How to Programatically Pre-Split a GUID Based Shard Key with MongoDB

Let's say I am using a fairly standard 32 character hex GUID, and I have determined that, because it is randomly generated for my users, it is perfect for use as a shard key to horizontally scale writes to the MongoDB collection that I will be storing the user information in (and write scaling is my primary concern).

I also know that I will need to start with at least 4 shards, because of traffic projections and some benchmark work done with a test environment.

Finally, I have a decent idea of my initial data size (average document size * number of initial users) - which comes to around ~120GB.

I'd like to make the initial load nice and fast and utilize all 4 shards as much as possible. How do I pre-split this data so that I take advantage of the 4 shards and minimize the number of moves, splits etc. that need to happen on the shards during the initial data load?

mongodb sharding

asked Oct 30 '13 at 1:35

Adam C

7,813 1 14 33

add comment

1 Answer

We know the intial data size (120GB) and we know the maximium chunk size in MongoDB is 64MB. If we divide 64MB into 120GB we get 1920 - so that is the minimum number of chunks we should look to start with. As it happens 2048 happens to be a power of 16 divided by 2, and given that the GUID (our shard key) is hex based, that's a much easier number to deal with than 1920 (see below).

NOTE: This pre-splitting must be done *before* any data is added to the collection. If you use the enableSharding() command on a collection that contains data, MongoDB will split the data itself and you will then be running this while chunks already exist - that can lead to quite odd chunk distribution, so beware.

For the purposes of this answer, let's assume that the database is going to be called `users` and the collection is called `userInfo`. Let's also assume that the GUID will be written into the `_id` field. With those parameters we would connect to a `mongos` and run the following commands:

```
// first switch to the users DB
use users;
// now enable sharding for the users DB
sh.enableSharding("users");
// enable sharding on the relevant collection
sh.shardCollection("users.userInfo", {"_id" : 1});
// finally, disable the balancer (see below for options on a per-collection basis)
// this prevents migrations from kicking off and interfering with the splits by co
sh.stopBalancer();
```

Now, per the calculation above, we need to split the GUID range into 2048 chunks. To do that we need at least 3 hex digits (16 ^ 3 = 4096) and we'll be putting them in the most significant digits (i.e. the 3 leftmost) for the ranges. Again, this should be run from a `mongos` shell

```
// Simply use a for loop for each digit
for ( var x=0; x < 16; x++ ){
  for( var y=0; y<16; y++ ) {
  // for the innermost loop we will increment by 2 to get 2048 total iterations
  // make this z++ for 4096 - that would give ~30MB chunks based on the original f
    for ( var z=0; z<16; z+=2 ) {
    // now construct the GUID with zeroes for padding - handily the toString metho
        var prefix = "" + x.toString(16) + y.toString(16) + z.toString(16) + "0000
        // finally, use the split command to create the appropriate chunk
        db.adminCommand( { split : "users.userInfo" , middle : { _id : prefix } }
    }
  }
}
```

Once that is done, let's check the state of play using the `sh.status()` helper:

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "version" : 3,
        "minCompatibleVersion" : 3,
        "currentVersion" : 4,
        "clusterId" : ObjectId("527056b8f6985e1bcce4c4cb")
}
  shards:
        {  "_id" : "shard0000",  "host" : "localhost:30000" }
        {  "_id" : "shard0001",  "host" : "localhost:30001" }
        {  "_id" : "shard0002",  "host" : "localhost:30002" }
        {  "_id" : "shard0003",  "host" : "localhost:30003" }
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "users",  "partitioned" : true,  "primary" : "shard0001" }
                users.userInfo
                        shard key: { "_id" : 1 }
                        chunks:
                                shard0001       2049
                        too many chunks to print, use verbose if you want to force
```

We have our 2048 chunks (plus one extra thanks to the min/max chunks), but they are all still on the original shard because the balancer is off. So, let's re-enable the balancer:

```
sh.startBalancer();
```

This will immediately begin to balance out, and it will be relatively quick because all the chunks are empty, but it will still take a little while (much slower if it is competing with migrations from other collections). Once some time has elapsed, run `sh.status()` again and there you (should) have it - 2048 chunks all nicely split out across 4 shards and ready for an initial data load:

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "version" : 3,
        "minCompatibleVersion" : 3,
        "currentVersion" : 4,
        "clusterId" : ObjectId("527056b8f6985e1bcce4c4cb")
}
  shards:
        {  "_id" : "shard0000",  "host" : "localhost:30000" }
        {  "_id" : "shard0001",  "host" : "localhost:30001" }
        {  "_id" : "shard0002",  "host" : "localhost:30002" }
        {  "_id" : "shard0003",  "host" : "localhost:30003" }
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "users",  "partitioned" : true,  "primary" : "shard0001" }
                users.userInfo
                        shard key: { "_id" : 1 }
                        chunks:
                                shard0000       512
                                shard0002       512
                                shard0003       512
                                shard0001       513
                        too many chunks to print, use verbose if you want to force
        {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0002" }
```

You are now ready to start loading data, but to absolutely guarantee that no splits or migrates happen until your data load is complete, you need to do one more thing - turn off the balancer and autosplitting for the duration of the import:

- To disable all balancing, run this command from the mongos: `sh.stopBalancer()`
- If you want to leave other balancing operations running, you can disable on a specific collection.

Using the namespace above as an example: `sh.disableBalancing("users.userInfo")`

- To turn off auto splitting during the load, you will need to restart each `mongos` you will be using to load the data with the `--noAutoSplit` option.

Once the import is complete, reverse the steps as needed (`sh.startBalancer()` , `sh.enableBalancing("users.userInfo")` , and restart the `mongos` without `--noAutoSplit`) to return everything to the default settings.

**

Update: Optimizing for Speed

**

The approach above is fine if you are not in a hurry. As things stand, and as you will discover if you test this, the balancer is not very fast - even with empty chunks. Hence as you increase the number of chunks you create, the longer it is going to take to balance. I have seen it take more than 30 minutes to finish balancing 2048 chunks though this will vary depending on the deployment.

That might be OK for testing, or for a relatively quiet cluster, but having the balancer off and requiring no other updates interfere will be much harder to ensure on a busy cluster. So, how do we speed things up?

The answer is to do some manual moves early, then split the chunks once they are on their respective shards. Note that this is only desirable with certain shard keys (like a randomly distributed UUID), or certain data access patterns, so be careful that you don't end up with poor data distribution as a result.

Using the example above we have 4 shards, so rather than doing all the splits, then balancing, we split into 4 instead. We then put one chunk on each shard by manually moving them, and then finally we split those chunks into the required number.

The ranges in the example above would look like this:

```
$min --> "40000000000000000000000000000000"
"40000000000000000000000000000000" --> "80000000000000000000000000000000"
"80000000000000000000000000000000" --> "c0000000000000000000000000000000"
"c0000000000000000000000000000000" --> $max
```

It's only 4 commands to create these, but since we have it, why not re-use the loop above in a simplified/modified form:

```
for ( var x=4; x < 16; x+=4){
    var prefix = "" + x.toString(16) + "00000000000000000000000000000000";
    db.adminCommand( { split : "users.userInfo" , middle : { _id : prefix } } );
}
```

Here's how thinks look now - we have our 4 chunks, all on shard0001:

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "version" : 4,
    "minCompatibleVersion" : 4,
    "currentVersion" : 5,
    "clusterId" : ObjectId("53467e59aea36af7b82a75c1")
}
  shards:
    {  "_id" : "shard0000",  "host" : "localhost:30000" }
    {  "_id" : "shard0001",  "host" : "localhost:30001" }
    {  "_id" : "shard0002",  "host" : "localhost:30002" }
    {  "_id" : "shard0003",  "host" : "localhost:30003" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0001" }
    {  "_id" : "users",  "partitioned" : true,  "primary" : "shard0001" }
        users.userInfo
            shard key: { "_id" : 1 }
            chunks:
                shard0001   4
            { "_id" : { "$minKey" : 1 } } -->> { "_id" : "40000000000000000000000000
            { "_id" : "40000000000000000000000000000000" } -->> { "_id" : "8000000
            { "_id" : "80000000000000000000000000000000" } -->> { "_id" : "c000000
            { "_id" : "c0000000000000000000000000000000" } -->> { "_id" : { "$maxK
```

We will leave the `$min` chunk where it is, and move the other three. You can do this programatically, but it does depend on where the chunks reside initially, how you have named your shards etc. so I will leave this manual for now, it is not too onerous - just 3 `moveChunk` commands:

```
mongos> sh.moveChunk("users.userInfo", {"_id" : "40000000000000000000000000000000"
{ "millis" : 1091, "ok" : 1 }
mongos> sh.moveChunk("users.userInfo", {"_id" : "80000000000000000000000000000000"
{ "millis" : 1078, "ok" : 1 }
mongos> sh.moveChunk("users.userInfo", {"_id" : "c0000000000000000000000000000000"
{ "millis" : 1083, "ok" : 1 }
```

Let's double check, and make sure that the chunks are where we expect them to be:

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "version" : 4,
    "minCompatibleVersion" : 4,
    "currentVersion" : 5,
    "clusterId" : ObjectId("53467e59aea36af7b82a75c1")
}
  shards:
    {  "_id" : "shard0000",  "host" : "localhost:30000" }
    {  "_id" : "shard0001",  "host" : "localhost:30001" }
    {  "_id" : "shard0002",  "host" : "localhost:30002" }
    {  "_id" : "shard0003",  "host" : "localhost:30003" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0001" }
    {  "_id" : "users",  "partitioned" : true,  "primary" : "shard0001" }
        users.userInfo
            shard key: { "_id" : 1 }
            chunks:
                shard0001   1
                shard0000   1
                shard0002   1
                shard0003   1
            { "_id" : { "$minKey" : 1 } } -->> { "_id" : "40000000000000000000000000
            { "_id" : "40000000000000000000000000000000" } -->> { "_id" : "8000000
            { "_id" : "80000000000000000000000000000000" } -->> { "_id" : "c000000
            { "_id" : "c0000000000000000000000000000000" } -->> { "_id" : { "$maxK
```

That matches our proposed ranges above, so all looks good. Now run the original loop above to split them "in place" on each shard and we should have a balanced distribution as soon as the loop finishes. One more `sh.status()` should confirm things:

```
mongos> for ( var x=0; x < 16; x++ ){
...   for( var y=0; y<16; y++ ) {
...   // for the innermost loop we will increment by 2 to get 2048 total iterati
...   // make this z++ for 4096 - that would give ~30MB chunks based on the orig
...     for ( var z=0; z<16; z+=2 ) {
...     // now construct the GUID with zeroes for padding - handily the toString
...         var prefix = "" + x.toString(16) + y.toString(16) + z.toString(16) +
...         // finally, use the split command to create the appropriate chunk
...         db.adminCommand( { split : "users.userInfo" , middle : { _id : prefi
...     }
...   }
... }
{ "ok" : 1 }
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "version" : 4,
    "minCompatibleVersion" : 4,
    "currentVersion" : 5,
    "clusterId" : ObjectId("53467e59aea36af7b82a75c1")
}
  shards:
    {  "_id" : "shard0000",  "host" : "localhost:30000" }
    {  "_id" : "shard0001",  "host" : "localhost:30001" }
    {  "_id" : "shard0002",  "host" : "localhost:30002" }
    {  "_id" : "shard0003",  "host" : "localhost:30003" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0001" }
    {  "_id" : "users",  "partitioned" : true,  "primary" : "shard0001" }
        users.userInfo
            shard key: { "_id" : 1 }
            chunks:
                shard0001   513
```

And there you have it - no waiting for the balancer, the distribution is already even.

edited Apr 10 at 12:04 answered Oct 30 '13 at 1:35


Adam C
7,813 1 14 33

Why not just split based on how many shards there are and let the splitting happen as data goes in? – Asya Kamsky Apr 16 at 0:40

Splitting is cheap now, more expensive later (though lightweight in general). This avoids the need to split unless you increase the amount of data you have (or screw up the distribution of data somehow) - splits are low cost, but not free, especially from the config server perspective and they could fail for various reasons (down config server, network etc.) - also, if you have a lot of mongos instances and an even traffic distribution (an edge case, granted) it can be particularly bad. Probably other reasons, so why take the chance? – Adam C Apr 16 at 2:35

add comment

Not the answer you're looking for? Browse other questions tagged mongodb sharding or ask your own question.