Odbiera zdarzenia binlog i je replikuje. To mechanizm dystrybucji.

Dla fanout (jedna wiadomość do wielu konsumentów) można mieć wielu slave'ów tego samego mastera — każdy slave otrzymuje niezależną kopię wszystkich wiadomości.

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

Konsument (Consumer)

Każdy konsument ma tabelę InnoDB przechowującą odebrane wiadomości oraz mechanizm śledzenia konsumpcji:

```
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;
```

Trigger przekształca replikowane INSERT w użyteczne wiersze:

```
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;
```

Konsumpcja odbywa się przez atomowe zapytanie:

```
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` w połączeniu z atomowym `UPDATE` gwarantuje, że tylko jeden konsument przetwarza każdą wiadomość (brak podwójnej konsumpcji).

Wiadomości w JSON

Natywny format JSON w MariaDB (od wersji 10.2) pozwala strukturyzować wiadomości z bogatymi payloadami:

```
-- Publikacja złożonego zdarzenia
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)
    )
);
```

Ograniczenia (a jest ich wiele)

Bądźmy szczerzy: MMQ to koncept, a nie rozwiązanie gotowe na produkcję.

Brak gwarantowanej niezawodnej dostawy. Jeśli replikacja padnie, wiadomości są tracone (lub opóźnione). Brak natywnego mechanizmu retry.

Brak partycjonowania. Wszystkie wiadomości przechodzą przez jeden binlog. Brak dystrybucji po topicach jak w Kafka.

Brak replay. Po skonsumowaniu wiadomość nie może być łatwo odtworzona (chyba że zachowamy binlogi na producencie).

Opóźnienie replikacji. Opóźnienie replikacji dodaje zwłokę między publikacją a dostępnością wiadomości. Jest to akceptowalne dla przetwarzania asynchronicznego, ale nie dla czasu rzeczywistego.

Brak rozproszonego potwierdzenia. Producent nie wie, czy konsument przetworzył wiadomość.

Dlaczego to jest mimo wszystko interesujące

Pomimo swoich ograniczeń, ten wzorzec demonstruje ważne koncepcje:

1. **Binlog jako event stream.** Binlog MariaDB / MySQL to uporządkowany, trwały i replikowalny strumień zdarzeń. Koncepcyjnie jest to bliskie logowi Kafka.
2. **Silnik Blackhole jako adapter.** Blackhole pozwala „publikować" bez przechowywania, wykorzystując binlog jako kanał transportowy.
3. **Replikacja jako mechanizm dystrybucji.** Replikacja multi-slave oferuje natywny fanout bez dodatkowej konfiguracji.
4. **Baza danych jako wszechstronna infrastruktura.** Jeśli masz już MariaDB na produkcji, masz już infrastrukturę do prostego messagingu.

Dla prostych przypadków użycia — wewnętrzne powiadomienia między usługami, audyt zdarzeń, replikacja zdarzeń między lokalizacjami — MMQ może być wystarczający bez dodawania kolejnego komponentu infrastruktury.

Podsumowanie

MariaDB jako kolejka wiadomości: szalony pomysł, zabawny proof-of-concept i demonstracja elastyczności silnika Blackhole + replikacji binlog. Nie używaj tego na produkcji do krytycznego messagingu. Ale zachowaj tę koncepcję w pamięci — czasem najlepsza architektura to ta, która wykorzystuje to, co już masz.

Ten artykuł został pierwotnie opublikowany na Medium.