

Lab Exercise 2

Alessandro Andrei Araza

Edison Javier III

Alyza Paige Ng

October 5, 2023

Chapter 1	Dimensional Model	Page 2
------------------	--------------------------	---------------

Chapter 2	Questions	Page 3
------------------	------------------	---------------

Chapter 3	Write-Up, thoughts, reviews, comparisons	Page 11
------------------	---	----------------

3.1	Difference of Lab Exercise 1 and Lab Exercise 2	11
-----	---	----

Dimensional Model

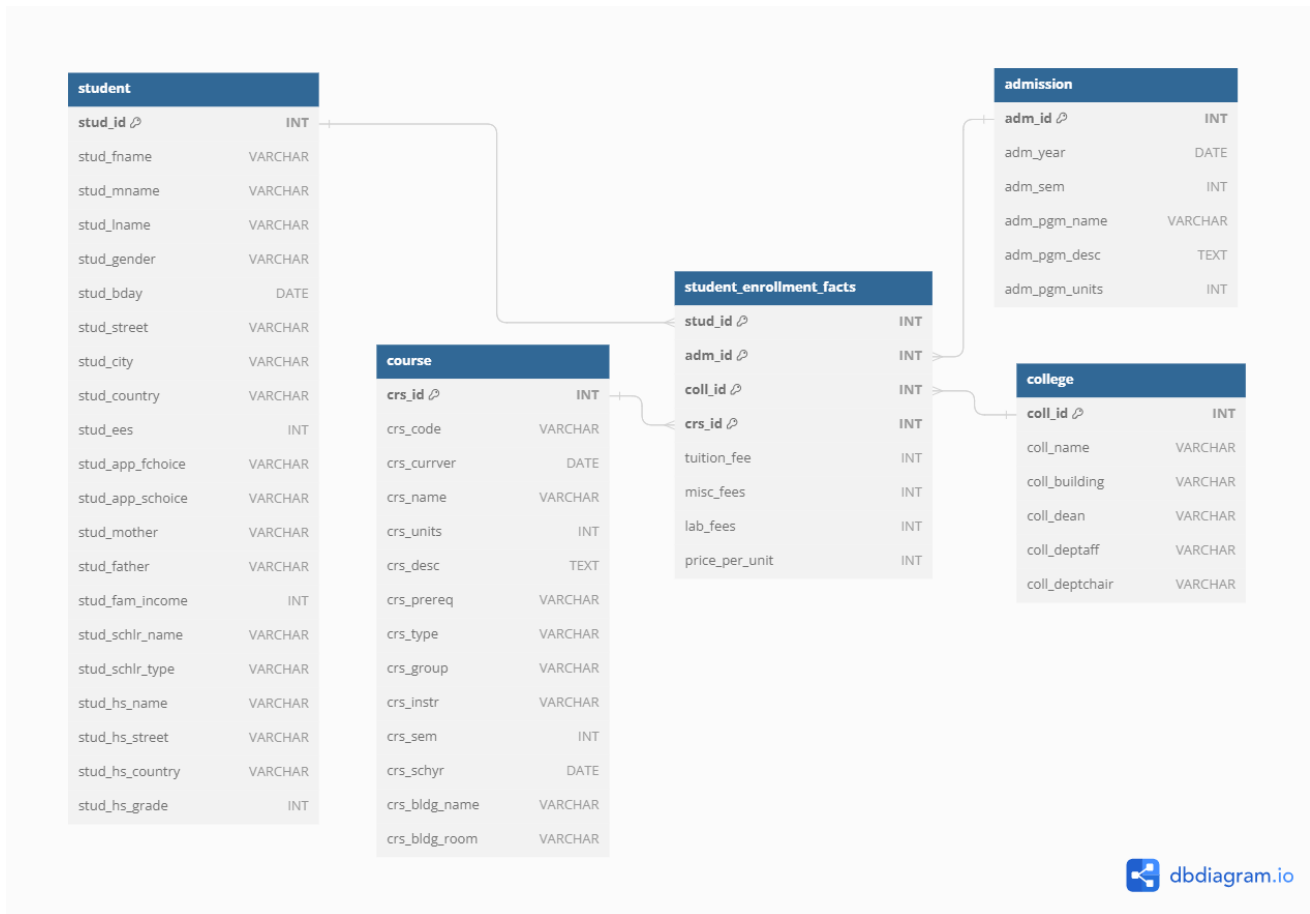


Figure 1.1: Dimensional Model; Refer to [this link](#) for the engine used to build this model

Questions

Question 1

What is the average tuition fee

Group by the enrollment records' college ID. Querying for the average should simply follow.

Solution: _____

```
create view tuitionPerCollege as
select college.coll_name as College, en.enr_tuition_fee as Tuition from enrollment en
join college on en.coll_id = college.coll_id group by college.coll_id;

select avg(Tuition) as "Average Tuition Fee" from tuitionPerCollege;
```

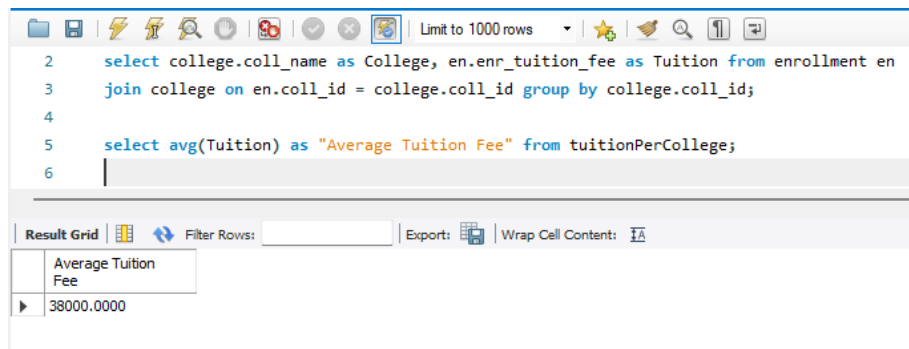


Figure 2.1: Question 1 Query and Output

Question 2

What is the average total school fees?

This directly adds all of the facts in the fact table since those refer to the school fees and then simply get the average of the sum.

Solution: _____

```
SELECT AVG(enr_tuition_fee + enr_misc_fee + enr_lab_fee + enr_ppu)
AS "Average Total School Fees" FROM enrollment;
```

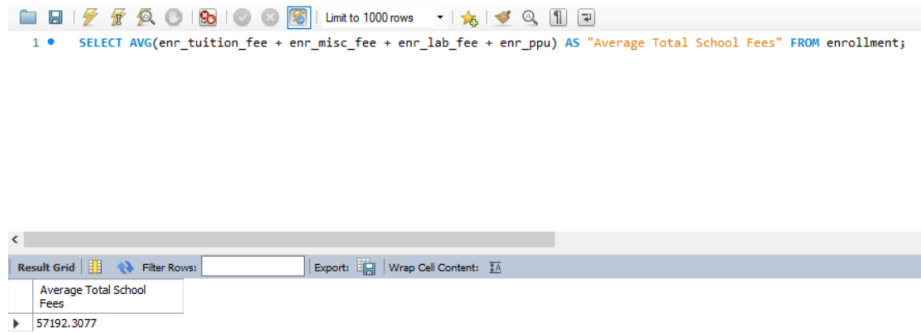


Figure 2.2: Question 2 Query and Output

Question 3

How many students are enrolled per subject?

Retrieve the number of students enrolled in each subject by joining the `course` and `enrollment` tables based on the course ID. `GROUP` the results by course name and `COUNT` the distinct student IDs to determine how many students are enrolled in each subject. The results are sorted in descending order using `DESC` to show the subjects with the highest enrollment first.

Solution:

```
SELECT c.crs_name AS Subject, COUNT(DISTINCT e.stud_id) AS Number_of_Students
FROM course c
LEFT JOIN enrollment e ON c.crs_id = e.crs_id
GROUP BY c.crs_name
ORDER BY Number_of_Students DESC;
```

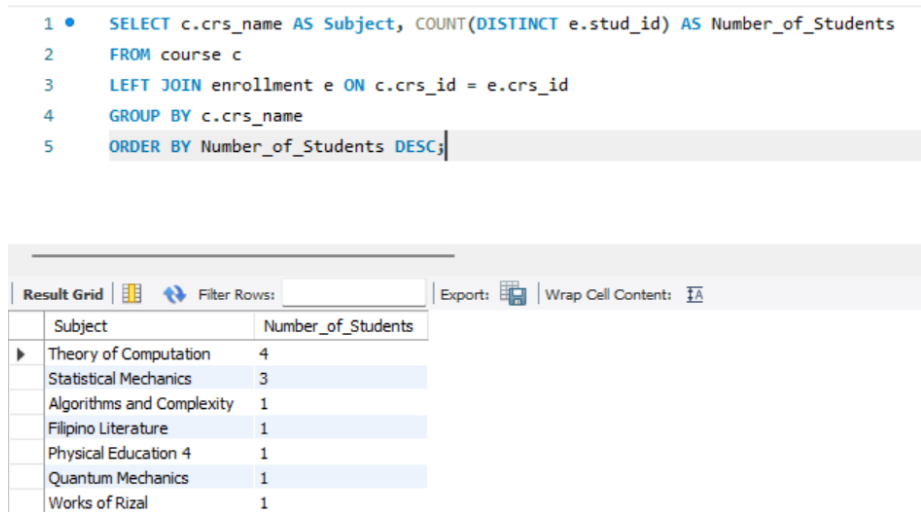


Figure 2.3: Question 3 Query and Output

Question 4

How many subjects does Taylor Sheesh have?

Create a view for Taylor's information. This view's `stud_id` serves as the filter for the query in the enrollment table.

Solution:

```
create view sheesh as
select * from student where stud_fname = "Taylor" and stud_lname = "Sheesh";

select count(crs_id) as "Taylor's Subjects" from enrollment en where
en.stud_id = (select stud_id from sheesh);
```

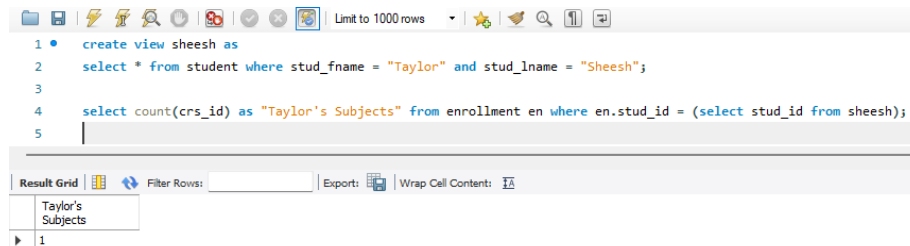


Figure 2.4: Question 4 Query and Output

Question 5

How many students have the same mother but different fathers?

This counts the number of students who have same mothers but different fathers by joining and comparing two student tables, and the **DISTINCT** keyword also ensures that each student is counted only once.

Solution:

```
SELECT COUNT(DISTINCT s1.stud_id) AS "Students with Same Mothers but Different Fathers"
FROM student s1
JOIN student s2 ON s1.stud_mother = s2.stud_mother AND s1.stud_father != s2.stud_father;
```

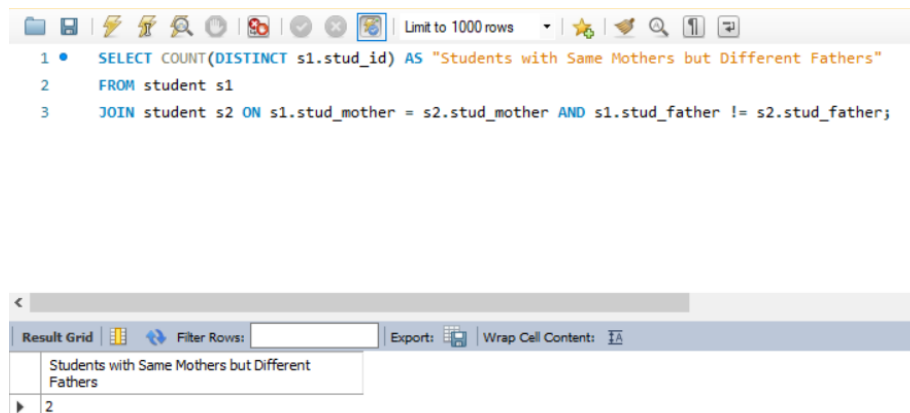


Figure 2.5: Question 5 Query and Output

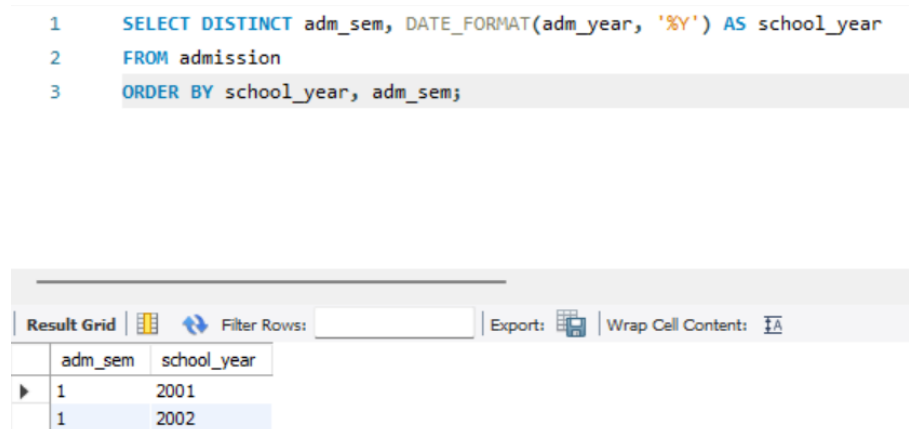
Question 6

What are the combinations of semester and school year?

Retrieve **DISTINCT** combinations of academic semesters (referred to as **adm_sem**) and the corresponding formatted admission years (as **school_year**) from the **admission** table. **ORDER** the results first by the school year and then by the academic semester, ensuring that the output is sorted in ascending order based on the school year and within the same year by academic semester.

Solution:

```
SELECT DISTINCT adm_sem, DATE_FORMAT(adm_year, '%Y') AS school_year
FROM admission
ORDER BY school_year, adm_sem;
```



The screenshot shows a SQL query editor with the following query:

```
1 SELECT DISTINCT adm_sem, DATE_FORMAT(adm_year, '%Y') AS school_year
2 FROM admission
3 ORDER BY school_year, adm_sem;
```

Below the query editor, the output is displayed in a table with the following columns: **adm_sem** and **school_year**. The table contains two rows:

adm_sem	school_year
1	2001
1	2002

Figure 2.6: Question 6 Query and Output

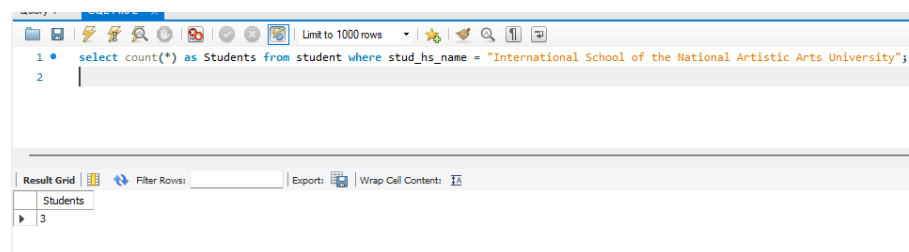
Question 7

How many students does International School of the National Artistic Arts University have?

Query the student table with a **WHERE** clause that only asks for students that have the ISNAA University in their records.

Solution:

```
select count(*) as Students from student where stud_hs_name =
"International School of the National Artistic Arts University";
```



The screenshot shows a SQL query editor with the following query:

```
1 select count(*) as Students from student where stud_hs_name = "International School of the National Artistic Arts University";
2
```

Below the query editor, the output is displayed in a table with the following columns: **Students**. The table contains one row with the value 3:

Students
3

Figure 2.7: Question 7 Query and Output

Question 8

How many courses are held in all the buildings?

This creates a view that selects the unique course code and building values from the course table using the DISTINCT keyword and then counts the number of courses grouped by the name of the building.

Solution:

```
CREATE VIEW crs_bldg_base AS
SELECT DISTINCT crs_code, crs_bldg_name
FROM course;

SELECT crs_bldg_name AS "Building Name", COUNT(*) AS "Number of Courses Held"
FROM crs_bldg_base
GROUP BY crs_bldg_name;
```

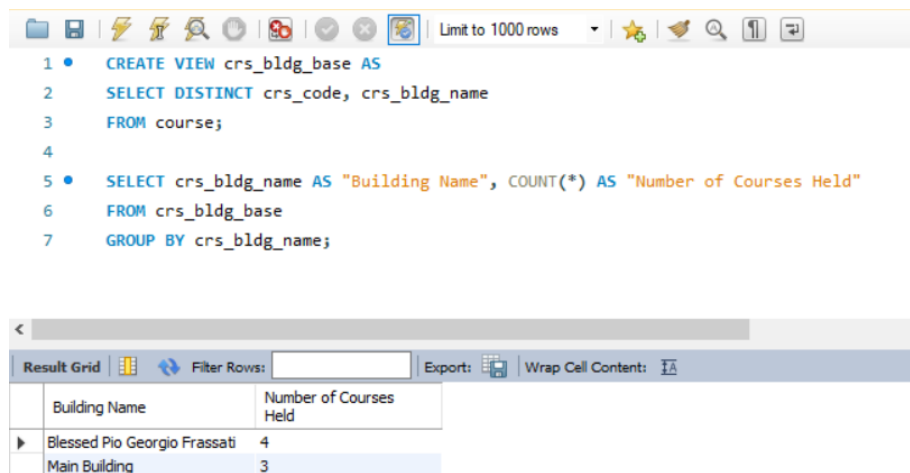


Figure 2.8: Question 8 Query and Output

Question 9

What is the average family income per scholarship?

Calculate the average family income per scholarship by first selecting DISTINCT scholarship names from the student table. Using AVG(), calculate the average family income for each scholarship and present the results in descending (denoted as DESC) order of average family income.

Solution:

```
SELECT stud_schlr_name AS scholarship_name, AVG(stud_fam_income) AS average_family_income
FROM student
WHERE stud_schlr_name IS NOT NULL
GROUP BY stud_schlr_name
ORDER BY average_family_income DESC;
```



```

1 • SELECT stud_schlr_name AS scholarship_name, AVG(stud_fam_income) AS average_family_income
2 FROM student
3 WHERE stud_schlr_name IS NOT NULL
4 GROUP BY stud_schlr_name
5 ORDER BY average_family_income DESC;

```

Result Grid		Filter Rows:	Exports:	Wrap Cell Contents:
scholarship_name	average_family_income			
nan	8200000.0000			
Entrance Merit Scholarship	1000000.0000			
POEA Scholarship	250000.0000			
DOST Scholarship	242500.0000			

Figure 2.9: Question 9 Query and Output

Question 10

Whose family has the least amount of money left after paying total school fees?

The base **view** serves as the central hub of all the information needed. It has the main price of their tuition fee, including miscellaneous and laboratory fees, the price of their course given that it's been multiplied to the price per unit of their college, as well as the income of their family.

the **courseTotals** aggregates all the price of the courses taken by each student, the **sum()** function was grouped according to the **stud_id** of each row.

deducts then calculates the total deductions and the remaining money of each family in the record. In this view, querying for the least amount of money left is possible by using the **min()** function to match the lowest record.

Solution:

```

create view base as
select en.stud_id, en.crs_id, (en.enr_tuition_fee + en.enr_misc_fee + en.enr_lab_fee)
as mainPrice, (en.enr_ppu * course.crs_units) as coursePrice, student.stud_fam_income
as income from enrollment en
join course on en.crs_id = course.crs_id
join student on en.stud_id = student.stud_id
group by en.crs_id;

create view courseTotals as
select stud_id, mainPrice, sum(coursePrice) as totals, income from base group by stud_id;

create view deducts as
select stud_id, (income - (mainPrice + totals)) as money from courseTotals;

select concat(student.stud_fname, " ", student.stud_lname) as Name, money from deducts
join student on deducts.stud_id = student.stud_id
where money = (select min(money) from deducts);

```

```

1 • create view base as
2 select en.stud_id, en.crs_id, (en.enr_tuition_fee + en.enr_misc_fee + en.enr_lab_fee) as mainPrice,
3 (en.enr_ppu * course.crs_units) as coursePrice, student.stud_fam_income as income from enrollment en
4 join course on en.crs_id = course.crs_id
5 join student on en.stud_id = student.stud_id
6 group by en.crs_id;
7
8 • create view courseTotals as
9 select stud_id, mainPrice, sum(coursePrice) as totals, income from base group by stud_id;
10
11 • create view deducts as
12 select stud_id, (income - (mainPrice + totals)) as money from courseTotals;
13
14 • select concat(student.stud_fname, " ", student.stud_lname) as Name, money from deducts
15 join student on deducts.stud_id = student.stud_id
16 where money = (select min(money) from deducts);
--

```

Result Grid

Name	money
▶ Zack Apron	80000

Figure 2.10: Question 10 Query and Output

Question 11

How many courses do each professor handle?

This counts uniquely the courses they handle, which are grouped according to the professor's name from the course table.

Solution:

```

SELECT crs_instr AS "Professor Name", COUNT(DISTINCT crs_code) AS "No. of Courses Handled"
FROM course
GROUP BY crs_instr;

```

```

1 • SELECT crs_instr AS "Professor Name", COUNT(DISTINCT crs_code) AS "No. of Courses Handled"
2 FROM course
3 GROUP BY crs_instr;

```

Result Grid

Professor Name	No. of Courses Handled
▶ Albert Trebla	2
Albus Subla	1
Henry Yrneh	3
Josie Eisoj	1
Madeline Eniledam	1

Figure 2.11: Question 11 Query and Output

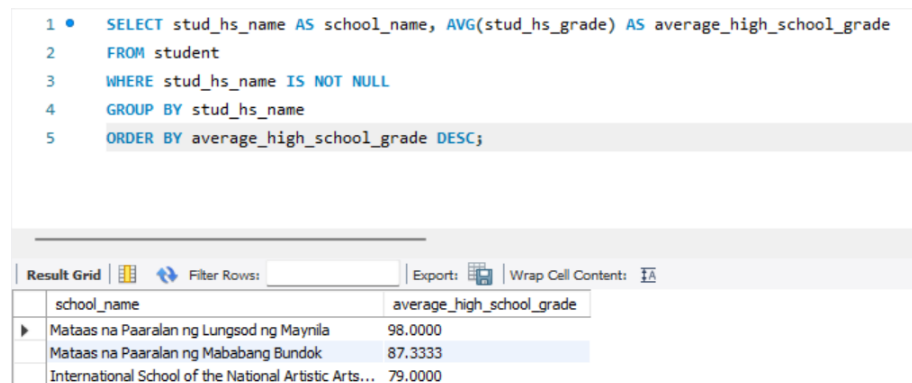
Question 12

What is the average high school grade for each school?

Calculate the average high school grade for each school by selecting **DISTINCT** school names from the **student** table. Compute the average of high school grades associated with each school using **AVG()**. Sort the results in descending order based on the average high school grade using **DESC**.

Solution:

```
SELECT stud_hs_name AS school_name, AVG(stud_hs_grade) AS average_high_school_grade
FROM student
WHERE stud_hs_name IS NOT NULL
GROUP BY stud_hs_name
ORDER BY average_high_school_grade DESC;
```



The screenshot shows a SQL query editor with five lines of code. Below the editor is a 'Result Grid' with a toolbar containing icons for grid view, refresh, filter rows, export, and wrap cell content. The grid displays the results of the query, sorted by average high school grade in descending order.

school_name	average_high_school_grade
Mataas na Paaralan ng Lungsod ng Maynila	98.0000
Mataas na Paaralan ng Mababang Bundok	87.3333
International School of the National Artistic Arts...	79.0000

Figure 2.12: Question 12 Query and Output

Write-Up, thoughts, reviews, comparisons

The Lab Exercise #2 significantly departed from our previous experience in Lab Exercise #1. Our group delved into the fascinating world of data modeling and schema design in this exercise. Specifically, we explored the dimensional model, a vital approach for structuring data in data warehousing and business intelligence scenarios.

The central focus of this exercise was the creation of a star schema, a powerful way to organize data into dimensions and facts. Our schema consisted of dimensions such as courses, students, enrollments, and admissions, with the student enrollment fact table at the core. Unlike the normalized schema, which prioritizes eliminating data redundancy and maintaining data integrity, the star schema emphasizes simplifying data for analytical purposes.

As we worked on this exercise, we grew to appreciate the nuances of both schema designs. With its complex web of interconnected tables, the normalized schema taught us the importance of data integrity and consistency. It demonstrated how to eliminate data anomalies and ensure that updates and modifications to the database would not result in inconsistencies.

However, the limitations of the normalized schema became evident when it came to analyzing and reporting on the data. Queries necessitated intricate JOIN operations across multiple tables, making them simple and efficient for business intelligence purposes. These are where the dimensional model schema shone. It introduced the concepts of facts and dimensions, with facts representing quantitative data like tuition fees, while dimensions simplified the schema.

Creating a dimensional model schema was truly eye-opening. We observed how this approach streamlined reporting and data analysis. Queries that were once convoluted with JOINS became straightforward, making extracting meaningful insights from the data more accessible. We realized that dimensional modeling was particularly well-suited for data warehousing and business intelligence scenarios, where quick access to summarized data was crucial.

In conclusion, our experience with Lab Exercise #2, using both dimensional and normalized schema diagrams, has been invaluable. While the normalized schema emphasized data integrity and consistency, the dimensional model schema showcased the significance of data accessibility and analysis. As future data science professionals, understanding when to use each schema approach is a valuable skill we will carry forward in our academic and professional journey.

3.1 Difference of Lab Exercise 1 and Lab Exercise 2

The querying approaches in Lab Exercise #1 and Lab Exercise #2 diverge significantly, reflecting the unique characteristics of each exercise's objectives and database schemas. In Lab Exercise #1, the primary focus was on SQL querying within a normalized schema. Here, SQL queries were employed to retrieve, filter, and manipulate data from a database structured into multiple related tables. While this normalization strategy was geared towards maintaining data integrity by eliminating redundancy and anomalies, it often entailed intricate JOIN operations to access information across various tables.

The Lab Exercise #2, on the other hand, centered on dimensional modeling. It entailed designing a star schema tailored for business intelligence and analytical reporting purposes. This schema simplified data organization by introducing a central fact table (e.g., student enrollment) surrounded by dimension tables (e.g., courses, students). This simplification led to more efficient and intuitive queries, as the need for complex JOINS was reduced. The introduction of fact and dimension tables facilitated data accessibility and analysis, making Lab Exercise #2 particularly suitable for data warehousing and reporting scenarios where quick access to summarized data is paramount. Thus, while Lab Exercise #1 underscored the significance of data integrity and complex

JOIN operations within a normalized schema, Lab Exercise #2 highlighted the importance of data accessibility, analysis, and efficient querying within a dimensional model schema, emphasizing their relevance in the broader field of data warehousing and business intelligence.