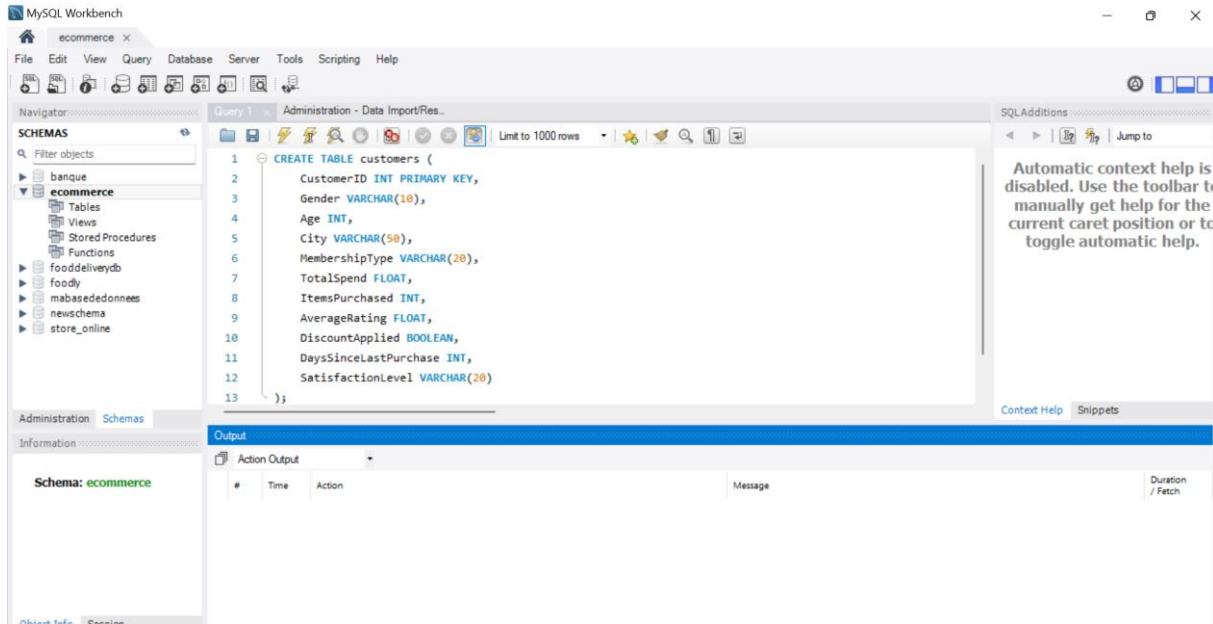


Comparaison SQL et NoSQL dans le cas d'usage de l'e-commerce

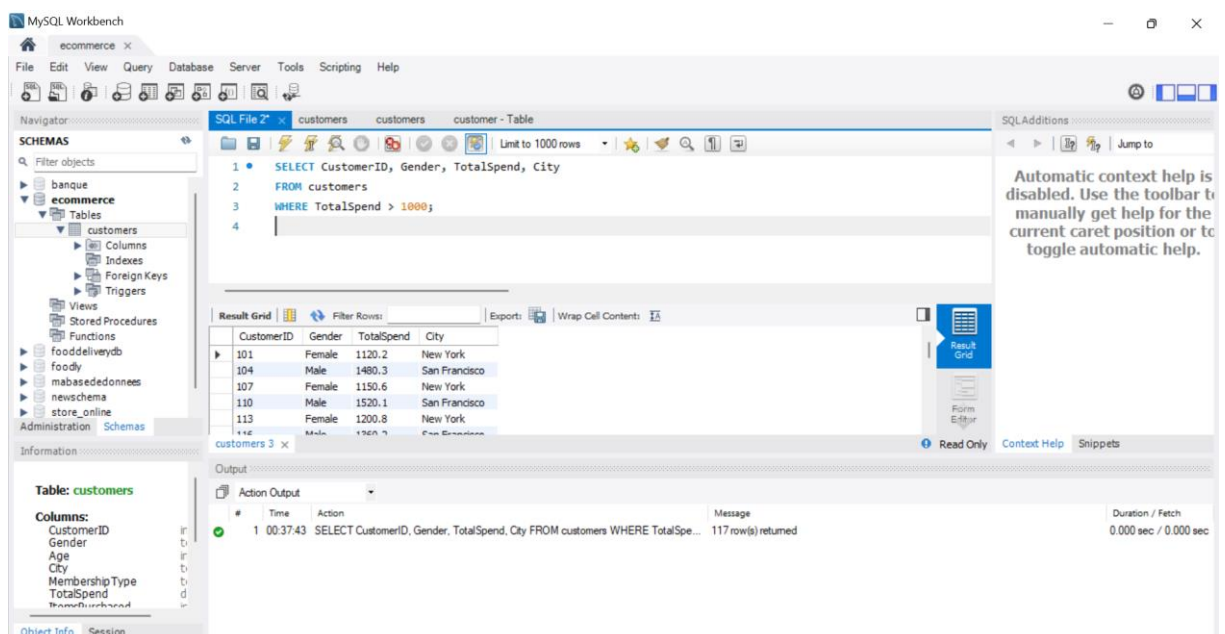
Création de la base de données et de la table :



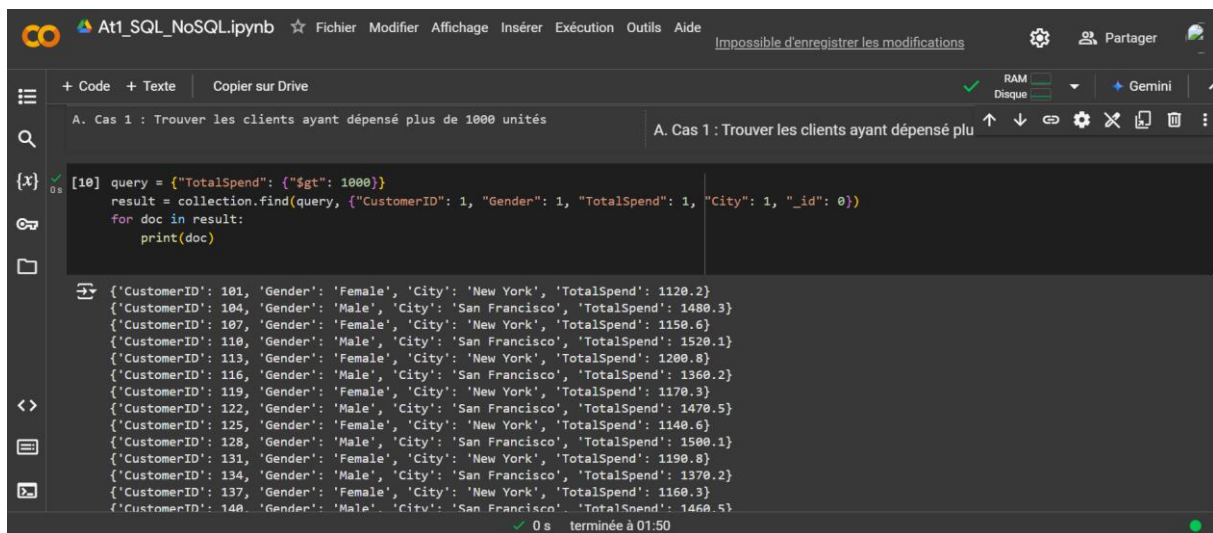
Requêtes et Comparaison SQL vs NoSQL

A. Cas 1 : Trouver les clients ayant dépensé plus de 1000 unités

1. SQL :



2. NoSQL (MongoDB) :



The screenshot shows a Jupyter Notebook with a code cell containing a MongoDB query. The query filters documents where 'TotalSpend' is greater than 1000. The output displays a list of 10 documents, each containing 'CustomerID', 'Gender', 'City', and 'TotalSpend'.

```
[10] query = {"TotalSpend": {"$gt": 1000}}
result = collection.find(query, {"CustomerID": 1, "Gender": 1, "TotalSpend": 1, "City": 1, "_id": 0})
for doc in result:
    print(doc)
```

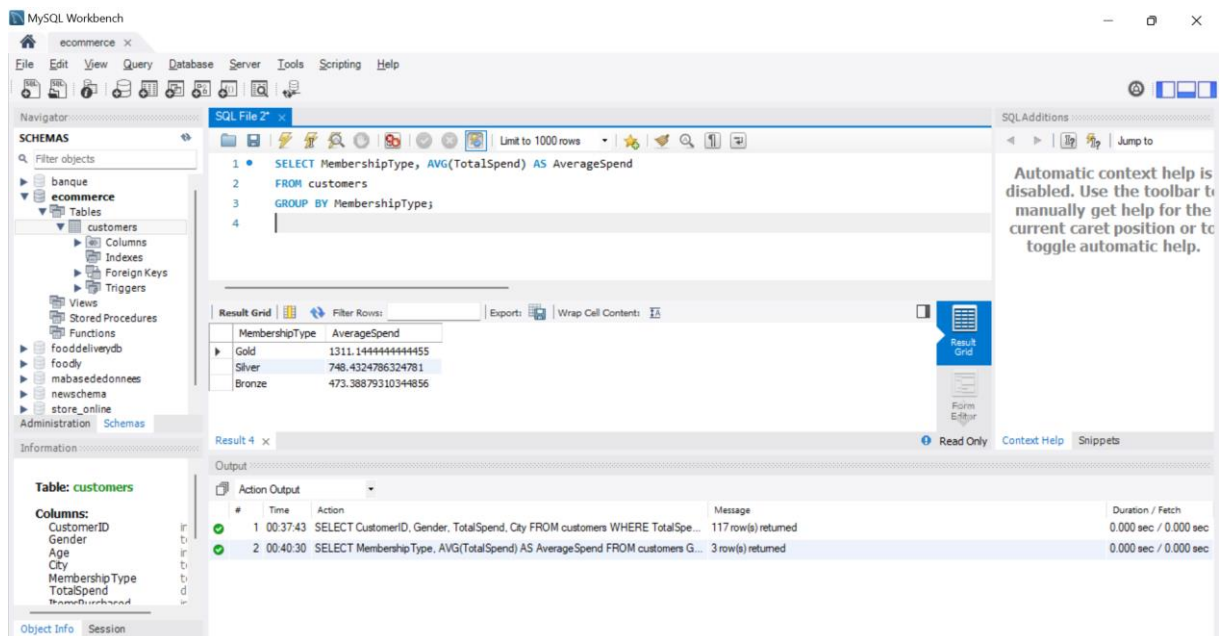
```
{'CustomerID': 101, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1120.2}
{'CustomerID': 104, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1480.3}
{'CustomerID': 107, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1150.6}
{'CustomerID': 110, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1520.1}
{'CustomerID': 113, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1200.8}
{'CustomerID': 116, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1360.2}
{'CustomerID': 119, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1170.3}
{'CustomerID': 122, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1470.5}
{'CustomerID': 125, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1140.6}
{'CustomerID': 128, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1500.1}
{'CustomerID': 131, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1190.8}
{'CustomerID': 134, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1370.2}
{'CustomerID': 137, 'Gender': 'Female', 'City': 'New York', 'TotalSpend': 1160.3}
{'CustomerID': 140, 'Gender': 'Male', 'City': 'San Francisco', 'TotalSpend': 1460.5}
```

Différence :

- SQL excelle dans la lisibilité et la gestion des relations complexes.
- NoSQL offre une syntaxe flexible mais moins intuitive pour des requêtes simples.

B. Cas 2 : Calculer la dépense moyenne par type d'abonnement

1. SQL :



The screenshot shows MySQL Workbench with a query window containing a SQL query. The query calculates the average spend by membership type. The results are displayed in a table with two columns: 'MembershipType' and 'AverageSpend'.

```
1 SELECT MembershipType, AVG(TotalSpend) AS AverageSpend
2 FROM customers
3 GROUP BY MembershipType;
```

MembershipType	AverageSpend
Gold	1311.1444444444455
Silver	748.4324786324781
Bronze	473.38879310344856

2. NoSQL (MongoDB) :

B. Cas 2 : Calculer la dépense moyenne par type d'abonnement

```

1 s pipeline = [
  {"$group": {"_id": "$MembershipType", "AverageSpend": {"$avg": "$TotalSpend"}}}]
result = collection.aggregate(pipeline)
for doc in result:
  print(doc)

```

```

{'_id': 'Gold', 'AverageSpend': 1311.1444444444444}
{'_id': 'Bronze', 'AverageSpend': 473.3887931034483}
{'_id': 'Silver', 'AverageSpend': 748.4324786324787}

```

Différence :

- SQL est optimisé pour ce type d'agrégations grâce à GROUP BY.
- MongoDB nécessite un pipeline, ce qui peut être moins direct mais offre une grande flexibilité.

C. Cas 3 : Identifier les clients insatisfaits

1. SQL :

MySQL Workbench interface showing a query to find dissatisfied customers. The query is:

```

1 SELECT CustomerID, Gender, SatisfactionLevel
2 FROM customers
3 WHERE SatisfactionLevel = 'Unsatisfied';
4

```

The result grid shows the following data:

CustomerID	Gender	SatisfactionLevel
103	Female	Unsatisfied
105	Male	Unsatisfied
109	Female	Unsatisfied
111	Male	Unsatisfied
115	Female	Unsatisfied

The output pane shows the execution of the query and a message indicating 116 rows returned.

2.

NoSQL (MongoDB) :

C. Cas 3 : Identifier les clients insatisfaits

```

0 s query = {"SatisfactionLevel": "Unsatisfied"}
result = collection.find(query, {"CustomerID": 1, "Gender": 1, "SatisfactionLevel": 1, "_id": 0})
for doc in result:
  print(doc)

```

```

{'CustomerID': 277, 'Gender': 'Female', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 279, 'Gender': 'Male', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 283, 'Gender': 'Female', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 285, 'Gender': 'Male', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 289, 'Gender': 'Female', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 291, 'Gender': 'Male', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 295, 'Gender': 'Female', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 297, 'Gender': 'Male', 'SatisfactionLevel': 'Unsatisfied'}
{'CustomerID': 302, 'Gender': 'Female', 'SatisfactionLevel': 'Unsatisfied'}

```

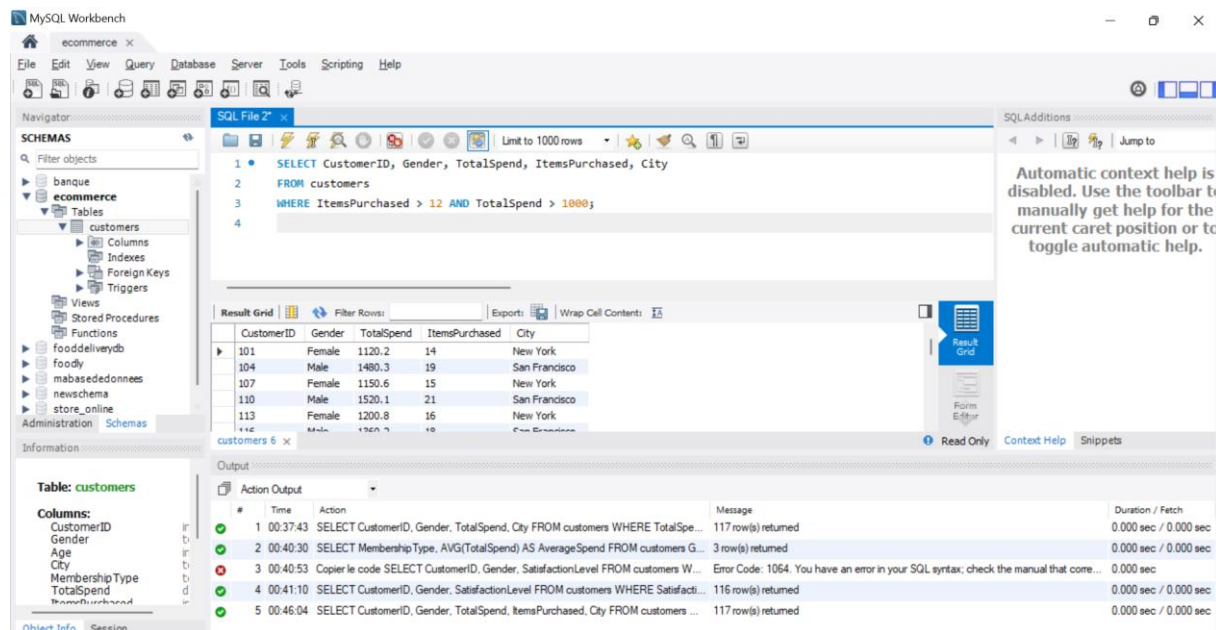
0 s terminée à 01:53

Différence :

Les deux systèmes gèrent efficacement ce type de requête simple. Cependant, SQL est souvent préféré pour des requêtes conditionnelles claires.

D Cas 4 : Trouver les clients ayant acheté plus de 12 articles ET dépensé plus de 1000 unités

SQL :



The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

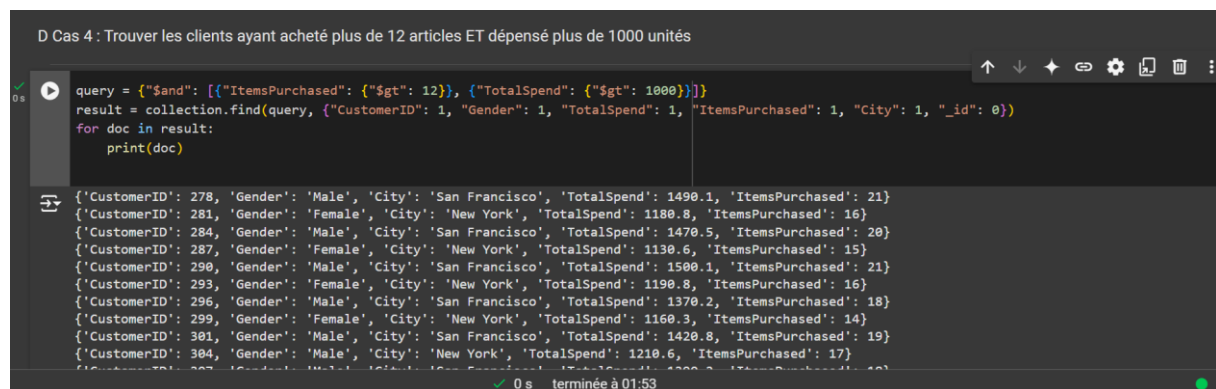
```
1 SELECT CustomerID, Gender, TotalSpend, ItemsPurchased, City
2 FROM customers
3 WHERE ItemsPurchased > 12 AND TotalSpend > 1000;
4
```

The Results grid shows 6 rows of data:

CustomerID	Gender	TotalSpend	ItemsPurchased	City
101	Female	1120.2	14	New York
104	Male	1480.3	19	San Francisco
107	Female	1150.6	15	New York
110	Male	1520.1	21	San Francisco
113	Female	1200.8	16	New York
116	Male	1560.7	18	San Francisco

The Output tab shows the execution log with 5 actions, all successful, returning 117 rows.

NoSQL (MongoDB) :



The screenshot shows the MongoDB Shell interface. The query is:

```
query = {"$and": [{"ItemsPurchased": {"$gt": 12}}, {"TotalSpend": {"$gt": 1000}}]}
result = collection.find(query, {"CustomerID": 1, "Gender": 1, "TotalSpend": 1, "ItemsPurchased": 1, "City": 1, "_id": 0})
for doc in result:
  print(doc)
```

The output shows 10 documents, each with fields: CustomerID, Gender, City, TotalSpend, and ItemsPurchased. The results are identical to the SQL query results.

Comparaison :

- SQL gère facilement les requêtes multi-conditionnelles avec une syntaxe intuitive.
- MongoDB est tout aussi performant, mais sa syntaxe peut paraître plus complexe.

E Cas 5 : Calculer la satisfaction moyenne par ville

SQL :

The screenshot shows the MySQL Workbench interface. The SQL Editor contains the following query:

```

3      WHEN SatisfactionLevel = 'Neutral' THEN 3
4      WHEN SatisfactionLevel = 'Unsatisfied' THEN 1
5      END) AS AverageSatisfaction
6 FROM customers
7 GROUP BY City;
8

```

The Results Grid shows the output of the query:

City	AverageSatisfaction
New York	5.0000
Los Angeles	3.2712
Chicago	1.0000
San Francisco	5.0000
Miami	1.0000

The Action Output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
2	00:40:30	SELECT MembershipType, AVG(TotalSpend) AS AverageSpend FROM customers ...	3 row(s) returned	0.000 sec / 0.000 sec
3	00:40:53	Copier le code SELECT CustomerID, Gender, SatisfactionLevel FROM customers ...	Error Code: 1064. You have an error in your SQL syntax; check the manual that cor...	0.000 sec
4	00:41:10	SELECT CustomerID, Gender, SatisfactionLevel FROM customers WHERE Satisf...	116 row(s) returned	0.000 sec / 0.000 sec
5	00:46:04	SELECT CustomerID, Gender, TotalSpend, ItemsPurchased, City FROM customers ...	117 row(s) returned	0.000 sec / 0.000 sec
6	00:46:41	SELECT City, AVG(CASE WHEN SatisfactionLevel = 'Satisfied' THEN 5...	6 row(s) returned	0.000 sec / 0.000 sec

NoSQL (MongoDB) :

The screenshot shows the MongoDB Shell interface. The pipeline being executed is:

```

pipeline = [
  { '$project': {
    'city': 1,
    'satisfactionScore': {
      '$switch': {
        'branches': [
          { 'case': { '$eq': ['$SatisfactionLevel', 'Satisfied'] }, 'then': 5 },
          { 'case': { '$eq': ['$SatisfactionLevel', 'Neutral'] }, 'then': 3 },
          { 'case': { '$eq': ['$SatisfactionLevel', 'Unsatisfied'] }, 'then': 1 }
        ],
        'default': 0
      }
    }
  } },
  { '$group': { '_id': '$city', 'AverageSatisfaction': { '$avg': '$satisfactionScore' } } }
]
result = collection.aggregate(pipeline)
for doc in result:
  print(doc)

```

The output of the aggregation is:

```

{ "_id": "New York", "AverageSatisfaction": 5.0 }
{ "_id": "Los Angeles", "AverageSatisfaction": 3.27124179131 }
{ "_id": "Houston", "AverageSatisfaction": 2.89551724137931 }
{ "_id": "Chicago", "AverageSatisfaction": 1.0 }
{ "_id": "Miami", "AverageSatisfaction": 1.0 }
{ "_id": "San Francisco", "AverageSatisfaction": 5.0 }

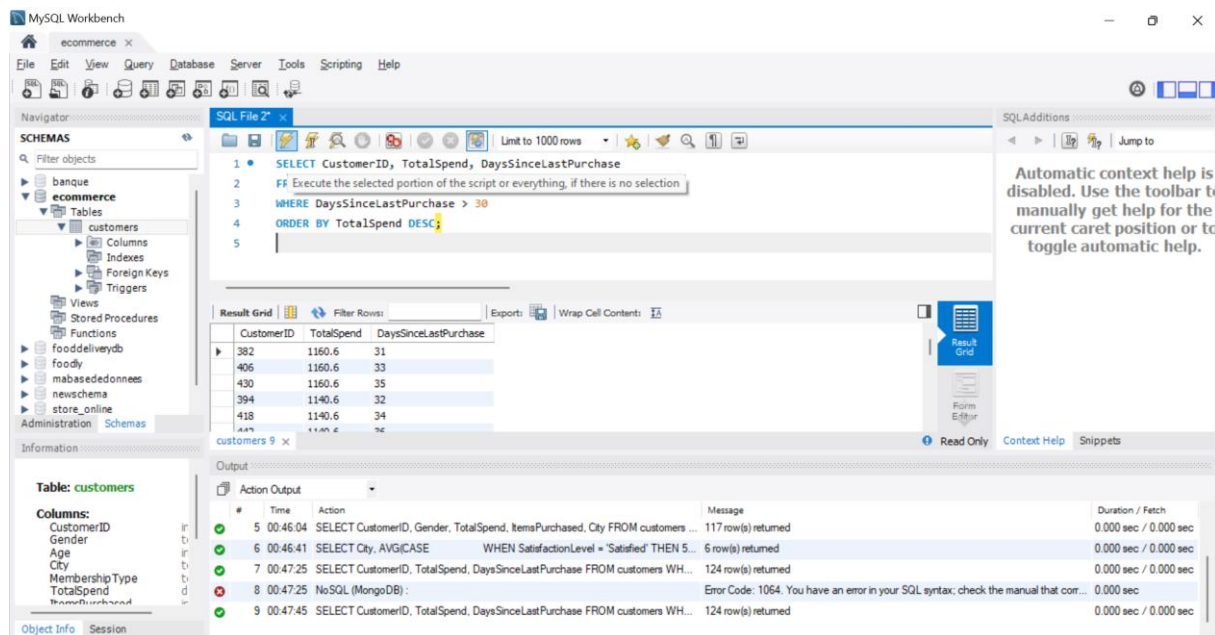
```

Comparaison :

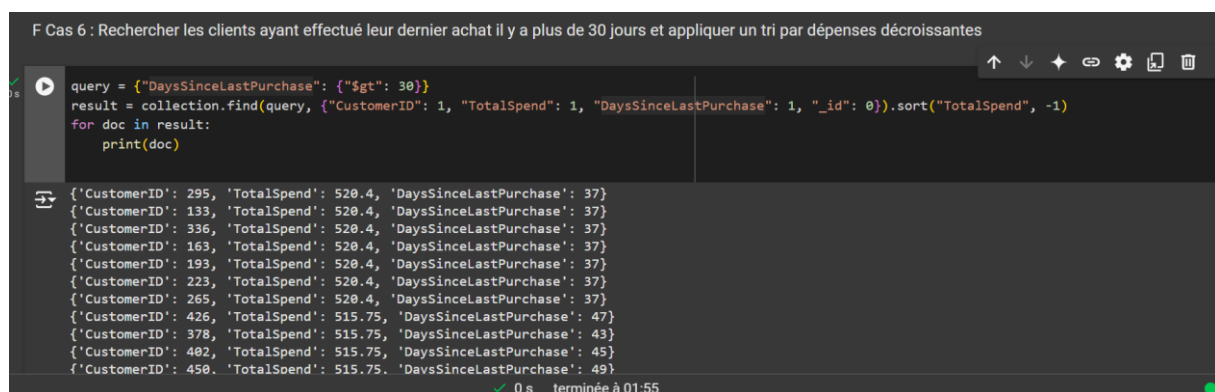
- SQL est extrêmement efficace avec des fonctions de calcul conditionnelles (CASE).
- MongoDB nécessite un pipeline plus complexe mais offre davantage de flexibilité pour des données non uniformes.

F Cas 6 : Rechercher les clients ayant effectué leur dernier achat il y a plus de 30 jours et appliquer un tri par dépenses décroissantes

SQL :



NoSQL :



Comparaison :

- SQL excelle dans le tri (ORDER BY) avec une syntaxe simple.
- MongoDB gère aussi efficacement le tri, mais il faut une étape supplémentaire pour ajouter la méthode .sort().

J Cas 7 : Segmenter les clients par tranche d'âge (moins de 30 ans, 30-40 ans, plus de 40 ans) et calculer la dépense moyenne par segment

SQL :

The screenshot shows the MySQL Workbench interface. On the left, the 'Schemas' pane shows a database named 'ecommerce' with tables 'customers', 'fooddeliverydb', 'foody', 'mabasededonnees', 'newschema', and 'store_online'. The 'customers' table is selected, showing columns: CustomerID, Gender, Age, City, MembershipType, and TotalSpend. The main window displays a SQL query in 'SQL File 2':

```

5      ELSE 'Above 40'
6      END AS AgeSegment,
7      AVG(TotalSpend) AS AverageSpend
8  FROM customers
9  GROUP BY AgeSegment;
10

```

Below the query, the 'Result Grid' shows the output:

AgeSegment	AverageSpend
Less than 30	1072.1840579710145
30-40	865.0659192825107
Above 40	499.8827586206897

At the bottom, the 'Output' pane shows the execution log with messages and durations for each step.

NoSQL (MongoDB) :

The screenshot shows the MongoDB Shell interface. The title bar reads 'J Cas 7 : Segmenter les clients par tranche d'âge (moins de 30 ans, 30-40 ans, plus de 40 ans) et calculer la dépense moyenne par segment'. The main window displays a MongoDB aggregation pipeline:

```

pipeline = [
  {"$project": {
    "AgeSegment": {
      "$cond": [
        {"$lt": [{"Age", 30}], "Less than 30",
        {"$cond": [{"$and": [{"$gte": [{"Age", 30}], {"$lte": [{"Age", 40}]}], "30-40", "Above 40"}]}
      ]
    },
    "TotalSpend": 1
  }},
  {"$group": {"_id": "$AgeSegment", "AverageSpend": {"$avg": "$TotalSpend"}}}
]
result = collection.aggregate(pipeline)
for doc in result:
  print(doc)

```

At the bottom, the output shows the results of the aggregation:

```

{'_id': 'Less than 30', 'AverageSpend': 1072.1840579710145}
{'_id': 'Above 40', 'AverageSpend': 499.8827586206897}

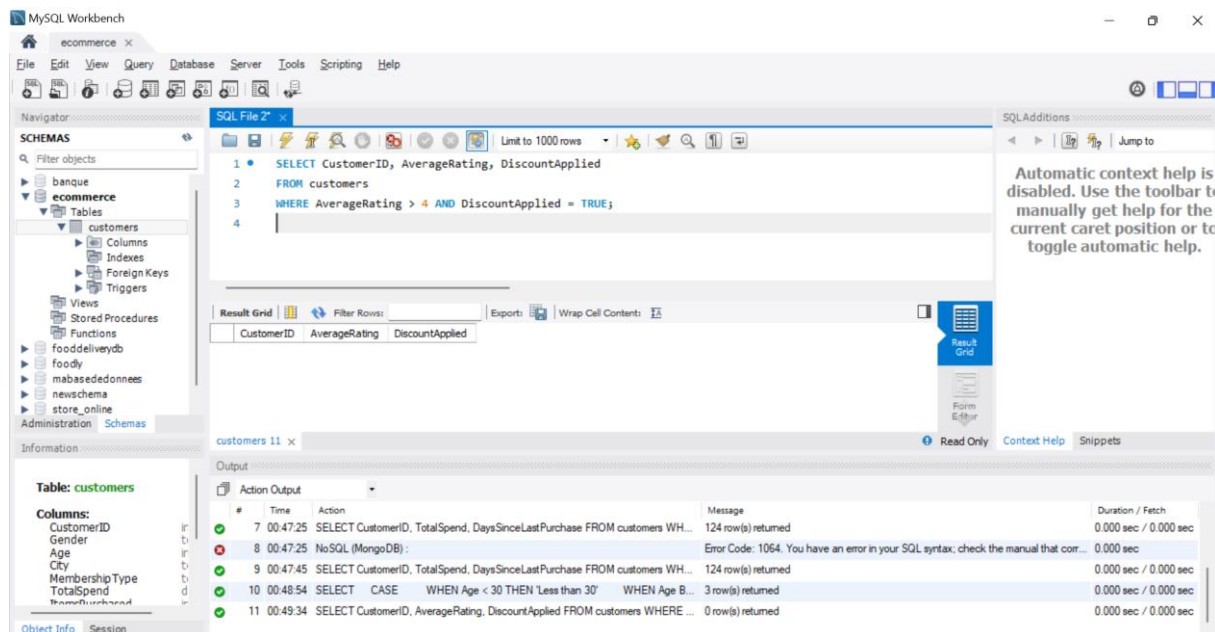
```

Comparaison :

- SQL est parfait pour créer des segments grâce à des blocs CASE.
- MongoDB peut le faire via un pipeline project et group, mais c'est plus verbeux.

H Cas 8 : Identifier les clients ayant une note moyenne supérieure à 4 ET ayant bénéficié d'une remise

SQL :



NoSQL (MongoDB) :



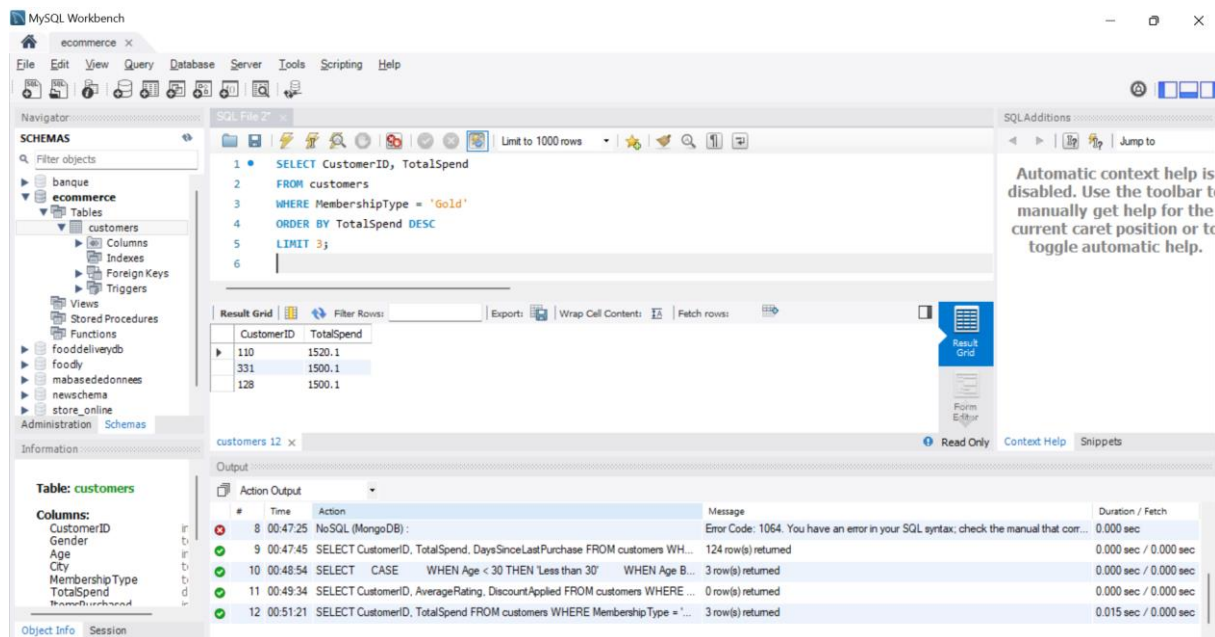
⇒ **There is no results**

Comparaison :

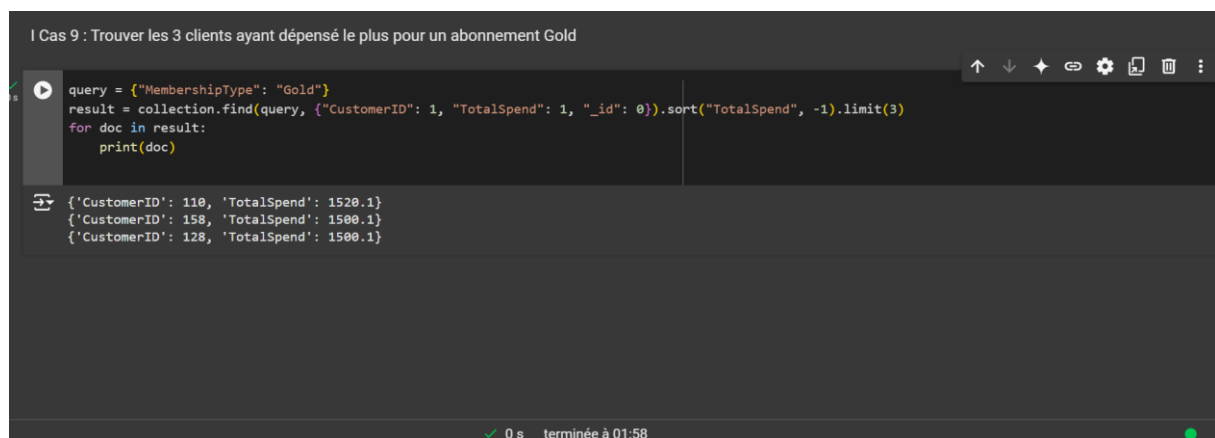
- Les deux systèmes gèrent efficacement cette requête simple.
- SQL reste plus lisible pour les utilisateurs habitués au langage relationnel.

I Cas 9 : Trouver les 3 clients ayant dépensé le plus pour un abonnement Gold

SQL :



NoSQL (MongoDB) :



Comparaison :

- SQL gère très efficacement le tri et la limitation des résultats grâce à ORDER BY et LIMIT.
- MongoDB nécessite une combinaison de .sort() et .limit(), qui est tout aussi performante.

Observations Globales

1. SQL excelle dans les requêtes impliquant des relations complexes, des agrégations analytiques et des transformations conditionnelles.
2. NoSQL brille dans la gestion des données non structurées et l'adaptabilité des schémas.
3. Une approche hybride est souvent la meilleure, exploitant les forces de chaque technologie en fonction des besoins.