DATABASE HOMEWORK 2

23rd May,2021

AYETIJHYA DESMUKHYA

309127

"I certify that this assignment is entirely my own work, performed independently and without any help from the sources which are not allowed."

Ayetijhya Desmukhya

TASK 1 – DATABASE MODELLING

We create the first table of our database i.e [dbo].[University] which is introduced in order to have a one to many relationship with [dbo].[Departments].

```
CREATE TABLE University(
    [UniversityID] [varchar](10) NOT NULL,
    [UniversityName] [varchar](50) NOT NULL,
    CONSTRAINT PK_University PRIMARY KEY (UniversityID)
 );

 CREATE TABLE Departments(
    [DepartmentID] [varchar](10) NOT NULL,
    [DepartmentName] [varchar](50) NOT NULL,
    [UniversityID] [varchar](10) NOT NULL,
    CONSTRAINT PK_Departments PRIMARY KEY (DepartmentID),
    CONSTRAINT FK_Departments_University FOREIGN KEY(UniversityID) REFERENCES University(UniversityID)
);
```

Next we will create [dbo].[Courses] and [dbo].[Prerequisites].

```
CREATE TABLE Courses(
    [CourseID] [varchar](10) NOT NULL,
    [CourseName] [varchar](50) NOT NULL,
    [DepartmentID] [varchar](10) NOT NULL,
    [CourseYear] [int] NOT NULL,
    [CourseSem] [int] NOT NULL,
    [isElective] [bit] NOT NULL,
    [MinStudents] [int] NOT NULL,
    [MaxStudents] [int] NULL,
    [CourseType] [varchar](20) NOT NULL,
    CONSTRAINT PK_Course PRIMARY KEY (CourseID),
    CONSTRAINT FK_Courses_Departments FOREIGN KEY(DepartmentID) REFERENCES Departments(DepartmentID),
);

CREATE TABLE Prerequisites(
    [PrerequisiteID] int NOT NULL IDENTITY(1000,1),
    [CourseID] [varchar](10) NOT NULL,
    [PrerequisiteCourseID] [varchar](10) NOT NULL,
    CONSTRAINT PK_Prerequisites PRIMARY KEY(PrerequisiteID),
    CONSTRAINT FK_Prerequisites_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_PrerequisitesCourse_Courses FOREIGN KEY(PrerequisiteCourseID) REFERENCES Courses(CourseID)
);
```

From the [dbo].[Courses] table shown above it is understandable that [dbo].[Departments] define a one to many relationship with [dbo].[Courses]. I decided to put nullability property to [MaxStudents] in [dbo].[Courses] since it can be extended to any limit for a particular course not implementing that attribute.

Next we have, [dbo].[Rooms] handled by individual departments

```sql
CREATE TABLE Rooms(
    [RoomID] [int] NOT NULL,
    [DepartmentID] [varchar](10) NOT NULL,
    [RoomType] [varchar](10) NOT NULL,
    CONSTRAINT PK_Rooms PRIMARY KEY(RoomID,DepartmentID),
    CONSTRAINT FK_Rooms_Departments FOREIGN KEY(DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Here, a composite primary key was created since different departments can have the same room ID or room number.If we kept RoomID as our only primary key our records wouldn't have been unique which would result in an error. So both RoomID and DepartmentID are required for unique records in the table.

Next up is, [dbo].[Employees]

```sql
CREATE TABLE Employees(
    [EmployeeID] [varchar](10) NOT NULL,
    [DepartmentID] [varchar](10) NOT NULL,
    [EmployeeName] [varchar](50) NOT NULL,
    [EmployeeAddress] [varchar](50) NOT NULL,
    [EmployeeEmail] [varchar](30) NOT NULL,
    [EmployeePhoneNumber] [varchar](20) NOT NULL,
    [SupervisorID] [int] NULL,
    CONSTRAINT PK_Employees PRIMARY KEY(EmployeeID),
    CONSTRAINT FK_Employees_Departments FOREIGN KEY(DepartmentID) REFERENCES Departments(DepartmentID),
);
```

Here, every employee has a supervisor. According to my imagination, a supervisor is also an employee so a supervisor has a supervisor? If it was true, then there should be a supervisor who supervises a supervisor and in turn has a supervisor. Sounds confusing.

In short, there should be a supervisor/head employee at the top of the hierarchy who doesn't have a supervisor. So SupervisorID at some point should be NULL.

With this we come to the one to one relationships with [dbo].[Employees] - [dbo].[Supervisors] and [dbo].[Employees]- [dbo].[Teachers]

```sql
CREATE TABLE Supervisors(
    [SupervisorID] [int] NOT NULL IDENTITY(2000,1),
    [EmployeeID] [varchar](10) NOT NULL,
    [SupervisorRole] [varchar](50) NULL,
    CONSTRAINT PK_Supervisors PRIMARY KEY (EmployeeID),
    CONSTRAINT FK_Supervisors_Employees FOREIGN KEY(EmployeeID) REFERENCES Employees(EmployeeID)
 );

CREATE TABLE Teachers(
    [TeacherID] [int] NOT NULL IDENTITY(3000,1),
    [EmployeeID] [varchar](10) NOT NULL,
    [TeacherRole] [varchar](50) NULL,
    CONSTRAINT PK_Teachers PRIMARY KEY (EmployeeID),
    CONSTRAINT FK_Teachers_Employees FOREIGN KEY(EmployeeID) REFERENCES Employees(EmployeeID),
);
```

Next, [dbo].[TeachersCourses] [dbo].[TeachingHistories] are basically tables defining the courses a teacher can take and the teacher's teaching history.

```
CREATE TABLE TeachersCourses(
    [TeacherCoursesID] [int] NOT NULL IDENTITY(4000,1),
    [EmployeeID] [varchar](10) NOT NULL,
    [CourseID] [varchar](10) NOT NULL,
    CONSTRAINT PK_TeachersCourses PRIMARY KEY (TeacherCoursesID),
    CONSTRAINT FK_TeachersCourses_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_TeachersCourses_Teachers FOREIGN KEY(EmployeeID) REFERENCES Teachers(EmployeeID)
);

CREATE TABLE TeachingHistories(
    [TeacherHistoryID] [int] NOT NULL IDENTITY(5000,1),
    [CourseID] [varchar](10) NOT NULL,
    [EmployeeID] [varchar](10) NOT NULL,
    [TeachingYear] [int] NOT NULL,
    CONSTRAINT PK_TeachingHistories PRIMARY KEY (TeacherHistoryID),
    CONSTRAINT FK_TeachingHistories_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_TeachersHistories_Teachers FOREIGN KEY(EmployeeID) REFERENCES Teachers(EmployeeID)
);
```

Next we have [dbo].[Students] which is pretty much self-explanatory

```
CREATE TABLE Students(
    [StudentID] [int] NOT NULL IDENTITY(6000,1),
    [StudentName] [varchar](50) NOT NULL,
    [DepartmentID] [varchar](10) NOT NULL,
    [StudentAddress] [varchar](50) NOT NULL,
    [StudentEmail] [varchar](50) NOT NULL,
    [StudentPhoneNumber] [varchar](50) NOT NULL,
    [StudentYear] [int] NOT NULL,
    [StudentSemester] [int] NOT NULL,
    CONSTRAINT PK_Student PRIMARY KEY (StudentID),
    CONSTRAINT FK_Students_Departments FOREIGN KEY(DepartmentID) REFERENCES Departments(DepartmentID),
);
```

[dbo].[Groups] define the the course and teacher the group is assigned to. So before creating tables for courses taken by students, we first create groups so that the student can be assigned to the group was they decide to take up a particular course.

```
CREATE TABLE Groups(
    [GroupID] [int] NOT NULL IDENTITY(7000,1),
    [CourseID] [varchar](10) NOT NULL,
    [EmployeeID] [varchar](10) NOT NULL,
    [GroupNumber] [int] NOT NULL,
    CONSTRAINT PK_Groups PRIMARY KEY (GroupID),
    CONSTRAINT FK_Groups_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_Groups_Teachers FOREIGN KEY(EmployeeID) REFERENCES Teachers(EmployeeID)
);
```

[dbo].[Classes] are the further additional specifications of a group

```sql
CREATE TABLE Classes(
    [GroupID] [int] NOT NULL,
    [DepartmentID] [varchar](10) NOT NULL,
    [RoomID] [int] NOT NULL,
    [ClassDay] [varchar](15) NOT NULL,
    [TimeFrom] [time](0) NOT NULL,
    [TimeTo] [time](0) NOT NULL,
    [TotalHours] [decimal](3,2) NOT NULL,
    CONSTRAINT PK_Classes PRIMARY KEY (GroupID),
    CONSTRAINT FK_Classes_Groups FOREIGN KEY(GroupID) REFERENCES Groups(GroupID),
    CONSTRAINT FK_Classes_Roooms FOREIGN KEY(RoomID,DepartmentID) REFERENCES Rooms(RoomID,DepartmentID)
)
```

Now, we create [dbo].[StudentCourses] and [dbo].[StudentHistories], which is pretty similar to the idea of [dbo].[TeachersCourses] and [dbo].[TeachingHistories] respectively.

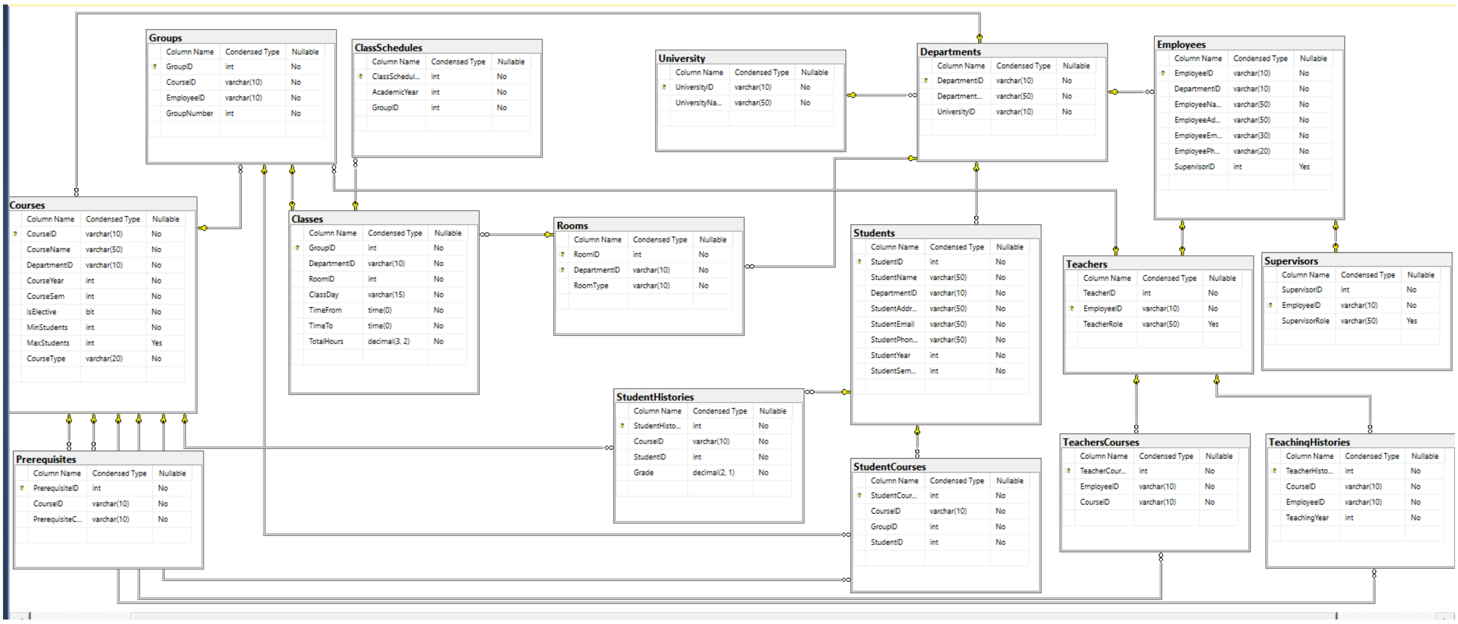
```sql
CREATE TABLE StudentCourses(
    [StudentCoursesID] [int] NOT NULL IDENTITY(8000,1),
    [CourseID] [varchar](10) NOT NULL,
    [GroupID] [int] NOT NULL,
    [StudentID] [int] NOT NULL,
    CONSTRAINT PK_StudentCourses PRIMARY KEY (StudentCoursesID),
    CONSTRAINT FK_StudentCourses_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_StudentCourses_Groups FOREIGN KEY(GroupID) REFERENCES Groups(GroupID),
    CONSTRAINT FK_StudentCourses_Students FOREIGN KEY(StudentID) REFERENCES Students(StudentID)
);

CREATE TABLE StudentHistories(
    [StudentHistoryID] [int] NOT NULL IDENTITY(9000,1),
    [CourseID] [varchar](10) NOT NULL,
    [StudentID] [int] NOT NULL,
    [Grade] [decimal](2, 1) NOT NULL,
    CONSTRAINT PK_StudentHistories PRIMARY KEY (StudentHistoryID),
    CONSTRAINT FK_StudentHistories_Courses FOREIGN KEY(CourseID) REFERENCES Courses(CourseID),
    CONSTRAINT FK_StudentHistories_Students FOREIGN KEY(StudentID) REFERENCES Students(StudentID)
);
```

Lastly, we have [dbo].[ClassSchedules] which is the summation of all the classes conducted in a particular academic year.

```sql
CREATE TABLE ClassSchedules(
    [ClassScheduleID] [int] NOT NULL IDENTITY(10000,1),
    [AcademicYear] [int] NOT NULL,
    [GroupID] [int] NOT NULL,
    CONSTRAINT PK_ClassSchedules PRIMARY KEY (ClassScheduleID),
    CONSTRAINT FK_ClassSchedules_Group FOREIGN KEY(GroupID)REFERENCES Classes(GroupID),
);
```

Now, our database tables are set, so we create our Entity Diagram for the database. After some adjustments, we get this:



<u>TASK 2 – SAMPLE INSERTS AND MODIFICATIONS ANALYSIS</u>

After our database is set up, we will insert some sample rows for testing out the queries, procedures. I have tried to create a decent number of rows in order to make things work out. Below is the total number of rows inserted in each table of the database.

TABLE_NAME	NO_OF_ROWS
University	1
Departments	5
Courses	15
Prerequisites	6
Rooms	10
Employees	17
Supervisors	5
Teachers	9
TeachersCourses	18
TeachingHistories	27
Students	14
Groups	17
Classes	17
StudentCourses	16
StudentHistories	21
ClassSchedule	17

Next, we carry out some modifications of rows as required by Task 2.3

```
--Modifications on Rooms Table
select * from Rooms;

begin transaction

--Insert 3 rows
INSERT INTO Rooms(RoomID,RoomType,DepartmentID) VALUES(500,'Lab','MIS');
INSERT INTO Rooms(RoomID,RoomType,DepartmentID) VALUES(401,'Lecture','EIT');
INSERT INTO Rooms(RoomID,RoomType,DepartmentID) VALUES(305,'Lab','PAE');

--Update 3 rows
UPDATE Rooms SET RoomType = 'Lecture' WHERE RoomID=500;
UPDATE Rooms SET DepartmentID = 'EIT' WHERE RoomID=500;
UPDATE Rooms SET RoomType = 'Lecture' WHERE RoomType = 'Lab';

--Delete the 3 rows created before
DELETE FROM Rooms WHERE RoomID <=500 AND RoomID >=305;

rollback transaction
```

Here, we have some DML operations to carry out and demonstrate the basic INSERT, UPDATE and DELETE statements. In order to be on the safe side and to avoid permanent changes in the data of the database we use begin transaction and rollback.

As shown by the picture, the modifications are carried out on [dbo].[Rooms] where I inserted 3 random rooms from 3 different departments; then updated some details using SET and WHERE clause and deleted the rooms the I inserted previously.

TASK 3 – INTRODUCING INDEXES

In this part I created Non-Clustered Index mostly in order to speed up queries for specific reasons. As we know a table can have only one Clustered Index per table which is created during the creation of primary keys of that table, we leave it untouched so as not to overwrite any undesired information.

```
--For query searches regarding StudentYear & Semesters
CREATE NONCLUSTERED INDEX [IDX_Sudents_StudentYear]
ON [dbo].[Students] ([StudentYear] ASC,[StudentSemester] ASC) INCLUDE(StudentID,StudentName)

--For query searches regarding students from a particular department
CREATE NONCLUSTERED INDEX [IDX_Sudents_DepartmentID]
ON [dbo].[Students] ([DepartmentID] ASC) INCLUDE(StudentID,StudentName)

--For query searches within specific time
CREATE NONCLUSTERED INDEX [IDX_Classes_TimeFrom]
ON [dbo].[Classes] ([DepartmentID] ASC,[TimeFrom] ASC,[TimeTo] ASC)
```

```sql
--For query searches specific to Electives
CREATE NONCLUSTERED INDEX [IDX_Courses_isElective]
ON [dbo].[Courses] ([CourseID] ASC,[isElective] DESC)

--For query searches regarding Year, Semester wise Courses
CREATE NONCLUSTERED INDEX [IDX_Courses_DepartmentID_CourseYear_CourseSem]
ON [dbo].[Courses] ([DepartmentID] ASC,[CourseYear] ASC,[CourseSem] ASC) INCLUDE (CourseID)

--For speed up of search of DepartmentID
CREATE NONCLUSTERED INDEX [IDX_Departments_DepartmentID]
ON [dbo].[Departments] ([DepartmentID] ASC)


-- For query searches on groups of a particular Department
CREATE NONCLUSTERED INDEX [IDX_Groups_CourseID]
ON [dbo].[Groups] ([CourseID] ASC) INCLUDE (GroupID,EmployeeID)

--Similar function to the previous index
CREATE NONCLUSTERED INDEX [IDX_Prerequisites_CourseID]
ON [dbo].[Prerequisites] ([CourseID] ASC) INCLUDE (PrerequisiteCourseID)

--For query searches regarding a particular department
CREATE NONCLUSTERED INDEX [IDX_Rooms_DepartmentID]
ON [dbo].[Rooms] ([DepartmentID] ASC) INCLUDE (RoomType)

--For query searches related to a particular student/students
CREATE NONCLUSTERED INDEX [IDX_StudentCourses_StudentID]
ON [dbo].[StudentCourses] ([StudentID] ASC) INCLUDE (CourseID,GroupID)

--Similar to previous index
CREATE NONCLUSTERED INDEX [IDX_StudentHistories_StudentID]
ON [dbo].[StudentHistories] ([StudentID] ASC) INCLUDE (CourseID,Grade)

--For query searches regarding a particular role
CREATE NONCLUSTERED INDEX [IDX_Supervisors_SupervisorRole]
ON [dbo].[Supervisors] ([SupervisorRole] ASC) INCLUDE (EmployeeID)


--Similar to previous index
CREATE NONCLUSTERED INDEX [IDX_Teachers_TeacherRole]
ON [dbo].[Teachers] ([TeacherRole] ASC) INCLUDE (EmployeeID)


--For query searches regarding a particular employee
CREATE NONCLUSTERED INDEX [IDX_TeachersCourses_EmployeeID]
ON [dbo].[TeachersCourses] ([EmployeeID] ASC) INCLUDE (CourseID)

--Similar to previous index
CREATE NONCLUSTERED INDEX [IDX_TeachingHistories_EmployeeID]
ON [dbo].[TeachingHistories] ([EmployeeID] ASC) INCLUDE (CourseID)
```

There are more possible ways to create indexes, but we stop here for the sake of this Project. Just the disadvantage of non-clustered index is that it stores the columns in a different table with it's row locater to trace back to the original row of the specified table. In short, lookup process for such indexes become costly but on the other hand <u>retrieving data becomes faster, we can avoid/reduce the overhead cost associated with clustered indexes</u>

TASK 4 – QUERIES

QUERY 1

```sql
--Q1 -> A total number of students at each department for each year in the database.

select Departments.DepartmentID,
sum(case when (Students.StudentYear=1) then 1 else 0 end) as TotalStudentsYear1,
sum(case when (Students.StudentYear=2) then 1 else 0 end) as TotalStudentsYear2,
sum(case when (Students.StudentYear=3) then 1 else 0 end) as TotalStudentsYear3,
sum(case when (Students.StudentYear=4) then 1 else 0 end) as TotalStudentsYear4
from Departments
inner join Students on Students.DepartmentID=Departments.DepartmentID
group by Departments.DepartmentID
```

108 %

Results | Messages | Execution plan

	DepartmentID	TotalStudentsYear1	TotalStudentsYear2	TotalStudentsYear3	TotalStudentsYear4
1	AWM	2	0	0	0
2	CEM	3	0	0	0
3	EIT	1	0	0	0
4	MIS	2	2	2	1
5	PAE	1	0	0	0

To prove:

Using, `select * from Students` we get all the students from all the departments

	StudentID	StudentName	DepartmentID	StudentAddress	StudentEmail	StudentPhoneNumber	StudentYear	StudentSemester
1	6000	Natasha Long	CEM	633 Nicholas Street	nlong@wut.edu.us	785-277-5294	1	2
2	6001	Barbara Kennedy	CEM	3524 New Street	bkennedy@wut.edu.us	541-295-5528	1	2
3	6002	Kyle Diaz	CEM	1729 Luke Lane	kdiaz@wut.edu.us	580-239-4646	1	2
4	6003	Bethany K Matthews	PAE	504 Willison Street	bmatthews@wut.edu.us	763-221-3154	1	1
5	6004	Brent Beckwith	EIT	3271 C Street	bbeckwith@wut.edu.us	508-391-3946	1	2
6	6005	Kirsten Tucker	AWM	34 Rothbury Terrace, Newcastle Upon Tyne	ktucker@wut.edu.us	765-765-5654	1	2
7	6006	Cooper Buck	AWM	1 Scarborough Grove, Shipley	cbuck@wut.edu.us	234-345-3444	1	2
8	6007	Giancarlo Roberson	MIS	8 Reed Close, Larkfield	groberson@wut.edu.us	766-134-6564	1	1
9	6008	Warren Cline	MIS	224 Towngate, Ossett	wcline@wut.edu.us	565-566-5294	1	2
10	6009	Stacy Hanson	MIS	8 Morpeth Avenue	shanson@wut.edu.us	750-098-5565	2	3
11	6010	Esther Golden	MIS	19 Parkway, Huntingdon	egolden@wut.edu.us	234-275-2334	2	4
12	6011	Angel Hodges	MIS	21 Gorse Bank Road, Hale Barns	ahodges@wut.edu.us	776-355-5347	3	5
13	6012	Cheyanne Mcconnell	MIS	3 Bell Close, Stilton	cmcconnell@wut.edu.us	908-899-8789	4	7
14	6013	Elisa Johns	MIS	59 James Bond Street, Tamarki	ejohns@wut.edu.us	354-768-1238	3	6

- CEM -> 3 1st year students
- PAE -> 1 1st year student
- EIT -> 1 1st year student
- AWM -> 2 1st year students
- MIS -> 2 1st year students, 2 2nd year students, 2 3rd year students, 1 4th year student.

Comparing the results, we see that our query statement was correct.

QUERY 2

```sql
--Q2 -> A list of 10 courses taken by most students in a single academic year

select top(10) CourseName,CourseID,Total_Students from
(select Courses.CourseName,Courses.CourseID,count(StudentCourses.CourseID)as Total_Students from Courses
inner join StudentCourses on StudentCourses.CourseID=Courses.CourseID
group by CourseName,Courses.CourseID)as Total_Courses_Taken
order by Total_Students desc,CourseID ASC;
```

108 %

Results | Messages | Execution plan

	CourseName	CourseID	Total_Students
1	Mathematics II	CEM-M2004	3
2	Engineering Analysis and Numerical Methods	AWM-ANM003	2
3	Artificial Intelligence Fundamentals	MIS-AIF009	2
4	Introduction to Machine Learning	MIS-IML012	2
5	Engineering Physics-II	EIT-EP2004	1
6	Algorithmns and Computability	MIS-AC011	1
7	Calculus-I	MIS-C1001	1
8	Calculus-II	MIS-C2004	1
9	Databases	MIS-D006	1
10	Introduction to Natural Processing Language	MIS-NPL013	1

To prove:

We know,

```sql
select count(*) as Total_Courses from Courses
```

108 %

Results | Messages

	Total_Courses
1	15

the total number of courses.

Courses taken by students include,

Results | Messages | Execution plan

	CourseName	CourseID	Total_Students
1	Engineering Analysis and Numerical Methods	AWM-ANM003	2
2	Mathematics II	CEM-M2004	3
3	Engineering Physics-II	EIT-EP2004	1
4	Algorithmns and Computability	MIS-AC011	1
5	Artificial Intelligence Fundamentals	MIS-AIF009	2
6	Calculus-I	MIS-C1001	1
7	Calculus-II	MIS-C2004	1
8	Databases	MIS-D006	1
9	Introduction to Machine Learning	MIS-IML012	2
10	Introduction to Natural Processing Language	MIS-NPL013	1
11	Elements of Aeronautics	PAE-EA003	1

Finally, we get the top 10 courses by sorting the total number of students in descending order.

<u>QUERY 3</u>

```sql
--Q3 -> A list of teachers whose classes were taken by more than 150% of average number of students
--per teacher (in all classes in the database, for all years).

with tot_studs as(select EmployeeID,count(EmployeeID) as tot from Groups
inner join StudentCourses on StudentCourses.GroupID=Groups.GroupID
group by EmployeeID)
select Employees.EmployeeID,DepartmentID,EmployeeName,EmployeeAddress,EmployeeEmail,EmployeePhoneNumber,SupervisorID
from Employees
inner join tot_studs on tot_studs.EmployeeID=Employees.EmployeeID
where tot_studs.tot > cast(1.5*(select avg(cast(tot_studs.tot as decimal(10,2))) from tot_studs)as decimal(10,2))
```

108 %

Results | Messages | Execution plan

	EmployeeID	DepartmentID	EmployeeName	EmployeeAddress	EmployeeEmail	EmployeePhoneNumber	SupervisorID
1	MIS011	MIS	Donna R Fox	221 Rosemont Avenue	uab0zt@temporary-mail.net	593-950-9669	2004
2	MIS012	MIS	Mark L Harris	801 Nutter Street	afrgh0zt@temporary-mail.net	492-789-1112	2004

To prove:

The total students per teacher in this case:

Results | Messages | Execution plan

	EmployeeID	tot
1	AWM002	2
2	CEM002	3
3	EIT002	1
4	MIS011	4
5	MIS012	4
6	MIS014	1
7	PAE002	1

So, if we calculate average, it is approximately equal to $(2+3+1+4+4+1+1)/7 \approx 2.29$

We have to find teachers who have more than 150% or 1.5 of the average ≈ 3.43

The only two teachers satisfying this criterion is MIS012 and MIS014.

QUERY 4

```sql
--Q4 -> Maximum number of hours taught in a single room in a week, and the year and the semester
--when it happened.

select  Courses.DepartmentID, CourseYear,CourseSem,RoomID,count(RoomID)as RoomUsage,sum(TotalHours) as MaxHours from Courses
inner join Groups on Groups.CourseID=Courses.CourseID
inner join Classes on Classes.GroupID=Groups.GroupID
group by Courses.DepartmentID, CourseYear,CourseSem,RoomID
```

108 %

Results | Messages | Execution plan

	DepartmentID	CourseYear	CourseSem	RoomID	RoomUsage	MaxHours
1	AWM	1	1	111	1	3.00
2	AWM	1	2	101	1	3.00
3	CEM	1	1	101	1	2.00
4	CEM	1	2	101	1	2.00
5	EIT	1	1	101	1	3.00
6	EIT	1	1	200	1	3.00
7	EIT	1	2	101	1	3.00
8	MIS	1	1	101	1	3.00
9	MIS	1	2	101	1	2.00
10	MIS	2	4	202	1	2.00
11	MIS	3	5	202	2	6.00
12	MIS	3	6	202	2	5.00
13	MIS	4	7	202	2	6.00
14	PAE	1	1	117	1	3.00

To prove:

```sql
select Groups.GroupID,Groups.CourseID,Classes.DepartmentID,
RoomID,TotalHours,CourseYear,CourseSem from Groups
inner join Classes on Classes.GroupID=Groups.GroupID
inner join Courses on Courses.CourseID=Groups.CourseID
```

.08 %

Results | Messages

	GroupID	CourseID	DepartmentID	RoomID	TotalHours	CourseYear	CourseSem
1	7001	AWM-ET002	AWM	111	3.00	1	1
2	7000	AWM-ANM003	AWM	101	3.00	1	2
3	7002	CEM-M1003	CEM	101	2.00	1	1
4	7003	CEM-M2004	CEM	101	2.00	1	2
5	7004	EIT-EP1001	EIT	101	3.00	1	1
6	7005	EIT-EP1002	EIT	200	3.00	1	1
7	7006	EIT-EP2004	EIT	101	3.00	1	2
8	7010	MIS-C1001	MIS	101	3.00	1	1
9	7011	MIS-C2004	MIS	101	2.00	1	2
10	7012	MIS-D006	MIS	202	2.00	2	4
11	7013	MIS-IML012	MIS	202	3.00	3	5
12	7014	MIS-IML012	MIS	202	3.00	3	5
13	7009	MIS-AIF009	MIS	202	2.00	3	6
14	7015	MIS-NPL013	MIS	202	3.00	3	6
15	7007	MIS-AC011	MIS	202	3.00	4	7
16	7008	MIS-AC011	MIS	202	3.00	4	7
17	7016	PAE-EA003	PAE	117	3.00	1	1

So here we can see how the rooms are used and total hours spent per year per semester and department wise.

In order to calculate the maximum number of hours spent in a room per year per semester, we just sum all the hours of the classes taken in that room with a span of a week.

For example,

We want to know the maximum hours spent in Room 202 from 'MIS' department during the 6th semester

From the picture we see it is a total of 5 hours and result is same for the query statement executed earlier.

QUERY 5

```sql
--Q5 -> Total number of students who have classes from 8:00 to 20:00 on Monday, without a break of at
--least 1h

with get_filtered_student as(select StudentCourses.StudentID,TimeFrom,TimeTo,ClassDay,
Lag(TimeTo, 1) OVER(PARTITION BY StudentID ORDER BY TimeFrom ASC) AS LastClassEnd from StudentCourses
inner join Classes on Classes.GroupID=StudentCourses.GroupID
where TimeFrom >= '8:00' and TimeTo <= '20:00'and ClassDay='Monday')
,get_time_gap as(select *,SUM(ABS(datediff(MINUTE,TimeFrom,LastClassEnd))) OVER (PARTITION BY StudentID) as TotalTimeGap
from get_filtered_student)
select count(distinct StudentID) as Total_Students from get_time_gap where TotalTimeGap <= 60;
```

08 %

Results | Messages | Execution plan

	Total_Students
1	0

To prove:

We will go step wise here, we first execute the select statement of get_filtered_student:

	StudentID	TimeFrom	TimeTo	ClassDay	LastClassEnd
1	6000	10:30:00	12:30:00	Monday	NULL
2	6001	10:30:00	12:30:00	Monday	NULL
3	6002	10:30:00	12:30:00	Monday	NULL
4	6004	11:30:00	14:30:00	Monday	NULL
5	6005	08:00:00	11:00:00	Monday	NULL
6	6006	08:00:00	11:00:00	Monday	NULL
7	6007	15:00:00	18:00:00	Monday	NULL
8	6012	08:00:00	11:00:00	Monday	NULL
9	6012	15:00:00	17:00:00	Monday	11:00:00
10	6013	15:00:00	17:00:00	Monday	NULL

Here, we see that we have added the end time(LastClassEnd) of the previous class of students partitioned by the student ID.

The NULL points to the fact that those students don't have any classes prior to the present one.

We have also filtered out students having classes before 8:00 and after 20:00. One more filter was added to the day of class and with WHERE clause we chucked out any classes that took place on any other day than Monday.

Next, we have get_time_gap, which when executed, we get:

	StudentID	TimeFrom	TimeTo	ClassDay	LastClassEnd	TotalTimeGap
1	6000	10:30:00	12:30:00	Monday	NULL	NULL
2	6001	10:30:00	12:30:00	Monday	NULL	NULL
3	6002	10:30:00	12:30:00	Monday	NULL	NULL
4	6004	11:30:00	14:30:00	Monday	NULL	NULL
5	6005	08:00:00	11:00:00	Monday	NULL	NULL
6	6006	08:00:00	11:00:00	Monday	NULL	NULL
7	6007	15:00:00	18:00:00	Monday	NULL	NULL
8	6012	08:00:00	11:00:00	Monday	NULL	240
9	6012	15:00:00	17:00:00	Monday	11:00:00	240
10	6013	15:00:00	17:00:00	Monday	NULL	NULL

Now, we simply find out the time gap between TimeFrom and LastClassEnd in minutes and sum up all the breaks the student has received throughout the day.

If this time gap is <= 60 then we count the student with the distinct student ID since we don't want duplicates in our total count.

The logic behind time gap <= 60:

Task provided said – 'without a break of at least an hour'

My logic said – negate the statement -> 'with a break of at most an hour'.

Since, in my database I don't have students continuously from 8:00 to 20:00 that is why Total_Students returned was 0.

TASK 5 – STORED PROCEDURE

```
CREATE PROCEDURE replaceTeacher
@GroupID int,
@Date date AS SET NOCOUNT ON;
with teacher_sick as(
select Groups.GroupID,EmployeeID,TimeFrom,TimeTo,CourseID,ClassDay from Classes
inner join Groups on Groups.GroupID=Classes.GroupID
where Groups.GroupID=@GroupID
),
teachers_available as(
select TeachersCourses.EmployeeID,teacher_sick.CourseID from teacher_sick
inner join TeachersCourses on TeachersCourses.CourseID=teacher_sick.CourseID
where TeachersCourses.EmployeeID!=teacher_sick.EmployeeID
),
get_timing as(
select Groups.EmployeeID,TimeFrom,teachers_available.CourseID from teachers_available
inner join Groups on Groups.EmployeeID=teachers_available.EmployeeID
inner join Classes on Classes.GroupID=Groups.GroupID
where ClassDay =datename(weekday,@Date)
)
select distinct @Date as DateOfClass,teacher_sick.EmployeeID as Teacher,teacher_sick.CourseID,teacher_sick.ClassDay,
teacher_sick.TimeFrom, teacher_sick.TimeTo,get_timing.EmployeeID as Substitute_Teacher from get_timing
join teacher_sick on teacher_sick.CourseID=get_timing.CourseID
where get_timing.TimeFrom < teacher_sick.TimeFrom or get_timing.TimeFrom>teacher_sick.TimeTo
GO
```

As per task, @GroupID and @Date are the two input parameters.

In teacher_sick, we find out the details of the teacher and the class specifications along with the course so that in teachers_available we can use this to join with TeachersCourses on the course ID and get the teachers who can teach that particular course except the teacher who needs to be substituted.

In get_timing we get all the classes of the applicable teachers on the particular day the class has to be substituted.

Finally, we find whether any teacher is available during the time the class has to be substituted. If there is , then it prints out the details otherwise NULL.

Now, we execute the procedure:

For example:

```
EXECUTE replaceTeacher 7008,N'2021-05-24';
```

89 %

Results | Messages

	DateOfClass	Teacher	CourseID	ClassDay	TimeFrom	TimeTo	Substitute_Teacher
1	2021-05-24	MIS014	MIS-AC011	Monday	08:00:00	11:00:00	MIS013

We are going to prove whether the substituted teacher selected is applicable or not.

First, we find the details of the group with GroupID=7008

```sql
select * from Groups where GroupID=7008
```

	GroupID	CourseID	EmployeeID	GroupNumber
1	7008	MIS-AC011	MIS014	2

Then we track down the teachers teaching the same course

```sql
select TeachersCourses.EmployeeID from Groups
inner join TeachersCourses on TeachersCourses.CourseID=Groups.CourseID where GroupID=7008
```

	EmployeeID
1	MIS013
2	MIS014

Of course, MIS014 won't be a valid option, so we look into MIS013, whether he/she is available

```sql
select Groups.GroupID,CourseID,EmployeeID,ClassDay,TimeFrom,TimeTo from Groups
inner join Classes on Classes.GroupID = Groups.GroupID where EmployeeID='MIS013'
```

	GroupID	CourseID	EmployeeID	ClassDay	TimeFrom	TimeTo
1	7007	MIS-AC011	MIS013	Monday	11:30:00	14:30:00

So, in his schedule he has only one class and thus he is available during the time period of 8:00-11:00 and thus he is applicable to be the substitute teacher on that day.

Thus, we can say that the procedure was correctly executed.