|  |  |
| --- | --- |
| -  Sindy Saintclair  Sunday, January 31st, 2022  Lesson 5: Sharding, More Methods and Projects | |
| **Learning Objectives and Questions** | **Notes and Answers** |
| **OVERVIEW** | I will be learning about a few more in-depth NoSQL terms and methods. I will also be working on an in-depth Lesson 5 HandsOn for NoSQL. It is time to dive right into Sharding. |
| **Sharding** | **Sharding** is a way to spread data across multiple machines and servers. MongoDB uses Sharding to support deployments and applications that contain huge data sets. This is because when database systems have large data sets, a single server may have trouble keeping up with all the data. There are two ways to deal with a situation like this: *Vertical* or *Horizontal* Scaling.  ***Vertical Scaling***  **Vertical Scaling** involves ways to increase the capacity of a server, such as using a much more powerful CPU, adding more RAM, or increasing the amount of storage space. There are limitations when using *Vertical Scaling*  because there may be restrictions on how much storage one machine can handle. Also, cloud-based providers have a maximum for how much storage they have.  ***Horizontal Scaling***  **Horizontal Scaling** is the process of spreading out the dataset between multiple servers and increasing the storage to those servers as needed. Even if a single machine out of the many handling the data may not be super highspeed, overall, it may increase the efficiency of the application having many machines. If the dataset expands, all that is needed is to add servers to handle that data as needed. MongoDB supports *Horizontal Scaling* through *Sharding*.  ***Enable Sharding***  **Sharding** is something that is done at a very high level in your database, usually on theadmin side of the database. The following command is used when you would like to create Sharding in your database:  db.runCommand({  shardCollection: "<database>.<collection>",  key: <shardkey>,  unique: <boolean>,  numInitialChunks: <integer>,  collation: { locale: "simple" }  })  As you can see, there are several options available to you when running this command; however, only the last is optional. Now it’s time to explore these parts:   * **shardCollection**: How do you name which collection in which database you would like to shard. It will always be a string. * **key**: the index specification document to use as the shard key. The shard key determines how MongoDB distributes the documents among the shards * **unique**: When true, the unique option ensures that the underlying index enforces a unique constraint. Hashed shard keys do not support unique constraints. Defaults to false. * **numInitialChunks**: Specifies the number of chunks to initially create when sharding a collection that is empty with a hashed shard key. Then, MongoDB will create and balance chunks across the cluster. The numInitialChunks must be less than 8192 per shard * MongoDB divides shared data into chunks. Each chunk has an inclusive lower and exclusive upper range based on the shard key * **collation**: *Optional*. If the collection specified to shardCollection has a default collation, you must include a collation document with { locale : "simple" }, or the shardCollection command fails. At least one of the indexes whose fields support the shard key pattern must have a simple collation. * Collation allows users to specify language-specific string comparison rules, such as letter case and accent marks. |
| **More Methods** | Now that I have made it this far in NoSQL, it is time to look into a few more available methods when working with a collection. Some of these methods can be in-depth, but it is good to know they are available to you.  ***aggregate( )***  This method calculates the aggregate (total) values for data in a collection. Below is the syntax:  db.collectionName.aggregate(pipeline, options);  Below is a description of the parameters of the above query:   * **pipeline**: an array that is a sequence of data aggregation operations or stages * 🡪$**project**: reshapes each document in the stream by adding new fields or removing existing fields * 🡪$**match**: filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage * 🡪$**redact**: reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of $project and $match. Can be used to implement field level redaction. For each input document, outputs either ne or zero document. * 🡪$**limit**: passes the first *n* documents unmodified to the pipeline where *n*  is the specified limit. For each input document, outputs either document (for the first *n* documents) or zero documents (after the first *n* documents). * 🡪$**skip**: skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents). * 🡪$**unwind**: deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array * 🡪$**group**: groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields. * 🡪$**sort**: reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document * 🡪$**geoNear**: returns an ordered stream of documents based on the proximity to a geospatial data. The output document include an additional distance field and can include a location identifier field. * 🡪$**out**: writes the resulting documents of the aggregation pipeline to a collection. TO use the $out stage, it must be the last stage in the pipeline. * **options**: *optional,* additional documents that are passed in when using aggregate * 🡪**explain**: specifies to return the information on the processing of the pipeline * 🡪**allowDiskUse**: enables writing to temporary files. When set to true, aggregation operations can write data to the \_tmp subdirectory in the dbPath directory. * 🡪**cursor**: specifies the *initial* batch size for the cursor. The value of the cursor field is a document with the field batchSize.   ***count( )***  This method will count and return the number of results based on a query. The syntax is below:  db.collectionName.count();  For example, if you wanted to count the number of documents in your inventory collection, you would run the following:  db.inventory.count();  The query above will return 10, or however many documents are currently in the inventory collection.  You could also run this query with a filter. Check to see how many of your app users in your appusers collection have an age greater than 20 by running the below query:  db.appusers.count( { age: { $gt : 20 } } )  After running the above query, it should return the number 4 or a number close, depending on your changes in that collection.  ***totalSize( )***  This method will return the total size in bytes of the data in the collection plus the size of every index on the collection. If you run the query below, a number around 16000 will be returned based on what your collection currently contains:  db.appusers.totalSize()  ***copyTo( )***  Deprecated. wraps eval to copy data between collections in a single MongoDB instance  db.appusers.copyTo()  ***dataSize( )***  Returns the size of the collection. Wraps the size field in the output of the collStats.  db.appusers.dataSize()  ***distinct( )***  Returns an array of documents that have distinct values for the specified field.  db.appusers.distinct()  ***findAndModify( )***  Atomically modifies and returns a single document  db.appusers.findAndModify()  ***getShardDistribution( )***  For collections in sharded clusters, this string reports data of chunk distribution  db.appusers.getShardDistribution()  ***getShardVersion( )***  Internal diagnostic method for shard cluster.  db.appusers.getShardVersion()  ***group( )***  Provides simple data aggregation function. Groups documents in a collection by a key and processes the results. Use aggregate( ) for more complex data aggregation.  db.appusers.group()  ***isCapped( )***  Reports if a collection is a capped collection.  db.appusers.isCapped()  ***mapReduce( )***  Performs map-reduce style data aggregation.  db.appusers.mapReduce()  ***reindex( )***  Rebuilds all existing indexes on a collection.  db.appusers.reIndex()  ***renameCollection( )***  Changes the name of a collection.  db.appusers.renameCollection()  ***save( )***  Provides a wrapper around an insert( ) and update( ) to insert new documents  db.appusers.save()  ***stats( )***  Reports on the state of a collection. Provides a wrapper around the collStats.  db.appusers.stats()  ***storageSize( )***  Reports the total size used by the collection in bytes. Provides a wrapper around the storageSize field of the collStats output  db.appusers.storageSize()  ***totalIndexSize( )***  Reports the total size used by the indexes on a collection. Provides a wrapper around the totalIndexSize field of the collStats output  db.appusers.totalIndexSize()  ***validate( )***  Performs diagnostic operations on a collection.  db.appusers.validate() |