# Data Modeling

# MongoDB sort() method

## Sorting Documents using sort() method

Using sort() method, you can sort the documents in ascending or descending order based on a particular field of document.

**Syntax of sort() method:**

db.collecttion\_name.find().sort({field\_key:1 or -1})

1 is for ascending order and -1 is for descending order. The default value is 1.  
 **For example:** collection studentdata contains following documents:

> db.studentdata.find().pretty()

{

"\_id" : ObjectId("59bf63380be1d7770c3982af"),

"student\_name" : "Steve",

"student\_id" : 2002,

"student\_age" : 22

}

{

"\_id" : ObjectId("59bf63500be1d7770c3982b0"),

"student\_name" : "Carol",

"student\_id" : 2003,

"student\_age" : 22

}

{

"\_id" : ObjectId("59bf63650be1d7770c3982b1"),

"student\_name" : "Tim",

"student\_id" : 2004,

"student\_age" : 23

}

Lets say I want to display the student\_id of all the documents in **descending order**:

To display only a particular field of document, I am using [MongoDB Projection](https://beginnersbook.com/2017/09/mongodb-projection/)

> db.studentdata.find({}, {"student\_id": 1, \_id:0}).sort({"student\_id": -1})

{ "student\_id" : 2004 }

{ "student\_id" : 2003 }

{ "student\_id" : 2002 }

To display the student\_id field of all the students in **ascending order**:

> db.studentdata.find({}, {"student\_id": 1, \_id:0}).sort({"student\_id": 1})

{ "student\_id" : 2002 }

{ "student\_id" : 2003 }

{ "student\_id" : 2004 }

**Default:** The default is ascending order so If I don’t provide any value in the sort() method then it will sort the records in ascending order as shown below:

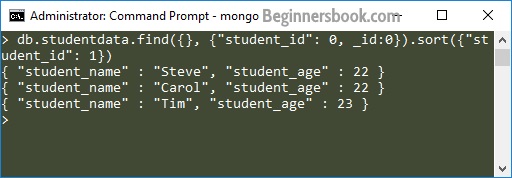
> db.studentdata.find({}, {"student\_id": 1, \_id:0}).sort({})

{ "student\_id" : 2002 }

{ "student\_id" : 2003 }

{ "student\_id" : 2004 }

**You can also sort the documents based on the field that you don’t want to display:** For example, you can sort the documents based on student\_id and display the student\_age and student\_name fields.



# MongoDB Indexing

# createIndex(), dropindex()

Indexes are very important in any database, and with MongoDB it's no different. With the use of Indexes, performing queries in MongoDB becomes more efficient.

If you had a collection with thousands of documents with no indexes, and then you query to find certain documents, then in such case MongoDB would need to scan the entire collection to find the documents. But if you had indexes, MongoDB would use these indexes to limit the number of documents that had to be searched in the collection.

Indexes are special data sets which store a partial part of the collection's data. Since the data is partial, it becomes easier to read this data. This partial set stores the value of a specific field or a set of fields ordered by the value of the field.

**Understanding Impact of Indexes**

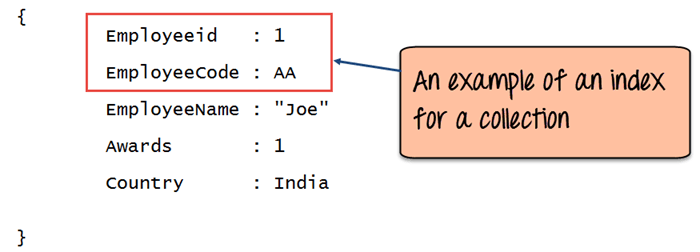
Now even though from the introduction we have seen that indexes are good for queries, but having too many indexes can slow down other operations such as the Insert, Delete and Update operation.

If there are frequent insert, delete and update operations carried out on documents, then the indexes would need to change that often, which would just be an overhead for the collection.

The below example shows an example of what field values could constitute an index in a collection. An index can either be based on just one field in the collection, or it can be based on multiple fields in the collection.

In the example below, the Employeeid "1" and EmployeeCode "AA" are used to index the documents in the collection. So when a query search is made, these indexes will be used to quickly and efficiently find the required documents in the collection.

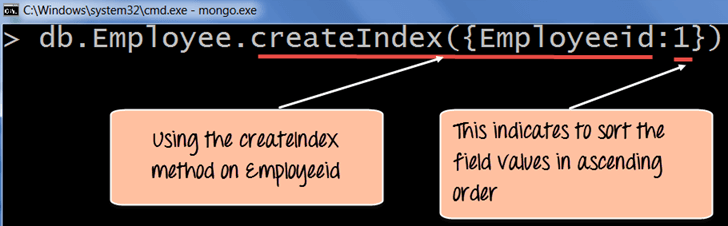
So even if the search query is based on the EmployeeCode "AA", that document would be returned.

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith1.png)

**How to Create Indexes: createIndex()**

Creating an Index in MongoDB is done by using the "**createIndex**" method.

The following example shows how add index to collection. Let's assume that we have our same Employee collection which has the Field names of "Employeeid" and "EmployeeName".

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith2.png)

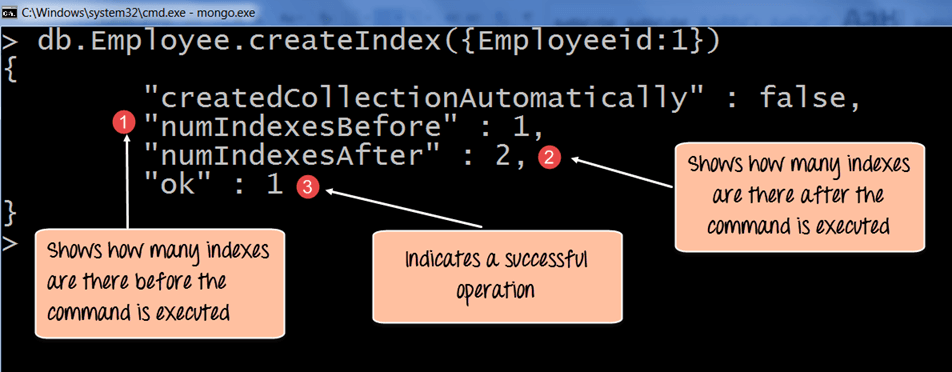
db.Employee.createIndex({Employeeid:1})

**Code Explanation:**

1. The **createIndex** method is used to create an index based on the "Employeeid" of the document.
2. The '1' parameter indicates that when the index is created with the "Employeeid" Field values, they should be sorted in ascending order. Please note that this is different from the \_id field (The id field is used to uniquely identify each document in the collection) which is created automatically in the collection by MongoDB. The documents will now be sorted as per the Employeeid and not the \_id field.

If the command is executed successfully, the following Output will be shown:

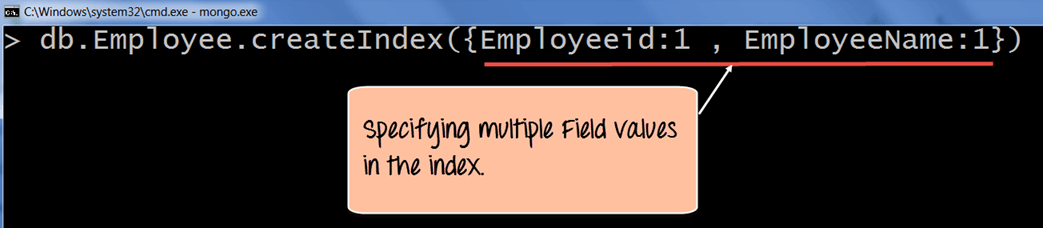
**Output:**

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith3.png)

1. The numIndexesBefore: 1 indicates the number of Field values (The actual fields in the collection) which were there in the indexes before the command was run. Remember that each collection has the \_id field which also counts as a Field value to the index. Since the \_id index field is part of the collection when it is initially created, the value of numIndexesBefore is 1.
2. The numIndexesAfter: 2 indicates the number of Field values which were there in the indexes after the command was run.
3. Here the "ok: 1" output specifies that the operation was successful, and the new index is added to the collection.

The above code shows how to create an index based on one field value, but one can also create an index based on multiple field values.

The following example shows how this can be done;

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith4.png)

db.Employee.createIndex({Employeeid:1, EmployeeName:1])

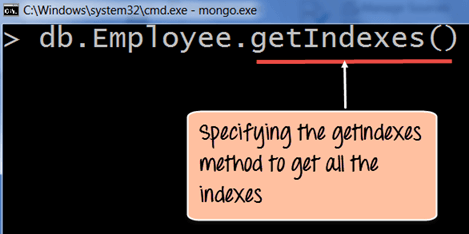
**Code Explanation:**

1. The createIndex method now takes into account multiple Field values which will now cause the index to be created based on the "Employeeid" and "EmployeeName". The Employeeid:1 and EmployeeName:1 indicates that the index should be created on these 2 field values with the :1 indicating that it should be in ascending order.

**How to Find Indexes: getindexes()**

Finding an Index in MongoDB is done by using the **"getIndexes"**method.

The following example shows how this can be done;

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith5.png)

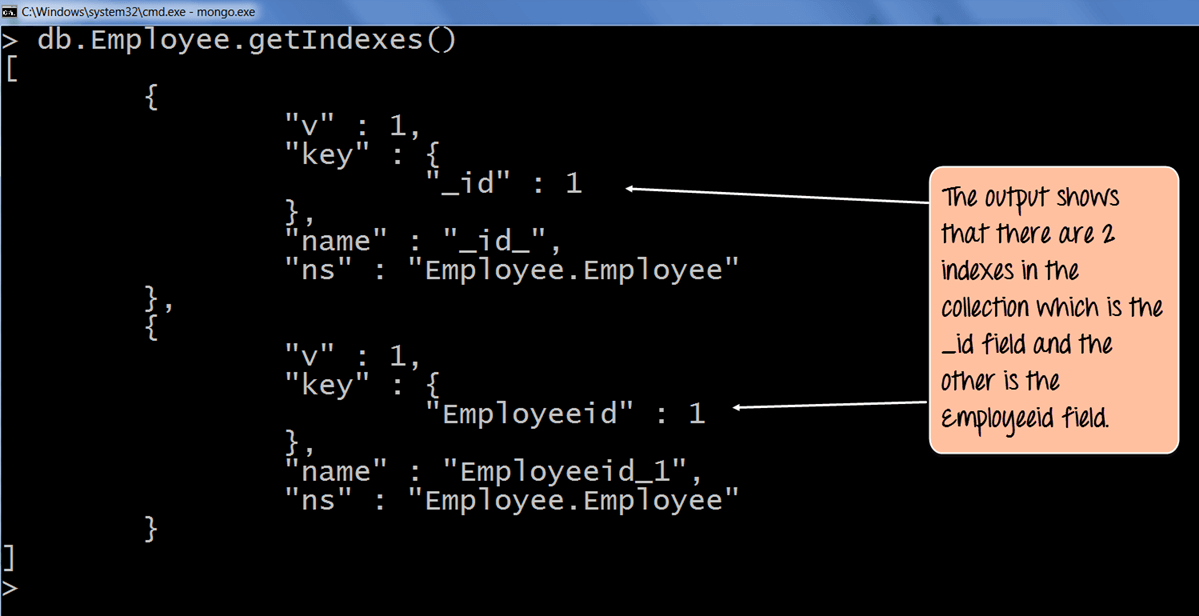
db.Employee.getIndexes()

**Code Explanation:**

1. The getIndexes method is used to find all of the indexes in a collection.

If the command is executed successfully, the following Output will be shown:

**Output:**

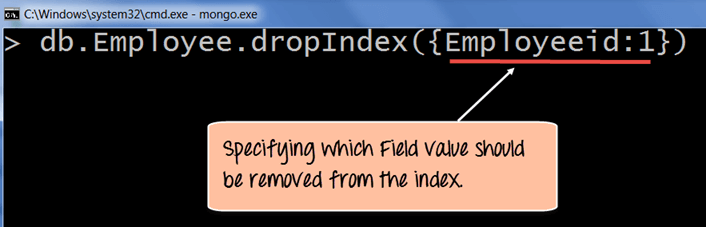
[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith6.png)

1. The output returns a document which just shows that there are 2 indexes in the collection which is the \_id field, and the other is the Employee id field. The :1 indicates that the field values in the index are created in ascending order.

**How to Drop Indexes: dropindex()**

Removing an Index in MongoDB is done by using the dropIndex method.

The following example shows how this can be done;

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith7.png)

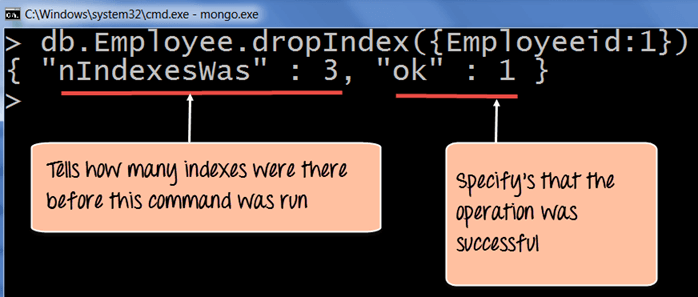
db.Employee.dropIndex(Employeeid:1)

**Code Explanation:**

1. The dropIndex method takes the required Field values which needs to be removed from the Index.

If the command is executed successfully, the following Output will be shown:

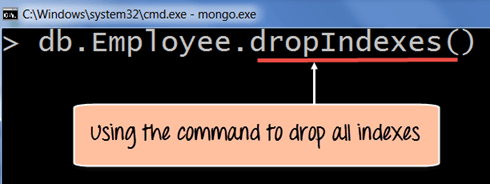
**Output:**

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith8.png)

1. The nIndexesWas: 3 indicates the number of Field values which were there in the indexes before the command was run. Remember that each collection has the \_id field which also counts as a Field value to the index.
2. The ok: 1 output specifies that the operation was successful, and the "Employeeid" field is removed from the index.

To remove all of the indexes at once in the collection, one can use the dropIndexes command.

The following example shows how this can be done.

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith9.png)

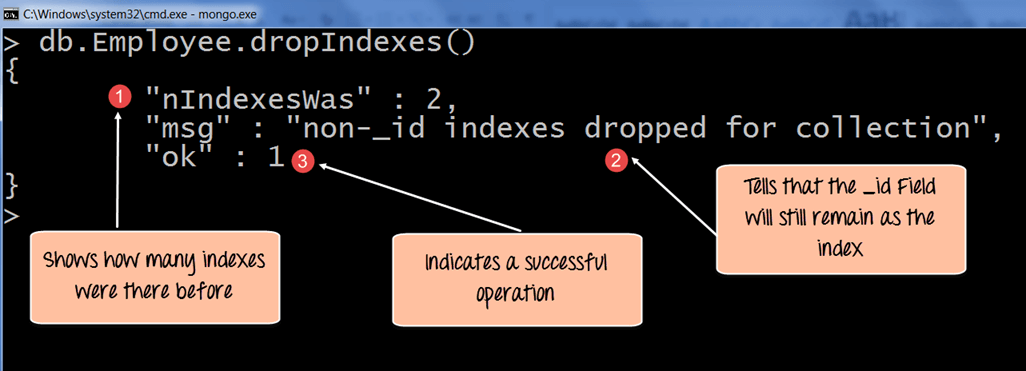
db.Employee.dropIndex()

**Code Explanation:**

1. The dropIndexes method will drop all of the indexes except for the \_id index.

If the command is executed successfully, the following Output will be shown:

**Output:**

[](https://www.guru99.com/images/MongoDB/112115_0455_Workingwith10.png)

1. The nIndexesWas: 2 indicates the number of Field values which were there in the indexes before the command was run.
2. Remember again that each collection has the \_id field which also counts as a Field value to the index, and that will not be removed by MongoDB and that is what this message indicates.
3. The ok: 1 output specifies that the operation was successful.

**Aggregation**

In aggregation operation, MongoDB processes the data records and returns a single computed result. It actually groups multiple documents and then performs aggregation operation on it and after that returns a single result to the end user.

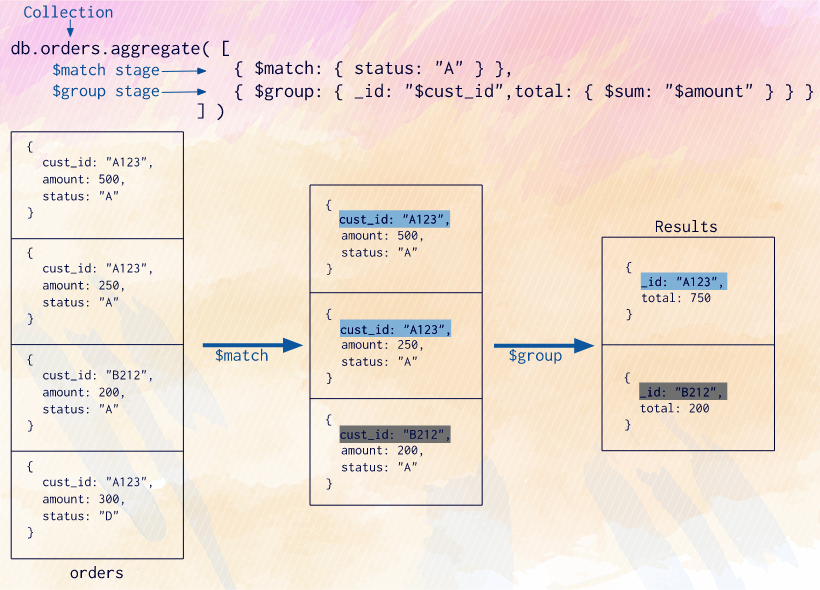
MongoDB can perform aggregation in 3 ways and they are as follows:

1. Aggregation Pipeline
2. Map-Reduce Function
3. Single Purpose Aggregation Method

### i. MongoDB Aggregation Pipeline

Aggregation process in[**MongoDB is modeled on the concept of data**](https://data-flair.training/blogs/mongodb-data-modeling/) processing pipelines. Multiple documents enter the pipeline and then these documents are being transformed into aggregated results. The operations being performed during pipeline include filter and document transformation in which they operate like queries and modify the form of output document respectively.

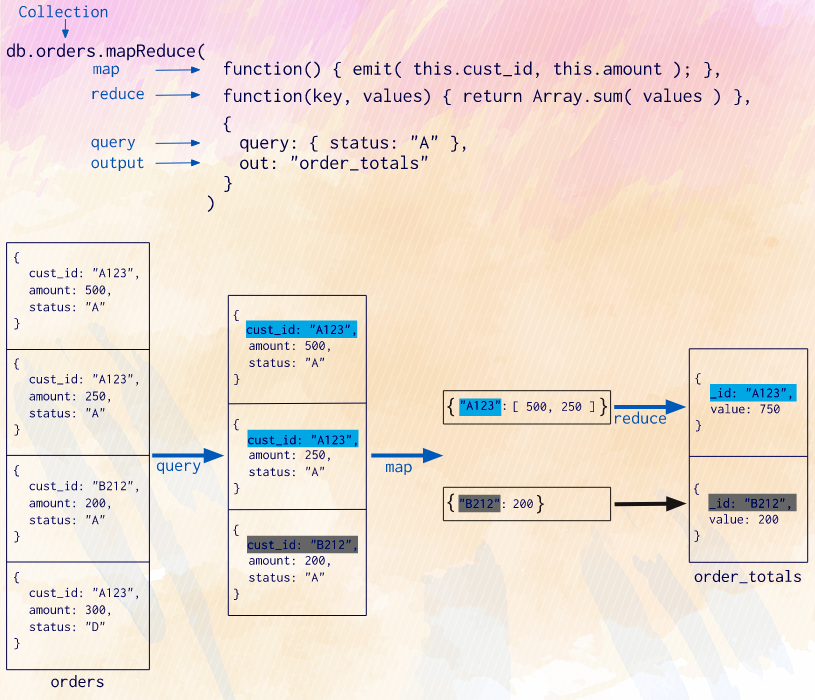
The pipeline method provides efficient data aggregation using other operations present in MongoDB. It can also operate on a sharded collection. It can also use **[MongoDB indexes](https://data-flair.training/blogs/mongodb-index/)** to improve its efficiency at some stages. With all this aggregation pipeline has an internal optimization phase due to which it can make this process optimal for the processor.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2019/02/MongoDB-Aggregation-pipeline-example.png)

*MongoDB Aggregation Pipeline Example*

### ii. Map-Reduce

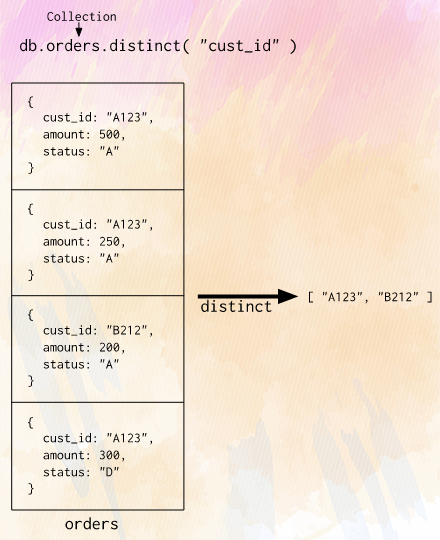
As the name suggests it has two operations in itself, they are a map in which each document is being processed along with emitting one or more objects for each document and reduce phase in which output of map operation are being combined together.[**MongoDB Map-reduce**](https://data-flair.training/blogs/mongodb-mapreduce/) can optionally have a finalize stage in which it can do some final modifications in the output of the document. Map-Reduce uses JavaScript to perform its operations including the finalize operation. Even though javascript provides greater flexibility than aggregation pipeline, but still it is less efficient and more complex operation as compared to the aggregation pipeline in MongoDB.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2019/02/MongoDB-Mapreduce-example.png)

*MongoDB Mapreduce Example*

### iii. Single Purpose Aggregation Operations

These operations aggregate all the documents from a single [**collection in MongoDB**](https://data-flair.training/blogs/mongodb-create-collection/). Even though they provide simple access to common aggregation operations they lack the flexibility and capability of map-reduce and aggregation pipeline.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2019/02/Single-Purpose-Aggregation-Operation-Example.png)

*Single Purpose Aggregation Operation Example*

# MongoDB - Create a Relationship

To create a relationship in MongoDB, either embed a BSON document within another, or reference it from another.

MongoDB databases work differently to relational databases. This is also true of relationships.

In MongoDB, you can create a relationship using one of the following two methods:

* Embedded documents.
* Referenced documents.

The method you use will depend on the data, and how you intend to query that data.

## Embedded Relationships

With MongoDB, you can embed documents within documents. Therefore, a single document can contain its own relationships.

In fact, we already created a relationship using this method back when we first [created a document](https://www.quackit.com/mongodb/tutorial/mongodb_create_a_document.cfm).

### One-to-One Relationship

A *one-to-one relationship* is where the parent document has one child, and the child has one parent.

For example, a business rule might say that an artist can only have one address and that the address can only belong to one artist.

The following code creates a one-to-one relationship, embedded within the document.

db.artists.insert(

  {

       \_id : 2,

       artistname : "Prince",

       address :   {

                       street : "Audubon Road",

                       city : "Chanhassen",

                       state : "Minnesota",

                       country : "United States"

                  }

  }

)

Result:

WriteResult({ "nInserted" : 1 })

### One-to-Many Relationship

A one-to-many relationship is where the parent document can have many child documents, but the child documents can only have one parent document.

So, another business rule might say that one artist can have many albums, but an album can only belong to one artist.

Running the following code will create a one-to-many relationship:

db.artists.insert(

  {

       \_id : 3,

       artistname : "Moby",

       albums : [

                  {

                       album : "Play",

                       year : 1999,

                       genre : "Electronica"

                  },

                  {

                       album : "Long Ambients 1: Calm. Sleep.",

                       year : 2016,

                       genre : "Ambient"

                  }

              ]

  }

)

Result:

WriteResult({ "nInserted" : 1 })

## Document Referenced Relationships

You can use a document reference to create a relationship. Rather than embedding the child document into the parent document (like we did above), you separate the child document out into to its own stand alone document.

So we could do this:

### Parent Document

db.artists.insert(

  {

       \_id : 4,

       artistname : "Rush"

  }

)

### Child Documents

We'll insert 3 child documents — one for each band member:

db.musicians.insert(

  {

       \_id : 9,

       name : "Geddy Lee",

       instrument : [ "Bass", "Vocals", "Keyboards" ],

       artist\_id : 4

  }

)

db.musicians.insert(

  {

       \_id : 10,

       name : "Alex Lifeson",

       instrument : [ "Guitar", "Backing Vocals" ],

       artist\_id : 4

  }

)

db.musicians.insert(

  {

       \_id : 11,

       name : "Neil Peart",

       instrument : "Drums",

       artist\_id : 4

  }

)

### Querying the Relationship

After inserting the above two documents, you can use $lookup to perform a left outer join on the two collections.

This, in conjunction with the aggregate() method, and $match to specify the specific artist you're interested in, will return parent and child documents in one.

db.artists.aggregate([

  {

     $lookup:

      {

         from: "musicians",

         localField: "\_id",

         foreignField: "artist\_id",

         as: "band\_members"

      }

  },

  { $match : { artistname : "Rush" } }

]).pretty()

Result:

{

"\_id" : 4,

"artistname" : "Rush",

"band\_members" : [

{

"\_id" : 9,

"name" : "Geddy Lee",

"instrument" : [

"Bass",

"Vocals",

"Keyboards"

],

"artist\_id" : 4

},

{

"\_id" : 10,

"name" : "Alex Lifeson",

"instrument" : [

"Guitar",

"Backing Vocals"

],

"artist\_id" : 4

},

{

"\_id" : 11,

"name" : "Neil Peart",

"instrument" : "Drums",

"artist\_id" : 4

}

]

}

You can see that the first two fields are from the artists collection, and the rest of it is from the musicians collection.

So if you only query the artists collection by itself:

db.artists.find( { artistname : "Rush" } )

You'd only get this:

{ "\_id" : 4, "artistname" : "Rush" }

No related data is returned.

## When to use Embedded Documents vs Referenced Documents

Both methods of creating relationships have their pros and cons. There are times you might use embedded documents, and other times you'll use referenced documents.

### When to use Embedded Relationships

One of the main benefits of using the embedded relationship method is performance. When the relationship is embedded within the document, queries will run faster than if they were spread out over multiple documents. MongoDB only needs to return the one document, rather than joining multiple documents in order to retrieve the relationships. This can provide a major performance boost — especially when working with lots of data.

Embedded relationships also make queries easier to write. Rather than writing complex queries that join many documents via their unique identifier, you can return all related data within a single query.

Another consideration to keep in mind is that, MongoDB can only ensure atomicity at a document level. Document updates to a single document are always atomic, but not for multiple documents.

When multiple users are accessing the data, there's always a chance that two or more users will try to update the same document with different data. In this case, MongoDB will ensure that no conflict occurs and only one set of data is updated at a time. MongoDB cannot ensure this across multiple documents.

So in general, embedded relationships can be used in most cases, as long as the document remains within the size limit (16 megabytes at the time of writing), and/or its nesting limit (100 levels deep at the time of writing).

However, embedded relationships aren't appropriate for all occasions. There may be situations where it makes more sense to create a document referenced relationship.

### When to use Referenced Relationships

For data that needs to be repeated across many documents, it can be helpful to have them in their own separate document. This can reduce errors and help in keeping the data consistent (while bearing in mind that multiple-document updates are not atomic).

Using the above example, one musician could be a member (or ex-member) of many bands. Some might also produce albums for other artists, teach students, run clinics, etc. Also, a lot of data could be stored against each musician. So having a separate document for each musician makes sense in this case.

Also, if you think your embedded documents might exceed the file size limit imposed by MongoDB, then you'll need to store some data in separate documents.

# MongoDB – limit( ) and skip( ) method

# The limit() method in MongoDB

This method limits the number of documents returned in response to a particular query.  
Syntax:

db.collection\_name.find().limit(number\_of\_documents)

Lets take an example to understand how to use this method. Lets say, I have a collection studentdata which has following documents:

> db.studentdata.find().pretty()

{

"\_id" : ObjectId("59bf63380be1d7770c3982af"),

"student\_name" : "Steve",

"student\_id" : 2002,

"student\_age" : 22

}

{

"\_id" : ObjectId("59bf63500be1d7770c3982b0"),

"student\_name" : "Carol",

"student\_id" : 2003,

"student\_age" : 22

}

{

"\_id" : ObjectId("59bf63650be1d7770c3982b1"),

"student\_name" : "Tim",

"student\_id" : 2004,

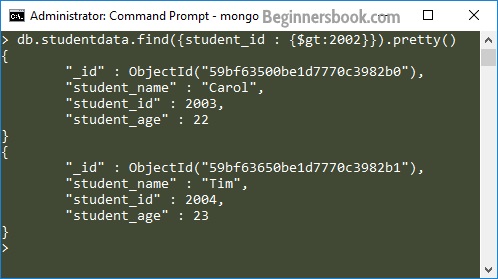
"student\_age" : 23

}

Lets say I want to find out the list of all the students, having the id > 2002. I would write a query like this using a criteria:

To learn how to specify a criteria while querying documents, read this: [MongoDB Query Document](https://beginnersbook.com/2017/09/mongodb-query-document-using-find-method/)

db.studentdata.find({student\_id : {$gt:2002}}).pretty()



**Using limit() method to limit the documents in the result:**  
Lets say I do not want all the documents matching the criteria. I want only selected number of documents then I can use limit() method to limit the number of documents. For example, if I want only one document in output then I would do this:

> db.studentdata.find({student\_id : {$gt:2002}}).limit(1).pretty()

{

"\_id" : ObjectId("59bf63500be1d7770c3982b0"),

"student\_name" : "Carol",

"student\_id" : 2003,

"student\_age" : 22

}

## MongoDB Skip() Method

The skip() method is used for skipping the given number of documents in the Query result.

To understand the use of skip() method, lets take the same example that we have seen above. In the above example we can see that by using limit(1) we managed to get only one document, which is the first document that matched the given criteria. What if you do not want the first document matching your criteria. For example we have two documents that have student\_id value greater than 2002 but when we limited the result to 1 by using limit(1), we got the first document, in order to get the second document matching this criteria we can use skip(1) here which will skip the first document.

**Without using skip():**

> db.studentdata.find({student\_id : {$gt:2002}}).limit(1).pretty()

{

"\_id" : ObjectId("59bf63500be1d7770c3982b0"),

"student\_name" : "Carol",

"student\_id" : 2003,

"student\_age" : 22

}

**Using skip:**

> db.studentdata.find({student\_id : {$gt:2002}}).limit(1).skip(1).pretty()

{

"\_id" : ObjectId("59bf63650be1d7770c3982b1"),

"student\_name" : "Tim",

"student\_id" : 2004,

"student\_age" : 23

}