Contents

1 Exercise 1 1

2 Exercise 2 2

3 Exercise 3 4

4 Exercise 4 6

5 Exercise 5 8

1 Exercise 1

Objective: Find sales made in specific locations with certain conditions on customers and items.

```
db.sales.find(
  {
    // Filtros
    storeLocation: {
      // Condition: sales made in "London", "Austin" or "San Diego"
      $in: ['London', 'Austin', 'San Diego']
    },
    'customer.age': {
      // Condition: customer age greater than or equal to 18
      $gte: 18
    },
    items: {
      // Condition: items must have a price of at least 1000 and be tagged as "school" o
      $elemMatch: {
        price: {
          $type: 'number',
          $gte: 1000
        },
        $or: [
          {
            tags: 'school'
          },
          {
            tags: 'kids'
          }
        ]
      }
    }
  },
  {
    // Proyección
    _id: 0,
    sale: '$_id',
    saleDate: 1,
    storeLocation: 1,
    customer_email: '$customer.email',
    age: '$customer.age',
    items: '$items'
  }
);
```

Explanation:

- `db.sales.find(...)`: This is a MongoDB query to find documents in the `sales` collection.
- The first argument is the query filter:
  - `storeLocation: { $in: [...] }`: Filters sales to only include those made in "London", "Austin", or "San Diego".
  - `'customer.age': { $gte: 18 }`: Filters to include only customers who are 18 years old or older.
  - `items: { $elemMatch: {...} }`: This checks that at least one item in the `items` array meets the specified conditions:
    - `price: { $type: 'number', $gte: 1000 }`: The item's price must be a number and at least 1000.
    - `$or: [...]`: The item must have a tag of either "school" or "kids".
- The second argument is the projection:
  - `_id: 0`: Excludes the default `_id` field from the results.
  - `sale: '$_id'`: Renames the `_id` field to `sale`.
  - `saleDate: 1`: Includes the `saleDate` field.
  - `storeLocation: 1`: Includes the `storeLocation` field.
  - `customer_email: '$customer.email'`: Renames the customer's email field.
  - `age: '$customer.age'`: Includes the customer's age.
  - `items: '$items'`: Includes the items array.

2 Exercise 2

Objective: Find sales in Seattle with specific purchase methods and date range, and calculate total amounts.

```
db.sales.aggregate([
  {
    // Filtros
    $match: {
      // Condition: sales from stores located in Seattle
      storeLocation: 'Seattle',
      // Condition: purchase method is ‘In store’ or ‘Phone’
      purchaseMethod: {
        $in: ['In store', 'Phone']
      },
      // Condition: sales made between 1st February 2014 and 31st January 2015
      saleDate: {
        $gte: new Date('2014-02-01'),
        $lte: new Date('2015-01-31')
      }
    }
  },
  {
    // Unwind items to process each item individually
    $unwind: '$items'
  },
  {
    // Grouping to calculate total sales
    $group: {
      _id: {
        sale_id: '$_id',
        email: '$customer.email',
        satisfaction: '$customer.satisfaction'
      },
      total: {
        // Calculate total price * quantity for each item
        $sum: {
          $multiply: [
            { $toDouble: '$items.price' },
            { $toDouble: '$items.quantity' }
          ]
        }
      }
    }
  },
  {
    // Proyección
    $project: {
      _id: 0,
      email: '$_id.email',
      satisfaction: '$_id.satisfaction',
      total: 1
    }
  },
  {
    // Sort by satisfaction (descending) and email (alphabetical)
    $sort: {
      satisfaction: -1,
      email: 1
    }
  }
]);
```

Explanation:

- `db.sales.aggregate([...])`: This is a MongoDB aggregation pipeline to process documents in the `sales` collection.
- `$match`: Filters documents based on specified conditions.
- `$unwind`: Deconstructs the `items` array, creating a separate document for each item.
- `$group`: Groups documents by sale ID, email, and satisfaction, calculating the total amount for each sale.
- `$project`: Restructures the output documents to include only the specified fields.
- `$sort`: Sorts the results first by satisfaction in descending order and then by email in ascending order.

3 Exercise 3

Objective: Create a view that calculates sales statistics by year and month.

```
salesInvoicedPipeline = [
  {
    // Unwind items to process each item individually
    $unwind: '$items'
  },
  {
    // Grouping to calculate total sales
    $group: {
      _id: {
        sale_id: '$_id',
        year: { $year: '$saleDate' },
        month: { $month: '$saleDate' }
      },
      total: {
        $sum: {
          $multiply: [
            { $toDouble: '$items.price' },
            { $toDouble: '$items.quantity' }
          ]
        }
      }
    }
  },
  {
    // Grouping by year and month to calculate statistics
    $group: {
      _id: {
        year: '$_id.year',
        month: '$_id.month'
      },
      min: { $min: '$total' },
      max: { $max: '$total' },
      total: { $sum: '$total' },
      average: { $avg: '$total' }
    }
  },
  {
    // Proyección
    $project: {
      _id: 0,
      year: '$_id.year',
      month: '$_id.month',
      min: 1,
      max: 1,
      total: 1,
      average: 1
    }
  },
  {
    // Sort by chronological order
    $sort: {
      year: 1,
      month: 1
    }
  }
];
```

```
// Create the view
db.createView('salesInvoiced', 'sales', salesInvoicedPipeline);
```

Explanation:

- `salesInvoicedPipeline`: Defines the aggregation pipeline for creating the view.
- `$unwind`: Deconstructs the `items` array.
- `$group`: First groups by sale ID, year, and month to calculate total sales for each sale.
- The second `$group` aggregates the results by year and month, calculating minimum, maximum, total, and average sales.
- `$project`: Restructures the output to include only the relevant fields.
- `$sort`: Sorts the results chronologically.
- `db.createView(...)`: Creates a view named `salesInvoiced` based on the defined pipeline.

4 Exercise 4

Objective: Show store location, average sales, objectives, and differences.

```
db.storeObjectives.aggregate(
  {
    // Join with sales data
    $lookup: {
      from: 'sales',
      localField: '_id',
      foreignField: 'storeLocation',
      as: 'store_info',
      pipeline: [
        {
          // Unwind items
          $unwind: '$items'
        },
        {
          // Group to calculate total sales
          $group: {
            _id: {
              sale_id: '$_id',
              storeLocation: '$storeLocation'
            },
            total: {
              $sum: {
                $multiply: [
                  { $toDouble: '$items.price' },
                  { $toDouble: '$items.quantity' }
                ]
              }
            }
          }
        },
        {
          // Calculate average sales per store
          $group: {
            _id: '$storeLocation',
            average: { $avg: '$total' }
          }
        }
      ]
    }
  },
  {
    // Set to get the first element of store_info
    $set: {
      store_info: {
        $first: '$store_info'
      }
    }
  },
  {
    // Proyección
    $project: {
      _id: 0,
      storeLocation: '$_id',
      average_sales: '$store_info.average',
      objective: 1,
      difference: {
        // Calculate difference between average and objective
        $subtract: [
          { $toDouble: '$store_info.average' },
          { $toDouble: '$objective' }
        ]
      }
    }
  }
);
```

Explanation:

- `db.storeObjectives.aggregate(...)`: Aggregates data from the `storeObjectives` collection.
- `$lookup`: Joins the `sales` collection based on store location.
- The inner pipeline unwinds the `items`, groups by store location, and calculates total sales.
- The outer pipeline sets the first element of `store_info` and projects the desired fields.
- The `difference` field calculates the difference between average sales and the objective.

5 Exercise 5

Objective: Specify validation rules for the `sales` collection using JSON Schema.

```
db.runCommand({
  collMod: 'sales',
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: [
        'saleDate',
        'storeLocation',
        'purchaseMethod',
        'customer',
        'items'
      ],
      properties: {
        saleDate: {
          bsonType: 'date'
        },
        storeLocation: {
          bsonType: 'string',
          enum: [
            'London',
            'New York',
            'Denver',
            'San Diego',
            'Austin',
            'Seattle'
          ]
        },
        purchaseMethod: {
          bsonType: 'string',
          enum: [
            'Online',
            'Phone',
            'In store'
          ]
        },
        customer: {
          bsonType: 'object',
          required: [
            'gender',
            'age',
            'email',
            'satisfaction'
          ],
          properties: {
            gender: {
              bsonType: 'string',
              enum: [
                'M',
                'F'
              ]
            },
            age: {
              bsonType: 'int',
              minimum: 0,
              maximum: 200
            },
            email: {
              bsonType: 'string',
              pattern: '^(.*)@(.*)\\.(.{2,4})$'
            },
            satisfaction: {
              bsonType: 'int',
              minimum: 1,
              maximum: 5
            }
          }
        },
        items: {
          bsonType: 'array',
          minLength: 1,
          required: [
            'name',
            'price',
            'quantity'
          ],
          properties: {
            name: {
              bsonType: 'string'
            },
            tags: {
              bsonType: ['string'],
            },
            price: {
              bsonType: 'double',
              minimum: 0
            },
            quantity: {
              bsonType: 'int',
              minimum: 1
            }
          }
        },
        couponUsed: {
          bsonType: 'bool'
        }
      }
    }
  }
});
```

Explanation:

- `db.runCommand(...)`: Executes a command to modify the `sales` collection.
- `validator`: Specifies the validation rules using JSON Schema.
- `bsonType`: Defines the expected data type for each field.
- `required`: Lists fields that must be present in each document.
- `enum`: Restricts the values for certain fields to a predefined list.
- `pattern`: Uses a regex pattern to validate the format of the email.
- `minimum` and `maximum`: Sets constraints on numerical fields.
- `minLength`: Ensures that the `items` array contains at least one item. ### Test Cases for Validation **Case A**: Successful insertion

```
db.sales.insertOne({
  saleDate: new Date('2023-11-17'),
  items: [
    {
      name: "printer paper",
      tags: [
        "office",
        "stationary"
      ],
      price: 40.01,
      quantity: 2
    }
  ],
  storeLocation: 'London',
  customer: {
    gender: "M",
    age: 20,
    email: "emanuelherrador2@gmail.com",
    satisfaction: 5
  },
  couponUsed: false,
  purchaseMethod: 'Online'
});
```

Case B: Failed insertion (missing items)

```
db.sales.insertOne({
  saleDate: new Date('2023-11-17'),
  storeLocation: 'London',
  customer: {
    gender: "M",
    age: 20,
    email: "emanuelherrador2@gmail.com",
    satisfaction: 5
  },
  couponUsed: false,
  purchaseMethod: 'Online'
});
```

Explanation:

- The first case is successful because it meets all validation criteria.
- The second case fails because it does not include any items, violating the `minLength` requirement for the `items` array. This detailed breakdown covers the purpose and functionality of each part of the code, providing a comprehensive understanding of the MongoDB operations involved.