# Question-4: Pig Query

Note: To run a pig query and make a time comparison, we can simply put:

```
time pig -x local sample.pig
```

The scripts that were used for querying are their in the **Q4 directory**.
The output from the pig query is there in the output directory

## Query 1: CGPA Calculation per Student

**Objective:**
Calculate the CGPA for each student along with their total credits completed using the institutional grading system.

**Steps and Logic:**

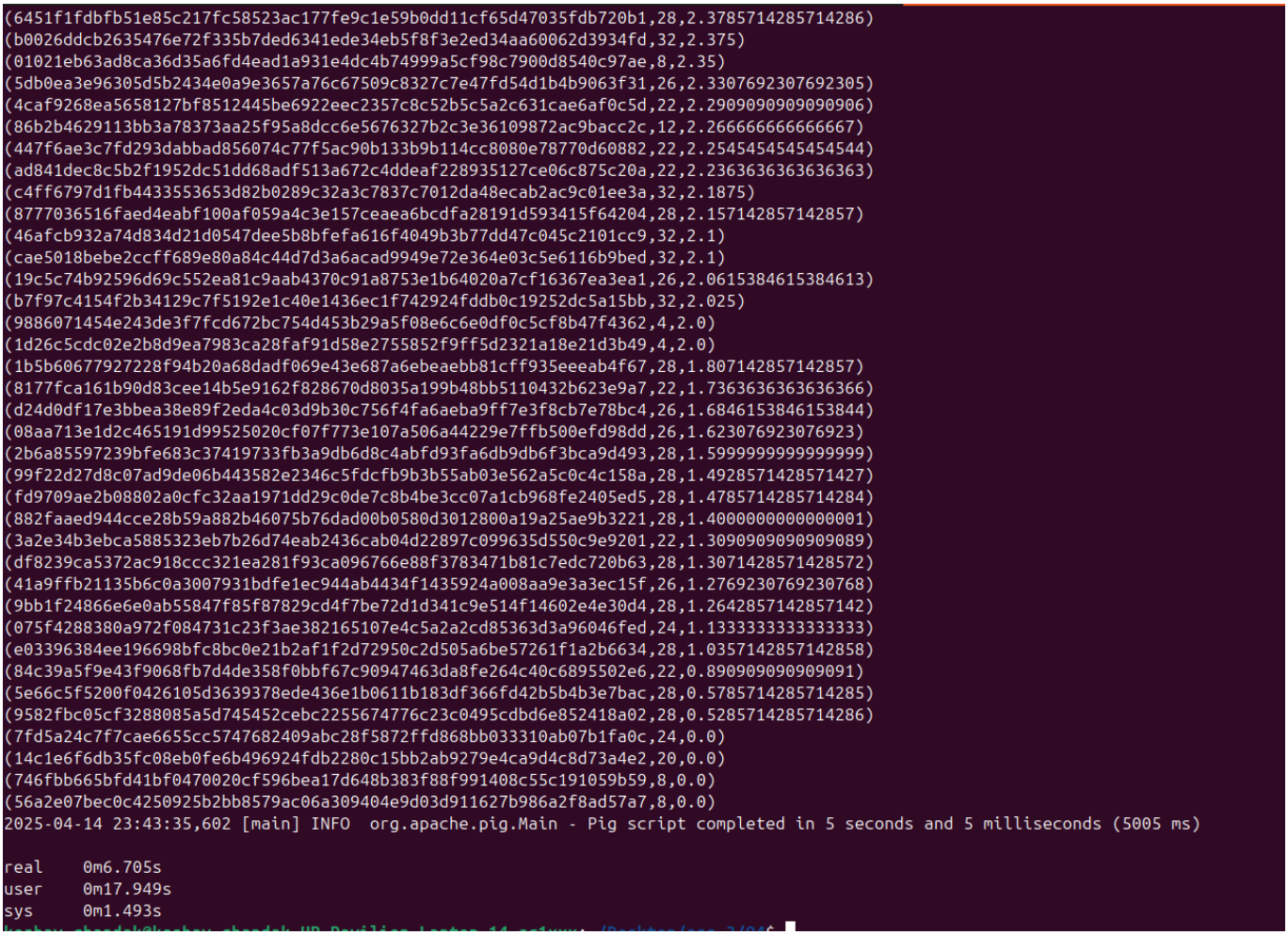
1. **Loading Data:**
   - Load `enrollment.csv` into `enrollment_data` and drop the header row.
   - Load `grade.csv` into `grade_roster` and filter out its header row.
   - Load `attendance.csv` and `fact_table_final1.csv` similarly, ensuring that header rows are filtered out.
2. **Joining Datasets:**
   - Join `grade_roster` and `fact_table` on matching student and course identifiers (i.e. `student_id` with `member_id` and `subject_code_name` with `course`).
3. **Calculating Weighted Points:**
   - Use a `CASE` expression within a `FOREACH` to calculate the weighted points for each course based on the letter grade (e.g., `'A'` as 4.0, `'A-'` as 3.7, etc.) multiplied by the course credit.
4. **Grouping and Aggregation:**
   - Group the resulting data by `student_id` to aggregate values.
   - Compute the total credits by summing `course_credit` and the total weighted points.
   - Calculate the CGPA as the ratio of total weighted points to total credits.
5. **Ordering and Storing:**
   - Order the results by CGPA (in descending order) and by total credits completed.
   - Dump and store the output using PigStorage into the `/output/Query-1` directory.

**Pig Script Excerpt:**

```
-- Join grade_roster and fact_table
joined_data = JOIN grade_roster BY (student_id, subject_code_name),
```

```
fact_table BY (member_id, course);

-- Calculate weighted points
cgpa_data = FOREACH joined_data GENERATE
    grade_roster::student_id AS student_id,
    grade_roster::course_credit AS course_credit,
    (CASE grade_roster::obtained_marks_grade
        WHEN 'A'  THEN 4.0 * grade_roster::course_credit
        WHEN 'A-' THEN 3.7 * grade_roster::course_credit
        WHEN 'B+' THEN 3.4 * grade_roster::course_credit
        WHEN 'B'  THEN 3.0 * grade_roster::course_credit
        WHEN 'B-' THEN 2.7 * grade_roster::course_credit
        WHEN 'C+' THEN 2.4 * grade_roster::course_credit
        WHEN 'C'  THEN 2.0 * grade_roster::course_credit
        WHEN 'D'  THEN 1.7 * grade_roster::course_credit
        ELSE 0.0
    END) AS weighted_points;

-- Group and compute totals and CGPA
grouped_data = GROUP cgpa_data BY student_id;
result = FOREACH grouped_data {
    total_credits = SUM(cgpa_data.course_credit);
    total_weighted_points = SUM(cgpa_data.weighted_points);
    cgpa = total_weighted_points / total_credits;
    GENERATE group AS student_id, total_credits AS total_credits_completed,
cgpa AS cgpa;
}
ordered_result = ORDER result BY cgpa DESC, total_credits_completed DESC;

DUMP ordered_result;
STORE ordered_result INTO '/output/Query-1' USING PigStorage(',');
```

**Image**




**Elapsed time:5.005 seconds**

## Query 2: Faculty-wise Summary of Attendance and Course Credit

**Objective:**
Calculate the number of students, average attendance percentage, and maximum course credit for each faculty by filtering for passed students.

**Steps and Logic:**

1. **Joining Datasets:**
   - Join the `fact_table` and `grade_roster` on student and course identifiers.
2. **Filtering:**
   - Filter the joined dataset for records where the exam result is `'Pass'` to focus on successful outcomes.
3. **Grouping by Faculty:**
   - Group the filtered records by `faculty_name`.
4. **Aggregation:**
   - Count distinct students per faculty.

- Compute the average attendance using the `average_attendance_percent` from the fact table.
- Determine the maximum course credit awarded for courses taught by each faculty.

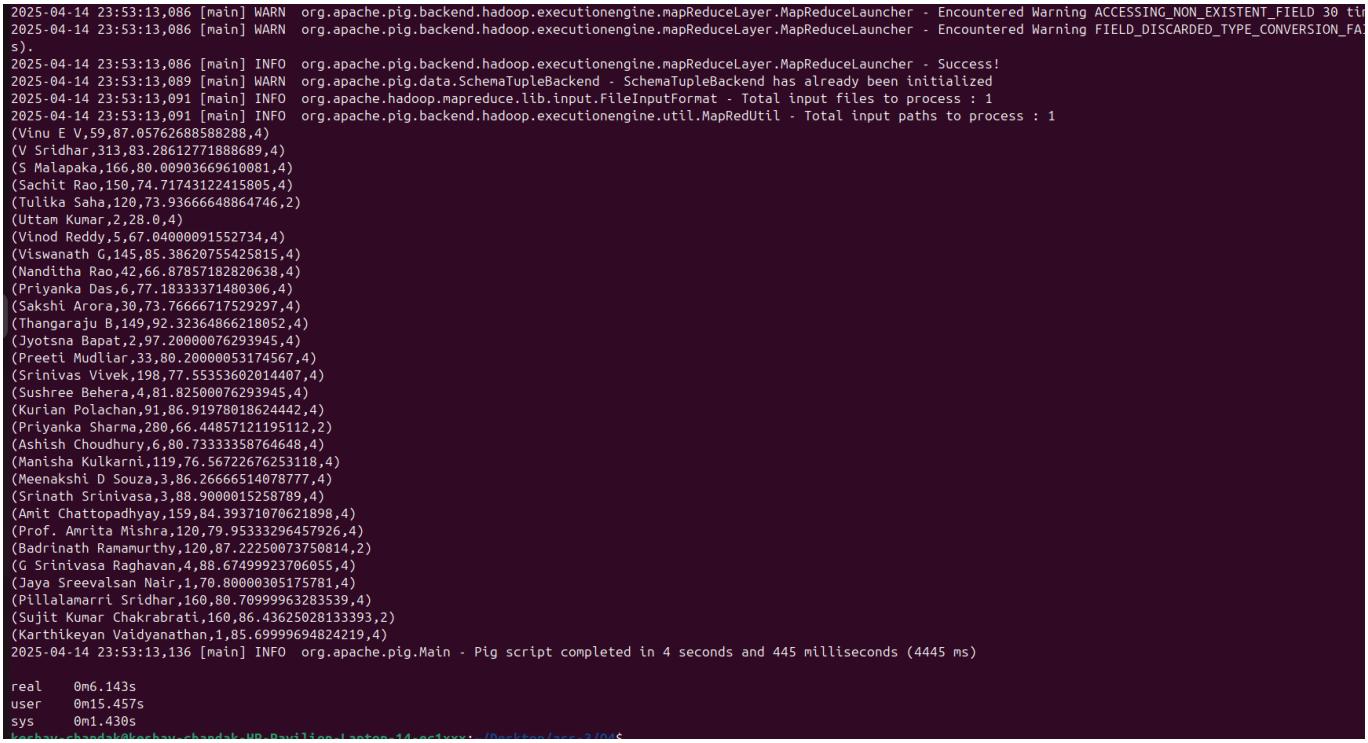
5. **Output:**

- Dump the results and store them into `/output/Query-2`.

**Pig Script Excerpt:**

```
-- Join fact_table and grade_roster
joined_data = JOIN fact_table BY (member_id, course), grade_roster BY
(student_id, subject_code_name);

-- Filter for students who passed
filtered_data = FILTER joined_data BY grade_roster::exam_result == 'Pass';

-- Group by faculty_name and compute aggregates
grouped_data = GROUP filtered_data BY grade_roster::faculty_name;
result = FOREACH grouped_data {
    unique_students = DISTINCT filtered_data.grade_roster::student_id;
    GENERATE group AS faculty_name,
             COUNT(unique_students) AS num_students,
             AVG(filtered_data.fact_table::average_attendance_percent) AS
avg_attendance,
             MAX(filtered_data.grade_roster::course_credit) AS
max_course_credit;
}

DUMP result;
STORE result INTO '/output/Query-2' USING PigStorage(',');
```

**Image**




**Elapsed time:4.445 seconds**

## Query 3: Identify Low Attendance (Below 75%) per Student-Course

**Objective:**
Determine the attendance percentage for each student in each course and identify those records where attendance is below 75%.

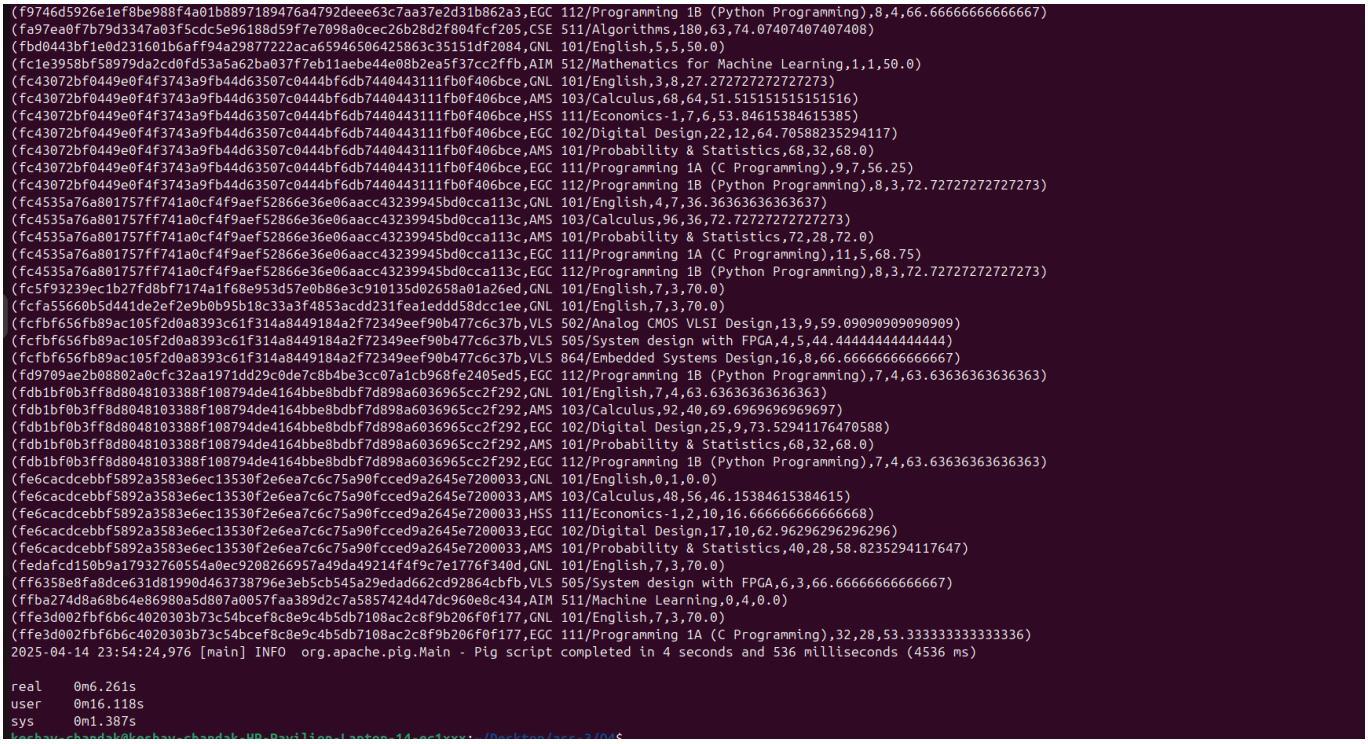
**Steps and Logic:**

1. **Joining Datasets:**
   - Join `fact_table` with `grade_roster` on matching student and course identifiers.
2. **Grouping Data:**
   - Group the joined data by both `student_id` and `subject_code_name` to work at the granularity of each student's course.
3. **Attendance Calculation:**
   - Compute total classes attended and absent for each group.
   - Calculate the overall attendance percentage using the formula: $$\text{attendance_percentage} = \frac{\text{total attended} \times 100}{\text{total attended} + \text{total absent}}$$
4. **Filtering:**
   - Filter out groups where the attendance percentage is less than 75%.
5. **Output:**
   - Dump and store the final filtered output into `/output/Query-3`.

**Pig Script Excerpt:**

```
-- Join fact_table and grade_roster
joined_data = JOIN fact_table BY (member_id, course), grade_roster BY
(student_id, subject_code_name);

-- Group by student_id and subject_code_name
grouped_data = GROUP joined_data BY (grade_roster::student_id,
grade_roster::subject_code_name);

-- Calculate attendance metrics per group
attendance_data = FOREACH grouped_data {
    total_attended =
SUM(joined_data.fact_table::number_of_classes_attended);
    total_absent = SUM(joined_data.fact_table::number_of_classes_absent);
    attendance_percentage = (total_attended * 100.0) / (total_attended +
total_absent);
    GENERATE FLATTEN(group) AS (student_id, course),
             total_attended AS total_classes_attended,
             total_absent AS total_classes_absent,
             attendance_percentage AS overall_attendance_percentage;
}

-- Filter groups with attendance below 75%
filtered_attendance = FILTER attendance_data BY
overall_attendance_percentage < 75;

DUMP filtered_attendance;
STORE filtered_attendance INTO '/output/Query-3' USING PigStorage(',');
```

**Image**




**Elapsed time:4.536 seconds**

# Comparison of Hive vs. Pig

Table 1: Comparison of HiveQL Query Execution and Pig Query Execution

| Query | Hive Without Bucketing(seconds) | Hive With Bucketing (seconds) | Pig performance (seconds |
|---|---|---|---|
| Query 1 | 6.54 | 1.643 | 5.005 |
| Query 2 | 0.912 | 0.793 | 4.445 |
| Query 3 | 1.23 | 1.018 | 4.536 |

1. **Installation & Setup**
    - **Hive:**
        - Typically involves setting up a Hive metastore along with Hadoop.
        - More components (HiveServer2, Metastore, etc.) need to be configured.
        - Can be complex to install and manage, especially in a production environment.
    - **Pig:**
        - Generally easier to install and lightweight.
        - Runs as a single script without the need for a separate metastore.
        - Quick to set up on local mode or within a Hadoop cluster.
2. **Query Language & Ease of Writing**
    - **Hive:**
        - Uses a SQL-like language (HiveQL) that is familiar to users with a relational database background.
        - Declarative queries make it easier for those accustomed to SQL.
        - Built-in functions and windowing can make complex queries simpler.
    - **Pig:**
        - Uses a scripting language called Pig Latin, which is procedural.
        - Offers more flexibility and control when writing data transformation logic.
        - Can be easier for iterative data processing tasks, but may require more lines of code for similar SQL operations.
3. **Query Performance & Optimization**
    - **Hive:**
        - Optimized for complex, long-running queries over large datasets.
        - Supports indexing, partitioning, and bucketing, which can significantly improve query performance when properly tuned.
        - More suitable for batch processing analytical queries.
    - **Pig:**
        - Also handles large datasets but can be more efficient for ETL tasks and transformations.
        - Performance can be comparable to Hive for many transformation operations; however, highly optimized **Hive queries may outperform Pig on complex aggregations.**
        - Less emphasis on indexing and more on user-defined optimizations via scripting logic.
4. **Suitability & Use Cases**
    - **Hive:**

        - Best suited for analysts comfortable with SQL.
        - Ideal for ad hoc queries and reporting where the data schema is well-defined.
        - Strong integration with BI tools and reporting systems.
    - **Pig:**
        - Excellent for ETL workflows and data processing pipelines.
        - Preferred when you need fine-grained control over data transformations.
        - Often used in scenarios where rapid prototyping of data flows is required.

5. **Community & Ecosystem**

    - **Hive:**
        - Widely adopted in enterprises, with robust community support and integration with many Hadoop components.
        - Part of the broader SQL-on-Hadoop ecosystem.
    - **Pig:**
        - Once very popular for data processing tasks, but usage has decreased in favor of Spark and other processing frameworks.
        - Still a viable option for specific transformation-heavy workflows.

In the table as well, we can see that Hive outperforms Pig on almost all queries in terms of time, and when Hive is optimised with partitioning and bucketing, it always outperforms Pig.
Also, it was a little difficult to write pig scripts since it is easier to write SQL statements, which is similar to Hive queries.