Name: Calle, Paola Date: 09/18/2025

NYU ID: N18787288

Course Section Number: 2433–001

Assignment 1

Total in points (100 points total): ___________________

Professor's Comments:

Calle, Paola
Assignment # 1

DBMS Report

Selection

For this project, I selected four database management systems (DBMS) that represent different models: MySQL (relational), MongoDB (document-oriented NoSQL), Google BigQuery (analytical/columnar), and Neo4j (graph database). MySQL is included as a baseline due to its widespread use and long-standing role in relational database management. The others were chosen to showcase the contrast between traditional relational systems and specialized non-relational approaches.

Reason Behind Choice

To guide this selection, I consulted the DB-Engines ranking, a site that tracks database popularity and adoption. According to this ranking, MySQL holds the #2 position, MongoDB ranks #4, Google BigQuery is placed at #19, and Neo4j is ranked #20 (Engines). These positions highlight both the broad adoption of MySQL and MongoDB and the more specialized but important roles of BigQuery and Neo4j.

Capabilities & Inner-Works

MySQL (Relational DBMS)

- Capabilities: MySQL is best suited for structured data and transactional workflows. It enforces schemas and consistency, making it widely used for commerce platforms, financial systems, and other business applications.
- High Level Inner Workings:
    - Storage: Data is organized into rows and columns using the InnoDB engine.
    - Query Execution:
        - Parsing: Queries are checked for correct syntax.
        - Optimization: The database evaluates possible strategies and selects the cheapest/fastest plan.
        - Execution: The chosen plan is run to retrieve the results.
    - Indexing: B-tree indexes support most lookups and spatial R-tree indexes speed up geospatial queries.

    - Transactions: MySQL is fully ACID-compliant, using Multi-Version Concurrency Control (MVCC), undo/redo logs, and isolation levels (default: REPEATABLE READ).

MongoDB (Document-Oriented NoSQL)

- Capabilities: MongoDB excels with semi-structured or evolving data. It is commonly used for web applications, content platforms, and real-time analytics due to its flexible schema and rapid development capabilities.
- High Level Inner Workings:
    - Storage: Data is stored as BSON documents within collections, managed by the WiredTiger storage engine.
    - Query Execution:
        - Aggregation pipelines: Documents pass through sequential stages ($match, $group, $project, etc.) for filtering and transformation.
        - $lookup operator: Simulates joins by pulling in data from another collection.
    - Indexing: Supports secondary indexes including compound, text, geospatial, and wildcard.
    - Transactions: Provides multi-document ACID transactions with snapshot isolation.

Neo4j (Graph Database)

- Capabilities: Neo4j is optimized for relationship-heavy workloads such as fraud detection, recommendation engines, and social networks.
- High Level Inner Workings:
    - Storage: Uses a native property graph model, where both nodes and relationships can store properties.
    - Query Execution:
        - Cypher query language: A declarative language for expressing graph patterns, making relationship queries intuitive.
        - Index-free adjacency: Each node stores direct pointers to its neighbors, enabling constant-time (O(1)) traversals without expensive joins.
- Indexing: Indexes are mainly used to locate starting nodes. That is, traversals rely on adjacency pointers.
- Transactions: ACID-compliant with strong isolation (serializable).

Calle, Paola
Assignment # 1

Google BigQuery

- Capabilities: BigQuery is a cloud-native, serverless data warehouse designed for large-scale analytics and business intelligence. It excels at scanning terabytes or even petabytes of data quickly with minimal setup.
- High Level Inner Workings:
    - Storage:
        - Columnar format (Capacitor): Stores data by columns rather than rows, enabling efficient scans of selected fields.
        - Partitioned: Tables can be divided by keys (often dates) so queries only scan relevant partitions.
        - Compressed: Data is stored in compressed format to save space and speed up reads.
    - Query Execution:
        - Powered by Dremel, which organizes execution into a distributed tree of servers:
            - Root nodes coordinate queries.
            - Intermediate nodes merge partial results.
            - Leaf nodes scan the data in parallel.
        - Queries are divided into pieces (partitioning), processed by massively parallel slots, and results are merged.
    - Indexing: BigQuery does not use traditional indexes. Instead, it relies on partitioning and clustering to reduce the amount of data scanned.

Together, these four systems illustrate how database internals shape their ideal use cases:

- MySQL ensures strong consistency and schema enforcement for structured, transactional systems.
- MongoDB provides flexibility and scalability for semi-structured or rapidly evolving data.
- Neo4j offers unmatched performance for traversing complex relationships.
- BigQuery enables lightning-fast analytics across massive datasets by combining columnar storage with distributed query execution.

Calle, Paola
Assignment # 1

Comparison Table of Selected DBMS

	MySQL	**MongoDB**	**Neo4j**	**Google BigQuery**
Paradigm	Relational (SQL)	Document-oriented (NoSQL)	Graph database	Analytical / Columnar SQL warehouse
DB Engines rank	#2	#5	#20	#19
Best at	Traditional OLTP apps with strict consistency, joins, and schemas	Flexible JSON-like data, rapid schema evolution	Relationship-heavy queries, multi-hop traversals, recommendations	Large-scale analytics, BI dashboards, ad-hoc queries over TB–PB datasets, data warehousing
Storage Model	Tables of rows/columns (InnoDB default engine)	BSON documents in collections (WiredTiger engine)	Property graph: nodes + relationships with properties	Columnar storage (Capacitor), partitioned & compressed
Data Model Flexibility	Rigid schemas, strong typing	Flexible schema, semi-structured	Schema-optional (labels + properties)	Semi-structured (arrays, nested, JSON columns)
Query Or Execution	SQL parser, then cost-based optimizer, then execution plan	Aggregation pipelines, $lookup for joins	Cypher query language, index-free adjacency (O(1) traversals)	Dremel execution tree, massively parallel slots, ANSI SQL
Indexing	B-tree (default), hash, R-tree for spatial	Secondary indexes: compound, text, geospatial, wildcard	Indexes only for entry points; most traversal via pointers	No traditional indexes; instead partitioning & clustering
Concurrency Control	MVCC with undo logs + locks	Document-level concurrency via WiredTiger	Locks + transactional MVCC	Snapshot isolation for queries
Transactions or Consistency	Full ACID compliance; redo logs for durability	Multi-document ACID transactions	ACID-compliant, transactional graph ops	Multi-statement ACID transactions, snapshot consistency
Scaling	Vertical scaling, replication, clustering	Horizontal scaling with sharding	Clustering for HA & scale (AuraDB cloud)	Infinite horizontal scale (serverless)
Local Deployment	Yes	Yes	Yes	No
Cloud Deployment	Yes	Yes	Yes	Yes

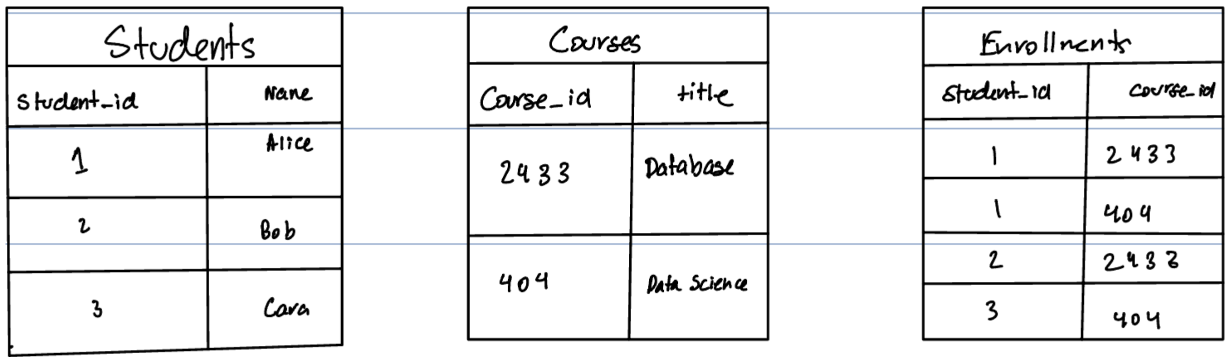
Calle, Paola
Assignment # 1

Use Case: Students and Course

To demonstrate the capabilities of these four different DBMS, we will use a simple, common example: the relationship between students and courses. By posing a few straightforward questions, we can highlight how each system approaches the same problem in different ways.

Data Models Drawn Out

MySQL - Everything is structured into tables with rows and columns, linked by foreign keys.



Students

student_id	Name
1	Alice
2	Bob
3	Cara

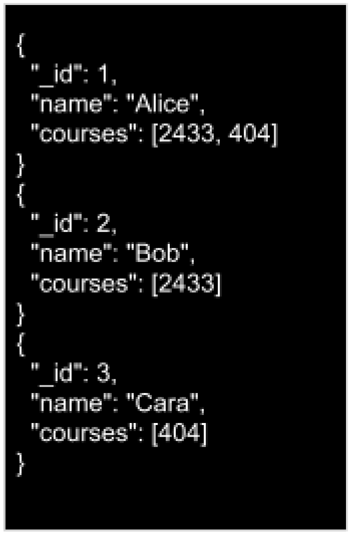
Courses

Course_id	title
2433	Database
404	Data Science

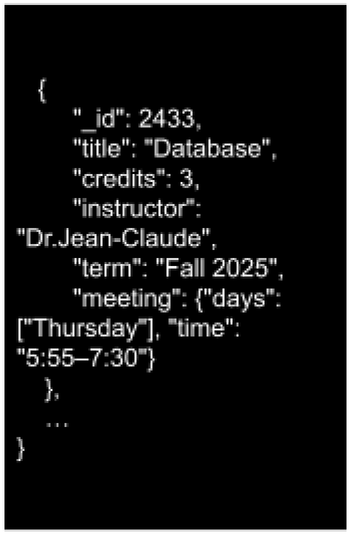
Enrollments

Student_id	course_id
1	2433
1	404
2	2433
3	404

MongoDB - Each student is a document, and courses are stored as arrays inside the student record.



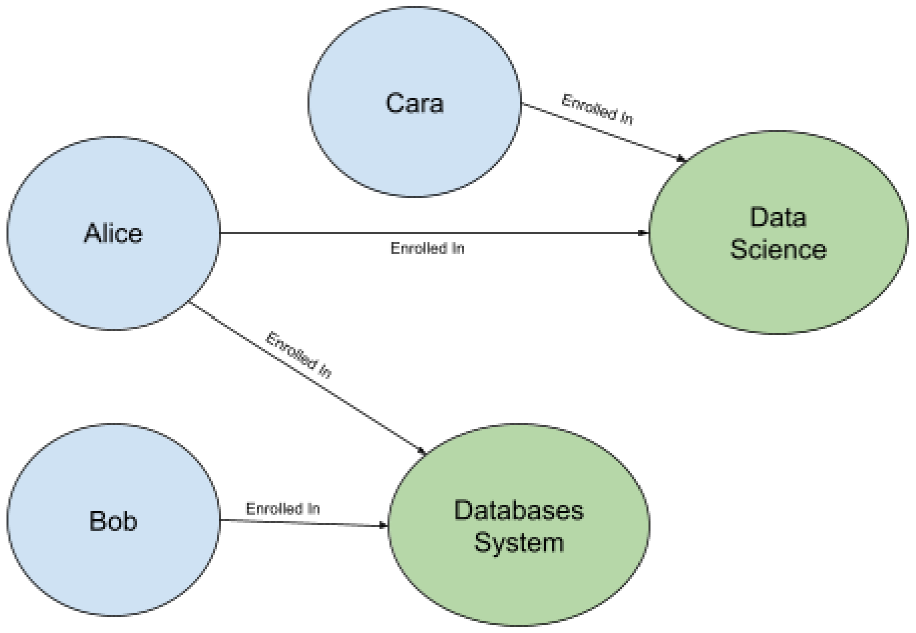
```
{
  "_id": 1,
  "name": "Alice",
  "courses": [2433, 404]
}
{
  "_id": 2,
  "name": "Bob",
  "courses": [2433]
}
{
  "_id": 3,
  "name": "Cara",
  "courses": [404]
}
```



```
  {
      "_id": 2433,
      "title": "Database",
      "credits": 3,
      "instructor":
"Dr.Jean-Claude",
      "term": "Fall 2025",
      "meeting": {"days":
["Thursday"], "time":
"5:55–7:30"}
    },
    ...
}
```

Calle, Paola
Assignment # 1

Neo4j (Graph) - Data is modeled as nodes (circles) and relationships (arrows).



Google BigQuery - BigQuery stores data in a single fact table that's partitioned/optimized for analytics. These MySQL diagram examples are very similar to BigQuery.

Calle, Paola
Assignment # 1

Setting Up Databases (Cloud + Local)

This section walks through setting up MongoDB, Neo4j, and Google BigQuery – both via their free cloud offerings and with local tooling. All scripts are provided.

MongoDB

1. Create a free cluster on Atlas
    a. Go to mongodb.com → Atlas → create a Free Cluster (pick any region).
    b. Under Network Access, temporarily allow your current IP.
    c. Under Database Access, create a DB user (e.g., appuser with a strong password).
    d. Copy the connection string.
2. . Set up your local tools
    a. Install the MongoDB shell (optional but handy): brew install mongosh
    b. Install the Python driver: pip install pymongo
3. Run the demo script
    a. Open paola_calle_mongo_db.py and replace the MONGO_URI value with your Atlas connection string.
    b. Run script: python mongo_school_with_classes.py

Neo4j

1. Create a free AuraDB instance
    a. Go to neo4j.com/cloud/platform/aura-graph-database
    b. Sign up (free tier available) and create an AuraDB Free instance.
    c. Once it's provisioned, note down:
        i. Bolt URI (e.g., neo4j+s://<id>.databases.neo4j.io)
        ii. Username (default: neo4j)
        iii. Password (you'll set this when the DB is created).
2. Set up your local tools
    a. Install the Neo4j driver: pip install neo4j
3. Run the demo script
    a. Open paola_calle_neo4j_auradb.py and
    b. Replace the NEO4J_URI, NEO4J_USER, and NEO4J_PASSWORD values with your AuraDB connection details.

    c. Run script: python neo4j_auradb_school_with_classes.py

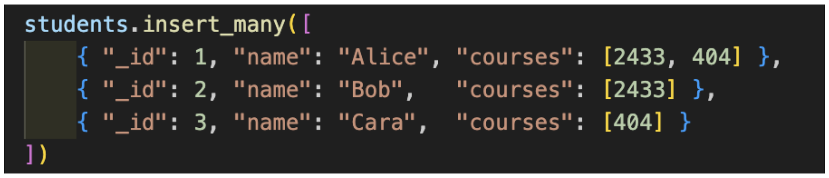
Google BigQuery

1. Create a Google Cloud Project
    a. Go to https://console.cloud.google.com/
    b. Click the project dropdown → New Project.
    c. Give it a name, leave organization as "No organization", click Create.
2. Enable BigQuery API
    a. In the search bar, type BigQuery API.
    b. Click it → Enable.
    c. This allows your project to use BigQuery.
3. Create a Service Account
    a. Go to IAM & Admin → Service Accounts.
    b. Click + Create Service Account.
    c. Give it a name.
    d. Click Create and Continue.
    e. Assign a role:
        i. BigQuery User (is recommended, but I did BigQuery Admin)
    f. Click Done.
4. Generate a JSON Key
    a. In the Service Accounts page, click your new account.
    b. Go to the Keys tab → Add Key → Create new key.
    c. Choose JSON → download the file.
        i. Save it somewhere secure.
5. Set up your local tools
    a. Install BigQuery client for Python: pip install google-cloud-bigquery
6. Free tier safety
    a. Storage: 10 GB per month free.
    b. Queries: 1 TB per month free.
    c. Stay within that and you won't be charged.
7. Run the demo script
    a. Open bigquery_school_with_classes.py
    b. Replace the PROJECT, DATASET, and LOCAL_KEY values
    c. Run script: python paola_calle_bigquery.py

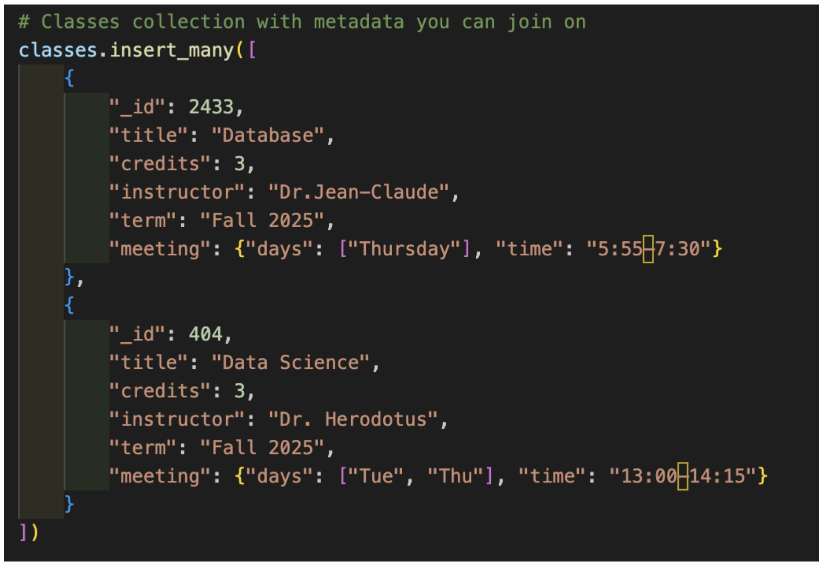
Calle, Paola
Assignment # 1

Data Model Implementation

MongoDB



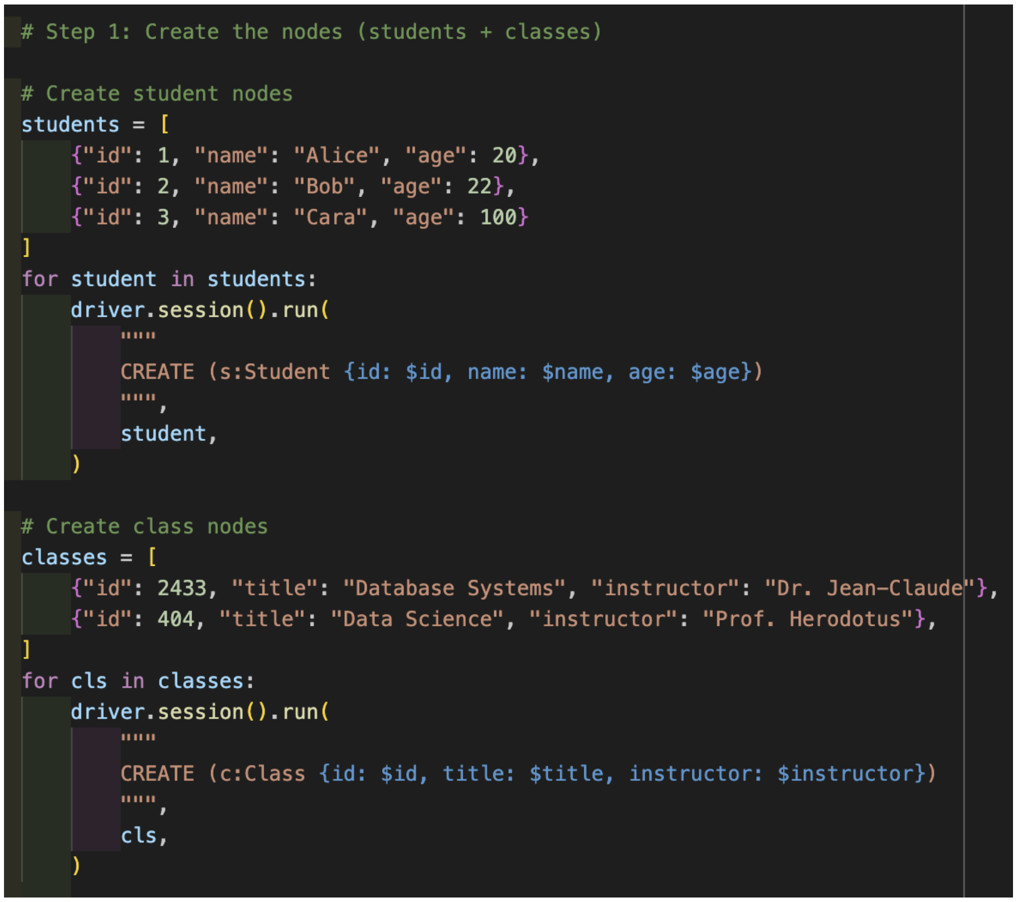
```
students.insert_many([
    { "_id": 1, "name": "Alice", "courses": [2433, 404] },
    { "_id": 2, "name": "Bob",   "courses": [2433] },
    { "_id": 3, "name": "Cara",  "courses": [404] }
])
```



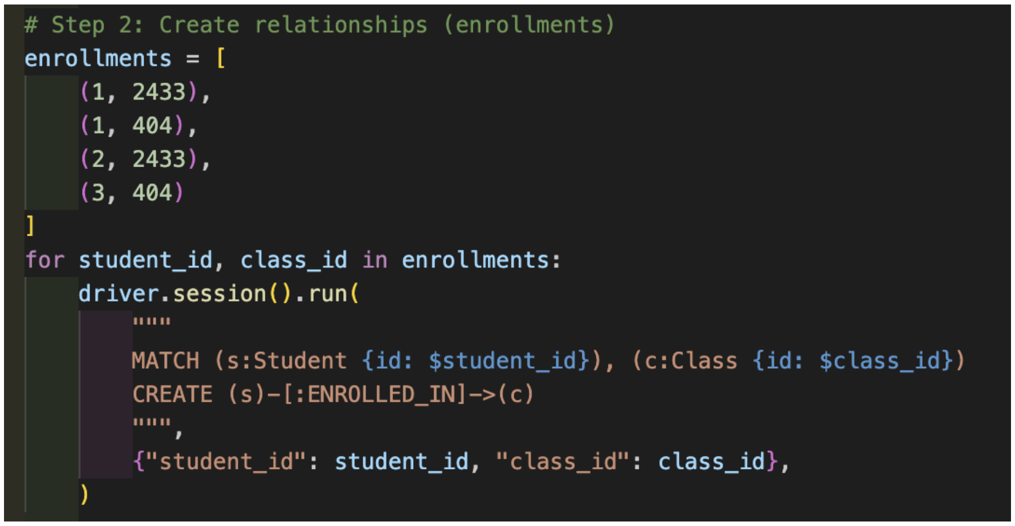
```
# Classes collection with metadata you can join on
classes.insert_many([
    {
        "_id": 2433,
        "title": "Database",
        "credits": 3,
        "instructor": "Dr.Jean-Claude",
        "term": "Fall 2025",
        "meeting": {"days": ["Thursday"], "time": "5:55–7:30"}
    },
    {
        "_id": 404,
        "title": "Data Science",
        "credits": 3,
        "instructor": "Dr. Herodotus",
        "term": "Fall 2025",
        "meeting": {"days": ["Tue", "Thu"], "time": "13:00–14:15"}
    }
])
```

Calle, Paola
Assignment # 1

Neo4j



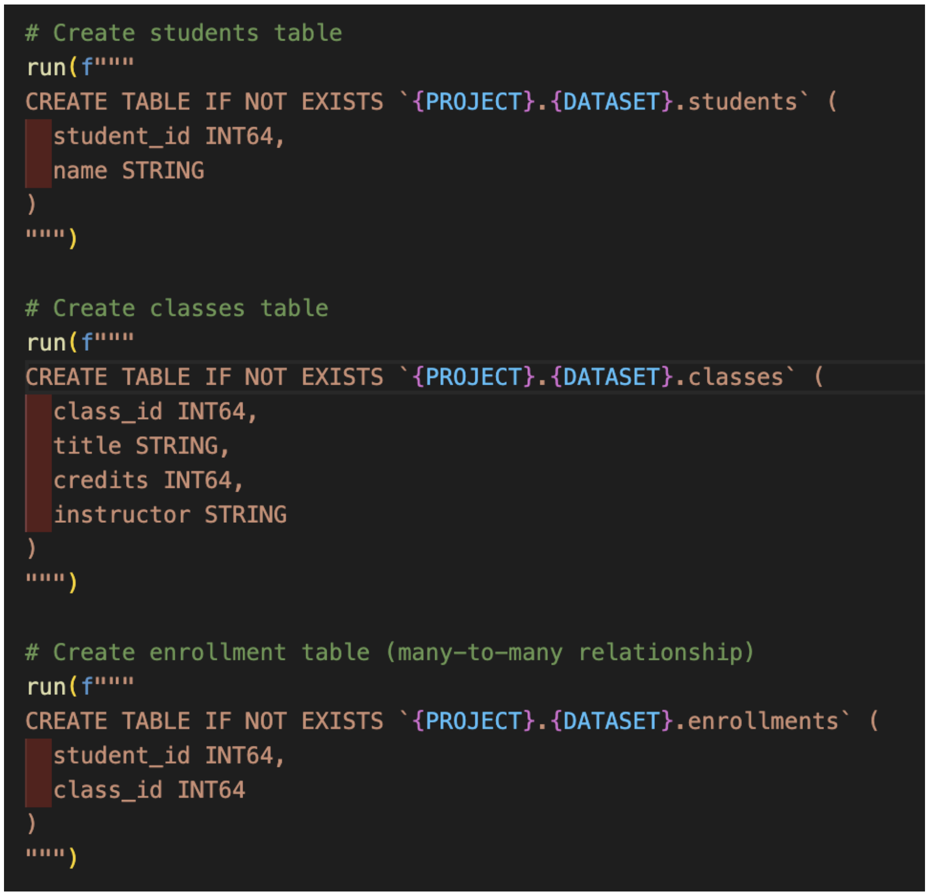
```python
# Step 1: Create the nodes (students + classes)

# Create student nodes
students = [
    {"id": 1, "name": "Alice", "age": 20},
    {"id": 2, "name": "Bob", "age": 22},
    {"id": 3, "name": "Cara", "age": 100}
]
for student in students:
    driver.session().run(
        """
        CREATE (s:Student {id: $id, name: $name, age: $age})
        """,
        student,
    )

# Create class nodes
classes = [
    {"id": 2433, "title": "Database Systems", "instructor": "Dr. Jean-Claude"},
    {"id": 404, "title": "Data Science", "instructor": "Prof. Herodotus"},
]
for cls in classes:
    driver.session().run(
        """
        CREATE (c:Class {id: $id, title: $title, instructor: $instructor})
        """,
        cls,
    )
```



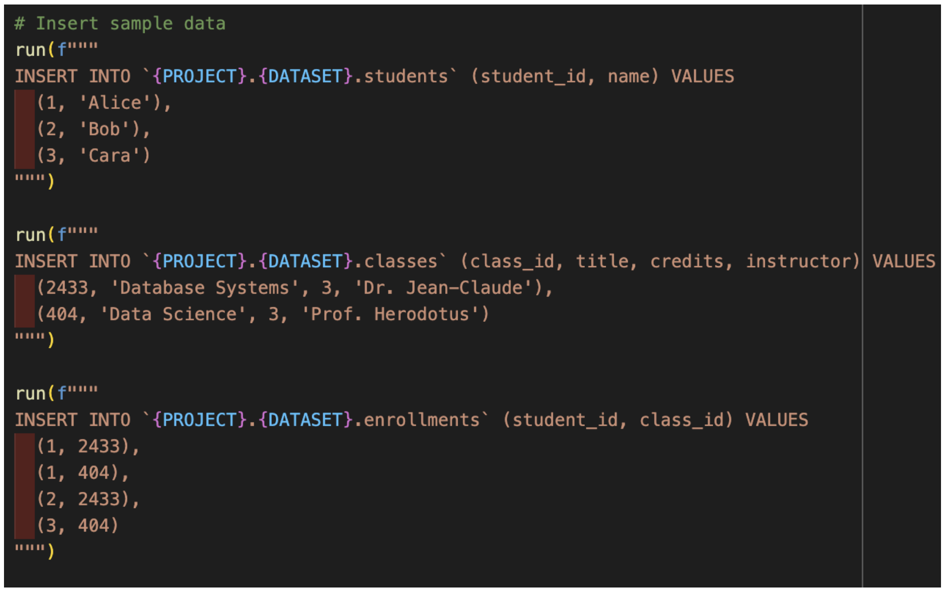
```python
# Step 2: Create relationships (enrollments)
enrollments = [
    (1, 2433),
    (1, 404),
    (2, 2433),
    (3, 404)
]
for student_id, class_id in enrollments:
    driver.session().run(
        """
        MATCH (s:Student {id: $student_id}), (c:Class {id: $class_id})
        CREATE (s)-[:ENROLLED_IN]->(c)
        """,
        {"student_id": student_id, "class_id": class_id},
    )
```

Calle, Paola
Assignment # 1

BigQuery



```
# Create students table
run(f"""
CREATE TABLE IF NOT EXISTS `{PROJECT}.{DATASET}.students` (
  student_id INT64,
  name STRING
)
""")

# Create classes table
run(f"""
CREATE TABLE IF NOT EXISTS `{PROJECT}.{DATASET}.classes` (
  class_id INT64,
  title STRING,
  credits INT64,
  instructor STRING
)
""")

# Create enrollment table (many-to-many relationship)
run(f"""
CREATE TABLE IF NOT EXISTS `{PROJECT}.{DATASET}.enrollments` (
  student_id INT64,
  class_id INT64
)
""")
```



```
# Insert sample data
run(f"""
INSERT INTO `{PROJECT}.{DATASET}.students` (student_id, name) VALUES
  (1, 'Alice'),
  (2, 'Bob'),
  (3, 'Cara')
""")

run(f"""
INSERT INTO `{PROJECT}.{DATASET}.classes` (class_id, title, credits, instructor) VALUES
  (2433, 'Database Systems', 3, 'Dr. Jean-Claude'),
  (404, 'Data Science', 3, 'Prof. Herodotus')
""")

run(f"""
INSERT INTO `{PROJECT}.{DATASET}.enrollments` (student_id, class_id) VALUES
  (1, 2433),
  (1, 404),
  (2, 2433),
  (3, 404)
""")
```

Calle, Paola
Assignment # 1

Setup & Implementation Comparison

From a data model implementation perspective, MongoDB Atlas is the easiest to create, you spin up a free cluster, connect with a URI, and collections are autocreated on first insert with no schema required. This makes it ideal for quick prototyping but leaves schema enforcement and indexing entirely up to you. Neo4j AuraDB is almost as simple: once the instance is provisioned, you can immediately define nodes and relationships using Cypher or the driver. Its strength is how naturally it maps the student course enrollment domain, though you'll need to learn graph-oriented thinking. Google BigQuery, by contrast, is the most complex to set up. It requires creating a cloud project, enabling APIs, generating service accounts, and writing SQL DDL statements to explicitly define tables before inserting data. While the initial overhead is higher, this approach enforces rigor and gives you warehouse-level control from the start.

Calle, Paola
Assignment # 1

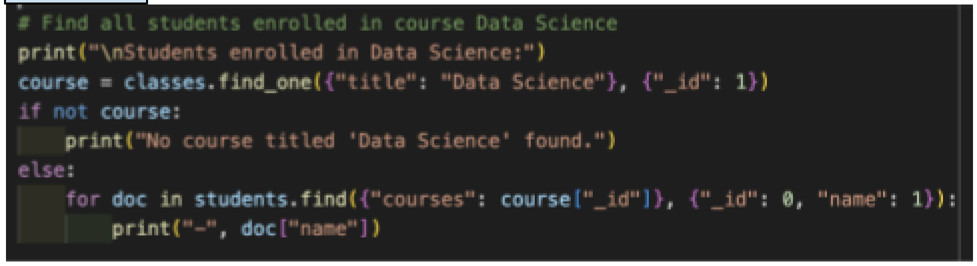
Question Query Analysis

Within this section, we focus on the query design of each of the DBMS that we implemented to answer the following questions:

1. Find all students enrolled in Data Science Class
2. Find students who share >=1 course with Alice

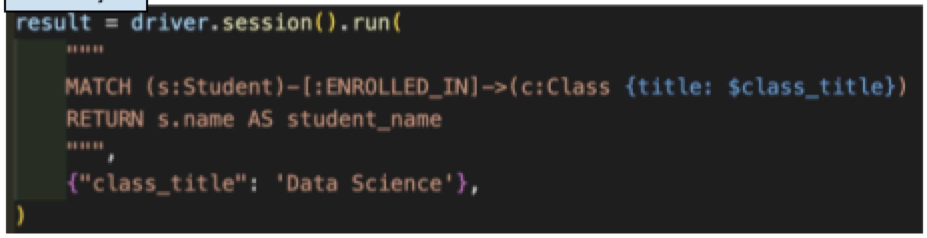
Question 1:

MongoDB



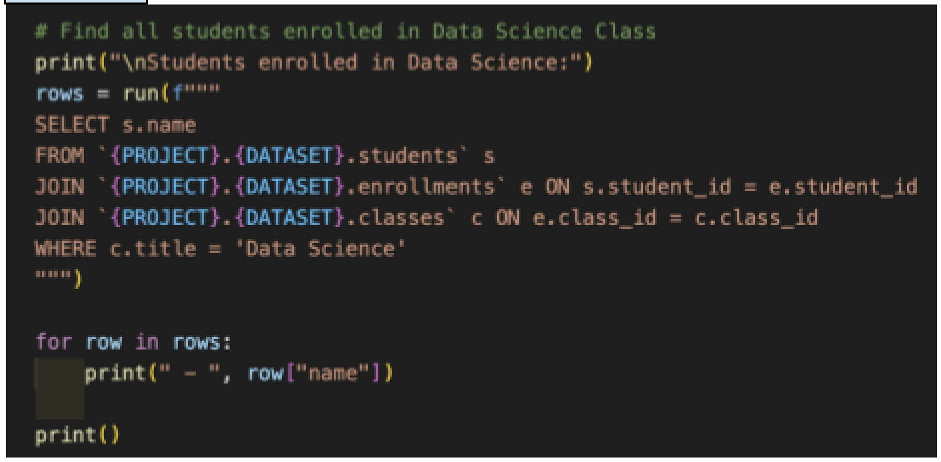
```python
# Find all students enrolled in course Data Science
print("\nStudents enrolled in Data Science:")
course = classes.find_one({"title": "Data Science"}, {"_id": 1})
if not course:
    print("No course titled 'Data Science' found.")
else:
    for doc in students.find({"courses": course["_id"]}, {"_id": 0, "name": 1}):
        print("-", doc["name"])
```

Neo4j



```python
result = driver.session().run(
    """
    MATCH (s:Student)-[:ENROLLED_IN]->(c:Class {title: $class_title})
    RETURN s.name AS student_name
    """,
    {"class_title": 'Data Science'},
)
```

BigQuery



```python
# Find all students enrolled in Data Science Class
print("\nStudents enrolled in Data Science:")
rows = run(f"""
SELECT s.name
FROM `{PROJECT}.{DATASET}.students` s
JOIN `{PROJECT}.{DATASET}.enrollments` e ON s.student_id = e.student_id
JOIN `{PROJECT}.{DATASET}.classes` c ON e.class_id = c.class_id
WHERE c.title = 'Data Science'
""")

for row in rows:
    print(" - ", row["name"])

print()
```

Calle, Paola
Assignment # 1

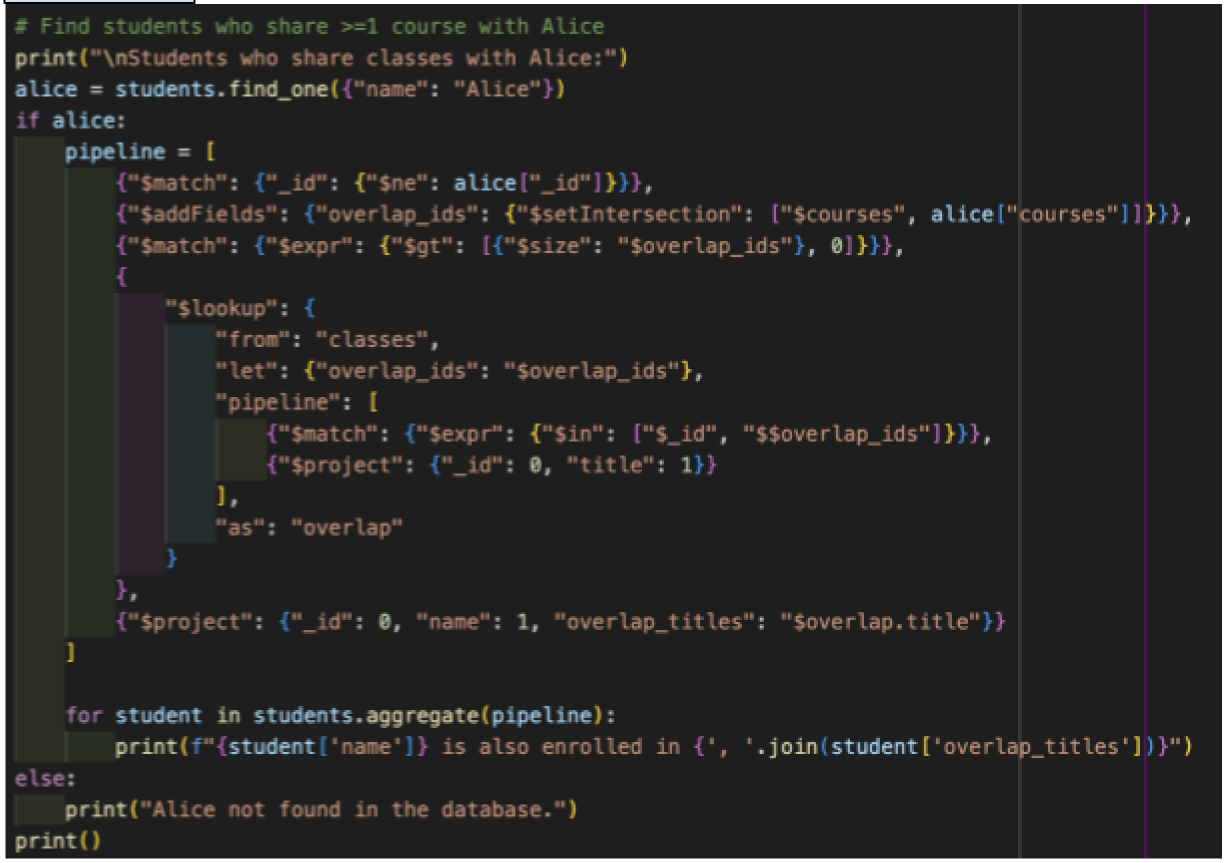
Looking at how each system retrieves “all students enrolled in Data Science” shows the flavor of each database:

- MongoDB uses a document query + array match, which is quick and flexible but relies on $lookup joins or manual reference handling.
- Neo4j expresses the query as a relationship traversal (Student → ENROLLED_IN → Class), making the logic natural if your domain is about connections.
- BigQuery requires a multi-table SQL join, which is more verbose but highly standardized and powerful for analytics.

For the specific query “find all students enrolled in Data Science”, Neo4j really is the cleanest fit for the question is literally a relationship traversal.

Question 2:

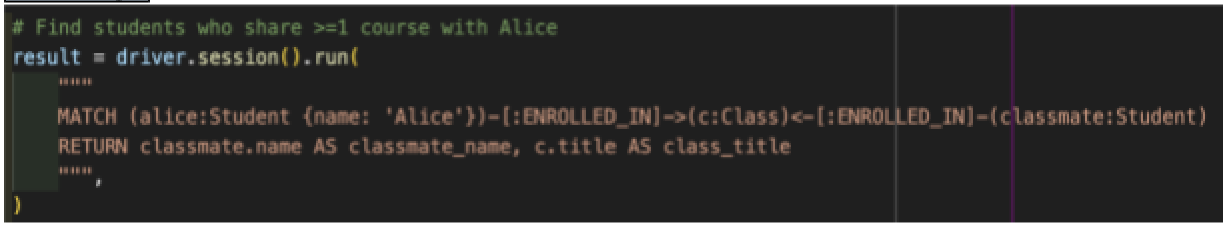
MongoDB



```python
# Find students who share >=1 course with Alice
print("\nStudents who share classes with Alice:")
alice = students.find_one({"name": "Alice"})
if alice:
    pipeline = [
        {"$match": {"_id": {"$ne": alice["_id"]}}},
        {"$addFields": {"overlap_ids": {"$setIntersection": ["$courses", alice["courses"]]}}},
        {"$match": {"$expr": {"$gt": [{"$size": "$overlap_ids"}, 0]}}},
        {
            "$lookup": {
                "from": "classes",
                "let": {"overlap_ids": "$overlap_ids"},
                "pipeline": [
                    {"$match": {"$expr": {"$in": ["$_id", "$$overlap_ids"]}}},
                    {"$project": {"_id": 0, "title": 1}}
                ],
                "as": "overlap"
            }
        },
        {"$project": {"_id": 0, "name": 1, "overlap_titles": "$overlap.title"}}
    ]

    for student in students.aggregate(pipeline):
        print(f"{student['name']} is also enrolled in {', '.join(student['overlap_titles'])}")
else:
    print("Alice not found in the database.")
print()
```

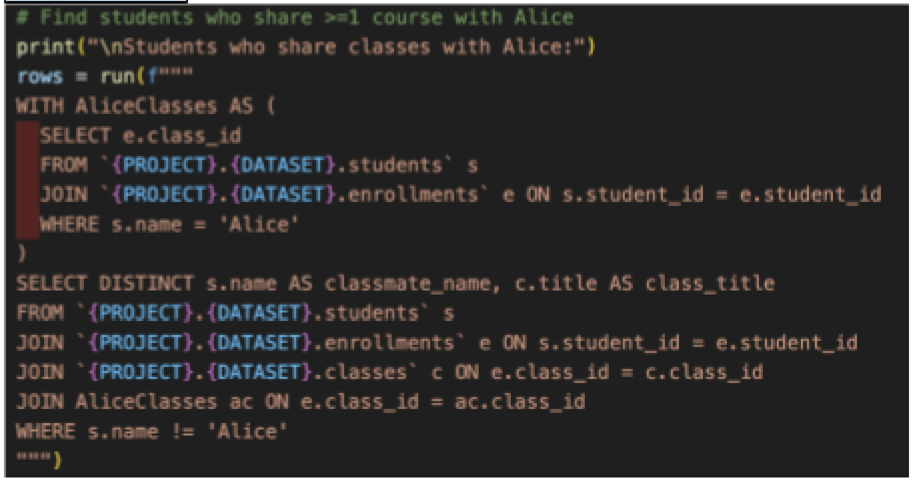
Calle, Paola
Assignment # 1

Neo4j



```
# Find students who share >=1 course with Alice
result = driver.session().run(
    """
    MATCH (alice:Student {name: 'Alice'})-[:ENROLLED_IN]->(c:Class)<-[:ENROLLED_IN]-(classmate:Student)
    RETURN classmate.name AS classmate_name, c.title AS class_title
    """,
)
```

BigQuery



```
# Find students who share >=1 course with Alice
print("\nStudents who share classes with Alice:")
rows = run(f"""
WITH AliceClasses AS (
  SELECT e.class_id
  FROM `{PROJECT}.{DATASET}.students` s
  JOIN `{PROJECT}.{DATASET}.enrollments` e ON s.student_id = e.student_id
  WHERE s.name = 'Alice'
)
SELECT DISTINCT s.name AS classmate_name, c.title AS class_title
FROM `{PROJECT}.{DATASET}.students` s
JOIN `{PROJECT}.{DATASET}.enrollments` e ON s.student_id = e.student_id
JOIN `{PROJECT}.{DATASET}.classes` c ON e.class_id = c.class_id
JOIN AliceClasses ac ON e.class_id = ac.class_id
WHERE s.name != 'Alice'
""")
```

All three databases can answer the query "find students who share at least one course with Alice." In MongoDB, this requires a long aggregation pipeline with intersections and lookups, making it functional but verbose. BigQuery handles it through SQL joins and CTEs, powerful but more complex to express. Neo4j, on the other hand, solves it with a single, natural relationship traversal, directly mirroring the question. For relationship-heavy queries like this, Neo4j once again is the most intuitive and concise choice

Takeaways

While Neo4j "won" for these two questions, if the focus is on analytics (e.g., grade distributions, average enrollments, term comparisons), then BigQuery is the clear winner. For app prototyping and schema flexibility, MongoDB is the easiest to get started with. Ultimately, the best database is the one that aligns with the user's needs, the shape of the data, and the scale of the problem.

Calle, Paola
Assignment # 1

Works Cited

AWS. *What is SQL (Structured Query Language)?*, AWS, https://aws.amazon.com/what-is/sql/.

Neo4j. "Build applications with Neo4j and Python." *Neo4j*, https://neo4j.com/docs/python-manual/current/. Accessed 18 September 2025.

Engines, DB. "DB-Engines Ranking - popularity ranking of database management systems." *DB-Engines*, https://db-engines.com/en/ranking. Accessed 17 September 2025.

SQL, My. "MySQL :: MySQL 8.4 Reference Manual :: 17.6.2.2 The Physical Structure of an InnoDB Index." *MySQL :: Developer Zone*, https://dev.mysql.com/doc/refman/8.4/en/innodb-physical-structure.html. Accessed 17 September 2025.