BDA Assignment 1 Report

Garv Makkar | 2021530 | IIITD

Code Files and How to Run Them

The directory structure
Directories are quoted in ‘’
Files are as it is

‘2021530_BDA_A1’
- ‘Part 1 PostgreSQL and MongoDB’
  - Step1_Creating_RDBMS.py
  - Step2_Inserting_Data_In_RDBMS.py
  - Step3_Checking_RDBMS.ipynb
  - Step4_Migration_RDBMS_To_Mongo.py
  - Step5_Step4_Migration_RDBMS_To_Mongo_With_Indexing.py
- ‘Part 2 MongoDB and Spark’
  - main.py
  - main_indexing.py
  - main_query_optimized.py
  - main_partioning.py
  - main_repartitioning.py
  - Spark_Queries.py
  - Spark_Queries_indexing.py
  - Spark_Queries_Optimized.py
  - Spark_Queries_Repartioned.py
  - utils.py
- ‘query_outputs’
- ‘query_outputs_indexing’
- ‘query_outputs_partitioning’
- ‘query_outputs_repartitioning’
- ‘query_outputs_spark_optimized’
- ‘Running Queries in MongoDB’
- Question.pdf

How to run

- To create a DB in PostgreSQL, run Step1_Creating_RDBMS.py
- To insert data in it, run Step2_Inserting_Data_In_RDBMS.py

- To migrate it to MongoDB, run Step4_Migration_RDBMS_To_Mongo.py
- To migrate it to the indexing version MongoDB, run Step4_Migration_RDBMS_To_Mongo.py
- To get results of sample queries using Spark, run main.py
- To get results of sample queries using Spark but accessing indexed db for optimized results, run main_indexing.py
- To get results of sample queries using Spark and with optimized Spark queries, run main_query_optimized.py
- To get results of sample queries using Spark with data partitioning using connector: run main_partioning.py
- To get results of sample queries using Spark with data partitioning at the query level while getting data frames, run main_repartitioning.py
- Files starting with Spark_Queries contain the query methods for respective optimization choice
- File utils.py has some functions to support main files.
- Directory names starting with query_outputs store the results of respective main files.
- The directory 'Running Queries in MongoDB' contains the MongoDB queries and screenshots of results.

Data Modelling and Schema Design

RDBMS Schema in PostgreSQL

A university system with six tables:

1. Departments (departments):
    - Stores department info.
    - Columns: department_id (PK), department_name.
2. Students (students):
    - Stores student info.
    - Columns: student_id (PK), student_name, student_email (unique), department_id (FK to departments).
3. Instructors (instructors):
    - Stores instructor info.
    - Columns: instructor_id (PK), instructor_name, instructor_email (unique), department_id (FK to departments).
4. Courses (courses):
    - Stores course info.
    - Columns: course_id (PK), course_code (unique), department_id (FK), is_core_course, offering_year.
5. Teaches (teaches):
    - Assign instructors to courses.

    - Columns: teaches_id (PK), instructor_id (FK to instructors), course_id (FK to courses, unique).
6. Enrollments (enrollments):
    - Tracks student course enrollments.
    - Columns: enrollment_id (PK), student_id (FK to students), course_id (FK to courses).

MongoDB Schema

A university system has four main collections:

1. Departments (departments):
    - Stores department info.
    - Fields: _id (department ID), department_name.
2. Students (students):
    - Stores student info and their enrollments.
    - Fields: _id (student ID), student_name, student_email, department_id, enrolled_courses (array of course IDs).
3. Instructors (instructors):
    - Stores instructor info.
    - Fields: _id (instructor ID), instructor_name, instructor_email, department_id.
4. Courses (courses):
    - Stores course info.
    - Fields: _id (course ID), course_code, department_id, is_core_course, offering_year, instructor_id.

Mapping and Denormalization from PostgreSQL to MongoDB

This is how mapping was done.

1. Each department in PostgreSQL maps to a document in the departments collection. No denormalization. Departments are referenced in other collections by department_id.
2. Student data is migrated to the students collection with a reference to department_id. The enrollments table is denormalized by embedding an array of course_ids (enrolled_courses) directly in each student document, optimizing for frequent student-course queries.
3. Instructors map to the instructors collection with reference to department_id. No denormalization. Instructor data is kept separate, with references to departments.
4. Courses map to the courses collection with a reference to department_id. The teaches table is denormalized by embedding the instructor_id directly within the courses documents, making course-instructor queries more efficient.

5. The relationship between courses and instructors from the teaches table is denormalized into the courses collection as the instructor_id field. Simplifies queries, as each course can have only one instructor.
6. Handled the enrollments table within the students collection as an embedded array of course_ids. By embedding courses in student documents, querying for student enrollments becomes faster, reducing the need for cross-collection queries.

Denormalization was primarily applied to enrollments and teaches tables, embedding course, and instructor data within related documents to improve read performance in MongoDB.

Data Migration

Migration Steps

1. Established connections to both PostgreSQL and MongoDB databases.

2. Resetting MongoDB Collections to ensure a clean slate for migration by dropping existing collections: departments, students, instructors, and courses.

Table rows were transformed to document structure to satisfy the way of how MongoDB stores data.

3. Migrate Departments: Retrieves department data from PostgreSQL. Bulk inserts documents into the 'departments' collection in MongoDB.
Data Transformation:

- PostgreSQL 'department_id' becomes MongoDB document '_id'.
- 'department_name' is maintained as is.

4. Migrate Students: Retrieves student data from PostgreSQL. Bulk inserts documents into the 'students' collection in MongoDB.
Data Transformation:

- PostgreSQL 'student_id' becomes MongoDB document '_id'.
- 'student_name', 'student_email', and 'department_id' are maintained as is.

5. Migrate Instructors: Retrieves instructor data from PostgreSQL. Bulk inserts documents into the 'instructors' collection in MongoDB.
Data Transformation:

- PostgreSQL 'instructor_id' becomes MongoDB document '_id'.
- 'instructor_name', 'instructor_email', and 'department_id' are maintained as is.

6. Migrate Courses: Retrieves course data from PostgreSQL, including related instructor information. Bulk inserts documents into the 'courses' collection in MongoDB.

Data Transformation:

- PostgreSQL 'course_id' becomes MongoDB document '_id'.
- 'course_code', 'department_id', 'is_core_course', and 'offering_year' are maintained as is.
- 'instructor_id' is added to link courses with their instructors.

7. Migrate Enrollments: Retrieves enrollment data from PostgreSQL. Groups enrollments by student. Updates each student document in MongoDB with their enrolled courses.
Data Transformation:

- Enrollments are embedded within student documents as an array of course IDs.
- This represents a shift from a relational model to a document-oriented model.

Data Cleaning and Transformation

1. All primary keys from PostgreSQL tables are used as MongoDB document '_id' fields.
2. The many-to-many relationship between students and courses (enrollments) is transformed into an embedded array within student documents.
3. The relationship between courses and instructors is maintained through a reference (instructor_id in the course document).
4. Course information is consolidated with instructor information in a single query, reducing the need for multiple data fetches.
5. Using bulk insert operations (insert_many()) for each collection migration improves efficiency.
6. The script drops existing MongoDB collections before migration, ensuring no duplicate or stale data remains.

Query Implementation using Apache Spark

Installed spark by following: [Setup Apache Spark Environment on Windows 11? Step By Step Guide | by Dinesh Thapa - Big Data, Analytics & AI | Medium]

PySpark Code Report

An application using pySpark is implemented to run queries on a university database stored in MongoDB. It sets up a Spark session, executes a series of predefined queries, measures their execution times, and saves the results.

Key Features

1. Spark Session Setup

    - Creates a Spark session configured for local execution and MongoDB integration.
2. Query Execution
    - Imports six predefined queries from a Spark_Queries module.
    - Executes each query and measures its runtime.
    - Saves query results and execution times to CSV files.
3. Performance Measurement
    - Uses a time_function wrapper to measure query execution times.
    - Stores execution times for later analysis.
4. Data Management
    - Saves query results to CSV files in a specified output directory.
    - Creates a summary CSV of query execution times.
5. Main Execution Flow
    - Sets up the Spark session.
    - Performs a warm-up query to initialize the Spark context.
    - Executes all queries and saves results.

Queries Executed and their results

Queries

1. Fetching all students enrolled in a specific course.
2. Calculating the average number of students enrolled in courses offered by a particular instructor at the university.
3. Listing all courses offered by a specific department.
4. Finding the total number of students per department.
5. Finding instructors who have taught all the BTech CSE core courses some time during their tenure at the university.
6. Finding top-10 courses with the highest enrollments.

Query results after using Spark

1.
```
student_name,student_email
Ms. Erika Johnson DVM,mtodd@example.org
Carly Ward,wagnertracey@example.net
Timothy Wilson,marcia29@example.net

```
2.
```
average_students
1.5

```

3.
```
course_code,department_id,is_core_course
CS-012,1,True
CS-001,1,False
CS-017,1,False
CS-005,1,True
CS-011,1,True
CS-006,1,True
CS-003,1,True
CS-007,1,False
CS-016,1,True
CS-004,1,True
CS-002,1,True
CS-013,1,True
CS-015,1,False
CS-009,1,True
CS-008,1,True
CS-018,1,False
CS-014,1,True
CS-010,1,True
```
4.
```
department_name,total_students
Computer Science,10
Mechanical,12
Electrical,13
```
5.
```
instructor_name
```
It is empty because, in our dataset, there was not any instructor satisfying the query requirements
6.
```
_id,course_code,num_enrollments
34,ME-004,6
44,ME-014,6
24,EE-006,6
22,EE-004,6
9,CS-009,5
20,EE-002,5
38,ME-008,5
30,EE-012,5
27,EE-009,5
19,EE-001,5
```

Performance of these queries

```
filename,run_time
./query_outputs\query1_result,0.6057071685791016
./query_outputs\query2_result,0.19138336181640625
./query_outputs\query3_result,0.06599974632263184
./query_outputs\query4_result,0.10014677047729492
./query_outputs\query5_result,0.21362900733947754
./query_outputs\query6_result,0.09799718856811523

```

The first query is taking so much time because it is the first query and this time also includes the time for setup and initialization, which further queries don't have to do.

Performance Analysis and Optimization

Optimization techniques used

1. Indexing the MongoDB Database
2. Spark Query Optimization Using Explode, Broadcast, and Cache
3. Data Partitioning Using Connector Configuration
4. Using the Repartition Function in Queries

Indexing the MongoDB Database
Reference: Create an Index - MongoDB Manual v7.0

The 'Create Indexes' function in the script creates indexes in MongoDB to improve query speed:

- Departments: Unique index on _id (department ID).
- Students: Unique index on _id, index on department_id for filtering, and on enrolled_courses for course queries.
- Instructors: Unique index on _id, index on department_id.
- Courses: Unique index on _id, index on department_id, and instructor_id for efficient filtering.

Indexes boost query performance by enabling faster lookups and filtering.

Spark Query Optimization Using Explode, Broadcast, and Cache

There can be some modifications in queries to boost the performance. I implemented these:

- .cache() method: to store the result in memory if it's used multiple times in the application.
- broadcast() method: to optimize join operations in Spark by reducing data shuffling across the cluster. Spark sends a copy of the entire data frame to every node in the cluster. This broadcasted data frame is stored in memory on each node. When performing a join operation, Spark can now use this in-memory copy instead of shuffling

data across the network. Ideal for joining a large DataFrame with a small DataFrame. Particularly useful when one DataFrame can fit comfortably in memory.

- explode() method: to transform an array column into multiple rows, creating a separate row for each element in the array. It takes an array column as input. Creates a new row for each element in the array. All other columns are duplicated for each new row. Useful when you need to perform operations on individual elements of an array column. Commonly used before aggregations or joins involving array elements.

Data Partitioning Using Connector Configuration
Reference [Configuration Options — MongoDB Spark Connector]

Added these to configuration:

```
    .config("spark.mongodb.input.partitioner", "MongoSplitVectorPartitioner") \
    .config("spark.mongodb.input.partitionerOptions.partitionKey", "_id") \
    .config("spark.mongodb.input.partitionerOptions.numberOfPartitions", "8") \
```

The MongoSplitVectorPartitioner is a tool or component designed to partition data in MongoDB databases, specifically for standalone instances or replica sets. It uses MongoDB's internal splitVector command to divide the data into partitions, helping with data distribution and parallel processing.

Using the Repartition Function in Queries
Reference https://sparkbyexamples.com/pyspark/pyspark-repartition-usage/

Whenever a dataframe is made, we can repartion it. For example:

```
def query1(spark):
    students_df = spark.read.format("mongodb") \
        .option("uri", "mongodb://localhost:27017/university_db.students") \
        .option("collection", "students") \
        .load()

    # Repartition based on student_id (assuming it's unique)
    students_df = students_df.repartition(8, col("_id"))

    result = students_df.filter(array_contains(col("enrolled_courses"), 15)).select("student_name", "student_email")
    return result
```

The repartition function redistributes the data across a specified number of partitions in a Spark DataFrame. In this case, the data in the students_df DataFrame is being repartitioned into 8 partitions based on the _id column (assumed to be a unique identifier for students).

Note
The partitioning methods allows spark to process the data in parallel

Performance Comparison

1. Basic Queries (Without any optimization)

```
filename,run_time
./query_outputs\query1_result,0.6057071685791016
./query_outputs\query2_result,0.19138336181640625
./query_outputs\query3_result,0.06599974632263184
./query_outputs\query4_result,0.10014677047729492
./query_outputs\query5_result,0.21362900733947754
./query_outputs\query6_result,0.09799718856811523

```

2. Basic Queries + Optimization as Indexing in MongoDB

```
filename,run_time
./query_outputs_indexing\query1_result,0.5386350154876709
./query_outputs_indexing\query2_result,0.21334552764892578
./query_outputs_indexing\query3_result,0.07830405235290527
./query_outputs_indexing\query4_result,0.11609625816345215
./query_outputs_indexing\query5_result,0.2060995101928711
./query_outputs_indexing\query6_result,0.10352706909179688

```

3. Optimization as Spark Optimized Queries

```
filename,run_time
./query_outputs_spark_optimized\query1_result,0.5189454555511475
./query_outputs_spark_optimized\query2_result,0.21600723266601562
./query_outputs_spark_optimized\query3_result,0.11481499671936035
./query_outputs_spark_optimized\query4_result,0.10673213005065918
./query_outputs_spark_optimized\query5_result,0.5658690929412842
./query_outputs_spark_optimized\query6_result,0.16176605224609375
```

4. Basic Queries + Optimization by Data Paritioning using Connector

```
filename,run_time
./query_outputs_partioning\query1_result,0.4020414352416992
./query_outputs_partioning\query2_result,0.1828770637512207
./query_outputs_partioning\query3_result,0.09073185920715332
./query_outputs_partioning\query4_result,0.14027905464172363
./query_outputs_partioning\query5_result,0.26857447624206543
./query_outputs_partioning\query6_result,0.1437511444091797
```

5. Optimization by Data Paritioning in dataframes generated in Queries

```
filename,run_time
./query_outputs_repartitioning\query1_result,0.6056199073791504
./query_outputs_repartitioning\query2_result,0.343670129776001
./query_outputs_repartitioning\query3_result,0.09524202346801758
./query_outputs_repartitioning\query4_result,0.15517091751098633
./query_outputs_repartitioning\query5_result,0.27550697326660156
./query_outputs_repartitioning\query6_result,0.14939498901367188
```

Analysis of performance, and impact of different strategies:

- Metric used to compare performance: Run time of query.
- The first query takes more time because of the first initialization and setup it has to do, it's not like it is a big command, this is the reason.
- How different strategies impact is mentioned above in the report under their respective sections
- A question comes to our mind. Why can not we see the performance boost?
  The answer is a small data size. This assignment focused on teaching various technologies and techniques used in the industry. Because I had to design the the database myself, the data size was very small. However these strategies will work wonders on large datasets.