|  |  |
| --- | --- |
| 4 | PROGRAM 4 |
| A | Write a MongoDB query to find the name and address of the restaurants that received a grade of 'B' or 'C' on a specific date |
|  | db.restaurants.find(  {  "grades": {  $elemMatch: {  "date": ISODate("2013-04-05"),  "grade": { $in: [ "B", "C" ] }  }  }  },  {  "name": 1,  "address": 1  }  ) |
|  | The said query in MongoDB that returns the documents that contains only the "name" and "address" fields of the matched documents.  The $elemMatch operator searches for an element in the "grades" array where both the "date" field is equal to the given date (2013-04-05) and the "grade" field is either "B" or "C". The $in operator is used to match either "B" or "C" in the "grade" field.  The date field is matched using the ISODate() function, which converts a date string to an ISODate object that can be compared with date objects of the collection restaurants. |
| B | Write a MongoDB query to find the name and address of the restaurants that have at least one 'A' grade and no 'B' grades. |
|  | db.restaurants.find({  $and: [  { "grades.grade": "A" },  { "grades.grade": { $not: { $eq: "B" } } },  { "grades.grade": { $not: { $eq: "C" } } }  ]  },  { name: 1, address: 1, "grades.grade":1, \_id: 0 }) |
|  | The said query in MongoDB that finds the name, address, and grades of restaurants that have at least one 'A' grade, no 'B' grades, and no 'C' grades.  The $and operator to combines three conditions, the first one checks for at least one 'A' grade. A dot notation have used to access the grade field within the grades array.  The second and third ones use the $not operator to exclude any documents that have a 'B' or 'C' grade, respectively which are coming to compare by the $eq operator.  The argument of the find() method which specifies which fields to include in the query results and it includes the name, address, and grades.grade fields, but excludes the \_id field. |
|  | db.restaurants.find(  {  $and: [  { "grades.grade": { $ne: "B" } },  { "grades.grade": { $ne: "C" } },  { "grades.grade": "A" }  ]  },  {  "name": 1,  "address": 1,  "grades": 1  }  ) |
|  | Explanation:  1. \*\*$and\*\*: This operator combines multiple conditions. Each condition must be true for the document to be included in the results.  2. \*\*$ne\*\*: This operator specifies that the field should not equal the value specified. Here, it's used to ensure that there are no 'B' or 'C' grades.  3. \*\*grades.grade: "A"\*\*: This condition ensures that there is at least one 'A' grade.  This query checks three conditions:  - There is no 'B' grade.  - There is no 'C' grade.  - There is at least one 'A' grade.  By explicitly listing these conditions, we ensure the desired criteria are met. The projection ensures that only the `name`, `address`, and `grades` fields are returned. |
| c | Write a MongoDB query to find the name and address of the restaurants that have the word 'coffee' in their name. |
|  | db.restaurants.find(  {  name: { $regex: "coffee", $options: "i" }  },  {  name: 1,  address: 1  }  ) |
|  | db.restaurants.find(  {  name: { $regex: /coffee/i }  },  {  name: 1,  address: 1,  cuisine: 1  }  ) |
| D | Write a MongoDB query to find the name, address, and cuisine of the restaurants that have a cuisine that contains the word 'Pizza' |
|  | db.restaurants.createIndex({ cuisine: "text" })  db.restaurants.find(  {  $text: { $search: "Pizza" }  },  {  name: 1,  address: 1,  cuisine: 1,  score: { $meta: "textScore" }  }  ).sort({ score: { $meta: "textScore" } }) |
|  | Let's break down the query and discuss the relevance of the `$meta` operator:  1. \*\*$text Operator\*\*: The `$text` operator enables MongoDB's full-text search functionality. It performs a text search on one or more fields that have a text index. In this case, we're searching for the word "Pizza" in the indexed `cuisine` field.  2. \*\*$search Parameter\*\*: Within the `$text` operator, the `$search` parameter specifies the search string. MongoDB will search for documents that contain this string in the indexed fields.  3. \*\*Projection\*\*: The projection part `{ name: 1, address: 1, cuisine: 1, score: { $meta: "textScore" } }` specifies the fields to include in the query results. Additionally, it includes the `score` field, which represents the relevance score of each document to the search query.  4. \*\*$meta Operator\*\*: The `$meta` operator is used to access metadata associated with the query. In this case, `"textScore"` is the metadata indicating the relevance score of the document to the search query.  5. \*\*Sort\*\*: The `sort()` method is used to sort the query results based on the relevance score (`textScore`) in descending order. This ensures that documents with higher relevance scores appear first in the results.  ### Importance of `$meta` Operator:  The `$meta` operator is specifically used with text search queries to access metadata related to the text search. In this query:  - The `score: { $meta: "textScore" }` projection includes the relevance score (`textScore`) for each document in the query results.  - The `sort()` method then sorts the documents based on their relevance scores.  ### Advantages over Regex Approach:  Using text search with a text index and the `$text` operator offers several advantages over the regex approach for certain scenarios:  1. \*\*Relevance Ranking\*\*: The text search approach provides relevance ranking, which allows you to sort the results based on how well they match the search query. This is particularly useful for ranking search results by relevance.    2. \*\*Efficiency\*\*: Text indexes are optimized for text search operations and can provide faster and more efficient search performance compared to regex-based searches, especially on large collections.  3. \*\*Language Support\*\*: Text search in MongoDB supports language-specific stemming and stop words, which can improve the accuracy of search results for natural language queries.  4. \*\*Index Utilization\*\*: Text search queries can leverage text indexes, which are specifically designed for text search operations. These indexes are more efficient for text search queries compared to regular indexes used with regex searches.  Overall, while regex-based searches are flexible and suitable for certain use cases, text search with text indexes and the `$text` operator offers enhanced functionality, performance, and relevance ranking for full-text search operations in MongoDB. |
| E | Create a query to delete all restaurants where the lowest grade score is below 50. |
|  | db.restaurants.aggregate([  {  $addFields: {  lowestGradeScore: { $min: "$grades.score" }  }  },  {  $match: {  lowestGradeScore: { $lt: 50 }  }  }  ]).forEach(function(doc) {  db.restaurants.deleteOne({ \_id: doc.\_id });  }); |
|  | **Explanation:**   1. **$addFields**: This stage adds a new field lowestGradeScore to each document. This field contains the minimum score from the grades.score array.    * $min: Aggregation operator to find the minimum value in an array. 2. **$match**: This stage filters documents where the lowestGradeScore is less than 50. 3. **forEach**: Iterates over the matched documents and deletes each one using deleteOne() based on its \_id.   This aggregation pipeline identifies restaurants with the lowest grade score below 50 and deletes them from the collection. Make sure to test this query in a safe environment before running it in production, as it will permanently remove data from your database. |
|  | // Step 1: Find restaurants with lowest grade score below 50  var restaurantsToDelete = db.restaurants.aggregate([  {  $unwind: "$grades" // Unwind the grades array  },  {  $group: {  \_id: "$\_id",  lowestGradeScore: { $min: "$grades.score" } // Calculate the lowest grade score for each restaurant  }  },  {  $match: {  lowestGradeScore: { $lt: 50 } // Filter restaurants with lowest grade score below 50  }  }  ]).toArray();  // Step 2: Delete identified restaurants  restaurantsToDelete.forEach(function(restaurant) {  db.restaurants.deleteOne({ \_id: restaurant.\_id }); // Delete each identified restaurant  }); |
|  | Explanation:  1. **$unwind**: This stage deconstructs the grades array, creating a separate document for each element of the array. This is necessary to perform aggregation operations on the elements of the array. 2. **$group**: This stage groups the documents by \_id (restaurant ID) and calculates the minimum grade score (lowestGradeScore) for each restaurant. 3. **$match**: This stage filters the grouped documents to include only those restaurants where the lowestGradeScore is less than 50. 4. **toArray()**: Converts the cursor result of the aggregation pipeline into an array, allowing us to iterate over the documents. 5. **forEach**: Iterates over the array of identified restaurants to delete each one using the deleteOne() method. |