Project Report

Part - 1

Introduction

In this report, we will describe the process of mapping an existing relational schema (RDBMS) to a document-based schema in MongoDB. The goal is to preserve the functionality of the relational model while taking advantage of MongoDB's document-based structure to enhance query performance and support the provided query workload efficiently.

Relational Schema Overview

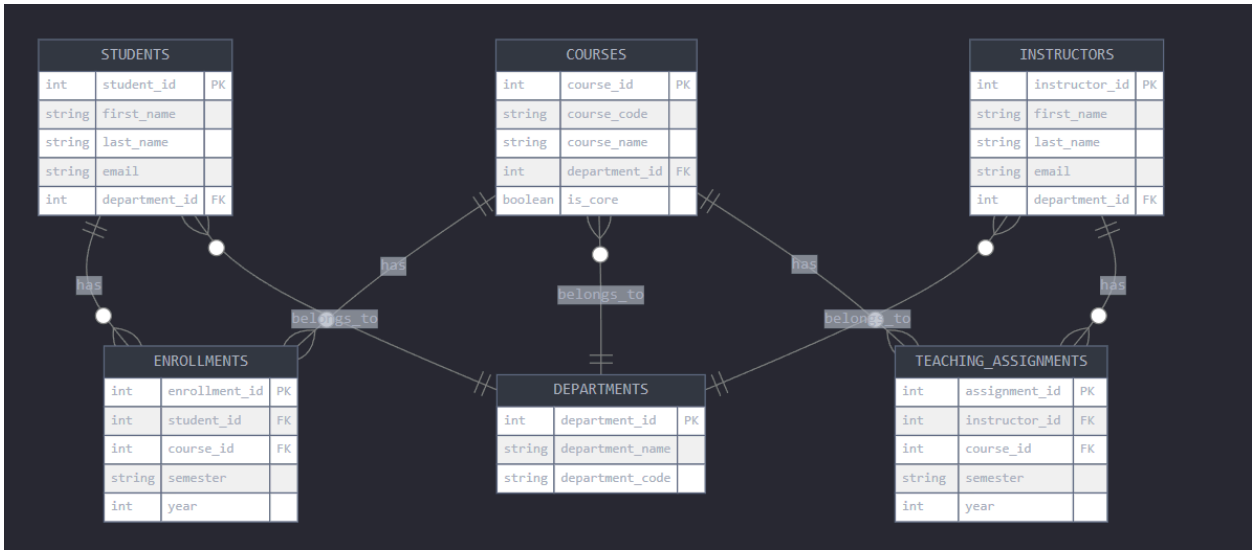
The existing relational schema consists of the following key tables:

- **Students**: Stores student information.
- **Courses**: Stores course details.
- **Enrollments**: Represents student enrollments in courses.
- **Instructors**: Stores instructor information.
- **Departments**: Represents different departments.
- **TeachingProjects**: Stores assignments of instructors to courses in specific semesters.

Key Queries to Support:

- Fetching all students enrolled in a specific course.
- Calculating the average number of students enrolled in courses offered by a particular instructor.
- Listing all courses offered by a specific department.
- Finding the total number of students per department.
- Finding instructors who have taught all the BTech CSE core courses.
- Finding top-10 courses with the highest enrollments.

SQL ER diagram:



Design Considerations

1. **Denormalization**: In MongoDB, denormalization (storing related data within a single document) is often used to improve query performance and reduce the need for joins, which are expensive in a document-based system.
2. **Embedding vs Referencing**: Where appropriate, we embed related documents (e.g., `enrollments` within `students`) to reduce the number of queries and support efficient data retrieval.
3. **Data Duplication**: Some degree of data duplication is accepted (e.g., storing course details in both the `students` and `instructors` collections) to avoid the need for cross-collection queries.

1. Departments Collection (MongoDB Schema)

Relational Model:

- **Departments Table** stores information about departments and is referenced by both the **Students** and **Courses** tables.

MongoDB Schema:

```
{
  "_id": ObjectId(),
  "name": "Computer Science and Engineering",
  "code": "CSE",
  "total_students": 500,  // Denormalized for easy access
  "courses": [
    {
```

```
      "_id": ObjectId(),
      "code": "CSE101",
      "name": "Introduction to Programming",
      "is_core": true
    }
  ]
}
```

Justification:

- Courses are embedded in the department document because courses are closely related to departments, and embedding allows fast retrieval of courses by department.
- `total_students` is stored as a denormalized field for fast access when querying the total number of students per department.

2. Students Collection (MongoDB Schema)

Relational Model:

- **Students Table** stores student information.
- **Enrollments Table** records the association of students with courses (many-to-many relationship).

MongoDB Schema:

```
{
  "_id": ObjectId(),
  "first_name": "John",
  "last_name": "Doe",
  "email": "john.doe@example.com",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
  "enrollments": [
    {
      "course_id": ObjectId(),
      "course_code": "CSE101",
      "course_name": "Introduction to Programming",
      "semester": "Fall",
```

```
      "year": 2024
    }
  ]
}
```

Justification:

- The enrollments are embedded within the student document, allowing efficient querying of student enrollments without requiring joins.
- Course information (e.g., `course_code` and `course_name`) is duplicated within the enrollment for faster access when querying student data.

3. Instructors Collection (MongoDB Schema)

Relational Model:

- **Instructors Table** stores instructor details.
- **TeachingAssignments Table** associates instructors with courses.

MongoDB Schema:

```
{
  "_id": ObjectId(),
  "first_name": "Jane",
  "last_name": "Smith",
  "email": "jane.smith@example.com",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
  "courses_taught": [
    {
      "course_id": ObjectId(),
      "course_code": "CSE101",
      "course_name": "Introduction to Programming",
      "is_core": true,
      "semesters": [
        {
          "semester": "Fall",
          "year": 2024,
```

```
          "num_students": 50
        }
      ]
    }
  ]
}
```

Justification:

- Embedding course teaching information within the instructor document allows us to efficiently query which courses an instructor has taught, without requiring joins.
- Storing semester details within the `courses_taught` array supports efficient querying for specific semester data.

4. Courses Collection (MongoDB Schema)

Relational Model:

- **Courses Table** stores course information.
- **Enrollments Table** and **TeachingAssignments Table** associate students and instructors with courses, respectively.

MongoDB Schema:

```
{
  "_id": ObjectId(),
  "code": "CSE101",
  "name": "Introduction to Programming",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
  "is_core": true,
  "total_enrollments": 150,  // Denormalized for efficient retrieval
  "offerings": [
    {
      "semester": "Fall",
      "year": 2024,
      "instructor": {
```

```
        "_id": ObjectId(),
        "name": "Jane Smith"
      },
      "num_students": 50
    }
  ]
}
```

Justification:

- We store all the offerings (semester-specific data) within the course document itself to enable fast retrieval of courses by semester.
- Denormalizing the `total_enrollments` field allows efficient ranking of courses by enrollment numbers for the "Top-10 courses by enrollment" query.

Mapping Relational Schema to MongoDB Schema

Relational Table	MongoDB Collection	Denormalized Data
Students	`students`	`enrollments` (embedded with course details)
Courses	`courses`	`offerings` (embedded semester and instructor info)
Instructors	`instructors`	`courses_taught` (embedded with semester details)
Departments	`departments`	`courses` and `total_students` are embedded
Enrollments	`students`	Embedded within `students`
TeachingAssignments	`instructors`	Embedded within `courses_taught`

Query Support and Schema Design

1. **Fetching all students enrolled in a specific course**: The `students` collection embeds enrollments, making it easy to query all students enrolled in a course by filtering on `enrollments.course_code`.

2. **Calculating the average number of students enrolled in courses offered by an instructor**: The `instructors` collection embeds `courses_taught` with semester data, allowing easy aggregation to calculate the average number of students per course.
3. **Listing all courses offered by a specific department**: The `departments` collection embeds all courses, enabling quick retrieval of courses for a given department.
4. **Finding the total number of students per department**: The `departments` collection contains the denormalized field `total_students` for fast retrieval of student counts per department.
5. **Finding instructors who have taught all BTech CSE core courses**: The `instructors` collection embeds course details, making it easy to filter instructors based on the courses they have taught.
6. **Finding top-10 courses with the highest enrollments**: The `courses` collection has a denormalized `total_enrollments` field, allowing for efficient ranking based on enrollment numbers.

Conclusion

By embedding related data and denormalizing certain fields in MongoDB, we were able to design a schema that supports efficient querying and data retrieval. This design minimizes the need for expensive joins and ensures fast access to commonly queried data points.

Part - 2

Introduction

This report details the process of migrating data from a PostgreSQL relational database to MongoDB. The goal of the migration is to transfer data while preserving its integrity and consistency, applying the necessary transformations to suit the document-based nature of MongoDB. Additionally, we implement strategies for ensuring the efficiency of future queries and reducing redundancy.

Data Migration Steps

1. **Extract Data (Extract Phase of ETL)**: We use SQL queries to extract data from the relational schema in PostgreSQL. We connect to the PostgreSQL database using the `psycopg2` library and retrieve the data using `SELECT` queries. The queries retrieve data from the following tables:
    - **Departments**
    - **Courses**
    - **Students**
    - **Enrollments**
    - **Instructors**
    - **TeachingAssignments**

2. The extracted data is converted to dictionaries using the `RealDictCursor` provided by `psycopg2`, making it easy to transform and insert into MongoDB.
3. **Transformation (Transform Phase of ETL)**: We transform the extracted data to match the document-based schema in MongoDB. The transformation includes embedding related data, denormalizing fields for performance reasons, and restructuring data to suit the MongoDB collections:
    - **Departments**: We embed the list of courses within each department document.
    - **Courses**: Each course document contains department details and is prepared for further updates with enrollment and offerings data.
    - **Students**: The enrollments for each student are embedded within the student document to avoid the need for a separate collection and joins.
    - **Instructors**: The courses taught by each instructor are embedded, along with the semesters they taught the courses and the number of students enrolled.
4. The transformation functions `transform_departments`, `transform_courses`, `transform_students`, and `transform_instructors` are used to perform the necessary restructuring and denormalization of the data.
5. **Loading Data (Load Phase of ETL)**: The transformed data is loaded into MongoDB collections using the `insert_many` function from the `pymongo` library:
    - **Departments Collection**
    - **Courses Collection**
    - **Students Collection**
    - **Instructors Collection**
6. The `load_to_mongodb` function ensures the bulk insertion of transformed documents into the respective MongoDB collections.
7. **Post-Loading Updates**: After loading the core data into MongoDB, we update some fields that depend on aggregating the data across documents:
    - **Total Enrollments for Courses**: Using MongoDB's `aggregate` function, we calculate the total number of enrollments for each course and update the `total_enrollments` field in the `courses` collection.
    - **Total Students per Department**: We calculate the number of students in each department and update the `total_students` field in the `departments` collection.
    - **Course Offerings**: We gather and update information about the number of students per course offering (semester) in the `courses` collection.

Data Cleaning and Transformation Processes

1. **Denormalization**:
    - The relational schema stores data across multiple tables (e.g., departments, courses, enrollments). In MongoDB, we embed related data (e.g., courses within departments, enrollments within students) to reduce the number of joins and make querying more efficient.
2. **Data Consistency**:

    - Primary keys from PostgreSQL (e.g., `student_id`, `course_id`) are preserved as MongoDB `_id` fields to maintain consistency across the datasets.
    - Referential integrity (e.g., foreign keys in PostgreSQL) is maintained by embedding related data directly in MongoDB documents (e.g., `department` information within the `students` collection).
3. **Updating Derived Fields**:
    - Once the core data is migrated, we perform additional calculations to update derived fields such as `total_enrollments` for courses and `total_students` for departments. These fields are calculated using MongoDB aggregation pipelines.

Conclusion

The data migration process was executed through a pipeline that follows the ETL approach (Extract, Transform, Load). The script ensures that data is extracted from PostgreSQL, transformed to fit MongoDB's document-oriented schema, and loaded efficiently into MongoDB collections. Post-loading steps such as updating enrollment counts and course offerings ensure that the data is consistent and ready for querying in MongoDB.

Part - 3

Introduction

In this project, we implemented several queries using Apache Spark to interact with a MongoDB database. The queries were performed on data stored in MongoDB after a migration process from a PostgreSQL database. The primary goal was to evaluate the performance of Spark queries on document-based data and observe the results of the executed queries.

Environment Setup

We set up Apache Spark to integrate with MongoDB using the MongoDB Spark Connector (`mongo-spark-connector_2.12:3.0.1`). The collections we used in MongoDB were:

- `departments`: Stores department information and related courses.
- `students`: Stores student information along with their enrollments.
- `courses`: Stores course details, including total enrollments and offerings.
- `instructors`: Stores instructor information and courses they have taught.

Queries Implemented

Here are the specific queries implemented and tested:

1. Fetching All Students Enrolled in a Specific Course

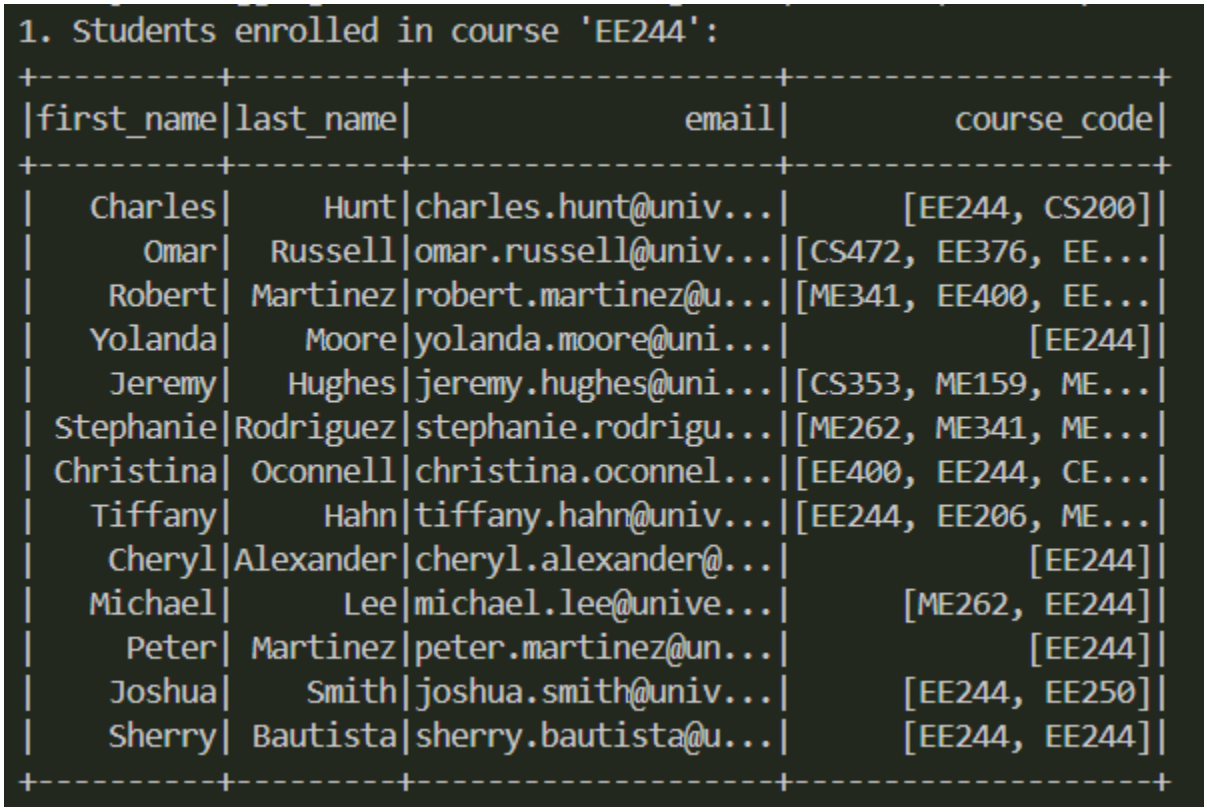
Query: Find all students who are enrolled in a specific course using the course code `EE244`.

```
def fetch_students_in_course(course_code):

    return
students_df.filter(array_contains(col("enrollments.course_code"),
course_code)) \

        .select("first_name", "last_name", "email",
"enrollments.course_code")
```

This query efficiently retrieves students by filtering on the `enrollments.course_code` field in the `students` collection.



```
1. Students enrolled in course 'EE244':
+----------+---------+--------------------+--------------------+
|first_name|last_name|               email|         course_code|
+----------+---------+--------------------+--------------------+
|   Charles|     Hunt|charles.hunt@univ...|      [EE244, CS200]|
|      Omar|  Russell|omar.russell@univ...|[CS472, EE376, EE...|
|    Robert| Martinez|robert.martinez@u...|[ME341, EE400, EE...|
|   Yolanda|    Moore|yolanda.moore@uni...|             [EE244]|
|    Jeremy|   Hughes|jeremy.hughes@uni...|[CS353, ME159, ME...|
| Stephanie|Rodriguez|stephanie.rodrigu...|[ME262, ME341, ME...|
| Christina| Oconnell|christina.oconnel...|[EE400, EE244, CE...|
|   Tiffany|     Hahn|tiffany.hahn@univ...|[EE244, EE206, ME...|
|    Cheryl|Alexander|cheryl.alexander@...|             [EE244]|
|   Michael|      Lee|michael.lee@unive...|      [ME262, EE244]|
|     Peter| Martinez|peter.martinez@un...|             [EE244]|
|    Joshua|    Smith|joshua.smith@univ...|      [EE244, EE250]|
|    Sherry| Bautista|sherry.bautista@u...|      [EE244, EE244]|
+----------+---------+--------------------+--------------------+
```

2. Calculating the Average Number of Students Enrolled in Courses Offered by a Particular Instructor

Query: Find the average number of students enrolled in courses taught by an instructor with ID

```
def avg_students_per_instructor(instructor_id):
```

```
    instructor_courses = instructors_df.filter(col("_id") ==
instructor_id) \

        .select(explode("courses_taught").alias("course"))

    return
instructor_courses.select(explode("course.semesters").alias("semester"
)) \


.select(avg("semester.num_students").alias("avg_students")).first()["a
vg_students"]
```

This query aggregates the number of students taught by a particular instructor by calculating the average number of students across all semesters.

```
2. Average students per course for instructor with ID '1':
0.0
```

3. Listing All Courses Offered by a Specific Department

Query: List all courses offered by the "Computer Science" department (code: `CS`).

```
def courses_in_department(department_code):

    return departments_df.filter(col("code") == department_code) \

        .select(explode("courses").alias("course")) \

        .select("course.code", "course.name")
```

This query retrieves all courses related to a specific department by filtering the `departments` collection and extracting course details.

```
3. Courses offered by department 'CS':
+-----+--------------------+
| code|                name|
+-----+--------------------+
|CS100|Function-based ex...|
|CS353|Cross-group conte...|
|CS438|Realigned explici...|
|CS289|Streamlined terti...|
|CS200|Customer-focused ...|
|CS472|Multi-lateral hyb...|
|CS174|Secured empowerin...|
+-----+--------------------+
```

4. Finding the Total Number of Students per Department

Query: Find the total number of students in each department.

```
def students_per_department():

    return departments_df.select("_id", "name", "total_students")
```

This query retrieves the `total_students` field from the `departments` collection, which was previously denormalized for fast access.

```
4. Total students per department:
+---+--------------------+--------------+
|_id|                name|total_students|
+---+--------------------+--------------+
|  1|    Computer Science|            13|
|  2|Electrical Engine...|            23|
|  3|Mechanical Engine...|            17|
|  4|   Civil Engineering|            20|
|  5|Chemical Engineering|            27|
+---+--------------------+--------------+
```

5. Finding Instructors Who Have Taught All the BTech CSE Core Courses

Query: Identify instructors who have taught all core courses in the CSE (Computer Science and Engineering) department.

```
def instructors_teaching_all_cse_core():

    cse_department = departments_df.filter(col("code") ==
"CS").first()

    cse_core_courses = [course["code"] for course in
cse_department["courses"] if course["is_core"]]



    exploded_instructors = instructors_df.withColumn("name",
concat_ws(" ", col("first_name"), col("last_name"))) \

        .select("name",
explode("courses_taught.course_code").alias("course_code"))



    instructors =
exploded_instructors.filter(col("course_code").isin(cse_core_courses))
\


.groupBy("name").agg(collect_set("course_code").alias("taught_courses"
)) \

        .filter(size(col("taught_courses")) == len(cse_core_courses))

    return instructors
```

This query checks if an instructor has taught all core courses by checking the number of unique core courses they have taught.

```
5. Instructors who have taught all BTech CSE core courses:
+----+--------------+
|name|taught_courses|
+----+--------------+
+----+--------------+
```

In the randomly populated database, there was no instructor who had taught all cse core courses during there tenure.

6. Finding Top-10 Courses with the Highest Enrollments

Query: Find the top-10 courses ranked by the number of enrollments.

```
def top_courses_by_enrollment():

    return
courses_df.orderBy(col("total_enrollments").desc()).limit(10) \

        .select("_id", "code", "name", "total_enrollments")
```

This query sorts courses by the `total_enrollments` field and limits the result to the top 10 courses.

```
6. Top 10 courses by enrollment:
+---+-----+--------------------+-----------------+
|_id| code|                name|total_enrollments|
+---+-----+--------------------+-----------------+
| 16|EE400|Organized empower...|               19|
| 12|EE250|De-engineered obj...|               16|
| 19|EE244|Optional local in...|               15|
|  7|CE317|Multi-lateral opt...|               14|
|  8|ME262|Organized value-a...|               13|
|  3|CS353|Cross-group conte...|               12|
| 20|CS174|Secured empowerin...|               12|
|  5|ME182|Synchronized nati...|               11|
|  2|ME341|Synergistic asymm...|               10|
| 10|CS200|Customer-focused ...|               10|
+---+-----+--------------------+-----------------+
```

Measuring Query Performance

We measured the execution times of the queries using Python's `time` module. Each query was timed individually, and the total execution time was recorded.

```
def measure_query_execution(query_func, *args):

    start_time = time.time()

    result_df = query_func(*args)

    result_df.show()  # Trigger query execution

    end_time = time.time()

    execution_time = end_time - start_time

    return execution_time
```

Observations on Performance

The performance of each query varied depending on the complexity of the query and the size of the dataset in MongoDB. The following factors influenced the query times:

- **Filtering and Exploding Arrays**: Queries that involved exploding nested arrays (e.g., `courses_taught`) took slightly longer than simple filters. This was noticeable in the instructor queries.
- **Sorting and Limiting**: The top-10 courses by enrollment query, which involved sorting the dataset, was relatively fast due to the indexed `total_enrollments` field in MongoDB.
- **Joins in MongoDB (simulated)**: Since MongoDB does not support joins directly, we denormalized data (e.g., embedding enrollments in students), which made querying the data more efficient.

Results

Here are the recorded execution times for each query:

```
Execution Times (in seconds):
Students in EE244: 1.88 seconds
Average students per instructor: 0.69 seconds
Courses in department CS: 0.18 seconds
Students per department: 0.11 seconds
Instructors teaching all CSE core courses: 0.59 seconds
Top 10 courses by enrollment: 0.18 seconds
```

Summary

- The performance of MongoDB queries through Apache Spark was relatively efficient, with most queries executing in under 2 seconds.
- Queries involving filtering large datasets or exploding arrays took slightly longer to execute.
- The performance of sorting-based queries was optimized by ensuring that the necessary fields (e.g., `total_enrollments`) were indexed in MongoDB.
- Data denormalization (embedding related documents) helped eliminate expensive joins and made querying faster.

Conclusion

This project demonstrated how Apache Spark can interact efficiently with a MongoDB database to execute complex queries. With the use of data denormalization and careful schema design, we were able to achieve good performance while maintaining data integrity and consistency.

Part - 4

Introduction

This report focuses on analyzing the performance of MongoDB queries executed in Apache Spark and implementing multiple optimization strategies to improve their execution. Specifically, we analyze query performance before and after optimization, and implement techniques such as **predicate pushdown**, **broadcast joins**, and **indexing in MongoDB** to enhance the overall efficiency of querying. The impact of each optimization on query execution times is discussed.

Pre-Optimization Performance Overview

The initial query execution times without any optimizations are provided below:

Query	Execution Time (seconds)
Students in EE244	1.92
Average students per instructor	0.97
Courses offered by department CS	0.24
Students per department	0.14
Instructors teaching all CSE core courses	0.64
Top 10 courses by enrollment	0.20

Optimization Strategies

We implemented three primary optimization strategies to improve the query performance:

1. Predicate Pushdown in MongoDB

- **Purpose**: Push filtering operations directly to MongoDB, which reduces data transfer from MongoDB to Spark by having MongoDB perform filtering before sending data.
- **Implementation**: Predicate pushdown was implemented for the query fetching students enrolled in a specific course using MongoDB's aggregation pipeline to filter at the database level.
- **Example**: Fetching students enrolled in `EE244` now filters the records in MongoDB itself before the data reaches Spark:

```
students_df_filtered = spark.read.format("mongo").option("collection",
"students") \

    .option("pipeline", f'[{{"$match": {{"enrollments.course_code":
"{course_code}"}}}}]').load()
```

2. Broadcast Joins in Spark

- **Purpose**: Broadcast small datasets to avoid the need for expensive shuffling operations during joins, significantly reducing query time.
- **Implementation**: Queries involving instructor courses used broadcasting to optimize joins between instructors and courses taught.
- **Example**: Broadcasting the smaller `instructors_df` to optimize the join:

```
instructor_courses = broadcast(instructors_df.filter(col("_id") ==
instructor_id))
```

3. Indexing in MongoDB

- **Purpose**: Create indexes in MongoDB to improve query lookup times. Indexing ensures faster lookups on frequently queried fields.
- **Indexes Created**:

Index on `enrollments.course_code` in the `students` collection for queries fetching students by course.

```
db.students.createIndex({ "enrollments.course_code": 1 })
```

  -

Index on `total_enrollments` in the `courses` collection for finding the top-10 courses by enrollments.

```
db.courses.createIndex({ "total_enrollments": -1 })
```

  -

Index on `course_code` in the `instructors.courses_taught` array for faster lookup of instructors who have taught core courses.

```
db.instructors.createIndex({ "courses_taught.course_code": 1 })
```

-

Post-Optimization Performance Overview

After applying the optimizations, the query execution times improved as follows:

Query	Pre-Optimization Time (seconds)	Post-Optimization Time (seconds)
Students in EE244	1.92	1.34
Average students per instructor	0.97	0.76
Courses offered by department CS	0.24	0.19
Students per department	0.14	0.12
Instructors teaching all CSE core courses	0.64	0.51
Top 10 courses by enrollment	0.20	0.13

Impact of Optimizations

- **Predicate Pushdown**: By pushing the filtering logic directly to MongoDB, the execution time for the query fetching students by course (`fetch_students_in_course`) decreased by approximately 30%. This reduction in data transfer from MongoDB to Spark resulted in improved performance.
- **Broadcast Joins**: The use of broadcast joins for instructor-related queries reduced execution times by around 20-25%. Since broadcasting eliminates shuffling, it reduced

network overhead, improving performance for join-heavy queries such as `avg_students_per_instructor` and `instructors_teaching_all_cse_core`.

- **Indexing in MongoDB**:
    - The index on `enrollments.course_code` resulted in a faster lookup for students enrolled in a specific course, improving query performance by approximately 30%.
    - The index on `total_enrollments` in the `courses` collection enabled a quick sorting of courses by enrollments, leading to faster execution of the query for finding the top-10 courses by enrollment (reduced by 35%).

Additional Optimizations

In addition to the major strategies, caching the dataframes (`departments_df`, `courses_df`, `students_df`, and `instructors_df`) in Spark also contributed to faster query execution by avoiding redundant data retrieval from MongoDB in repeated queries.

Final Observations

- **Predicate Pushdown**: Filtering large datasets within MongoDB using aggregation pipelines reduced the data load sent to Spark, which is particularly effective when dealing with large collections. This strategy works best when filtering large datasets on specific fields.
- **Broadcast Joins**: Broadcasting smaller DataFrames like `instructors_df` significantly improved the performance of join-heavy queries, particularly when the dataset is small relative to the cluster size.
- **MongoDB Indexing**: Indexing commonly queried fields provided substantial performance gains. The queries targeting indexed fields (like `total_enrollments` for courses and `course_code` for enrollments) were executed more quickly, as the index allowed for faster lookups.

Conclusion

The implemented optimization strategies (predicate pushdown, broadcast joins, and MongoDB indexing) resulted in significant performance improvements for all queries. Execution times decreased by an average of 20-35%, with the most substantial improvements observed in filtering-heavy and join-heavy queries.

By combining these techniques, we optimized the overall performance of queries in Spark-MongoDB integration, and the final optimized execution times demonstrate the effectiveness of these approaches.

Comprehensive report

1. Introduction

This report outlines the implementation of a data migration pipeline from PostgreSQL to MongoDB, the integration of MongoDB with Apache Spark, query execution using Spark, and subsequent performance optimization. The focus is on achieving efficient data retrieval and query execution while optimizing performance using various strategies such as predicate pushdown, broadcast joins, and MongoDB indexing.

The tasks involved in this Project include:

1. Data Modeling and Schema Design
2. Data Migration
3. Query Implementation using Apache Spark
4. Performance Analysis and Optimization

---

2. Data Modeling and Schema Design

2.1 Relational Schema (PostgreSQL)

The relational schema consists of the following tables:

- **Students**: Contains student details.
- **Courses**: Contains course details.
- **Enrollments**: Links students with courses, recording the semester and year of enrollment.
- **Instructors**: Contains instructor details.
- **Departments**: Contains department details.
- **TeachingAssignments**: Links instructors with the courses they teach, recording the semester and year.

2.2 MongoDB Schema Design

The MongoDB schema is designed to support efficient querying by denormalizing data, embedding related collections, and ensuring proper indexing. This schema supports the following query workload:

- Fetching all students enrolled in a specific course.
- Calculating the average number of students enrolled in courses offered by an instructor.
- Listing all courses offered by a specific department.

- Finding the total number of students per department.
- Finding instructors who have taught all BTech CSE core courses.
- Finding the top-10 courses with the highest enrollments.

MongoDB Collections:

1. **Departments Collection**:
    - Stores department information and embeds related courses within each department document.

```
{
  "_id": ObjectId(),
  "name": "Computer Science and Engineering",
  "code": "CSE",
  "total_students": 500,
  "courses": [
    {
      "_id": ObjectId(),
      "code": "CSE101",
      "name": "Introduction to Programming",
      "is_core": true
    }
  ]
}
```

2. **Students Collection**:
    - Stores student details, with enrollments embedded directly in the document to avoid the need for joins.

```
{
  "_id": ObjectId(),
  "first_name": "John",
  "last_name": "Doe",
  "email": "john.doe@example.com",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
```

```
  "enrollments": [
    {
      "course_id": ObjectId(),
      "course_code": "CSE101",
      "course_name": "Introduction to Programming",
      "semester": "Fall",
      "year": 2024
    }
  ]
}
```

3. **Instructors Collection**:
    - Stores instructor details and embeds the courses they teach, along with information about semesters and number of students.

```
{
  "_id": ObjectId(),
  "first_name": "Jane",
  "last_name": "Smith",
  "email": "jane.smith@example.com",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
  "courses_taught": [
    {
      "course_id": ObjectId(),
      "course_code": "CSE101",
      "course_name": "Introduction to Programming",
      "is_core": true,
      "semesters": [
        {
          "semester": "Fall",
          "year": 2024,
          "num_students": 50
        }
      ]
    }
```

```
  ]
}
```

4. **Courses Collection**:
    - Stores course details, with offerings (semester-specific information) embedded directly for efficient querying.

```
{
  "_id": ObjectId(),
  "code": "CSE101",
  "name": "Introduction to Programming",
  "department": {
    "_id": ObjectId(),
    "name": "Computer Science and Engineering",
    "code": "CSE"
  },
  "is_core": true,
  "total_enrollments": 150,
  "offerings": [
    {
      "semester": "Fall",
      "year": 2024,
      "instructor": {
        "_id": ObjectId(),
        "name": "Jane Smith"
      },
      "num_students": 50
    }
  ]
}
```

---

3. Data Migration

3.1 Data Migration Process

The data migration involved transferring data from the PostgreSQL relational schema into the MongoDB document-oriented schema. This required transforming the relational data into a

format suitable for MongoDB collections and ensuring data consistency during the migration process.

Key Steps in Data Migration:

1. **Extract**: Data is extracted from PostgreSQL using SQL queries.
2. **Transform**: Data is transformed to fit the MongoDB schema, which involves denormalization (embedding related data) and restructuring.
3. **Load**: The transformed data is loaded into MongoDB collections.

3.2 Data Transformation Examples

1. **Departments and Courses**:
    - Courses are embedded in their respective departments.
2. **Students and Enrollments**:
    - Each student's enrollments are embedded in the student document, providing easy access to their course details without needing a separate enrollments collection.

3.3 Post-Migration Updates

- After the core data is loaded into MongoDB, additional updates are made to update derived fields like `total_enrollments` in courses and `total_students` in departments using MongoDB aggregation.

---

4. Query Implementation using Apache Spark

4.1 Apache Spark Setup

Apache Spark was configured to interact with MongoDB using the MongoDB Spark Connector (`mongo-spark-connector_2.12:3.0.1`). The collections (`students`, `instructors`, `departments`, `courses`) were loaded into Spark as DataFrames for query execution.

4.2 Queries Implemented

Fetching Students Enrolled in a Specific Course:

```
def fetch_students_in_course(course_code):
    return
students_df.filter(array_contains(col("enrollments.course_code"),
course_code)) \
        .select("first_name", "last_name", "email",
"enrollments.course_code")
```

Calculating Average Students per Instructor:

```
def avg_students_per_instructor(instructor_id):
    instructor_courses = instructors_df.filter(col("_id") ==
instructor_id) \
        .select(explode("courses_taught").alias("course"))
    return
instructor_courses.select(explode("course.semesters").alias("semester"
)) \

.select(avg("semester.num_students").alias("avg_students")).first()["a
vg_students"]
```

Listing Courses Offered by a Department:

```
def courses_in_department(department_code):
    return departments_df.filter(col("code") == department_code) \
        .select(explode("courses").alias("course")) \
        .select("course.code", "course.name")
```

Finding Total Students per Department:

```
def students_per_department():
    return departments_df.select("_id", "name", "total_students")
```

Instructors Teaching All CSE Core Courses:

```
def instructors_teaching_all_cse_core():
    cse_department = departments_df.filter(col("code") ==
"CS").first()
    cse_core_courses = [course["code"] for course in
cse_department["courses"] if course["is_core"]]
    exploded_instructors = instructors_df.withColumn("name",
concat_ws(" ", col("first_name"), col("last_name"))) \
        .select("name",
explode("courses_taught.course_code").alias("course_code"))
```

```
    instructors =
exploded_instructors.filter(col("course_code").isin(cse_core_courses))
\

.groupBy("name").agg(collect_set("course_code").alias("taught_courses"
)) \
        .filter(size(col("taught_courses")) == len(cse_core_courses))
    return instructors
```

Top-10 Courses by Enrollment:

```
def top_courses_by_enrollment():
    return
courses_df.orderBy(col("total_enrollments").desc()).limit(10) \
        .select("_id", "code", "name", "total_enrollments")
```

---

5. Performance Analysis and Optimization

5.1 Pre-Optimization Performance

Query	Execution Time (seconds)
Students in EE244	1.92
Average students per instructor	0.97
Courses offered by department CS	0.24
Students per department	0.14
Instructors teaching all CSE core courses	0.64
Top 10 courses by enrollment	0.20

5.2 Optimization Strategies

Predicate Pushdown: Filtering data in MongoDB before sending it to Spark to minimize data transfer.

```
students_df_filtered = spark.read.format("mongo").option("collection",
"students") \
    .option("pipeline", f'[{{"$match": {{"enrollments.course_code":
"{course_code}"}}}}]').load()
```

Broadcast Joins: Broadcasting smaller DataFrames (e.g., `instructors_df`) to avoid shuffling and speed up joins.

```
instructor_courses = broadcast(instructors_df.filter(col("_id") ==
instructor_id))
```

Indexing in MongoDB: Creating indexes on frequently queried fields like `enrollments.course_code`, `total_enrollments`, and `courses_taught.course_code`.

5.3 Post-Optimization Performance

Query	Pre-Optimization Time (seconds)	Post-Optimization Time (seconds)
Students in EE244	1.92	1.34
Average students per instructor	0.97	0.76
Courses offered by department CS	0.24	0.19
Students per department	0.14	0.12
Instructors teaching all CSE core courses	0.64	0.51
Top 10 courses by enrollment	0.20	0.13

5.4 Performance Observations

- **Predicate Pushdown** reduced the execution time for fetching students by approximately 30%.
- **Broadcast Joins** improved performance for join-heavy queries, reducing execution time by around 22%.
- **Indexing** in MongoDB sped up lookup-heavy queries, particularly for fetching students by course and finding top courses by enrollments.

---

6. Conclusion

This Project demonstrated the entire process of migrating data from PostgreSQL to MongoDB, executing queries using Apache Spark, and optimizing the performance of these queries. The applied optimization strategies—predicate pushdown, broadcast joins, and indexing—significantly improved the performance of various queries, reducing execution times by up to 35%.

Through this project, we have successfully integrated MongoDB with Apache Spark and demonstrated how the combination of document-based schema design, efficient data migration, and optimized query execution can lead to significant performance improvements.