One of the biggest [differences between SQL and NoSQL databases](https://www.sitepoint.com/sql-vs-nosql-differences/) is JOIN. In relational databases, the SQL JOIN clause allows you to combine rows from two or more tables using a common field between them. For example, if you have tables of books and publishers, you can write SQL commands such as:

SELECT book.title, publisher.name

FROM book

LEFT JOIN book.publisher\_id ON publisher.id;

In other words, the book table has a publisher\_id field which references the id field in the publisher table.

This is practical, since a single publisher could offer thousands of books. If we ever need to update a publisher’s details, we can change a single record. Data redundancy is minimized, since we don’t need to repeat the publisher information for every book. The technique is known as **normalization**.

SQL databases offer a range of normalization and constraint features to ensure relationships are maintained.

**NoSQL == No JOIN?**

*Not always …*

Document-oriented databases such as [MongoDB](https://www.mongodb.com/cloud/atlas) are designed to store denormalized data. Ideally, there should be no relationship between collections. If the same data is required in two or more documents, it must be repeated.

This can be frustrating, since there are few situations where you *never* need relational data. Fortunately, MongoDB 3.2 introduces a new [$lookup operator](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/) which can perform a LEFT-OUTER-JOIN-like operation on two or more collections. But there’s a catch …

**MongoDB Aggregation**

$lookup is only permitted in [aggregation operations](https://docs.mongodb.com/manual/aggregation/). Think of these as a pipeline of operators which query, filter and group a result. The output of one operator is used as the input for the next.

Aggregation is more difficult to understand than simpler [find queries](https://docs.mongodb.com/manual/tutorial/query-documents/) and will generally run slower. However, they are powerful and an invaluable option for complex search operations.

Aggregation is best explained with an example. Presume we’re creating a social media platform with a user collection. It stores every user’s details in separate documents. For example:

{

"\_id": ObjectID("45b83bda421238c76f5c1969"),

"name": "User One",

"email: "userone@email.com",

"country": "UK",

"dob": ISODate("1999-09-13T00:00:00.000Z")

}

We can add as many fields as necessary, but all MongoDB documents require an \_idfield which has a unique value. The \_id is similar to an SQL primary key, and will be inserted automatically if necessary.

Our social network now requires a post collection, which stores numerous insightful updates from users. The documents store the text, date, a rating and a reference to the user who wrote it in a user\_id field:

{

"\_id": ObjectID("17c9812acff9ac0bba018cc1"),

"user\_id": ObjectID("45b83bda421238c76f5c1969"),

"date: ISODate("2016-09-05T03:05:00.123Z"),

"text": "My life story so far",

"rating": "important"

}

We now want to show the last twenty posts with an “important” rating from all users in reverse chronological order. Each returned document should contain the text, the time of the post and the associated user’s name and country.

The MongoDB [aggregate query](https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/) is passed an array of [pipeline operators](https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/) which define each operation in order. First, we need to extract all documents from the postcollection which have the correct rating using the [$match filter](https://docs.mongodb.com/manual/reference/operator/aggregation/match/):

{ "$match": { "rating": "important" } }

We must now sort the matched items into reverse date order using the [$sortoperator](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/):

{ "$sort": { "date": -1 } }

Since we only require twenty posts, we can apply a [$limit stage](https://docs.mongodb.com/manual/reference/operator/aggregation/limit/) so MongoDB only needs to process data we want:

{ "$limit": 20 }

We can now join data from the user collection using the new [$lookup operator](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#pipe._S_lookup). It requires an object with four parameters:

* localField: the lookup field in the input document
* from: the collection to join
* foreignField: the field to lookup in the from collection
* as: the name of the output field.

Our operator is therefore:

{ "$lookup": {

"localField": "user\_id",

"from": "user",

"foreignField": "\_id",

"as": "userinfo"

} }

This will create a new field in our output named userinfo. It contains an array where each value is the matching the user document:

"userinfo": [

{ "name": "User One", ... }

]

We have a one-to-one relationship between the post.user\_id and user.\_id, since a post can only have one author. Therefore, our userinfo array will only ever contain one item. We can use the [$unwind operator](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/) to deconstruct it into a sub-document:

{ "$unwind": "$userinfo" }

The output will now be converted to a more practical format which can have further operators applied:

"userinfo": {

"name": "User One",

"email: "userone@email.com",

…

}

Finally, we can return the text, the time of the post, the user’s name and country using a [$project stage](https://docs.mongodb.com/manual/reference/operator/aggregation/project/) in the pipeline:

{ "$project": {

"text": 1,

"date": 1,

"userinfo.name": 1,

"userinfo.country": 1

} }

**Putting It All Together**

Our final aggregate query matches posts, sorts into order, limits to the latest twenty items, joins user data, flattens the user array and returns necessary fields only. The full command:

db.post.aggregate([

{ "$match": { "rating": "important" } },

{ "$sort": { "date": -1 } },

{ "$limit": 20 },

{ "$lookup": {

"localField": "user\_id",

"from": "user",

"foreignField": "\_id",

"as": "userinfo"

} },

{ "$unwind": "$userinfo" },

{ "$project": {

"text": 1,

"date": 1,

"userinfo.name": 1,

"userinfo.country": 1

} }

]);

The result is a collection of up to twenty documents. For example:

[

{

"text": "The latest post",

"date: ISODate("2016-09-27T00:00:00.000Z"),

"userinfo": {

"name": "User One",

"country": "UK"

}

},

{

"text": "Another post",

"date: ISODate("2016-09-26T00:00:00.000Z"),

"userinfo": {

"name": "User One",

"country": "UK"

}

}

...

]

**Great! I Can Finally Switch to NoSQL!**

MongoDB $lookup is useful and powerful, but even this basic example requires a complex aggregate query. It’s not a substitute for the more powerful JOIN clause offered in SQL. Neither does MongoDB offer constraints; if a user document is deleted, orphan post documents would remain.

Ideally, the $lookup operator should be required infrequently. If you need it a lot, you’re possibly using the wrong data store …

If you have relational data, use a relational (SQL) database!

That said, $lookup is a welcome addition to MongoDB 3.2. It can overcome some of the more frustrating issues when using small amounts of relational data in a NoSQL database.