CIS 552-Database Design

Final Project

Professor:

Name: Yukui Luo, PhD

Email: yluo2@umassd.edu

Name	Student Id
Veera Venkata Sai Deekshith Mamillapalli	02084126
Nagendra Muchinapalli	02085892

Student Enrollment and Academic Performance Tracking

Introduction:

The introduction of our Student Performance and Tracking System signifies a transformative leap in how our educational institution approaches the management and oversight of student academics. This state-of-the-art system has been meticulously crafted to bring automation and cohesion to several critical educational processes, including student enrollment, course management, and academic performance evaluation. Its primary objective is to streamline the enrollment process, making it more efficient and less cumbersome for both students and administrators. Simultaneously, it offers an effective solution for course management, encompassing aspects like scheduling, resource allocation, and curriculum tracking. A notable feature of this system is its capability to monitor and analyze student academic performance in real time. This functionality is pivotal in identifying and addressing educational needs promptly, ensuring that students receive the support they need to succeed. By moving away from traditional manual methods and embracing this automated, integrated system, the institution is poised to significantly enhance administrative efficiency, reduce errors associated with manual data handling, and foster a more engaging and responsive educational environment.

Problem Statement:

In our educational institution, there is an urgent need for a comprehensive and advanced system to manage Student Enrollment and Academic Performance Tracking. The current processes in place, which are mostly manual, are handling student enrollment, managing courses, and monitoring academic performance.

These procedures are not only consuming a considerable amount of time but are also prone to inaccuracies. The reliance on manual methods for these critical functions has become increasingly problematic, leading to inefficiencies and potential mistakes in managing essential academic records. As a result, it has become clear that implementing an automated, integrated system is essential for streamlining these operations and improving overall accuracy and efficiency in academic administration.

Table Definitions:

We used 5 different entities related to each other, to design a database for our system.

1. Student Table

Purpose: Stores information about students.

Key Attributes:

studentId: A unique identifier for each student (Primary Key).

student_name: The full name of the student.

DOB: Date of birth of the student.

gender: Gender of the student.

2. Course Table

Purpose: Contains details about courses offered.

Key Attributes:

courseId: A unique identifier for each course (Primary Key).

sub: The subject or title of the course.

credit_hrs: Number of credit hours for the course.

dept: The department that offers the course.

3. Enrollment Table

Purpose: Records the enrollment of students in various courses.

Key Attributes:

eId: A unique identifier for each enrollment record (Primary Key).

studentId: References studentId in the Student Table (Foreign Key).

courseId: References courseId in the Course Table (Foreign Key).

semester: The academic semester during which the enrollment is valid.

4. Grades Table

Purpose: Keeps track of students' grades in their enrolled courses.

Key Attributes:

gradeId: A unique identifier for each grade record (Primary Key).

studentId: References studentId in the Student Table (Foreign Key).

courseId: References courseId in the Course Table (Foreign Key).

grade: The grade (e.g., A, B, C) achieved by the student.

semester: The semester for which this grade is recorded.

5. Attendance Table

Purpose: Monitors the attendance of students in their courses.

Key Attributes:

id: A unique identifier for each attendance record (Primary Key).

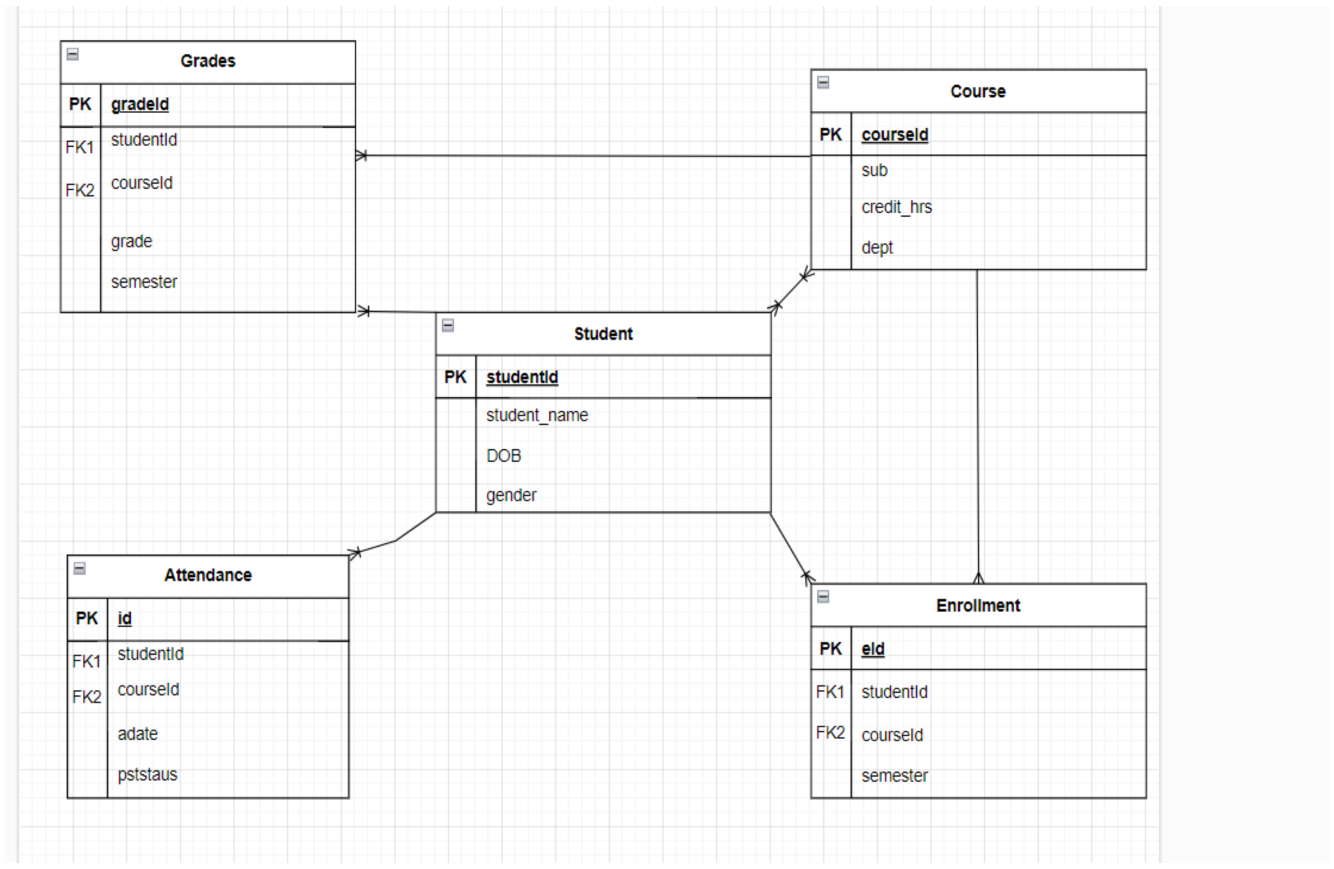
studentId: References studentId in the Student Table (Foreign Key).

courseId: References courseId in the Course Table (Foreign Key).

adate: The date of the class.

pstatus: Attendance status (e.g., Present, Absent).

Entity-Relationship Diagram (ERD):



Entity Relationships:

Student-Course Relationship: Many-to-many, resolved by the Enrollment entity.

Student-Grades Relationship: One-to-many (one student, many grades).

Course-Grades Relationship: One-to-many (one course, many students' grades).

Student-Attendance Relationship: One-to-many (one student, many attendance records).

Course-Enrollment Relationship: One-to-many (one course, many enrollments).

Student-Enrollment Relationship: One-to-many (one student, many enrollments).

Normalization:

Normalization in the context of database systems is a process used to organize data to reduce redundancy and improve data integrity. It involves structuring a relational database in accordance with a series of so-called "normal forms" in order to reduce data redundancy and improve data integrity. The primary goal of normalization is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.

First Normal Form (1NF):

- Ensures that the table is free from repeating groups or arrays.
- Each field contains only atomic (indivisible) values, and each record is unique.

Second Normal Form (2NF):

- Achieved when a database is in 1NF and all the non-key attributes are fully functional dependent on the primary key.
- It eliminates redundancy and dependency issues.

Third Normal Form (3NF):

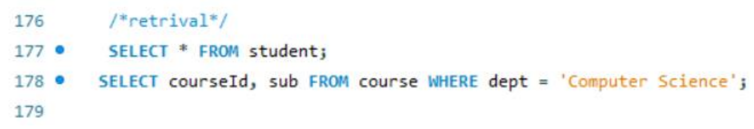
- A database is in 3NF if it is in 2NF and all its columns are not transitively dependent on the primary key.
- Ensures that all column values are only dependent on the primary key.

Benefits of Normalization

- **Reduces Data Redundancy**: Avoids duplicate data, thereby saving storage space and ensuring consistency.
- **Improves Data Integrity**: By reducing redundancy, normalization makes it easier to maintain data accuracy and integrity.
- **Facilitates Database Design**: Helps in structuring a database in a logical and efficient manner.

SQL queries and statements:

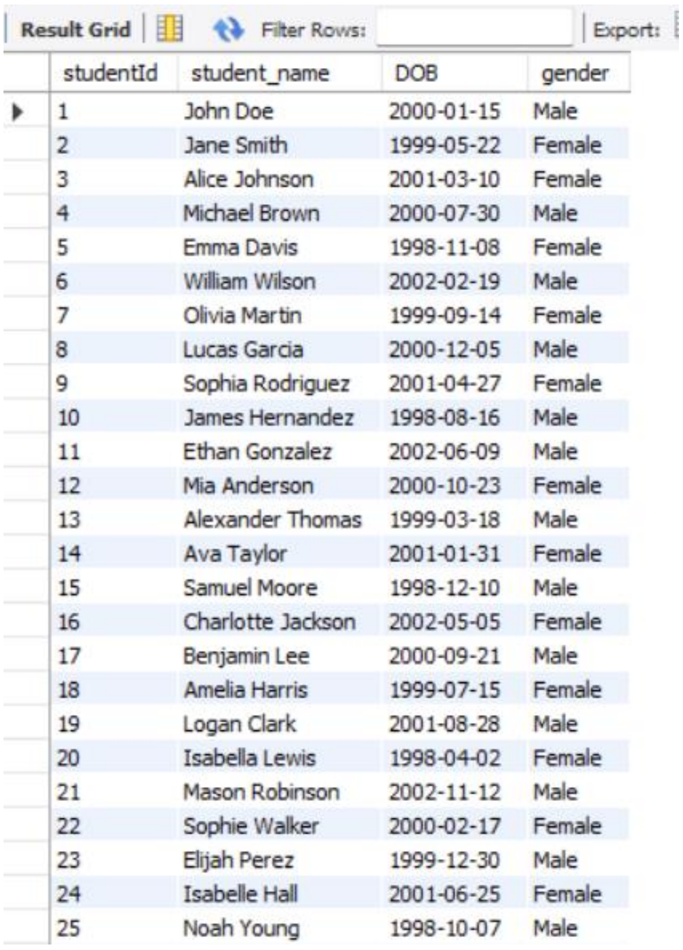
1)Data retrieval



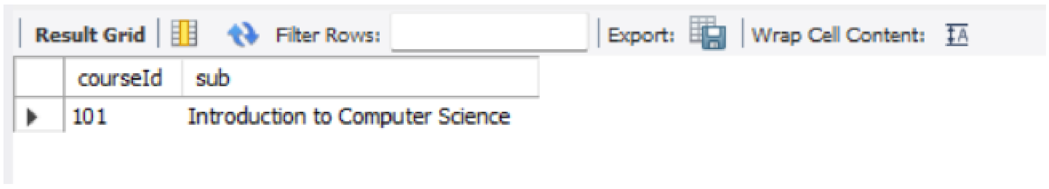
```
/*retrival*/
SELECT * FROM student;
SELECT courseId, sub FROM course WHERE dept = 'Computer Science';
```

The above statements will give list of all the students and course id and subject from from computer science department.

Output:

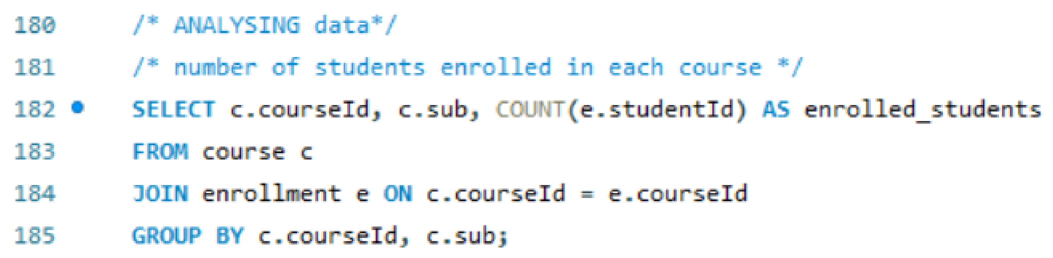


studentId	student_name	DOB	gender
1	John Doe	2000-01-15	Male
2	Jane Smith	1999-05-22	Female
3	Alice Johnson	2001-03-10	Female
4	Michael Brown	2000-07-30	Male
5	Emma Davis	1998-11-08	Female
6	William Wilson	2002-02-19	Male
7	Olivia Martin	1999-09-14	Female
8	Lucas Garcia	2000-12-05	Male
9	Sophia Rodriguez	2001-04-27	Female
10	James Hernandez	1998-08-16	Male
11	Ethan Gonzalez	2002-06-09	Male
12	Mia Anderson	2000-10-23	Female
13	Alexander Thomas	1999-03-18	Male
14	Ava Taylor	2001-01-31	Female
15	Samuel Moore	1998-12-10	Male
16	Charlotte Jackson	2002-05-05	Female
17	Benjamin Lee	2000-09-21	Male
18	Amelia Harris	1999-07-15	Female
19	Logan Clark	2001-08-28	Male
20	Isabella Lewis	1998-04-02	Female
21	Mason Robinson	2002-11-12	Male
22	Sophie Walker	2000-02-17	Female
23	Elijah Perez	1999-12-30	Male
24	Isabelle Hall	2001-06-25	Female
25	Noah Young	1998-10-07	Male



courseId	sub
101	Introduction to Computer Science

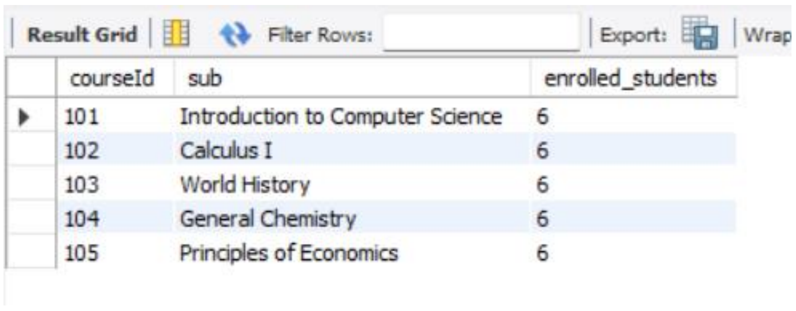
2)Data analysis



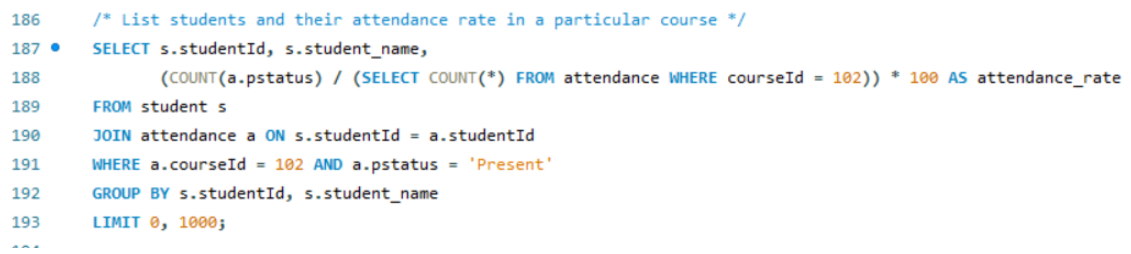
```sql
/* ANALYSING data*/
/* number of students enrolled in each course */
SELECT c.courseId, c.sub, COUNT(e.studentId) AS enrolled_students
FROM course c
JOIN enrollment e ON c.courseId = e.courseId
GROUP BY c.courseId, c.sub;
```

The above statements will give number of students enrolled in each course.

Output:



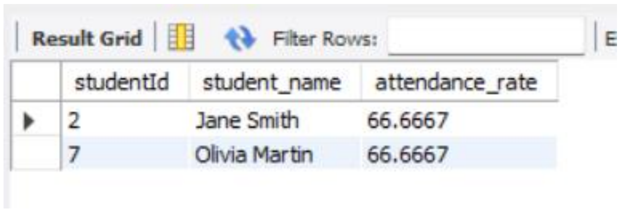
courseId	sub	enrolled_students
101	Introduction to Computer Science	6
102	Calculus I	6
103	World History	6
104	General Chemistry	6
105	Principles of Economics	6



```sql
/* List students and their attendance rate in a particular course */
SELECT s.studentId, s.student_name,
       (COUNT(a.pstatus) / (SELECT COUNT(*) FROM attendance WHERE courseId = 102)) * 100 AS attendance_rate
FROM student s
JOIN attendance a ON s.studentId = a.studentId
WHERE a.courseId = 102 AND a.pstatus = 'Present'
GROUP BY s.studentId, s.student_name
LIMIT 0, 1000;
```

The above sql queries will result in list of students and their attendance rate in a particular course.

Output:

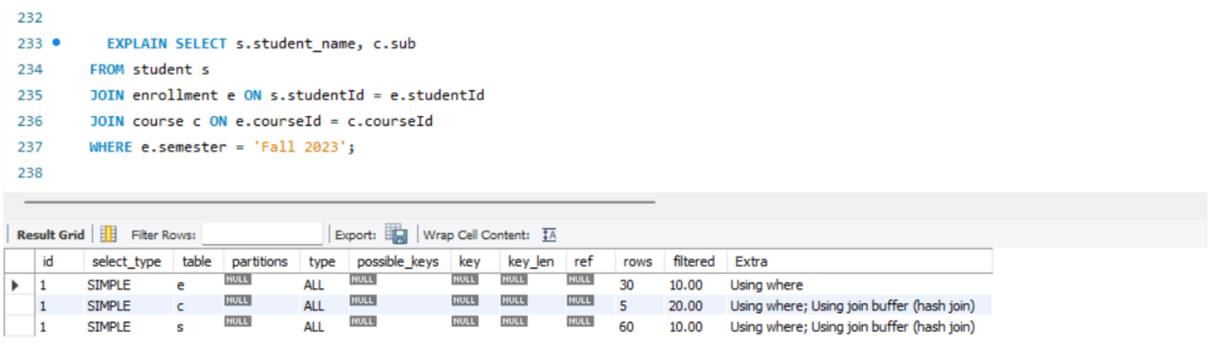


studentId	student_name	attendance_rate
2	Jane Smith	66.6667
7	Olivia Martin	66.6667

Performance Tuning:

Indexing and optimization are indispensable components in the realm of database management, addressing critical needs for improved system performance. By strategically employing indexes, databases can significantly enhance query execution speed, providing a structured pathway to access data efficiently. This proves particularly vital in large datasets where swift retrieval of information is paramount. Moreover, indexing plays a crucial role in minimizing disk I/O operations, reducing the strain on resources and contributing to a more responsive system. The efficiency of search operations is markedly improved through indexing, allowing databases to swiftly locate and retrieve specific records. Beyond mere speed enhancements, optimization, including indexing strategies, accelerates sorting and filtering processes, ensuring that data is organized and presented rapidly. Furthermore, indexing supports the enforcement of constraints, such as unique constraints and primary key constraints, promoting data integrity and reliability. In relational databases, properly indexed tables optimize join operations and aggregations, streamlining the combination of data from different tables and the calculation of aggregated values. Ultimately, the synergy of indexing and optimization techniques results in a more resource-efficient, scalable, and performant database system.

Let us consider an example and see how indexing is useful in optimization and improve efficiency.

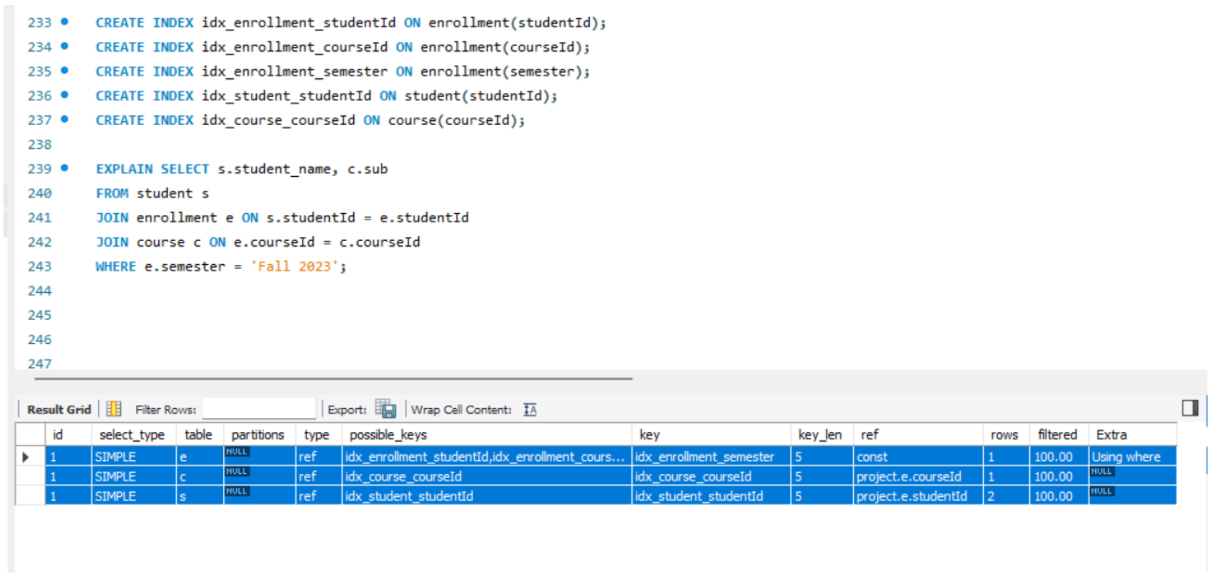


```sql
 EXPLAIN SELECT s.student_name, c.sub
FROM student s
JOIN enrollment e ON s.studentId = e.studentId
JOIN course c ON e.courseId = c.courseId
WHERE e.semester = 'Fall 2023';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	e	NULL	ALL	NULL	NULL	NULL	NULL	30	10.00	Using where
1	SIMPLE	c	NULL	ALL	NULL	NULL	NULL	NULL	5	20.00	Using where; Using join buffer (hash join)
1	SIMPLE	s	NULL	ALL	NULL	NULL	NULL	NULL	60	10.00	Using where; Using join buffer (hash join)

This above example shows how the query work without indexing.



```sql
CREATE INDEX idx_enrollment_studentId ON enrollment(studentId);
CREATE INDEX idx_enrollment_courseId ON enrollment(courseId);
CREATE INDEX idx_enrollment_semester ON enrollment(semester);
CREATE INDEX idx_student_studentId ON student(studentId);
CREATE INDEX idx_course_courseId ON course(courseId);

EXPLAIN SELECT s.student_name, c.sub
FROM student s
JOIN enrollment e ON s.studentId = e.studentId
JOIN course c ON e.courseId = c.courseId
WHERE e.semester = 'Fall 2023';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	e	NULL	ref	idx_enrollment_studentId,idx_enrollment_cours...	idx_enrollment_semester	5	const	1	100.00	Using where
1	SIMPLE	c	NULL	ref	idx_course_courseId	idx_course_courseId	5	project.e.courseId	1	100.00	NULL
1	SIMPLE	s	NULL	ref	idx_student_studentId	idx_student_studentId	5	project.e.studentId	2	100.00	NULL

The above query shows how the systems work with indexing.

The EXPLAIN output indicates efficient use of indexes in all parts of the query. The ref join type and low row counts suggest that the query is well-optimized. The indexes you have created (idx_enrollment_semester, idx_course_courseId, idx_student_studentId) are being effectively used. The low estimated row counts (1 for e and c, 2 for s) suggest that the query is accessing only a small number of rows, which is great for performance. No major inefficiencies are indicated in this EXPLAIN output. The query seems to be well-optimized in terms of index usage.

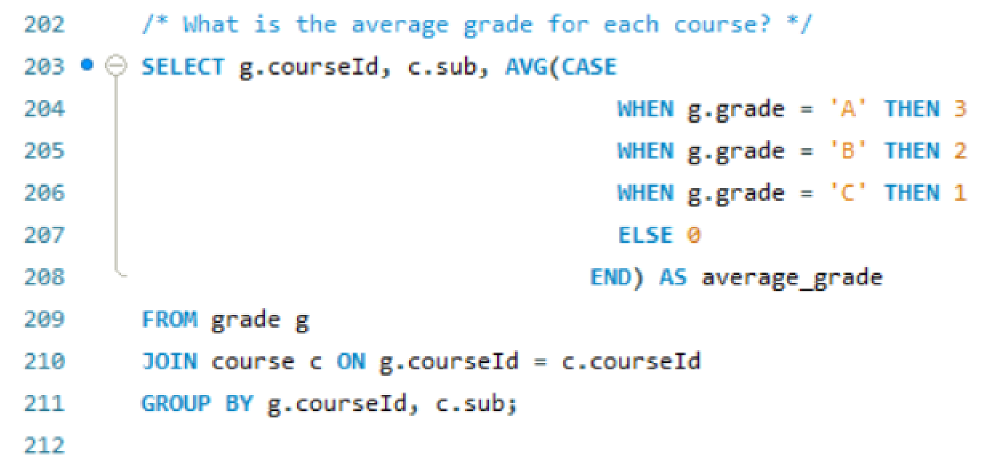
Indexing leads to enhanced performance, resulting in increased optimization.

Questions and Solutions:

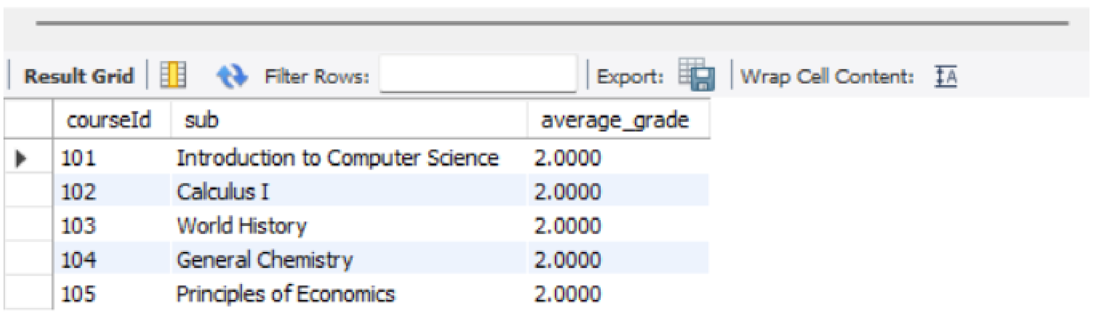
We will observe the outcomes that the database will yield in response to the queries.

1. What is the average grade for each course?
2. Calculate GPA for each student based on their grades.
3. Which students have an attendance rate below 75% in any course?

1.



```
/* What is the average grade for each course? */
SELECT g.courseId, c.sub, AVG(CASE
                                  WHEN g.grade = 'A' THEN 3
                                  WHEN g.grade = 'B' THEN 2
                                  WHEN g.grade = 'C' THEN 1
                                  ELSE 0
                                END) AS average_grade
FROM grade g
JOIN course c ON g.courseId = c.courseId
GROUP BY g.courseId, c.sub;
```



courseId	sub	average_grade
101	Introduction to Computer Science	2.0000
102	Calculus I	2.0000
103	World History	2.0000
104	General Chemistry	2.0000
105	Principles of Economics	2.0000

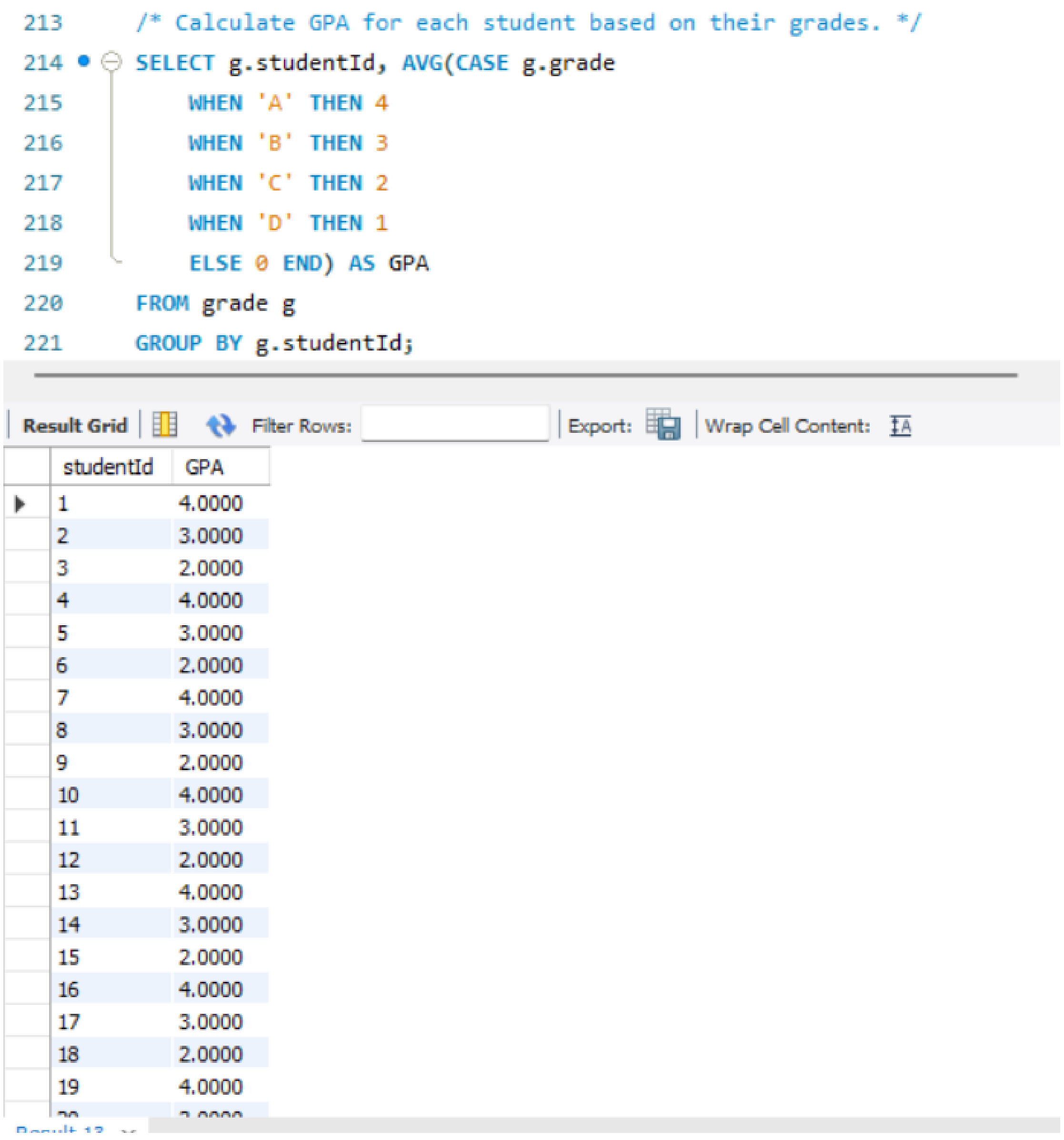
The provided SQL query joins the grade and course tables on their courseId columns, combining related records from both tables. It uses a CASE statement to convert letter grades from the grade table into numerical values, with 'A' as 3, 'B' as 2, 'C' as 1, and other grades as 0. The query then calculates the average of these numeric grades for each course, effectively representing the average performance in each course. The results are grouped by courseId and sub (subject) from the course table, and the query outputs these along with the calculated average grade for each course.

Tuple Relation Calculus:

π<sub>courseId, sub, AVG(GRADE(g))</sub> (σ<sub>∀c (c.courseId=g.courseId → c.sub=sub')</sub> (g, c))

- π represents projection
- σ represents selection
- `g` and `c` represent the relations `grade` and `course`
- `∀c (c.courseId=g.courseId → c.sub=sub')`: This predicate ensures that for every course (`c`) associated with a grade (`g`), the `sub` attribute of the course matches the desired subject (`sub'`)

2.



```
/* Calculate GPA for each student based on their grades. */
SELECT g.studentId, AVG(CASE g.grade
    WHEN 'A' THEN 4
    WHEN 'B' THEN 3
    WHEN 'C' THEN 2
    WHEN 'D' THEN 1
    ELSE 0 END) AS GPA
FROM grade g
GROUP BY g.studentId;
```

studentId	GPA
1	4.0000
2	3.0000
3	2.0000
4	4.0000
5	3.0000
6	2.0000
7	4.0000
8	3.0000
9	2.0000
10	4.0000
11	3.0000
12	2.0000
13	4.0000
14	3.0000
15	2.0000
16	4.0000
17	3.0000
18	2.0000
19	4.0000

Grade Conversion: It converts letter grades into numerical values using a CASE statement: 'A' is translated to 4, 'B' to 3, 'C' to 2, 'D' to 1, and all other grades to 0.

Average Calculation: The query computes the average of these numerical values for each student, representing their GPA.

Grouping: Results are grouped by studentId, ensuring that the GPA is calculated individually for each student.

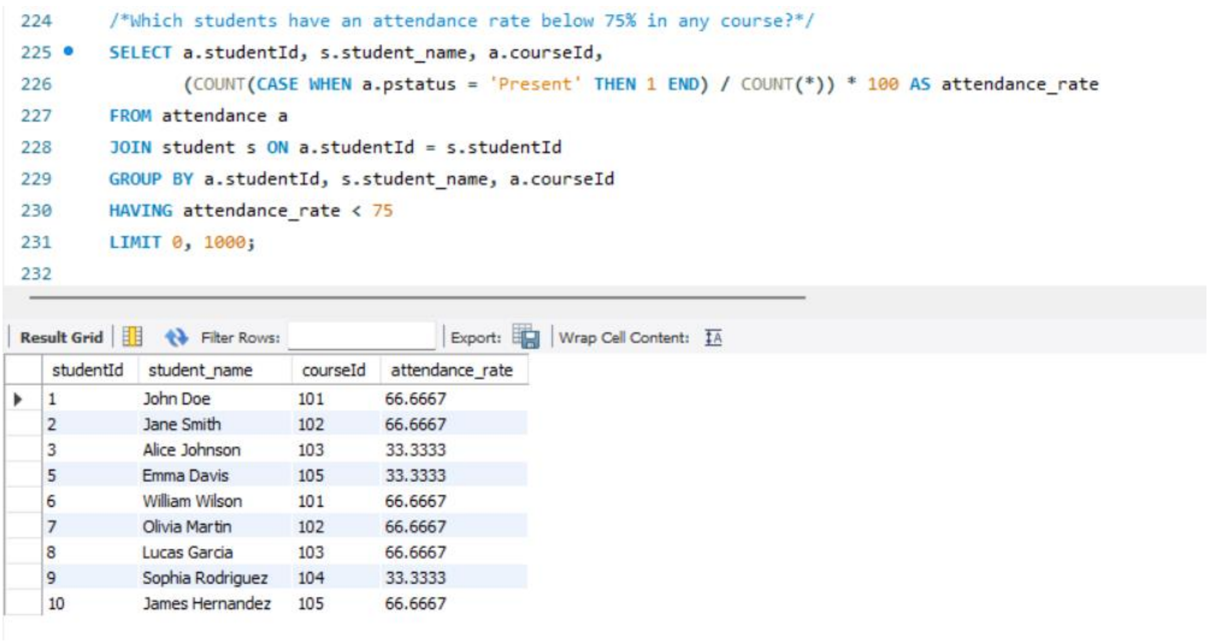
Output: It selects each student's studentId and their calculated GPA.

Tuple Relational Calculus:

```
π<sub>studentId, AVG(GRADE(g))</sub> (g)
```

- π represents projection
- g represents the relation `grade`.

3.



```
/*Which students have an attendance rate below 75% in any course?*/
SELECT a.studentId, s.student_name, a.courseId,
       (COUNT(CASE WHEN a.pstatus = 'Present' THEN 1 END) / COUNT(*)) * 100 AS attendance_rate
FROM attendance a
JOIN student s ON a.studentId = s.studentId
GROUP BY a.studentId, s.student_name, a.courseId
HAVING attendance_rate < 75
LIMIT 0, 1000;
```

studentId	student_name	courseId	attendance_rate
1	John Doe	101	66.6667
2	Jane Smith	102	66.6667
3	Alice Johnson	103	33.3333
5	Emma Davis	105	33.3333
6	William Wilson	101	66.6667
7	Olivia Martin	102	66.6667
8	Lucas Garcia	103	66.6667
9	Sophia Rodriguez	104	33.3333
10	James Hernandez	105	66.6667

The given SQL query calculates and retrieves the attendance rate of students for each course, specifically focusing on cases where the attendance rate is below 75%. Here's a brief explanation of its components:

Joining Tables: The query joins the attendance table (aliased as a) with the student table (aliased as s) on their studentId columns. This combines the attendance records with corresponding student names.

Calculating Attendance Rate: It calculates the attendance rate for each student in each course. This is done by counting the number of times a student is marked 'Present' in the attendance table (using a CASE statement), dividing this count by the total number of attendance records for that student in a course, and then multiplying by 100 to get a percentage.

Grouping Results: The results are grouped by both studentId and courseId, ensuring that the attendance rate is calculated for each student-course combination.

Filtering Results: The query uses a HAVING clause to filter out only those records where the attendance rate is below 75%.

Selecting Columns: It selects the studentId, student_name, and courseId, along with the calculated attendance_rate.

Limiting Results: The LIMIT 0, 1000 clause limits the results to the first 1000 records.

Tuple Relational Calculus:

π<sub>studentId, student_name, courseId, (COUNT(p: a.pstatus = 'Present') / COUNT(*)) * 100 AS attendance_rate</sub> (σ<sub>attendance_rate < 75 ∧ ∃s (a.studentId = s.studentId)</sub> (a ⋈ s))

- `π` represents projection
- `σ` represents selection
- `a ⋈ s`: This represents the join operation between the `attendance` (a) and `student` (s) relations based on the equality of `studentId`.
- `COUNT(p: a.pstatus = 'Present')`: This nested function counts the number of attendance records with `pstatus` equal to "Present" for each student in a course.
- `COUNT(*)`: This function counts the total number of attendance records for each student in a course.
- `(COUNT(p: a.pstatus = 'Present') / COUNT(*)) * 100`: This calculates the attendance rate (percentage of "Present" occurrences) for each student in a course.

Limitations:

1. Scalability

Data Volume: As the number of students, courses, and records grows, the system may face performance issues unless it's designed to scale efficiently.

Handling Concurrent Requests: In large institutions with many users, the system might struggle with a high volume of concurrent requests.

2. Data Quality and Accuracy

Data Entry Errors: Manual data entry can lead to errors and inconsistencies.

Up-to-Date Information: The system's effectiveness is contingent on the regular updating of its data.

3. System Functionality

Limited Analytics Capability: Depending on the implementation, the system might have limited capabilities in terms of advanced data analytics and reporting.

Feature Limitations: There might be features (like advanced student behavior tracking, predictive analytics, etc.) that are not supported.

4. Security and Privacy Concerns

Data Security: Protecting sensitive student information is crucial, and any vulnerability could lead to data breaches.

Compliance with Regulations: The system must comply with educational data privacy laws like FERPA, GDPR, etc.

6. User Interface and Experience

Ease of Use: A non-intuitive user interface can hinder the effectiveness of the system.

Accessibility: The system might not be fully accessible to all users, including those with disabilities.

7. Integration with Other Systems

Compatibility: Challenges may arise in integrating with other existing educational software systems or platforms.

Data Synchronization: Keeping data synchronized across different systems can be challenging.

8. Dependence on Technology

Technical Issues: The system's reliance on technology means that technical failures can disrupt its functionality.

Upgrades and Maintenance: Regular updates and maintenance are required, which can be resource-intensive.

9. User Adoption and Training

Learning Curve: Users (staff, faculty, and students) might need training to effectively use the system.

Resistance to Change: Resistance from users accustomed to previous systems or methods can hinder adoption.

Conclusion:

In conclusion, the implementation of a sophisticated Student Enrollment and Academic Performance Tracking System within your educational institution marks a significant step forward in modernizing academic administration. By transitioning from manual to automated processes, the institution stands to gain immensely in terms of efficiency, accuracy, and reliability.

This system will not only streamline student enrollment and course management but also enhance the monitoring and analysis of academic performance, leading to more informed decision-making. It will mitigate the risks associated with manual data handling, such as errors and data loss, and free up valuable time and resources that can be redirected towards more critical educational tasks. Furthermore, the system will provide real-time access to data, facilitating a more responsive and dynamic educational environment.

Overall, the new Student Enrollment and Academic Performance Tracking System is expected to transform the administrative landscape of the institution, paving the way for a more organized, data-driven, and student-centric approach to education management. This change is essential in adapting to the evolving needs of the educational sector and maintaining a high standard of academic excellence.

References:

1. https://kitaboo.com/student-performance-tracking-for-individualized-instruction/#:~:text=Student%20performance%20tracking%20systems%20are%20crucial%20tools%20that%20enable%20educators,and%20improve%20the%20outcomes%20accordingly.
2. https://files.eric.ed.gov/fulltext/EJ1136803.pdf
3. https://www.inettutor.com/diagrams/student-performance-tracking-er-diagram/
4. https://www.academia.edu/41728072/DESIGN_AND_IMPLEMENTATION_OF_STUDENT_PROGRESS_MANAGEMENT_SYSTEM

Git Repository:

https://github.com/SaiDeekshith17/DatabaseDesign-FinalProject