



Higher Diploma in Science in Data Analytics

ASSESSMENT

Module Code: **B8IT155**

Module Description: **Big Data Managing and Processing**

Student Name: **Javier Ignacio Astorga Di Pauli**

Student ID: **20023978**

Table of Contents

[Question 1 - Structured Data - SQL](#)

[Task 1](#)

[Task 2: Counting Films Listed](#)

[Task 3: Actor Names in Upper Case](#)

[Task 4: Actor Named John](#)

[Task 5: Actors with 'OO' in Last Name](#)

[Task 6: Specific Countries' information](#)

[Task 7: Films in Mandarin](#)

[Task 8: Schema of Address Table](#)

[Task 9: Staff Names and Addresses](#)

[Task 10: Actors in 'Thief Pelican'](#)

[Task 11: Family Films](#)

[Task 12: Top 10 Frequently Rented Movies](#)

[Task 13: Top Five Genres in Gross Revenue](#)

[Question 2 - Semi-Structured Data - NoSQL - MongoDB](#)

[Task 1: Excluding Address Information](#)

[Task 2: Querying Carol's Age](#)

[Task 3: Sorting by Age in Descending Order](#)

[Task 4: Identifying Tennis Enthusiasts](#)

[Task 5: Counting Tennis Fans](#)

[Task 6: Locating Individuals Older Than 30](#)

[Task 7: Finding Names Containing 'a'](#)

[Task 8: Calculating the Average Age of Everyone](#)

[Task 9: Calculating the Average Age of Tennis Enthusiasts](#)

[Task 10: Counting Unique Cities Listed in the Collection](#)

[Question 3 - Reflection on changes on the Big Data landscape](#)

ASSESSMENT

Question 1 - Structured Data - SQL

Code:

<https://colab.research.google.com/drive/1M4YiBXBwNSzm22bjm-9r-ghwzHgRoHMa?usp=sharing>

The first section of the code is dedicated to setting up the MySQL database system and initializing it with the Sakila database. The process is broken down into several steps, each playing a crucial role in preparing the environment for database operations.

```
Setting Up SQL and Sakila Database

✓ # Install MySQL on the environment
!apt -y install mysql-server -V

✓ [3] # Start the MySQL server
!service mysql start

✓ [4] # Check the MySQL version to ensure it's installed correctly
!mysql --version

[5] # Download and unzip the Sakila database
!wget https://downloads.mysql.com/docs/sakila-db.zip

✓ [6] !unzip sakila-db.zip

✓ [7] %cd sakila-db

✓ [8] !ls

[9] # Load the Sakila schema and data into MySQL
!mysql -e 'SOURCE sakila-schema.sql'
!mysql -e 'SOURCE sakila-data.sql'
```

1. **Installing MySQL Server:** The command `!apt -y install mysql-server -V` is executed to install the MySQL server on the environment. The `-y` flag indicates that the process should proceed without prompting for confirmations, facilitating an automated setup. The `-V` option is used to display version information, confirming the successful installation of the package.
2. **Starting the MySQL Server:** Once installed, the MySQL server is started with the command `!service mysql start`. This is necessary to enable the creation, modification, and querying of databases.
3. **Checking MySQL Version:** The command `!mysql --version` is used to verify the installation by outputting the version of the MySQL server. This step is essential to ensure that MySQL is properly installed and ready for use.
4. **Downloading Sakila Database:** The Sakila sample database is downloaded using `!wget https://downloads.mysql.com/docs/sakila-db.zip`. This database is a well-known

sample provided by MySQL to assist users in learning and testing database concepts and operations.

5. **Unzipping the Sakila Database:** After downloading, the Sakila database is unzipped with the command `!unzip sakila-db.zip`. This step extracts the database schema and data files required for loading into the MySQL server.
6. **Navigating to the Sakila Database Directory:** The `%cd sakila-db` command changes the current working directory to sakila-db, where the unzipped database files are located. This is important for executing subsequent operations directly on the database files.
7. **Listing Files in the Directory:** `!ls` is executed to list the contents of the sakila-db directory, providing visibility into the files available for loading into the database.
8. **Loading Sakila Schema and Data:** Finally, the database schema is loaded with `!mysql -e 'SOURCE sakila-schema.sql'`, followed by the loading of the database data with `!mysql -e 'SOURCE sakila-data.sql'`. These commands use the `-e` option to execute the SQL commands contained within the sakila-schema.sql and sakila-data.sql files, respectively. Executing these files sets up the database structure (tables, indexes, etc.) and populates it with sample data.


Each step in this sequence is intentional and builds upon the previous step to ensure that the MySQL server is installed, started, and loaded with a sample database for experimentation and learning purposes. This setup is foundational for executing further SQL queries and operations on the Sakila database.

Task 1: Counting Tables in the Database

The approach taken to solve this task involves querying the `information_schema.tables`, which is a meta-database that provides information about all other databases and tables within a MySQL instance.

1. `!mysql -D "sakila" -e`: This initiates a MySQL command line interface operation targeting the "sakila" database. The `-D` flag is used to specify the database, and `-e` is utilized to execute the SQL statement that follows in double quotes directly from the command line.
2. `SELECT COUNT(*) AS 'Number of Tables'`: This SQL statement counts the number of rows that satisfy the query's conditions, which in this context corresponds to the number of tables within the specified schema. `COUNT(*)` is a function that counts the number of rows, and `AS 'Number of Tables'` renames the output column for clarity.
3. `FROM information_schema.tables`: The query is executed against the `information_schema.tables` table, which stores information about all tables across all databases in the MySQL server.
4. `WHERE table_schema = 'sakila'`: The `WHERE` clause filters the results to include only those tables that belong to the 'sakila' database (`table_schema = 'sakila'`).

The output of the executed command is a table format that presents the count resulting from the query.




Number of Tables
23

Task 2: Counting Films Listed

For the second task, the objective is to determine the total count of films listed in the Sakila database. This is achieved by running a SQL query on the film table:

```
## This query counts the number of films listed in the film table.
!mysql -D "sakila" -e "SELECT COUNT(*) AS 'Number of Films' FROM film;"
```

1. The command `!mysql -D "sakila" -e` specifies that the query is to be executed on the "sakila" database.
2. The SQL query `SELECT COUNT(*) AS 'Number of Films' FROM film;` calculates the total number of entries in the film table. `COUNT(*)` counts all rows, and `AS 'Number of Films'` renames the output column for better clarity.
3. According to the output, there are 1000 films listed in the database:



Number of Films
1000

Task 3: Actor Names in Upper Case

In task 3, the goal is to provide a reformatted listing of actor names from the Sakila database, with specific formatting requirements. The SQL query employed here achieves this by manipulating the data within the actor table:

```
## This query concatenates the first_name and last_name of each actor,
## converts the fullname to upper case, and labels it as 'Actor Name'.
!mysql -D "sakila" -e "SELECT UPPER(CONCAT(first_name, ' ', last_name)) AS 'Actor Name'
FROM actor;"
```

1. The command `!mysql -D "sakila" -e` specifies execution on the "sakila" database.
2. The query `SELECT UPPER(CONCAT(first_name, ' ', last_name)) AS 'Actor Name' FROM actor;` performs the following operations:

- CONCAT(first_name, ' ', last_name): Concatenates the first_name and last_name of each actor with a space in between, forming the full name.
 - UPPER(...): Converts the concatenated full name into upper case letters.
 - AS 'Actor Name': Labels the resulting column as "Actor Name".
3. The output presents a list of actor names in a single column, in upper case letters. The list starts with "PENELOPE GUINNESS", followed by other names like "NICK WAHLBERG", "ED CHASE", and continues down the list.

Actor Name
PENELOPE GUINNESS
NICK WAHLBERG
ED CHASE
JENNIFER DAVIS
JOHNNY LOLLOBRIGIDA
BETTE NICHOLSON
GRACE MOSTEL

Task 4: Actor Named John

In task 4, the query aims to identify actors within the Sakila database who have the first name "John". Here's how the task was executed:

```
[ ] ## This query retrieves the actor_id, first_name, and last_name
    ## for actors with the first name 'John'.
    !mysql -D "sakila" -e "SELECT actor_id, first_name, last_name \
                           FROM actor WHERE first_name = 'John';"
```

1. The SQL statement SELECT actor_id, first_name, last_name FROM actor WHERE first_name = 'John'; performs the following operations:
 - Selects actor_id, first_name, and last_name columns from the actor table to retrieve relevant actor information.
 - The WHERE first_name = 'John' clause filters the results to only include actors whose first_name is "John".
2. The output displays the requested information for an actor matching the given first name condition:

actor_id	first_name	last_name
192	JOHN	SUVARI

It reveals that there is one actor in the database with the first name "John", having the actor ID '192' and the last name "SUVARI".


Task 5: Actors with 'OO' in Last Name

For Task 5, the underlying query focuses on identifying actors within the Sakila database whose last names incorporate the character sequence "OO". The operation is carried out by a meticulously constructed SQL query followed by ordering the results first by the actors' last names and subsequently by their first names. This ensures a systematic and reader-friendly output.

```
[ ] ## Actors whose last names contain 'OO' are found and ordered by last_name then first_name.
mysql -D "sakila" -e "SELECT first_name, last_name \
                      FROM actor WHERE last_name LIKE '%OO%' \
                      ORDER BY last_name, first_name;"
```

1. **Selection and Filtering:** The core of the SQL query is encapsulated in the SELECT statement, which extracts the first_name and last_name from the actor table. The WHERE clause is crucial here; it filters the dataset to only include those records where the last_name column contains the character sequence "OO". This filtering is achieved through the use of the LIKE '%OO%' operator, with % acting as a wildcard for zero or more characters either side of "OO".
2. **Sorting:** Following the selection and conditional filtering of records, the ORDER BY clause introduces a two-tiered sorting sequence. Initially, the outcomes are sorted based on the last_name, which ensures that records are alphabetically grouped by surname. Subsequently, a secondary sort is applied using first_name, offering an alphabetical ordering within each last name group. This detailed sorting mechanism enhances the readability and utility of the output.

The output is presented in a tabular format, comprising two columns: first_name and last_name. Each row represents an actor whose last name meets the criteria of containing "OO", illustrating the successful execution of the filtering condition. The results are systematically ordered, first by the last_name and then by the first_name, as evidenced by the arrangement seen from "BLOOM" through to "WOOD". This ordered presentation aids in the quick identification and analysis of actors fitting the said condition within the Sakila database.



first_name	last_name
KEVIN	BLOOM
EWAN	GOODING
GREGORY	GOODING
ANGELA	WITHERSPOON
FAY	WOOD
UMA	WOOD

Task 6: Specific Countries' information

In Task 6, the task's objective is to extract and display the country_id and country columns for a select group of countries—specifically Poland, Angola, and Zambia—from the Sakila

database. This task is accomplished through a succinct SQL query that targets the country table within the database.

```
[ ] ## Retrieves country_id and country from the country table for specified countries.
!mysql -D "sakila" -e "SELECT country_id, country FROM country \
                        WHERE country IN ('Poland', 'Angola', 'Zambia');"
```

1. **Column Selection:** The SELECT statement specifies that only the country_id and country columns are to be retrieved from the country table.
2. **Filtering Conditions :** The WHERE country IN ('Poland', 'Angola', 'Zambia'); clause is used to filter the results to only those rows where the country matches one of the values listed ("Poland", "Angola", or "Zambia"). The IN operator simplifies the inclusion of multiple specific conditions into a single statement.

The output is a straightforward table structure with two columns: country_id and country. Each row represents one of the specified countries, listing its corresponding ID number along with the country's name. The country_id appears to be an identifier unique to each country within the database. The table shows that Angola has a country_id of 4, Poland is identified by 76, and Zambia by 109.

country_id	country
4	Angola
76	Poland
109	Zambia

Task 7: Films in Mandarin

The objective of Task 7 is twofold: firstly, to list the titles of all films within the Sakila database that are in the Mandarin language, and secondly, to count how many films in the database are in Mandarin. To achieve this, two SQL queries were executed. It became necessary to perform a manual inspection of the film table to verify the accuracy of the results obtained from these queries.

```
[ ] ## Lists titles of all films in Mandarin and counts them.
!mysql -D "sakila" -e "SELECT title FROM film WHERE language_id = \
                      (SELECT language_id FROM language \
                       WHERE name = 'Mandarin');"

[ ] !mysql -D "sakila" -e "SELECT COUNT(*) AS 'Number of Films in Mandarin' \
                          FROM film WHERE language_id = (SELECT language_id \
                                                           FROM language WHERE name = 'Mandarin');"

[ ] ## Manual Inspection of the film Table
!mysql -D "sakila" -e "SELECT film_id, title, language_id FROM film;"
```


1. **Listing Titles of Films in Mandarin:** This query aimed to retrieve the titles of films in Mandarin by matching their `language_id` to the `language_id` of the Mandarin language in the language table.
2. **Counting Films in Mandarin:** This query was designed to count the number of films available in Mandarin within the database, using a count operation combined with a similar filtering condition based on `language_id`.
3. **Manual Inspection of the film Table:** In response to receiving an unexpected output from the above queries, a manual inspection of the film table was executed. This inspection aimed to examine the `film_id`, `title`, and `language_id` columns to confirm the accuracy of the previous query results.

The output of the query intended to count the films available in Mandarin concluded with a result of zero - indicating that, according to the database queries executed, there are no films in the Mandarin language within the Sakila database.

```
+-----+
| Number of Films in Mandarin |
+-----+
|                                0 |
+-----+
```

Given this unexpected outcome, the subsequent manual inspection of the film table became a necessary step to verify the accuracy of the initial query results. The inspection revealed that all films in the database were associated with a `language_id` that did not correspond to Mandarin, specifically indicating that they were all in English.

```
+-----+-----+-----+
| film_id | title                | language_id |
+-----+-----+-----+
| 1 | ACADEMY DINOSAUR    | 1 |
| 2 | ACE GOLDFINGER      | 1 |
| 3 | ADAPTATION HOLES    | 1 |
| 4 | AFFAIR PREJUDICE    | 1 |
| 5 | AFRICAN EGG         | 1 |
| 6 | AGENT TRUMAN        | 1 |
| 7 | AIRPLANE STRESS     | 1 |
```

Task 8: Schema of Address Table

The purpose of Task 8 was to determine the specific SQL query that could elucidate the schema of the address table within the Sakila database—a necessity prompted by the absence of accessible documentation or schema details. To achieve this objective, a query was executed designed to produce the CREATE TABLE statement for the address table, effectively revealing its structure and constraints.

```
## Provides the CREATE statement needed to recreate the address table schema.
!mysql -D "sakila" -e "SHOW CREATE TABLE address;"
```

1. **Schema Retrieval** - The `SHOW CREATE TABLE address;` command was used for its capability to display the CREATE TABLE statement that can be used to recreate the address table's schema. This is particularly useful for understanding table

structure, including column definitions and any constraints or keys that have been established.

The output from this command presented the entire CREATE TABLE statement necessary to recreate the address table. This includes detailed descriptions of each column, such as address_id as a smallint unsigned type with the NOT NULL AUTO_INCREMENT property, and address as a varchar(50) also marked NOT NULL. Other descriptors detail the presence of additional columns (address2, district, city_id, etc.), their data types, default values, and certain constraints like NOT NULL and AUTO_INCREMENT.

Additionally, the schema outlines key constraints including a PRIMARY KEY on address_id, a foreign key constraint (fk_address_city) referencing the city table, and spatial keys. The statement also includes the table's engine type (InnoDB), its default charset (utf8mb4), and collation setting.

```
+-----+
| Table | Create Table
+-----+
| address | CREATE TABLE `address` (
  `address_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `address` varchar(50) NOT NULL,
  `address2` varchar(50) DEFAULT NULL,
  `district` varchar(20) NOT NULL,
  `city_id` smallint unsigned NOT NULL,
  `postal_code` varchar(10) DEFAULT NULL,
  `phone` varchar(20) NOT NULL,
  `location` geometry NOT NULL /*!80003 SRID 0 */,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`address_id`),
  KEY `idx_fk_city_id` (`city_id`),
  SPATIAL KEY `idx_location` (`location`),
  CONSTRAINT `fk_address_city` FOREIGN KEY (`city_id`) REFERENCES `city` (`city_id`) ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=606 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+-----+
```

Task 9: Staff Names and Addresses

In Task 9, the objective was to combine information from both the staff and address tables within the Sakila database to display the first and last names of each staff member alongside their addresses. This task underscores the utility of the SQL JOIN operation in merging data from multiple tables based on a common column.

```
## Uses a JOIN to display first and last names along with the address of each staff member.
!mysql -D "sakila" -e "SELECT staff.first_name, staff.last_name, address.address \
                        FROM staff JOIN address ON staff.address_id = address.address_id;"
```

1. **Column Selection** - The SELECT statement specifies that the first_name and last_name columns from the staff table, as well as the address column from the address table, should be retrieved.
2. **JOIN Operation** - The query employs the JOIN keyword to merge the staff and address tables. The ON staff.address_id = address.address_id; condition ensures that rows are joined where the address_id values in both tables match, indicating the corresponding address for each staff member.

The resulting output convey that there are two staff members listed in the Sakila database, each with a distinct address linked through their address_id.

first_name	last_name	address
Mike	Hillyer	23 Workhaven Lane
Jon	Stephens	1411 Lillydale Drive

Task 10: Actors in 'Thief Pelican'

Task 10 sets out to identify and list all actors appearing in the film titled 'Thief Pelican' within the Sakila database. This task is notably carried out via the deployment of subqueries within an SQL command, exemplifying a practical application of subqueries in filtering results based on conditions applied to other tables in a relational database.

```
## Uses subqueries to find all actors acting in 'Thief Pelican'.
!mysql -D "sakila" -e "SELECT actor.first_name, actor.last_name \
                        FROM actor \
                        JOIN film_actor ON actor.actor_id = film_actor.actor_id \
                        WHERE film_actor.film_id = \
                        (SELECT film_id FROM film WHERE title = 'Thief Pelican');"
```

1. **Data Retrieval and Join** - This command begins with a command to retrieve the first_name and last_name of actors from the actor table. It then establishes a JOIN between the actor and film_actor tables, where rows from both tables are merged based on a match between their actor_id values.
2. **Subquery for Film Identification** - Crucially, the WHERE clause incorporates a subquery that identifies the film_id of 'Thief Pelican' from the film table. This subquery allows for filtering actors associated with the identified film in the outer query.

The output is a table displaying the first and last names of actors. According to the returned data, actors such as KIRSTEN PALTROW, JOHNNY CAGE, and JAMES PITT, among others, are identified as part of the cast of 'Thief Pelican'. This list informs us that seven actors were found to be associated with this particular film within the Sakila database.

first_name	last_name
KIRSTEN	PALTROW
JOHNNY	CAGE
JAMES	PITT
WARREN	NOLTE
HARRISON	BALE
RENEE	TRACY
JAYNE	NOLTE

Task 11: Family Films

Task 11 involves pinpointing all movies within the Sakila database that are categorised under 'Family' films. This is achieved through a SQL query incorporating nested subqueries, showcasing an advanced technique for filtering data across related tables based on a specified category.

```
## Identifies all movies categorised as family films.
!mysql -D "sakila" -e "SELECT title FROM film \
    WHERE film_id IN \
        (SELECT film_id FROM film_category WHERE category_id = \
            (SELECT category_id FROM category WHERE name = 'Family')));"

```

1. **Nested Subqueries for Filtering** - The command follows a hierarchical approach:
 - The innermost subquery selects the category_id corresponding to the 'Family' category from the category table.
 - The middle subquery retrieves all film_id values that match the category_id of 'Family' from the film_category table.
 - The outer query then selects the titles of films from the film table where the film_id matches any of those identified in the list of family film IDs.

The output lists the titles of movies categorised as 'Family' films. Some of these titles include "AFRICAN EGG", "APACHE DIVINE", and "ATLANTIS CAUSE", among others—indicating a variety of films that fall under the family category within the database. The list starts with "AFRICAN EGG", followed by other names like "APACHE DIVINE", "ATLANTIS CAUSE", and continues down the list.

```
+-----+
| title |
+-----+
| AFRICAN EGG |
| APACHE DIVINE |
| ATLANTIS CAUSE |
| BAKED CLEOPATRA |
| BANG KWAI |
| BEDAZZLED MARRIED |
| BILKO ANONYMOUS |
| BLANKET BEVERLY |
| BLOOD ARCADE |

```

Task 12: Top 10 Frequently Rented Movies

The aim of Task 12 is to identify and list the 10 movies that have been rented out the most frequently from the Sakila database. This task is accomplished through an SQL query that utilizes JOIN operations to link the film, inventory, and rental tables, along with GROUP BY and ORDER BY clauses to aggregate and sort the data accordingly.

```
## Displays the 10 most frequently rented movies.
!mysql -D "sakila" -e "SELECT film.title, COUNT(rental.rental_id) AS rental_count \
                        FROM film JOIN inventory ON film.film_id = inventory.film_id \
                        JOIN rental ON inventory.inventory_id = rental.inventory_id \
                        GROUP BY film.title \
                        ORDER BY rental_count DESC \
                        LIMIT 10;"
```

1. **Data Joining:** The query connects the film, inventory, and rental tables, using the common keys (film_id and inventory_id) to link film titles with their rental records.
2. **Aggregation of Rental Counts:** The use of COUNT(rental.rental_id) AS rental_count calculates the total number of rentals for each film, categorizing these counts under the alias rental_count.
3. **Grouping and Ordering:** The GROUP BY film.title clause groups the results by film title, ensuring that the counts are aggregated per film rather than across the entire dataset. The film titles with their respective rental counts are then ordered in descending order (ORDER BY rental_count DESC), highlighting those with the highest counts first.
4. **Limiting Results:** The LIMIT 10 clause constrains the output to the top 10 films by rental frequency, focusing on the most relevant data.

The resulting output is a table featuring two columns: title and rental_count. This table showcases the 10 most frequently rented movies in descending order of their rental counts. Titles such as "BUCKET BROTHERHOOD", "ROCKETEER MOTHER", and "RIDGEMONT SUBMARINE" appear at the top of the list, indicating their popularity among renters. The rental count signals the total number of times each film was rented, with "BUCKET BROTHERHOOD" leading with 34 rentals.

title	rental_count
BUCKET BROTHERHOOD	34
ROCKETEER MOTHER	33
RIDGEMONT SUBMARINE	32
GRIT CLOCKWORK	32
SCALAWAG DUCK	32
JUGGLER HARDLY	32
FORWARD TEMPLE	32
HOBBIT ALIEN	31
ROBBERS JOON	31
ZORRO ARK	31

Task 13: Top Five Genres in Gross Revenue

Task 13 aims to identify and list the top five movie genres by gross revenue within the Sakila database, offering insights into which genres generate the most income from rentals. This task is addressed through an SQL query that employs a series of JOIN operations to correlate data across multiple tables, including payment, rental, inventory, film_category, and category. Aggregation and sorting mechanisms are also essential in producing the final ranked list.

```
## Lists the top five genres in terms of gross revenue, ordered in descending order.
!mysql -D "sakila" -e "SELECT category.name AS genre, SUM(payment.amount) AS gross_revenue \
    FROM payment \
    JOIN rental ON payment.rental_id = rental.rental_id \
    JOIN inventory ON rental.inventory_id = inventory.inventory_id \
    JOIN film_category ON inventory.film_id = film_category.film_id \
    JOIN category ON film_category.category_id = category.category_id \
    GROUP BY genre \
    ORDER BY gross_revenue DESC \
    LIMIT 5;"
```

1. **Data Integration:** The command meticulously integrates data from the payment, rental, inventory, film_category, and category tables through JOIN clauses. This integration is predicated on matching keys across these tables to link each payment made to its corresponding film genre.
2. **Revenue Aggregation:** The SUM(payment.amount) AS gross_revenue part computes the total revenue generated from rentals for each genre, grouping these sums under the alias gross_revenue.
3. **Grouping and Ordering:** The results are grouped by genre (GROUP BY genre) to consolidate revenue sums by category. The data is then ordered in descending order based on gross revenue (ORDER BY gross_revenue DESC) to prioritize genres with the highest earnings.
4. **Limiting Output:** The LIMIT 5 clause restricts the output to the top five genres, focusing the analysis on the highest-grossing categories.

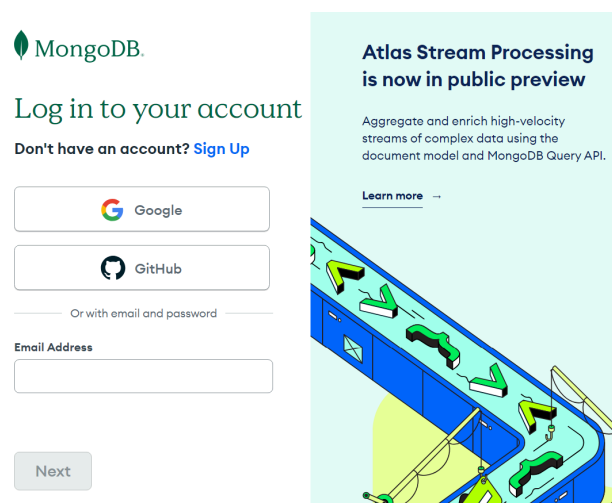
The output presents a table with two columns: genre and gross_revenue. The genres are listed alongside their corresponding gross revenue from rental payments, with the data sorted in descending order of revenue. According to the output, the Sports genre leads with a gross revenue of 5314.21, followed by Sci-Fi, Animation, Drama, and Comedy. This ranking provides clear insight into the most financially successful genres within the Sakila database.

genre	gross_revenue
Sports	5314.21
Sci-Fi	4756.98
Animation	4656.30
Drama	4587.39
Comedy	4383.58

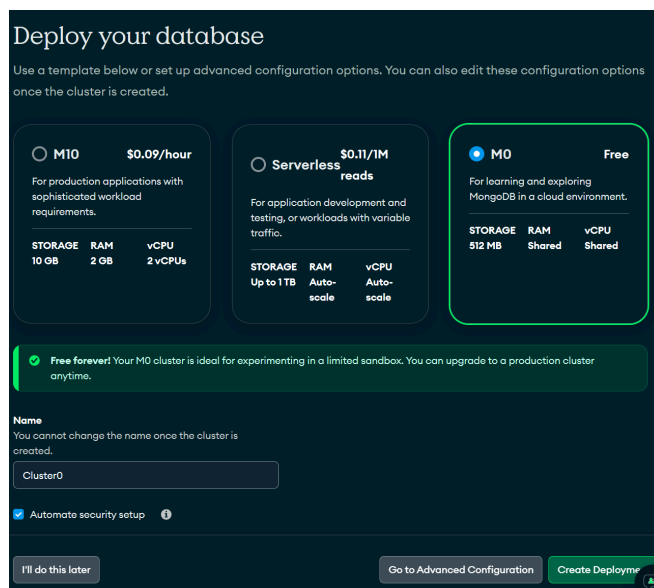
Question 2 - Semi-Structured Data - NoSQL - MongoDB

The process of setting up MongoDB Atlas and importing data into a new collection, followed by the insertion of specific documents as outlined, comprises a series of steps aimed at establishing a structured database environment suitable for NoSQL operations. The steps detailed below illustrate a comprehensive method for creating a functional database environment, ready for data storage and query operations.

1. **Sign Up/Log In to MongoDB Atlas:** The initial step involved accessing MongoDB Atlas by either signing into an existing account or registering a new one. MongoDB Atlas is the cloud database service provided by MongoDB, offering a platform for managing database instances in a cloud environment. This process grants users access to a suite of tools and functionalities for database setup and management.



2. **Create a Cluster:** Utilizing the MongoDB Atlas user interface, a new cluster was created by selecting the free tier option (M0). This tier allows users to experiment with MongoDB's capabilities without incurring costs, providing sufficient resources for preliminary development and testing. Clusters in MongoDB Atlas serve as the primary organizational units, housing the deployed databases and their respective collections.



3. **Configure Database Access:** To ensure data security and manage access permissions, a new database user with read and write privileges was created within the cluster's security settings. Granting read and write permissions is crucial for executing a wide range of database operations, including data insertion, updating, and querying, while also protecting the database from unauthorized access.

The screenshot shows the 'Connect to Cluster0' dialog box with a progress bar at the top indicating three steps: 1. Set up connection security, 2. Choose a connection method, and 3. Connect. Step 1 is completed, and Step 2 is the current step. The dialog box contains a warning message: 'Set up your user security permission below.' Below this, it states: 'This first user will have atlasAdmin permissions for this project. You'll need your database user's credentials in the next step. We autogenerated a username and password. You can use this or create your own.' There are two input fields: 'Username' with the value 'javierastorgo00' and 'Password' with the value 'Ay4ktwvd0jFndgsY'. A 'Copy' button is next to the password field. A 'Create Database User' button is at the bottom left. At the bottom right, there are 'Cancel' and 'Choose a connection method' buttons.

4. **Connect to a Cluster:** With the cluster set up, connectivity was established via the "Connect" button. This facilitated connection through the Data Explorer tool, an integral feature of MongoDB Atlas that enables users to navigate and manipulate their collections directly from the web interface. This tool provides a visual and interactive means of managing databases, eliminating the need for command-line interactions for database exploration.

The screenshot shows the 'Connect to Cluster0' dialog box with a progress bar at the top indicating three steps: 1. Set up connection security, 2. Choose a connection method, and 3. Connect. Steps 1 and 2 are completed, and Step 3 is the current step. The dialog box contains a warning message: 'You can't connect yet. Set up your user security permission in the first step.' Below this, there is a 'Data Explorer' section with a description: 'Use the browser based interface to manage databases, collections, and indexes in your deployment.' and an 'Open Data Explorer' button. There is also an 'Example Queries' section with two example queries: '{ <field>: <value>, ... }' and '{genres : [\"Drama\", \"Fantasy\"]}'. At the bottom, there is a 'RESOURCES' section with links: 'Query Data in Atlas', 'Access your Database Users', and 'Troubleshoot Connections'. At the bottom left, there is a 'Go Back' button. At the bottom right, there are 'Close' and 'Review setup steps' buttons.

5. **Create the Database and Collection:** The final preparatory step involved creating a new database named "members" and within it, a collection named "people". This was accomplished through the 'Browse Collections' section, employing the Data Explorer tool for intuitive database and collection setup. Following the creation of the collection, data insertion was performed using the Insert Document dialog, enabling the storage of specific data records into the MongoDB database.

Create Database

Database name [?]
members

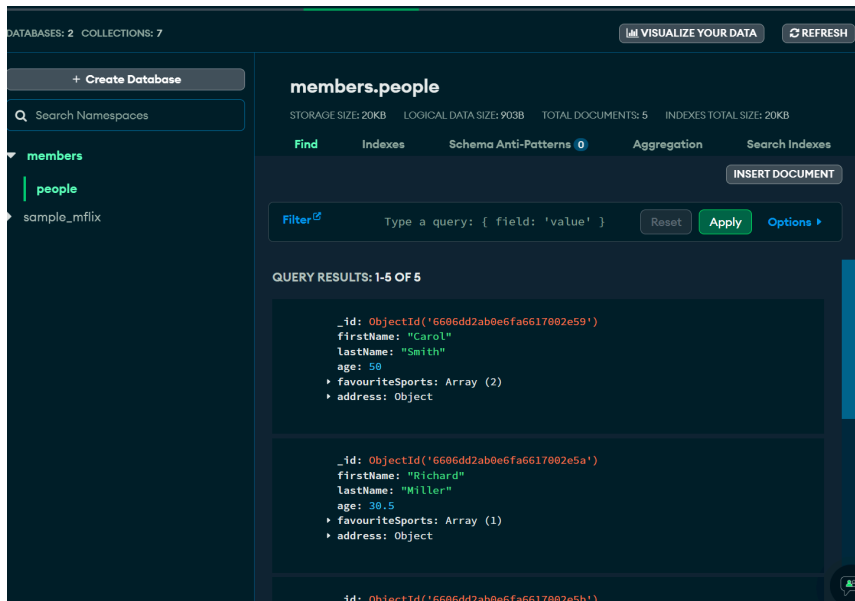
Collection name [?]
people

Additional Preferences
Select

Cancel Create

The data was prepared in JSON (JavaScript Object Notation) format, which is a lightweight data interchange format easily readable by humans and machines. JSON is particularly well-suited for NoSQL databases like MongoDB that store data in BSON format, a binary representation of JSON. Each entry, or document, encapsulates the data for an individual, including their first name, last name, age, favourite sports, and address details. Here's how the documents may look in JSON format for insertion:

```
[
  {
    "firstName": "Carol",
    "lastName": "Smith",
    "age": 50,
    "favouriteSports": ["Tennis", "Badminton"],
    "address": {"city": "Dublin", "street": "Abbey Rd.", "number": 1}
  },
  {
    "firstName": "Richard",
    "lastName": "Miller",
    "age": 30.5,
    "favouriteSports": ["Tennis"],
    "address": {"city": "Cork", "street": "Temple St.", "number": null}
  },
  {
    "firstName": "Thomas",
    "lastName": "Garcia",
    "age": 55,
    "favouriteSports": ["Football"],
    "address": {"city": "Galway", "street": "Henry St.", "number": 3}
  },
  // Additional documents truncated for brevity
];
```



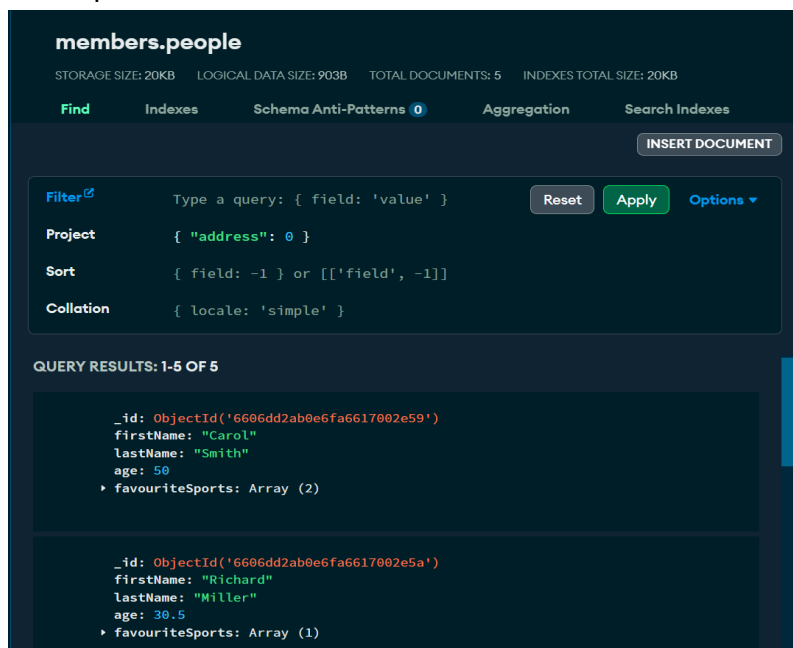
By following these steps, a fully-configured database environment was established on MongoDB Atlas, with a newly created database and collection primed for storing and managing data.

Tasks

To address the tasks outlined, the capabilities of the Filter tab in MongoDB Atlas were utilized, leveraging options for projection, sorting, and collation to execute specific database queries and manipulations. Here's how each task was approached based on the defined requirements:


Task 1: Excluding Address Information

By setting a projection with { "address": 0 } and an empty filter {}, all document fields except for the address were displayed. This projection effectively excludes the address details from the output.



Task 2: Querying Carol's Age

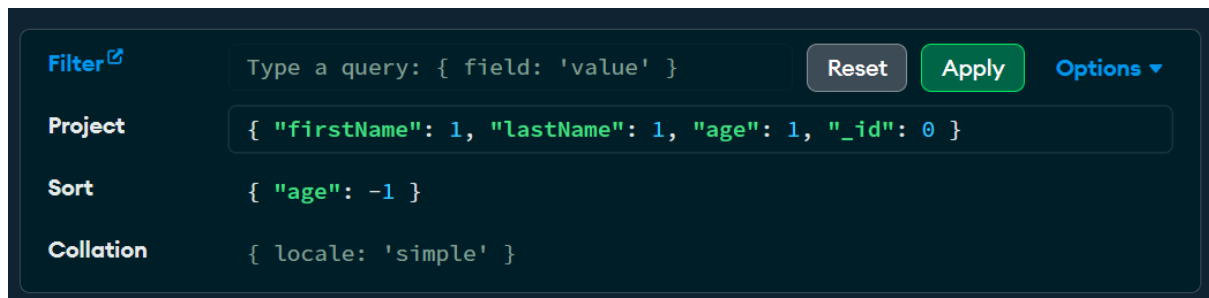
A filter `{ "firstName": "Carol" }` was applied to identify documents matching the first name "Carol," and a projection `{ "age": 1, "_id": 0 }` was used to return only Carol's age while excluding the document ID.



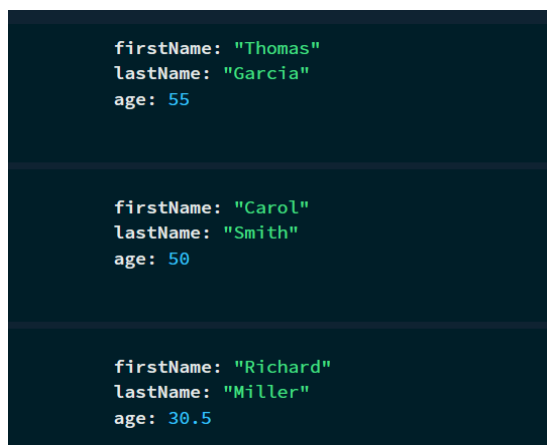
The screenshot shows the MongoDB Compass interface. At the top, there are tabs: Find, Indexes, Schema Anti-Patterns (0), Aggregation, and Search Indexes. The 'Find' tab is active. On the right, there is a button labeled 'INSERT DOCUMENT'. Below this, there are four sections: Filter, Project, Sort, and Collation. The Filter section contains the JSON `{ "firstName": "Carol" }` and buttons for 'Reset', 'Apply', and 'Options'. The Project section contains the JSON `{ "age": 1, "_id": 0 }`. The Sort section contains the JSON `{ field: -1 } or [['field', -1]]`. The Collation section contains the JSON `{ locale: 'simple' }`. Below these sections, it says 'QUERY RESULTS: 1-1 OF 1'. The result is displayed in a box: `age: 50`.

Task 3: Sorting by Age in Descending Order

The entire collection was considered with an empty filter `{}`, while the sort option `{ "age": -1 }` sorted the documents by age in descending order. A projection `{ "firstName": 1, "lastName": 1, "age": 1, "_id": 0 }` specified that only the names and ages of individuals should be displayed, excluding their MongoDB ID.



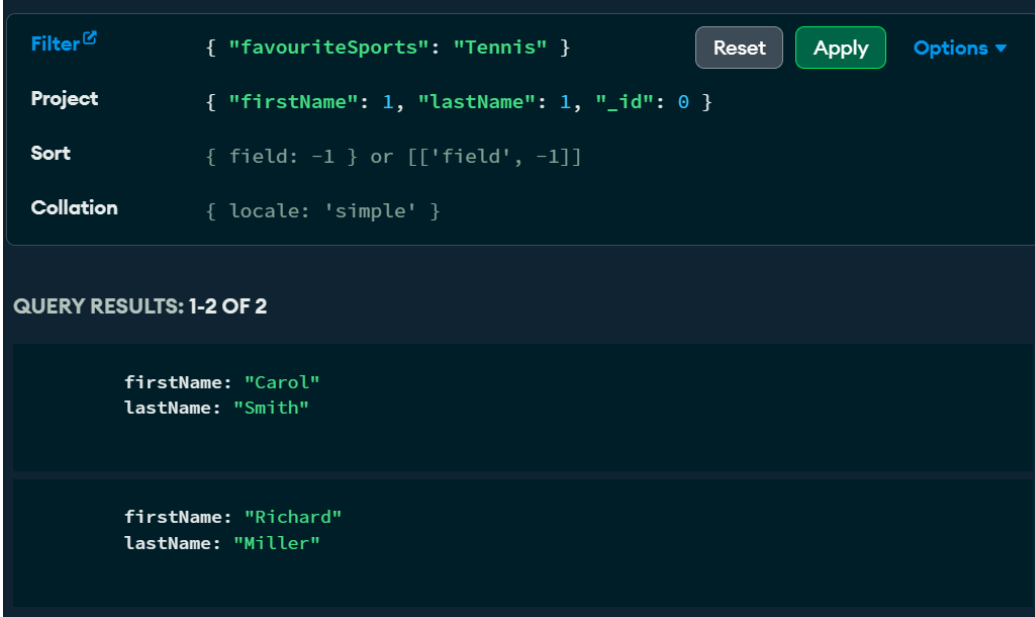
The screenshot shows the MongoDB Compass interface. At the top, there are tabs: Find, Indexes, Schema Anti-Patterns (0), Aggregation, and Search Indexes. The 'Find' tab is active. On the right, there is a button labeled 'INSERT DOCUMENT'. Below this, there are four sections: Filter, Project, Sort, and Collation. The Filter section contains the text 'Type a query: { field: 'value' }' and buttons for 'Reset', 'Apply', and 'Options'. The Project section contains the JSON `{ "firstName": 1, "lastName": 1, "age": 1, "_id": 0 }`. The Sort section contains the JSON `{ "age": -1 }`. The Collation section contains the JSON `{ locale: 'simple' }`.



The screenshot shows the query results for Task 3. The results are displayed in a list of three documents, sorted by age in descending order. Each document is shown in a box with its fields and values: `firstName: "Thomas"`, `lastName: "Garcia"`, `age: 55`; `firstName: "Carol"`, `lastName: "Smith"`, `age: 50`; and `firstName: "Richard"`, `lastName: "Miller"`, `age: 30.5`.

Task 4: Identifying Tennis Enthusiasts

Using a filter { "favouriteSports": "Tennis" }, documents where tennis is listed under favourite sports were identified. A projection { "firstName": 1, "lastName": 1, "_id": 0 } ensured that only the first and last names were returned, excluding document IDs.



The screenshot shows the MongoDB Atlas query builder interface. The query is defined as follows:

- Filter:** { "favouriteSports": "Tennis" }
- Project:** { "firstName": 1, "lastName": 1, "_id": 0 }
- Sort:** { field: -1 } or [['field', -1]]
- Collation:** { locale: 'simple' }

The query results are displayed below the query builder, showing two documents:

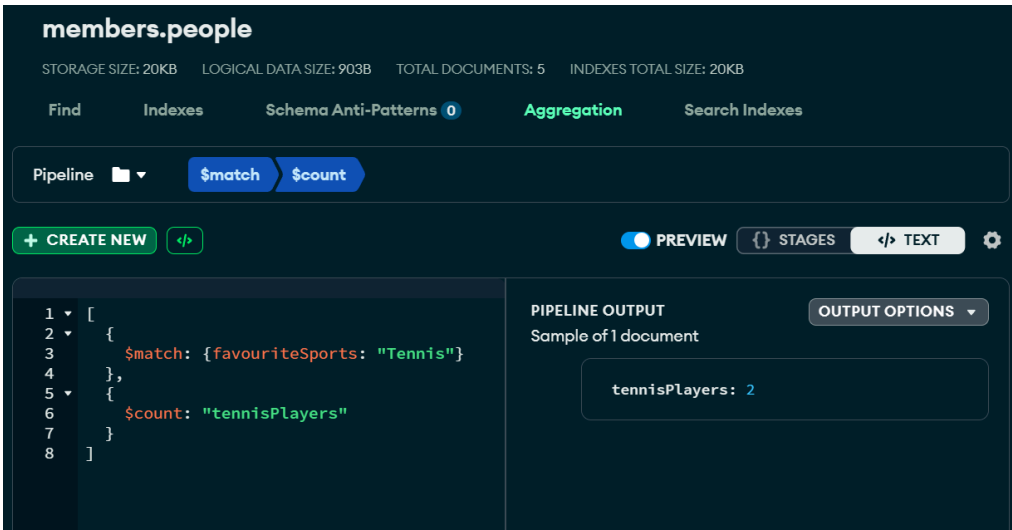
```
firstName: "Carol"
lastName: "Smith"

firstName: "Richard"
lastName: "Miller"
```

Task 5: Counting Tennis Fans

To address the task of counting members who favour tennis, more advanced aggregation operations were needed beyond simple filters and projections. Therefore, the 'Aggregation' feature of MongoDB Atlas was utilised, a powerful tool enabling the execution of data manipulation sequences known as aggregation pipelines. This approach facilitates complex data analysis and manipulation operations directly from the graphical interface of MongoDB Atlas.

The following aggregation pipeline was deployed to count the number of members who have a preference for tennis:



The screenshot shows the MongoDB Atlas aggregation pipeline interface for the 'members.people' collection. The pipeline is defined as follows:

```
1 [
2   {
3     $match: { favouriteSports: "Tennis" }
4   },
5   {
6     $count: "tennisPlayers"
7   }
8 ]
```

The pipeline output is displayed on the right, showing a sample of 1 document:

```
tennisPlayers: 2
```

- **\$match:** The first stage of the pipeline employs the \$match operator to filter documents in the database, only including those where the array favouriteSports

contains the value "Tennis". This step ensures that only relevant documents, pertaining to members who favour tennis, are considered in subsequent stages of the pipeline.

- **\$count:** The second stage applies the \$count operator, which proceeds to tally the total number of documents meeting the criteria established in the \$match stage. This count results in a specific tally of members for whom tennis is listed among their favourite sports. The outcome of the count is labelled as tennisPlayers, providing a descriptive name for the tally obtained.

To execute this aggregation pipeline in MongoDB Atlas, one must follow the outlined steps: access the corresponding cluster, navigate to the relevant collection, select the "Aggregations" button in the collection interface, and finally, enter and execute the pipeline in the provided aggregation editor.

Task 6: Locating Individuals Older Than 30

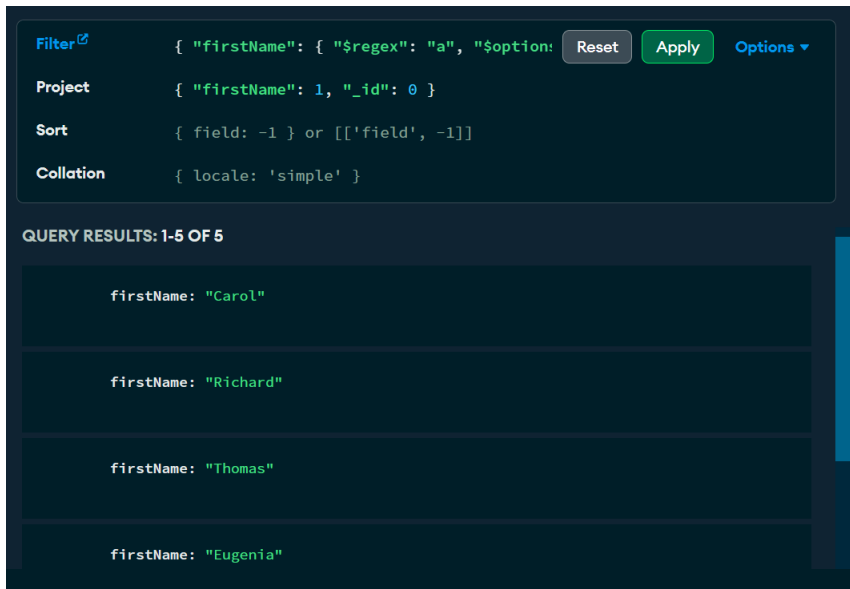
A filter { "age": { "\$gt": 30 } } was applied to select individuals over 30 years of age. With a projection { "firstName": 1, "lastName": 1, "age": 1, "_id": 0 }, only the names and ages were displayed, without the MongoDB document IDs.

The screenshot shows the MongoDB Atlas aggregation editor interface. It includes a 'Filter' stage with the query { "age": { "\$gt": 30 } }, a 'Project' stage with the query { "firstName": 1, "lastName": 1, "age": 1, "_id": 0 }, a 'Sort' stage with the query { field: -1 } or [['field', -1]], and a 'Collation' stage with the query { locale: 'simple' }. Below the editor, the query results are displayed, showing three documents with their first names, last names, and ages.

firstName	lastName	age
Carol	Smith	50
Richard	Miller	30.5
Thomas	Garcia	55

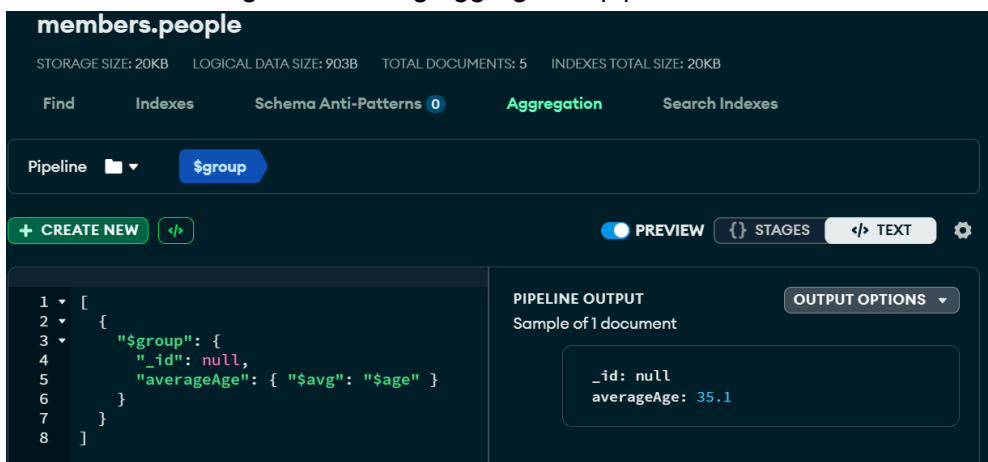
Task 7: Finding Names Containing 'a'

To find individuals whose first name contains the letter 'a', a case-insensitive regex filter { "firstName": { "\$regex": "a", "\$options": "i" } } was utilized. The projection { "firstName": 1, "_id": 0 } displayed only the first names, omitting the IDs.



Task 8: Calculating the Average Age of Everyone

The goal was to calculate the overall average age of individuals within the database. This was achieved using the following aggregation pipeline:



Here, `$group` aggregates all documents (denoted by `_id: null` to group them all together) and calculates the average age using `$avg`. This approach resulted in yielding the collective average age of all individuals in the database, offering insights into the demographic distribution.

Task 9: Calculating the Average Age of Tennis Enthusiasts

To find the average age of individuals who list tennis among their favourite sports, the following pipeline was utilized:

members.people
 STORAGE SIZE: 20KB LOGICAL DATA SIZE: 903B TOTAL DOCUMENTS: 5 INDEXES TOTAL SIZE: 20KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

Pipeline Pipeline ▾ \$match \$group

+ CREATE NEW `</>` PREVIEW {} STAGES `</>` TEXT ⚙️

```

1 [
2   {
3     "$match": { "favouriteSports": "Tennis" }
4   },
5   {
6     "$group": {
7       "_id": null,
8       "averageAge": { "$avg": "$age" }
9     }
10  }
11 ]
  
```

PIPELINE OUTPUT
Sample of 1 document

OUTPUT OPTIONS ▾

```

{
  "_id": null,
  "averageAge": 40.25
}
  
```

Initially, \$match filters the documents for those including "Tennis" in favouriteSports. Subsequently, \$group calculates the average age of the filtered documents. This method offers a means to understand the age distribution amongst tennis enthusiasts in the dataset.

Task 10: Counting Unique Cities Listed in the Collection

The task of counting unique cities within the collection was approached as follows:

members.people
 STORAGE SIZE: 20KB LOGICAL DATA SIZE: 903B TOTAL DOCUMENTS: 5 INDEXES TOTAL SIZE: 20KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

Pipeline Pipeline ▾ \$group \$count

+ CREATE NEW `</>` PREVIEW {} STAGES `</>` TEXT ⚙️

```

1 [
2   {
3     "$group": {
4       "_id": "$address.city"
5     }
6   },
7   {
8     "$count": "uniqueCities"
9   }
10 ]
  
```

PIPELINE OUTPUT
Sample of 1 document

OUTPUT OPTIONS ▾

```

{
  "uniqueCities": 4
}
  
```

This pipeline first groups documents by the city, resulting in a unique group for each city. \$count then provides the total number of unique groups, which translates to the count of unique cities within the collection. This approach gives an overview of the geographical diversity of the dataset's individuals without needing to manually parse through each document.

Question 3 - Reflection on changes on the Big Data landscape

After looking at the main points of the two articles about big data, it's clear they have different opinions on the role and future of big data.

The first article suggests that the excitement over big data might be over. It talks about the difficulties in managing a lot of data, like keeping old data that may not be useful anymore and asks if we really need to save so much information. It hints at the idea that maybe focusing on smaller, more relevant data sets could be more beneficial.

On the other hand, the second article believes big data still has a big part to play. It argues that big data has changed over time and continues to be essential for finding new insights and innovations. This article suggests that big data tools have evolved and can cater to different needs, whether it's analyzing small or large amounts of data.

These viewpoints highlight an ongoing discussion about big data. While one views it as becoming less important, the other sees its lasting value in helping us understand and use vast amounts of information. This debate reflects broader trends in technology, particularly in artificial intelligence and machine learning, where big data remains key to developing smarter systems.

In summary, the articles make us think about how we use data today. They suggest that perhaps the focus should shift from collecting massive amounts of data to understanding what this data can tell us and how it can be used more effectively.

END OF ASSIGNMENT

Javier Astorga