Topics

Neo4j basics

HQ_6_1

How can you find the total number of documents in a collection named "enrolls" in MongoDB? Select all the methods that work. Assume the collection is not sharded, and the MongoDB version is 3.4.

✔ `db.enrolls.count()`

- Note: count() method does not work robustly if the collection is sharded (distributed across multiple servers)

✔ `db.enrolls.find().count()`

✔ `db.enrolls.aggregate([{$count: "count"}])`

- The $count stage was introduced in version 3.4.

✔ `db.enrolls.aggregate([{$group: {_id: null, count: {$sum: 1}}}])`

✖ `db.enrolls.aggregate([{$group: {_id: null, count: {$count: 1}}}])`

- *countisanaggregationstage*, *itcannotbeusedasagroupaccumulatorlikesum*

HQ_6_2

In the MongoDB primer collection restaurants (see figure below for a sample), is it possible to find those restaurants that have received at least one score (i.e. deficiency points, lower is better) of greater than 100, using the `find` method? If so, how?

✔ Yes. `db.restaurants.find({"grades.score": {$gt: 100}})`

- The find() method is able to look through an array, in this case, "grades.score", and a document is matched if at least one element in the array satisfies the condition.

✖ No. One must unwind the `grades` array by using the `aggregate` method.

- Aggregate() is more general and powerful than find(), but to deal with arrays in documents, we have to unwind first, and here we show that sometimes find() provides a good shortcut to look through arrays without explicit unwinding.

HQ_6_3

In the MongoDB primer collection restaurants, how do you find those "inconsistent" restaurants that have received at least one grade "A", as well as at least one grade of "C"?

✔

```
db.restaurants.find(
    {"grades.grade": {$all: ["A", "C"]}}
)
```

- The *alloperatorwillmatchanarray*(*inthiscase*, " *grades*. *grade* all, namely ["A", "C"]. ")*if thearraycontainsalltheelementsintheargumentof*

✔

```
db.restaurants.find(
    {$and: [
        {"grades.grade": "A"},
        {"grades.grade": "C"}
        ]
    }
)
```

- The *alloperatorisequivalenttoth*and operator (behavior since version 2.6).

✖

```
db.restaurants.find(
    {"grades.grade": ["A", "C"]}
)
```

- This query matches documents whose "grades.grade" array is exactly ["A", "C"], that is, the restaurant has been graded exactly twice, and received "A" the first time, and "C" the second time.

HQ_6_4

In the MongoDB primer collection restaurants, how can you sort the restaurants by the number of grade A ratings they have received, in descending order?

✔

```
db.restaurants.aggregate([
        {$unwind: "$grades"},
        {$match: {"grades.grade": "A"}},
        {$group: {_id: "$_id", count_gradeA: {$sum: 1}}},
        {$sort: {count_gradeA: -1}}
])
```

- Before matching for grade A ratings, we again must unwind the grades array. After

the
matchstage, sum,
wehavethedocumentseachrepresentingoneinstanceofsomerestaurantreceivingagradeA
. Sonowifwegroupbyrestaurant IDs, andgetthesizeofeachgroupvia
we have essentially computed the number of times each restaurant has received grade A. Finally we sort the results in descending order.

✖

```
db.restaurants.find({"grades.grade": "A"}).count().sort("grades.grade": -1)
```

- Here count() only counts the total number of restaurants that have received at least one grade A. And thus calling sort() on a single count value is illegal.

HQ_6_5

In the MongoDB primer collection restaurants, how can you find all the "straight-A" restaurants, namely those that have received an "A" every time they are being graded. Sort the results by the total number of "A"s received, in descending order.

✔

```
db.restaurants.aggregate([
    {$unwind: "$grades"},
    {$group: {
        _id: "$_id",
        count_grades: {$sum: 1},
        count_A: {$sum: {$cond: [{$eq: ["$grades.grade", "A"]}, 1, 0]}}
        }
    },
    {$project: {
        is_straight_A: {$eq: ["$count_grades", "$count_A"]},
        count_A: 1,
        }
    },
    {$match: {is_straight_A: true}},
    {$sort: {"count_A": -1}},
])
```

- After unwinding, we need to obtain both the total number of grades a restaurant has received (count_grades), and the total number of "A"s received (count_A). Also note that we cannot filter out all the non-A documents prior to grouping as we did in Question 3. Next we need to test if count_grades == count_A, which is the definition of "straight-A". This can be done in a $project stage. Then we can filter out all those that are not straight-A restaurants. Finally we sort the results by count_A in descending order as required.

✖

```
db.restaurants.aggregate([
    {$unwind: "$grades"},
    {$match: {"grades.grade": "A"}},
    {$group: {
        _id: "$_id",
        count_A: {$sum: 1},
        }
    },
    {$sort: {"count_A": -1}},
])
```

- This only counts the number of "A"s a restaurant has received, but without the knowledge of the total number of grades,we cannot determine whether a restaurant is straight-A.

HQ_6_6

In the MongoDB primer collection restaurants, how can you find the average score for each of the cuisines, and sort them by the average score in the ascending order? Here we define the average score of a cuisine such that every restaurant belonging to a certain cuisine is weighted equally, regardless of how many times it has been graded. That is, every restaurant contributes a single average score of its own to the cuisine's average score.

✔

```
db.restaurants.aggregate([
    {$unwind: "$grades"},
    {$group: {
        _id: "$_id",
        avg_restaurant_score: {$avg: "$grades.score"},
        cuisine: {$first: "$cuisine"}
        }
    },
    {$group: {
        _id: "$cuisine",
        avg_cuisine_score: {$avg: "$avg_restaurant_score"},
        }
    },
    {$sort: {avg_cuisine_score: 1}}
])
```

- Since the average defined in the question weights each restaurant of certain cuisine equally, we need to first find the average score for each restaurant by using the first group stage. Note that since the second group stage takes the input only from the first group result, we have to retain the "cuisine" attribute, which can be obtained using the $first$(or last) operator (this works because restaurant ID functionally determines its cuisine). Now in the second group stage, we can compute the average score over all the restaurants belonging to a specific cuisine, effectively, we are taking the average of averages.

✖

```
db.restaurants.aggregate([
    {$unwind: "$grades"},
    {$group: {
        _id: "$cuisine",
        avg_score: {$avg: "$grades.score"}
        }
    },
    {$sort: {avg_score: 1}}
])
```

- This computes the average score over all the grades received for a given cuisine, that is, each restaurant of that cuisine is not weighted equally (those that have been graded more frequently will bear a larger weight), which violates the requirement.

HQ_6_7

We just imported two collections, "students", with fields "_id", "sid", "major", and "enrolls", with fields "_id", "student_id", "course_id", "term", into the database "academic_world" in MongoDB (version 3.4). **1)** How can you join "enrolls" into "students", which is the main collection that you want to work with? **2)** Now say we have only 3 students in the "students" collection, their sID being: "alice1", "bob2", "cate3", and we know that "alice1" has taken 3 courses, "bob2" has taken 2 courses, and "cate3", as a new freshman, hasn't taken any courses yet. How many documents in total would you get as the output of the join?

✔

```
db.students.aggregate([
    {$lookup: {
        from: "enrolls",
        localField: "sid",
        foreignField: "student_id",
        as: "enroll_records"
        }
    }
])
```

Output: 3 documents in total

- "localField" is the field from the collection on which aggregate() is called, while "foreignField" is the field from the "from" collection. The "lookup" stage does not generate a new, "combined" document for every matching pair of documents as a SQL join would do, instead, it brings in all the matching documents as an array, whose name is specified in the "as" field. If there are no matching documents from the other collection, the array is simply empty, but the document itself is not eliminated from the result (hence "lookup" is regarded as an outer join). Therefore, "lookup" does not change the total number of documents in the output.

✖

```
db.students.aggregate([
    {$lookup: {
        from: "enrolls",
        localField: "sid",
        foreignField: "student_id",
        As: "enroll_records"
        }
    }
])
```

Output: 5 documents in total

✖

```
db.students.aggregate([
    {$lookup: {
        from: "enrolls",
        localField: "student_id",
        foreignField: "sid",
        as: "enroll_records"
        }
    }
])
```

Output: 3 documents in total

✖

```
db.students.aggregate([
    {$lookup: {
        from: "enrolls",
        localField: "student_id",
        foreignField: "sid",
        as: "enroll_records"
        }
    }
])
```

Output: 5 documents in total

HQ_6_8

In Cypher, some of the SQL clauses are apparently "missing", such as SELECT, FROM, GROUP BY, and HAVING. However, Cypher is expressive enough to offer the functionalities of all these clauses. From what you have learned, what are the Cypher clauses that provide the corresponding functionalities of SELECT, FROM, GROUP BY, and HAVING, respectively?

✔ RETURN, MATCH, RETURN or WITH, WITH...WHERE

- RETURN not only performs projection (like SELECT), but also specifies the grouping key as all the returned expressions that are not aggregate functions. WITH is similar to RETURN in that it does grouping as well as aggregation, but instead of

terminating the query, it passes the result to the next stage in the pipeline. FROM is handled as part of the MATCH clause (actually MATCH is more like FROM...WHERE). Also note that in Cypher, WHERE cannot be used as a standalone clause, instead, it must be used as part of MATCH, OPTIONAL MATCH, or WITH. Since in SQL, the HAVING clause operates on the grouping result, like the pipeline stage $match in MongoDB, the corresponding Cypher clause is WITH...WHERE, in which WITH passes down the expressions (including aggregate functions), and WHERE does the filtering..

✖ MATCH, WITH, RETURN or WITH, MATCH

✖ RETURN, MATCH, MATCH, WITH

HQ_6_9

The figure below shows the graph model for "Friday Night". Using Cypher, how can you find the beers (by name) with the highest alcohol among all the beers sold at Sober Bar?

✔

```
MATCH (beer:Beer)<-[:Sells]-(:Bar {name: "Sober Bar"})
WITH MAX(beer.alcohol) as max_alcohol
MATCH (beer:Beer)<-[:Sells]-(:Bar {name: "Sober Bar"})
WHERE beer.alcohol = max_alcohol
RETURN beer.name
```

- First we find the max_alcohol among all the beers sold at Sober Bar. Then among the same set of Sober Bar beers, we find the beer whose alcohol level matches the maximum that we found in the previous stage. Note that we need to use WITH to pass the max_alcohol value down the pipeline. Side note: to avoid the two duplicate MATCH clauses, we can use collect() to "save" the list of beers sold at Sober Bar, and then UNWIND them in the second stage. In this case, using COLLECT() might look like an overkill, but can serve as a good excersise.

✖

```
MATCH (beer:Beer)
WITH MAX(beer.alcohol) as max_alcohol
MATCH (beer:Beer)<-[:Sells]-(:Bar {name: "Sober Bar"})
WHERE beer.alcohol = max_alcohol
RETURN beer.name
```

- The beers in the first MATCH clause are not necessarily from Sober Bar, hence the max_alcohol value we get is the global max, hence in case the global max is greater than the local max within Sober Bar, none of the beers sold at Sober Bar will match that level of alcohol, therefore, nothing will be returned.

✖

```
MATCH (beer:Beer)<-[:Sells]-(:Bar {name: "Sober Bar"})
WITH MAX(beer.alcohol) as max_alcohol
MATCH (beer:Beer)
WHERE beer.alcohol = max_alcohol
RETURN beer.name
```

- The beers in the second MATCH clause are not necessarily from Sober Bar, so if a beer from another bar happens to have the same alcohol content as max_alcohol in Sober Bar, it will be returned as well, which violates the requirement.

HQ_6_10

[Food for Thought] The figure below shows the graph model for "Friday Night". Using Cypher, how can you find the beers (by name) with the highest *average* rating? Hint: learning how to use the COLLECT() function will be useful.

✔

```
MATCH (:Drinker)-[drinks:Drinks]->(beer:Beer)
WITH beer.name AS beer_name, AVG(drinks.rating) as avg_rating
WITH COLLECT([beer_name, avg_rating]) AS beer_avg_list, MAX(avg_rating) AS max_avg_rating
UNWIND beer_avg_list AS beer_avg
WITH beer_avg[0] AS beer_name, beer_avg[1] AS avg_rating, max_avg_rating
WHERE avg_rating = max_avg_rating
RETURN beer_name
```

- In the first stage, we compute the average rating for each beer. In the second stage, we compute the max among all the average ratings, obtaining the highest average rating, and at the same time, we save the "beer & avg_rating" relation obtained in stage one using COLLECT(). Finally, in the third stage, we find the beer(s) whose average rating matches the highest average rating. Note that before using the saved relation (as a single list), we must UNWIND the list first (similar to unwinding an array in MongoDB).

✖

```
MATCH (:Drinker)-[drinks:Drinks]->(beer:Beer)
WITH beer.name AS beer_name, AVG(drinks.rating) as avg_rating
WITH COLLECT([beer_name, avg_rating]) AS beer_avg, MAX(avg_rating) AS max_avg_rating
WITH beer_avg[0] AS beer_name, beer_avg[1] AS avg_rating, max_avg_rating
WHERE avg_rating = max_avg_rating
RETURN beer_name
```

- Here the stored list resulting from COLLECT() is not unwound before being used in the next stage, which is illegal.

✖

```
MATCH (:Drinker)-[drinks:Drinks]->(beer:Beer)
WITH beer.name AS beer_name, AVG(drinks.rating) as avg_rating
WITH beer_name, avg_rating, MAX(avg_rating) AS max_avg_rating
WHERE avg_rating = max_avg_rating
RETURN beer_name
```

- The second WITH clause intends to obtain the highest average rating among all the average ratings, however, because beer_name and avg_rating appear in the same clause, they will be used as the grouping key, and MAX() is called on each group, which is just a single beer-rating pair, hence this does not work as intended.
