Summer
2021

INSY661 – Database and Distributed Systems for Analytics: Group Project Report

TEAM 16:

ARUNIMA AGRAWAL (261001915)
ATRIN MORTEZA GHASEMI (261005342)
JIAHUA LIANG (260711529)
KEXIN WANG (260787422)
MOHAMAD KHALILI (260746712)
YINGXIN JIANG (261007353)

DESAUTELS
Faculty of Management
Faculté de gestion

Presented to : Prof. Animesh Animesh

September 14th, 2021

Table of Contents

1. Overview of the business scenario

- Online teaching platform, similar to Udemy and Coursera
- Goals to be achieved with the database system: have an optimized database, allowing to easily query it to provide interesting insights from an analytics perspective
- The main users will be the students who enroll for an online course, as well as the instructors, who can create and upload online courses on the platform.
- Students can enroll for one or multiple courses. They can also refer other students to a class (ex: Student 1 can share a SQL class with Student 2, and Student 2 can choose to enroll or not in the class).
- Instructors can teach one or multiple courses, and one course can be taught by one or multiple instructors.
- The courses are classified using a category system. Each course can be under one and only one category.
- Each course contains videos, presenting the taught subject, as well as quizzes to assess student performance in the course. One course can contain multiple videos and quizzes.
- A credit system based on gift cards applicable into student accounts is also going to be implemented within the database.

2. Mission Statement & Objective

Mission Statement:

The purpose of this database is to maintain and store data about the contents and users of the online learning platform. The data stored and generated will be used to build a rich and interactive platform, where instructors can register and upload courses, students can register and enroll in these courses, and recommend these courses to each other, thus building a rich and interactive learning platform.

Objectives:

Update, enter and delete data on:

- Students
- Instructors
- Courses
- Categories

Perform searches on:

- Course popularity
- Course availability
- Course & Instructor rating
- Course enrollment

Track the following:

- Course completion
- Course duration
- Course success / fail rate
- Course sharing between students

3. ERD

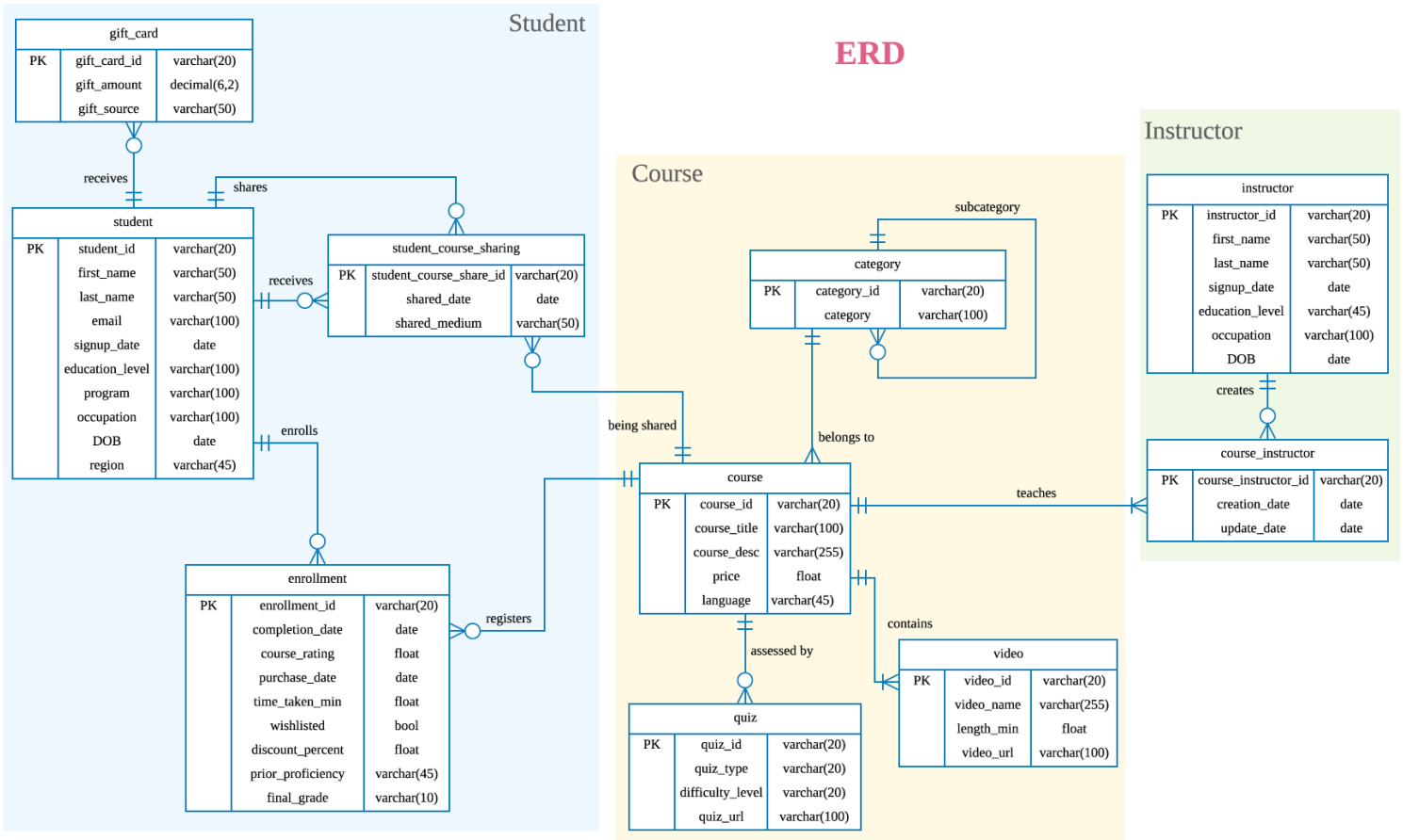


Figure 1 – Entity-Relationship Diagram

4. Data Dictionaries

4.1. Description of Entities

Table 1 – Description of Entities

ENTITY NAME	DESCRIPTION	ALIASES	OCCURRENCE
COURSE	Entity that represents the online courses that are uploaded on the platform	course	One course can contain up to multiple quizzes and videos, can be taught by multiple instructors and belongs to only one category.
STUDENT	Entity that represents each student that registers on the platform	student	One student can register for up to multiple classes, can share up to multiple classes and can buy more than one gift card
CATEGORY	Entity that contains a descriptive of each category	category	One category can have multiple subcategories. Each course is unique under a category. For instance, the SQL class is only under the Analytics category, and nowhere else
INSTRUCTOR	Entity that represents the instructor for each course	instructor	One instructor can teach multiple courses, and one course can be taught by multiple instructors.
VIDEO	Entity that represents the video linked to the uploaded course	video	One course can contain multiple videos
QUIZ	Entity that represents the quiz linked to the uploaded course	quiz	One course can contain multiple quizzes
GIFT CARD	Entity that represents gift cards applicable to a student's account balance	gift_card	One student can buy multiple gift cards
SHARING	Entity that contains the information relative to course sharing among students	student_course_sharing	One student can share multiple courses with many other students
ENROLLMENT	Entity that contains the enrollment information of each student for each course	enrollment	One student can enroll for multiple courses
COURSE INSTRUCTOR	Entity that links the instructor to the course	course_instructor	One instructor can teach multiple courses, and one course can be taught by multiple instructors.

4.2. Description of Attributes

Note: To make the table more concise, there are no Nulls, no multivalued attributes, no derived attributes as well as no default values set for any attribute.

Table 2 – Description of Attributes

ENTITY NAME	ATTRIBUTES	DESCRIPTION	DATA TYPE
COURSE	course_id	Course Identifier	varchar(20)
	course_title	Course Title	varchar(100)
	course_desc	Course Description	varchar(255)
	price	Price of the Course	float
	language	Language of the Course	varchar(45)
STUDENT	student_id	Student Identifier	varchar(20)
	first_name	Student's first name	varchar(50)
	last_name	Student's last name	varchar(50)
	email	Student's email	varchar(100)
	signup_date	Student's signup date	date
	education_level	Highest education level	varchar(100)
	program	Field of studies	varchar(100)
	occupation	Job	varchar(100)
	DOB	Date of birth	date
	region	Region of the world	varchar(45)
CATEGORY	category_id	Category Identifier	varchar(20)
	category	Name of the Category	varchar(100)
INSTRUCTOR	instructor_id	Instructor Identifier	varchar(20)
	first_name	Instructor's first name	varchar(50)
	last_name	Instructor's last name	varchar(50)
	email	Instructor's email	varchar(100)
	signup_date	Instructor's signup date	date
	education_level	Instructor's education	varchar(45)
	occupation	Instructor's job	varchar(100)
	DOB	Instructor's date of birth	date

ENTITY NAME	ATTRIBUTES	DESCRIPTION	DATA TYPE
VIDEO	video_id	Video Identifier	varchar(20)
	video_name	Video's title	varchar(255)
	length_min	Video's length	float
	video_url	URL of the video	varchar(100)
QUIZ	quiz_id	Quiz identifier	varchar(20)
	quiz_type	Quiz Type	varchar(20)
	difficulty_level	Difficulty Level	varchar(20)
	quiz_url	URL of the quiz	varchar(100)
GIFT_CARD	gift_card_id	Identifier for the Gift Card	varchar(20)
	gift_amount	Amount of the gift	decimal(6,2)
	gift_source	Source of the gift	varchar(50)
STUDENT_COURSE_SHARING	student_course_share_id	Identifier of the share	varchar(20)
	shared_date	Date the course has been shared	date
	shared_medium	Sharing medium	varchar(50)
ENROLLMENT	enrollment_id	Enrollment Identifier	varchar(20)
	completion_date	Date the course has been completed	date
	course_rating	Rating	float
	purchase_date	Date the course was purchased	date
	time_taken_min	Time taken to complete the course	float
	wishlisted	Was the course wishlisted?	bool
	discount_percent	Discount percentage	float
	prior_proficiency	Prior proficiency in said course	varchar(45)
	final_grade	Obtained Final Grade	varchar(10)
COURSE_INSTRUCTOR	course_instructor_id	Identifier linking course and instructor	varchar(20)
	creation_date	Date a course has been created	date
	update_date	Date a course has been updated	date

5. Relational/Logical Model

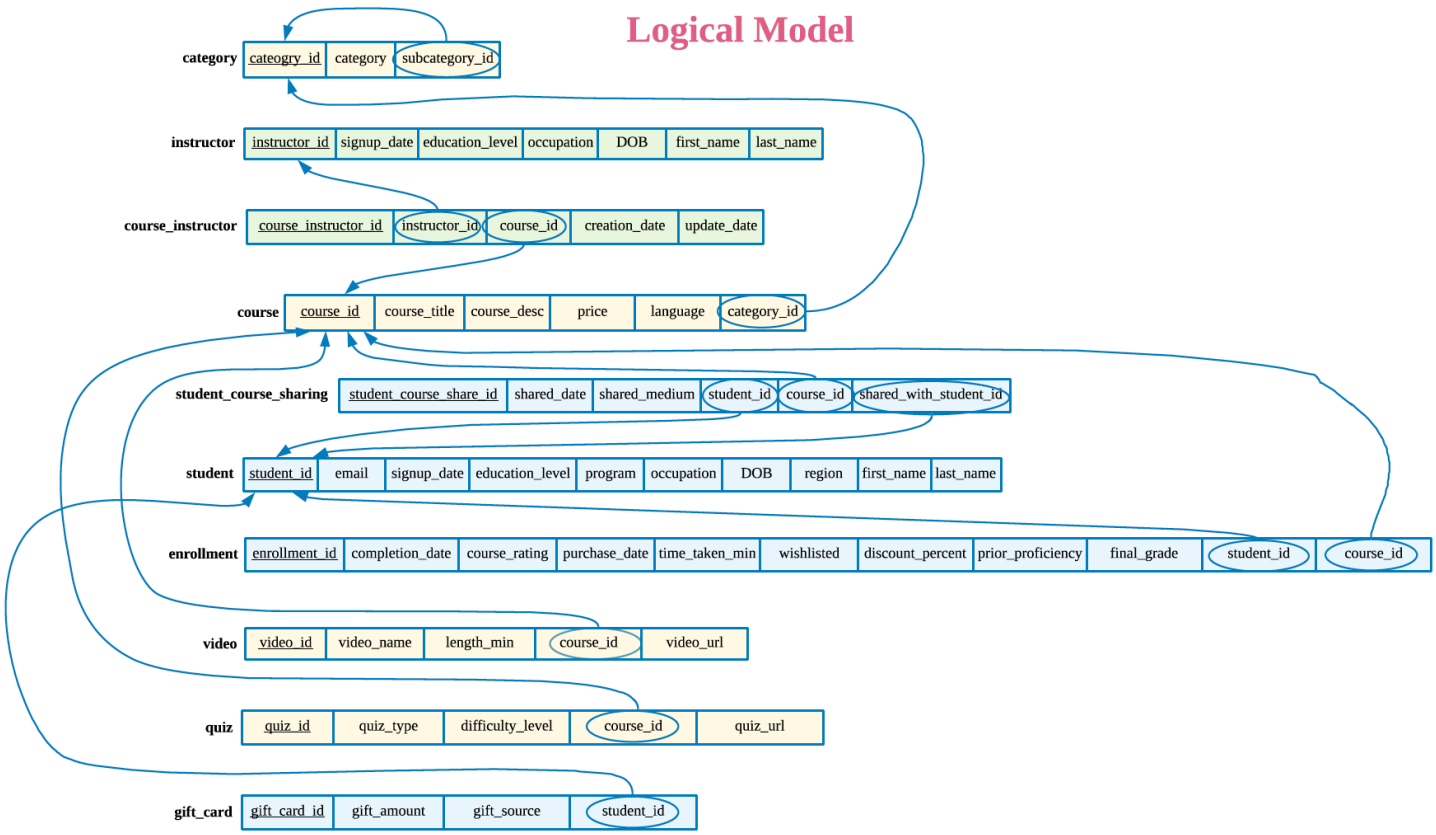


Figure 2 – Logical Model

6. Relational Database Queries (DDL, DML)

6.1. Creating the tables and inserting data

To make this document easier to read, the DDL is not presented here. Rather, it is available in the DDL.sql file joined with this document. Please refer to this file if you wish to see the queries that allowed us to build our tables, define primary and foreign keys, and insert data.

6.2. SQL Queries

Query 1:

Obtain the total credits available for each student. The total credits available can be calculated by: referral points + gift card balance

The company wants to analyze the total credits available for each student. Credits can be used by the students to purchase new courses. These credits include the following:

- Gift cards received from the company as part of their attractive benefits program.
- Gift cards received from credit card companies like Visa and Mastercard.
- Gift cards received from friends.
- Referral points received when your friend redeems a course with your referral code. Each referral can earn you $10.

```sql
CREATE VIEW a1 AS
SELECT scs.student_id, COUNT(*)*10 AS Credits
FROM student_course_sharing scs
    INNER JOIN enrollment AS e
    ON scs.shared_with_student_id = e.student_id
WHERE scs.course_id = e.course_id
GROUP BY scs.student_id;

CREATE VIEW a2 AS
SELECT student_id, gift_amount AS Credits
FROM gift_card
GROUP BY student_id;

SELECT s.student_id, s.first_name, s.last_name, CONCAT("$ ", SUM(Credits)) AS total_credits
FROM student s INNER JOIN
    (
    SELECT *
    FROM a1
    UNION ALL
    SELECT *
    FROM a2
    ) a3
    ON a3.student_id = s.student_id
GROUP BY s.student_id
ORDER BY SUM(Credits) DESC;
```

student_id	first_name	last_name	total_credits
76	Carmon	Collington	$ 210.00
163	Loise	Lowre	$ 200.00
72	Claudetta	Capner	$ 197.00
103	Erin	Folliott	$ 194.00
75	Harlen	Vallender	$ 193.00
107	Roselle	Longfellow	$ 192.00
178	Amelia	Cozins	$ 190.00

Query 2:

The company wants to re-evaluate its pricing strategy and wants to analyze the relationship between the student's age and the price of the course he/she enrolls in.
Insights: The students aged below 30 years tend to spend the most on purchasing courses. A probable explanation for this observation could be that the young students are mindful about investing more in learning to give a head-start to their careers.

```
32  SELECT
33  CASE WHEN Student_Age>=0 AND Student_Age<20 THEN "< 20"
34  WHEN Student_Age>=20 AND Student_Age<30 THEN "20-29"
35  WHEN Student_Age>=30 AND Student_Age<40 THEN "30-39"
36  WHEN Student_Age>=40 THEN ">= 40"
37  END AS Student_Age_Group,
38  COUNT(*) AS Number_of_Students,
39  CONCAT("$ ",ROUND(AVG(Average_Price),2)) AS Average_Price
40  FROM
41  (
42      SELECT ROUND(DATEDIFF(date(now()), s.DOB)/365,0) AS Student_Age, Average_Price
43      FROM student s INNER JOIN
44          (
45          SELECT e.student_id, ROUND(AVG(c.price),2) AS Average_Price
46          FROM enrollment e INNER JOIN course c
47              ON e.course_id = c.course_id
48          GROUP BY student_id
49          ) AS ec
50          ON s.student_id = ec.student_id
51  ) AS a4
52  GROUP BY Student_Age_Group
53  ORDER BY ROUND(AVG(Average_Price),2) DESC;
```

Student_Age_Group	Number_of_Students	Average_Price
< 20	22	$ 94.25
20-29	65	$ 91.64
30-39	43	$ 88.64
>= 40	70	$ 85.54

Query 3:

The company wants to analyze if the length of the video affects the number of students who enroll in the course.

Insights: The students are mostly inclined towards enrolling in courses which are 3 to 14 hours long. They do not prefer very quick/high-level courses which are less than 3 hours long, and they also do not prefer extremely long/detailed courses. This can help the instructors understand the preferred duration of the courses.

```
57  SELECT
58  CASE WHEN a5.Video_Length_Per_Course>=0 AND a5.Video_Length_Per_Course<3 THEN "< 3"
59  WHEN a5.Video_Length_Per_Course>=3 AND a5.Video_Length_Per_Course<9 THEN "3-8"
60  WHEN a5.Video_Length_Per_Course>=9 AND a5.Video_Length_Per_Course<15 THEN "9-14"
61  WHEN a5.Video_Length_Per_Course>=15 AND a5.Video_Length_Per_Course<21 THEN "15-20"
62  WHEN a5.Video_Length_Per_Course>=21 THEN ">= 21"
63  END AS Video_Length_Interval_Hrs,
64  SUM(a6.Number_of_Enrollments_Per_Course) AS Number_of_Enrollments
65  FROM
66      (
67          SELECT c.course_id, ROUND(SUM(length_min)/60,0) AS Video_Length_Per_Course
68          FROM course c INNER JOIN video v
69              ON c.course_id = v.course_id
70          GROUP BY c.course_id
71      ) AS a5
72      INNER JOIN
73      (
74          SELECT c.course_id, COUNT(*) AS Number_of_Enrollments_Per_Course
75          FROM course c INNER JOIN enrollment e
76              ON c.course_id = e.course_id
77          GROUP BY c.course_id
78      ) AS a6
79      ON a5.course_id = a6.course_id
80  GROUP BY Video_Length_Interval_Hrs
81  ORDER BY Number_of_Enrollments DESC;
```

Video_Length_Interval_Hrs	Number_of_Enrollments
3-8	426
9-14	424
15-20	75
< 3	67
>= 21	8

Query 4:

Obtain the percentage of courses passed by students in each difficulty level

Assumptions:

- A course is passed by a student if he/she obtains a grade higher than 50% in that course.
- The difficulty level of a course is determined based on the average difficulty level of all its quizzes.
- To calculate average difficulty of quizzes, we assign numbers to difficulties and calculate the average of that number. Based on the average, we map the number to one of the difficulty levels and consider the course having that difficulty level.

Easy = 1, Medium = 2, Hard = 3, Very Hard = 4

```
CREATE VIEW Course_difficulty_number AS

WITH CTE_Quiz_difficulty_level(course_id, quiz_id, Difficulty_score) AS (
  SELECT course_id, quiz_id, CASE
  WHEN difficulty_level = 'Easy' THEN 1
  WHEN difficulty_level = 'Medium' THEN 2
  WHEN difficulty_level = 'Hard' THEN 3
  WHEN difficulty_level = 'Very_Hard' THEN 4
  END As Difficulty_score
  FROM quiz
  ORDER BY course_id
    )

SELECT course_id, FORMAT(AVG(Difficulty_score),2) AS Course_Difficulty_Number
FROM CTE_Quiz_difficulty_level
GROUP BY course_id
ORDER BY course_id ASC ;

CREATE VIEW Course_Difficulties_AND_Pass_FAIL AS (
  WITH CTE_Course_Difficulty (course_id, Course_Difficulty) AS (
    SELECT course_id, CASE
    WHEN Course_Difficulty_Number >= 3.5 THEN "Very_Hard"
    WHEN Course_Difficulty_Number < 3.5 AND Course_Difficulty_Number >= 2.5 THEN "Hard"
    WHEN Course_Difficulty_Number < 2.5 AND Course_Difficulty_Number >= 1.5 THEN "Medium"
    WHEN Course_Difficulty_Number < 1.5 THEN "Easy"
    END AS Course_Difficulty
    FROM Course_difficulty_number
    )

SELECT COUNT(*) AS Number, Course_Difficulty, CASE
  WHEN final_grade >= 50 THEN "Pass"
  WHEN final_grade <50 THEN "Fail"
  END AS Pass_Fail
FROM CTE_Course_Difficulty INNER JOIN course USING(course_id) INNER JOIN enrollment USING(course_id)
GROUP BY Course_Difficulty, Pass_Fail
ORDER BY COurse_Difficulty DESC ;
```

```
SELECT FORMAT((Temp1.Number/Temp2.Total)*100, 2) AS Passing_Rate_Percent, Temp1.Course_Difficulty
FROM Course_Difficulties_AND_Pass_FAIL AS Temp1 LEFT JOIN (
  SELECT SUM(Number) AS Total ,Course_Difficulty
  FROM Course_Difficulties_AND_Pass_FAIL
  GROUP BY Course_Difficulty) AS Temp2
  ON Temp1.Course_Difficulty = Temp2.Course_Difficulty

WHERE Pass_Fail = "Pass"
ORDER BY Passing_Rate_Percent DESC;
```

Passing_Rate_Percent	Course_Difficulty
74.39	Easy
72.60	Very_Hard
68.59	Hard
67.72	Medium

Query 5:

Identify the customers who need urgent "attention" from the company. Marketing/Sales staff may need to implement retaining measures to get in touch with these customers whether by a coupon specifically designed for them or a friendly reminder e-mail that shows the company values them.

To identify these customers, we need to segment the company's customers.

Assumptions:

- It is assumed customers who have purchased more courses from the company than average are "Loyal" customers. Others are labeled as "Normal" customers.
- It is assumed customers who have a greater number of days_since_last_purchase than average are "IDLE" customers. Others are labeled as "Active" customers.
- It is assumed customers who have paid the company more than the average are "Valuable" customers. Others are labeled as "Typical" customers.
- Based on previous segments, "Loyal" customers who are "Valuable" but are "IDLE" need urgent attention from the company.
- It is assumed that the current date of this analysis is the day after the last purchased ticket we produced in the fake data. In other words, days_since_last_purchase is based on November 9th, 2020.

```sql
-- Attention-needing Customers! An RFM Model (Recency-Frequency-Monetary) for Customer Segmentation
-- Creating a table with all the information we need to segment the customers

CREATE VIEW All_Needed_Data AS (
  SELECT student_id, DATEDIFF("2020-11-09", MAX(purchase_date)) AS Days_Since_Last_Purchase, COUNT(*) AS Purchase_Frequency, SUM(Price) AS Total_Monetary_Value
  FROM enrollment INNER JOIN course USING(course_id)
  GROUP BY student_id
);

-- Labeling customers using CASE statements. The details of labeling are explained in query assumptions.

CREATE VIEW Labeled_Customers AS (
  SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,
  CASE
  WHEN Purchase_Frequency > (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN "Loyal"
  WHEN Purchase_Frequency < (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN "Normal"
  END AS Customer_Frequency_Type,
  CASE
  WHEN Days_Since_Last_Purchase > (SELECT AVG(Days_Since_Last_Purchase) FROM All_Needed_Data) THEN "IDLE"
  WHEN Days_Since_Last_Purchase < (SELECT AVG(Days_Since_Last_Purchase) FROM All_Needed_Data) THEN "ACTIVE"
  END AS Customer_Status,
  CASE
  WHEN Total_Monetary_Value > (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN "Valuable"
  WHEN Total_Monetary_Value < (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN "Typical"
  END AS Customer_Value_Type
  FROM All_Needed_Data );

-- Finally, we identify the customers who need attention by filtering on those who are 'Loyal', 'Valuable', and 'IDLE'

SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,  Customer_Frequency_Type, Customer_Value_Type, Customer_Status
FROM Labeled_Customers
WHERE Customer_Frequency_Type = 'Loyal' AND Customer_Value_Type = 'Valuable' AND Customer_Status = 'IDLE'
ORDER BY Days_Since_Last_Purchase DESC
```

student_id	Days_Since_Last_Purchase	Purchase_Frequency	Total_Monetary_Value	Customer_Frequency_Type	Customer_Status	Customer_Value_Type
99	535	6	577	Loyal	IDLE	Valuable
41	468	6	524	Loyal	IDLE	Valuable
122	383	6	461	Loyal	IDLE	Valuable
46	373	7	537	Loyal	IDLE	Valuable
50	371	6	602	Loyal	IDLE	Valuable
6	363	7	672	Loyal	IDLE	Valuable
27	357	7	580	Loyal	IDLE	Valuable
55	343	9	806	Loyal	IDLE	Valuable
197	337	6	650	Loyal	IDLE	Valuable

Query 6:

Find the most recommended course by students, namely the course with the most shares, in each primary category. Primary category means the categories at the top of hierarchy and are not subcategory of any other categories. All courses in a category with the same maximum number of shares will be displayed.

```
-- create a table with the total number of shares for each course
CREATE VIEW course_num_shares AS
SELECT student_course_sharing.course_id, COUNT(student_course_sharing.course_id) AS num_shares
FROM student_course_sharing
GROUP BY student_course_sharing.course_id
ORDER BY num_shares DESC;

-- create a table with the maximum number of shares for each category
CREATE VIEW cat_max_shares AS
SELECT category.category_id, MAX(course_num_shares.num_shares) AS max_shares
FROM category INNER JOIN course ON course.category_id = category.category_id
INNER JOIN
course_num_shares ON course.course_id = course_num_shares.course_id
GROUP BY category.category_id;

-- inner join the two views above with the course and category tables to find the course(s) in each
-- category with the maximum number of shares
SELECT category.category, course.course_title, cat_max_shares.max_shares
FROM cat_max_shares INNER JOIN course ON course.category_id = cat_max_shares.category_id
INNER JOIN course_num_shares ON course_num_shares.course_id = course.course_id
INNER JOIN category ON cat_max_shares.category_id = category.category_id
WHERE course_num_shares.num_shares = cat_max_shares.max_shares
AND cat_max_shares.category_id NOT IN (SELECT category.subcategory_id FROM category);
```

category	course_title	max_shares
RTI	Python	9
WWII	HD Video	6
LNG	Numerical Analysis	5
Ebay Sales	Observational Studies	4
Sales Management	Atmel AVR	4
Sales Management	OHSAS 18001	4
GDP	nCode	3
MHE	Ministry Of Defence	3
IV	JSONP	3
UCS	ERD	2
Hardware Archite…	Auditing	2
Hardware Archite…	VSEO	2
UEFI	Cognitive Psychology	1

Query 7:

Find the most popular primary category of courses, namely the category with maximum number of enrollments, for students in each region. Primary category means the categories at the top of hierarchy and are not subcategory of any other categories. All categories in a region with the same maximum number of enrollments will be displayed.

```
-- create a table containing info about the number of enrollments in each category in each region
CREATE VIEW region_cat_qty AS
SELECT student.region, category.category_id, COUNT(category.category_id) AS num_enroll
FROM student INNER JOIN enrollment ON student.student_id = enrollment.student_id
INNER JOIN course ON enrollment.course_id = course.course_id
INNER JOIN category ON course.category_id = category.category_id
WHERE category.category_id NOT IN (SELECT category.subcategory_id FROM category)
GROUP BY student.region , category.category_id
ORDER BY student.region ASC , quantity DESC;

-- get the category with max number of enrollments (most popular) in each region
SELECT region_cat_qty.region, category.category AS MostPopularCategory, max_enroll AS NumOfEnroll
FROM
(SELECT region_cat_qty.region, MAX(quantity) AS max_enroll
    FROM region_cat_qty
    GROUP BY region_cat_qty.region) AS subquery
INNER JOIN region_cat_qty ON subquery.region = region_cat_qty.region
        AND subquery.max_enroll = region_cat_qty.quantity
INNER JOIN category ON category.category_id = region_cat_qty.category_id
ORDER BY max_enroll DESC;
```

region	MostPopularCategory	NumOfEnroll
China	Sales Management	12
Indonesia	WWII	8
Russia	UEFI	6
France	Sales Management	5
United States	Ebay Sales	4
Portugal	Sales Management	4
Poland	UCS	4
Nigeria	Sales Management	4
Brazil	WWII	4
Latvia	RTI	3
Afghanistan	UEFI	3
Ukraine	UCS	3
Dominican	WWII	3

Query 8:

Of all the students who followed the Financial Advisory Course (Course ID 22), does the final grade correlate with education level?

```
SELECT education_level AS EducationLevel, FORMAT(avg(enrollment.final_grade),2) AS AverageFinalGrade
FROM student, enrollment, (SELECT  enrollment.student_id, enrollment.final_grade
                           FROM enrollment
                           WHERE course_id = 22
                           ) AS t1
WHERE student.student_id = enrollment.student_id
GROUP BY education_level;
```

EducationLevel	AverageFinalGrade
High-school	59.35
PhD	59.31
Master	59.33
Bachelor	60.78
Elementary-school	55.29

Query 9:

Find the average rating of courses in each primary category offered by instructors in each education level. This is aimed at studying the correlation between the education level of the instructors and the quality of their courses in different primary categories. Primary category means the categories at the top of hierarchy and are not subcategory of any other categories.

```
SELECT category.category, instructor.education_level AS instructor_edu_level,
        FORMAT(AVG(enrollment.course_rating),2) AS avg_course_rating
FROM enrollment INNER JOIN course ON enrollment.course_id = course.course_id
INNER JOIN course_instructor ON course.course_id = course_instructor.course_id
INNER JOIN instructor ON instructor.instructor_id = course_instructor.instructor_id
INNER JOIN category ON category.category_id = course.category_id
WHERE course.category_id NOT IN (SELECT category.subcategory_id FROM category)
GROUP BY course.category_id, instructor.education_level
ORDER BY course.category_id;
```

category	instructor_edu_level	avg_course_rating
MHE	Bachelor	2.90
MHE	Master	2.90
MHE	PhD	2.94
GDP	Master	3.51
GDP	PhD	3.39
Ebay Sales	Bachelor	3.18
Ebay Sales	PhD	3.15
LNG	Bachelor	2.92
LNG	Master	2.97
LNG	PhD	2.93
WWII	Bachelor	3.05
WWII	Master	2.96
WWII	PhD	2.86
IV	Bachelor	3.34

Query 10:

Of all the students who were recommended a class, how many actually enrolled for the class?

```
CREATE VIEW CourseSharing AS(
SELECT student_id, shared_with_student_id, course_id
FROM student_course_sharing
);

SELECT COUNT(DISTINCT enrollment_id) AS TotalEnrollments,
       COUNT(DISTINCT CourseSharing.student_id) AS EnrollmentsAfterRecommendation
FROM enrollment, CourseSharing
WHERE CourseSharing.shared_with_student_id = enrollment.student_id
```

	TotalEnrollments	EnrollmentsAfterRecommendati...
▶	650	126

Query 11:

Find the top 1 bestseller course(s) (the paid course(s) with the most enrollments) in each category. The top 1 bestseller is based on ranking of total number enrollments. If there are multiple courses in top 1 rank with the same number of total enrollments, they will all be displayed.

```
-- Create a table containing the information about the total number of enrollments of each
-- course in each category
CREATE VIEW course_totalEnroll AS
SELECT category, course.course_title, t1.TotalEnrollments
FROM course, category,
(SELECT course_id, COUNT(enrollment.enrollment_id) AS TotalEnrollments
FROM enrollment
GROUP BY course_id
ORDER BY TotalEnrollments DESC) AS t1
WHERE t1.course_id = course.course_id
AND course.category_id = category.category_id
AND course.price != 0
ORDER BY category, TotalEnrollments DESC;

-- Select the course with the highest total number of enrollments (bestseller) in each category
WITH category_enroll_rank AS
(SELECT category, course_title AS BestSeller, TotalEnrollments,
ROW_NUMBER() OVER(PARTITION BY course_totalEnroll.category ORDER BY TotalEnrollments DESC) `rank`
FROM course_totalEnroll)
SELECT category, BestSeller, TotalEnrollments
FROM category_enroll_rank
WHERE `rank` = 1;
```

category	BestSeller	TotalEnrollments
Academic Tutoring	RRDTool	7
Analytical Skills	Fume FX	7
BDC programming	Risk Assessment	15
BSF	Fire Alarm	13
Candidate Generation	Image Processing	10
Ebay Sales	CFM	15
GDP	nCode	14
Hardware Architecture	Auditing	12
HD Camera Operation	GMC PrintNet T	13
Hydraulic Systems	WSS 2.0	13
IDS	Zero Waste	12
IV	JSONP	16
Know-how	DMB	16

Query 12:

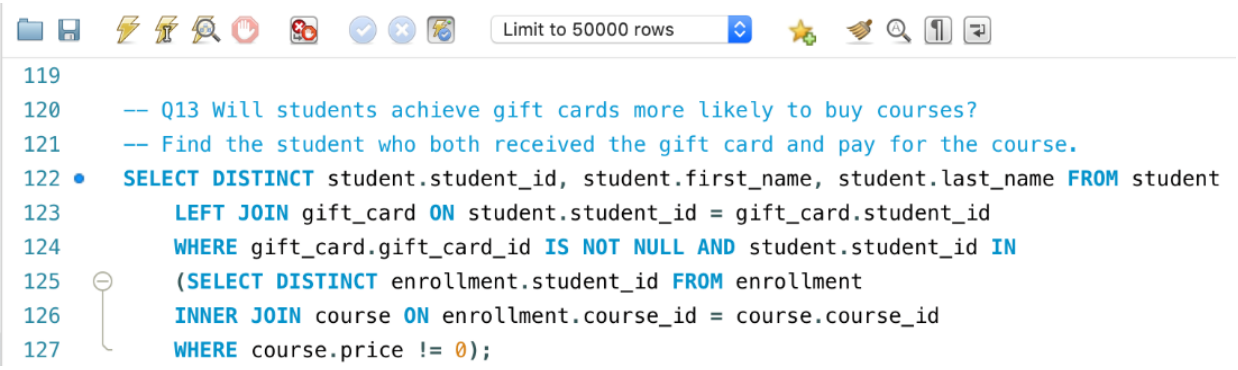
Find the 5 most popular courses among students with a Bachelor's degree and the 5 most popular courses among students with a Master's degree. If multiple courses have the same total number of enrollments, the system will randomly select the ones to be displayed under the limit of 5 courses.

```
(SELECT course_title AS CourseName, COUNT(enrollment_id) AS TotalEnrollment,
"Bachelor" AS Degree
FROM enrollment, course, student
WHERE enrollment.course_id = course.course_id
AND enrollment.student_id = student.student_id
AND education_level = "Bachelor"
GROUP BY course_title
ORDER BY TotalEnrollment DESC
LIMIT 5)
UNION
(SELECT course_title AS CourseName, COUNT(enrollment_id) AS TotalEnrollment,
"Master" AS Degree
FROM enrollment, course, student
WHERE enrollment.course_id = course.course_id
AND enrollment.student_id = student.student_id
AND education_level = "Master"
GROUP BY course_title
ORDER BY TotalEnrollment DESC
LIMIT 5);
```

CourseName	TotalEnrollme...	Degree
SMO	10	Bachelor
nCode	9	Bachelor
Training	9	Bachelor
Equities	8	Bachelor
Crisis Communications	8	Bachelor
WSS 2.0	6	Master
GBA	6	Master
DMR	6	Master
HD Video	5	Master
Atmel AVR	5	Master

Query 13:

We are curious whether students who achieve gift cards are more likely to buy courses. Find the students who both received gift cards and pay for the non-free courses. Show the id, first name, and last name of each of those students.




```
119
120     -- Q13 Will students achieve gift cards more likely to buy courses?
121     -- Find the student who both received the gift card and pay for the course.
122 •   SELECT DISTINCT student.student_id, student.first_name, student.last_name FROM student
123         LEFT JOIN gift_card ON student.student_id = gift_card.student_id
124         WHERE gift_card.gift_card_id IS NOT NULL AND student.student_id IN
125         (SELECT DISTINCT enrollment.student_id FROM enrollment
126         INNER JOIN course ON enrollment.course_id = course.course_id
127         WHERE course.price != 0);
```

student_id	first_name	last_name
1	Dolph	Townson
100	Doris	Kenshole
103	Erin	Folliott
104	Cristal	Mundwell
106	Dedie	Tunmore
107	Roselle	Longfellow
108	Aguie	Mainson
109	Elia	Danson
110	Sig	Delahunty
111	Jorey	Batchellor
113	Pierson	Ferrolli
115	Vicki	McKeand
117	Lancelot	Du Pre
118	Giselbert	Lowdham
119	Viviene	Wane
121	Leese	Woollam
122	Ralf	Pritty
123	Maximilien	Jeandin
124	Ricki	Trathan

Query 14:

Let's discover the free courses provided and how students feel about the free courses. Find the proportion of the free course, as well as the proportion of students enrolling in a free course. Assumptions: proportion of the free course = number of free courses/ total number of courses proportion of students enrolling in a free course = number of students enroll in free courses/ total number of students.

```
145    -- Q14 final answer:
146 •  SELECT (SELECT COUNT(course.course_id) FROM course WHERE course.price = 0)/COUNT(course.course_id) AS free_course_proportion,
147        (SELECT (SELECT COUNT(DISTINCT enrollment.student_id) FROM enrollment
148        INNER JOIN course ON enrollment.course_id = course.course_id
149        WHERE course.price = 0)/COUNT(student.student_id) FROM student) AS free_course_participation
150        FROM course;
```

free_course_proportion	free_course_participation
0.0300	0.1250

Query 15:

For each applicable course, determine whether students with advanced prior proficiency have higher average final grades than those at the beginner level.
Assumption: Applicable courses are those that have both students with advanced prior proficiency and students with beginner prior proficiency. Only those courses will be compared in this question.

```
Limit to 50000 rows
152   -- Q15
153 • SELECT B.course_id, FORMAT(advanced_average,2) AS advanced_average, FORMAT(beginner_average,2) AS beginner_average,
154   CASE WHEN advanced_average > beginner_average THEN 'True' ELSE 'False' END AS advanced_better
155   FROM
156   (SELECT enrollment.course_id, AVG(final_grade) AS advanced_average
157       FROM enrollment
158       WHERE prior_proficiency = 'Advanced'
159       GROUP BY enrollment.course_id) AS A INNER JOIN
160   (SELECT enrollment.course_id, AVG(final_grade) AS beginner_average
161       FROM enrollment
162       WHERE prior_proficiency = 'Beginner'
163       GROUP BY enrollment.course_id) AS B ON A.course_id = B.course_id
164       ORDER BY B.course_id;
```

course_id	advanced_average	beginner_average	advanced_better
1	58.10	69.41	False
10	89.82	60.60	True
100	34.20	47.57	False
11	63.65	49.87	True
12	57.15	54.75	True
13	60.40	52.82	True
14	77.30	58.52	True
15	33.12	61.27	False
16	39.60	48.89	False
17	50.70	34.90	True
18	51.18	60.20	False
19	63.30	43.95	True
2	65.23	56.97	True
20	55.00	56.22	False
21	58.13	49.40	True
22	61.23	53.82	True
23	70.32	48.00	True
24	57.03	41.03	True
25	68.92	70.70	False

Query 16:

What's the average period between purchase date and complete date based on course? We want to find the complete cycle of the course provided, so to further analyze the time cost on each course.

```
2 •   SELECT course_id, AVG(complete_period) AS Average_period
3     FROM
4     (SELECT student_id, course_id, DATEDIFF(completion_date,purchase_date) AS complete_period
5     FROM enrollment) AS CP
6     GROUP BY course_id
7     ORDER BY Average_period DESC;
8
```

00% 30:7

Result Grid Filter Rows: Search Export:

course_id	Average_period
2	444.0000
95	406.2222
72	404.9231
52	402.0000
28	401.6667
97	398.3333
71	392.1250
36	374.8889
82	369.2000
30	352.0833
74	349.0000
63	348.0769
20	343.0909
77	337.6154
4	336.7500
85	336.3750

Query 17:

Find the quantity of each quiz type for each difficulty level. To evaluate the task intensity of quiz's difficulty level.

```
12 •  SELECT difficulty_level, quiz_type,COUNT(quiz_type) AS quantity,
13    SUM(COUNT(quiz_type)) OVER (PARTITION BY difficulty_level) AS Total
14    FROM quiz
15    GROUP BY difficulty_level, quiz_type
16    ORDER BY difficulty_level;
17
18    # 6)Find the instructors who teach courses with the highest diversity
```

100% 27:16

Result Grid Filter Rows: Search Export:

difficulty_level	quiz_type	quantity	Total
Easy	Multiple-choice	32	138
Easy	Short-answer	37	138
Easy	Multiple-answer	37	138
Easy	Long-answer	32	138
Hard	Multiple-choice	32	121
Hard	Short-answer	37	121
Hard	Multiple-answer	24	121
Hard	Long-answer	28	121
Medium	Short-answer	24	121
Medium	Long-answer	22	121
Medium	Multiple-choice	37	121
Medium	Multiple-answer	38	121
Very_Hard	Multiple-answer	26	120
Very_Hard	Long-answer	34	120
Very_Hard	Short-answer	30	120
Very_Hard	Multiple-choice	30	120

Query 18:

Observe the diversity of audiences and course categories for each instructor.

Assumptions:

- Diversity of audiences refers to the total number of different regions for all students in all courses taught by an instructor.
- Diversity of course categories refer to the total number of different categories for all courses taught by an instructor.
- Category here refers to the main category, not the subcategory.

```
20  SELECT T1.instructor_id, T1.first_name, T1.last_name, category_diversity, region_diversity
21  FROM
22
23  (SELECT course_instructor.instructor_id, instructor.first_name, instructor.last_name,
24  COUNT(DISTINCT category.category_id) AS category_diversity
25  FROM category LEFT JOIN course ON category.category_id = course.category_id
26  INNER JOIN course_instructor ON course.course_id = course_instructor.course_id
27  INNER JOIN instructor ON course_instructor.instructor_id = instructor.instructor_id
28  GROUP BY instructor.instructor_id
29  ORDER BY category_diversity DESC) AS T1 INNER JOIN
30
31  (SELECT instructor.instructor_id, instructor.first_name, instructor.last_name,
32  COUNT(DISTINCT student.region) AS region_diversity
33  FROM instructor,course_instructor,course,enrollment,student
34  WHERE instructor.instructor_id=course_instructor.instructor_id
35  AND course_instructor.course_id=course.course_id
36  AND course.course_id=enrollment.course_id
37  AND enrollment.student_id=student.student_id
38  GROUP BY instructor.instructor_id
39  ORDER BY region_diversity DESC) AS T2 ON T1.instructor_id = T2.instructor_id
40  LIMIT 10;
41
```

100% 51:35

Result Grid | Filter Rows: Search | Export: | Fetch rows:

instructor_id	first_name	last_name	category_diversity	region_diversity
30	Lindon	Bene	11	35
41	Modestia	Coat	10	38
26	Bari	Wootton	9	30
13	Alfonso	Licciardo	8	30
39	Richart	Lavery	8	33
38	Bill	Tredger	8	34
37	Elsinore	Mabone	8	28
3	Haze	Karolewski	8	29
20	Janeen	Mindenhall	8	32
9	Courtenay	Pauls	7	28

Query 19:

Which student is shared more courses than he/she takes? And which student enrolls more courses than he/she is shared? Assumption: students are shared by someone would increase the probability to know the course or this course learning website.

```
44 •  SELECT E.student_id, E.Total_enrollment, S.Total_Shared
45    FROM
46    (SELECT student_id, COUNT(enrollment_id) AS Total_enrollment
47    FROM enrollment
48    GROUP BY student_id
49    ORDER BY COUNT(enrollment_id) DESC ) AS E,
50
51    (SELECT student_id, COUNT(shared_with_student_id) AS Total_Shared
52    FROM student_course_sharing
53    GROUP BY student_id
54    ORDER BY COUNT(shared_with_student_id) DESC) AS S
55    WHERE E.student_id=S.student_id
56    AND E.Total_enrollment < S.Total_Shared
57
100%   40:56   1 error found
```

Result Grid Filter Rows: Search Export:

student_id	Total_enrollment	Total_Shared
100	1	2

```
44 •  SELECT E.student_id, E.Total_enrollment, S.Total_Shared
45    FROM
46    (SELECT student_id, COUNT(enrollment_id) AS Total_enrollment
47    FROM enrollment
48    GROUP BY student_id
49    ORDER BY COUNT(enrollment_id) DESC ) AS E,
50
51    (SELECT student_id, COUNT(shared_with_student_id) AS Total_Shared
52    FROM student_course_sharing
53    GROUP BY student_id
54    ORDER BY COUNT(shared_with_student_id) DESC) AS S
55    WHERE E.student_id=S.student_id
56    AND E.Total_enrollment > S.Total_Shared
57
100%   40:56   2 errors found
```

Result Grid Filter Rows: Search Export:

student_id	Total_enrollment	Total_Shared
101	5	4
58	8	4
16	7	3
169	4	3
4	7	3
50	6	3
57	4	3
76	5	3
10	5	2
106	8	2
114	6	2
119	5	2

Note: 19 SQL queries were written, and 3 graph databases queries were written, bringing the total amount of queries to 22. The 3 remaining queries are presented in the upcoming section.

7. Neo4j Graph Database

- Business Objective

The company wanted to analyze their referral system and extract the following information:

- The most influential students who can potentially play the role of an ambassador for marketing their courses.
- The second-level connections of the potential ambassadors to assess the wider network and influence of that student.
- In case the company selects two ambassadors, they want to ensure that both the connections do not have the same network, and that the shortest connection path is at least above 3.

We performed the analysis using graph databases because analyzing connections between nodes was simpler and quicker in graph databases. Obtaining the same results using relational databases would have required complex joins and sub-queries.

- Importing the Data into Neo4j and Creating the Graph Database

We used Neo4j Desktop locally to import the dataset into Neo4j Database. The data we imported is the data from the shared_with_student_id table we used in our relational database.

This model contains the student nodes and shared relationship between them to capture the information of which students shared which courses with which other students.
The below code was written in Neo4j Desktop to import the data and create the nodes and the relationship:

```
LOAD CSV WITH HEADERS FROM 'file:///student.csv'
AS row FIELDTERMINATOR ';'
WITH row.student_id AS student_id
WHERE row.student_id IS NOT NULL
CREATE (w:Student {student_id: student_id});

LOAD CSV WITH HEADERS FROM 'file:///student_course_sharing.csv'
AS row FIELDTERMINATOR ';'
WITH row.student_id AS student_id, row.shared_with_student_id AS shared_with_student_id,
row.course_id AS course_id
MATCH (u:Student {student_id: student_id})
MATCH (v:Student {student_id: shared_with_student_id})
MERGE (u)-[rel:SHARED{course_id: course_id}]->(v)
RETURN   u, rel, v;
```

The image shown is the image of the entire graph created in Neo4j and exported from Neo4j Database.

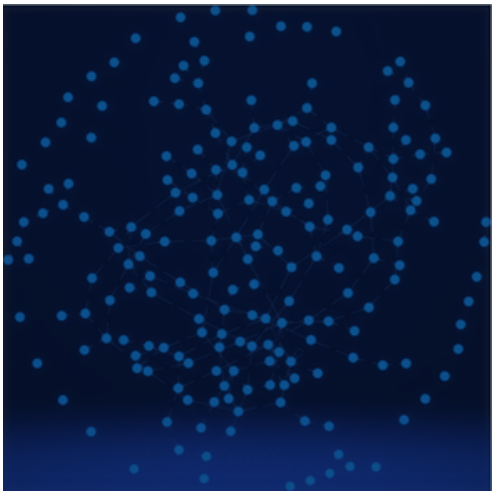


Figure 3 – Neo4j database graph

- **Query 20: Ambassadors - Most Influential Students on the Platform**

Objective: The company wants to identify influential students who can potentially play the role of an ambassador for marketing their courses.

Assumptions:

- We used Harmonic Centrality Algorithm to calculate the most central nodes in the network. The top nodes there would be the most influential students in the network.

```
CALL gds.alpha.closeness.harmonic.stream({
  nodeProjection: 'Student',
  relationshipProjection: 'SHARED'
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).student_id AS Student_ID, ROUND(centrality,4) AS Centrality
ORDER BY Centrality DESC
LIMIT 5;
```

"Student_ID"	"Centrality"
"67"	0.2071
"10"	0.1974
"50"	0.1947
"45"	0.1936
"159"	0.1904

And just to get a visual sense of the central nodes, we wrote match query to show all connections till the 5th level connections for one of these central nodes:

```
MATCH p=(s:Student {student_id:"50"})-[*0..5]→()
RETURN p
```

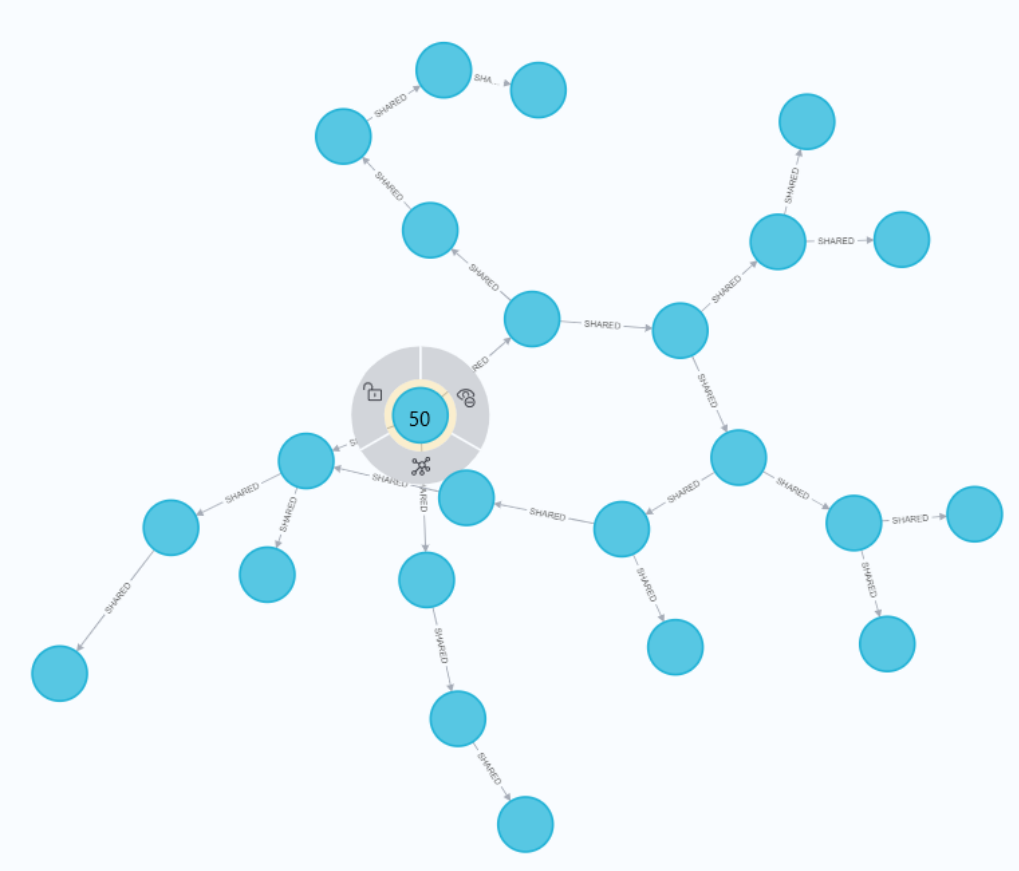


Figure 4 – Multi-layer connection graph for the 50th student

- Query 21: The Network of Potential Ambassadors – 2nd-level Connections

Objective: The company wants to identify the second-level connections of the potential ambassadors found in the previous query to assess the wider network and influence of that student.

Assumptions:

- The query identifies the second-level connections of the student_id = 50 as a sample of the ambassadors introduced previously.
- SoS students are the second-level connections of the focal student.

```
MATCH(focal:Student {student_id: "50"})-[:SHARED]→(friend)-[:SHARED]→(SoS)
WHERE NOT (focal)-[:SHARED]→(SoS) AND NOT SoS=focal
RETURN focal.student_id, SoS.student_id
```

	focal.student_id	SoS.student_id
1	"50"	"8"
2	"50"	"21"
3	"50"	"125"
4	"50"	"98"
5	"50"	"138"

- Query 22: Optimizing in the Selection of Ambassadors

Objective: The company wants to know what is the best usage of its marketing budget in order to choose ambassadors and allocate money.

If two of the chosen ambassadors have a very similar network, or in other words, their network can be easily and quickly reached with the another ambassador, it's not efficient to choose both of them as ambassadors.

For that, we try to find the shortest path between two of the potential ambassadors and if that length is short, then it would be best to choose only one of them as ambassador.

Assumptions:

- The query finds the shortest path between student 159 and 50, which both were among the central nodes and potential ambassadors for the company.

```
MATCH(a:Student{student_id: "159"}),(b:Student {student_id: "50"}), p = shortestPath((a)-
[:SHARED*]-(b))
RETURN length(p)
```

	length(p)
1	4

The shortest path is relatively small, and therefore it might be more efficient to choose only one of 159 and 50 as ambassadors, since each of their network can be reached relatively quickly through the other.

8. Additional insights

8.1. Logic behind the most complex query

Most Complex Query A) Query 5: RFM Model to Segment and Identify Critical Customers

Objective: Identify the customers who need urgent "attention" from the company. Marketing/Sales staff may need to implement retaining measures to get in touch with these customers whether by a coupon specifically designed for them or a friendly reminder e-mail that shows the company values them.

Logic:

To identify these customers, we first need to segment the company's customers. We take the approach of what is called an RFM model used in marketing which considers three different attributes of customers to segment them.

First is *Recency*, which refers to the number of days passed since a customer has purchased from the company.
Second is *Frequency*, which refers to the total number of times a customer has purchased from the company.
Third is *Monetary*, which refers to the total money a customer has spent on the products or services of the company.

In our company and model, we calculated all these three attributes and categorized customers based on them.
For Frequency, customers who purchased more courses from the company than average are labeled "Loyal" customers. Others are labeled as "Normal" customers.

For Recency, Customers who had a greater number of days_since_last_purchase than average are labeled "IDLE" customers. Others are labeled as "Active" customers.

For Monetary, Customers who paid the company more than the average are labeled "Valuable" customers. Others are labeled as "Typical" customers.

The attention-needing customers based on these segments are "Loyal" customers who are "Valuable" but are "IDLE".

Also, it is assumed that the current date of this analysis is the day after the last purchased ticket we produced in the fake data. In other words, days_since_last_purchase is based on November 9th, 2020.

```sql
-- Attention-needing Customers! An RFM Model (Recency-Frequency-Monetary) for Customer Segmentation
-- Creating a table with all the information we need to segment the customers

CREATE VIEW All_Needed_Data AS (
  SELECT student_id, DATEDIFF("2020-11-09", MAX(purchase_date)) AS Days_Since_Last_Purchase, COUNT(*) AS Purchase_Frequency, SUM(Price) AS Total_Monetary_Value
  FROM enrollment INNER JOIN course USING(course_id)
  GROUP BY student_id
);

-- Labeling customers using CASE statements. The details of labeling are explained in query assumptions.

CREATE VIEW Labeled_Customers AS (
  SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,
  CASE
  WHEN Purchase_Frequency > (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN "Loyal"
  WHEN Purchase_Frequency < (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN "Normal"
  END AS Customer_Frequency_Type,
  CASE
  WHEN Days_Since_Last_Purchase > (SELECT AVG(Days_Since_Last_Purchase) FROM All_Needed_Data) THEN "IDLE"
  WHEN Days_Since_Last_Purchase < (SELECT AVG(Days_Since_Last_Purchase) FROM All_Needed_Data) THEN "ACTIVE"
  END AS Customer_Status,
  CASE
  WHEN Total_Monetary_Value > (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN "Valuable"
  WHEN Total_Monetary_Value < (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN "Typical"
  END AS Customer_Value_Type
  FROM All_Needed_Data );

-- Finally, we identify the customers who need attention by filtering on those who are 'Loyal', 'Valuable', and 'IDLE'

SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,  Customer_Frequency_Type, Customer_Value_Type, Customer_Status
FROM Labeled_Customers
WHERE Customer_Frequency_Type = 'Loyal' AND Customer_Value_Type = 'Valuable' AND Customer_Status = 'IDLE'
ORDER BY Days_Since_Last_Purchase DESC
```

student_id	Days_Since_Last_Purchase	Purchase_Frequency	Total_Monetary_Value	Customer_Frequency_Type	Customer_Status	Customer_Value_Type
99	535	6	577	Loyal	IDLE	Valuable
41	468	6	524	Loyal	IDLE	Valuable
122	383	6	461	Loyal	IDLE	Valuable
46	373	7	537	Loyal	IDLE	Valuable
50	371	6	602	Loyal	IDLE	Valuable
6	363	7	672	Loyal	IDLE	Valuable
27	357	7	580	Loyal	IDLE	Valuable
55	343	9	806	Loyal	IDLE	Valuable
197	337	6	650	Loyal	IDLE	Valuable

Most Complex Query B) Query 4: Percentage of Courses Passed in each Difficulty Level

Objective: The company wants to know the passing rate of its courses based on their difficulty level.

Logic:

Since courses do not have difficulty level, we used the quiz difficulty level of each course to define a difficulty level for each course.

To calculate the average difficulty of quizzes, we assigned numbers to difficulties (Easy = 1, Medium = 2, Hard = 3, Very Hard = 4) and calculated the average of that number. Then, based

on the average, we mapped that to one of the difficulty levels and considered the entire course having that difficulty level. The mapping is as follows:
If Average Quiz Difficulty >= 3.5: Course Difficulty Very Hard
If Average Quiz Difficulty < 3.5 and Average Quiz Difficulty >= 2.5: Course Difficulty Hard
If Average Quiz Difficulty < 2.5 and Average Quiz Difficulty >= 1.5: Course Difficulty Medium
If Average Quiz Difficulty < 1.5: Course Difficulty Easy
Also, we defined a passing level of 50% to categorize students to those who failed the course and those who passed. Students with a grade lower than 50% failed the course, and students with a grade equal to or higher than 50% passed.
In the end, we counted the number of passes and fails for each difficulty level and calculate the ratio of those passed. The ratio is (Passed / (Passed + Failed)) and we can see from the results, the passing rate of each of the difficulty levels.

```
CREATE VIEW Course_difficulty_number AS

WITH CTE_Quiz_difficulty_level(course_id, quiz_id, Difficulty_score) AS (
  SELECT course_id, quiz_id, CASE
  WHEN difficulty_level = 'Easy' THEN 1
  WHEN difficulty_level = 'Medium' THEN 2
  WHEN difficulty_level = 'Hard' THEN 3
  WHEN difficulty_level = 'Very_Hard' THEN 4
  END As Difficulty_score
  FROM quiz
  ORDER BY course_id
    )

SELECT course_id, FORMAT(AVG(Difficulty_score),2) AS Course_Difficulty_Number
FROM CTE_Quiz_difficulty_level
GROUP BY course_id
ORDER BY course_id ASC ;

CREATE VIEW Course_Difficulties_AND_Pass_FAIL AS (
  WITH CTE_Course_Difficulty (course_id, Course_Difficulty) AS (
    SELECT course_id, CASE
    WHEN Course_Difficulty_Number >= 3.5 THEN "Very_Hard"
    WHEN Course_Difficulty_Number < 3.5 AND Course_Difficulty_Number >= 2.5 THEN "Hard"
    WHEN Course_Difficulty_Number < 2.5 AND Course_Difficulty_Number >= 1.5 THEN "Medium"
    WHEN Course_Difficulty_Number < 1.5 THEN "Easy"
    END AS Course_Difficulty
    FROM Course_difficulty_number
    )

SELECT COUNT(*) AS Number, Course_Difficulty, CASE
  WHEN final_grade >= 50 THEN "Pass"
  WHEN final_grade <50 THEN "Fail"
  END AS Pass_Fail
FROM CTE_Course_Difficulty INNER JOIN course USING(course_id) INNER JOIN enrollment USING(course_id)
GROUP BY Course_Difficulty, Pass_Fail
ORDER BY COurse_Difficulty DESC ;


SELECT FORMAT((Temp1.Number/Temp2.Total)*100, 2) AS Passing_Rate_Percent, Temp1.Course_Difficulty
FROM Course_Difficulties_AND_Pass_FAIL AS Temp1 LEFT JOIN (
  SELECT SUM(Number) AS Total ,Course_Difficulty
  FROM Course_Difficulties_AND_Pass_FAIL
  GROUP BY Course_Difficulty) AS Temp2
  ON Temp1.Course_Difficulty = Temp2.Course_Difficulty

WHERE Pass_Fail = "Pass"
ORDER BY Passing_Rate_Percent DESC;
```

Passing_Rate_Percent	Course_Difficulty
74.39	Easy
72.60	Very_Hard
68.59	Hard
67.72	Medium

8.2. Insights gained & key learning points

This project allowed us to practically apply the entirety of the concepts that were seen in class. As a complete wrap-up, the project familiarized us with the entire database development lifecycle, from the mission statement, ERD and logical model to the physical model. While creating the database, we instinctively tried to structure it in the most optimal way to be able to query it and extract valuable information. In addition, we also made sure to eliminate any type of dependencies that could exist, therefore avoiding anomalies in our database. This was the normalization of our database. Overall, doing this project gave us the chance to sharpen our ability to design a well-structured and normalized database. Once this was completed, querying the database allowed us to practice writing SQL queries. The entirety of the SQL functions that were seen in class were implemented in our queries. In fact, our queries incorporate joins, unions, views, multiple aggregate functions, "group by" statements, as well as elements that are beyond the scope of the course, such as Window Functions, Common Table Expressions (CTEs) and even a Recency-Frequency-Monetary (RFM) Model, to extract even deeper information from the database, and make the queries even richer, and the insights more meaningful.

Creating Views is useful in terms of making the queries tidier and more readable. View can be used to break down a large code chunk into smaller parts. However, nesting Views should be avoided since it can cause too many data returns for every query, making the database crash. Also be mindful about the execution time, queries that take a long time to get results may need to be reconsidered. We also realized that sometimes it is difficult and not efficient to solve complex queries with only the SQL commands that we have learned in class, and we should explore more advanced SQL commands. For example, the GROUP BY statement can partition the rows of a table into different groups and we can use the aggregation functions to get some statistics about all the rows in each group. However, their power is limited when we would like to look further into each group, for instance, finding the maximum value or ranking the value of a column in each group. In this scenario, the Window Functions like ROW_NUMBER() and RANK()

would do us a huge favour for this type of objectives. Otherwise, we will have to do more table JOINs with only the SQL commands we learned in class.

Our team decided to go one step further by incorporating graph databases into our project. As we know, some queries are complicated to implement while using a traditional relational database, and graph databases offer a more convenient solution. Doing so allowed us to become more familiarized with graph databases, and Neo4j more specifically. We learned how to import .csv files into a graph and create a graph model inside of Neo4j. Furthermore, we had dived into various concepts, such as centrality and multi-level connections. Therefore, we were able to compare the two approaches: using a relational database versus using a graph database, and we were able to appreciate the difference that graph database makes for some complex queries.

We performed the referral system analysis using Neo4j graph databases because analyzing connections between nodes was simpler and quicker in graph databases. Obtaining the same results using relational databases would have required complex joins and sub-queries. We learned to use Neo4j Desktop for importing local database and files and implement query concepts learnt during the course. We also explored centrality algorithms and implemented harmonic centrality to solve one of the business problems. All in all, we learned that Neo4j is more effective to show the relationship between instances of entities, in comparison to SQL. It's more time-consuming and complex to write queries that extract the relationship between entity instances in SQL. When switching from SQL to Neo4j, make sure to change the mindsets from macro to micro.

9. Appendices

Query 1

```
CREATE VIEW a1 AS
SELECT scs.student_id, COUNT(*)*10 AS Credits
FROM student_course_sharing scs
INNER JOIN enrollment AS e
ON scs.shared_with_student_id = e.student_id
WHERE scs.course_id = e.course_id
GROUP BY scs.student_id;

CREATE VIEW a2 AS
SELECT student_id, gift_amount AS Credits
FROM gift_card
GROUP BY student_id;

SELECT s.student_id, s.first_name, s.last_name, CONCAT("$ ", SUM(Credits)) AS total_credits
FROM student s INNER JOIN
(
SELECT *
FROM a1
UNION ALL
SELECT *
FROM a2
) a3
ON a3.student_id = s.student_id
GROUP BY s.student_id
ORDER BY SUM(Credits) DESC;
```

Query 2

```
SELECT
CASE WHEN Student_Age>=0 AND Student_Age<20 THEN "< 20"
WHEN Student_Age>=20 AND Student_Age<30 THEN "20-29"
WHEN Student_Age>=30 AND Student_Age<40 THEN "30-39"
WHEN Student_Age>=40 THEN ">= 40"
END AS Student_Age_Group,
COUNT(*) AS Number_of_Students,
CONCAT("$ ",ROUND(AVG(Average_Price),2)) AS Average_Price
FROM
(
SELECT ROUND(DATEDIFF(date(now()), s.DOB)/365,0) AS Student_Age, Average_Price
FROM student s INNER JOIN
(
SELECT e.student_id, ROUND(AVG(c.price),2) AS Average_Price
FROM enrollment e INNER JOIN course c
```

```
ON e.course_id = c.course_id
GROUP BY student_id
) AS ec
ON s.student_id = ec.student_id
) AS a4
GROUP BY Student_Age_Group
ORDER BY ROUND(AVG(Average_Price),2) DESC;
```

Query 3

```
SELECT
CASE WHEN a5.Video_Length_Per_Course>=0 AND a5.Video_Length_Per_Course<3 THEN "< 3"
WHEN a5.Video_Length_Per_Course>=3 AND a5.Video_Length_Per_Course<9 THEN "3-8"
WHEN a5.Video_Length_Per_Course>=9 AND a5.Video_Length_Per_Course<15 THEN "9-14"
WHEN a5.Video_Length_Per_Course>=15 AND a5.Video_Length_Per_Course<21 THEN "15-20"
WHEN a5.Video_Length_Per_Course>=21 THEN ">= 21"
END AS Video_Length_Interval_Hrs,
SUM(a6.Number_of_Enrollments_Per_Course) AS Number_of_Enrollments
FROM
(
SELECT c.course_id, ROUND(SUM(length_min)/60,0) AS Video_Length_Per_Course
      FROM course c INNER JOIN video v
ON c.course_id = v.course_id
GROUP BY c.course_id
   ) AS a5
INNER JOIN
(
SELECT c.course_id, COUNT(*) AS Number_of_Enrollments_Per_Course
FROM course c INNER JOIN enrollment e
ON c.course_id = e.course_id
GROUP BY c.course_id
   ) AS a6
ON a5.course_id = a6.course_id
GROUP BY Video_Length_Interval_Hrs
ORDER BY Number_of_Enrollments DESC;
```

Query 4

```
CREATE VIEW Course_difficulty_number AS
WITH CTE_Quiz_difficulty_level(course_id, quiz_id, Difficulty_score ) AS (
  SELECT course_id, quiz_id, CASE
  WHEN difficulty_level = 'Easy' THEN 1
  WHEN difficulty_level = 'Medium' THEN 2
  WHEN difficulty_level = 'Hard' THEN 3
  WHEN difficulty_level = 'Very_Hard' THEN 4
  END As Difficulty_score
FROM quiz
ORDER BY course_id
```

```
          )
          SELECT course_id, FORMAT(AVG(Difficulty_score),2)AS
          Course_Difficulty_Number
          FROM CTE_Quiz_difficulty_level
          GROUP BY course_id
          ORDER BY course_id ASC;

         CREATE VIEW Course_Difficulties_AND_Pass_FAIL AS (
         WITH CTE_Course_Difficulty (course_id, Course_Difficulty) AS (
          SELECT course_id, CASE
          WHEN Course_Difficulty_Number >= 3.5 THEN "Very_Hard"
          WHEN Course_Difficulty_Number < 3.5 AND Course_Difficulty_Number >= 2.5  THEN "Hard"
          WHEN Course_Difficulty_Number < 2.5
AND Course_Difficulty_Number >= 1.5  THEN "Medium"
          WHEN Course_Difficulty_Number < 1.5 THEN "Easy"
          END AS Course_Difficulty
          FROM Course_difficulty_number
          )
         SELECT COUNT(*) AS Number, Course_Difficulty, CASE
         WHEN final_grade >= 50 THEN "Pass"
         WHEN final_grade <50 THEN "Fail"
         END AS Pass_Fail
         FROM CTE_Course_Difficulty INNER JOIN course USING(course_id) INNER JOIN enrollment
USING(course_id)
         GROUP BY Course_Difficulty, Pass_Fail
         ORDER BY COurse_Difficulty DESC
          );
         SELECT FORMAT((Temp1.Number/Temp2.Total)*100, 2) AS Passing_Rate_Percent,
Temp1.Course_Difficulty
         FROM Course_Difficulties_AND_Pass_FAIL AS Temp1 LEFT JOIN (
          SELECT SUM(Number) AS Total
,Course_Difficulty FROM Course_Difficulties_AND_Pass_FAIL  GROUP BY Course_Difficulty) AS Temp2
          ON Temp1.Course_Difficulty = Temp2.Course_Difficulty
         WHERE Pass_Fail = "Pass"
         ORDER BY Passing_Rate_Percent DESC;
```

Query 5

```
-- Attention-needing Customers! An RFM Model (Recency-Frequency-Monetary) for Customer
Segmentation

-- Creating a table with all the information we need to segment the customers
CREATE VIEW All_Needed_Data AS (
  SELECT student_id, DATEDIFF("2020-11-09", MAX(purchase_date))
AS Days_Since_Last_Purchase, COUNT(*) AS Purchase_Frequency, SUM(Price)
AS Total_Monetary_Value
FROM enrollment INNER JOIN course USING(course_id)
GROUP BY student_id
```

```
);

-- Labeling customers using CASE statements. The details of labeling are explained in query
assumptions.
CREATE VIEW Labeled_Customers AS (
SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,
CASE
   WHEN Purchase_Frequency > (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN
"Loyal"
   WHEN Purchase_Frequency < (SELECT AVG(Purchase_Frequency) FROM All_Needed_Data) THEN
"Normal"
END AS Customer_Frequency_Type,
CASE
   WHEN Days_Since_Last_Purchase > (SELECT AVG(Days_Since_Last_Purchase)
FROM All_Needed_Data) THEN "IDLE"
   WHEN Days_Since_Last_Purchase < (SELECT AVG(Days_Since_Last_Purchase)
FROM All_Needed_Data) THEN "ACTIVE"
END AS Customer_Status,
CASE
  WHEN Total_Monetary_Value > (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN
"Valuable"
   WHEN Total_Monetary_Value < (SELECT AVG(Total_Monetary_Value) FROM All_Needed_Data) THEN
"Typical"
END AS Customer_Value_Type
FROM All_Needed_Data );


-- Finally, we identify the customers who need attention by filtering on those who are 'Loyal', 'Valuable',
and 'IDLE'
SELECT student_id, Days_Since_Last_Purchase, Purchase_Frequency, Total_Monetary_Value,  Customer
_Frequency_Type, Customer_Value_Type, Customer_Status
FROM Labeled_Customers
WHERE Customer_Frequency_Type = 'Loyal' AND Customer_Value_Type = 'Valuable'
AND Customer_Status = 'IDLE'
ORDER BY Days_Since_Last_Purchase DESC
```

Query 6

```
-- create a table with the total number of shares for each course
CREATE VIEW course_num_shares AS
        SELECT student_course_sharing.course_id, COUNT(student_course_sharing.course_id)
        AS num_shares
        FROM student_course_sharing
        GROUP BY student_course_sharing.course_id
        ORDER BY num_shares DESC;

-- create a table with the maximum number of shares for each category
```

```
CREATE VIEW cat_max_shares AS
        SELECT category.category_id, MAX(course_num_shares.num_shares) AS max_shares
        FROM category INNER JOIN course ON course.category_id = category.category_id
        INNER JOIN
        course_num_shares ON course.course_id = course_num_shares.course_id
        GROUP BY category.category_id;

 -- inner join the two views above with the course and category tables to find the course(s) in each

-- category with the maximum number of shares

SELECT category.category, course.course_title, cat_max_shares.max_shares

FROM cat_max_shares INNER JOIN course ON course.category_id = cat_max_shares.category_id

INNER JOIN course_num_shares ON course_num_shares.course_id = course.course_id

INNER JOIN category ON cat_max_shares.category_id = category.category_id

WHERE course_num_shares.num_shares = cat_max_shares.max_shares

AND cat_max_shares.category_id NOT IN (SELECT category.subcategory_id FROM category);
```

Query 7

```
-- create a table containing info about the number of enrollments in each category in each region

CREATE VIEW region_cat_qty AS

SELECT student.region, category.category_id, COUNT(category.category_id) AS num_enroll

FROM student INNER JOIN enrollment ON student.student_id = enrollment.student_id

INNER JOIN course ON enrollment.course_id = course.course_id

INNER JOIN category ON course.category_id = category.category_id

WHERE category.category_id NOT IN (SELECT category.subcategory_id FROM category)

GROUP BY student.region , category.category_id

ORDER BY student.region ASC , quantity DESC;
```

```
-- get the category with max number of enrollments (most popular) in each region

SELECT region_cat_qty.region, category.category AS MostPopularCategory, max_enroll AS NumOfEnroll

FROM

(SELECT region_cat_qty.region, MAX(quantity) AS max_enroll

FROM region_cat_qty

   GROUP BY region_cat_qty.region) AS subquery

INNER JOIN region_cat_qty ON subquery.region = region_cat_qty.region

      AND subquery.max_enroll = region_cat_qty.quantity

INNER JOIN category ON category.category_id = region_cat_qty.category_id

ORDER BY max_enroll DESC;
```

Query 8

```
SELECT education_level AS EducationLevel, FORMAT(avg(enrollment.final_grade),2) AS
AverageFinalGrade

FROM student, enrollment, (SELECT  enrollment.student_id, enrollment.final_grade FROM enrollment
WHERE course_id = 22) AS t1

WHERE student.student_id = enrollment.student_id

GROUP BY education_level
```

Query 9

```
SELECT course.category_id, instructor.education_level,

        FORMAT(AVG(enrollment.course_rating),2) AS avg_rating

FROM enrollment INNER JOIN course ON enrollment.course_id = course.course_id

INNER JOIN course_instructor ON course.course_id = course_instructor.course_id
```

```
INNER JOIN instructor ON instructor.instructor_id = course_instructor.instructor_id

WHERE course.category_id NOT IN (SELECT category.subcategory_id FROM category)

GROUP BY course.category_id , instructor.education_level

ORDER BY course.category_id;
```

Query 10

```
CREATE  VIEW CourseSharing AS
(SELECT student_id, shared_with_student_id, course_id
FROM student_course_sharing);

SELECT  COUNT(DISTINCT enrollment_id) AS TotalEnrollments ,
COUNT(DISTINCT CourseSharing.student_id) AS EnrollmentsAfterRecommendation
FROM enrollment, CourseSharing
WHERE CourseSharing.shared_with_student_id = enrollment.student_id
```

Query 11

```
-- Create a table containing the information about the total number of enrollments of each

-- course in each category

CREATE VIEW course_totalEnroll AS

SELECT category, course.course_title, t1.TotalEnrollments

FROM course, category,

(SELECT course_id, COUNT(enrollment.enrollment_id) AS TotalEnrollments

FROM enrollment

GROUP BY course_id

ORDER BY TotalEnrollments DESC) AS t1

WHERE t1.course_id = course.course_id

AND course.category_id = category.category_id

AND course.price != 0
```

```
ORDER BY category, TotalEnrollments DESC;


-- Select the course with the highest total number of enrollments (bestseller) in each category

WITH category_enroll_rank AS

(SELECT category, course_title AS BestSeller, TotalEnrollments,

ROW_NUMBER() OVER(PARTITION BY course_totalEnroll.category ORDER BY TotalEnrollments DESC)
`rank`

FROM course_totalEnroll)

SELECT category, BestSeller, TotalEnrollments

FROM category_enroll_rank

WHERE `rank` = 1;
```

Query 12

```
(SELECT course_title AS CourseName, COUNT(enrollment_id) AS TotalEnrollment,
"Bachelor" AS Degree
FROM enrollment, course, student
WHERE enrollment.course_id = course.course_id
AND enrollment.student_id = student.student_id
AND education_level = "Bachelor"
GROUP BY course_title
ORDER BY TotalEnrollment DESC
LIMIT 5)
UNION
(SELECT course_title AS CourseName, COUNT(enrollment_id) AS TotalEnrollment,
"Master" AS Degree
FROM enrollment, course, student
WHERE enrollment.course_id = course.course_id
AND enrollment.student_id = student.student_id
AND education_level = "Master"
GROUP BY course_title
ORDER BY TotalEnrollment DESC
LIMIT 5);
```

Query 13

```
SELECT DISTINCT student.student_id FROM student
```

```
        LEFT JOIN gift_card ON student.student_id = gift_card.student_id
WHERE gift_card.gift_card_id IS NOT NULL AND student.student_id IN
   (SELECT DISTINCT enrollment.student_id FROM enrollment
   INNER JOIN course ON enrollment.course_id = course.course_id
   WHERE course.price != 0);
```

Query 14

```
SELECT (SELECT COUNT(course.course_id) FROM course WHERE course.price =
0)/COUNT(course.course_id) AS free_course_proportion,
(SELECT (SELECT COUNT(DISTINCT enrollment.student_id) FROM enrollment
INNER JOIN course ON enrollment.course_id = course.course_id
WHERE course.price = 0)/COUNT(student.student_id) FROM student) AS free_course_participation
FROM course;
```

Query 15

```
SELECT B.course_id, FORMAT(advanced_average,2)
AS advanced_average, FORMAT(beginner_average,2) AS beginner_average,
CASE WHEN advanced_average > beginner_average THEN 'True' ELSE 'False' END AS advanced_better
FROM
(SELECT enrollment.course_id, AVG(final_grade) AS advanced_average
FROM enrollment
WHERE prior_proficiency = 'Advanced'
   GROUP BY enrollment.course_id) AS A INNER JOIN
(SELECT enrollment.course_id, AVG(final_grade) AS beginner_average
FROM enrollment
WHERE prior_proficiency = 'Beginner'
   GROUP BY enrollment.course_id) AS B ON A.course_id = B.course_id
ORDER BY B.course_id;
```

Query 16

```
SELECT course_id, FORMAT(AVG(complete_period),2) AS Average_period
FROM
(SELECT student_id, course_id, DATEDIFF(completion_date,purchase_date) AS complete_period
FROM enrollment) AS CP
GROUP BY course_id
ORDER BY Average_period DESC;
```

Query 17

```
SELECT difficulty_level, quiz_type,COUNT(quiz_type) AS quantity, SUM(COUNT(quiz_type)) OVER
(PARTITION BY difficulty_level) AS Total
FROM quiz
GROUP BY difficulty_level, quiz_type
ORDER BY difficulty_level;
```

Query 18

```
SELECT T1.instructor_id, T1.first_name, T1.last_name, category_diversity, region_diversity FROM
(SELECT course_instructor.instructor_id, instructor.first_name, instructor.last_name,
COUNT(DISTINCT category.category_id) AS category_diversity
    FROM category LEFT JOIN course ON category.category_id = course.category_id
INNER JOIN course_instructor ON course.course_id = course_instructor.course_id
INNER JOIN instructor ON course_instructor.instructor_id = instructor.instructor_id
    GROUP BY instructor.instructor_id
    ORDER BY category_diversity DESC) AS T1 INNER JOIN
(SELECT instructor.instructor_id, instructor.first_name, instructor.last_name, COUNT(DISTINCT student.r
egion) AS region_diversity
FROM instructor,course_instructor,course,enrollment,student
WHERE instructor.instructor_id=course_instructor.instructor_id
AND course_instructor.course_id=course.course_id
AND course.course_id=enrollment.course_id
AND enrollment.student_id=student.student_id
GROUP BY instructor.instructor_id
ORDER BY region_diversity DESC) AS T2 ON T1.instructor_id = T2.instructor_id
LIMIT 10;
```

Query 19

```
SELECT E.student_id, E.Total_enrollment, S.Total_be_Shared
FROM
(SELECT student_id, COUNT(enrollment_id) AS Total_enrollment
FROM enrollment
GROUP BY student_id
ORDER BY COUNT(enrollment_id) DESC ) AS E,
(SELECT student_id, COUNT(shared_with_student_id) AS Total_be_Shared
FROM student_course_sharing
GROUP BY student_id
ORDER BY COUNT(shared_with_student_id) DESC) AS S
WHERE E.student_id=S.student_id
AND E.Total_enrollment < S.Total_be_Shared

SELECT E.student_id, E.Total_enrollment, S.Total_be_Shared
FROM
(SELECT student_id, COUNT(enrollment_id) AS Total_enrollment
FROM enrollment
GROUP BY student_id
ORDER BY COUNT(enrollment_id) DESC ) AS E,
(SELECT student_id, COUNT(shared_with_student_id) AS Total_be_Shared
FROM student_course_sharing
GROUP BY student_id
ORDER BY COUNT(shared_with_student_id) DESC) AS S
WHERE E.student_id=S.student_id
```

```
AND E.Total_enrollment > S.Total_be_Shared
```

Query 20

```
CALL gds.alpha.closeness.harmonic.stream({
  nodeProjection: 'Student',
  relationshipProjection: 'SHARED'
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).student_id AS Student_ID, ROUND(centrality,4) AS Centrality
ORDER BY Centrality DESC
LIMIT 5;

MATCH p=(s:Student {student_id:"50"})-[*0..5]->()
RETURN p
```

Query 21

```
MATCH(focal:Student {student_id: "50"})-[:SHARED]->(friend)-[:SHARED]->(SoS)
WHERE NOT (focal)-[:SHARED]->(SoS) AND NOT SoS=focal
RETURN focal.student_id, SoS.student_id
```

Query 22

```
MATCH(a:Student{student_id: "159"}),(b:Student {student_id: "50"}), p = shortestPath((a)-[:SHARED*]-
(b))
RETURN length(p)
```