CHAPTER 14

Sharding

In the previous chapter, we covered the most commonly deployed MongoDB configuration: replica sets. Replica sets are essential for modern applications requiring availability that a single MongoDB instance cannot provide. As we've seen, replica sets can do some limited scaling of reads through secondary write. But, for large applications, particularly where the write workload exceeds the capability of a single cluster, sharded clusters may be deployed.

Everything we have covered in previous chapters is entirely applicable to sharded MongoDB servers. Indeed, it's probably best not to consider sharding until you have optimized your application workload and individual server configuration using the techniques covered in previous chapters.

However, there are some significant performance opportunities and challenges presented by sharded MongoDB deployments, and these will be covered in this chapter.

Sharding Fundamentals

We introduced sharding in Chapter 2. In a sharded database cluster, selected collections are partitioned across multiple database instances. Each partition is referred to as a "shard." This partitioning is based on a shard key value.

While replica sets are designed to provide high availability, sharding is designed to provide greater scalability. When your workload – particularly your write workload – exceeds the capacity of your server, then sharding provides a way to spread that workload across multiple nodes.

G. Harrison and M. Harrison, *MongoDB Performance Tuning*, https://doi.org/10.1007/978-1-4842-6879-7_14

Scaling and Sharding

Sharding is an architectural pattern developed to allow databases to support the massive workloads of the world's largest websites.

As application load grows, at some point the workload exceeds the capability of a single server. The capability of the server can be extended by shifting some read workload to secondary nodes, but eventually the amount of write workload to the primary becomes too great. We can no longer "scale up."

When "scaling up" becomes impossible, we turn to "scaling out." We add more primary nodes and split the workload across those primaries using sharding.

Sharding at scale was critical to the establishment of the modern Web – Facebook and Twitter were both early adopters of large-scale sharding using MySQL. However, it's not universally loved – sharding with MySQL involves a huge amount of manual configuration and breaks some of the core database capabilities. However, sharding in MongoDB is fully integrated into the core database and is relatively easy to configure and manage.

Sharding Concepts

Sharding is a big topic, and we can't provide a tutorial for all sharding considerations here. Please consult the MongoDB documentation or the book *MongoDB Topology Design* by Nicholas Cottrell (Apress, 2020) for a full review of sharding concepts.

The following sharding concepts are particularly significant:

- **Shard key**: The shard key is the attributes which determine into which shard any given document will be placed. Shard keys should have high cardinality (lots of unique values) to ensure that the data can be evenly distributed across shards.
- **Chunks**: Documents are contained within chunks, and chunks are allocated to specific shards. Chunking avoids MongoDB having to laboriously move individual documents across shards.
- **Range sharding**: With range sharding, contiguous groups of shard keys are stored within the same chunk. Range sharding allows for efficient shard key range scans but can result in "hot" chunks if the shard value is monotonically increasing.

- **Hash sharding**: In hash-based sharding, keys are distributed based on a hash function applied to the shard key.
- **The balancer**: MongoDB tries to keep the data and workload attributed to each shard equal. The balancer periodically moves data from one shard to another to maintain this balance.

To Shard or Not to Shard?

Sharding is the most sophisticated MongoDB configuration topology, and sharding is used by some of the world's largest and most performant websites. So sharding must be good for performance, right? Well, it is not quite that simple.

Sharding adds a layer of complexity and processing on top of your MongoDB database that – as often as not – makes individual operations a little slower. However, it allows you to throw more hardware resources at your workload. If – and only if – you have a hardware bottleneck involving operations to a replica set primary, then sharding might be the best solution. However, in most other circumstances, sharding adds complexity and overhead to your deployment.

Figure 14-1 compares the performance for sharded and unsharded collections for a few simple operations on equivalent hardware.[1] In most cases, operations against sharded collections are slower than against unsharded collections. Of course, every workload will be different, but the point is that sharding alone does not make things go faster!

[1]To make it a fair comparison, the shards were located on the same host as the single replica set option. Each node had an equivalent cache size, and there was no memory bottleneck.

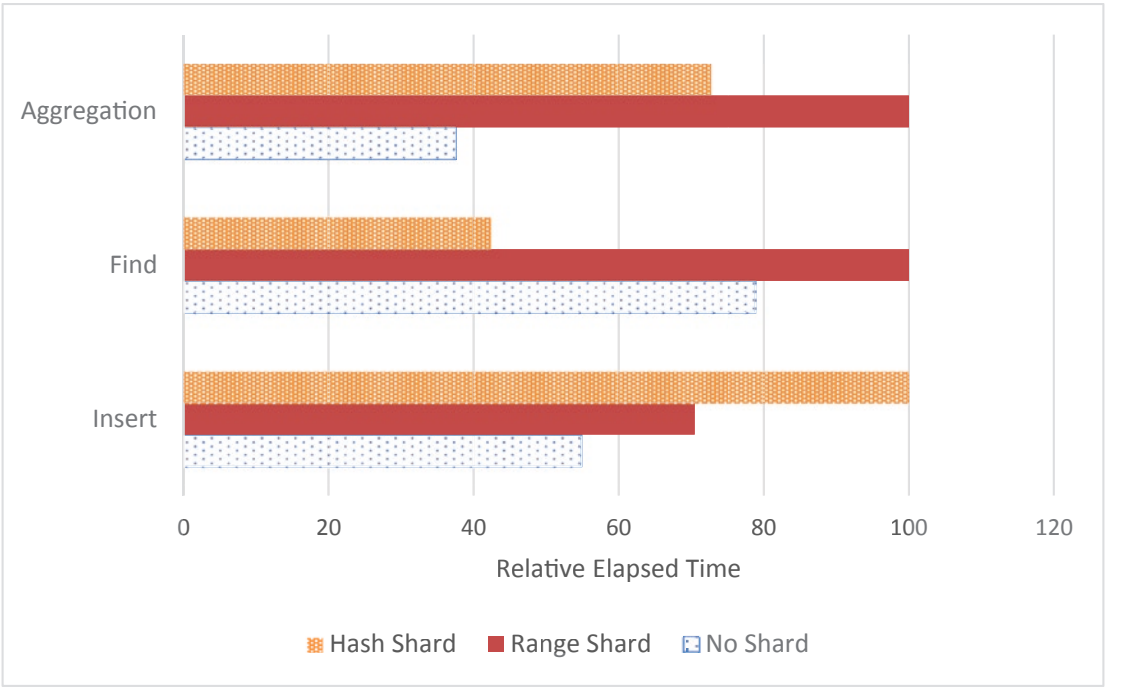

Figure 14-1. *Sharding doesn't always help performance*

Sharding is expensive in terms of dollar costs for the hardware and in terms of operational overhead. It should truly only be a recourse of last resort. Only when you have exhausted all other tuning measures, and all "scale-up" options, should you consider sharding. In particular, make sure that the disk subsystem on your primary is optimized before considering sharding. It's much cheaper and easier to buy and deploy some new SSDs than to shard a primary!

Warning Sharding should be the last resort for scaling a MongoDB deployment. Make sure your workload, server, and replica set configuration are optimized before commencing a sharding project.

Even if you believe that sharding is inevitable, you should still thoroughly tune your database before commencing the sharding project. If your workload and configuration are creating unnecessary load, then you may end up creating more shards than are necessary. Only when your workload is tuned can you make a rational determination of your sharding requirements.

Shard Key Selection

Sharding occurs at the collection level. While the number of shards in a cluster is the same for all collections, not all collections need be sharded and collections need not all have the same shard key.

Collections should be sharded if the aggregate IO write demand on the collection exceeds the capacity of a single primary. We then choose the shard key based on the following criteria:

- The keys should have a **high cardinality** so that data can be divided into small chunks if necessary.
- The keys should have an **even distribution** of values. If any single value is particularly common, then the shard key may be a poor choice.
- The key should be **frequently included in queries** so that queries can be routed to specific shards.
- The key should be **non-monotonically** increasing. When a shard key value increases monotonically (e.g., always increases by a set value), then the new documents appear in the same chunk, causing a hot spot. If you do have a monotonically increasing key value, consider using a hashed shard key.

Tip Choosing the correct sharding key is critical to the success of your sharding project. A shard key should support a good balance of documents across shard and support as many query filter conditions as possible.

Range- vs. Hash-Based Sharding

Distribution of data across shards can be either *range-based* or *hash-based*. In range-based partitioning, each shard is allocated a specific range of shard key values. MongoDB consults the distribution of key values in the index to ensure that each shard is allocated approximately the same number of keys. In hash-based sharding, keys are distributed based on a hash function applied to the shard key.

There are advantages and compromises involved in each scheme. Figure 14-2 illustrates the performance trade-offs inherent in range and hash sharding for inserts and range queries.

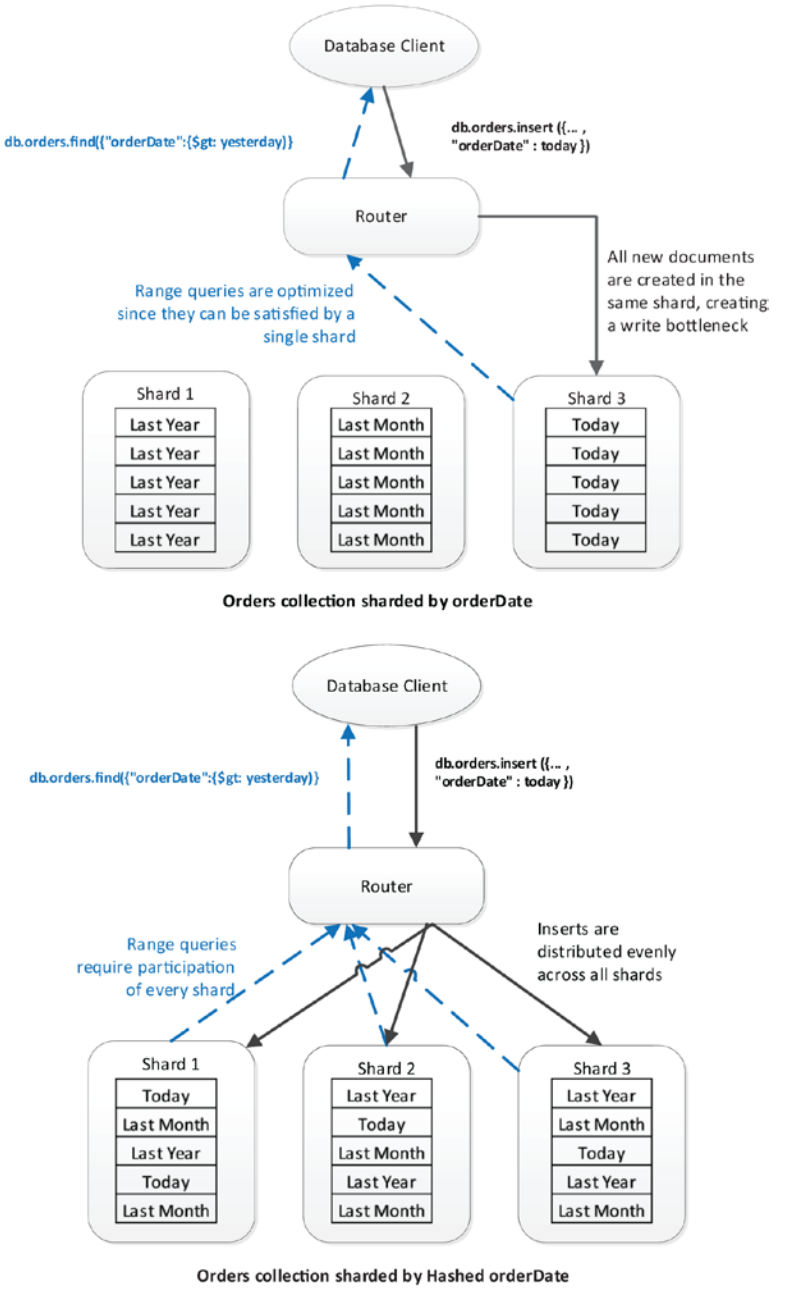

Figure 14-2. *Range- and hash-based sharding compared*

Range-based partitioning allows for efficient execution of shard key range scans since these queries can often be resolved by accessing a single shard. Hash-based sharding requires that range queries be resolved by accessing all shards. On the other hand, hash-based sharding is more likely to distribute "hot" documents (unfilled orders or recent posts, for instance) evenly across the cluster, thus balancing load more effectively.

Tip Hashed shard keys result in more evenly distributed data and workload. However, they result in poor performance for range-based queries.

Hashed shard keys do result in a more even distribution of data. However, as we'll soon see, hashed shard keys do create significant challenges for a variety of query operations, particularly those which involve sorting or range queries. Furthermore, we can only hash on a single attribute, while our ideal shard key is often composed of multiple attributes.

However, there is one use case in which a hashed shard key is clearly indicated. If we must shard on an attribute which is constantly increasing – often referred to as *monotonically increasing* – then a range sharding strategy will result in all new documents being inserted into a single shard. This shard will become "hot" in terms of inserts and probably in terms of reads as well since recent documents are often more likely to be updated and read than older documents.

Hashed shard keys come to the rescue here because the hashed values will be evenly distributed across the shards.

Figure 14-3 illustrates how monotonically increasing shard keys affect inserts into collections using hashed or range shard keys. In this example, the shard key is the `orderDate` which is always increasing as time moves forward. With hash sharding, inserts are distributed evenly between shards. In the range sharded scenario, all documents are inserted into a single shard. The hashed shard key not only distributes the workload across multiple nodes, it also results in greater throughput since there is less contention on that single node.

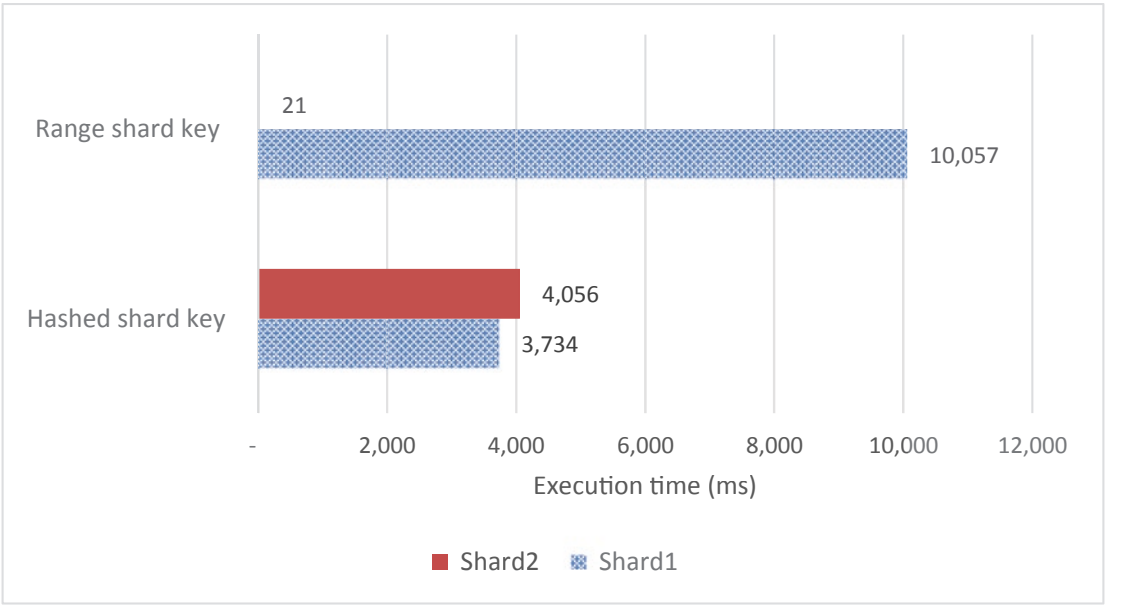

Figure 14-3. *Time to insert 120,000 documents into a sharded collection – hash vs. range monotonically increasing key*

Tip If your shard key must be a perpetually (monotonically) increasing value, then a hashed shard key is preferable. However, consider the possibility of sharding on another attribute if range queries on the shard key are required.

Zone Sharding

Most of the time, our sharding strategy is to distribute documents and workload evenly across all shards. Only by distributing the load evenly can we hope to gain effective scalability. If one shard is responsible for a disproportionate amount of the workload, then that shard may become a limiting factor in our overall application throughput.

However, there's another possible motivation for sharding – to distribute workload across shards so that data is close, in network terms, to the applications that want that data or to distribute data so that "hot" data is on expensive high-powered hardware, while "cold" data is stored on cheaper hardware.

Zone sharding allows the MongoDB administrator to fine-tune the distribution of documents to shards. By associating a shard with a zone and associating a range of

keys within a collection within that zone, the administrator can explicitly determine the shard on which these documents will reside. This can be used to archive data to shards on cheaper, but slower storage or to direct particular data to a specific data center or geography.

To create zones, we first allocate shards to zones. Here, we create one zone for the United States and another zone for the rest of the world:

```
sh.addShardToZone("shardRS2", "US");
sh.addShardToZone("shardRS", "TheWorld");
```

Even though we have only two zones, we can have as many shards as we want – each zone can have multiple shards.

Now we assign shard key ranges to each zone. Here, we have sharded by Country and City, so we use `minKey` and `maxKey` as proxies for the high and low City values within a Country range:

```
sh.addTagRange(
  "MongoDBTuningBook.customers",
  { "Country" : "Afghanistan", "City" : MinKey },
  { "Country" : "United Kingdom", "City" : MaxKey },
  "TheWorld");

sh.addTagRange(
  "MongoDBTuningBook.customers",
  { "Country" : "United States", "City" : MinKey },
  { "Country" : "United States", "City" : MaxKey },
  "US");

sh.addTagRange(
  "MongoDBTuningBook.customers",
  { "Country" : "Venezuela", "City" : MinKey },
  { "Country" : "Zambia", "City" : MaxKey },
  "TheWorld");
```

We would then locate the hardware for the “US” zone somewhere in the United States and the hardware for “TheWorld” somewhere in the rest of the world (Europe maybe). We would also deploy mongos routers in each of these regions. Figure 14-4 illustrates what this deployment might look like.

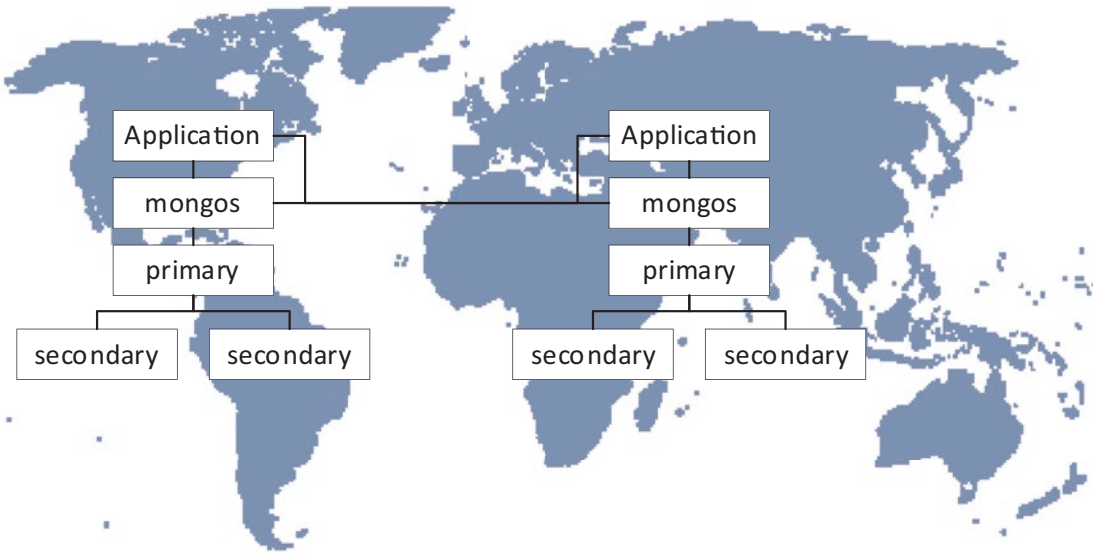

Figure 14-4. *Zone sharding to reduce geographic network latency*

The end result is lower latency for US queries issued from the US router and similarly for other geographies. Of course, if you issue a query for US data from Europe, your round trip time would be higher. But if queries issued from a region are mainly for data zoned to that region, then overall performance is improved.

We could add more zones in other regions as our application grows.

Tip Zone sharding can be used to distribute data across geographies, reducing latencies for region-specific queries.

Another use of zone sharding is to create archives of old data on slow but cheap hardware. For instance, if we have decades of order data, we could create a zone for older data which is hosted on VMs or servers with less CPU, memory, and maybe even using magnetic disk rather than premium SSD. Recent data could be kept on high-speed servers. This might result in a better overall performance for a given hardware budget.

Shard Balance

The `getShardDistribution()` method can show the breakdown of data across shards. Here's an example of a well-balanced sharded collection:

```
mongo> db.iotDataHshard.getShardDistribution()

Shard shard02 at shard02/localhost:27022,localhost:27023
 data : 304.04MiB docs : 518520 chunks : 12
 estimated data per chunk : 25.33MiB
 estimated docs per chunk : 43210

Shard shard01 at shard01/localhost:27019,localhost:27020
 data : 282.33MiB docs : 481480 chunks : 11
 estimated data per chunk : 25.66MiB
 estimated docs per chunk : 43770

Totals
 data : 586.38MiB docs : 1000000 chunks : 23
 Shard shard02 contains 51.85% data, 51.85% docs in cluster, avg obj size
 on shard : 614B
 Shard shard01 contains 48.14% data, 48.14% docs in cluster, avg obj size
 on shard : 614B
```

In a well-balanced sharded cluster, there are approximately the same number of chunks and the same amount of data in each shard. If the number of chunks between shards is inconsistent, then the balancer should be able to migrate chunks to return balance to the cluster.

If the number of chunks is roughly equivalent, but the amount of data in each shard varies significantly, then it may be that your shard key is not evenly distributed. A single shard key value cannot span chunks, so if some shard keys have massive document counts, then massive "jumbo" chunks will result. Jumbo chunks are sub-optimal, as the data within cannot be effectively distributed across shards and thus a larger proportion of queries may be sent to a single shard.

Rebalancing Shards

Let's say you have selected an appropriate shard key type (range or hashed) and the key possesses the right attributes – high cardinality, even distribution, frequently queried, non-monotonically increasing. In that case, your chunks will likely be well balanced across shards, and consequently, you will achieve a well-distributed workload. However, several factors may cause the shards to fall out of balance, with many more chunks

existing on one shard than another. When this occurs, that single node will become a bottleneck until the data can be evenly redistributed across multiple nodes – as shown in Figure 14-5.

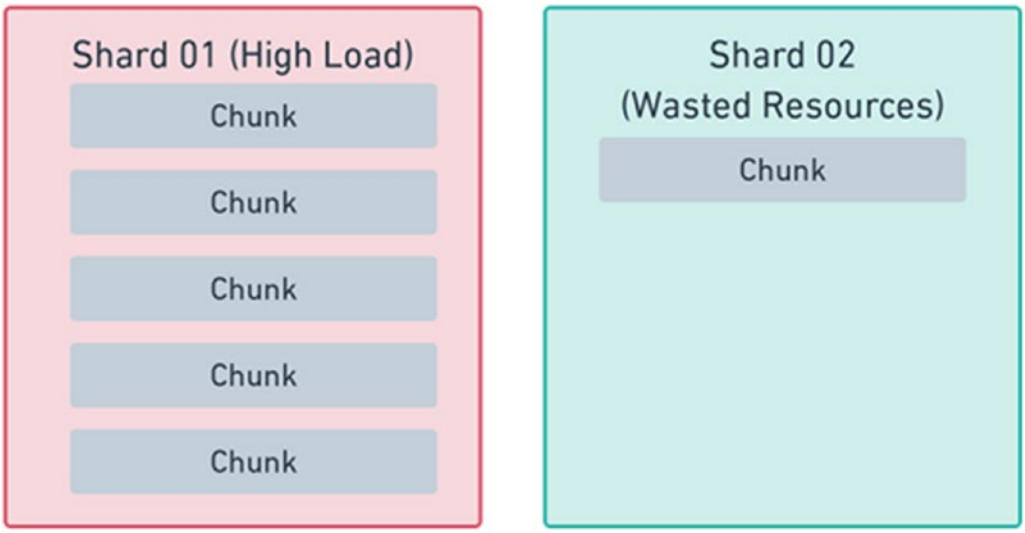

Figure 14-5. *A set of poorly balanced shards, most queries will go to Shard 01*

If we can maintain an appropriate balance among our shards, query load is more likely to be divided evenly among the nodes – as shown in Figure 14-6.

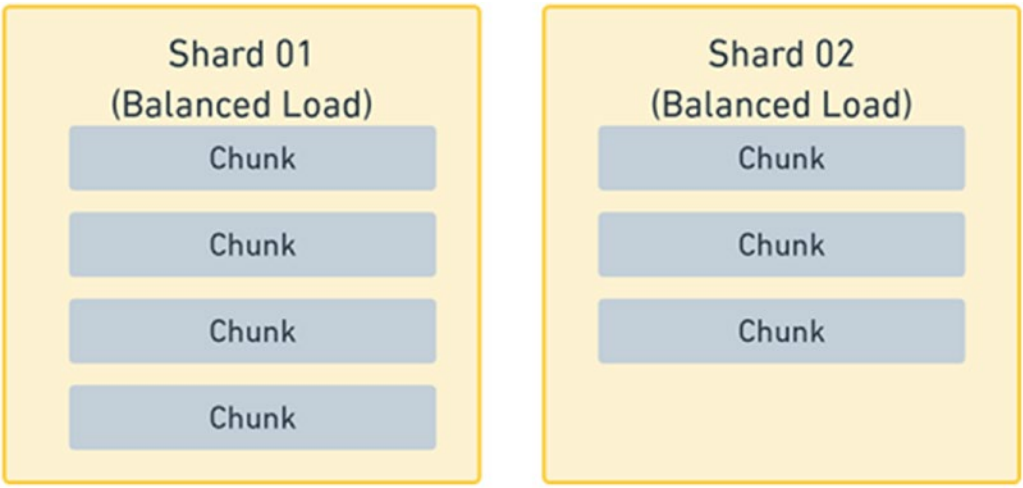

Figure 14-6. *A set of well-balanced shards: query load will be evenly distributed*

Fortunately, MongoDB will automatically rebalance a sharded collection whenever a large enough disparity is detected between shards. The threshold for this disparity depends on the number of total chunks. For example, if there are 80 or more chunks, the threshold will be a difference of eight between the most chunks on a shard and the least. For between 20 and 80, the threshold is four, and if there are fewer than 20 chunks, the threshold is two.

If this disparity is detected, the shard balancer will begin to migrate chunks to rebalance the distribution of data. This migration might be triggered by large amounts of new data being inserted within a specific range or simply by the addition of a shard. A new shard is initially empty and therefore causes a large disparity in chunk distribution that requires rebalancing.

The `balancerStatus` command allows you to see the current balancer status:

```
mongos> db.adminCommand({ balancerStatus: 1})
{
        "mode" : "full",
        "inBalancerRound" : false,
        "numBalancerRounds" : NumberLong(64629),
        "ok" : 1,
        "operationTime" : Timestamp(1604706062, 1),
        . . .
}
```

In the preceding output, the `mode` field indicates that the balancer is enabled, and the `inBalancerRound` field indicates the balancer is not currently distributing chunks.

Although MongoDB automatically handles the rebalancing, rebalancing does not come without performance implications. Bandwidth, workload, and disk space usage will all increase during chunk migration. To mitigate this performance hit, MongoDB will only migrate a single shard at a time. Additionally, each shard can only participate in one migration at a time. If the impact of chunk migrations is affecting your application performance, there are a few things to try:

- Modifying the balancer window
- Manually enabling and disabling the balancer
- Changing the chunk size

We'll discuss each of these options in the following few pages.

Modifying the Balancer Window

The balancer window defines the time periods during which the balancer will be active. Modifying the balancer window will prevent the balancer from running outside of a given time window; for example, you may only want to balance chunks when application load is at its lowest. In this example, we limit rebalancing to a 90-minute window starting at 10:30 PM:

```
mongos> use config
switched to db config
mongos> db.settings.update(
... { _id: "balancer" },
... { $set: {activeWindow :{ start: "22:30", stop: "23:59" } } },
... { upsert: true })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Note When selecting a balancing window, you must ensure that enough time is provided to balance all the new documents from that day. If your window is too small, there will be a cumulative effect of leftover chunks which will increasingly unbalance your shards.

Disabling the Balancer

It is possible to disable the balancer and re-enable it later. You could, for instance, disable the balancer during a nightly batch window that modifies lots of documents because you don't want the balancer to "thrash" during the process.

Be careful when using this approach, however, as failing to re-enable the balancer could cause shards becoming heavily out of balance. Here's some code showing the balancer being stopped and restarted:

```
mongos> sh.getBalancerState()
true
mongos> sh.stopBalancer()
{
        "ok" : 1,
```

```
        "operationTime" : Timestamp(1604706472, 3),
        . . .
}
mongos> sh.getBalancerState()
false
mongos> sh.startBalancer()
{
        "ok" : 1,
        "operationTime" : Timestamp(1604706529, 3),
        . . .
        }
mongos> sh.getBalancerState()
true
```

Note Migrations may still be in progress after the balancer is disabled. You may need to wait until `sh.isBalancerRunning()` returns `false` to be sure that the balancer has completely stopped.

Changing the Chunk Size

The `chunksize` option – 64MB by default – will determine how large a chunk will grow before being split. By reducing the `chunksize` option, you will have a larger number of small chunks. This will increase migrations and query routing time, but also provide a more even distribution of data. By increasing the chunk size, you will have fewer, larger chunks; this will be more efficient in terms of migrations and routing but may result in a larger proportion of your data sitting in a single chunk. This option won't take effect immediately, and you will have to update or insert into an existing chunk to trigger a split.

Note Once chunks are split, they cannot be recombined by increasing the `chunksize` option, so be careful when reducing this parameter. Additionally, sometimes a chunk may grow beyond this parameter but cannot be split because all the documents have the same shard key. These unsplittable chunks are known as jumbo chunks.

Each of these rebalancing options involves a trade-off between maintaining cluster balance and optimizing the overhead of rebalancing. Continual rebalancing might create a noticeable drag on your throughput, while allowing a cluster to become out of balance may create a performance bottleneck on a single shard. There's no "one-size-fits-all" solution, but establishing a maintenance window for rebalancing operations is a low-risk and low-impact way of ensuring that rebalancing operations do not cause performance degradation during peak periods.

Tip Establishing a maintenance window for rebalancing operations is often the best way to maintain cluster balance while avoiding excessive rebalancing overhead.

Before using any of these methods to control the balancer directly, try to avoid the shards getting out of balance in the first place! Careful selection of a well-distributed shard key is a good first step. Hashed shard keys might also be worth considering if a cluster is experiencing a continually high rebalancing overhead.

Changing Shard Keys

If you have determined that a poorly chosen shard key is creating performance overheads, there are ways to change that shard key. Changing or recreating your shard key is not an easy or quick process in MongoDB. There is no automatic process or command you can run. The process of changing the shard key for a collection is even more work than creating it in the first place. The procedure to change an existing shard key is to

1. Back up your data
2. Drop the entire collection
3. Create a new shard key
4. Import the old data

As you can imagine, with large datasets, this can be a prolonged and tedious process.

This awkward procedure makes it even more important to consider, design, and implement a good shard key from the beginning. If you're not sure you have the right shard key, it can be useful to create a test collection with a smaller subset of the data.

You can then create and recreate the shard keys while observing the distribution. Just remember, when selecting the subset of data to test on, it must be representative of the whole dataset, not just a single chunk.

Although MongoDB does not explicitly support changing shard keys, starting in version 4.4, it does support a method for improving existing sharded collection performance without fully recreating it. In MongoDB, this is called *refining* a shard key.

When refining a shard key, we can add additional fields to the shard key, but not remove or edit the existing fields. These suffix fields can be added to increase the granularity and reduce the size of our chunks. Remember, the balancer cannot split or move jumbo chunks (chunks larger than the `chunksize` option) consisting of documents for a single shard key. By refining our shard key, we may be able to break a jumbo chunk into many smaller chunks which can then be rebalanced.

Imagine our application was relatively small, and initially, sharding by the `country` field was good enough. However, as our application grew, we have a lot of users in a single country, creating jumbo chunks. By refining this shard key with the `district` field, we have increased the granularity of our chunks and thus removed the permanent imbalance created by jumbo chunks.

Here is an example of refining the `country` shard key with the `district` attribute:

```
mongos> db.adminCommand({
   refineCollectionShardKey:
     "MongoDBTuningBook.customersSCountry",
     key: {
         Country: 1, District: 1}
})
{
        "ok" : 1,
        "operationTime" : Timestamp(1604713390, 40),
        . . .
}
```

Note To refine a shard key, you must ensure that a matching index exists on the new shard key attributes. For example, in the preceding code snippet, an index must exist on `{Country: 1, District: 1}`.

Keep in mind that refining a shard key will not have an immediate effect on the data distribution: it will merely increase the ability for the balancer to split and rebalance existing data. Furthermore, newly inserted data will be of finer granularity, and this should lead to fewer jumbo chunks and more balanced sharding.

Sharded Queries

Sharding might help you escape a write bottleneck, but if critical queries are negatively affected, then your sharding project is unlikely to be deemed a success. We want to be sure that sharding is not causing any degradation in queries.

Sharded Explain Plans

As usual, we can use the `explain()` method to see how MongoDB will execute a request – even if the request is executed across multiple nodes of a sharded cluster. Generally, we'll want to use the `executionStats` option when looking at a sharded query, since only that option will show us how work was distributed across the cluster.

Here's an example of the `executionStats` section for a sharded query. Within the output, we should see a `shards` step, which has child steps for each shard. Here's a truncated version of explain output for a sharded query:

```
var exp=db.customers.explain('executionStats').
    find({'views.title':'PRINCESS GIANT'}).next();
```

```
mongos > exp.executionStats {
    "nReturned": 17874,
    "executionTimeMillis": 9784,
    "executionStages": {
        "stage": "SHARD_MERGE",
        "nReturned": 17874,
        "executionTimeMillis": 9784,
        "shards": [
            {"shardName": "shard01",
             "executionStages": {
                 "stage": "SHARDING_FILTER",
                  "inputStage": {
```

```
                "stage": "COLLSCAN"}}},
            {"shardName": "shard02",
             "executionStages": {
                "stage": "SHARDING_FILTER",
                 "inputStage": {
                    "stage": "COLLSCAN"}}}}}
```

This plan shows that the query was resolved by performing collection scans on each shard, then merging the results – SHARD_MERGE – before returning the data to the client.

Our tuning script (see Chapter 3) generates an easy-to-read execution plan for a sharded query. Here's an example of this output which shows the plans on each shard:

```
mongos> var exp=db.customers.explain('executionStats').
      find({'views.title':'PRINCESS GIANT'}).next();

mongos> mongoTuning.executionStats(exp)

1     COLLSCAN ( ms:4712 returned:6872 docs:181756)
2    SHARDING_FILTER ( ms:4754 returned:6872)
3   Shard ==> shard01 ()
4     COLLSCAN ( ms:6395 returned:11002 docs:229365)
5    SHARDING_FILTER ( ms:6467 returned:11002)
6   Shard ==> shard02 ()
7  SHARD_MERGE ( ms:6529 returned:17874)

Totals:  ms: 6529  keys: 0  Docs: 411121
```

The SHARD_MERGE step occurs when we combine output from multiple shards. It indicates that the mongos router received data from multiple shards and combined them into unified output.

However, if we issue a query filtered against the shard key, then we may see a SINGLE_SHARD plan. In the following example, the collection was sharded on LastName, so the mongos was able to retrieve all the needed data from a single shard:

```
mongos> var exp=db.customersShardName.explain('executionStats').
      find({'LastName':'HARRISON'})

mongos> mongoTuning.executionStats(exp)
```

```
1      IXSCAN ( LastName_1_FirstName_1 ms:0
                returned:730 keys:730)
2     SHARDING_FILTER ( ms:0 returned:730)
3    FETCH ( ms:149 returned:730 docs:730)
4   Shard ==> shard01 ()
5  SINGLE_SHARD ( ms:158 returned:730)

Totals:  ms: 158  keys: 730  Docs: 730
```

Shard Key Lookups

As we've seen, when a query contains the shard key, MongoDB may be able to satisfy the query from a single shard.

For instance, if we have sharded on `LastName`, then a query on `LastName` resolves as follows:

```
mongos> var exp=db.customersSLName.explain('executionStats').
              find({LastName:'SMITH','FirstName':'MARY'});

mongo> mongoTuning.executionStats(exp);
1      IXSCAN ( LastName_1 ms:0 returned:711 keys:711)
2     FETCH ( ms:93 returned:9 docs:711)
3    SHARDING_FILTER ( ms:93 returned:9)
4   Shard ==> shardRS ( ms:97 returned:9)
5  SINGLE_SHARD ( ms:100 returned:9)

Totals:  ms: 100  keys: 711  Docs: 711
```

However, note that in the preceding example, we lack a combined index on `LastName` and `FirstName` so the query is less efficient than it might be. We should refine the shard key to include the `FirstName`, or we can simply create a new compound index on both attributes:

```
mongo> var exp=db.customersSLName.explain('executionStats').
             find({LastName:'SMITH','FirstName':'MARY'});

mongo> mongoTuning.executionStats(exp);

1      IXSCAN ( LastName_1_FirstName_1 ms:0 returned:9 keys:9)
```

```
2     SHARDING_FILTER ( ms:0 returned:9)
3    FETCH ( ms:0 returned:9 docs:9)
4   Shard ==> shardRS ( ms:1 returned:9)
5  SINGLE_SHARD ( ms:2 returned:9)

Totals:  ms: 2  keys: 9  Docs: 9
```

Tip If a query contains the shard key and additional filter conditions, you can optimize the query by creating an index that includes both the shard key and those additional attributes.

Accidental Shard Merge

Wherever possible, we want to send queries to a single shard. To achieve this, we should make sure that our shard key is aligned with our query filters.

For instance, if we shard by `Country`, but query by `City`, MongoDB will need to do a shard merge, even though all the documents for a given City will be in the shard that contains that City's Country:

```
mongo> var exp=db.customersSCountry.explain('executionStats').
            find({City:"Hiroshima"});

mongo> mongoTuning.executionStats(exp);

1      IXSCAN ( City_1 ms:0 returned:544 keys:544)
2     FETCH ( ms:0 returned:544 docs:544)
3    SHARDING_FILTER ( ms:0 returned:0)
4   Shard ==> shardRS ( ms:2 returned:0)
5      IXSCAN ( City_1 ms:0 returned:684 keys:684)
6     FETCH ( ms:0 returned:684 docs:684)
7    SHARDING_FILTER ( ms:0 returned:684)
8   Shard ==> shardRS2 ( ms:2 returned:684)
9  SHARD_MERGE ( ms:52 returned:684)

Totals:  ms: 52  keys: 1228  Docs: 1228
```

It may have been better to shard by City, not Country – since City has a higher cardinality. However, in this case, it's equally effective to simply add Country to the query filter:

```
mongo> var exp=db.customersSCountry.explain('executionStats').
           find({Country:'Japan',City:"Hiroshima"});

mongo> mongoTuning.executionStats(exp);

1      IXSCAN ( City_1 ms:0 returned:684 keys:684)
2     FETCH ( ms:0 returned:684 docs:684)
3    SHARDING_FILTER ( ms:0 returned:684)
4   Shard ==> shardRS2 ( ms:2 returned:684)
5  SINGLE_SHARD ( ms:55 returned:684)

Totals:  ms: 55  keys: 684  Docs: 684
```

Tip Whenever it makes sense, add the shard key to queries that execute against a sharded cluster. If the shard key is not included in a query filter, then the query will be sent to all shards even if the data is only present in one of the shards.

Shard Key Range

If the shard key is range sharded, then we can use the key to perform an index range scan. For instance, in this example, we have sharded orders by orderDate:

```
mongo> var startDate=ISODate("2018-01-01T00:00:00.000Z");
mongo> var exp=db.ordersSOrderDate.explain('executionStats').
           find({orderDate:{$gt:startDate}});

mongo> mongoTuning.executionStats(exp);

1      IXSCAN ( orderDate_1 ms:0 returned:7191 keys:7191)
2     SHARDING_FILTER ( ms:0 returned:7191)
3    FETCH ( ms:0 returned:7191 docs:7191)
4   Shard ==> shardRS2 ( ms:16 returned:7191)
5  SINGLE_SHARD ( ms:68 returned:7191)

Totals:  ms: 68  keys: 7191  Docs: 7191
```

However, if hash sharding is implemented, then collection scans in every shard are required:

```
mongo> var exp=db.ordersHOrderDate.explain('executionStats').
      find({orderDate:{$gt:startDate}});
mongo> mongoTuning.executionStats(exp);

1     COLLSCAN ( ms:1 returned:2615 docs:28616)
2    SHARDING_FILTER ( ms:1 returned:2615)
3   Shard ==> shardRS ( ms:17 returned:2615)
4     COLLSCAN ( ms:1 returned:4576 docs:29881)
5    SHARDING_FILTER ( ms:1 returned:4576)
6   Shard ==> shardRS2 ( ms:20 returned:4576)
7  SHARD_MERGE ( ms:72 returned:7191)

Totals:  ms: 72  keys: 0  Docs: 58497
```

Tip If you frequently perform range scans on the sharding key, range sharding is preferable to hash sharding. However, remember that range sharding can lead to hot spots if the key values are constantly incrementing.

Sorting

When sorted data is retrieved from more than one shard, the sort operation occurs in two stages. First, data is sorted on each shard and then returned to the `mongos` where a `SHARD_MERGE_SORT` combines the sorted inputs into a consolidated, sorted output.

Indexes that exist to support the sort – including the shard key index if appropriate – can be used on each shard to facilitate the sort, but even if you are sorting by shard key, a final sort operation must still be performed on the `mongos`.

Here's an example of a query which sorts orders by `orderDate`. The shard key is used to return data in sorted order from each shard before a final `SHARD_MERGE_SORT` is performed on the mongos:

```
1      IXSCAN ( orderDate_1 ms:22 returned:527890 keys:527890)
2     SHARDING_FILTER ( ms:58 returned:527890)
```

```
3    FETCH ( ms:87 returned:527890 docs:527890)
4   Shard ==> shardRS2 ( ms:950 returned:527890)
5      IXSCAN ( orderDate_1 ms:29 returned:642050 keys:642050)
6     SHARDING_FILTER ( ms:58 returned:642050)
7    FETCH ( ms:102 returned:642050 docs:642050)
8   Shard ==> shardRS ( ms:1011 returned:642050)
9  SHARD_MERGE_SORT ( ms:1013 returned:1169940)

Totals:  ms: 1013  keys: 1169940  Docs: 1169940
```

If there is no appropriate index to support the sort, then blocking sorts will need to be performed on each shard:

```
1      COLLSCAN ( ms:37 returned:564795 docs:564795)
2     SHARDING_FILTER ( ms:70 returned:564795)
3    SORT ( ms:237 returned:564795)
4   Shard ==> shardRS ( ms:1111 returned:564795)
5      COLLSCAN ( ms:30 returned:605145 docs:605145)
6     SHARDING_FILTER ( ms:78 returned:605145)
7    SORT ( ms:273 returned:605145)
8   Shard ==> shardRS2 ( ms:1315 returned:605145)
9  SHARD_MERGE_SORT ( ms:1363 returned:1169940)

Totals:  ms: 1363  keys: 0  Docs: 1169940
```

The normal considerations for optimizing sorts apply to each of the shard sorts. In particular, you need to make sure you don't exceed the sort memory limit on each shard – see Chapter 6 for more details.

Non-Shard Key Lookups

If a query does not include a shard key predicate, then the query is sent to each shard, and the results merged back on the mongos. For instance, here we perform a collection scan on each shard and merge the results in the SHARD_MERGE step:

```
mongo> var exp=db.customersSCountry.explain('executionStats').
                  find({'views.filmId':637});

mongo> mongoTuning.executionStats(exp);
```

```
1     COLLSCAN ( ms:648 returned:10331 docs:199078)
2    SHARDING_FILTER ( ms:648 returned:10331)
3   Shard ==> shardRS ( ms:1602 returned:10331)
4     COLLSCAN ( ms:875 returned:4119 docs:212043)
5    SHARDING_FILTER ( ms:882 returned:4119)
6   Shard ==> shardRS2 ( ms:1954 returned:4119)
7  SHARD_MERGE ( ms:2002 returned:14450)

Totals:  ms: 2002  keys: 0  Docs: 411121
```

There's nothing wrong with a `SHARD_MERGE` – we should totally expect that many queries will need to resolve in this manner. However, you should make sure that the query that runs on each shard is optimized. In the preceding example, a need for an index on `views.filmId` is clearly indicated.

Tip For queries that must be executed against every shard, ensure that each shard's workload is minimized using the indexing and document design principles outlined in previous chapters.

Aggregations and Sorts

When performing aggregation operations, MongoDB tries to push as much work as possible to the shards. The shards are responsible not just for the data access portions of the aggregation (such as `$match` and `$project`) but also pre-aggregations necessary to satisfy `$group` and `$unwind` operations.

The explain plan for a sharded aggregation includes unique sections to illustrate how the aggregation was resolved.

For instance, consider this aggregation:

```
db.customersSCountry.aggregate([
  { $unwind:  "$views" },
  { $group:{     _id:{ "views_title":"$views.title"  },
             "count":{$sum:1}
      }
  },
]);
```

An execution plan for this aggregation contains a unique section showing how the work will be split across the aggregation:

```
"mergeType": "mongos",
"splitPipeline": {
  "shardsPart": [
    {
      "$unwind": {
        "path": "$views"
      }
    },
    {
      "$group": {
        "_id": {
          "views_title": "$views.title"
        },
        "count": {
          "$sum": {
            "$const": 1
          }
        }
      }
    }
  ],
  "mergerPart": [
    {
      "$group": {
        "_id": "$$ROOT._id",
        "count": {
          "$sum": "$$ROOT.count"
        },
        "$doingMerge": true
      }
    }
  ]
},
```

The mergeType section tells us which component will perform the merge. We expect to see mongos here, but in some circumstances, we might see the merge allocated to one of the shards, in which case we'd see "primaryShard" or "anyShard".

The splitPipeLine shows the aggregation stages that are sent to the shards. In this example, we can see that the $group and $unwind operations will be performed on the shards.

Finally, mergerPart shows us what operations will occur in the merging node – in this case, on the mongos.

For the most commonly used aggregate steps, MongoDB will push down the majority of work to the shards and combine output on the mongos.

Sharded $lookup Operations

Join operations using $lookup are only partially supported on sharded collections. The collection referenced in the from section of the $lookup stage *cannot* be sharded. Consequently, The work of the $lookup cannot be distributed across the shard. All the work will occur on the master shard that contains the lookup collection.

Warning $lookup is not fully supported on sharded collections. A collection referenced in a $lookup pipeline stage cannot be a sharded collection, although the initiating collection may be sharded.

Summary

Sharding provides a scale-out solution for very large MongoDB implementations. In particular, it allows the write workload to be spread across multiple nodes. However, sharding adds operational complexity and performance overhead and should not be implemented lightly.

The most important consideration for sharded cluster implementation is to pick a shard key with care. The shard key should have a high cardinality to allow for chunks to split as data grows, should support queries that can operate against individual shards, and should distribute workload evenly across shards.

Rebalancing is a background operation that MongoDB performs to keep shards balanced. Rebalancing operations can cause performance degradation: you may wish to tweak rebalancing to avoid this or limit rebalancing to a maintenance window.

Query tuning on a sharded cluster is driven by most of the same considerations that exist for single node MongoDB – indexing and document design are still the most important factors. However, you should make sure that queries that can include the shard key do include that key and that indexes exist to support the queries that are routed to each shard.