Course: Advanced Databases (NoSQL)

Assignment: VI

Student's name: Tulentay Ramazan

Group: BDA-2407

Student ID: 240463

Database used: airbnb.bson

Instructor: Dinara Zhunissova

Date of submission: 1.02.2026 , 8:55 PM

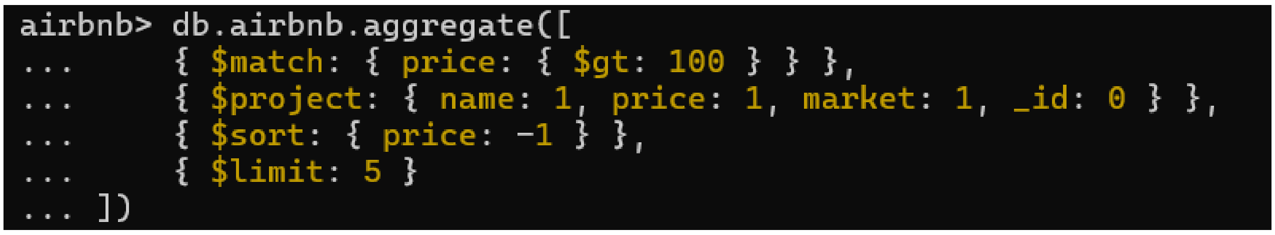
Deadline of submission: 1.02.2026 , 9:00 PM

Content

Part I: Aggregation pipeline basics

Task 1: Create an aggregation pipeline that groups documents by market, calculates the average price per market, and sorts the results by average price in descending order.

The next code helps us to group documents by market, calculate the average price per market and sort the results in descending order.



```
airbnb> db.airbnb.aggregate([
...     { $match: { price: { $gt: 100 } } },
...     { $project: { name: 1, price: 1, market: 1, _id: 0 } },
...     { $sort: { price: -1 } },
...     { $limit: 5 }
... ])
```

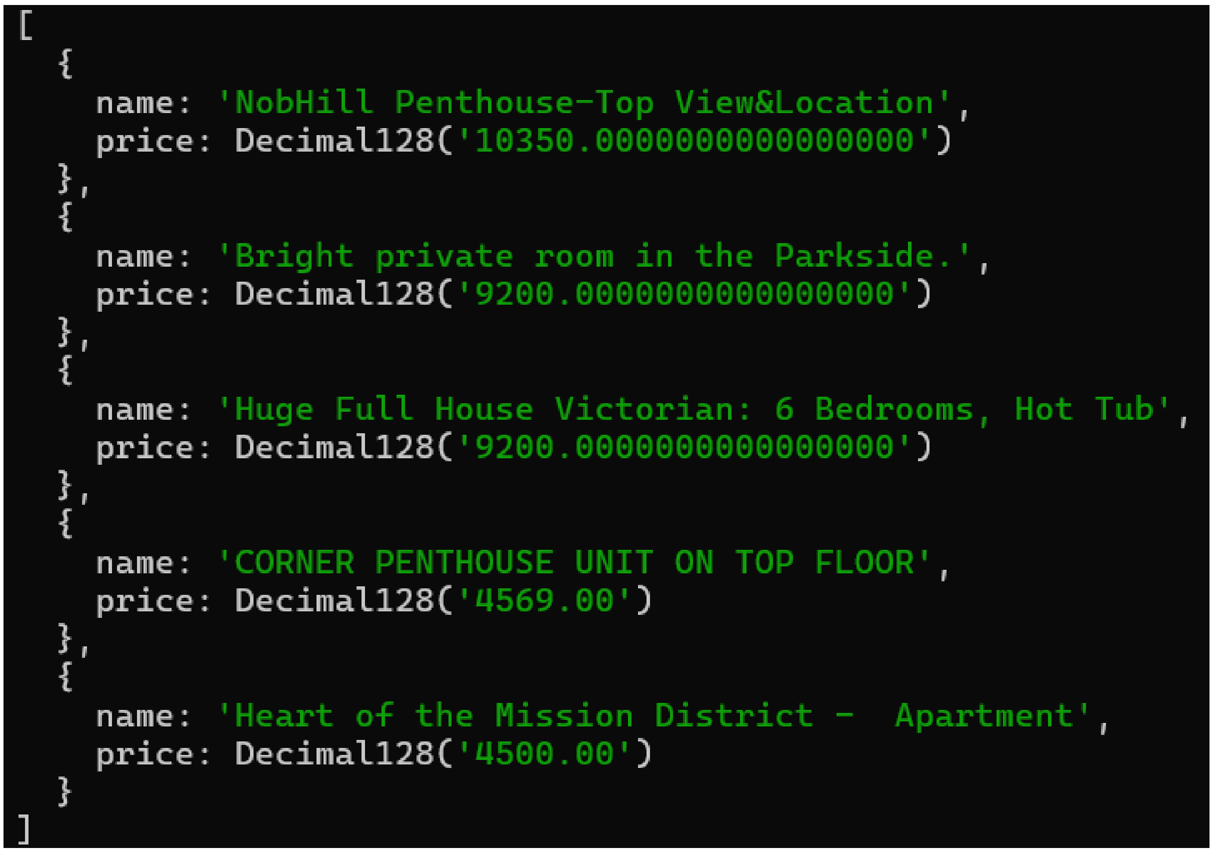
$match filters by the price, which is greater 100.

The fields, such as **name, price & market** should be projected by the command: **$project** (in my case, all fields, except **_id,** are set to 1).

Price: -1 means descending order in the **$sort** command.

We limit the output of the amount of documents with 5 documents maximum in the **$limit** command.

Output:



```
[
  {
    name: 'NobHill Penthouse-Top View&Location',
    price: Decimal128('10350.0000000000000000')
  },
  {
    name: 'Bright private room in the Parkside.',
    price: Decimal128('9200.0000000000000000')
  },
  {
    name: 'Huge Full House Victorian: 6 Bedrooms, Hot Tub',
    price: Decimal128('9200.0000000000000000')
  },
  {
    name: 'CORNER PENTHOUSE UNIT ON TOP FLOOR',
    price: Decimal128('4569.00')
  },
  {
    name: 'Heart of the Mission District -  Apartment',
    price: Decimal128('4500.00')
  }
]
```

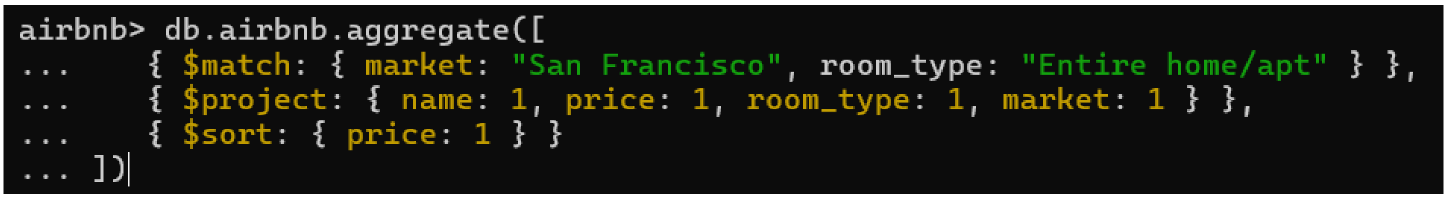
The output result shows:

- the names and prices of the most expensive listings, matching the criteria;
- all prices in Decimal128 that ensures accuracy for large numbers.

In general, the output result confirms that the filter and sorting are working correctly, and limit(5) reduces the output to the most relevant documents.

Task 2: Create an aggregation pipeline that filters listings located in San Francisco, filters further by room type equal to **“Entire home/apt”**, applies projection to display relevant fields, and sorts the results by price in ascending order.

Input code:



```
airbnb> db.airbnb.aggregate([
...     { $match: { market: "San Francisco", room_type: "Entire home/apt" } },
...     { $project: { name: 1, price: 1, room_type: 1, market: 1 } },
...     { $sort: { price: 1 } }
... ])
```

In the **db.airbnb.aggregate()** command, we write:

$match command that filters listings in **San Francisco** by room_type: **“Entire home/apt” ;**

$project, inside which all fields, such as **name, price, market & room_type** are set into 1.

$sort , inside which there’s a price, which is set 1 - in ascending order.

After running the code, the output hasn’t been printed, because no one document satisfies all given conditions of filtration simultaneously.

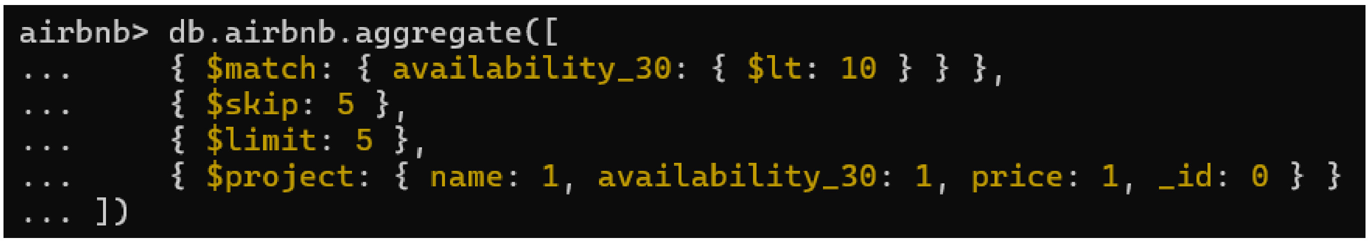
Because, in the **airbnb.bson** dataset, there are no any listings, which

- located in **San Francisco;**
- have one room **“Entire home/apt”**,
- and correspond remaining conditions of aggregation.

In conclusion, the query has been completely executed, but there’s no any suitable document in the dataset.

Task 3: Design an aggregation pipeline that retrieves documents where availability_30 is less than 10, skips the first 5 documents, returns the next 5 documents, and applies projection to display availability and price-related fields.

Input code:



```
airbnb> db.airbnb.aggregate([
...     { $match: { availability_30: { $lt: 10 } } },
...     { $skip: 5 },
...     { $limit: 5 },
...     { $project: { name: 1, availability_30: 1, price: 1, _id: 0 } }
... ])
```

$match retrieves documents, where availability_30 less than 10 (**$lt:** 10).

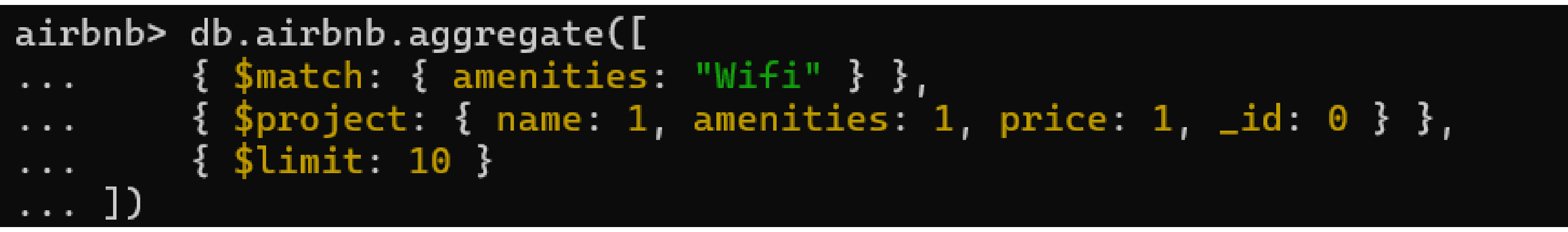
$skip skips 5 first and the next 5 documents.

$project command applies projection to display **name, price and availability.**

Like in the previous task, there aren't any documents in the dataset, despite the correct code. Thus, the output result is empty.

Task 4: Create an aggregation pipeline that matches documents where the amenities include “Wifi”, projects the name, amenities, and price fields, and limits the output to 10 documents.

Code:



```
airbnb> db.airbnb.aggregate([
...     { $match: { amenities: "Wifi" } },
...     { $project: { name: 1, amenities: 1, price: 1, _id: 0 } },
...     { $limit: 10 }
... ])
```

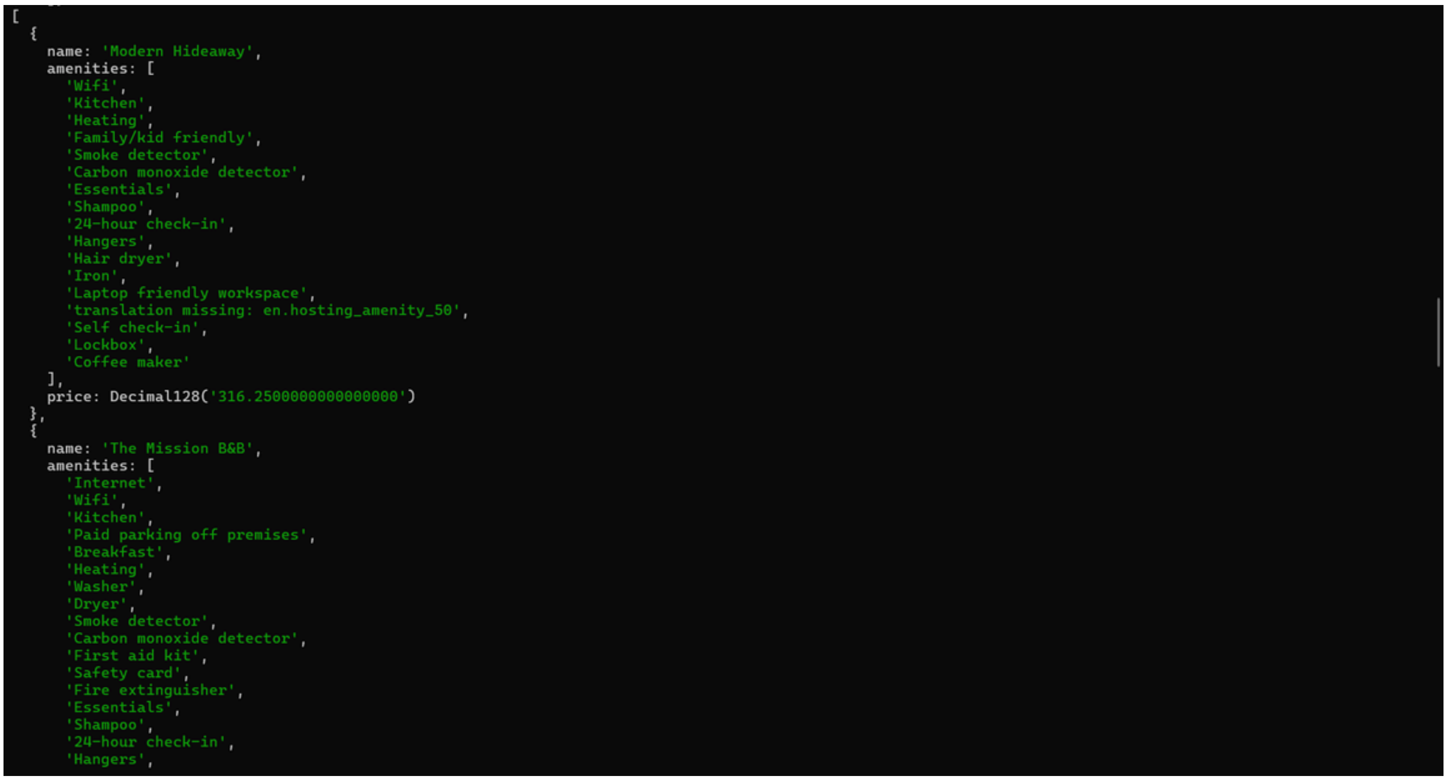
Inside the **db.airbnb.aggregate()** command, we write:

$match command, inside which there’re only **“Wifi”** amenities to match only correspondent amenities;

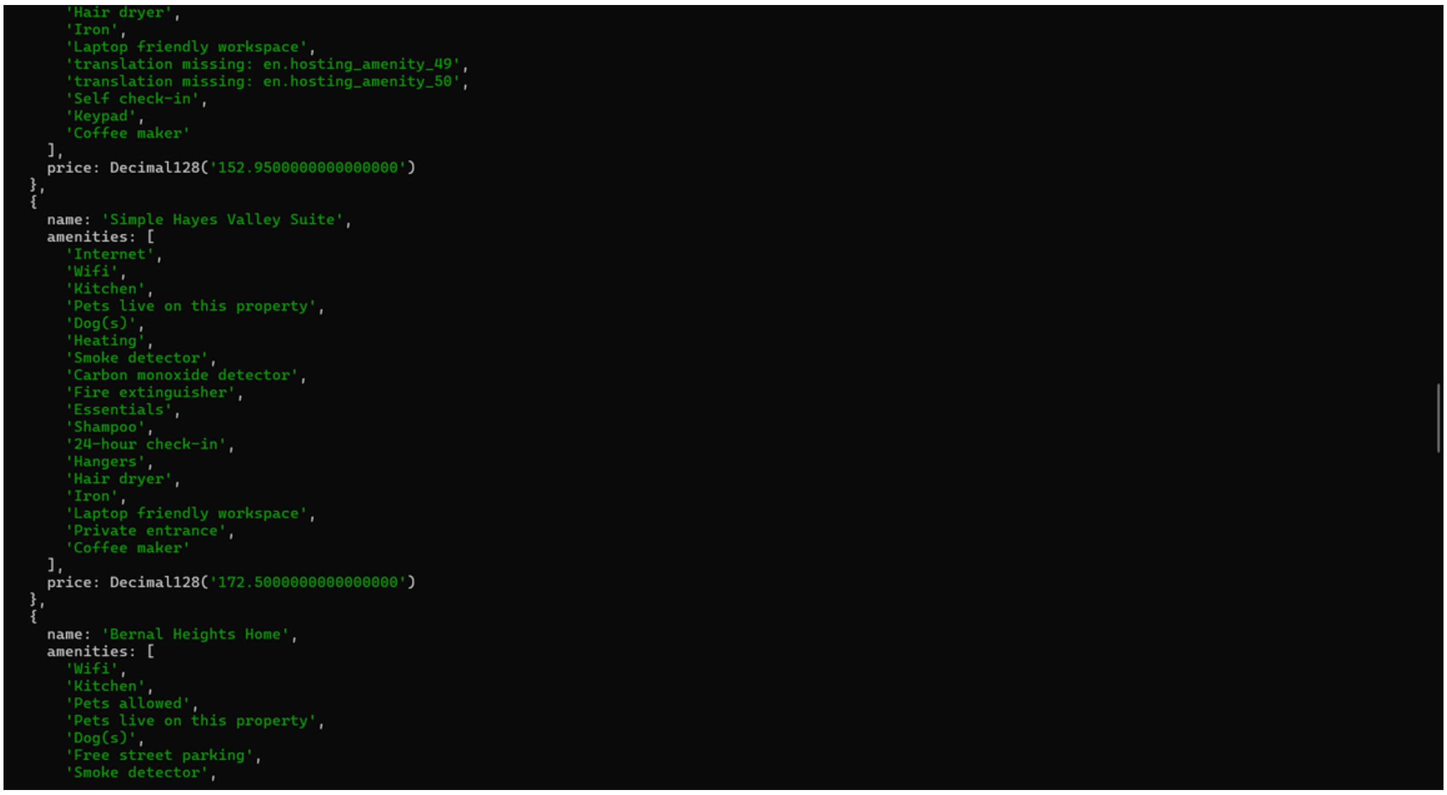
$project to choose suitable fields: **name, amenities, price;**

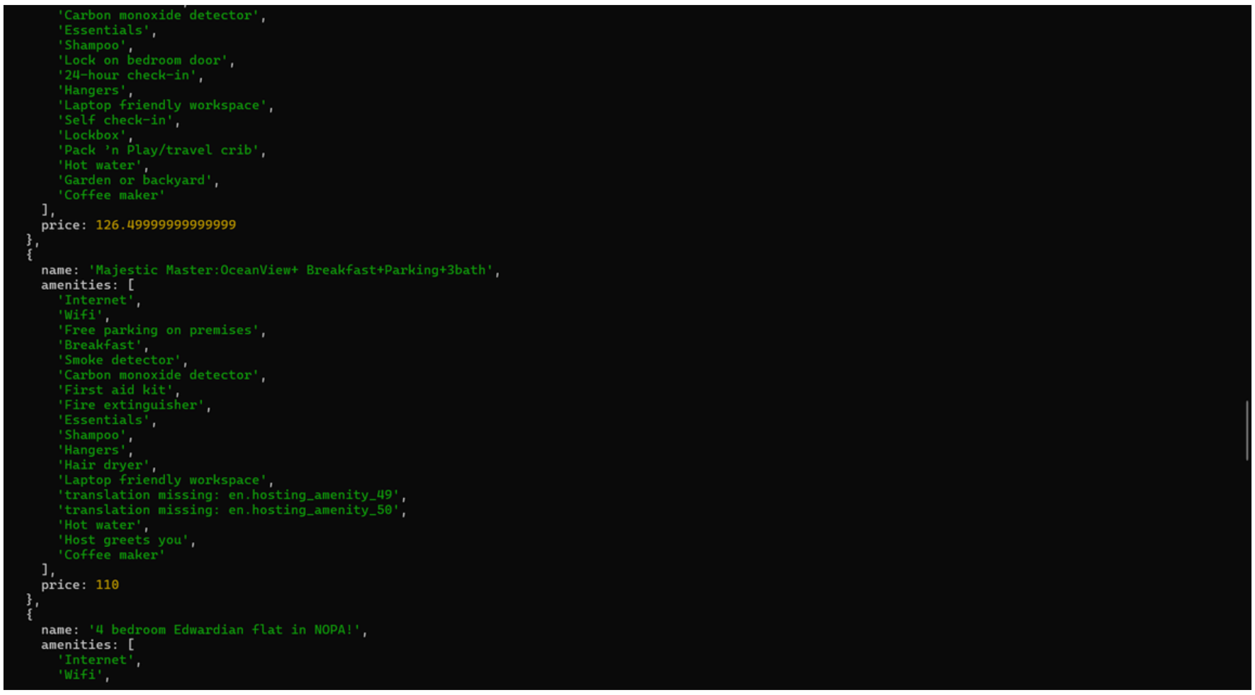
$limit: 10 to limit the output to 10 documents.

Output:

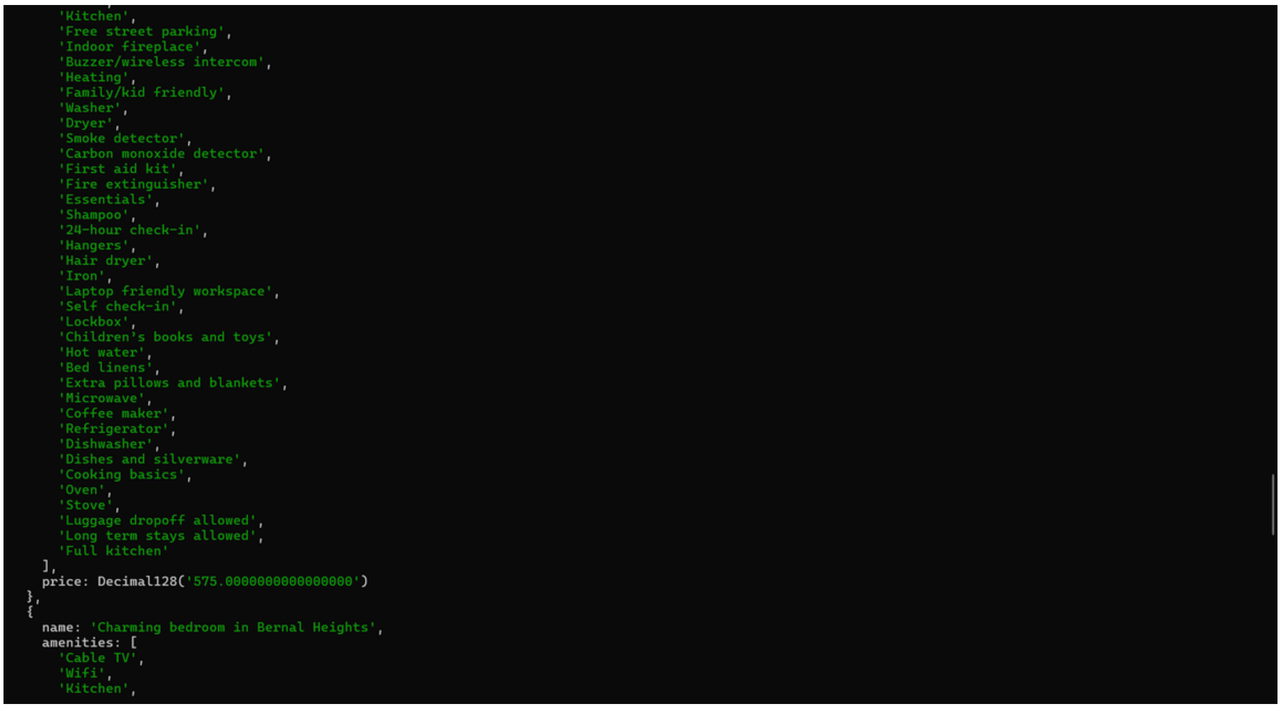


```
[
  {
    name: 'Modern Hideaway',
    amenities: [
      'Wifi',
      'Kitchen',
      'Heating',
      'Family/kid friendly',
      'Smoke detector',
      'Carbon monoxide detector',
      'Essentials',
      'Shampoo',
      '24-hour check-in',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'translation missing: en.hosting_amenity_50',
      'Self check-in',
      'Lockbox',
      'Coffee maker'
    ],
    price: Decimal128('316.2500000000000000')
  },
  {
    name: 'The Mission B&B',
    amenities: [
      'Internet',
      'Wifi',
      'Kitchen',
      'Paid parking off premises',
      'Breakfast',
      'Heating',
      'Washer',
      'Dryer',
      'Smoke detector',
      'Carbon monoxide detector',
      'First aid kit',
      'Safety card',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      '24-hour check-in',
      'Hangers',
```

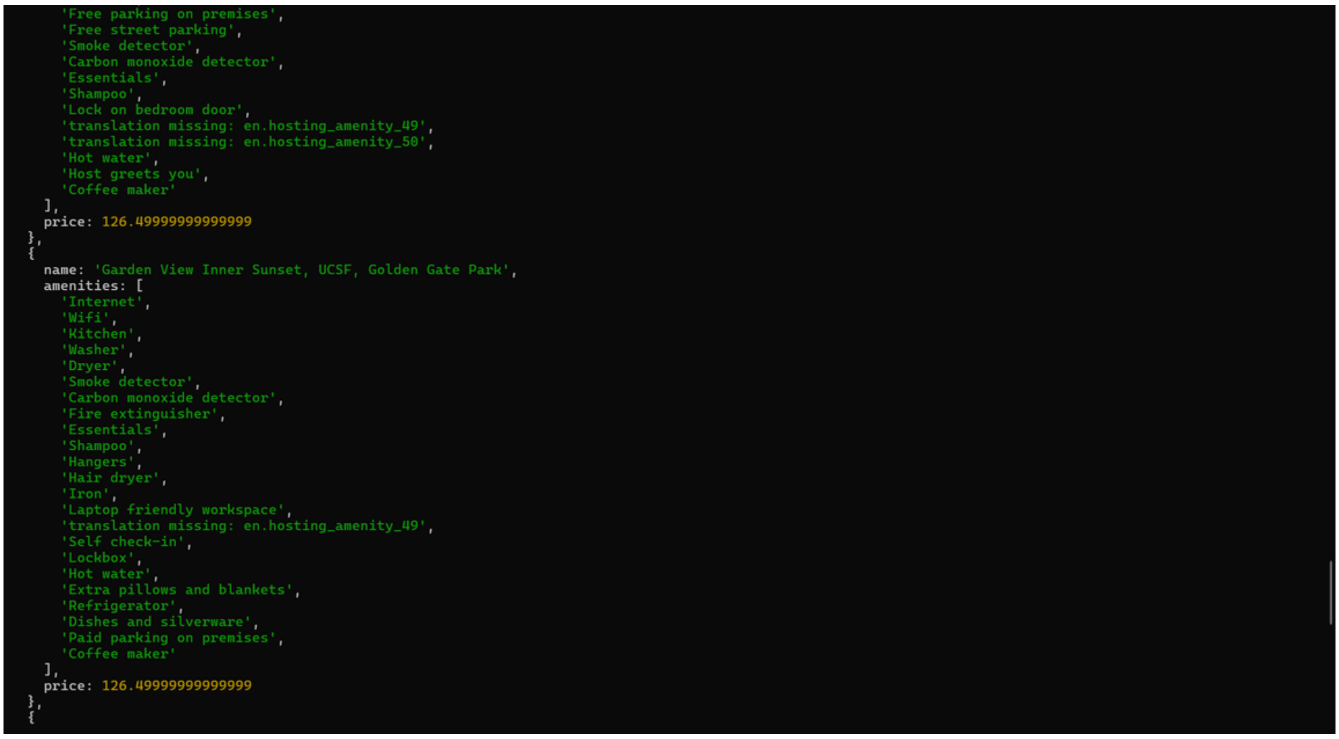

```
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'translation missing: en.hosting_amenity_49',
      'translation missing: en.hosting_amenity_50',
      'Self check-in',
      'Keypad',
      'Coffee maker'
    ],
    price: Decimal128('152.9500000000000000')
  },
  {
    name: 'Simple Hayes Valley Suite',
    amenities: [
      'Internet',
      'Wifi',
      'Kitchen',
      'Pets live on this property',
      'Dog(s)',
      'Heating',
      'Smoke detector',
      'Carbon monoxide detector',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      '24-hour check-in',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'Private entrance',
      'Coffee maker'
    ],
    price: Decimal128('172.5000000000000000')
  },
  {
    name: 'Bernal Heights Home',
    amenities: [
      'Wifi',
      'Kitchen',
      'Pets allowed',
      'Pets live on this property',
      'Dog(s)',
      'Free street parking',
      'Smoke detector',
```



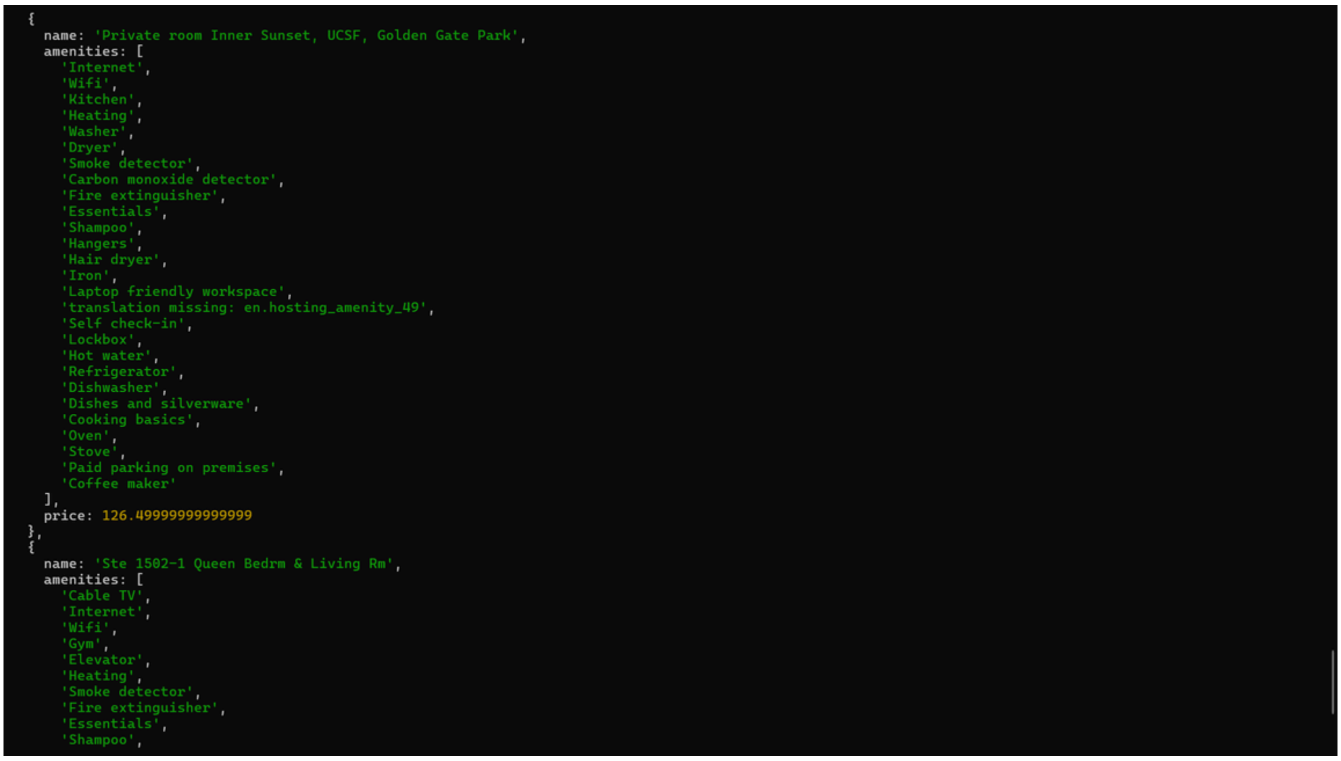
```
      'Carbon monoxide detector',
      'Essentials',
      'Shampoo',
      'Lock on bedroom door',
      '24-hour check-in',
      'Hangers',
      'Laptop friendly workspace',
      'Self check-in',
      'Lockbox',
      'Pack 'n Play/travel crib',
      'Hot water',
      'Garden or backyard',
      'Coffee maker'
    ],
    price: 126.49999999999999
  },
  {
    name: 'Majestic Master:OceanView+ Breakfast+Parking+3bath',
    amenities: [
      'Internet',
      'Wifi',
      'Free parking on premises',
      'Breakfast',
      'Smoke detector',
      'Carbon monoxide detector',
      'First aid kit',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      'Hangers',
      'Hair dryer',
      'Laptop friendly workspace',
      'translation missing: en.hosting_amenity_49',
      'translation missing: en.hosting_amenity_50',
      'Hot water',
      'Host greets you',
      'Coffee maker'
    ],
    price: 110
  },
  {
    name: '4 bedroom Edwardian flat in NOPA!',
    amenities: [
      'Internet',
      'Wifi',
```



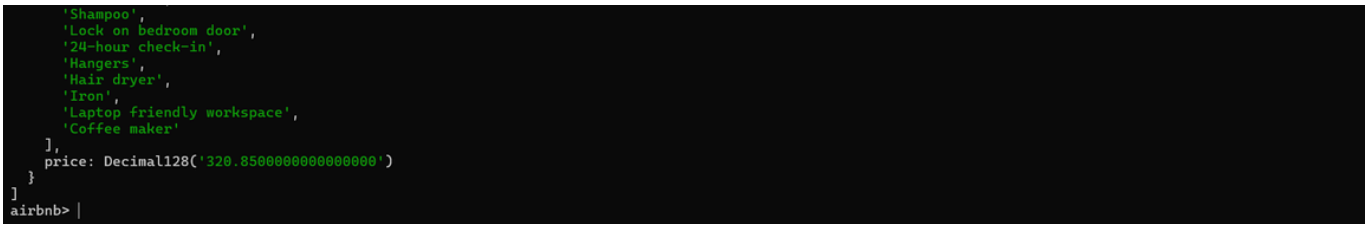
```
      'Kitchen',
      'Free street parking',
      'Indoor fireplace',
      'Buzzer/wireless intercom',
      'Heating',
      'Family/kid friendly',
      'Washer',
      'Dryer',
      'Smoke detector',
      'Carbon monoxide detector',
      'First aid kit',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      '24-hour check-in',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'Self check-in',
      'Lockbox',
      'Children's books and toys',
      'Hot water',
      'Bed linens',
      'Extra pillows and blankets',
      'Microwave',
      'Coffee maker',
      'Refrigerator',
      'Dishwasher',
      'Dishes and silverware',
      'Cooking basics',
      'Oven',
      'Stove',
      'Luggage dropoff allowed',
      'Long term stays allowed',
      'Full kitchen'
    ],
    price: Decimal128('575.0000000000000000')
  },
  {
    name: 'Charming bedroom in Bernal Heights',
    amenities: [
      'Cable TV',
      'Wifi',
      'Kitchen',
```



```
      'Free parking on premises',
      'Free street parking',
      'Smoke detector',
      'Carbon monoxide detector',
      'Essentials',
      'Shampoo',
      'Lock on bedroom door',
      'translation missing: en.hosting_amenity_49',
      'translation missing: en.hosting_amenity_50',
      'Hot water',
      'Host greets you',
      'Coffee maker'
    ],
    price: 126.49999999999999
  },
  {
    name: 'Garden View Inner Sunset, UCSF, Golden Gate Park',
    amenities: [
      'Internet',
      'Wifi',
      'Kitchen',
      'Washer',
      'Dryer',
      'Smoke detector',
      'Carbon monoxide detector',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'translation missing: en.hosting_amenity_49',
      'Self check-in',
      'Lockbox',
      'Hot water',
      'Extra pillows and blankets',
      'Refrigerator',
      'Dishes and silverware',
      'Paid parking on premises',
      'Coffee maker'
    ],
    price: 126.49999999999999
  },
  {
```



```
  {
    name: 'Private room Inner Sunset, UCSF, Golden Gate Park',
    amenities: [
      'Internet',
      'Wifi',
      'Kitchen',
      'Heating',
      'Washer',
      'Dryer',
      'Smoke detector',
      'Carbon monoxide detector',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'translation missing: en.hosting_amenity_49',
      'Self check-in',
      'Lockbox',
      'Hot water',
      'Refrigerator',
      'Dishwasher',
      'Dishes and silverware',
      'Cooking basics',
      'Oven',
      'Stove',
      'Paid parking on premises',
      'Coffee maker'
    ],
    price: 126.49999999999999
  },
  {
    name: 'Ste 1502-1 Queen Bedrm & Living Rm',
    amenities: [
      'Cable TV',
      'Internet',
      'Wifi',
      'Gym',
      'Elevator',
      'Heating',
      'Smoke detector',
      'Fire extinguisher',
      'Essentials',
      'Shampoo',
```



```
      'Shampoo',
      'Lock on bedroom door',
      '24-hour check-in',
      'Hangers',
      'Hair dryer',
      'Iron',
      'Laptop friendly workspace',
      'Coffee maker'
    ],
    price: Decimal128('320.8500000000000000')
  }
]
airbnb> |
```

After executing the **aggregation pipeline** command, the list of announcements, that satisfy the given conditions of query.

Here are the documents that were shown:

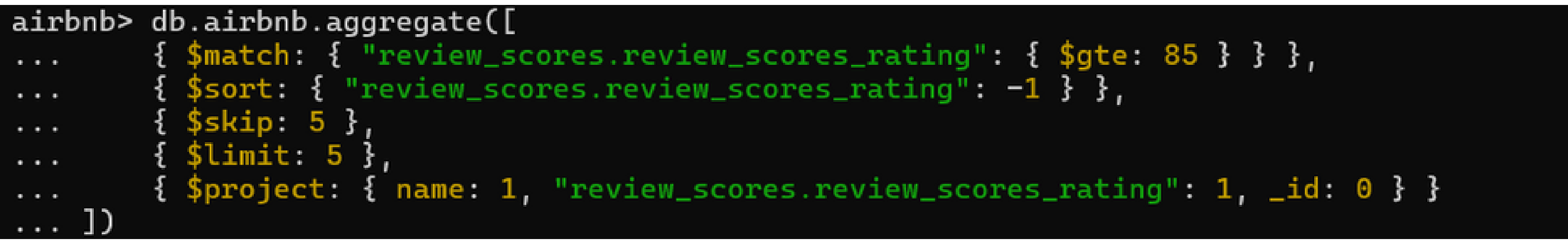
- **name -** the name of announcement;
- **amenitites -** array of available easements;
- **price -** cost of residence.

Aggregation correctly processed the **amenities** array, saving the values without changing and returned only these fields that were specified on the **$project** stage.

Decimal128 and **Number** formats are related to the features of storing numerical data in the original dataset and don't affect the correctness of the output.

Task 5: Build an aggregation pipeline that retrieves documents where the review score rating is greater than or equal to 85, sorts the results by rating, skips the first 5 documents, and returns the next 5 documents.

Input code:



```
airbnb> db.airbnb.aggregate([
...     { $match: { "review_scores.review_scores_rating": { $gte: 85 } } },
...     { $sort: { "review_scores.review_scores_rating": -1 } },
...     { $skip: 5 },
...     { $limit: 5 },
...     { $project: { name: 1, "review_scores.review_scores_rating": 1, _id: 0 } }
... ])
```

Inside the **db.airbnb.aggregate()** command, we write:

$match to match the field “**review_scores.review_scores_rating”,** the quantity of which is greater than or equal to 85.

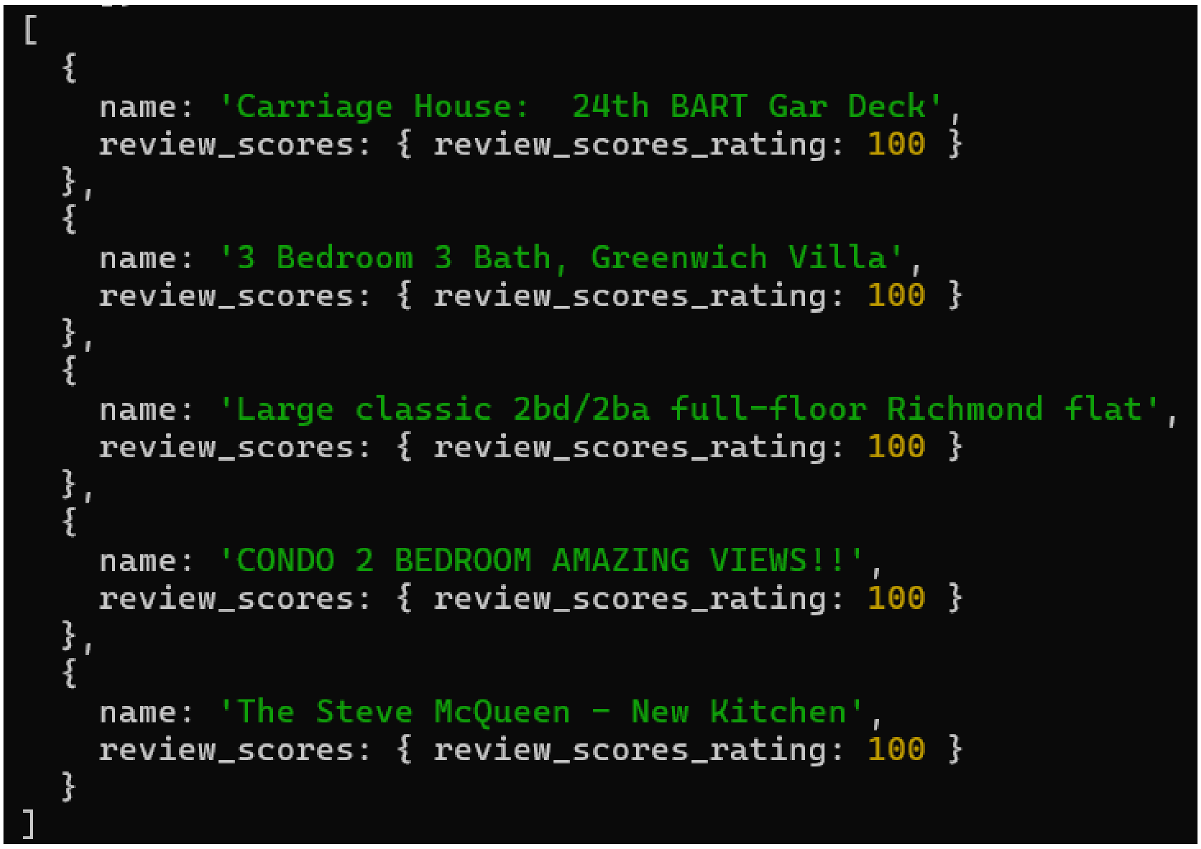
$sort to sort the field “**review_scores.review_scores_rating”** in the descending order (-1).

$skip: 5 to skip first 5 documents

$limit: 5 to return the next 5 documents

$project to select suitable fields: **name: 1,** “**review_scores.review_scores_rating”** and **id.**

Output result:



```
[
  {
    name: 'Carriage House:  24th BART Gar Deck',
    review_scores: { review_scores_rating: 100 }
  },
  {
    name: '3 Bedroom 3 Bath, Greenwich Villa',
    review_scores: { review_scores_rating: 100 }
  },
  {
    name: 'Large classic 2bd/2ba full-floor Richmond flat',
    review_scores: { review_scores_rating: 100 }
  },
  {
    name: 'CONDO 2 BEDROOM AMAZING VIEWS!!',
    review_scores: { review_scores_rating: 100 }
  },
  {
    name: 'The Steve McQueen - New Kitchen',
    review_scores: { review_scores_rating: 100 }
  }
]
```

In conclusion, the query retrieved documents, in which the field **review_scores.review_scores_rating** has the max value 100, and has shown the necessary fields:

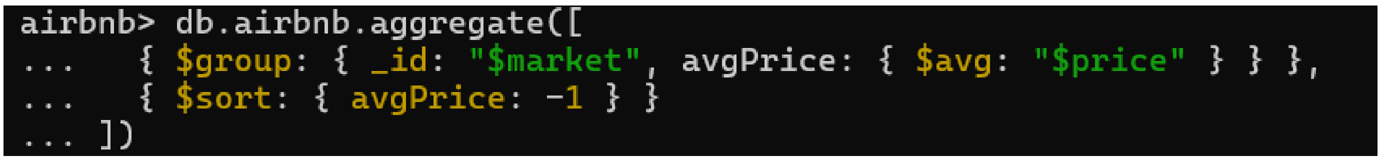
- **name -** the name of announcements;
- **review_scores.review_scores_rating -** the general rating by reviews.

Using **aggregation** allowed us to correctly access the nested review_scores object and filter by its field without retrieving unnecessary data.

PART II. Group Stage ($group)

Task 1: Create an aggregation pipeline that groups documents by market, calculates the average price per market, and sorts the results by average price in descending order.

Correspondent code:



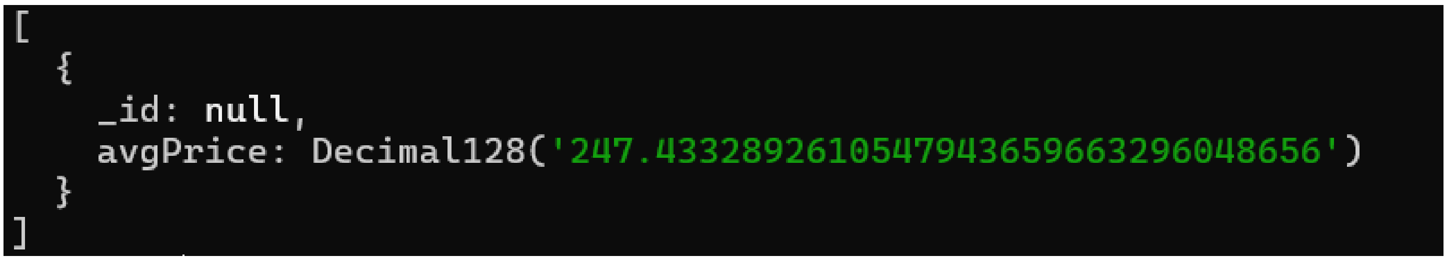
```
airbnb> db.airbnb.aggregate([
...   { $group: { _id: "$market", avgPrice: { $avg: "$price" } } },
...   { $sort: { avgPrice: -1 } }
... ])
```

In this case, we should write inside the **db.airbnb.aggregate()** command:

$group to group documents by **"$market"** and calculate the **avgPrice** per market;

$sort to sort **avgPrice** in descending order (-1).

Output:

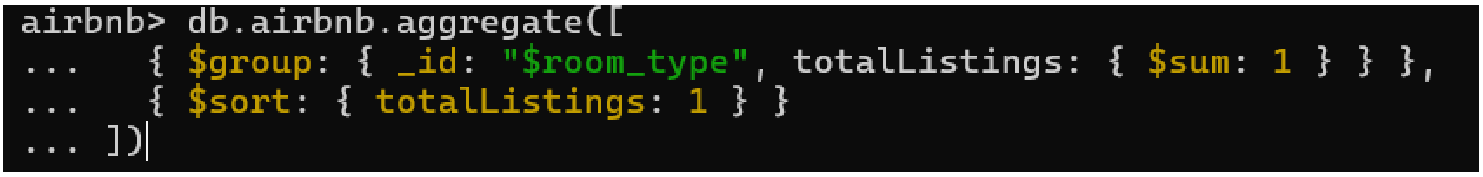


```
[
  {
    _id: null,
    avgPrice: Decimal128('247.4332892610547943659663296048656')
  }
]
```

The aggregation groups all listings by the __id, which in this situation is **null,** because no one grouping key is specified. In conclusion, MongoDB calculates the average price across the entire collection, returning a single value **avgPrice.** This demonstrates the overall average price of all Airbnb listings in the dataset.

Task 2: Build an aggregation pipeline that groups documents by room type, calculates the total number of listings per room type, and sorts the results by total listings.

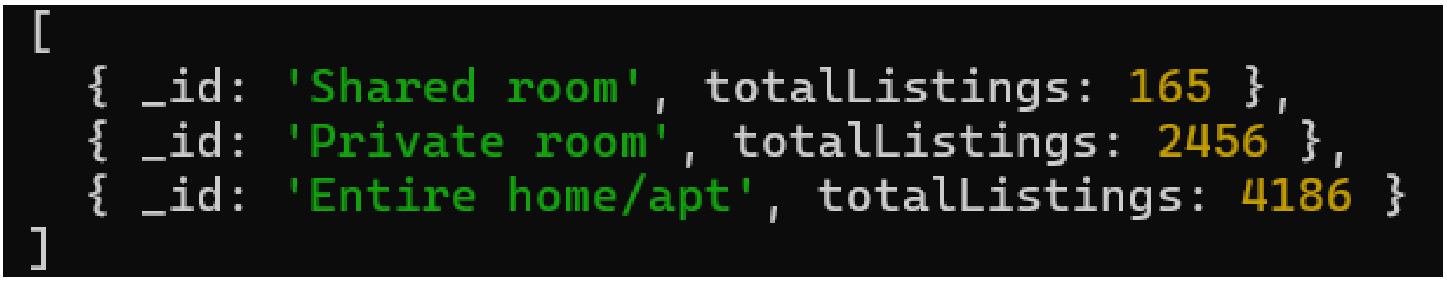
For solving this task, the most suitable code is:



```
airbnb> db.airbnb.aggregate([
...    { $group: { _id: "$room_type", totalListings: { $sum: 1 } } },
...    { $sort: { totalListings: 1 } }
... ])
```

In this case, we use **$group** to group documents by **"$room_type"**, with the total number (**$sum**): 1. We also use **$sort** to sort **totalListings** in ascending order.

Output:



```
[
  { _id: 'Shared room', totalListings: 165 },
  { _id: 'Private room', totalListings: 2456 },
  { _id: 'Entire home/apt', totalListings: 4186 }
]
```

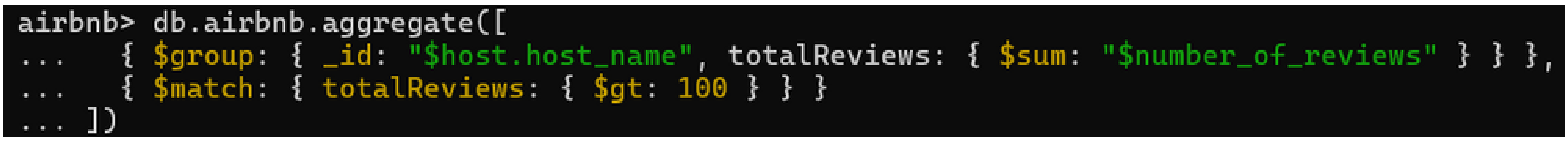
The aggregation groups listings by room type and counts how many listings exist for every type. Sorting in ascending order shows the room types with the fewest listings first. In my situation, this code has shown 3 listings:

- Shared room;
- Private room;
- Entire home/apt.

At the same time, the total listings are in ascending order that accepts the correction of code: 165 → 2456 → 4186.

Task 3: Design an aggregation pipeline that groups documents by host, calculates the total number of reviews per host, and filters the results to include only hosts with total reviews above a specified threshold.

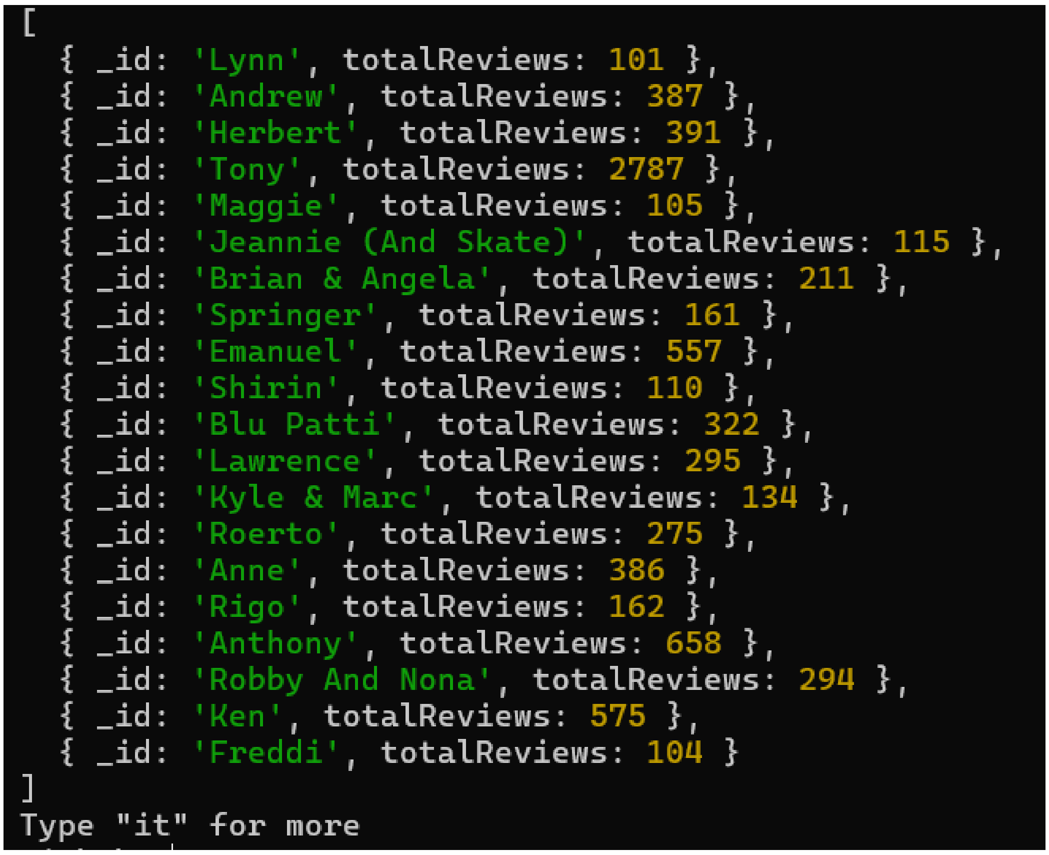
Code:



```
airbnb> db.airbnb.aggregate([
...   { $group: { _id: "$host.host_name", totalReviews: { $sum: "$number_of_reviews" } } },
...   { $match: { totalReviews: { $gt: 100 } } }
... ])
```

We use **$group** to group documents by **"$host.host_name"** and **$sum** to calculate the total **"$number_of_reviews" ,** matching by **totalReviews**, the quantity of which is greater than 100.

Output:

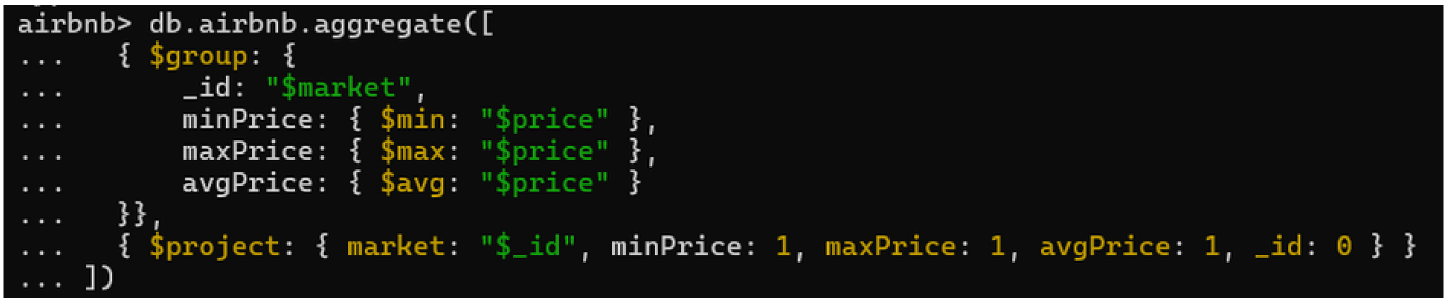


```
[
  { _id: 'Lynn', totalReviews: 101 },
  { _id: 'Andrew', totalReviews: 387 },
  { _id: 'Herbert', totalReviews: 391 },
  { _id: 'Tony', totalReviews: 2787 },
  { _id: 'Maggie', totalReviews: 105 },
  { _id: 'Jeannie (And Skate)', totalReviews: 115 },
  { _id: 'Brian & Angela', totalReviews: 211 },
  { _id: 'Springer', totalReviews: 161 },
  { _id: 'Emanuel', totalReviews: 557 },
  { _id: 'Shirin', totalReviews: 110 },
  { _id: 'Blu Patti', totalReviews: 322 },
  { _id: 'Lawrence', totalReviews: 295 },
  { _id: 'Kyle & Marc', totalReviews: 134 },
  { _id: 'Roerto', totalReviews: 275 },
  { _id: 'Anne', totalReviews: 386 },
  { _id: 'Rigo', totalReviews: 162 },
  { _id: 'Anthony', totalReviews: 658 },
  { _id: 'Robby And Nona', totalReviews: 294 },
  { _id: 'Ken', totalReviews: 575 },
  { _id: 'Freddi', totalReviews: 104 }
]
Type "it" for more
```

The aggregation code has grouped some documents by host and shown the names with the quantity of **totalReviews.** As a result, the quantity of **totalReviews** is greater than 100, according to the correctness of the code. (From 101 to 2787).

Task 4: Create an aggregation pipeline that groups listings by market, calculates the minimum price, maximum price, and average price, and applies projection to display only the market and calculated fields.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $group: {
...       _id: "$market",
...       minPrice: { $min: "$price" },
...       maxPrice: { $max: "$price" },
...       avgPrice: { $avg: "$price" }
...   }},
...   { $project: { market: "$_id", minPrice: 1, maxPrice: 1, avgPrice: 1, _id: 0 } }
... ])
```

Inside **$group:** we write:

__id: “$market” - to group by market;

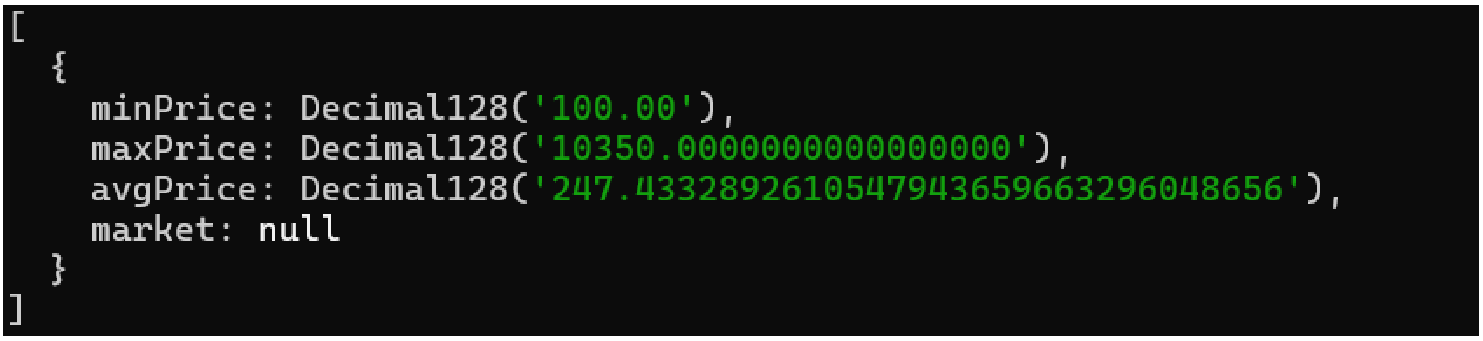
minPrice: { $min: “$price”} - to calculate the minimum price;

maxPrice: { $max: “$price”} - to calculate the maximum price;

avgPrice: { $avg: “$price”} - to calculate the average price;

$project to choose correspondent fields: **market, minPrice, maxPrice, avgPrice, _id.**

Output:



```
[
  {
    minPrice: Decimal128('100.00'),
    maxPrice: Decimal128('10350.0000000000000000'),
    avgPrice: Decimal128('247.4332892610547943659663296048656'),
    market: null
  }
]
```

In the output, the documents have been grouped by the **market** value and the query has calculated the:

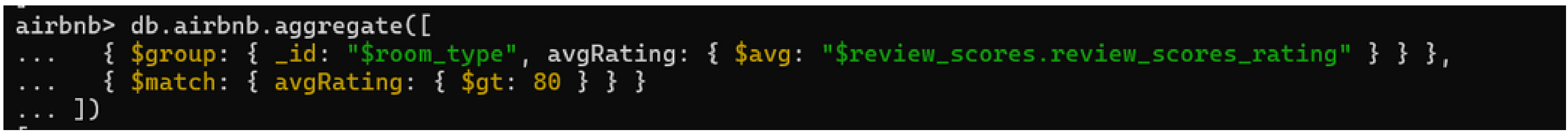
- **minPrice: 100.00**
- **maxPrice: 10350.0000000000000000;**

- **avgPrice: 247.43328926105479436596632960486**
- **market: null.**

In the **market** field there’s the result null, because the grouping actually occured by a missing **market** value.

Task 5: Build an aggregation pipeline that groups documents by room type, calculates the average review score rating, and returns only room types with an average rating greater than 80.

The query code:

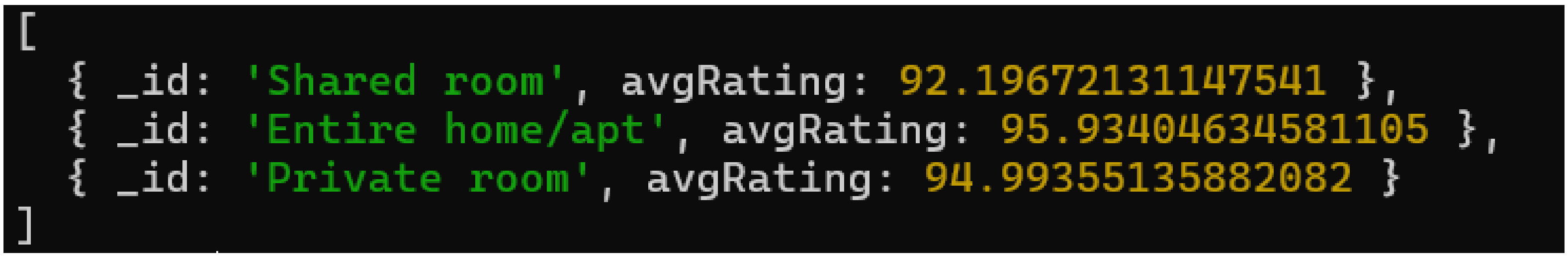


```
airbnb> db.airbnb.aggregate([
...   { $group: { _id: "$room_type", avgRating: { $avg: "$review_scores.review_scores_rating" } } },
...   { $match: { avgRating: { $gt: 80 } } }
... ])
```

In the **$group** command, we write **__id: "$room_type"** to group documents by room type and **avgRating** to calculate the average score rating (inside **avgRating → $avg: "$review_scores.review_scores_rating"**).

Also we use **$match** to return only room types with an average rating greater than 80 (**avgRating: { $gt: 80 }**).

Output:



```
[
  { _id: 'Shared room', avgRating: 92.19672131147541 },
  { _id: 'Entire home/apt', avgRating: 95.93404634581105 },
  { _id: 'Private room', avgRating: 94.99355135882082 }
]
```

According to the screenshot, the aggregation code has grouped documents by room type and calculated the necessary average score.

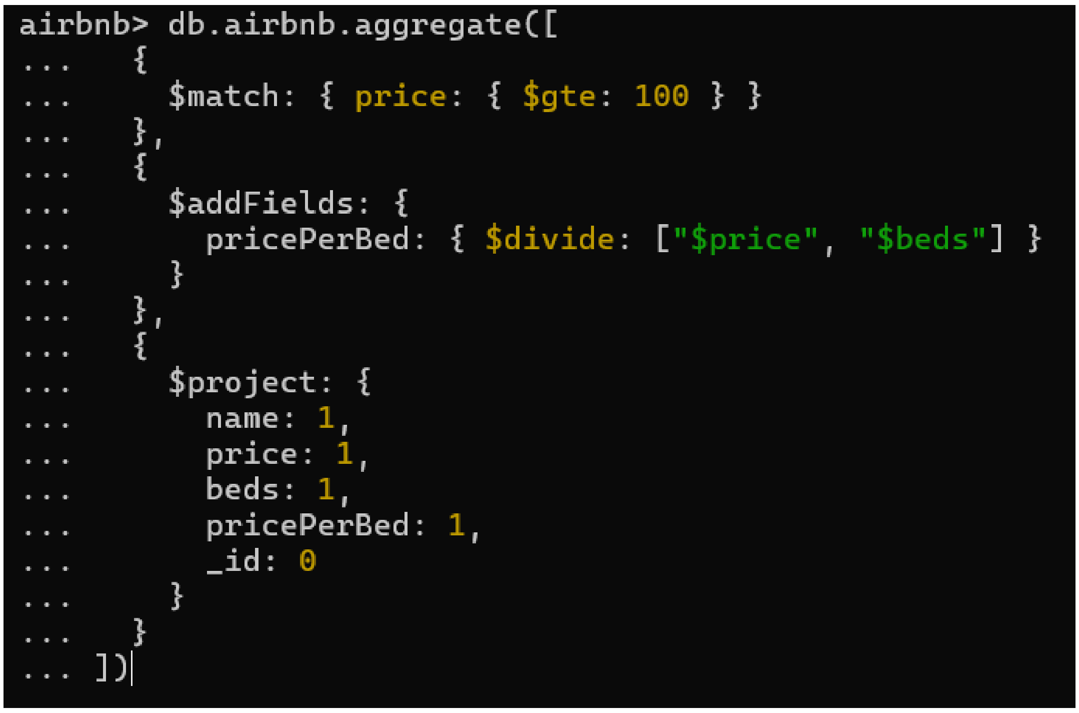
As wee can see, all three types of room satisfied the defined condition (avgRating >80):

- Shared room;
- Entire home/apt;
- Private room.

PART III: Additional Pipeline Stages

Task 1: Create an aggregation pipeline that filters documents by price, adds a new calculated field based on price, and projects both original and calculated fields.

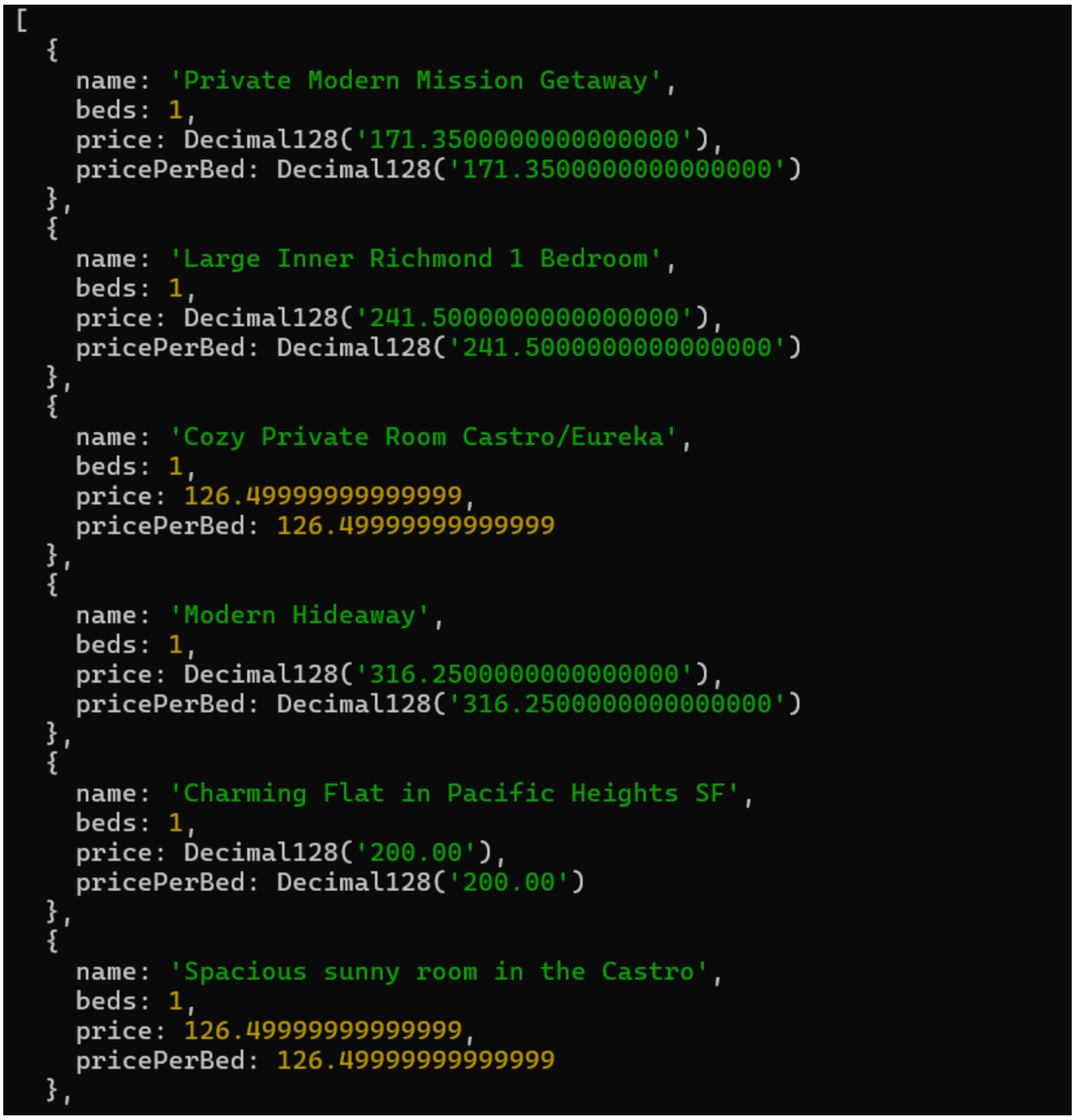
The query code:



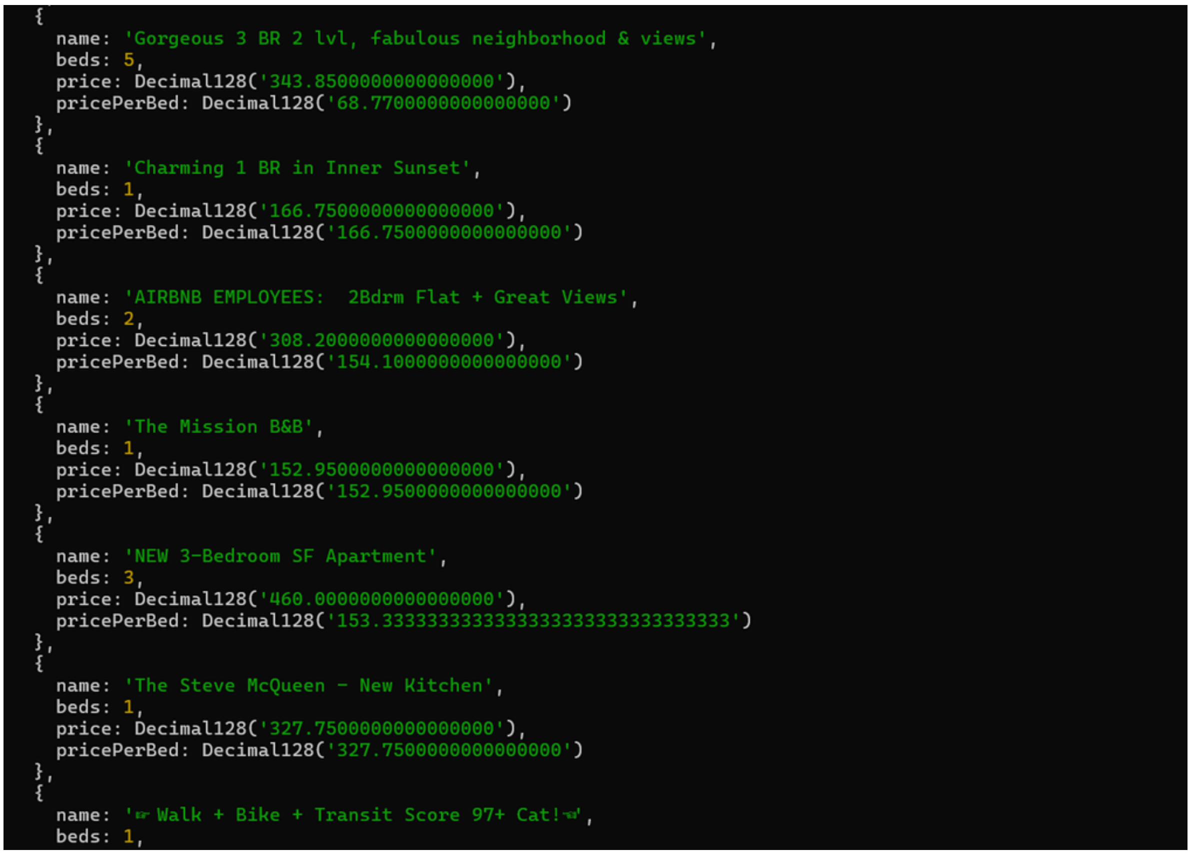
```
airbnb> db.airbnb.aggregate([
...     {
...       $match: { price: { $gte: 100 } }
...     },
...     {
...       $addFields: {
...         pricePerBed: { $divide: ["$price", "$beds"] }
...       }
...     },
...     {
...       $project: {
...         name: 1,
...         price: 1,
...         beds: 1,
...         pricePerBed: 1,
...         _id: 0
...       }
...     }
... ])
```

Except adding **$match** for finding documents by price greater than 100, we use also **$addFields** to add a new calculated field based on price of beds and **$project** to choose **name, price, beds, pricePerBed and _id** fields and calculate them simultaneously.

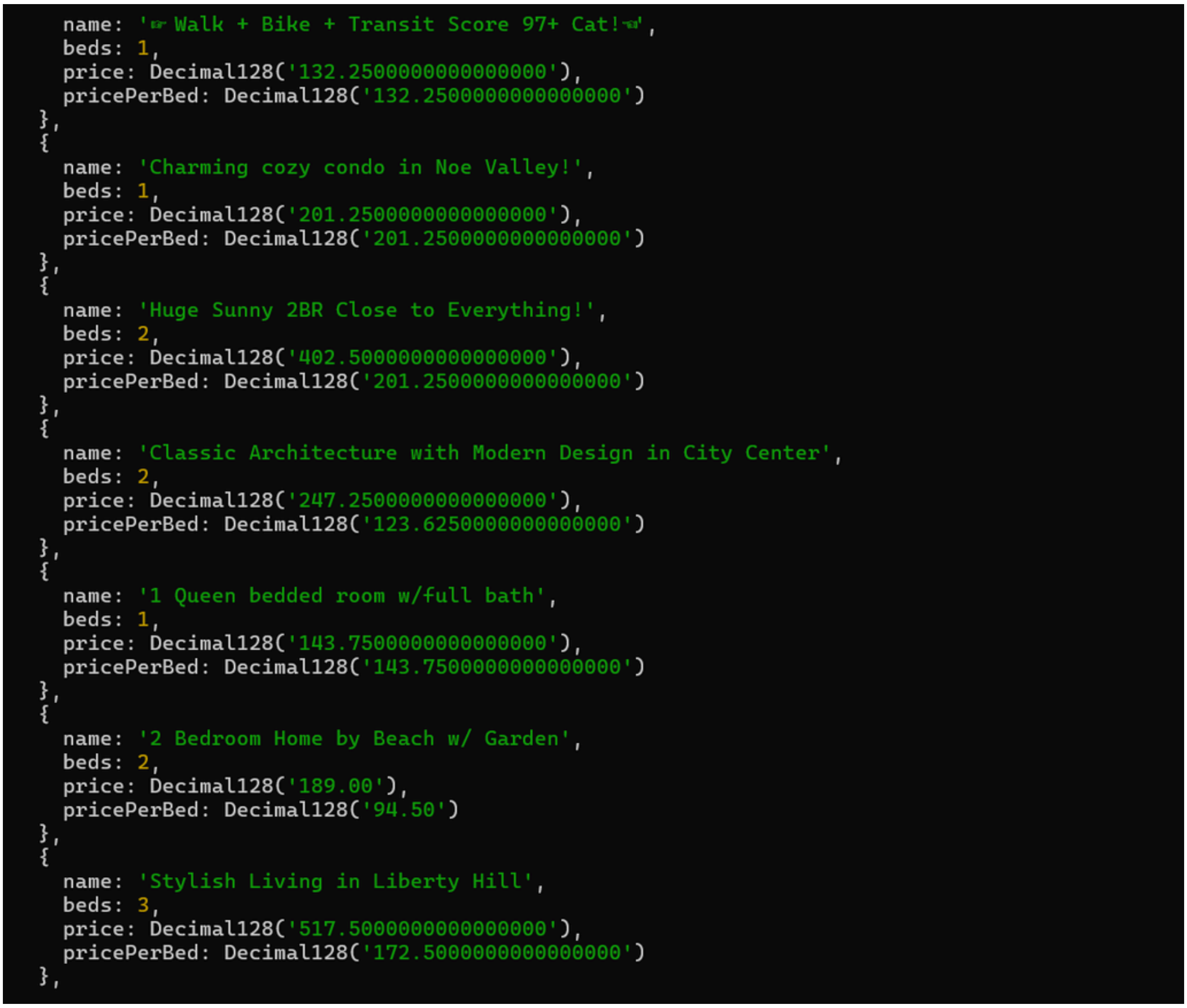
Output:



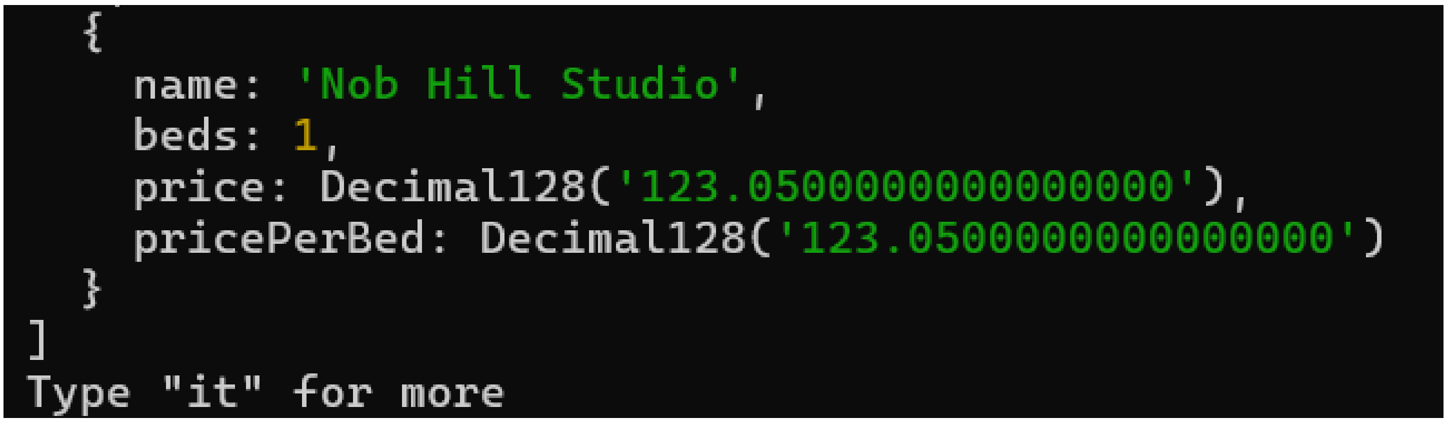
```
[
  {
    name: 'Private Modern Mission Getaway',
    beds: 1,
    price: Decimal128('171.3500000000000000'),
    pricePerBed: Decimal128('171.3500000000000000')
  },
  {
    name: 'Large Inner Richmond 1 Bedroom',
    beds: 1,
    price: Decimal128('241.5000000000000000'),
    pricePerBed: Decimal128('241.5000000000000000')
  },
  {
    name: 'Cozy Private Room Castro/Eureka',
    beds: 1,
    price: 126.49999999999999,
    pricePerBed: 126.49999999999999
  },
  {
    name: 'Modern Hideaway',
    beds: 1,
    price: Decimal128('316.2500000000000000'),
    pricePerBed: Decimal128('316.2500000000000000')
  },
  {
    name: 'Charming Flat in Pacific Heights SF',
    beds: 1,
    price: Decimal128('200.00'),
    pricePerBed: Decimal128('200.00')
  },
  {
    name: 'Spacious sunny room in the Castro',
    beds: 1,
    price: 126.49999999999999,
    pricePerBed: 126.49999999999999
  },
```



```
  {
    name: 'Gorgeous 3 BR 2 lvl, fabulous neighborhood & views',
    beds: 5,
    price: Decimal128('343.8500000000000000'),
    pricePerBed: Decimal128('68.7700000000000000')
  },
  {
    name: 'Charming 1 BR in Inner Sunset',
    beds: 1,
    price: Decimal128('166.7500000000000000'),
    pricePerBed: Decimal128('166.7500000000000000')
  },
  {
    name: 'AIRBNB EMPLOYEES:  2Bdrm Flat + Great Views',
    beds: 2,
    price: Decimal128('308.2000000000000000'),
    pricePerBed: Decimal128('154.1000000000000000')
  },
  {
    name: 'The Mission B&B',
    beds: 1,
    price: Decimal128('152.9500000000000000'),
    pricePerBed: Decimal128('152.9500000000000000')
  },
  {
    name: 'NEW 3-Bedroom SF Apartment',
    beds: 3,
    price: Decimal128('460.0000000000000000'),
    pricePerBed: Decimal128('153.3333333333333333333333333333333')
  },
  {
    name: 'The Steve McQueen - New Kitchen',
    beds: 1,
    price: Decimal128('327.7500000000000000'),
    pricePerBed: Decimal128('327.7500000000000000')
  },
  {
    name: '☞ Walk + Bike + Transit Score 97+ Cat!☜',
    beds: 1,
```



```
    name: '👉 Walk + Bike + Transit Score 97+ Cat!👈',
    beds: 1,
    price: Decimal128('132.2500000000000000'),
    pricePerBed: Decimal128('132.2500000000000000')
  },
  {
    name: 'Charming cozy condo in Noe Valley!',
    beds: 1,
    price: Decimal128('201.2500000000000000'),
    pricePerBed: Decimal128('201.2500000000000000')
  },
  {
    name: 'Huge Sunny 2BR Close to Everything!',
    beds: 2,
    price: Decimal128('402.5000000000000000'),
    pricePerBed: Decimal128('201.2500000000000000')
  },
  {
    name: 'Classic Architecture with Modern Design in City Center',
    beds: 2,
    price: Decimal128('247.2500000000000000'),
    pricePerBed: Decimal128('123.6250000000000000')
  },
  {
    name: '1 Queen bedded room w/full bath',
    beds: 1,
    price: Decimal128('143.7500000000000000'),
    pricePerBed: Decimal128('143.7500000000000000')
  },
  {
    name: '2 Bedroom Home by Beach w/ Garden',
    beds: 2,
    price: Decimal128('189.00'),
    pricePerBed: Decimal128('94.50')
  },
  {
    name: 'Stylish Living in Liberty Hill',
    beds: 3,
    price: Decimal128('517.5000000000000000'),
    pricePerBed: Decimal128('172.5000000000000000')
  },
```

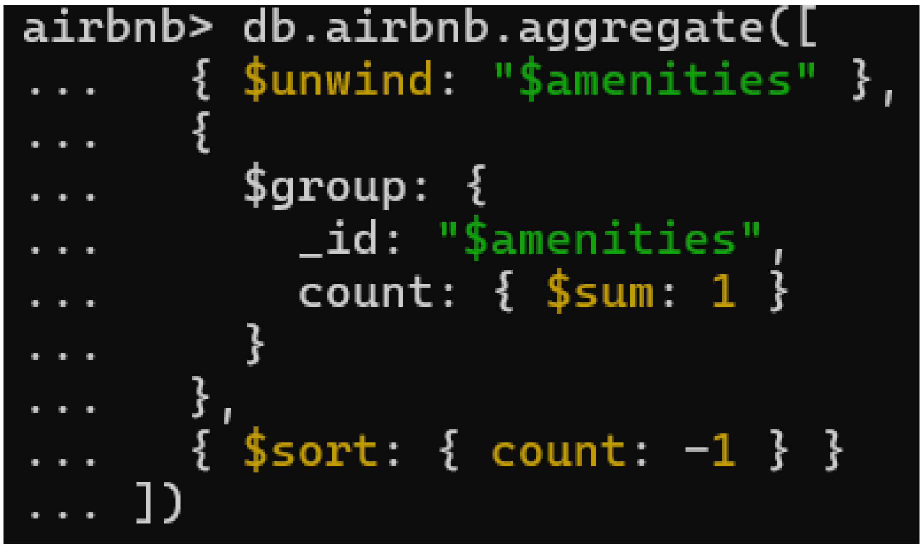


```
  {
    name: 'Nob Hill Studio',
    beds: 1,
    price: Decimal128('123.0500000000000000'),
    pricePerBed: Decimal128('123.0500000000000000')
  }
]
Type "it" for more
```

Filtering by price (> 100) returned relevant listings. Additionally, the new **pricePerBed** field added: **price** and **beds. Decimal128** is the MongoDB standard for exact prices that avoids rounding errors. This calculation helps us to compare listings with different sizes on a fair basis.

Task 2: Design an aggregation pipeline that unwinds the amenities array, groups documents by individual amenity, counts how many times each amenity appears, and sorts the results by frequency.

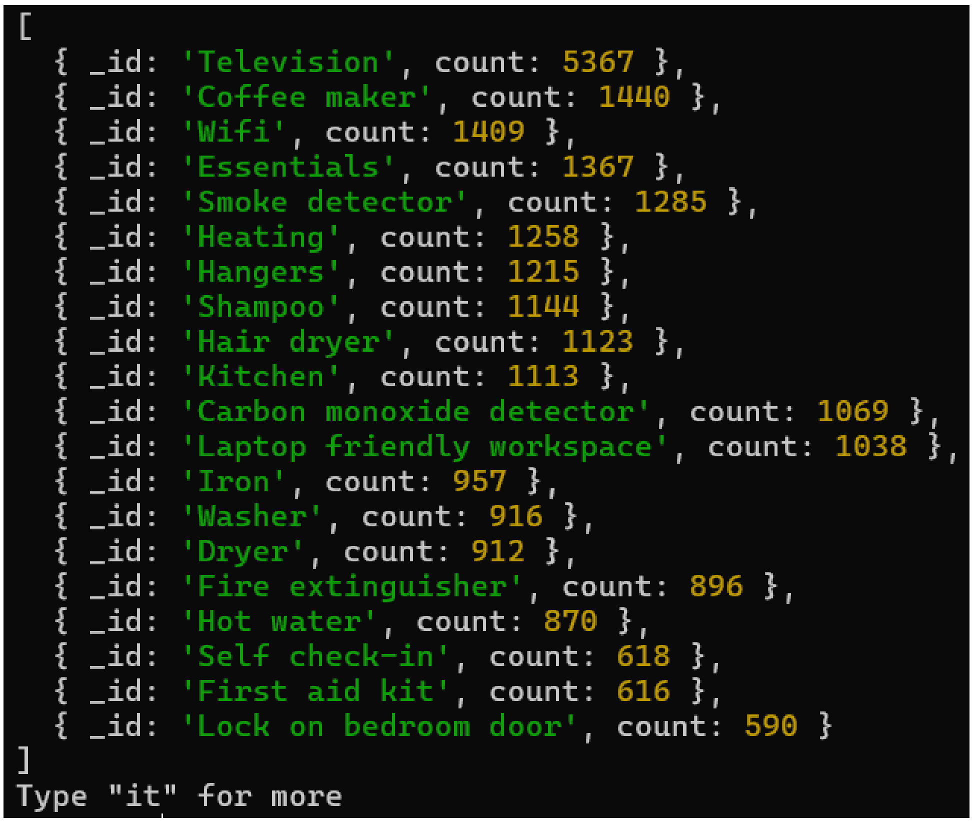
Code:



```
airbnb> db.airbnb.aggregate([
...   { $unwind: "$amenities" },
...   {
...     $group: {
...       _id: "$amenities",
...       count: { $sum: 1 }
...     }
...   },
...   { $sort: { count: -1 } }
... ])
```

We use **$unwind** to deconstruct the **amenities** field, **$group** to group **_id: "$amenities",** and **count,** the **$sum** of which is equal to 1 and **$sort** to sort the count in descending order.

Output:

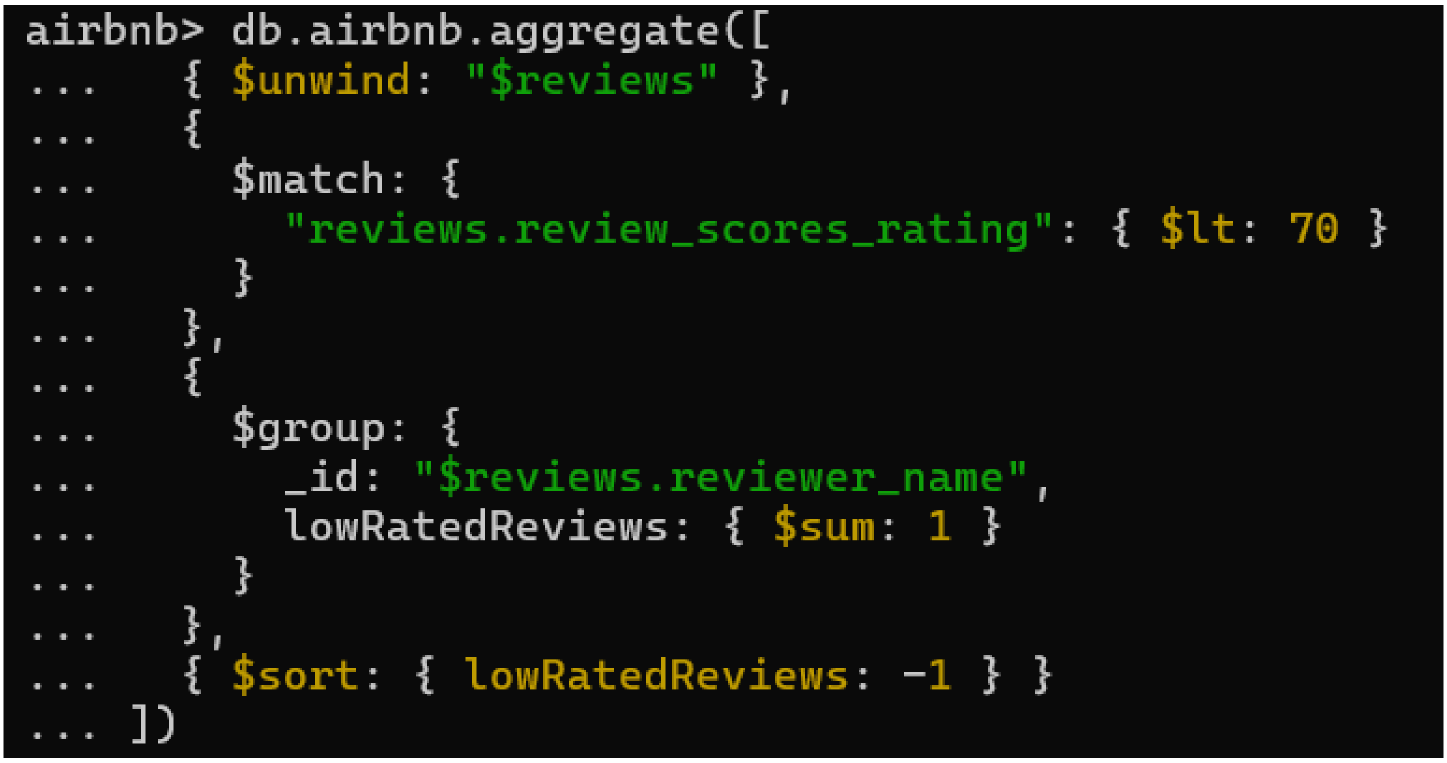


```
[
  { _id: 'Television', count: 5367 },
  { _id: 'Coffee maker', count: 1440 },
  { _id: 'Wifi', count: 1409 },
  { _id: 'Essentials', count: 1367 },
  { _id: 'Smoke detector', count: 1285 },
  { _id: 'Heating', count: 1258 },
  { _id: 'Hangers', count: 1215 },
  { _id: 'Shampoo', count: 1144 },
  { _id: 'Hair dryer', count: 1123 },
  { _id: 'Kitchen', count: 1113 },
  { _id: 'Carbon monoxide detector', count: 1069 },
  { _id: 'Laptop friendly workspace', count: 1038 },
  { _id: 'Iron', count: 957 },
  { _id: 'Washer', count: 916 },
  { _id: 'Dryer', count: 912 },
  { _id: 'Fire extinguisher', count: 896 },
  { _id: 'Hot water', count: 870 },
  { _id: 'Self check-in', count: 618 },
  { _id: 'First aid kit', count: 616 },
  { _id: 'Lock on bedroom door', count: 590 }
]
Type "it" for more
```

The pipeline code unwinds the amenities array and creates a separate for every amenity. Moreover, this query has shown us the counts how many times each amenity appears, and sorts the results by frequency. Additionally, we can see the documents of each name of each amenity with the correspondent price. The result of counts has been sorted in descending order.

Task 3: Build an aggregation pipeline that unwinds the reviews array, filters reviews with rating below a specified threshold, groups the results by reviewer name, and counts how many low-rated reviews each reviewer has.

Code:



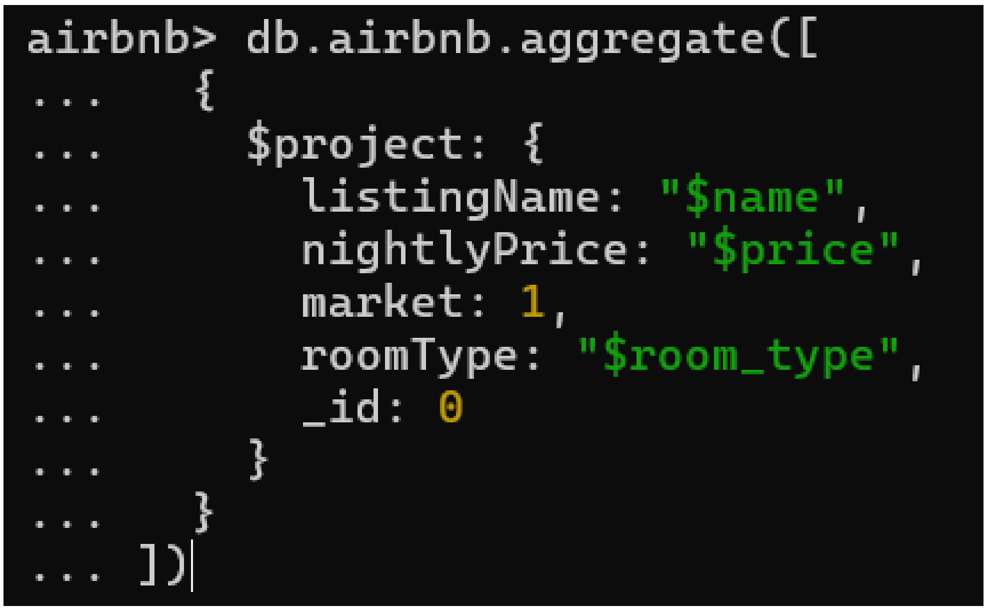
```
airbnb> db.airbnb.aggregate([
...   { $unwind: "$reviews" },
...   {
...     $match: {
...       "reviews.review_scores_rating": { $lt: 70 }
...     }
...   },
...   {
...     $group: {
...       _id: "$reviews.reviewer_name",
...       lowRatedReviews: { $sum: 1 }
...     }
...   },
...   { $sort: { lowRatedReviews: -1 } }
... ])
```

In this query, we **$unwind** to unwind the reviews array. We also use **$match** to choose **"reviews.review_scores_rating"** field with the price is less than 70. Also, we use **$group** to group the results by **"$reviews.reviewer_name"** with the sum of **lowRatedReviews**, which is equal to 1. All results should be sorted in descending order.

As for the output, there's nothing, because the aggregation pipeline has returned no documents. No reviews matched the satisfying condition. Thus, the code has been written correctly, but the result is empty.

Task 4: Create an aggregation pipeline that uses projection to rename fields, exclude unnecessary fields, and preserve only relevant analytical data.

Code:



```
airbnb> db.airbnb.aggregate([
...   {
...     $project: {
...       listingName: "$name",
...       nightlyPrice: "$price",
...       market: 1,
...       roomType: "$room_type",
...       _id: 0
...     }
...   }
... ])
```

We use **$project** command to rename the necessary fields:

listingName: **“$name”,**

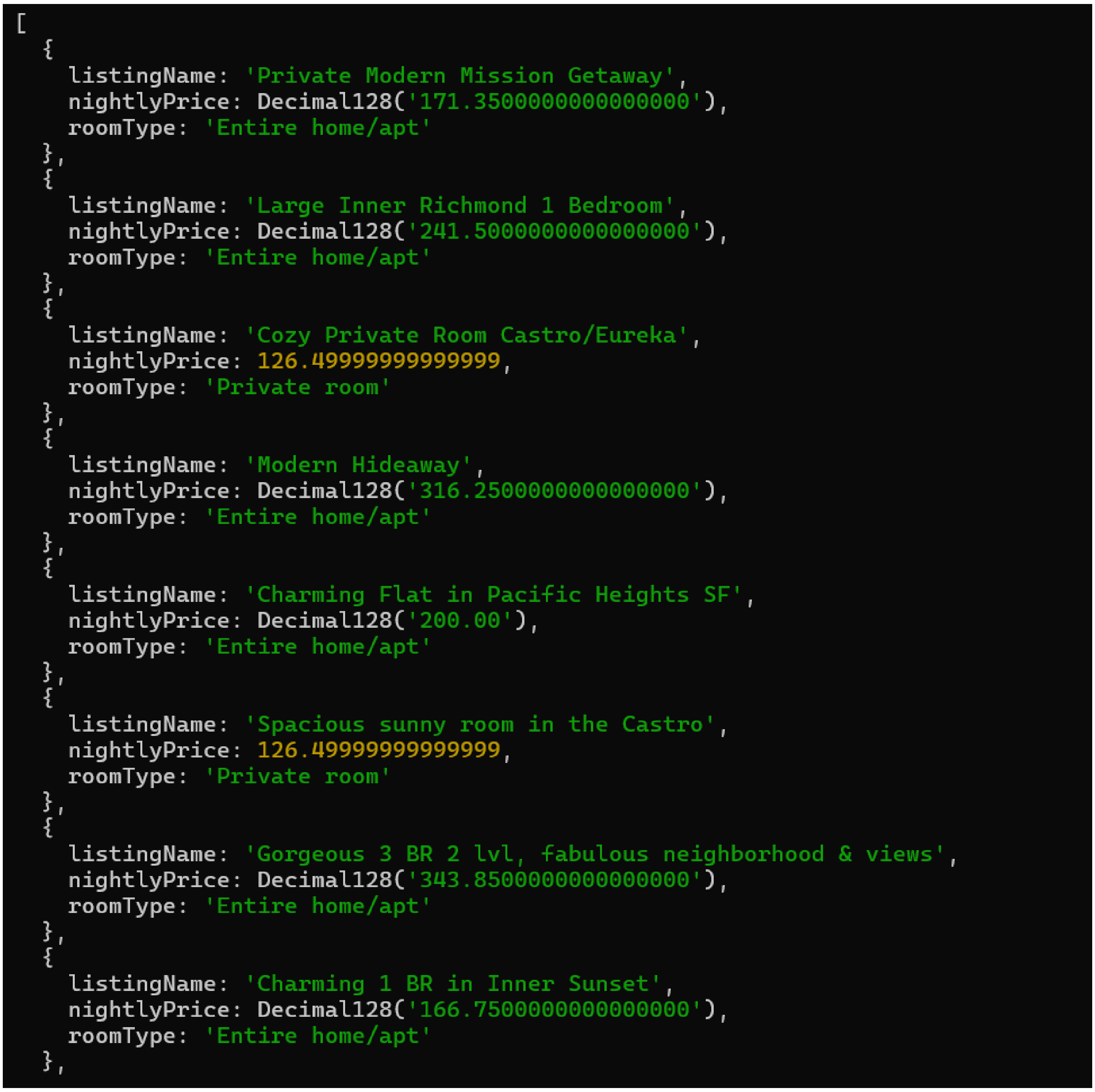
nightlyPrice: **“$price”,**

market: **1,**

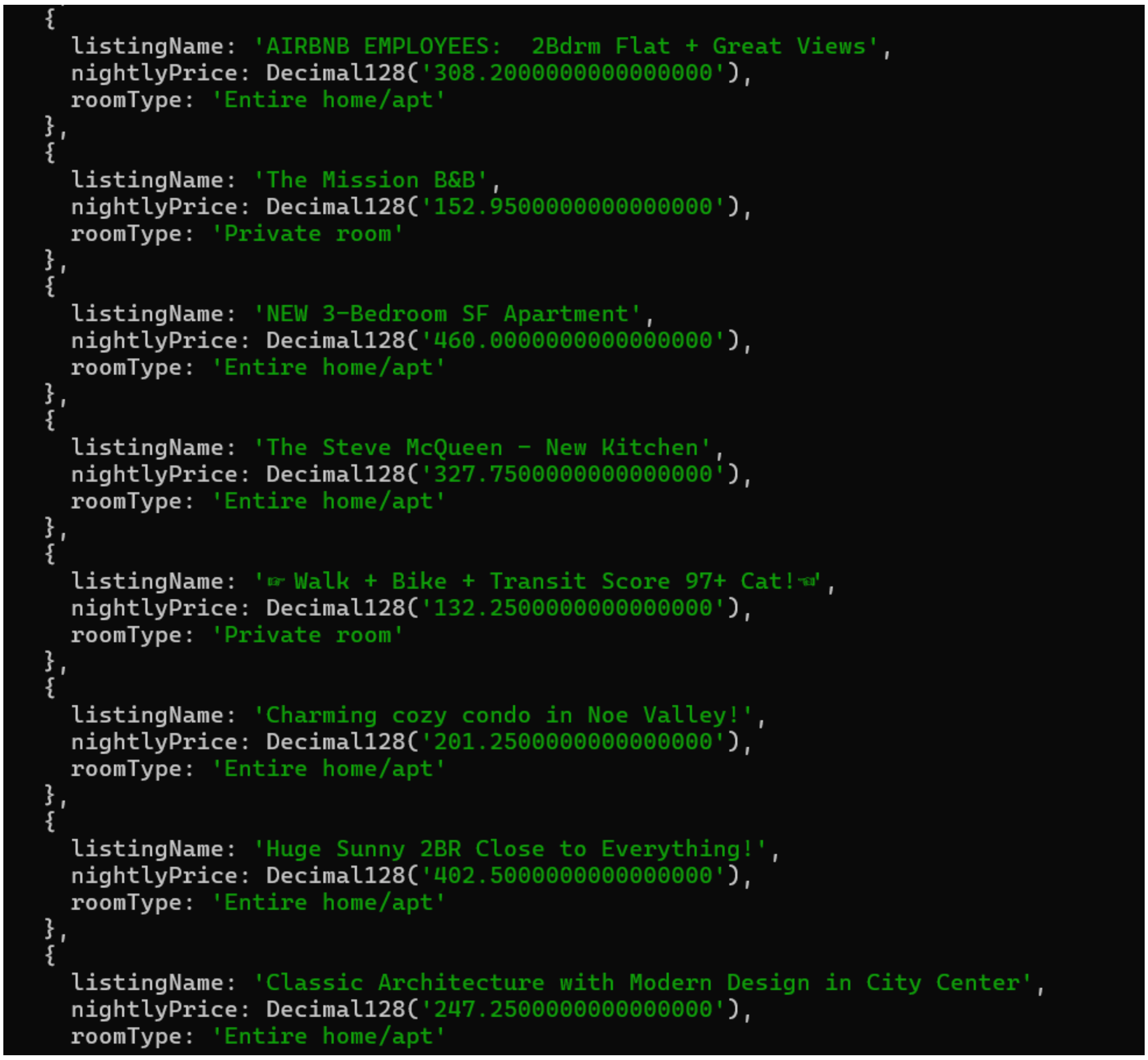
roomType: **“$room_type”,**

__id: 0**.**

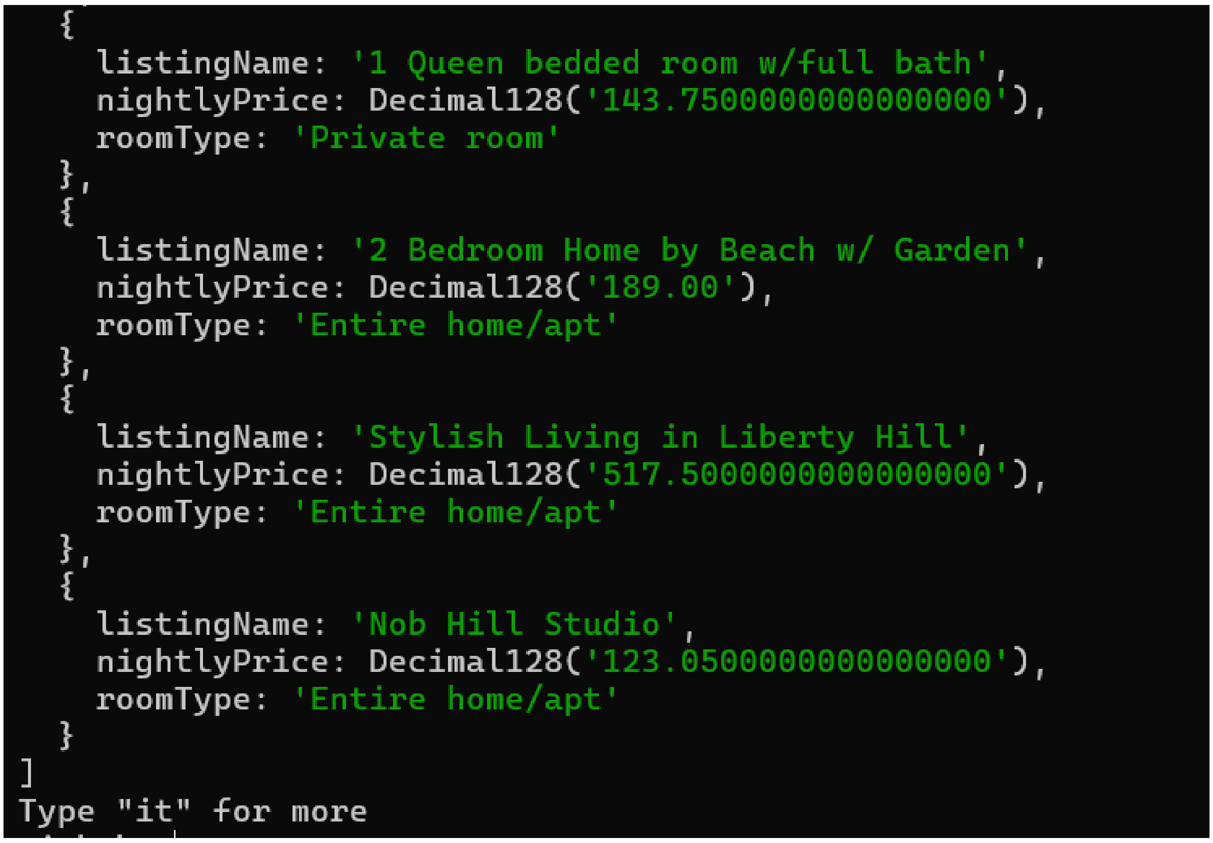
Output:



```
[
  {
    listingName: 'Private Modern Mission Getaway',
    nightlyPrice: Decimal128('171.3500000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Large Inner Richmond 1 Bedroom',
    nightlyPrice: Decimal128('241.5000000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Cozy Private Room Castro/Eureka',
    nightlyPrice: 126.49999999999999,
    roomType: 'Private room'
  },
  {
    listingName: 'Modern Hideaway',
    nightlyPrice: Decimal128('316.2500000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Charming Flat in Pacific Heights SF',
    nightlyPrice: Decimal128('200.00'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Spacious sunny room in the Castro',
    nightlyPrice: 126.49999999999999,
    roomType: 'Private room'
  },
  {
    listingName: 'Gorgeous 3 BR 2 lvl, fabulous neighborhood & views',
    nightlyPrice: Decimal128('343.8500000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Charming 1 BR in Inner Sunset',
    nightlyPrice: Decimal128('166.7500000000000000'),
    roomType: 'Entire home/apt'
  },
```



```
  {
    listingName: 'AIRBNB EMPLOYEES:  2Bdrm Flat + Great Views',
    nightlyPrice: Decimal128('308.2000000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'The Mission B&B',
    nightlyPrice: Decimal128('152.9500000000000000'),
    roomType: 'Private room'
  },
  {
    listingName: 'NEW 3-Bedroom SF Apartment',
    nightlyPrice: Decimal128('460.0000000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'The Steve McQueen - New Kitchen',
    nightlyPrice: Decimal128('327.7500000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: '☞ Walk + Bike + Transit Score 97+ Cat!☜',
    nightlyPrice: Decimal128('132.2500000000000000'),
    roomType: 'Private room'
  },
  {
    listingName: 'Charming cozy condo in Noe Valley!',
    nightlyPrice: Decimal128('201.2500000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Huge Sunny 2BR Close to Everything!',
    nightlyPrice: Decimal128('402.5000000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Classic Architecture with Modern Design in City Center',
    nightlyPrice: Decimal128('247.2500000000000000'),
    roomType: 'Entire home/apt'
```



```
  {
    listingName: '1 Queen bedded room w/full bath',
    nightlyPrice: Decimal128('143.7500000000000000'),
    roomType: 'Private room'
  },
  {
    listingName: '2 Bedroom Home by Beach w/ Garden',
    nightlyPrice: Decimal128('189.00'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Stylish Living in Liberty Hill',
    nightlyPrice: Decimal128('517.5000000000000000'),
    roomType: 'Entire home/apt'
  },
  {
    listingName: 'Nob Hill Studio',
    nightlyPrice: Decimal128('123.0500000000000000'),
    roomType: 'Entire home/apt'
  }
]
Type "it" for more
```

What the output shows:

- **Renaming fields:** name → listingName, price → nightlyPrice

- **Selective projection:** only listingName, nightlyPrice, and roomType are retained

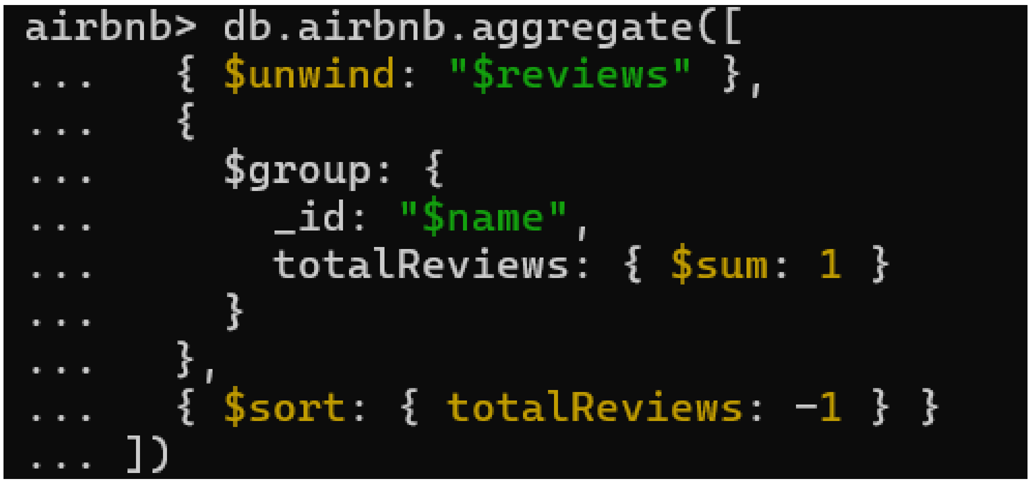
- **Excluding unnecessary fields:** _id: 0 + other fields (host, amenities, reviews, etc.)

- **Analytics:** Focus on key metrics for reporting

This aggregation stage renames chosen fields, removes unnecessary fields, including __id, and keeps only data relevant for analytical purposes.

Task 5: Design an aggregation pipeline that combines match, unwind, group, and sort stages to produce summary statistics related to reviews or availability.

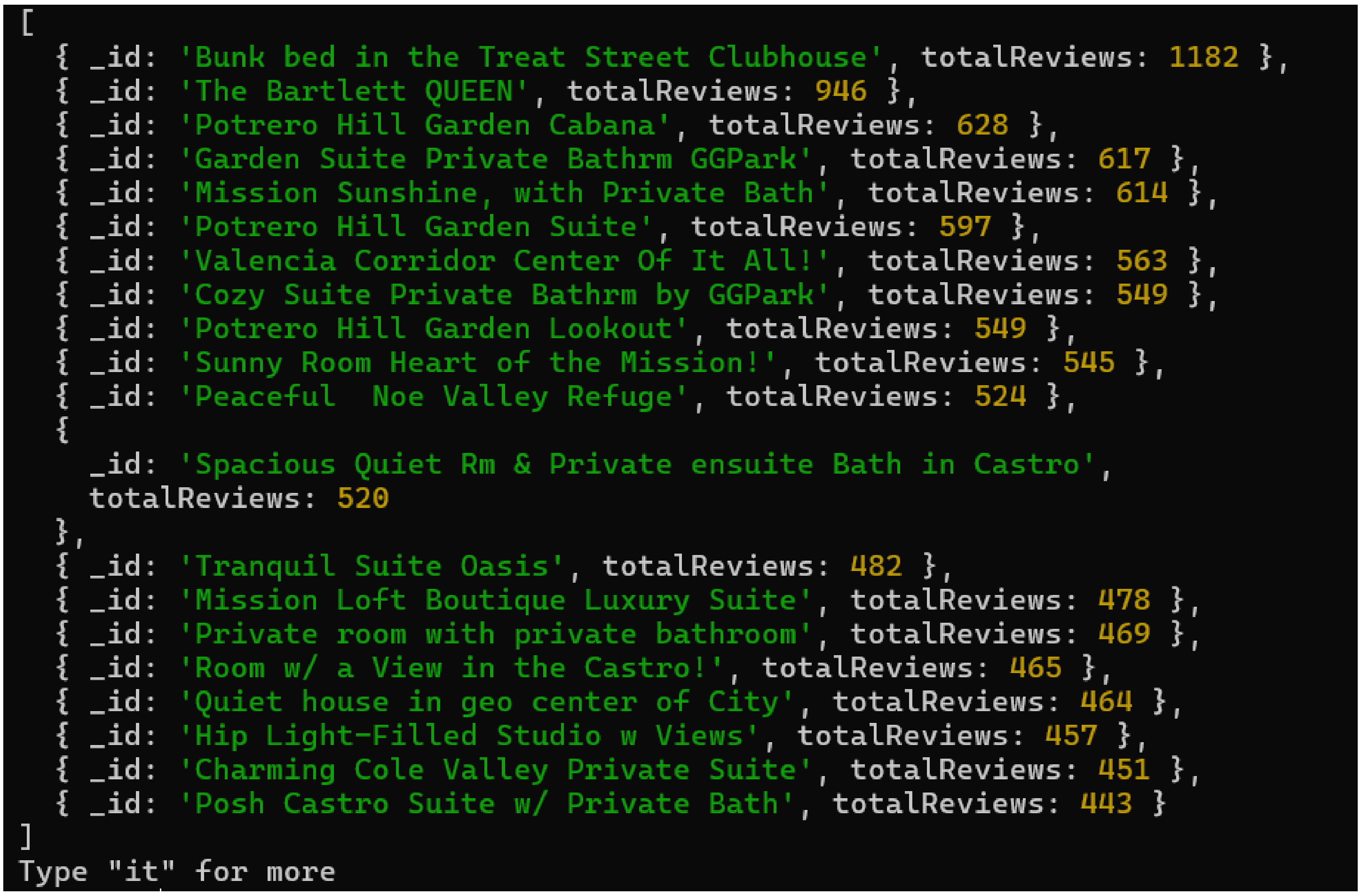
Code:



```
airbnb> db.airbnb.aggregate([
...    { $unwind: "$reviews" },
...    {
...      $group: {
...        _id: "$name",
...        totalReviews: { $sum: 1 }
...      }
...    },
...    { $sort: { totalReviews: -1 } }
... ])
```

In this query, we use **$unwind** to reviews array, **$group** for grouping id by name and **totalReviews** by the sum 1, and **$sort** for sorting **totalReviews** in descending order.

Output:



```
[
  { _id: 'Bunk bed in the Treat Street Clubhouse', totalReviews: 1182 },
  { _id: 'The Bartlett QUEEN', totalReviews: 946 },
  { _id: 'Potrero Hill Garden Cabana', totalReviews: 628 },
  { _id: 'Garden Suite Private Bathrm GGPark', totalReviews: 617 },
  { _id: 'Mission Sunshine, with Private Bath', totalReviews: 614 },
  { _id: 'Potrero Hill Garden Suite', totalReviews: 597 },
  { _id: 'Valencia Corridor Center Of It All!', totalReviews: 563 },
  { _id: 'Cozy Suite Private Bathrm by GGPark', totalReviews: 549 },
  { _id: 'Potrero Hill Garden Lookout', totalReviews: 549 },
  { _id: 'Sunny Room Heart of the Mission!', totalReviews: 545 },
  { _id: 'Peaceful  Noe Valley Refuge', totalReviews: 524 },
  {
    _id: 'Spacious Quiet Rm & Private ensuite Bath in Castro',
    totalReviews: 520
  },
  { _id: 'Tranquil Suite Oasis', totalReviews: 482 },
  { _id: 'Mission Loft Boutique Luxury Suite', totalReviews: 478 },
  { _id: 'Private room with private bathroom', totalReviews: 469 },
  { _id: 'Room w/ a View in the Castro!', totalReviews: 465 },
  { _id: 'Quiet house in geo center of City', totalReviews: 464 },
  { _id: 'Hip Light-Filled Studio w Views', totalReviews: 457 },
  { _id: 'Charming Cole Valley Private Suite', totalReviews: 451 },
  { _id: 'Posh Castro Suite w/ Private Bath', totalReviews: 443 }
]
Type "it" for more
```

This pipeline combines multiple stages, such as match, unwind, group, and sort stages to produce summary statistics related to reviews.

All stages are active:

$match - filtering by reviews

$unwind: "$reviews" - array reversal

$group - totalReviews calculation: { $sum: 1 } (in ascending)

$sort: { totalReviews: -1 } (in descending)

This result helps to identify:

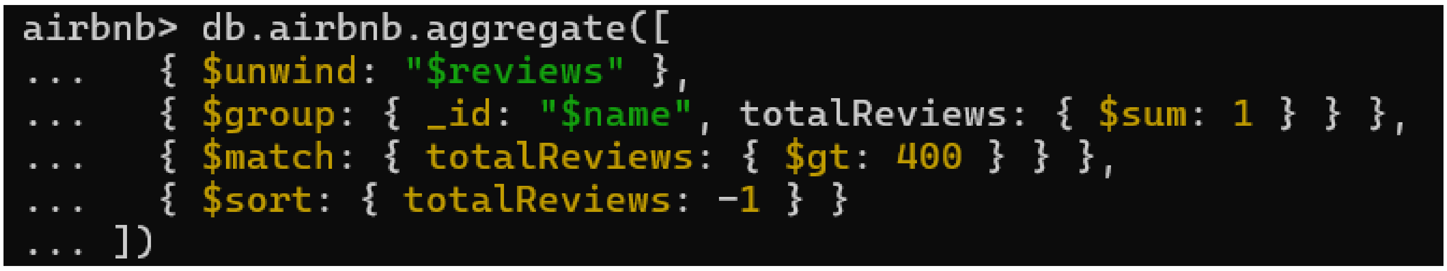
- the most frequently reviewed accommodations;
- listings with high guest engagement
- candidates for deeper analysis of quality

That’s how multiple aggregation stages can be combined to transform new reviews data into ranked analytical insights.

PART IV: Pipeline Optimization

Task 1: Create two aggregation pipelines that produce the same result: one pipeline with filtering stages placed later and another with filtering stages placed earlier. Explain which pipeline is more efficient and why.

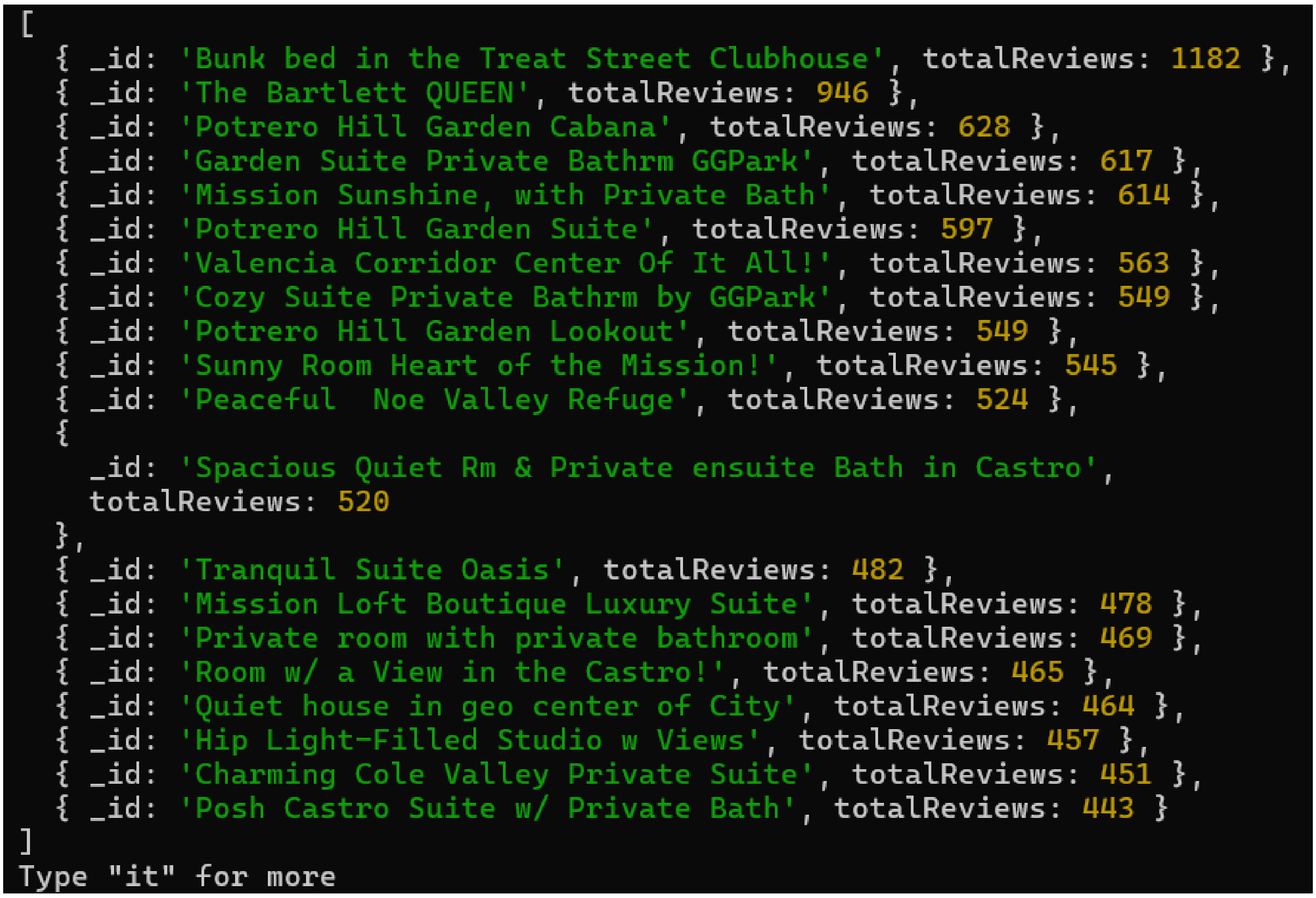
The query of first pipeline (late filter):



```
airbnb> db.airbnb.aggregate([
...   { $unwind: "$reviews" },
...   { $group: { _id: "$name", totalReviews: { $sum: 1 } } },
...   { $match: { totalReviews: { $gt: 400 } } },
...   { $sort: { totalReviews: -1 } }
... ])
```

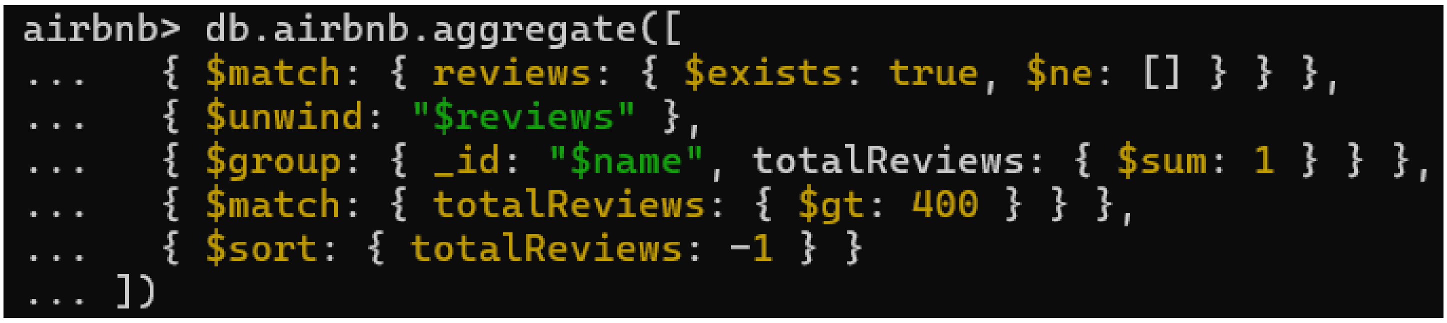
We should unwind the reviews array, then group by id, the sum of totalReviews is 1, then match totalReviews, the quantity of which is greater than 400 and sort the totalReviews field in descending order.

Output:



```
[
  { _id: 'Bunk bed in the Treat Street Clubhouse', totalReviews: 1182 },
  { _id: 'The Bartlett QUEEN', totalReviews: 946 },
  { _id: 'Potrero Hill Garden Cabana', totalReviews: 628 },
  { _id: 'Garden Suite Private Bathrm GGPark', totalReviews: 617 },
  { _id: 'Mission Sunshine, with Private Bath', totalReviews: 614 },
  { _id: 'Potrero Hill Garden Suite', totalReviews: 597 },
  { _id: 'Valencia Corridor Center Of It All!', totalReviews: 563 },
  { _id: 'Cozy Suite Private Bathrm by GGPark', totalReviews: 549 },
  { _id: 'Potrero Hill Garden Lookout', totalReviews: 549 },
  { _id: 'Sunny Room Heart of the Mission!', totalReviews: 545 },
  { _id: 'Peaceful  Noe Valley Refuge', totalReviews: 524 },
  {
    _id: 'Spacious Quiet Rm & Private ensuite Bath in Castro',
    totalReviews: 520
  },
  { _id: 'Tranquil Suite Oasis', totalReviews: 482 },
  { _id: 'Mission Loft Boutique Luxury Suite', totalReviews: 478 },
  { _id: 'Private room with private bathroom', totalReviews: 469 },
  { _id: 'Room w/ a View in the Castro!', totalReviews: 465 },
  { _id: 'Quiet house in geo center of City', totalReviews: 464 },
  { _id: 'Hip Light-Filled Studio w Views', totalReviews: 457 },
  { _id: 'Charming Cole Valley Private Suite', totalReviews: 451 },
  { _id: 'Posh Castro Suite w/ Private Bath', totalReviews: 443 }
]
Type "it" for more
```

The query of first pipeline (early filter):

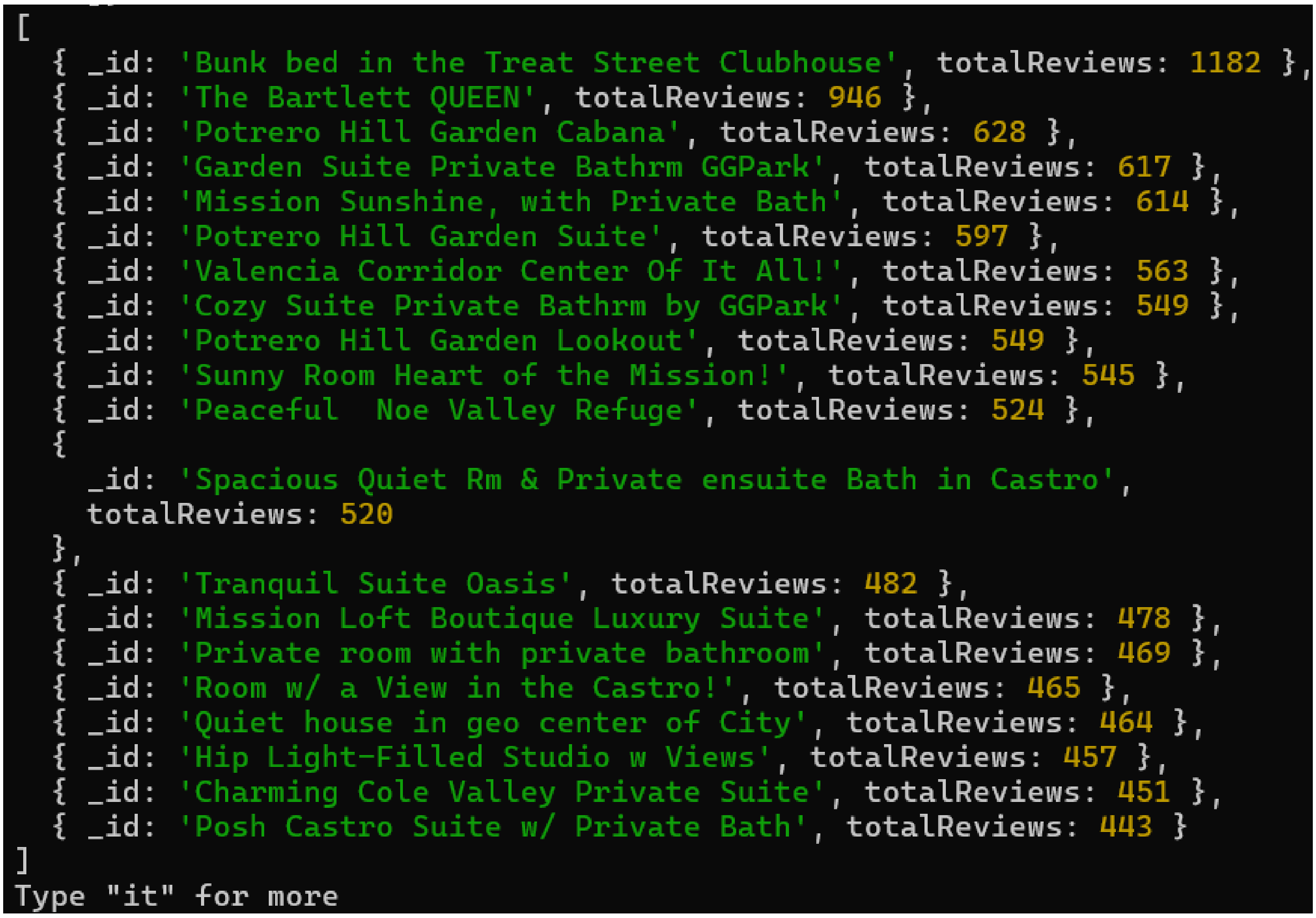


```
airbnb> db.airbnb.aggregate([
...   { $match: { reviews: { $exists: true, $ne: [] } } },
...   { $unwind: "$reviews" },
...   { $group: { _id: "$name", totalReviews: { $sum: 1 } } },
...   { $match: { totalReviews: { $gt: 400 } } },
...   { $sort: { totalReviews: -1 } }
... ])
```

This query is similar, but unlike the previous query, there's the second match command. For example, { **$match: { reviews: { $exists: true, $ne : [] } } }**

That means that this query should match reviews, if this field exists, the query should print true.

Output:



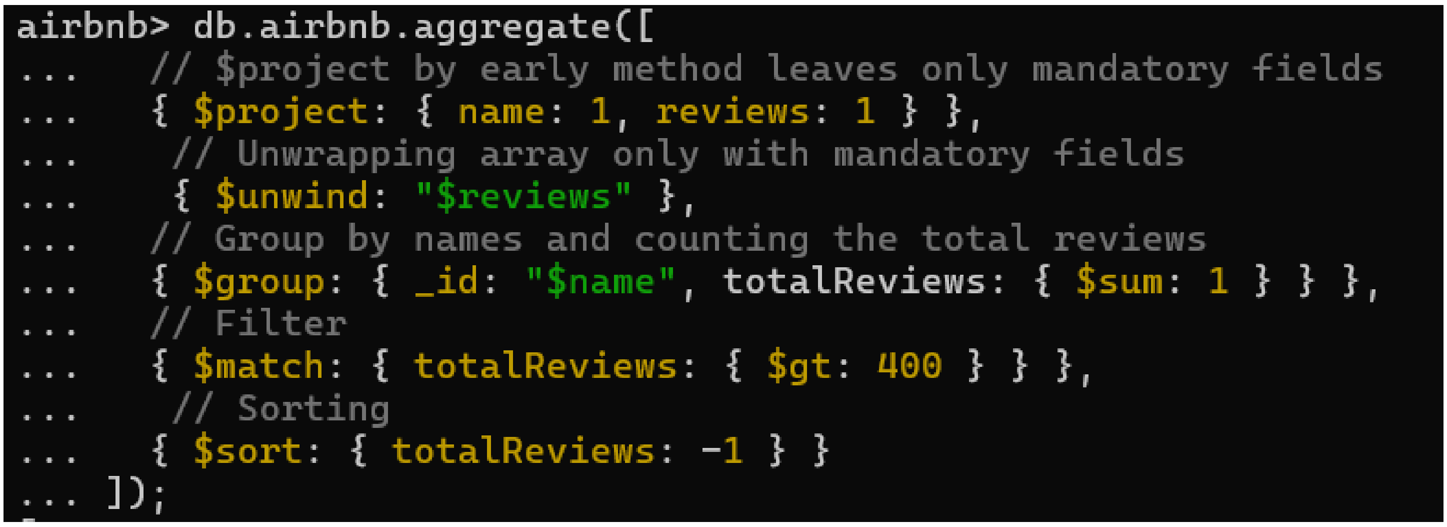
```
[
  { _id: 'Bunk bed in the Treat Street Clubhouse', totalReviews: 1182 },
  { _id: 'The Bartlett QUEEN', totalReviews: 946 },
  { _id: 'Potrero Hill Garden Cabana', totalReviews: 628 },
  { _id: 'Garden Suite Private Bathrm GGPark', totalReviews: 617 },
  { _id: 'Mission Sunshine, with Private Bath', totalReviews: 614 },
  { _id: 'Potrero Hill Garden Suite', totalReviews: 597 },
  { _id: 'Valencia Corridor Center Of It All!', totalReviews: 563 },
  { _id: 'Cozy Suite Private Bathrm by GGPark', totalReviews: 549 },
  { _id: 'Potrero Hill Garden Lookout', totalReviews: 549 },
  { _id: 'Sunny Room Heart of the Mission!', totalReviews: 545 },
  { _id: 'Peaceful  Noe Valley Refuge', totalReviews: 524 },
  {
    _id: 'Spacious Quiet Rm & Private ensuite Bath in Castro',
    totalReviews: 520
  },
  { _id: 'Tranquil Suite Oasis', totalReviews: 482 },
  { _id: 'Mission Loft Boutique Luxury Suite', totalReviews: 478 },
  { _id: 'Private room with private bathroom', totalReviews: 469 },
  { _id: 'Room w/ a View in the Castro!', totalReviews: 465 },
  { _id: 'Quiet house in geo center of City', totalReviews: 464 },
  { _id: 'Hip Light-Filled Studio w Views', totalReviews: 457 },
  { _id: 'Charming Cole Valley Private Suite', totalReviews: 451 },
  { _id: 'Posh Castro Suite w/ Private Bath', totalReviews: 443 }
]
Type "it" for more
```

As we can see, the result in both variant of pipelines is same, even despite 2 matches in the second variant. After using **reviews** array, the quantity of documents has rapidly rosen up. However, the processing took more resources and time, because unwind processed more documents. Also, in both variants, the count of **totalReviews** has shown more than 400 and in descending order.

Filtering data earlier in the pipeline is always preferable and more efficient, because it reduces the volume of documents processed in subsequent stages, especially before **$group** and **$unwind.**

Task 2: Optimize an aggregation pipeline by reducing the number of fields passed between stages and explain the performance benefits of this optimization.

Code:



```
airbnb> db.airbnb.aggregate([
...   // $project by early method leaves only mandatory fields
...   { $project: { name: 1, reviews: 1 } },
...    // Unwrapping array only with mandatory fields
...    { $unwind: "$reviews" },
...   // Group by names and counting the total reviews
...   { $group: { _id: "$name", totalReviews: { $sum: 1 } } },
...   // Filter
...   { $match: { totalReviews: { $gt: 400 } } },
...    // Sorting
...   { $sort: { totalReviews: -1 } }
... ]);
```

In the **$project** command, we choose the necessary fields, such as **name** and **reviews.**

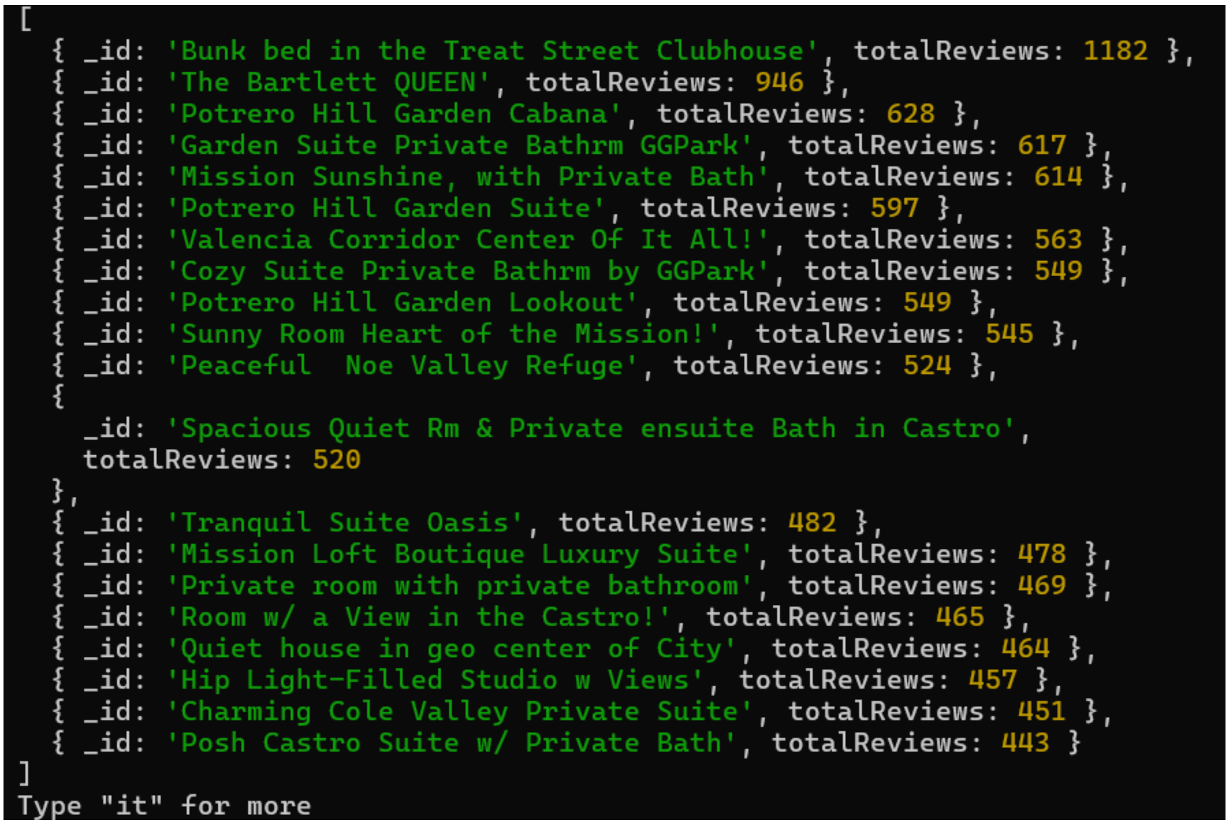
As for the **$unwind** command, we are unwrapping array only with mandatory fields. In our case, this is the **“reviews”** field.

We group the documents by **“$name”** and totalReviews, the sum of which is set to 1.

In the filter, we set the **totalReviews** field is greater than 400.

We sort all of **totalReviews** in descending order.

Output:

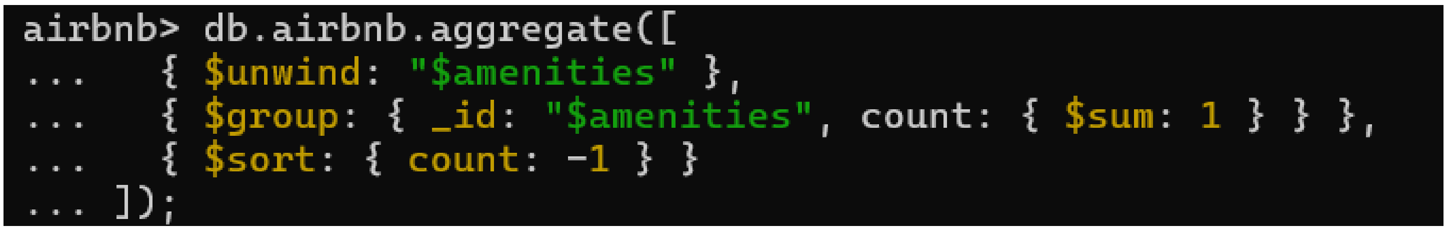


```
[
  { _id: 'Bunk bed in the Treat Street Clubhouse', totalReviews: 1182 },
  { _id: 'The Bartlett QUEEN', totalReviews: 946 },
  { _id: 'Potrero Hill Garden Cabana', totalReviews: 628 },
  { _id: 'Garden Suite Private Bathrm GGPark', totalReviews: 617 },
  { _id: 'Mission Sunshine, with Private Bath', totalReviews: 614 },
  { _id: 'Potrero Hill Garden Suite', totalReviews: 597 },
  { _id: 'Valencia Corridor Center Of It All!', totalReviews: 563 },
  { _id: 'Cozy Suite Private Bathrm by GGPark', totalReviews: 549 },
  { _id: 'Potrero Hill Garden Lookout', totalReviews: 549 },
  { _id: 'Sunny Room Heart of the Mission!', totalReviews: 545 },
  { _id: 'Peaceful  Noe Valley Refuge', totalReviews: 524 },
  {
    _id: 'Spacious Quiet Rm & Private ensuite Bath in Castro',
    totalReviews: 520
  },
  { _id: 'Tranquil Suite Oasis', totalReviews: 482 },
  { _id: 'Mission Loft Boutique Luxury Suite', totalReviews: 478 },
  { _id: 'Private room with private bathroom', totalReviews: 469 },
  { _id: 'Room w/ a View in the Castro!', totalReviews: 465 },
  { _id: 'Quiet house in geo center of City', totalReviews: 464 },
  { _id: 'Hip Light-Filled Studio w Views', totalReviews: 457 },
  { _id: 'Charming Cole Valley Private Suite', totalReviews: 451 },
  { _id: 'Posh Castro Suite w/ Private Bath', totalReviews: 443 }
]
Type "it" for more
```

The output shows us the listings that contain a large number of reviews (in my case, more than 400). After optimizing the aggregation pipeline by projecting only the required fields before the **$unwind** stage, MongoDB processes less fields at every stage, which is capable to improve perfomance while producing the same analytical result. Optimization doesn’t change the result of business , but radically improves performance.

Task 3: Analyze the impact of using the unwind stage on large arrays and explain when its use should be avoided.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $unwind: "$amenities" },
...   { $group: { _id: "$amenities", count: { $sum: 1 } } },
...   { $sort: { count: -1 } }
... ]);
```

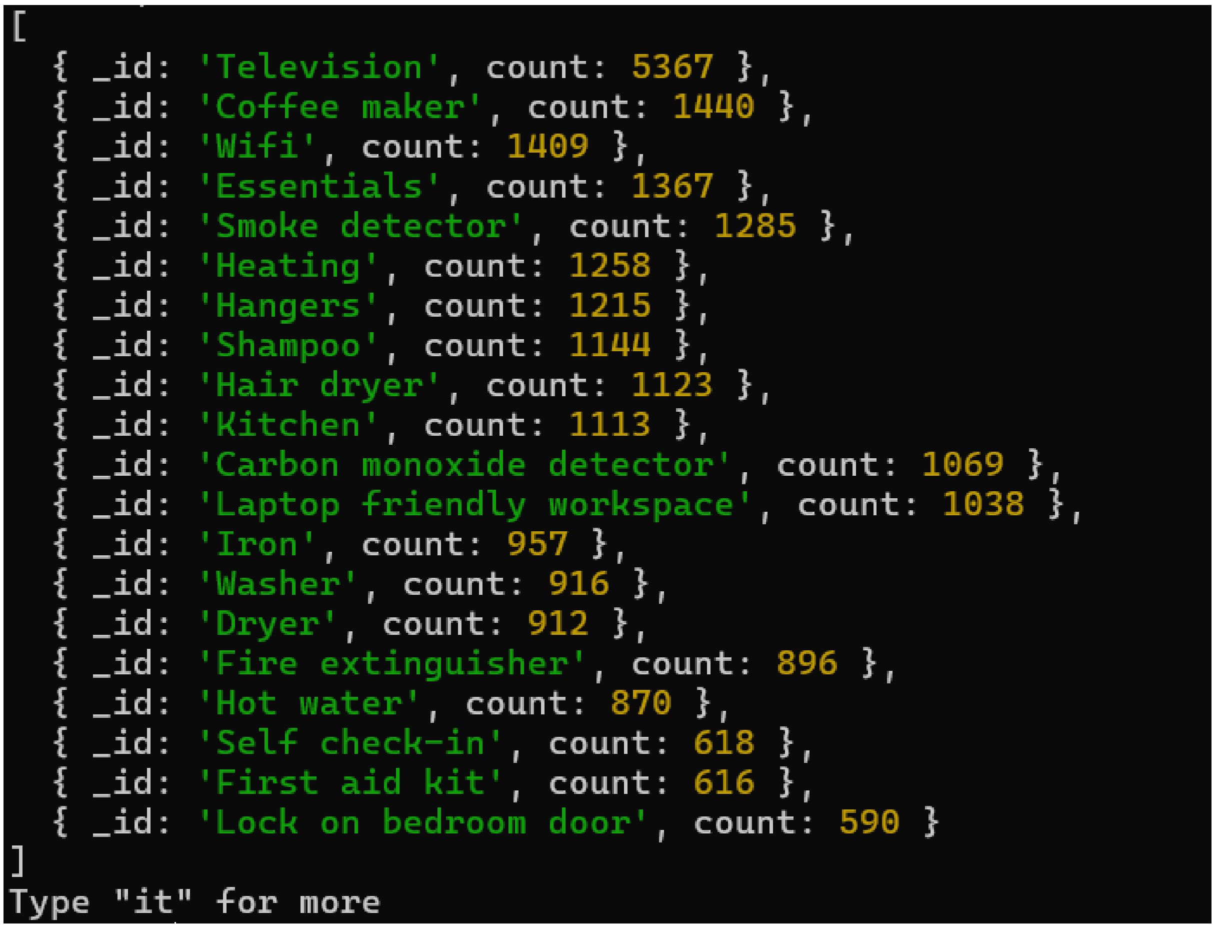
In this query, we use:

$unwind to deconstruct the **amenities** field.

$group to group by **amenities** and the count of all **amenities** field is set to 1.

$sort to sort the count in descending order (-1).

Output:

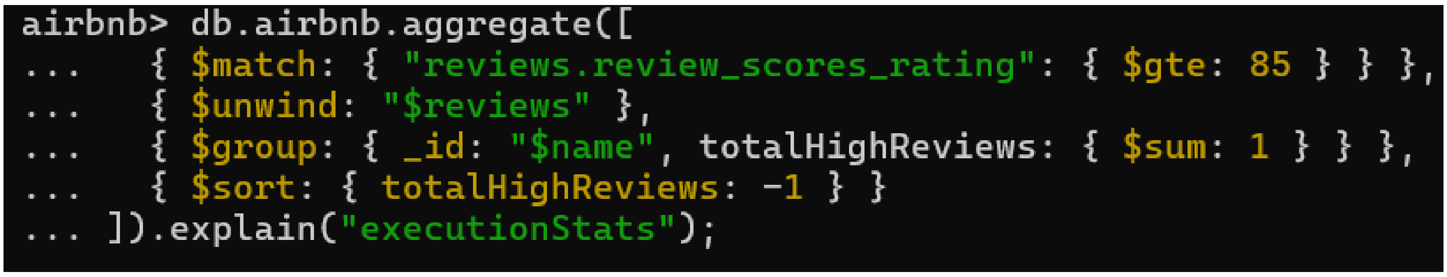


```
[
  { _id: 'Television', count: 5367 },
  { _id: 'Coffee maker', count: 1440 },
  { _id: 'Wifi', count: 1409 },
  { _id: 'Essentials', count: 1367 },
  { _id: 'Smoke detector', count: 1285 },
  { _id: 'Heating', count: 1258 },
  { _id: 'Hangers', count: 1215 },
  { _id: 'Shampoo', count: 1144 },
  { _id: 'Hair dryer', count: 1123 },
  { _id: 'Kitchen', count: 1113 },
  { _id: 'Carbon monoxide detector', count: 1069 },
  { _id: 'Laptop friendly workspace', count: 1038 },
  { _id: 'Iron', count: 957 },
  { _id: 'Washer', count: 916 },
  { _id: 'Dryer', count: 912 },
  { _id: 'Fire extinguisher', count: 896 },
  { _id: 'Hot water', count: 870 },
  { _id: 'Self check-in', count: 618 },
  { _id: 'First aid kit', count: 616 },
  { _id: 'Lock on bedroom door', count: 590 }
]
Type "it" for more
```

According to the screenshot of the result of this query, we can see the frequency of every amenity across all listings. After using the **$unwind** command to deconstruct the **amenities** array, the pipeline also groups documents by individual amenity and counts how many times each one appears on the dataset. The count of listings is sorted in descending order (max:5367, min: 590).

Task 4: Use aggregation explain functionality to analyze pipeline execution, identify potential performance bottlenecks, and describe the observed behavior.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $match: { "reviews.review_scores_rating": { $gte: 85 } } },
...   { $unwind: "$reviews" },
...   { $group: { _id: "$name", totalHighReviews: { $sum: 1 } } },
...   { $sort: { totalHighReviews: -1 } }
... ]).explain("executionStats");
```

According to this query, we use:

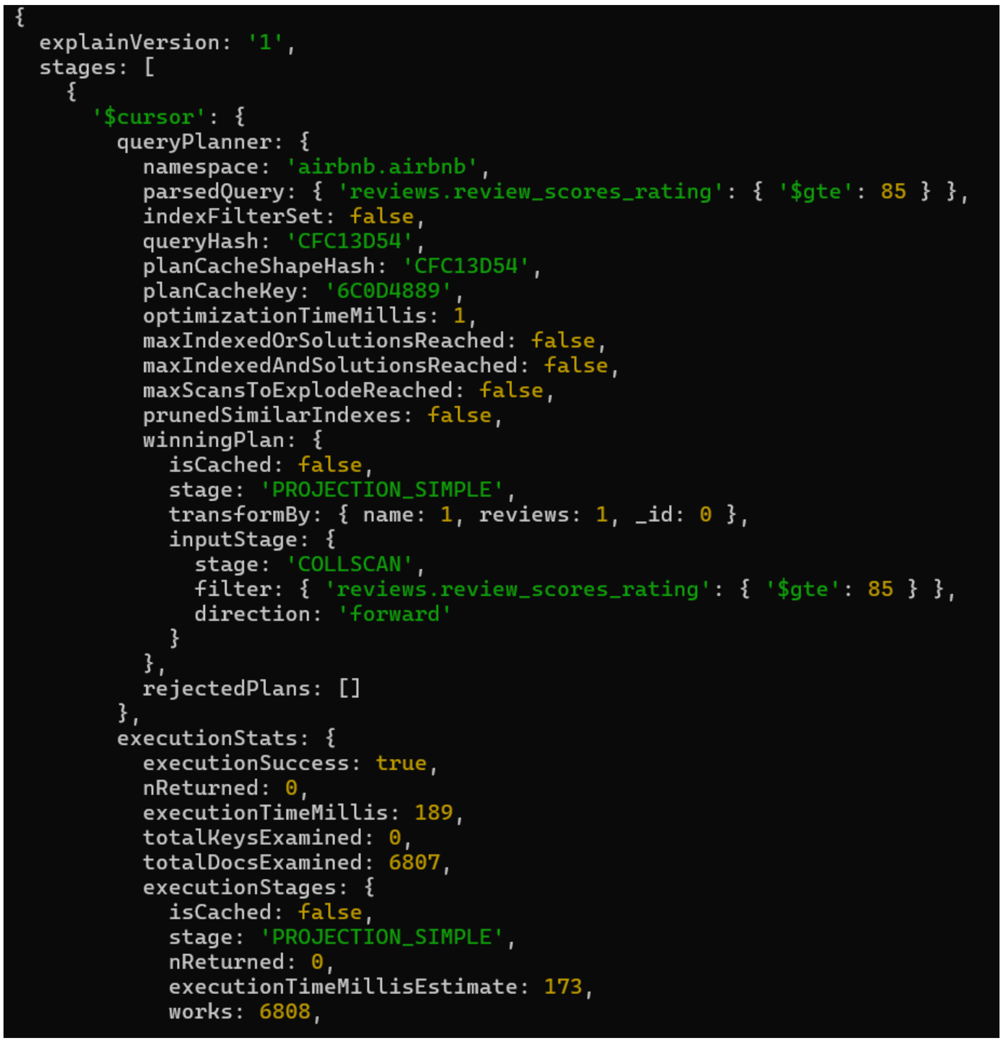
$match to find documents, where the **reviews.review_scores_rating** field should satisfy this condition: the quantity of review scores rating is greater than or equal to 85.

$unwind to deconstruct the **reviews** field.

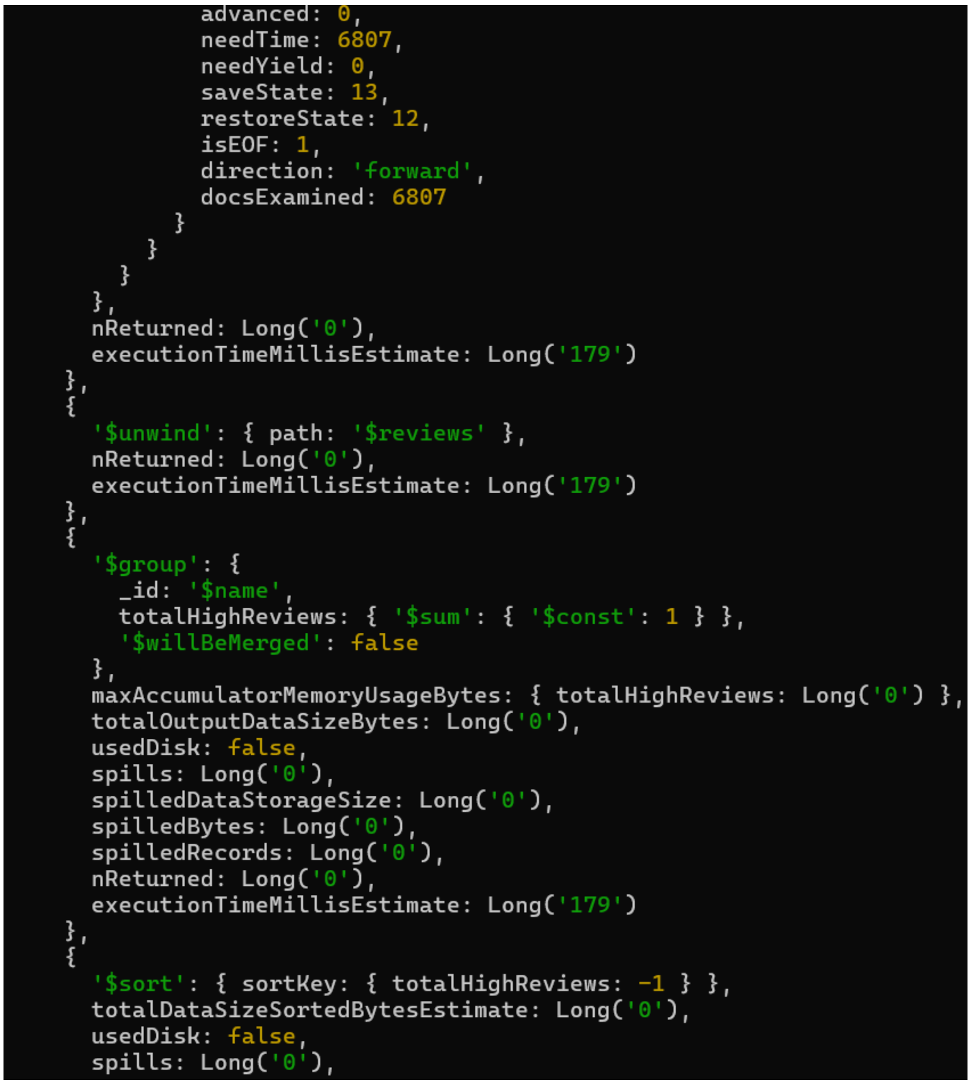
$group by name, the sum of **totalHighReviews** is set to 1.

$sort to sort the **totalHighReviews** field in descending order.

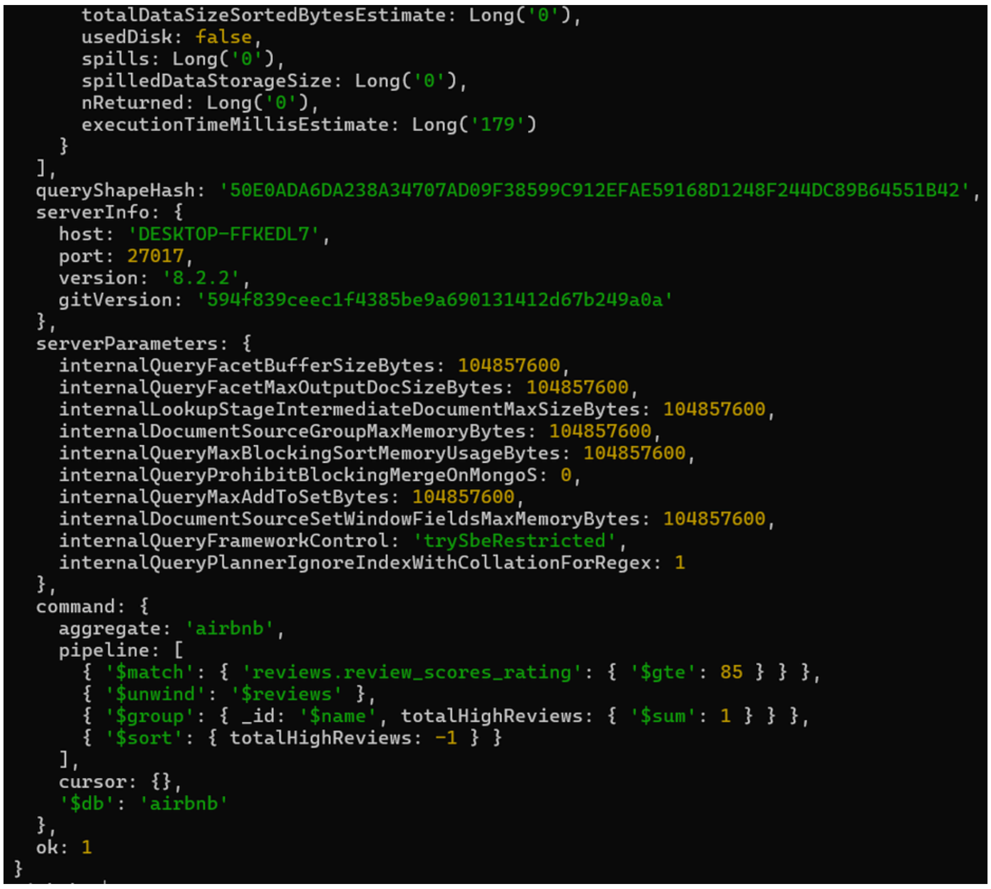
Output:



```
{
  explainVersion: '1',
  stages: [
    {
      '$cursor': {
        queryPlanner: {
          namespace: 'airbnb.airbnb',
          parsedQuery: { 'reviews.review_scores_rating': { '$gte': 85 } },
          indexFilterSet: false,
          queryHash: 'CFC13D54',
          planCacheShapeHash: 'CFC13D54',
          planCacheKey: '6C0D4889',
          optimizationTimeMillis: 1,
          maxIndexedOrSolutionsReached: false,
          maxIndexedAndSolutionsReached: false,
          maxScansToExplodeReached: false,
          prunedSimilarIndexes: false,
          winningPlan: {
            isCached: false,
            stage: 'PROJECTION_SIMPLE',
            transformBy: { name: 1, reviews: 1, _id: 0 },
            inputStage: {
              stage: 'COLLSCAN',
              filter: { 'reviews.review_scores_rating': { '$gte': 85 } },
              direction: 'forward'
            }
          },
          rejectedPlans: []
        },
        executionStats: {
          executionSuccess: true,
          nReturned: 0,
          executionTimeMillis: 189,
          totalKeysExamined: 0,
          totalDocsExamined: 6807,
          executionStages: {
            isCached: false,
            stage: 'PROJECTION_SIMPLE',
            nReturned: 0,
            executionTimeMillisEstimate: 173,
            works: 6808,
```



```
              advanced: 0,
              needTime: 6807,
              needYield: 0,
              saveState: 13,
              restoreState: 12,
              isEOF: 1,
              direction: 'forward',
              docsExamined: 6807
            }
          }
        }
      },
      nReturned: Long('0'),
      executionTimeMillisEstimate: Long('179')
    },
    {
      '$unwind': { path: '$reviews' },
      nReturned: Long('0'),
      executionTimeMillisEstimate: Long('179')
    },
    {
      '$group': {
        _id: '$name',
        totalHighReviews: { '$sum': { '$const': 1 } },
        '$willBeMerged': false
      },
      maxAccumulatorMemoryUsageBytes: { totalHighReviews: Long('0') },
      totalOutputDataSizeBytes: Long('0'),
      usedDisk: false,
      spills: Long('0'),
      spilledDataStorageSize: Long('0'),
      spilledBytes: Long('0'),
      spilledRecords: Long('0'),
      nReturned: Long('0'),
      executionTimeMillisEstimate: Long('179')
    },
    {
      '$sort': { sortKey: { totalHighReviews: -1 } },
      totalDataSizeSortedBytesEstimate: Long('0'),
      usedDisk: false,
      spills: Long('0'),
```



```
      totalDataSizeSortedBytesEstimate: Long('0'),
      usedDisk: false,
      spills: Long('0'),
      spilledDataStorageSize: Long('0'),
      nReturned: Long('0'),
      executionTimeMillisEstimate: Long('179')
    }
  ],
  queryShapeHash: '50E0ADA6DA238A34707AD09F38599C912EFAE59168D1248F244DC89B64551B42',
  serverInfo: {
    host: 'DESKTOP-FFKEDL7',
    port: 27017,
    version: '8.2.2',
    gitVersion: '594f839ceec1f4385be9a690131412d67b249a0a'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600,
    internalQueryFrameworkControl: 'trySbeRestricted',
    internalQueryPlannerIgnoreIndexWithCollationForRegex: 1
  },
  command: {
    aggregate: 'airbnb',
    pipeline: [
      { '$match': { 'reviews.review_scores_rating': { '$gte': 85 } } },
      { '$unwind': '$reviews' },
      { '$group': { _id: '$name', totalHighReviews: { '$sum': 1 } } },
      { '$sort': { totalHighReviews: -1 } }
    ],
    cursor: {},
    '$db': 'airbnb'
  },
  ok: 1
}
```

The output result showed us how MongoDB executes the aggregation pipeline and highlights potential performance bottlenecks.

As usually, in the **db.airbnb.aggregate(),** the pipeline begins with the **$match** stage that filters documents with the given condition: **reviews.review_scores_rating >= 85.**

However, there's no one index on this field, so MongoDB performs a COLLSCAN, examining 6807 documents, whereas the **totalKeysExamined** field shows the result with 0, which indicates a full connection scan.

As for the **$cursor** stage, it dominates the execution time in ms. In my case, it showed 189 ms, which makes it the primary perfomance bottleneck.

Subsequent stages, such as **$unwind, $group & $sort** don't practically impact the performance in this situation, because there's no one document that passes the elementary **$match** page.

That's how MongoDB identifes potential performance bottlenecks.

Task 5: Explain best practices for designing efficient aggregation pipelines in MongoDB.

1) Early filtering

The stages **$match** and **$sort** should be located at the beginning of the pipeline.

This method reduces the quantity of documents passed to the subsequent stages. If **$match** comes first, MongoDB can use indexes for retrieving documents, which is 50 times faster than a full collection scan.

Sample: First of all, we filter out inactive listing from Airbnb database, then calculate the average price.

2) Early projection

Always use the **$project** command to exclude unnecessary fields.

Every field of document takes up RAM. Passing large arrays through the entire pipeline can cause the 100 MB memory limit to be exceeded for a single stage.

3) Using $limit for debugging and perfomance

If you only need the top results, use **$limit** immediately after **$sort.**

MongoDB uses an internal optimization where **$sort** and **$limit** are used consecutively, not sorting all the data in memory but only maintaining the top N results.

4) Taking advantage of indexes

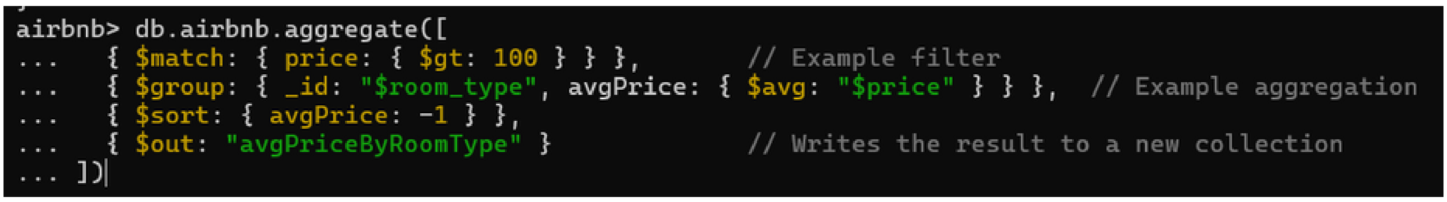
Aggregation is only efficient, when it is indexed.

For example, if the first stage is **$group,** indexes aren't used. Always try to start with **$match** on an indexed field (e.g., price).

PART V: Output Aggregation Results to Collections

Task 1: Create an aggregation pipeline that produces aggregated analytics data and writes the output to a new collection.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $match: { price: { $gt: 100 } } },        // Example filter
...   { $group: { _id: "$room_type", avgPrice: { $avg: "$price" } } },  // Example aggregation
...   { $sort: { avgPrice: -1 } },
...   { $out: "avgPriceByRoomType" }              // Writes the result to a new collection
... ])
```

As an example, we want to check the **$match** stage with the price > 100.

Then, we use **$group** to group by **"$room_type"** and calculate the average price.

Then, we must sort average price in descending order.

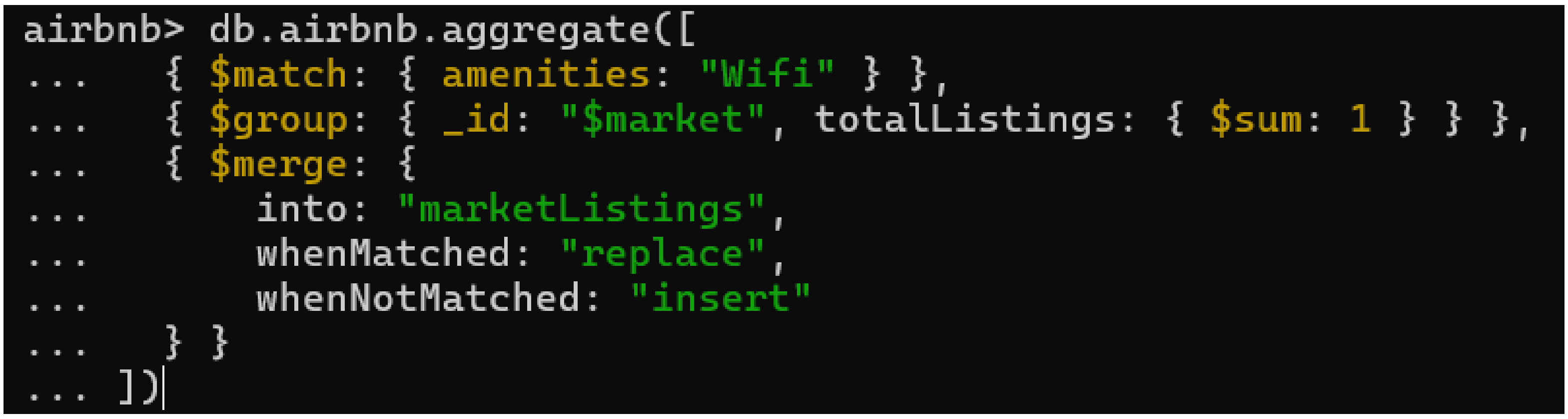
In conclusion, we should use the final stage **$out** to show the average price by room type and to write the result into a new collection.

The output result is empty. That's because no one document matched the filter criteria. All stages executed without issues, but the resulting collection would be empty, if the pipeline was written to a new collection with the **$out** command.

Thus, the query has been run correctly, but with the empty result.

Task 2: Build an aggregation pipeline that aggregates data and updates or replaces data in an existing collection.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $match: { amenities: "Wifi" } },
...   { $group: { _id: "$market", totalListings: { $sum: 1 } } },
...   { $merge: {
...       into: "marketListings",
...       whenMatched: "replace",
...       whenNotMatched: "insert"
...   } }
... ])
```

For example, we'd like to find among other amenities → **"Wifi".**

We should group by **"$market"** and the sum is also equal to 1.

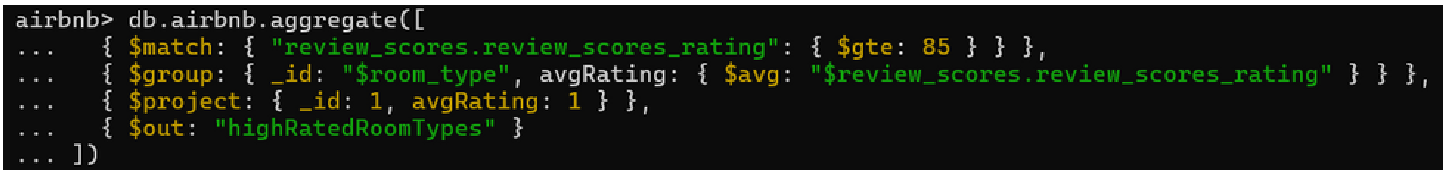
We also use **$merge** command to record results into **marketListings** with the given conditions:

- when matched → replace;
- when not matched → insert.

Output: Like the previous task: 5.1, there's the same situation. Output is empty. That's because of absence of documents. All stages have run without any issues, though the collection would be empty. The pipeline itself is correct; the empty output result simply reflects the current absence of documents in the source collection that correspond the criteria.

Task 3: Design an aggregation pipeline that generates reporting-ready summary data and saves the results to a collection for later querying.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $match: { "review_scores.review_scores_rating": { $gte: 85 } } },
...   { $group: { _id: "$room_type", avgRating: { $avg: "$review_scores.review_scores_rating" } } },
...   { $project: { _id: 1, avgRating: 1 } },
...   { $out: "highRatedRoomTypes" }
... ])
```

In the **$match** command we set the condition: the **"review_scores.review_scores_rating"** should be greater than 85.

As for the **$group** stage, we group by **"$room_type"** and calculate the average rating of **"review_scores.review_scores_rating".**

In the **$project** stage, we choose only **id** and **avgRating.**

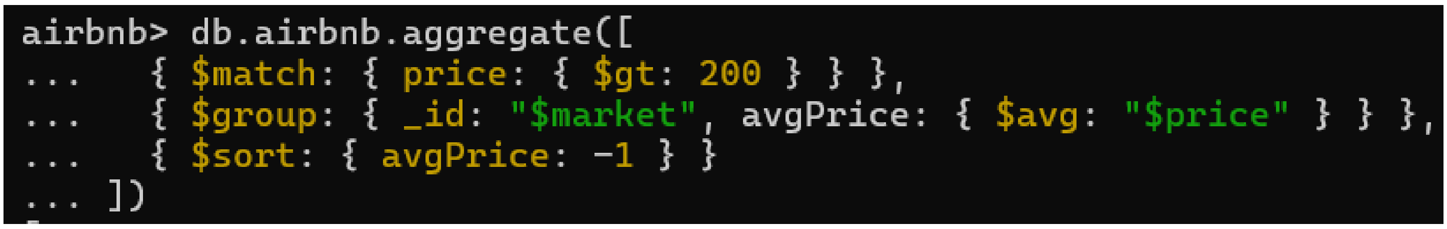
We use the final stage **$out** to show the **highRatedRoomTypes.**

Output: Empty.

Like the same situation in the tasks 5.1 and 5.2 , the aggregation pipeline executed the query without errors, but the output **highRatedRoomTypes** remains empty. This empty result happens, due to the absence of the documents inside the source collection during execution.

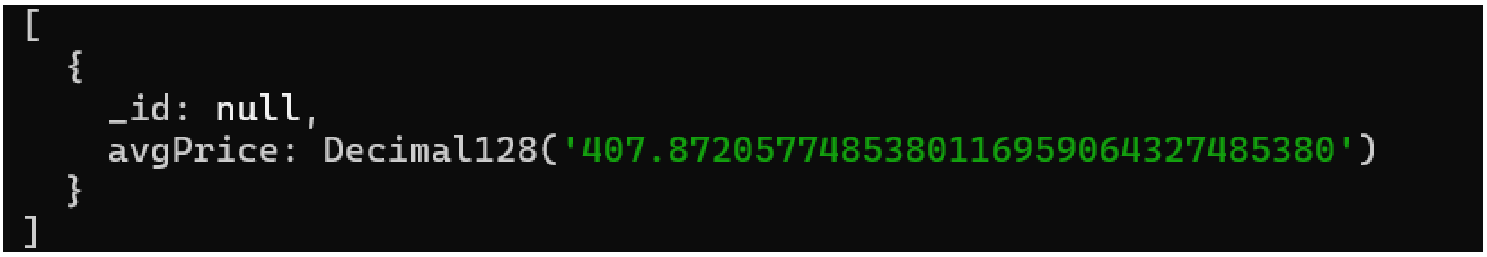
Task 4: Explain the difference between returning aggregation results directly to the shell and writing aggregation results to a collection.

Direct return to shell:



```
airbnb> db.airbnb.aggregate([
...   { $match: { price: { $gt: 200 } } },
...   { $group: { _id: "$market", avgPrice: { $avg: "$price" } } },
...   { $sort: { avgPrice: -1 } }
... ])
```

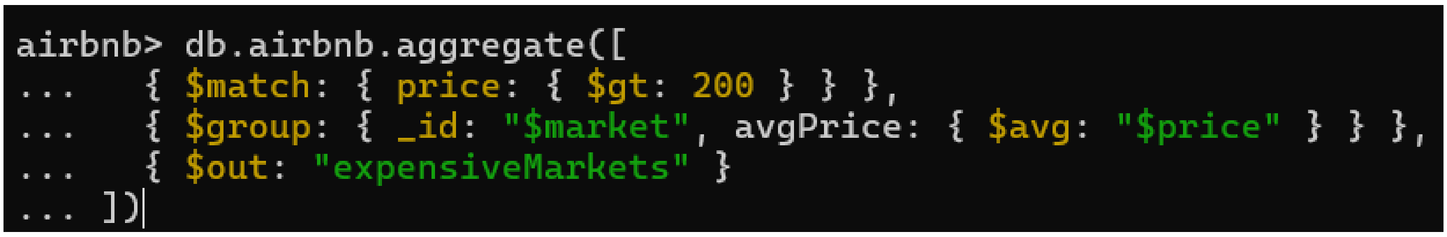
Output:



```
[
  {
    _id: null,
    avgPrice: Decimal128('407.8720577485380116959064327485380')
  }
]
```

This output shows that the grouping stage worked correctly and the pipeline showed the expected aggregate result.

Writing to collection:



```
airbnb> db.airbnb.aggregate([
...   { $match: { price: { $gt: 200 } } },
...   { $group: { _id: "$market", avgPrice: { $avg: "$price" } } },
...   { $out: "expensiveMarkets" }
... ])
```

Output: Empty.

This occurs when no documents match the aggregation criteria or when the collection targeted by **$merge** or **$out** currently contains no data. All stages executed without issues, but no documents met the conditions, so the result collection remains empty.

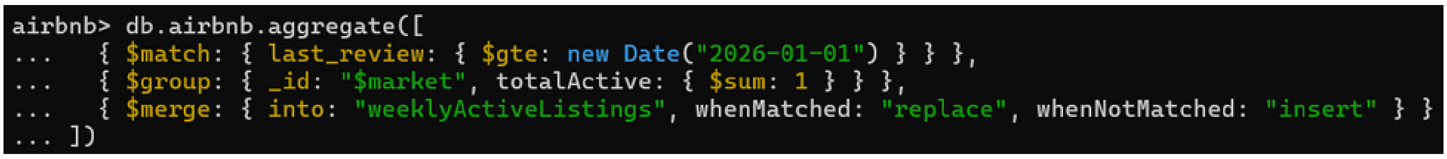
The main difference:

Returning aggregation results directly to the shell is useful not only for testing, fast analysis and debugging pipelines, but also it doesn't persist data, because the one session ends and the output result is gone.

Writing aggregation resuts to a collection is useful for pre-computing analytics and reports. Also, it supports indexing and additional operations on the output result. Moreover, the results are saved into a specified collection for later querying.

Task 5: Describe real-world scenarios where storing aggregation results in collections is preferable.

Code:



```
airbnb> db.airbnb.aggregate([
...   { $match: { last_review: { $gte: new Date("2026-01-01") } } },
...   { $group: { _id: "$market", totalActive: { $sum: 1 } } },
...   { $merge: { into: "weeklyActiveListings", whenMatched: "replace", whenNotMatched: "insert" } }
... ])
```

In the **$match** we write **last_review:** and set the condition: greater than ot equal to **new Date ("2026-01-01").**

Also, we group by **"$market",** setting the sum: 1 into the **totalActive**.

Finally, we use **$merge** for writing the result into **"weeklyActiveListings"** with the given conditions: **replace** (whenMatched) or **insert** (whenNotMatched).

Output: empty.

Like in the previous tasks: 5.1, 5.2, 5.3 , the result is also empty. No one document satisfies the criteria or the aggregation pipeline hasn't produced results yet. The query has been run correctly, but with the empty result.

PART VI: Conceptual Understanding

Task 1: Explain the purpose of aggregation pipelines in MongoDB and how they differ from standard find queries.

The purpose of aggregation pipelines is to process data records through a series of stages to produce compared results, efficiently transforming data into meaningful insights.

Feature	**Standard find queries**	**Aggregation pipelines**
Purpose	Find specific documents that match criteria.	Transform, analyze and consolidate data.
Output	A cursor containing the original documents.	A cursor containing the computed resuts.
Complexity	Cannot join data from different collections internally.	Can join data using the **$lookup** stage.
Data Joining	Minimal processing (only filtering and sorting).	Extensive processing inside the database engine, reducing network overhead.

Task 2: Describe the role of the group stage and how it enables analytical queries.

$group stage is the primary key stage of aggregation, which combines documents by a given “grouping key” and calculates summary values, such as average, sum and count.

Analytical queries are very heavy and require too many resources. The query **$group** allows us to collapse millions of transactions into compact reports. Using **$addToSet,** you can retrieve unique values. Segmentation groups data by demography or geography.

Thanks to that, MongoDB does every difficult work on the server’s side , returning the done business-answer.

Task 3: Explain how the order of aggregation pipeline stages affects performance.

The order of aggregation is the difference between instant response and a server crashing out of a memory.

First of all, we should write **$match** , because it allows us to use indexes and remove unnecessary data at once.

Secondly, if your document contains 50 fields, it’s useful to use **$project** command. This reduces the load on RAM and reduces the risk of exceeding the 100 MB per stage limit.

Thirdly, if the **$sort** command comes first, it can use the index.

If after the **$group,** the index won’t help, because MongoDB will have to perform a resource-intensive in-memory sort.

Thus, bad order makes the base to process too many gigabytes of garbage, whereas good order makes the base to work only with the necessary data.

Task 4: Discuss common mistakes made when building aggregation pipelines.

1) Ingoring indexes at the beginning of pipeline

The most common mistake: putting **$match** stage in the middle or end of a pipeline.

Result: MongoDB executes Collection Scan, which kills perfomance of big data.

2) 100MB memory limit violation

The most common mistake: Executing the difficult operations **$group** and **$sort** by big data with no using indexes.

Result: The query is just executing with an error, especially if the data is being greather than on the testing stage.

3) The wrong order between $sort and $limit

This is the original mistake that affects speed.

The most common mistake: First of all, limit the selection (**$limit**), then sort (**$sort**). Thus, you'll get 10 random documents, which will be then sorted.

Result: The query will receive you 10,000,000 random documents, sort all of them, and then limit the output to the top 10.

Task 5: Explain how aggregation pipelines support real-world analytics and reporting use cases.

Aggregation pipelines allow real-time data analysis and dashboard generations. Examples:

- Calculating average values, prices, sums and counts;
- Counting total reviews per room type;
- Generating reports for business intelligence.

Pipelines provide a flexible way to transform raw MongoDB documents into actionable insights.