TDMQ for Pulsar

Best Practices

Product Documentation

Contents

Best Practices
Transaction Reconciliation

Last updated：2024-01-03 14:27:38

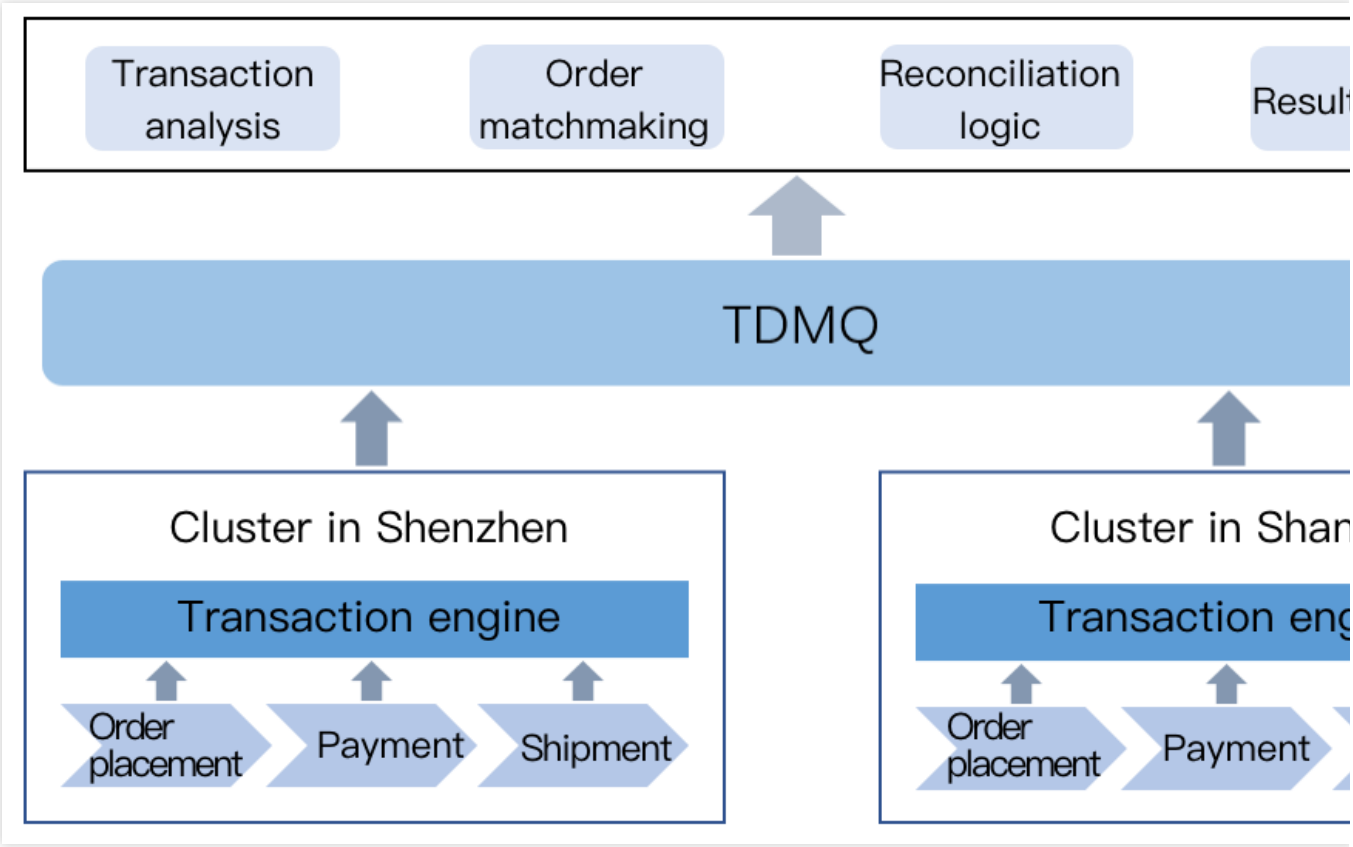
Scenario Description

Reconciliation is an **auxiliary system** required in all billing systems. No matter whether used as a primary or secondary payment system, reconciliation is always necessary during or after payment to ensure billing accuracy. TDMQ for Pulsar can guarantee reconciliation timeliness without affecting the critical path of transactions.

Problems Encountered

1. **System decoupling**
A transaction involves many systems, which need to be **decoupled** so as not to affect each other.
2. **Time difference of data arrival**
There is a **time difference** of data arrival between systems, so it must be ensured that data arriving at different times can be aggregated.
3. **Data consistency**
In order to eliminate exceptional reconciliation results, it must be ensured that data will never be **lost**.
4. **Cross-region data transfer**
Systems are deployed in different regions, so **cross-region** data transfer is involved.

Deployment Architecture Diagram



Problem Solving

TDMQ for Pulsar can be used to solve the problems mentioned above.

1. System decoupling

Intuitively, to implement reconciliation between different systems, messages can be reported directly to the reconciliation system for reconciliation processing. However, this causes some new problems; for example, many existing and future systems need to be connected, which is time-consuming and intrusive to the system process in the production environment. Obviously, such a design is very unreasonable. You can adopt TDMQ for Pulsar and connect different systems to it in a unified manner, so that failures in an individual system will not affect other services.

2. Time difference of data arrival

Reconciliation requires the aggregation of data from different systems. Under normal circumstances, the data arrival times of different systems don't differ significantly, but the processes between systems are always sequential; therefore, after the data is delayed for one system, to implement data aggregation, you need to control the data read speed and thus prevent a large amount of data from simply waiting in the reconciliation system. The temporary message storage feature of TDMQ for Pulsar makes it possible that data sent at the same time arrives at different systems generally at the same time.

3. Data consistency

TDMQ for Pulsar provides highly consistent and reliable data storage to ensure that data will never get lost. In addition, it features a high service availability and automatic failover.

4. Cross-region data transfer

TDMQ for Pulsar offers two schemes to implement data replication between regions and serves as a real-time data replication channel for the business layer.
For scenarios where critical data requires cross-region disaster recovery, TDMQ for Pulsar supports cross-region strong sync that distributes messages in different regions.
For scenarios with low requirements for data consistency, TDMQ for Pulsar provides a cross-region async replication scheme to achieve eventual data consistency in multiple regions.
The two cross-region sync schemes are compared as follows:

Cross-region Sync Scheme	Production Duration	Disaster Recovery Performance	Storage Costs
Cross-region strong consistency	High	High	Low
Cross-region eventual consistency	Low	Low	High

By leveraging TDMQ for Pulsar and real-time computing, you can upgrade daily transaction reconciliation to real-time reconciliation, which allows you to check the transaction accuracy more quickly.

Message Idempotency

Last updated：2024-01-03 14:27:38

Idempotency is a key to the design of distributed systems. If idempotency is not taken into account, a message may be consumed repeatedly in case of a business processing failure, which doesn't meet your business expectations. To avoid such an exception, the consumer of a message queue needs to ensure the idempotency of the received messages based on the unique business key.

What Is Message Idempotency

Definition

The consumption of a message is deemed idempotent if consuming the message multiple times yields the same result as consuming it only once and the duplicate consumption of the same message doesn't have any negative impact on the business system.

Scenario example

Take a bank's payment system as an example. After the consumer consumes a deduction message, the payment system will deduct an amount (say, 1 USD in this example) for the order. If the consumer consumes that deduction message again due to network instability or other reasons, yet only 1 USD is finally deducted and the user has only one deduction record, then the deduction operation is valid because the amount hasn't been deducted multiple times. In this scenario, the whole consumption process is idempotent.

Use Cases

Message duplication caused by message sending

After a message is sent by a producer and is successfully received and persistently stored by the server, it will be resent if the producer hasn’t received the server's acknowledgment. This happens when a client restart or momentary

network disconnection prevents the server and client from communicating. In this case, the same message will be sent twice to the customer, but with different IDs.

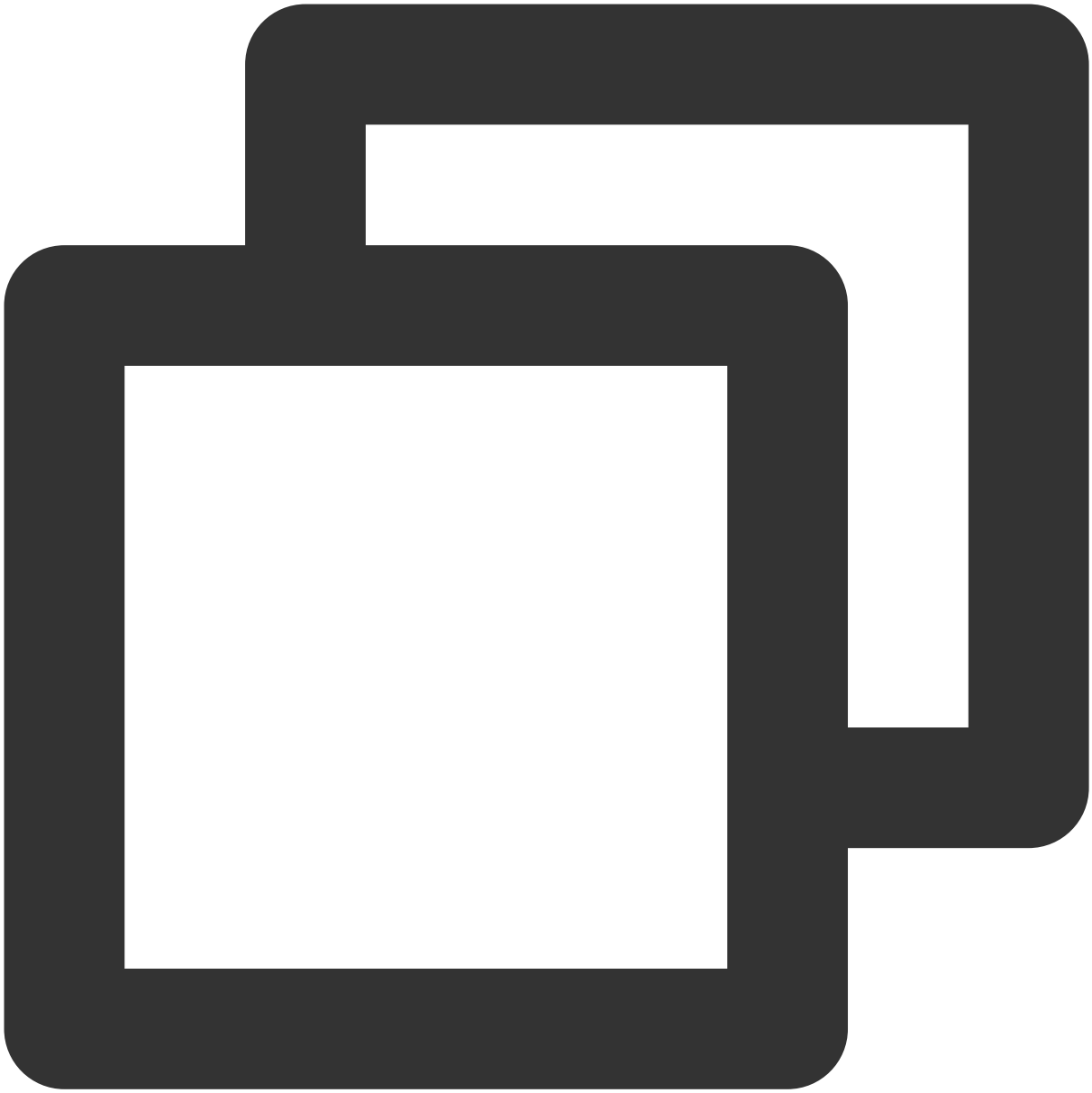
Message duplication caused by message consumption

After a message is consumed by a consumer for related business, it will be consumed again if the consumer fails to send the acknowledgment to the server due to network exception. In this case, the same message will be consumed twice by the consumer, with the same ID.

Solutions

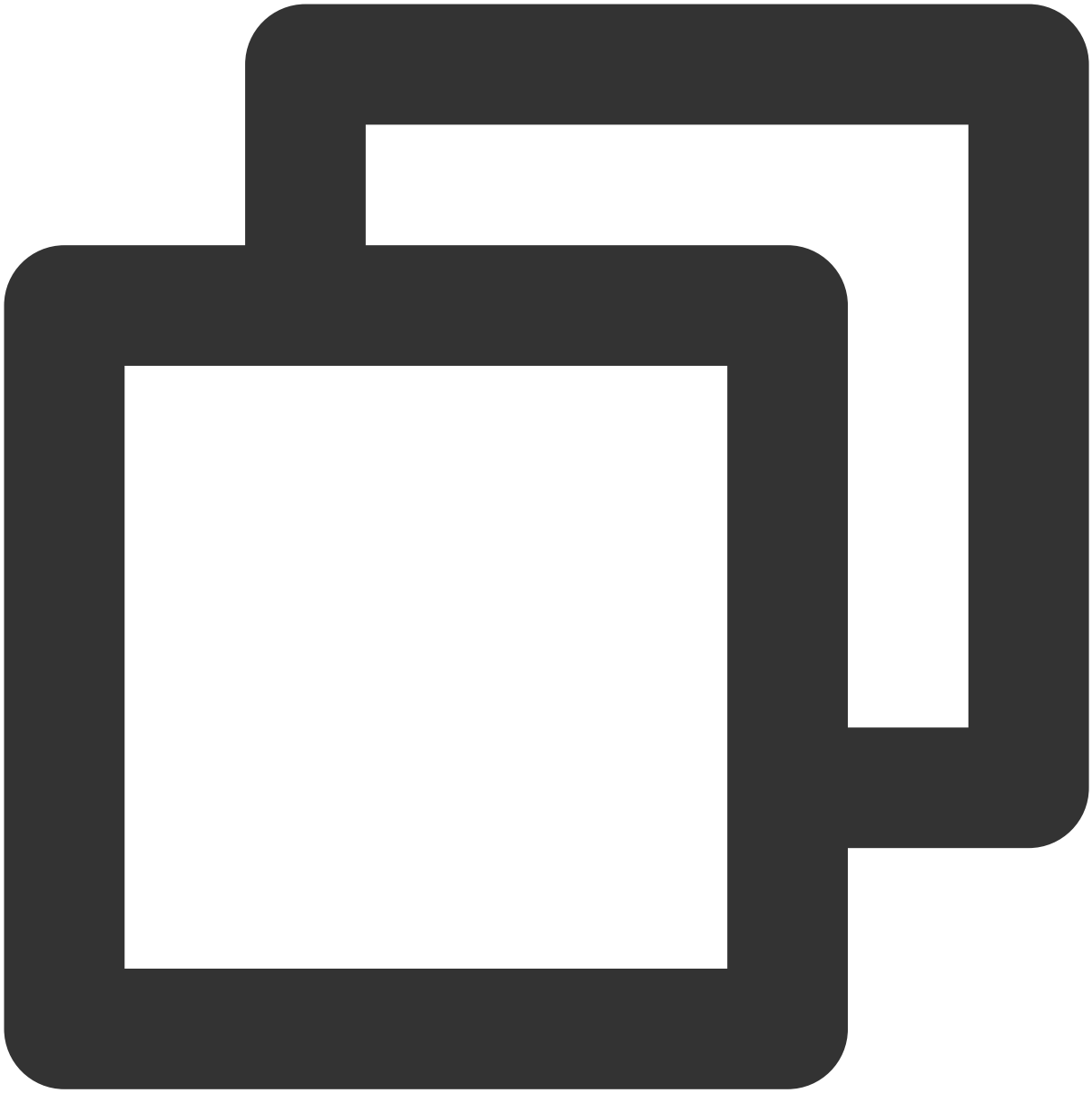
We can see from the above use cases that there are two message duplication scenarios:
The same message but with different IDs;
The same message with the same ID.
Therefore, we recommend that you use the unique business identifier instead of the message ID to implement idempotency. For example, in the payment scenario, you can use the order ID to implement idempotency. After a message is successfully consumed, the order ID of a business can be used to determine whether the business has been processed.

Sample code



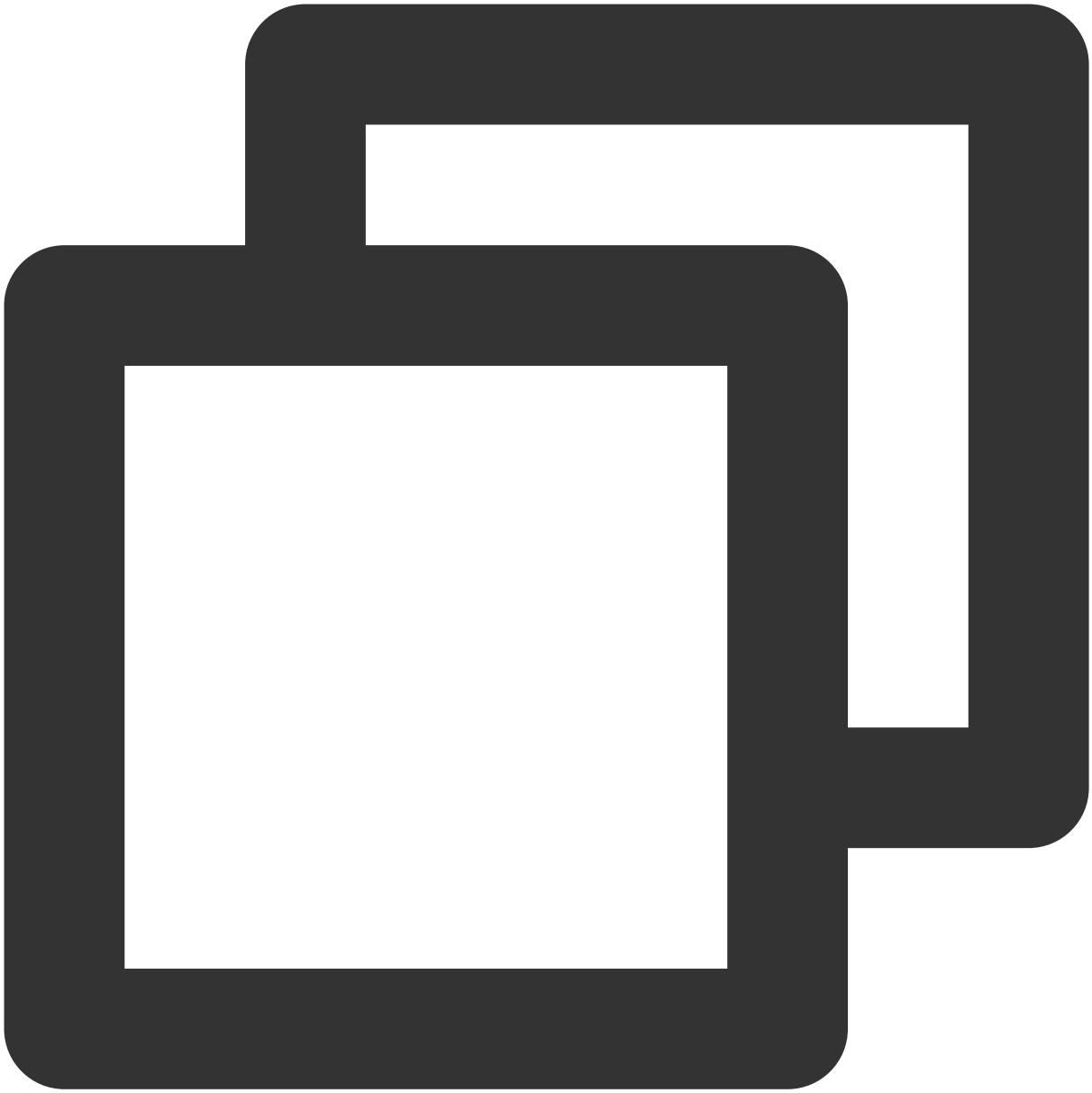
```
public static class Order {
    public String orderId;
    public String orderData;
}
```

Producer:



```
Producer<Order> producer = client.newProducer(Schema.AVRO(Order.class)).create();
producer.newMessage().value(new Order("orderid-12345678", "orderData")).send();
```

Consumer:



```
Consumer<Order> consumer = client.newConsumer(Schema.AVRO(Order.class)).subscribe()
Order order = consumer.receive().getValue();
String key = order.orderId;
```

After obtaining the unique business identifier `orderId`, deduplicate it.

Common Deduplication Methods

Deduplicating data in the database

You can filter duplicate data by adding a deduplication table or unique index in the database. This allows you to ensure idempotency on the business side.
For example, if you need to write order flow messages to an order log table, you can use the order ID or the modification timestamp as the unique index.
When the consumer consumes the same message multiple times, the order log table will be written each time, but only the first time takes effect because of the unique index. This implements idempotency to ensure that the consumption result is the same even in case of repeated consumption.

Setting the globally unique message/task ID

The call trace ID can also be set as a globally unique ID. The producer can add a unique ID to each message when producing messages. The consumer can set a key (which corresponds to the unique ID) in the cache to identify the consumed message. In this way, the consumer can determine whether the message has been consumed when consuming messages.

Message Compression

Last updated：2024-01-03 14:27:38

Background

In Pulsar, a message of over 5 MB cannot be successfully sent. To send such a large message, you need to compress it in the client first.

Processing Large Message in Pulsar

As the default size limit for a single message is 5 MB in Pulsar, the producer will fail to send a message exceeding this limit. You can handle this in the following two ways:
Message chunking: Message chunking enables Pulsar to process large payload messages by splitting the message into chunks at the producer side and aggregating chunked messages at the consumer side.
Message compression: The message size can be compressed by replacing the same character sequences in the message data. Pulsar supports four compression algorithms: LZ4, ZLIB, ZSTD, and Snappy.
We recommend that you compress large messages before sending them.

Compression Algorithm Introduction and Comparison

Introduction

LZ4
LZ4 is a lossless data compression algorithm that consumes a small amount of CPU. It features extremely fast compression/decompression speed.
ZLIB
ZLIB is a common lossless data compression algorithm that can improve network transfer efficiency and network capacity because it can effectively reduce the size of transferred data. As a variant of the Lempel-Ziv compression

algorithm, it can compress data to half the original size or even less. It can be used for data compression and decompression.

ZSTD

ZSTD is a variant of the LZ77 compression algorithm and is based on Huffman coding. It is an effective compression algorithm for different compression scenarios. Compared with other compression algorithms, it compresses data faster and more efficiently because it features real-time encoding. It can guarantee a high compression ratio and high compression speed at the same time.

Snappy

Snappy is a lossless compression algorithm based on LZ77. Its core principle lies in the replacement of the repetitive character strings in a data stream with shorter codes to reduce the stream size.

Comparison

Compression Algorithm	Compression Ratio	Compression Speed	Decompression Speed
ZLIB 1.2.11 -1	2.743	110 MB/sec	400 MB/sec
LZ4 1.8.1	2.101	750 MB/sec	3,700 MB/sec
ZSTD 1.3.4-1	2.877	470 MB/sec	1,380 MB/sec
Snappy 1.1.4	2.091	530 MB/sec	1,800 MB/sec

Throughput: LZ4 > Snappy > ZSTD > ZLIB

Compression ratio: ZSTD > ZLIB > LZ4 > Snappy

Physical resource occupation: Snappy occupies the most network bandwidth while ZSTD occupies the least.

Compression Algorithm Test

Test result

Note:

The following test results are for reference only. The actual compression effect is subject to the specific message content.

Message	Message	Compression	Monitored	Message	Message

Size		Algorithm	Message Size	Compression Duration	Sending Duration
5 MB	Random message body	LZ4 (threshold: 5 MB)	9.95 MB	31 ms	0.049 ms
		ZLIB	7.26 MB	31 ms	0.038 ms
		ZSTD	8.20 MB	31 ms	0.039 ms
		Snappy (threshold: 5 MB)	9.70 MB	33 ms	0.046 ms
6 MB	Random message body	ZLIB (threshold: 6 MB)	8.71 MB	35 ms	0.044 ms
		ZSTD (threshold: 6 MB)	9.84 MB	35 ms	0.046 ms
20 MB	Same message body	LZ4	0.16 MB	41 ms	0.006 ms
		ZLIB	0.20 MB	42 ms	0.006 ms
		ZSTD	0.01 MB	42 ms	0.003 ms
		Snappy	2.47 MB	41 ms	0.021 ms
40 MB	Same message body	LZ4	0.32 MB	123 ms	0.008 ms
		ZLIB	0.39 MB	122 ms	0.008 ms
		ZSTD	0.01 MB	124 ms	0.004 ms
		Snappy	4.95 MB	123 ms	0.036 ms
80 MB	Same message body	LZ4	0.63 MB	241 ms	0.009 ms
		ZLIB	0.39 MB	244 ms	0.01 ms
		ZSTD	0.01 MB	243 ms	0.004 ms
		Snappy (threshold: 80 MB)	9.9 MB	243 ms	0.056 ms
160 MB	Same message body	LZ4	1.26 MB	484 ms	0.013 ms
		ZLIB	1.56 MB	479 ms	0.016 ms
		ZSTD	0.03 MB	481 ms	0.004 ms

320 MB	Same message body	LZ4	2.5 MB	1,035 ms	0.03 ms
		ZLIB	3.1 MB	1,008 ms	0.027 ms
		ZSTD	0.03 MB	949 ms	0.004 ms
585 MB	Same message body	LZ4	4.59 MB	1,705 ms	0.027 ms
		ZLIB	5.67 MB	1,733 ms	0.03 ms
		ZSTD	0.11 MB	1,722 ms	0.006 ms

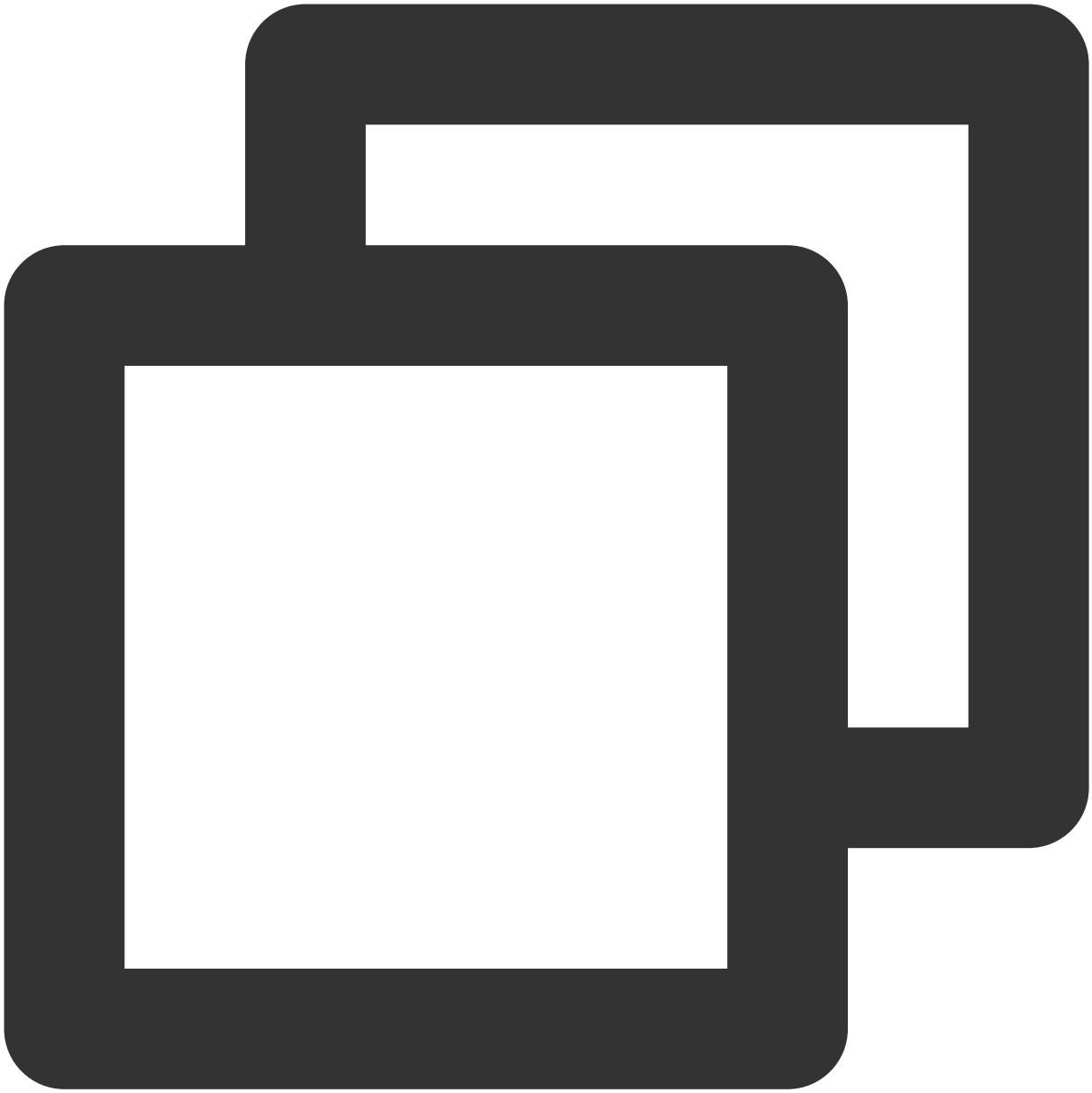
Summary:
For data streams with random message body (non-repetitive strings), the four compression algorithms show low compression ratios. When the message is larger than 5 MB, none of the four algorithms can compress it to less than 5 MB.
For data streams with same message body (repetitive strings), all the compression algorithms show high compression ratios. Especially, LZ4, ZLIB, and ZSTD can compress a message of 5–600 MB to less than 5 MB.

Message compression demo and test

For the demo, see tdmq-sdk-demo.

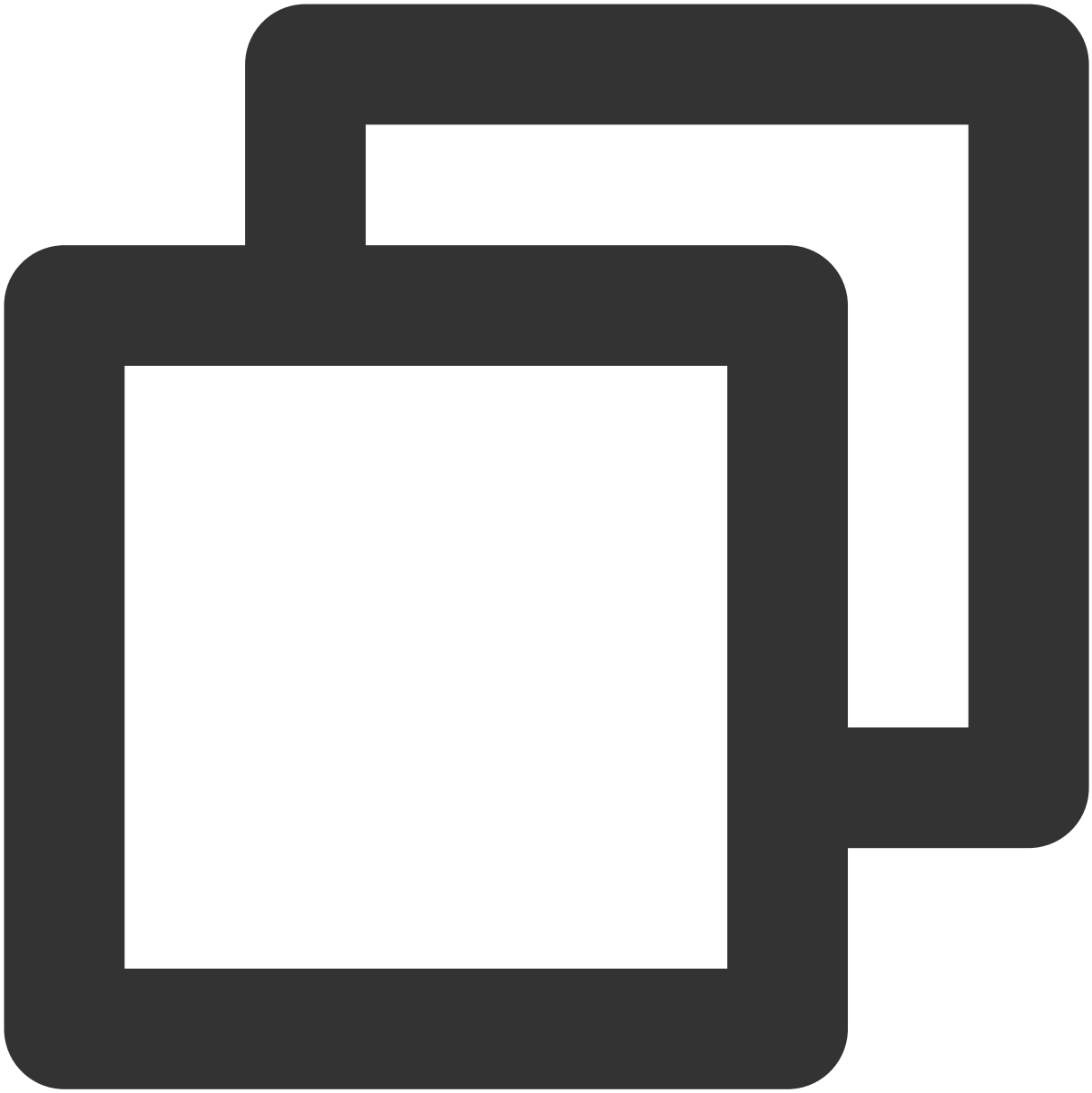
Test

Parameters called by the producer:



```
java -jar tdmq-sdk-demo-1.0-SNAPSHOT-jar-with-dependencies.jar pulsar://xxxx:6650
eyJrZXlJZCI6ImRlZmF1bHRfa2V5SWQiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzdXBlcnVzZXIifQ.dYc
pulsar-78ra8ownxb7d/BigMSGSpace/BigMSGTopic subname 1 500 0 1 20480 1 0
```

Parameters called by the consumer:



```
java -jar tdmq-sdk-demo-1.0-SNAPSHOT-jar-with-dendencies.jar pulsar://xxxx:6650
eyJrZXlJZCI6ImRlZmF1bHRfa2V5SWQiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzdXBlcnVzZXIifQ.dYc
pulsar-92d7w2mjwmv9/BigMessSpace/BigMessTopic subname 1 500 1
```