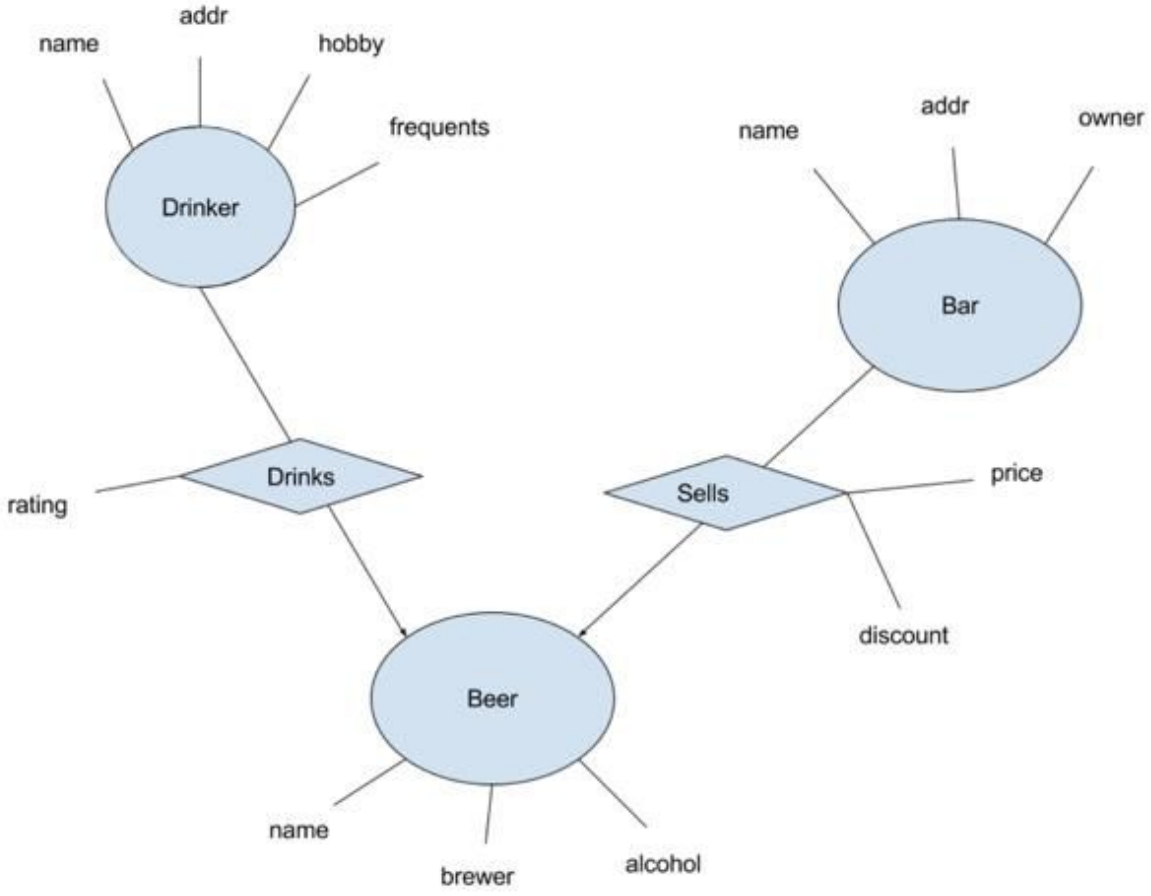
1 - Find the beers with the highest average

[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.

---

2 - SQL correspondence in cypher

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?

- noFigure

✔ 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

---

3 - Find all the "straight-A" restaurants

For a restaurant collection like the one shown in the figure, 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.


```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

✔

```
db.restaurants.aggregate([
   {$unwind: "$grades"},
   {$group: {
      _id: "$restaurant_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: "$restaurant_id",
      count_A: {$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.

---

4 - Find those "inconsistent" restaurants

For a restaurant collection like the one shown in the figure, how do you find those "inconsistent" restaurants that have received at least one grade "A", as well as at least one grade of "C"?


```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```


- ✔

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

- The $all operator will match an array (in this case, "grades.grade") if the array contains all the elements in the argument of $all, namely ["A", "C"].

✔

db.restaurants.find(
  {$and: [

```
      {"grades.grade": "A"},
      {"grades.grade": "C"}
      ]
   }
)
```

- The $all operator is equivalent to the $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.

---

5 - Find the average score for each of the cuisines and sort them

For a restaurant collection like the one shown in the figure, 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.

```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```


- ✔

```
db.restaurants.aggregate([
   {$unwind: "$grades"},
   {$group: {
      _id: "$restaurant_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(orfirst(orlast) 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.

---

6 - Join two collections

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?

- noFigure

✔

```
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

---

7 - Characteristics of different Databases

Here is a brief story about a startup that develops a user-contributed product review service. In order to manage and analyze their user data: they started off using database system X, and soon they realized that, as a new service, there was a constant need to modify their database schema, which hindered their effort to release new features quickly. Therefore, they switched to database system Y, and their schema flexibility is increased significantly.. However, as the service grew in popularity, their users became more and more connected, and system Y started to perform worse and worse, especially in displaying a user's feed page, where the content from all the user's friends needed to be fetched. Finally they discovered database system Z, and after the switch, system Z has been handling all the social-heavy functionalities of the service swiftly, and the developers have never looked back. Can you infer what X, Y, and Z are?

- noFigure

✔ X: PostgreSQL, Y: MongoDB, Z: Neo4j

- MySQL and PostgreSQL need to have fixed schema. MongoDB as a document database, the documents in a collection are not necessary to have the same schema and thus it is flexible on modifying the database schema. Neo4j is graph database where the relationships between entities are emphasized and related query are handled efficiently.

✖ X: MySQL, Y: MongoDB, Z: PostgreSQL

- MySQL and PostgreSQL need to have fixed schema. MongoDB as a document database, the documents in a collection are not necessary to have the same schema and thus it is flexible on modifying the database schema. Neo4j is graph database where the relationships between entities are emphasized and related query are handled efficiently.

✖ X: MySQL, Y: Neo4j, Z: MongoDB

- MySQL and PostgreSQL need to have fixed schema. MongoDB as a document database, the documents in a collection are not necessary to have the same schema and thus it is flexible on modifying the database schema. Neo4j is graph database where the relationships between entities are emphasized and related query are handled efficiently.

---

8 - Finding real buddies

Given the following collection “Favorites”:
{ "_id" : ObjectId("5b30800fea23cdc4ca8b61d6"), "drinker" : "Alex", "bar" : "Sober Bar", "beer" : "bud", "season" : "Spring" }
{ "_id" : ObjectId("5b30801eea23cdc4ca8b61d7"), "drinker" : "Alex", "bar" : "Green Bar", "beer" : "Sam Adams", "season" : "Summer" }
{ "_id" : ObjectId("5b30802aea23cdc4ca8b61d8"), "drinker" : "Cindy", "bar" : "Purple Bar", "beer" : "Sam Adams", "season" : "Summer" }
{ "_id" : ObjectId("5b308044ea23cdc4ca8b61d9"), "drinker" : "Mike", "bar" : "Green Bar", "beer" : "Sam Adams", "season" : "Winter" }

What result will you get after running the following code?

```
db.Favorites.aggregate([
{$lookup:{from:"Favorites", localField:"beer", foreignField:"beer", as:"buddy"}},
{$unwind:"$buddy"},
{$project:{
   drinker:1, "buddy.drinker":1,
   cmp_bars:{$cmp:["$bar", "$buddy.bar"]},
   cmp_drinkers:{$cmp:["$drinker", "$buddy.drinker"]}}},
{$match:{cmp_bars:0, cmp_drinkers:1}},
{$project:{"_id":0, "real_buddy1":"$drinker", "real_buddy2":"$buddy.drinker"}}
])
```

- noFigure

✔ { "real_buddy1" : "Mike", "real_buddy2" : "Alex" }

- This aggregate function will find drinkers in the Favorites collection with the same bar and beer, while the output has different names (to avoid outputting one to himself). Specifically, “real_buddy1” should be greater than “real_buddy2” alphabetically.

✖ { "real_buddy1" : "Alex", "real_buddy2" : "Cindy" }

- “real_buddy1” should be greater than “real_buddy2” alphabetically.

✖ { "real_buddy1" : "Mike", "real_buddy2" : "Alex" } { "real_buddy1" : "Cindy", "real_buddy2" : "Alex" }

- Cindy does not have the same favorite bar as Alex.

✖ { "real_buddy1" : "Alex", "real_buddy2" : "Alex" } { "real_buddy1" : "Alex", "real_buddy2" : "Mike" } { "real_buddy1" : "Mike", "real_buddy2" : "Mike" } { "real_buddy1" : "Mike", "real_buddy2" : "Alex" }

- “real_buddy1” should be greater than “real_buddy2” alphabetically, and identical match will be removed.

---

9 - MongoDB & SQL aggregates

Which of the following SQL terminology is the same as $match in MongoDB?

- noFigure

✔ WHERE

- WHERE is for filtering records in SQL, and we can use $match in MongoDB to filter records that satisfy certain conditions.

✔ HAVING

- HAVING is for filtering groups in SQL, and we can use $match in MongoDB to filter the groups that satisfy certain conditions.

✖ GROUP BY

- GROUP BY is for grouping the results, not selecting.

---

10 - db.collection.find() capabilities

Which of the following relational operations are supported in db.collection.find()?

- noFigure

✔ Selection, Projection

- db.collection.find() only directly supports Selection and Projection.

✖ Selection, Projection, Renaming

- db.collection.find() only directly supports Selection and Projection.

✖ Selection, Cartesian Product

- db.collection.find() only directly supports Selection and Projection.

✖ Set Union, Set Difference

- db.collection.find() only directly supports Selection and Projection.
