JOIN is one of the key distinct features between SQL and NoSQL databases. In SQL databases, we can perform a JOIN between two tables within the same or different databases. However, this is not the case for MongoDB as it allows JOIN operations between two collections in the same database.

The way data is presented in MongoDB makes it almost impossible to relate it from one collection to another except when using basic script query functions. MongoDB either de-normalizes data by storing related items in a separate document or it relates data in some other separate document.

One could relate this data by using manual references such as the \_id field of one document that is saved in another document as a reference. Nevertheless, one needs to make multiple queries in order to fetch for some required data, making the process a bit tedious.

We therefore resolve to using the JOIN concept which facilitates the relation of the data. JOIN operation in MongoDB is achieved through the use of $lookup operator, which was introduced in version 3.2.

# $lookup operator

The main idea behind the JOIN concept is to get correlation between data in one collection to another. The basic syntax of $lookup operator is:

|  |
| --- |
| {  $lookup:  {  from: <collection to join>,  localField: <field from the input documents>,  foreignField: <field from the documents of the "from" collection>,  as: <output array field>  }  } |

Regarding the SQL knowledge, we always know that the result of a JOIN operation is a separate row linking all fields from the local and foreign table. For MongoDB, this is a different case in that the result documents are added as an array of local collection document. For example, let’s have two collections; ‘students’ and ‘units’

**students**

|  |
| --- |
| {"\_id" : 1,"name" : "James Washington","age" : 15.0,"grade" : "A","score" : 10.5}  {"\_id" : 2,"name" : "Clinton Ariango","age" : 14.0,"grade" : "B","score" : 7.5}  {"\_id" : 3,"name" : "Mary Muthoni","age" : 16.0,"grade" : "A","score" : 11.5} |

**Units**

|  |
| --- |
| {"\_id" : 1,"Maths" : "A","English" : "A","Science" : "A","History" : "B"}  {"\_id" : 2,"Maths" : "B","English" : "B","Science" : "A","History" : "B"}  {"\_id" : 3,"Maths" : "A","English" : "A","Science" : "A","History" : "A"} |

We can retrieve the students units with respective grades using the $lookup operator with the JOIN approach .i.e

|  |
| --- |
| db.getCollection('students').aggregate([{  $lookup:  {  from: "units",  localField: "\_id",  foreignField : "\_id",  as: "studentUnits"  }  }]) |

Which will give us the results below:

|  |
| --- |
| {"\_id" : 1,"name" : "James Washington","age" : 15,"grade" : "A","score" : 10.5,  "studentUnits" : [{"\_id" : 1,"Maths" : "A","English" : "A","Science" : "A","History" : "B"}]}  {"\_id" : 2,"name" : "Clinton Ariango","age" : 14,"grade" : "B","score" : 7.5,  "studentUnits" : [{"\_id" : 2,"Maths" : "B","English" : "B","Science" : "A","History" : "B"}]}  {"\_id" : 3,"name" : "Mary Muthoni","age" : 16,"grade" : "A","score" : 11.5,  "studentUnits" : [{"\_id" : 3,"Maths" : "A","English" : "A","Science" : "A","History" : "A"}]} |

Like mentioned before, if we do a JOIN using the SQL concept, we will be returned with separate documents in the Studio3T platform .i.e

|  |
| --- |
| SELECT \*  FROM students  INNER JOIN units  ON students.\_id = units.\_id |

Is an equivalent of

|  |
| --- |
| db.getCollection("students").aggregate(  [  {  "$project" : {  "\_id" : NumberInt(0),  "students" : "$$ROOT"  }  },  {  "$lookup" : {  "localField" : "students.\_id",  "from" : "units",  "foreignField" : "\_id",  "as" : "units"  }  },  {  "$unwind" : {  "path" : "$units",  "preserveNullAndEmptyArrays" : false  }  }  ]  ); |

The above SQL query will return the results below:

|  |
| --- |
| { "students" : {"\_id" : NumberInt(1),"name" : "James Washington","age" : 15.0,"grade" : "A","score" : 10.5},  "units" : {"\_id" : NumberInt(1),"Maths" : "A","English" : "A","Science" : "A","History" : "B"}}  { "students" : {"\_id" : NumberInt(2), "name" : "Clinton Ariango","age" : 14.0,"grade" : "B","score" : 7.5 },  "units" : {"\_id" : NumberInt(2),"Maths" : "B","English" : "B","Science" : "A","History" : "B"}}  { "students" : {"\_id" : NumberInt(3),"name" : "Mary Muthoni","age" : 16.0,"grade" : "A","score" : 11.5},  "units" : {"\_id" : NumberInt(3),"Maths" : "A","English" : "A","Science" : "A","History" : "A"}} |

The performance duration will obviously be dependent on the structure of your query. For instance, if you have many documents in one collection over the other, you should do the aggregation from the collection with lesser documents and then lookup in the one with more documents. This way, a lookup for the chosen field from the lesser documents collection is quite optimal and takes lesser time than doing multiple lookups for a chosen field in the collection with more documents. It is therefore advisable to put the smaller collection first.

For a relational database, the order of the databases does not matter since most SQL interpreters have optimizers, which have access to extra information for deciding which one should be first.

In the case of MongoDB, we will need to use an index to facilitate the JOIN operation. We all know that all MongoDB documents have an \_id key which for a relational DBM can be considered as the primary key. An index provides a better chance of reducing the amount of data that needs to be accessed besides supporting the operation when used in the $lookup foreign key.

In the aggregation pipeline, to use an index, we must ensure the $match is done first stage in order to filter out documents that do not match the criteria. For example if we want to retrieve the result for the student with \_id field value equal to 1:

|  |
| --- |
| select \*  from students  INNER JOIN units  ON students.\_id = units.\_id  WHERE students.\_id = 1; |

The equivalent MongoDB code you will get in this case is:

|  |
| --- |
| db.getCollection("students").aggregate(  [{"$project" : { "\_id" : NumberInt(0), "students" : "$$ROOT" }},  { "$lookup" : {"localField" : "students.\_id", "from" : "units", "foreignField" : "\_id", "as" : "units"} },  { "$unwind" : { "path" : "$units","preserveNullAndEmptyArrays" : false } },  { "$match" : {"students.\_id" : NumberLong(1) }}  ]); |

The returned result for the query above will be:

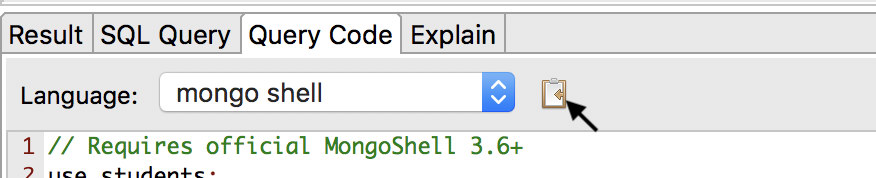
|  |
| --- |
| {"\_id" : 1,"name" : "James Washington","age" : 15,"grade" : "A","score" : 10.5,  "studentUnits" : [{"\_id" : 1,"Maths" : "A","English" : "A","Science" : "A","History" : "B"}]} |

When we don’t use the $match stage or rather not at the first stage, if we check with the explain function, we will get the COLLSCAN stage also included. Doing a COLLSCAN for a large set of documents will generally take a lot of time. We thereby resolve to using an index field which in the explain function involves the IXSCAN stage only. The latter has an advantage since we are checking on an index in the documents and not scanning through all the documents; it will not take long to return the results. You may have a different data structure like:

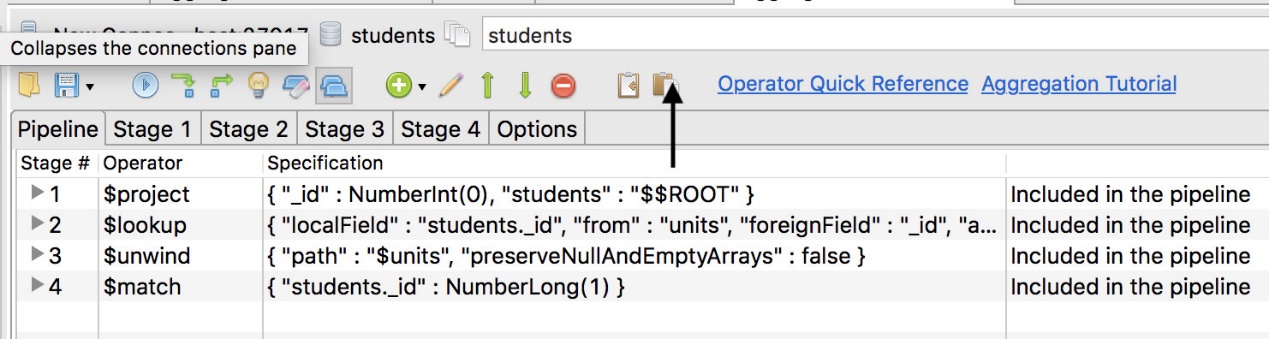
|  |
| --- |
| { "\_id" : NumberInt(1),  "grades" : {"Maths" : "A", "English" : "A", "Science" : "A", "History" : "B"  }  } |

We may want to return the grades as different entities in an array rather than a whole embedded grades field.

After writing the SQL query above, we need to modify the resulting MongoDB code. To do so, click on the copy icon on the right as below to copy the aggregation code:



Next go to the aggregation tab and on the presented pane, there is a paste icon, click it to paste the code.



Click the $match row and then the green up-arrow to move the stage to top as the first stage. However, you will need to create an index in your collection first like:

|  |
| --- |
| db.students.createIndex(  { \_id: 1 },  { name: studentId }  ) |

You will get the code sample below:

|  |
| --- |
| db.getCollection("students").aggregate(  [{ "$match" : {"\_id" : 1.0}},  { "$project" : {"\_id" : NumberInt(0),"students" : "$$ROOT"}},  { "$lookup" : {"localField" : "students.\_id","from" : "units","foreignField" : "\_id","as" : "units"}},  { "$unwind" : {"path" : "$units", "preserveNullAndEmptyArrays" : false}} |

With this code we will get the result below:

|  |
| --- |
| { "students" : {"\_id" : NumberInt(1), "name" : "James Washington","age" : 15.0,"grade" : "A", "score" : 10.5},  "units" : {"\_id" : NumberInt(1), "grades" : {"Maths" : "A", "English" : "A", "Science" : "A", "History" : "B"}}} |

But all we need is to have the grades as a separate document entity in the returned document and not as the above example. We will hence add the $addfields stage hence the code as below.

|  |
| --- |
| db.getCollection("students").aggregate(  [{ "$match" : {"\_id" : 1.0}},  { "$project" : {"\_id" : NumberInt(0),"students" : "$$ROOT"}},  { "$lookup" : {"localField" : "students.\_id","from" : "units","foreignField" : "\_id","as" : "units"}},  { "$unwind" : {"path" : "$units", "preserveNullAndEmptyArrays" : false}},  { "$addFields" : {"units" : "$units.grades"} }] |

The resulting documents will then be:

|  |
| --- |
| {  "students" : {"\_id" : NumberInt(1), "name" : "James Washington", "grade" : "A","score" : 10.5},  "units" : {"Maths" : "A", "English" : "A", "Science" : "A", "History" : "B"}  } |

The returned data is quite neat, as we have eliminated embedded documents from the units’ collection as a separate field.

In our next tutorial, we are going to look into queries with several joins.