|  |  |
| --- | --- |
| -  Sindy Saintclair  Saturday, January 29, 2022  Lesson 4 : Deleting and Indexing Documents | |
| **Learning Objectives and Questions** | **Notes and Answers** |
| **OVERVIEW** | I have made it to the final CRUD operation (Delete) in NoSQL. This operation is relatively straightforward because there is only so much you can do when deleting documents. Time for me to dive in! |
| **Delete All Documents** | If you wanted to remove all documents from a collection, pass in an empty filter document with empty curly braces ( {} ). See below:  db.collectionName.deleteMany({})  The query above would delete all documents from a collection.  Now its your turn to try it! Run the following query:  db.cars.insertMany([  {  make: 'Hyundai',  model: 'Santa Fe',  price: 8000,  year: 2003,  used: true,  color: 'Black'  },  {  make: 'BMW',  model: 'ALPINA B6 Gran Coupe',  price: 124300,  year: 2017,  used: false,  color: 'Mediterranean Blue Metallic'  },  {  make: 'Subaru',  model: 'Crosstrek 2.0i Premium',  price: 22595,  year: 2014,  used: true,  color: 'Sunshine Orange'  },  {  make: 'Ford',  model: 'F-350 XL',  price: 33705,  year: 2017,  used: false,  color: 'Race Red'  },  {  make: 'Toyota',  model: 'Acura MDX',  price: 28800,  year: 2014,  used: true,  color: 'Graphite Luster Metallic'  },  {  make: 'Volkswagen',  model: 'Jetta 1.4T S',  price: 19495,  year: 2018,  used: false,  color: 'Silk Blue Metallic'  }  ]);  Now you have a cars collection within your database. If you want to delete those documents, run the following query:  db.cars.deleteMany({})  As you can see, the collection cars still exists, but it now contains zero documents.  ***Delete Documents with Filter***  Just like updating and reading documents, you can pass in a filter to identify the documents that need to be deleted. The syntax is the same as you have seen in previous lessons.  Go ahead and re-insert the cars documents that were given to me above.  You can specify equality conditions like below:  db.cars.deleteMany({ used : true })  The above query will delete all cars in your collection that are used.  You can also filter using query operators you have learned previously:  db.cars.deleteMany({ price: { $lt : 30000 }})  Above, you are deleting all documents that have a price that is less than $30K.  ***Delete One Document***  If you wanted to delete only one document, you could use the deleteOne( ) method. The code below will delete the first document that is used:  db.cars.deleteOne({ used : true })  You can again use query operators to delete documents:  db.cars.deleteOne({ price: { $lt : 30000 }})  The above query will delete the first document that has a price of less than $30K.  Great! Now you know how to delete documents. Be careful when deleting by making sure your filters are exact, so you don’t delete something you want to keep. Feel free to practice deleting with filters.  ***Find and Delete***  The findOneAndDelete() works the same way as the methods above, but this will return the document you are deleting right before it deletes it. Run the following query:  db.cars.findOneAndDelete({ price: 8000 });  After running the above query, you will see:  {  "\_id": {  "$oid": "6102f827e0e1d80d2f03ac5c"  },  "make": "Hyundai",  "model": "Santa Fe",  "price": 8000,  "year": 2003,  "used": true,  "color": "Black"  }  And if you run a find( ) query or look in Atlas, you will see that document no longer exists.  ***Delete a Collection***  Now, if you wanted to delete a collection, you can run the following query:  db.collectionName.drop();  That will delete the collection specified along with all its documents. It will return true if the collection has been dropped or will return false if no collection of that name exists.  There should still be a collection named products from the previous lessons. Time for you to practice this query with that collection. Go ahead and run the following query, the nlook in Atlas to see if the collection has been deleted:  db.products.drop(); |
| **Indexes** | Indexing improves the execution of queries in MongoDB. If you do not have an index, MongoDB must perform a *collection scan* which scans every document in the collection to find the documents which match the query. This causes the query to take longer than it would if it had an index. If an index does not exist, MongoDB can use that index to limit the number of documents it must inspect.  **Indexes** store small options of documents in a form that is easy to traverse. The index stores a specific field or set of fields and is then ordered by the field. This ordering makes the search for the query match much more efficient.  ***Default Index***  When creating a new collection, MongoDB creates a unique index on the \_id  field. This prevents clients from inserting two documents with the same value in the \_id field. This index that is created on the \_id field cannot be dropped.  ***Create an Index***  When you want to create an index, use the following syntax:  db.collection.createIndex( <key and index type specification>, <options> )  This query will only create an index if an index with the same specifications does not exist already.  Now that you understand what an index is in MongoDB, it is time to explore the different ypes of indexes available. |
| **Single and Compound Indexing** | MongoDB provides several different types of indexes that support specific types of data and queries. Time to explore these indexes.  ***Single Field Indexing***  MongoDB supports the creation of ascending/descending indexes on a single field of a document.  If you wanted to create an index on a field in ascending order, you would use the number 1 to define that.  See below:  db.records.createIndex( { score: 1 } )  The above query creates an index on the score field in the records collection in *ascending* order.  If you wanted the index to be in *descending* order, you would use -1 as the value:  db.records.createIndex( { score: -1 } )  ***Compound Indexing***  Compound indexing is where a single index structure references multiple fields within a collection’s document. To create a compound index, use the following syntax:  db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )  Again, the field’s value within the createIndex will define either ascending (using 1) or descending (using -1) order.  Consider a collection named products that holds the below documents:  {  "\_id": ObjectId(...),  "item": "Banana",  "category": ["food", "produce", "grocery"],  "location": "4th Street Store",  "stock": 4,  "type": "cases"  }  If you want to create two ascending indexes on the item and stock fields, it will look like this:  db.products.createIndex( { item: 1, stock: 1 } )  When creating a compound index, the order in which you list the fields indexed is important. The index will sort the values of the first field, and then within each value of the first field, it will sort the values of the second field. Along with supporting queries that match all field defined within the index, compound indexes can support queries that match the first field in the index. The index above will support queries on the item field and the item and stock fields. See below:  db.products.find( { item: "Banana" } )  db.products.find( { item: "Banana", stock: { $gt: 5 } } ) |
| **Indexing on Embedded Fields and Documents** | When creating indexes, you may run into wanting to create one on a field that is either an embedded document as a whole or an embedded field. Now you will explore how.  ***Create Index on Embedded Field***  You can create indexes on embedded fields just like you can on top-level fields. Indexes on embedded fields are different from indexes on embedded documents, which you will explore next. When creating an index on an embedded field, you need to use the dot notation to define where the embedded document is located. For example, if you had the following document:  {  "\_id": ObjectId("570c04a4ad233577f97dc459"),  "score": 1034,  "location": { state: "NY", city: "New York" }  }  And you wanted to create an *ascending*  index on the state field in the location document, and it would look like this:  db.records.createIndex( { location.state: 1 } )  ***Create Index on Embedded Document***  You can also create indexes on embedded documents as a whole. Look at the same example as above:  {  "\_id": ObjectId("570c04a4ad233577f97dc459"),  "score": 1034,  "location": { state: "NY", city: "New York" }  }  The location field is an embedded document with two fields: state and city. The following query will create an index on the location field as a whole:  db.records.createIndex( { location: 1 } )  If you want to use that index, you can run the below query:  db.records.find( { location: { city: "New York", state: "NY" } } )  While creating an index on an embedded field or document, you are still performing a Singe Field Index, as you are only adding an index to one field. Next, you will look into creating compound indexes.  ***Multikey Indexes***  When an index is created on a field that has an array as its value, MongoDB creates an index for each element in the array. This is called a Multikey Index. These multikey indexes make queries against arrays much more efficient. They can be created over arrays that hold strings or numbers *and* nested documents. So the array could look like this:  ['Apples', 'Cherries', 'Pineapple'];  or  [  { item: 'Picture Frame', price: 3 },  { item: 'Canvas', price: 15 },  { item: 'Pencils', price: 4 },  ];  To create a Multikey Index takes no extra effort from you. If MongoDB see that you have created an index on a field that holds an array as its value, it will automatically create a Multikey Index.  ***Viewing Your Indexes***  Sometimes, you may want to look at the indexes you have created. You can use the getIndexes( ) method. So far, you haven’t created any indexes on your current data in your database. Now you will do that quickly, then view the indexes you have created.  Run the following query:  db.cars.createIndex({ make: 1, model: -1 });  Above, you have created an index on the make field in *ascending* order and an index on the model field in *descending* order.  Run the following query:  db.cars.getIndexes();  This query returns an array of documents that hold index information for the collection. Index information includes the keys and options used to create the index. You should see the following array as your output:  Get Indexes. Several lines of code that use key ID and db cars.  Figure 4-2: Get Indexes  ***Dropping Indexes***  Sometimes, you may no longer want to keep a specific index. You can drop it using the following queries:  db.collectionName.dropIndex(<indexName>)  <!-- OR -->  db.collectionName.dropIndex( { <indexSpecificationDocument> } )  The index specification ndocument is what you specified the index upon which to be created.  If you need to drop all indexes, you can run the following query:  db.collectionName.dropIndexes();  That will drop every index created, not including the required index on the \_id field. |
| **Viewing Your Indexes** | Sometimes, you may want to look at the indexes you have created. You can use the getIndexes() method. So far, you haven’t created any indexes on your current data in your database. Now you will do that quickly, then view the indexes you have created.  Run the following query:  db.cars.createIndex({ make: 1, model: -1 });  Above, you have created an index on the make field in *ascending* order and an index on the model field in *descending* order.  Run the following query:  db.cars.getIndexes();  This query returns an array of documents that hold index information for the collection. Index information includes the keys and options used to create the index. You should see the following array as your output:  Get Indexes. Several lines of code that use key ID and db cars.  Figure 4-2: Get Indexes  ***Dropping Indexes***  Sometimes, you may no longer want to keep a specific index. You can drop it using the following queries:  db.collectionName.dropIndex(<indexName>)  <!-- OR -->  db.collectionName.dropIndex( { <indexSpecificationDocument> } )  The index specification document is what you specified the index upon which to be created. If you need to drop all indexes, you can run the following query:  db.collectionName.dropIndexes();  That will drop every index created, not including the required index on the \_id field. |