Shardmonster Documentation

Release 0.2.0

Colin Howe

November 09, 2015

Contents

Mongo has great support for sharding. However, the system must be entirely the same version. This means that you cannot selectively move data from Mongo 2.4 to Mongo 3.0 (for example). Big bang upgrades of databases are scary (and we've had downtime in the past when we tried to do an upgrade to 2.6). Writing this ourselves allows us to move data between different shards of differing versions at will.

CHAPTER 1

Terminology

This is made more difficult by the fact that a Mongo server can contain multiple databases.

Cluster - The complete set of mongo servers that store all data.

Location - A server (or replica set) combined with a database name that contains data. A single cluster will contain multiple locations.

Realm - A collection that spans the cluster. May have data stored in multiple versions of Mongo. A realm may span multiple locations (depending on how much the data has been sharded). A cluster may contain multiple realms of data.

Shard - A set of data denoted by a field (the shard field) and a value (the shard key). The shard, during a migration, may be stored in multiple locations. A realm typically contains multiple shards. A location also contains multiple shards.

CHAPTER 2

Further documentation

2.1 Getting Started

2.1.1 Connecting the Controller

Shardmonster relies on a connection to a controller. This is where all the shard metadata is stored. It is recommended that metadata is stored on a dedicated cluster. This helps ensure that reads/writes to metadata are very fast.

```
import shardmonster
shardmonster.connect_to_controller(
    "mongodb://localhost:27017/?replicaset=cluster-1",
    "sharding_db"
```

2.1.2 Activate Caching

Shardmonster makes use of caching of metadata to ensure that queries are as fast as possible. The cache length is set in seconds. All migrations are paused for at least as long as the cache length to ensure caches are clear before writes are moved to new locations. Having a high cache time will make migrations slower. Having a low cache time will result in more hits to the metadata cluster.

```
shardmonster.activate_caching(5)
```

2.1.3 Describe Clusters

A cluster describes a replica set that is to store data. Typically, you will have multiple clusters with data spread across them.

```
shardmonster.ensure_cluster_exists(
    'cluster-1', 'mongodb://localhost:27017/?replicaset=cluster-1')
```

2.1.4 Create Realms

A realm is a sharded collection. To create a realm you need to have defined a field to use as a shard field - this will be used to determine on which shard a document belongs.

The shard field must only contain strings or integers. No other data type is currently supported here.

```
# Create a realm called messages that is a sharded version of the
# messages_coll collection. By default shards will go to some_db on
# cluster-1. The data will be sharded by the field called "account".
shardmonster.ensure_realm_exists(
    'messages', 'account', 'messages_coll', 'cluster-1/some_db')
```

2.1.5 Preparing for Queries

To actually do sharded queries and inserts you will need a handle to a shard aware collection.

```
sharded_collection = \
    shardmonster.make_collection_shard_aware("messages")
sharded_collection.insert({"text": "Hello!", "account": 5})
```

2.1.6 Move some data around

Before a shard can be moved to a new cluster it must be first placed at rest at its current location.

```
# Following on from the previous block...
# Set all data in the messages realm with an account value of 5 to be at
# rest on cluster-1. As the data is already there this does do any movement
# of data.
shardmonster.set_shard_at_rest('messages', 5, 'cluster-1/some_db')
```

Once this is done the data can be migrated to a new location:

```
# This moves data from the messages collection with an account value of 5
# to a different cluster. The method returns when it is completed.
shardmonster.do_migration('messages', 5, 'cluster-2/some_other_db')
```

2.1.7 Where is my data?

After you've been using shardmonster for some time you might want some help interrogating your data and finding out where it is.

```
>>> shardmonster.where_is('messages', 5)
'cluster-2/some_other_db'
```

2.2 API

shardmonster.**activate_caching**(*timeout*)

Activates caching of metadata.

Parameters **timeout** (*int*) – Number of seconds to cache metadata for.

Caching is generally a good thing. However, during a migration there will be a pause equal to whatever the caching timeout. This is to avoid stale reads and writes when the source of truth for a shard changes location.

shardmonster.**connect_to_controller**(*uri*, *db_name*)

Connects to the controlling database. This contains information about the realms, shards and clusters.

Parameters

- **uri** (*str*) – The Mongo URI to connect to. This should typically detail several replica members to ensure connectivity.
- **db_name** (*str*) – The name of the database to connect to on the given replica set.

shardmonster.**do_migration**(*collection_name*, *shard_key*, *new_location*)

Migrates the data with the given shard key in the given collection to the new location. E.g.

```
>>> do_migration('some_collection', 1, 'cluster-1/some_db')
```

Would migrate everything from some_collection where the shard field is set to 1 to the database some_db on cluster-1.

Parameters

- **collection_name** (*str*) – The name of the collection to migrate
- **shard_key** – The key of the shard that is to be moved
- **new_location** (*str*) – Location that the shard should be moved to in the format "cluster/database".

This method blocks until the migration is completed.

shardmonster.**ensure_cluster_exists**(*name*, *uri*)

Ensures that a cluster with the given name exists. If it doesn't exist, a new cluster definition will be created using name and uri. If it does exist then no changes will be made.

Parameters

- **name** (*str*) – The name of the cluster
- **uri** (*str*) – The URI to use for the cluster

shardmonster.**ensure_realm_exists**(*name*, *shard_field*, *collection_name*, *default_dest*)

Ensures that a realm of the given name exists and matches the expected settings.

Parameters

- **name** (*str*) – The name of the realm
- **shard_field** – The field in documents that should be used as the shard field. The only supported values that can go in this field are strings and integers.
- **collection_name** (*str*) – The name of the collection that this realm corresponds to. In general, the collection name should match the realm name.
- **default_dest** (*str*) – The default destination for any data that isn't explicitly sharded to a specific location.

Returns None

shardmonster.**make_collection_shard_aware**(*collection_name*)

Returns a new object that proxies the given collection and makes it shard aware.

shardmonster.**set_shard_at_rest**(*realm*, *shard_key*, *location*, *force=False*)

Marks a shard as being at rest in the given location. This is used for initiating shards in preparation for migration. Unless force is True this will raise an exception if a shard is already at rest in a specific location.

Parameters

- **realm** (*str*) – The name of the realm for the shard
- **shard_key** – The key of the shard
- **location** (*str*) – The location that the data is at (or should be in the case of a brand new shard)

- **force** (*bool*) – Force a shard to be placed at rest in a specific location even if it has already been placed somewhere.

Returns None

shardmonster.**where_is**(*collection_name*, *shard_key*)

Returns a string of the form cluster/database that says where a particular shard of data resides.

Parameters

- **collection_name** – The collection name for the shard
- **shard_key** – The shard key to look for

shardmonster.**wipe_metadata**()

Wipes all metadata. Should only be used during testing. There is no undo.

Wipes caches as well.

2.3 How it works

2.3.1 Realms

A realm is a mapping describing a collection's sharding strategy. It consists of documents of the form:

```
{
    name: 'some-realm',
    shard_field: 'some-field',
    collection: 'collection-name',
    default_location: 'cluster/database'
}
```

2.3.2 Shard data

There is a shards collection that stores documents of the form:

```
{
    realm: 'some-realm',
    shard_key: 'some-key',
    status: 'migrating-copy'
            | 'migrating-sync'
            | 'post-migration-paused-at-source'
            | 'post-migration-paused-at-destination'
            | 'post-migration-delete'
            | 'at-rest',
    location: 'cluster/database'
}
```

This describes the position of the data where some-field is equal to some-key. In the event that the data is in a migrating phase then there will be additional fields:

```
new_location: 'cluster2/database',
```

2.3.3 Migration phases

Copy

The copy phase is copying all data that matches the shard query from the current location to the new location. Once this phase is finished, the sync phase begins.

During this phase all reads and writes will go to the original location of the data.

Sync

The sync phase is where the oplog is replayed since the start of the migration.

Once the oplog is replayed and we are close to realtime the post migration pause begins.

During this phase all reads and writes will go to the original location of the data.

Post migration pause at source

Once data has been migrated the shard will enter a pause state. This allows for someone (or something) to verify that the data has been migrated successfully and everything is OK. This state shouldn't really be needed, but, it's a nice safety valve during testing.

During this phase all reads and writes will go to the original location of the data. The oplog continues to be synced during this period.

Post migration pause at destination

Once any data validation has been performed the shard will be moved to "paused at destination". During this phase all reads will go to the new location of the data. Writes will be suspended. The oplog continues to be synced during this period to catch any stragglers.

Due to the suspension of writes during this period it is expected that this phase will be very short.

Post migration delete

Once a shard has been migrated the original location should have the data removed. Doing this deletion helps with reducing the amount of query customisation that has to happen to cope with data being in two places at once.

Crucially, the oplog is NOT synced during this phase. That would copy the deletes and lead to all kinds of sadness. Specifically, the removal of most of the data.

Only once this phase is completed is the shard moved to at rest and the location field is updated.

At rest

Data that is at rest is no longer being migrated in any form. Lookups for shard information will be cached for a short length of time.

2.4 Developing Shardmonster

2.4.1 Running tests

```
> cp sample_test_settings.py test_settings.py
```

Update test_settings.py to reflect your local environment.

```
> python setup.py nosetests
```

2.4.2 Building Docs

```
> python setup.py build_sphinx
```

CHAPTER 3

Indices and tables

- genindex
- modindex

S

A

C

D

E

M

S

W