MongoShake Detailed Documentation

This article is a detailed document about MongoShake, including introduction, application scenarios, function introduction, architecture design, and user use cases.

## Introduction

Mongo-Shake is a universal platform for services based on MongoDB's oplog. It reads the Oplog operation logs of a MongoDB cluster and replicates MongoDB data, and subsequently implements specific requirements through operation logs. Logs can provide a lot of scene-based applications. For this reason, we have considered MongoShake as a universal platform-based service when designing. Through the operation logs, we provide log data subscriptions to consume PUB/SUB functions and can be flexibly connected to adapt to different scenarios (such as log subscription, data center synchronization, cache asynchronous elimination, etc.) through SDK, Kafka, MetaQ, etc. Cluster data synchronization is a core application scenario. Synchronization is achieved through playback after grabbing oplogs.

## Application Scenario

1. Asynchronous replication of MongoDB data between clusters eliminates the need for double write costs.
2. Mirror backup of MongoDB cluster data. (Not support in this open source version)
3. Log offline analysis.
4. Log subscription.
5. Cache synchronization.

Through the results of the log analysis, it is known which caches can be eliminated and which caches can be preloaded to prompt the cache to be updated.

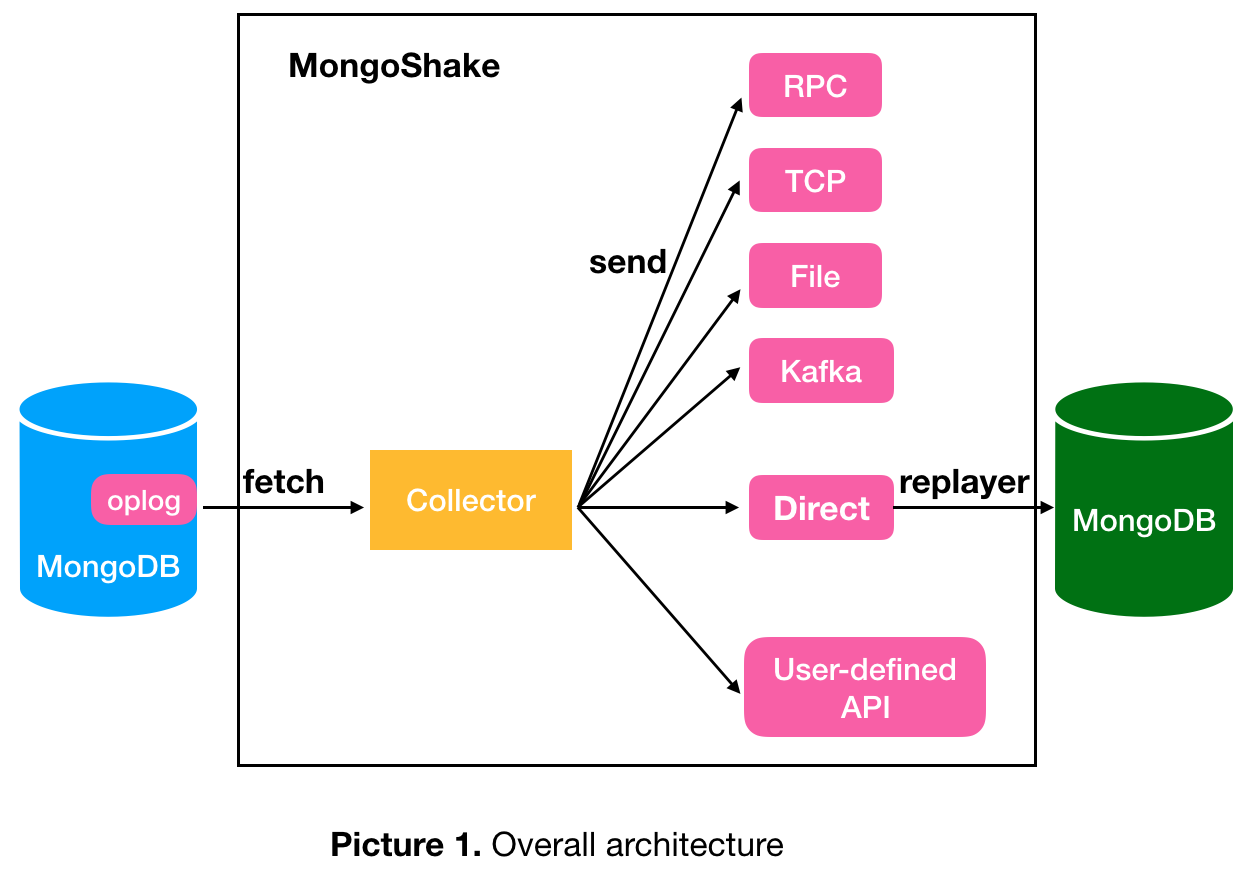
1. Monitor base on log.

## Features

MongoShake fetches oplog from source mongo database, and replays in the target mongo database or sends to other ends in different tunnels. There are the following types of existing tunnels:

* Direct. Write into target MongoDB directly.
* RPC. Connected via net/rpc in go.
* TCP. Connected via tcp.
* File. Connected via file.
* Kafka. Connected via Kafka channel.
* Mock. Used in testing, discarding all data instead of writing to tunnel.

Users can also define their own tunnel type which is pluggable. The following figure shows the general data flow.



The source can be either single mongod, replica set or sharding while target can be mongod or mongos. If the source is replica set, we suggest fetching data from secondary/hidden to ease the primary pressure. If the source is sharding, every shard should connect to Mongo-Shake. There can be several mongos in the target side to keep high availability, and different data will be hashed and written to different mongos.

* **Parallel Replication**

There are three options for parallel replication that we call “shad\_key”: “id”, “collection” and “auto”. “id” means the concurrency granularity is document while “collection” means the granularity is collection(table). “auto” option is decided by if there has unique index of any collection, it will change to “collection” with unique index exist otherwise “id”.

* **High Availability**

Mongo-Shake periodically persistent its context into register center which by default is the source database. Currently, the context is checkpoint that marks the position of successfully replay oplog. In this case, when the service is switched or restarted, the new service can continue to provide service by connecting to this API.

Hypervisor mechanism is also supported so that it will restart immediately when dies.

* **Filter**

MongoShake supports filtering database and collection namespace with whitelist and blacklist.

* **Compression**

MongoShake supports oplog compression before sending, the compression format can be gzip, zlib, or deflate.

* **Gid**

A database's data may contain different sources: self-generated data and data copied from elsewhere. If there is no corresponding measure, it may cause the data to be copied in a ring. For example, if A's data is copied to B and then B is copied to A, it causes the service generation storm to be crashed. Or write back to A from B failed because of the unique key constraint. This leads to instability in the service.

In the MongoDB version on Alibaba Cloud, we provide a feature to prevent ring replication. The main principle is that by modifying the MongoDB kernel, adding gid in the oplog identifies the current database information, and in the copying process, the op\_command command carries the gid information, then each piece of data has source information. If only the data generated by the current database is needed, only the oplog with gid equal to the database id can be fetched. Therefore, in the scenario of a ring copy, MongoShake fetches data with gid equal to id\_A (gid of database A) from the A database, and grabs data with gid equal to id\_B (gid of database B) from the B database to solve this problem.

Since the modification of MongoDB kernel is not yet open source, this feature is not supported in this open source version, but is supported in the MongoDB on the Alibaba Cloud (<https://www.aliyun.com/product/mongodb?spm=5176.8142029.388261.351.a7236d3evDQVW1>). This is also why we mentioned at the beginning that "the mirror backup of MongoDB cluster data" is limited in the current open source version.

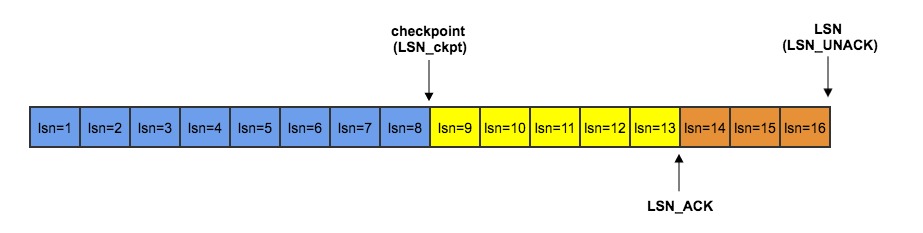
* **checkpoint**

MongoShake uses ACK mechanism to ensure oplog replay successfully, it will trigger retransmission when failed. This is mainly designed to ensure the reliability of the application layer, such as decompression failures, etc.

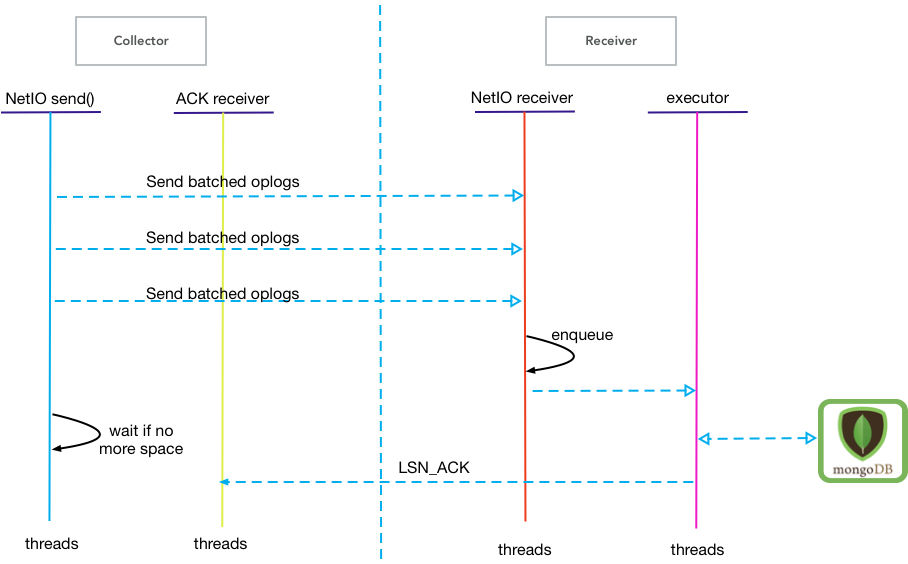
For better explanation, let me define several nouns first:

* LSN(Log Sequence Number). Indicates the newest oplog sequence number that already been transferred.
* LSN\_ACK(Acked Log Sequence Number). Indicates the maximum LSN acknowledged has been received, that is, the successful LSN is written to the consumer.
* LSN\_CKPT(Checkpoint Log Sequence Number). Indicates the checkpoint that already been persisted.

All the value of LSN, LSN\_ACK and LSN\_CKPT are from the “ts” in the oplog that is a timestamp mark the generation time, the implicit constraint is LSN\_CKPT <= LSN\_ACK <= LSN.



The above figure is an example show the relationship of LSN, LSN\_ACK and LSN\_CKPT. LSN=16 means MongoShake has already transmitted 16 oplogs, next time MongoShake will transmit LSN=17 if no retransmission trigger; LSN\_ACK=13 means the both the previous 13 oplogs has received acknowledgment, the minimum index is 14 if need retransmit; LSN\_CKPT=8 means checkpoint equal to 8 is persisted, the meaning of persistent is that if service restart or switch, it can fetch oplog start from LSN\_CKPT instead of the beginning. Because of the idempotent of oplog DML, multiple transmissions of the same data will not cause problems. But for DDL, retransmissions may cause errors.



The above figure shows the process of update the LSN\_ACK asynchronously.

* **Troubleshooting & Speed Limit**

MongoShake offer Restful API through which user can troubleshooting inner queue workload, LSN update, data synchronization etc. Beside, user can also control the speed according to their own needs, e.g., database pressure, network pressure.

* **Conflict Detection**

MongoShake supports concurrency based on collection or document (id). Concurrency at the document level requires that database has no unique index constraint, and that table level concurrency has poor performance when the number of tables is small or some tables are very unevenly distributed. Therefore, in the case of collection-level concurrency, it is necessary to be able to distribute the concurrency evenly and also to solve the case of a unique key conflict within the collection. For this reason, if the tunnel type is direct, we provide a collision detection function before writing.

Currently, the index type only supports unique indexes. It does not support sparse indexes, TTL indexes, and other indexes.

The premise of the conflict detection function needs to meet two prerequisite constraints:

1. MongoShake believes that the synchronized MongoDB Schema is consistent and does not listen to the changes of oplog's “system.indexes“ collection.
2. The conflict indexes is based on the records in oplog, not the current MongoDB indexes.

In addition, MongoShake's operation of the index during synchronization may cause exception:

1. Index is being created. If the index is built in the background, the index is not visible to the write request during this period, but the index is visible internally, and memory usage may be too high. If the foreground is indexed, all user requests are blocked. If the blocking time is too long, retransmission will be triggered.
2. If destination database exist indexes while source not, the data maybe inconsistent but we don’t handle.
3. After the oplog is generated, the source database adds or deletes the unique index. Retransmission may cause problems with the addition or deletion of the index, but we do not handle this case.

In order to support conflict detection, we modify the MongoDB kernel to insert “uk” field into oplog to indicate the unique index information involved. For example, below shows an oplog contain “uk”:

|  |
| --- |
| {  "ts" : Timestamp(1484805725, 2),  "t" : NumberLong(3),  "g": 123, //gid  "h" : NumberLong("-6270930433887838315"),  "v" : 2,  "op" : "u",  "ns" : "benchmark.sbtest10",  "o" : { "\_id" : 1, "uid" : 1111, "other.sid":"22222", "mid":8907298448, "bid":123 }  "o2" : {"\_id" : 1}  "uk" : { // conflict unique keys  "uid": "1110"  "mid^bid": [8907298448, 123]  "other.sid\_1": "22221"  }  } |

There’re three unique keys in the “uk” field: “uid”, “mid^bid” and "other.sid\_1". “mid^bid” is a unique compound index that contain “mid” and “bid”. The value of every key in the “uk” has different meaning: value is empty if the operation is insertion, but if the operation is delete or modification, value stores the before modification result.

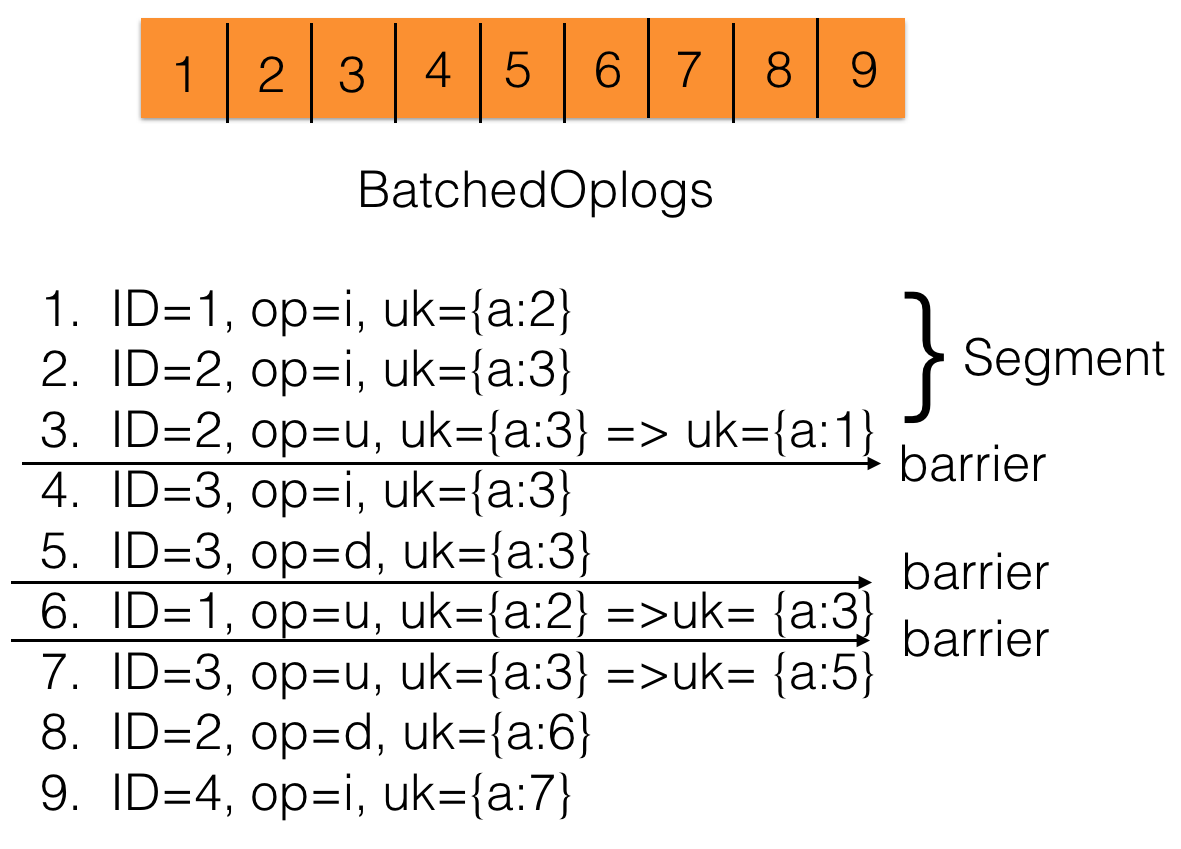
The specific processing flow is as follows. The continuous k oplogs are packaged into one batch, and the pipelines analyze the dependencies in each batch and divide them into segments. If there is a conflict, the batch is sliced into segments based on dependency and timing relationships; if there is no conflict, it is divided into one segment. Then write concurrently within the segment and write sequentially between segments. Concurrency in segments means that multiple concurrent threads perform write operations on segment data at the same time, but the same document id in the same segment must be guaranteed to be in order; the segments must be guaranteed to execute sequentially: only the previous segment is completely executed, and the next segment will start writing.

If an oplog with different document id operates on the same unique key at the same time in a batch, then these oplogs are considered to have a timing relationship, which is also called a dependency. We must split these oplog into at least 2 segments.

There are two ways to handle dependencies in MongoShake:

* Insert Barrier

Divide the batch by inserting a barrier and perform concurrency within each segment. Here comes the example:



“ID” represents the document id, “op” represents the operation, “i” is the insertion, “u” is the update, “d” is the deletion, “uk” represents all the unique keys under the document, “uk={a:3} => uk={a:1}” indicates that the value of unique key “a” is changed from 3 to 1.

At the beginning, there were 9 oplogs in the batch, which were split by analyzing the “uk” relationship. For example, articles 3 and 4 operate on the same unique key “a” with value 3 but their document id is 2 and 3 respectively. Therefore, a barrier needs to be inserted between articles 3 and 4 (no matter which is the same before or after modification). Similarly, articles 5 and 6, articles 6 and 7. The same id operation with the same uk is allowed within a segment, so articles 2 and 3 can be assigned to the same segment. After splitting, segment is concurrently writing according to the document id, and the same id remains in order. For example, in the first segment, article 1 and article 2 and 3 can be concurrently writing, but article 2 and article 3 need to be executed sequentially.

* Split According to a Dependency Graph

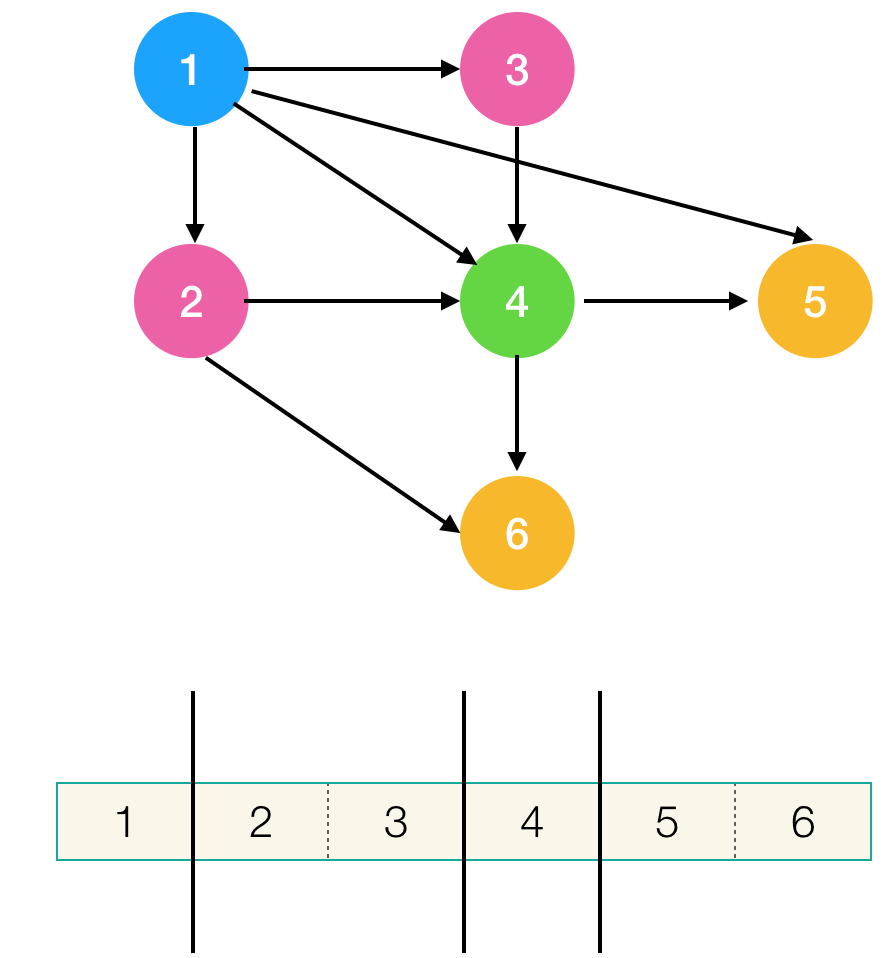
Every oplog has a timestamp sequence number(seq\_id), so we can build graph base on dependent rules. In this graph, every oplog is a node. Here comes two rules of build graph.

1. If node M and node N operate the same unique key with same value no matter before or after, and the seq\_id of M is less than N, then we build a directed edge from M to N.
2. If node M and node N has the same document ID and the seq\_id of M is less than N, then also build a directed edge from M to N.

Because the edge is connected by time, so there mustn’t exist loop in this graph, that means it’s a directed acyclic graph. So we can use topology sort algorithm to solve this problem:

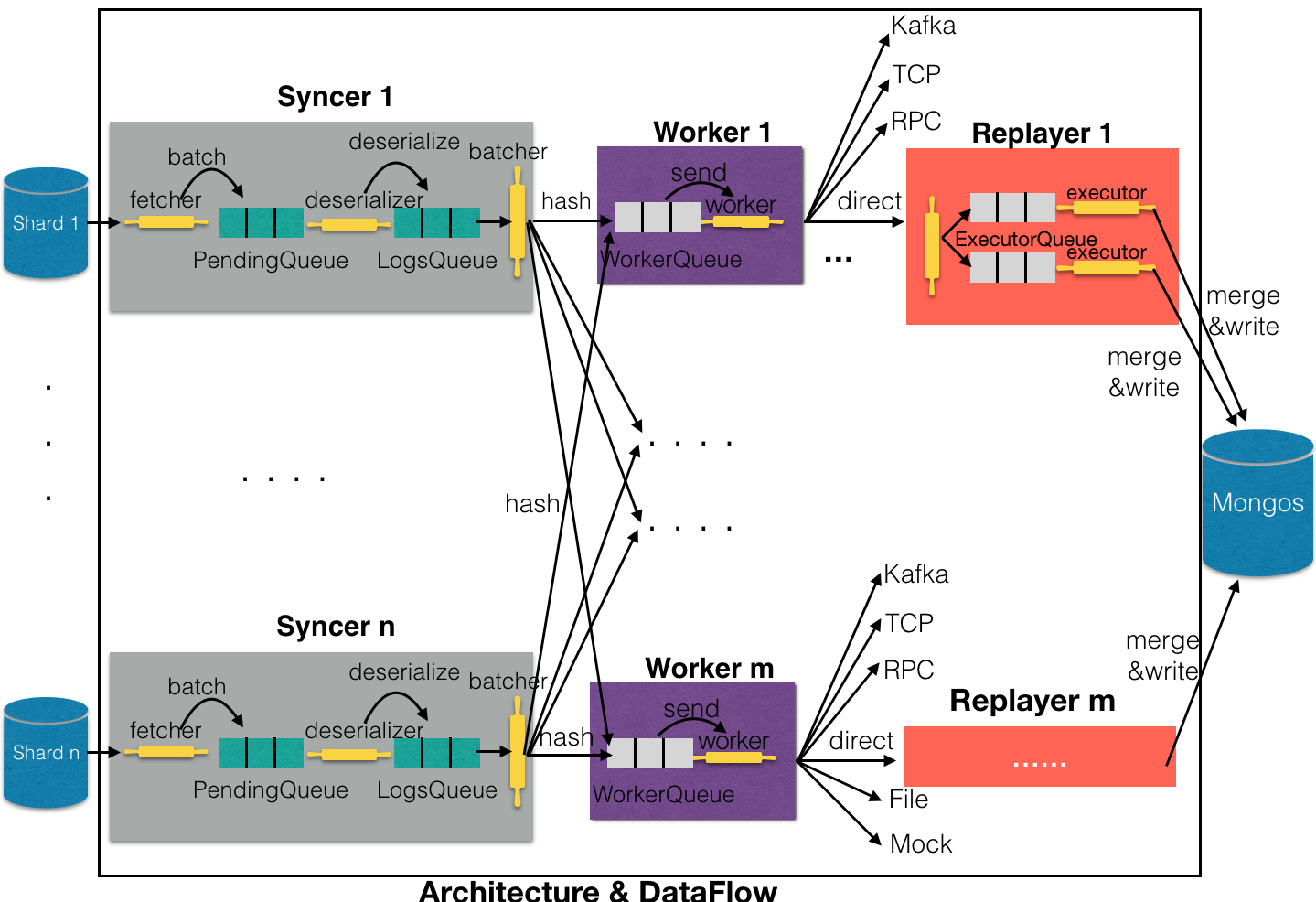
|  |
| --- |
| 1. Build graph. 2. Find nodes with ingress degree zero, and add into set. 3. Perform concurrent writes in this set. 4. Delete all the edges starting from nodes in this set and minus their successor ingress degree. 5. Empty set. 6. Loop to 2 until set is empty. |

The example is shown below. There’re six oplog nodes in the graph, the figure is divided into four sets. First we write node 1, then we concurrent write node 2 and 3, next is node 4, and node 5 and 6 are the last.



Since the modification of MongoDB kernel is not yet open source, this feature is also not supported in this open source version, but is supported in the MongoDB on the Alibaba Cloud (<https://www.aliyun.com/product/mongodb?spm=5176.8142029.388261.351.a7236d3evDQVW1>). This is also why we mentioned at the beginning that "the mirror backup of MongoDB cluster data" is limited in the current open source version.

## Architecture & Data Flow



The figure above shows the MongoShake internal architecture and data flow details. In general, the entire MongoShake can be roughly divided into three major parts: Syncer, Worker, and Replayer, where Replayer is only used when the tunnel type is direct.

Syncer is responsible for fetching data from the source database. If the source is a Mongod or a ReplicaSet, then there is only one Synder. If it is a Sharding mode, then there needs to be multiple syncers connecting to each shard. Inside Synder, Fetcher uses mgo.v2 library to fetch data from the source library and then put the packed batch into the PendingQueue queue. The deserializer thread fetches data from the PendingQueue for deserialization. Batcher reorganizes the data captured from the LogsQueue, gathering data to the same worker into batch again, and then hash and sent to the corresponding Worker queue.

The main function of the worker is to grab data from the WorkerQueue and then send it. Because of the ACK mechanism, it maintains internally several queues, which are unsent queues and sent queues. The former stores unsent data, the latter stores the transmits but not acknowledged data. After sending, the data of the unsent queue will be transferred to the sent queue; when the peer's ack reply is received, the data in the sent queue whose sequence id is smaller than ack will be deleted, thus ensuring the reliability.

Workers can connect different channels to meet different needs of users. If the channel type is direct, then the Replayer will be directly written to the target MongoDB operation, and the Worker corresponds to the Replayer. First, Replayer distributes the received data to different ExecutorQueues according to conflict detection rules, and the executor then fetches from the queue for concurrent writes. In order to ensure the efficiency of writing, MongoShake will also merge oplogs of the same Operation and the same namespace before the write.

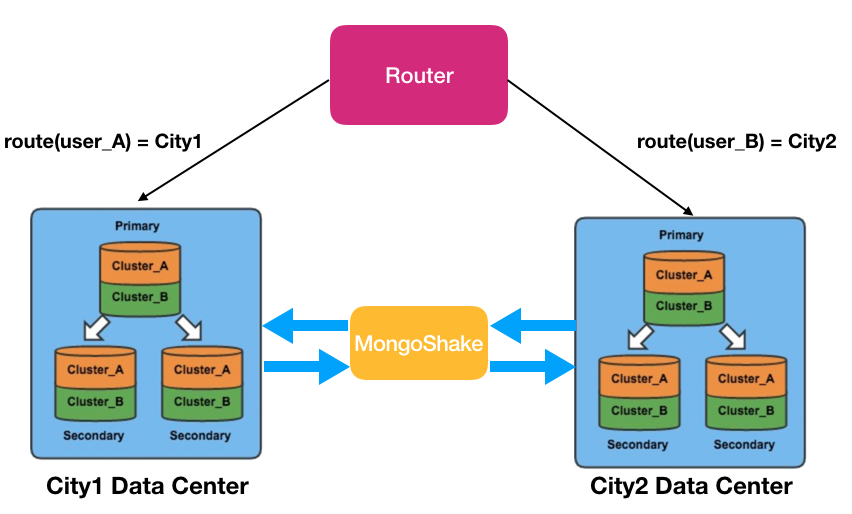
## User use case

Gao De Map App is the leading map and navigation application in China. Alibaba Cloud MongoDB database service provides storage support for this application and stores billion-level data. The Gaode map now uses the domestic dual-center strategy to route the user to the nearest center through geographical information to improve service quality. There’re three data centers and no data calculation dependence. That means the service in data center 1 won’t use data from other data centers like data center 2 or 3.

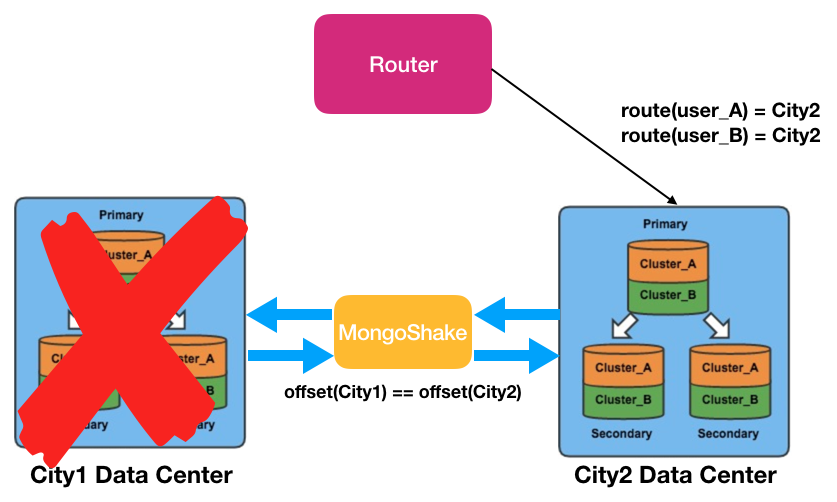


These three cities span the whole of China geographically from north to south. This poses a challenge to how data centers do a good job of copying and disaster recovery. If there is a problem in the data center or network of a certain geographical area, the traffic can be smoothly switched to another place where users are almost unaware?

At present, our strategy is to use the two-to-two interconnection of the data centers, and the data of each data center will be synchronized to the other two. Then through the German routing layer, user requests are routed to different data centers, and both read and write are sent in the same data center. Then through MongoShake, two-way asynchronous replication of data in two data centers, so as to ensure that each data center has the full amount of data (to ensure eventual consistency). If there is a problem in any data center, one of the other two rooms can provide read and write services after switching. The figure below shows the synchronization between the City 1 and City 2 rooms.



When a certain node cannot be accessed, the synchronization offset and timestamp of each data center can be obtained through the Restful management interface offered by MongoShake. By judging the collected and written values, whether the asynchronous replication is finished at a certain time can be determined. Coupled with the DNS switch, switching the traffic from old node to new node so that the new node can continue to provide services. Here comes the picture.



## Performance

Please visit the performance test document.

## What’s More

MongoShake will be maintained for a long period of time, and major and minor versions will continue to iterate. Please feel free to leave a message and join us for open source development.

In addition, Alibaba Cloud MongoDB team is recruiting, welcomes to deliver resumes and letters: [zhuzhao.cx@alibaba-inc.com](mailto:zhuzhao.cx@alibaba-inc.com)