Documentation of Student Administration System (Better RAMMS)

Anthony Thanpoovong (501036051), Caleb Lam (501012133), Danny Lao (501032489)

Toronto Metropolitan University

CPS510- Database Systems 1

Dr. Soheila Bashardoust-Tajali

December 2, 2022

Table of Contents

Documentation of A1

Application Overview

The application is an online self-service administrative platform that allows students and staff in the university to view or update courses, grades, academic letters, personal and financial information. There are three different users for this application which help maintain a stable database. The first user is a student, they will be able to view their courses, grades, academic letter, and personal and financial information. The second user is an instructor who updates the end-of-the-semester GPA. Lastly, the system administrator will update the platform to keep track of all the students' information.

System Functions

The application consists of multiple functions/features that manage all essential aspects of the administration of a university.

Student Information

The application stores a student’s information including name, contact information, program, and financial account. The student account can view these details, but only an administrator can change them.

Course Enrolment

Administrators can add, edit, and delete different courses, each belonging to one department. Students can enroll and drop courses. Each course has sections managed by instructors which have a configurable capacity.

Academic Reports

Administrators can report grades to the system in order to add them to students' academic records. The records can be edited in case amendments need to be made. A student can view their records and grades through the Academic Reports page.

Financial Account

Administrators can apply fees/charges to students with a certain due date, and students can pay their financial accounts. Each day overdue increases the total charges by 0.1%, i.e. multiplying by 1.001 per day after the due date.

Tables

USER

user_id	username	password	role	program_id	enrollment_date

PROGRAM

program_id	name	department_id

DEPARTMENT

department_id	name	description

TERM

term_id	name	start_date	end_date

COURSE

course_code	name	hour_units	department_id

COURSE_SECTIONS

course_code	section_id	capacity	instructor_id	term_id

STUDENT_ENROLMENTS

student_id	course_code	term_id	section_id

GRADES

grade_id	course_code	student_id	credit_units	grade	gpa_achieved

ACADEMIC_LETTER

grade_id	date_taken	status

PERSONAL_INFORMATION

full_name	user_id	phone_ number	email_address	address	emergenc y_contact	emergency_ contact_pho ne_number

FINANCIAL_INFORMATION

item_id	item	student_id	term_id	amount_inc l_tax	balance	last_activit y_date

Table Relationships

The relationships between tables are as follows:

- Each USER record relates to one PROGRAM record.
- Each PROGRAM record relates to one DEPARTMENT record.
- Each COURSE record relates to one DEPARTMENT record.
- Each COURSE_SECTION record relates to one COURSE record, one TERM record, and one USER record.
- Each STUDENT_ENROLMENT record relates to one COURSE_SECTION record and one TERM record.
- Each GRADE record relates to a COURSE record and USER record.
- Each ACADEMIC_LETTER record relates to a GRADE record.
- Each PERSONAL_INFORMATION record relates to one USER record and FINANCIAL_INFORMATION record.
- Each FINANCIAL_INFORMATION relates to one USER record and one TERM record.

TA Recommendations

- Clarifying the primary keys of each table
- Making sure that the keys in each table make sense

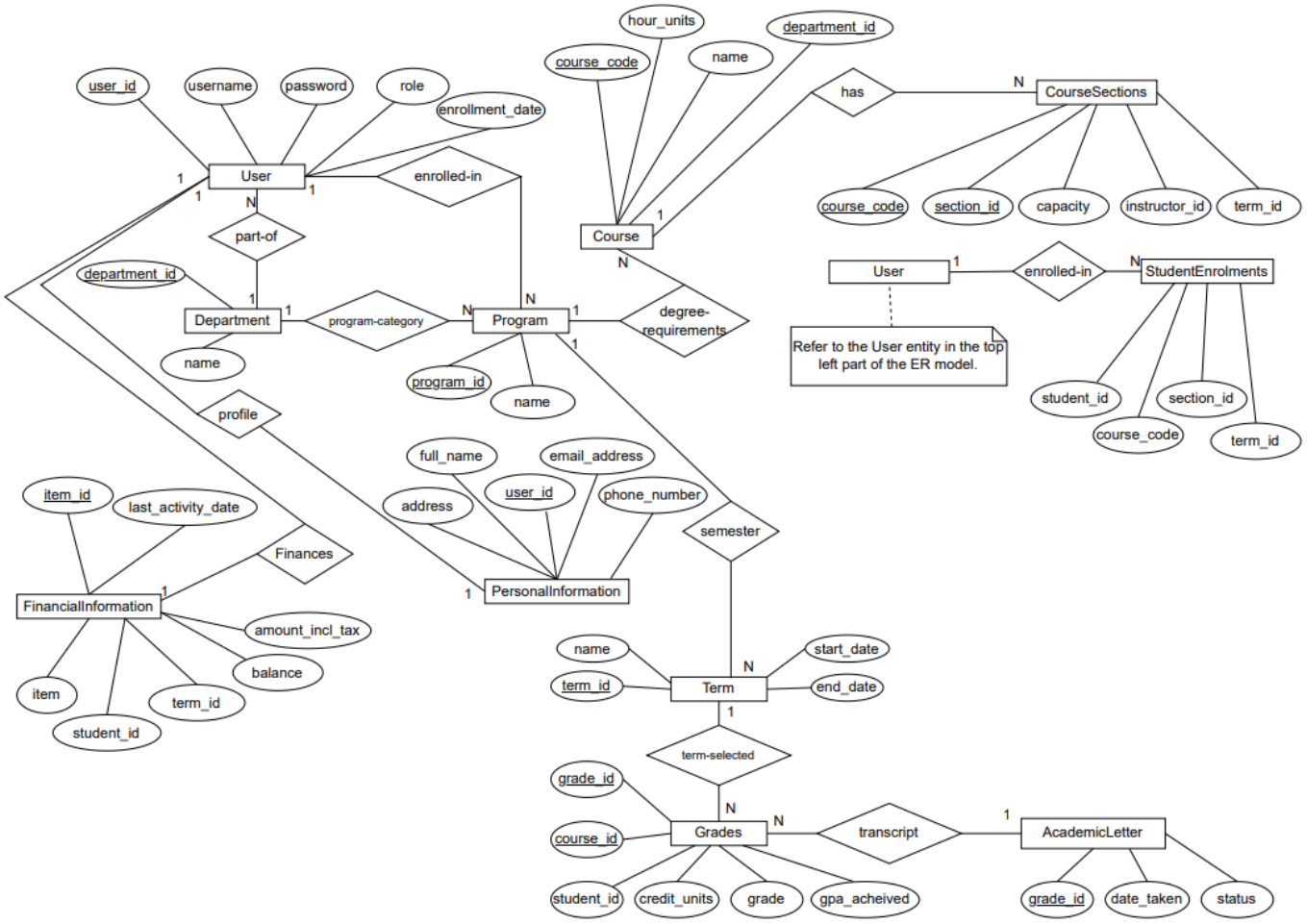
DEPARTMENTS

department_id	name

DEPARTMENT

department_id	name	description

Documentation of A2



TA Recommendations

- No TA recommendations for this assignment

Documentation of A3

Description of Tables

STUDENT_ENROLLMENT:
This table stores all the student enrollment information when administrators add them to the school's database. The TRANSACTION_ID column is an auto-generated number column that acts as the unique identifier (primary key). This allows a specification of that certain transaction. STUDENT_ID helps reference the student when enrolling them into the database. COURSE_CODE helps reference the specific course that the student is enrolling in. SECTION_ID references the specific section the student is placed in. TERM_ID references the specific term that the student is in.

```
CREATE TABLE "STUDENT_ENROLLMENTS" (
    TRANSACTION_ID NUMBER GENERATED ALWAYS AS IDENTITY,
    STUDENT_ID NUMBER REFERENCES USERS(USER_ID) NOT NULL,
    COURSE_CODE VARCHAR2(128) REFERENCES COURSES(COURSE_CODE) NOT NULL,
    SECTION_ID NUMBER REFERENCES COURSE_SECTIONS(SECTION_ID) NOT NULL,
    TERM_ID NUMBER REFERENCES TERMS(TERM_ID) NOT NULL,
    PRIMARY KEY(TRANSACTION_ID)
);
```

ACADEMIC_LETTER
This table stores all the student's academic records for each term. This includes the grades of the student per semester, the date it was taken and the courses the academic letter references too. RECORD_ID column is an auto-generated number column that acts as the unique identifier (primary key). It is used as an identifier for each individual entry. GRADE_ID references GRADES(GRADE_ID) as it refers to the grades of that student. DATE_TAKEN refers to the date stamp of when that course was taken by the student. STATUS refers to the current status of the student doing the course.

```
CREATE TABLE ACADEMIC_LETTER (
    RECORD_ID NUMBER GENERATED ALWAYS AS IDENTITY,
    GRADE_ID NUMBER REFERENCES GRADES(GRADE_ID),
    DATE_TAKEN DATE NOT NULL,
    STATUS VARCHAR2(32) NOT NULL,
    PRIMARY KEY(RECORD_ID)
);
```

FINANCIAL_INFORMATION:
This table stores the financial information of a user. ITEM_ID column is an auto-generated number column that acts as the unique identifier (primary key). This column helps refer to the specific item. LAST_ACTIVITY_DATE refers to the date when the item is displayed on their financial information table. AMOUNT_INCLUDING_TAX describes the cost of the item including tax. BALANCE refers to the balance of the student's account. TERM_ID references TERMS (TERM_ID) which refers to the term that the student is in. STUDENT_ID references USERS(USER_ID) which refers to the student's specific student ID. ITEM_NAME refers to the name of the item on the student's financial information balance.

```
CREATE TABLE "FINANCIAL_INFORMATION" (
    ITEM_ID NUMBER GENERATED ALWAYS AS IDENTITY,
    LAST_ACTIVITY_DATE DATE NOT NULL,
    AMOUNT_INCLUDING_TAX NUMBER NOT NULL,
    BALANCE NUMBER NOT NULL,
    TERM_ID NUMBER REFERENCES TERMS(TERM_ID) NOT NULL,
    STUDENT_ID NUMBER REFERENCES USERS(USER_ID) NOT NULL,
    ITEM_NAME VARCHAR (30) NOT NULL,
    PRIMARY KEY(ITEM_ID)
);
```

TERMS:
This table stores all existing academic terms in order to organize course sections and financial items. The TERM_ID column is an auto-generated number column that acts as the unique identifier (primary key) of a term. The TERM_NAME represents the term's name via the NVARCHAR type. The START_DATE and END_DATE (both being the DATE type) define the date interval of the term. There is a constraint on the START_DATE and END_DATE where START_DATE must be earlier than or the same as the END_DATE in order to define a valid range. None of the columns in this table are nullable.

```sql
CREATE TABLE TERMS (
    TERM_ID      NUMBER GENERATED ALWAYS AS IDENTITY,
    TERM_NAME    VARCHAR2(128) NOT NULL,
    START_DATE   DATE NOT NULL,
    END_DATE     DATE NOT NULL,
    PRIMARY KEY(TERM_ID),
    CONSTRAINT CK_Terms_DateRange CHECK(START_DATE <= END_DATE)
);
```

GRADES:
This table stores all submitted course grades for all students. The GRADE_ID column is an auto-generated number column that acts as the unique identifier (primary key). The purpose of this column is to be able to specifically refer to the ID of the grade. COURSE_SECTION references COURSE_SECTIONS(COURSE_SECTION) where which is used for updating the section of that course. STUDENT_ID references USERS(USER_ID) as it is used for specifying the student according to their student ID. CREDIT_UNITS helps demonstrate the number of credits for that course. GRADE refers to the letter grade for that course. GPA_ACHIEVED is the conversion from letter grade to numeric grade according to the Toronto Metropolitan GPA scale. CONSTRAINT (CK_GPA) CHECK helps keep the GPA_ACHIEVED between the boundaries.

```sql
CREATE TABLE "GRADES"(
    GRADE_ID       NUMBER GENERATED ALWAYS AS IDENTITY,
    COURSE_SECTION NUMBER REFERENCES COURSE_SECTIONS(COURSE_SECTION),
    STUDENT_ID     NUMBER REFERENCES USERS(USER_ID),
    CREDIT_UNITS   NUMBER,
    GRADE          CHAR(3),
    GPA_ACHEIVED   FLOAT NOT NULL,
    CONSTRAINT CK_GPA CHECK (GPA_ACHIEVED >= 0 AND GPA_ACHIEVED <= 4.33),
    PRIMARY KEY(GRADE_ID)
);
```

PROGRAMS:
This table contains records of all existing university programs. The PROGRAM_ID column holds the unique numeric identifier of a PROGRAM record. The PROGRAM_ID in each record is generated automatically. It is also the table's primary key. The DEPARTMENT_ID column is the numeric identifier of a DEPARTMENT record. There is a foreign key that maps one DEPARTMENT record to many PROGRAM records. The PROGRAM_NAME represents the name of the program. It holds up to 128 characters per record as a VARCHAR2 type and cannot be null.

```
CREATE TABLE PROGRAMS (
    DEPARTMENT_ID    NUMBER REFERENCES DEPARTMENTS(DEPARTMENT_ID),
    PROGRAM_NAME     VARCHAR2(128) NOT NULL,
    PROGRAM_ID       NUMBER GENERATED ALWAYS AS IDENTITY,
    PRIMARY KEY(PROGRAM_ID)
);

```

COURSE_SECTIONS:
The COURSE_SECTIONS table contains all the available sections for courses. (ie CPS510 could have 10 sections) The COURSE_CODE in COURSE_SECTIONS is related to the COURSE_CODE in the COURSES table. The instructors' ID will be listed with each section (ie a CPS510 section could contain Prof Abhari or Prof Tajali's corresponding ID) The CAPACITY represents the number of seats in a particular section.

```
CREATE TABLE COURSE_SECTIONS (
    SECTION_ID       NUMBER GENERATED ALWAYS AS IDENTITY,
    COURSE_CODE      VARCHAR2(128) REFERENCES COURSES(COURSECODE),
    INSTRUCTOR_ID    NUMBER REFERENCES USERS(USER_ID),
    TERM_ID          NUMBER REFERENCES TERMS(TERM_ID),
    CAPACITY         NUMBER NOT NULL,
    PRIMARY KEY(SECTION_ID)
)
```

COURSES:
The COURSES table is a table that stores information about a particular course including its code, hours required, course name and what department the course belongs to. COURSE_CODE is a user-generated sequence of characters (ie. "CPS510") and acts as the unique identifier of a course (primary key). HOUR_UNITS defined the number of hours to achieve the credit for a course. (ie 40 hours) COURSE_NAME is the name of the course that corresponds to the COURSE_CODE. (ie CPS510: Database Systems I) DEPARTMENT_ID is related to the

DEPARTMENT_ID in the DEPARTMENTS table.

```
CREATE TABLE COURSE (
    COURSE_CODE      VARCHAR2(128) PRIMARY KEY,
    HOUR_UNITS       NUMBER NOT NULL,
    COURSE_NAME      VARCHAR(128) NOT NULL,
    DEPARTMENT_ID    NUMBER NOT NULL REFERENCES DEPARTMENTS(DEPARTMENT_ID)
)
```

USERS:
This table stores all user information, including account login details. The USER_ID column is an auto-generated number column that acts as the unique identifier (primary key). This column helps identify the user by their user ID. USERNAME refers to the usernames of the students. The PASSWORD column stores all the password hashes of the users. (As a best practice, it is never a good idea to store passwords in plain text.) ROLE helps choose the role of that user. ENROLLMENT_DATE refers to when the user was enrolled into the system. The PHONE_NUMBER column stores the phone numbers of all the users. The EMAIL_ADDRESSES column stores the email addresses of all the users. The FULL_NAME column stores the full names of all the users. The HOME_ADDRESS stores the home addresses of all the users.

```
CREATE TABLE "USERS" (
    USER_ID NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 100000000),
    USERNAME VARCHAR2 (25),
    PASSWORD VARCHAR2 (32),
    ROLE NUMBER NOT NULL,
    ENROLLMENT_DATE VARCHAR2 (12),
    PHONE_NUMBER NUMBER (13) UNIQUE,
    EMAIL_ADDRESS VARCHAR2 (30) UNIQUE,
    FULL_NAME VARCHAR2 (30) NOT NULL,
    HOME_ADDRESS VARCHAR2 (30),
    PRIMARY KEY(USER_ID)
);
```

DEPARTMENTS:
DEPARTMENTS is a table that stores the ID of each department in order to organize the IDs with the corresponding department name. (i.e. a random number is assigned to the Faculty of Engineering) DEPARTMENT_ID is a number and acts as the primary key and key identifier of a department.

```
CREATE TABLE DEPARTMENTS (
    DEPARTMENT_ID    NUMBER GENERATED ALWAYS AS IDENTITY,
    DEPARTMENT_NAME  VARCHAR2(128) NOT NULL,
    PRIMARY KEY(DEPARTMENT_ID)
);
```

TA Recommendations

- In the USERS table, PHONE_NUMBER should not be UNIQUE since a student can use their parent's phone number

Documentation of A4a

1. List all names of users

```
1    SELECT FULL_NAME
2    FROM USERS;
```

This query demonstrates a basic search of all users' full names.

2. List all names of students

```
1    SELECT FULL_NAME
2    FROM USERS
3    WHERE ROLE = 'student';
```

This query expands from query 1. It only shows the full names of students using the WHERE clause.

3. List all students with their overall GPA achieved (pg 41 of Topic5-SQL lecture)

```
1   SELECT GRADES.STUDENT_ID, 'overall GPA is: ', AVG(GRADES.GPA_ACHIEVED)
2   FROM GRADES
3   GROUP BY GRADES.STUDENT_ID
```

This query lists the overall GPA of each student. This is done by taking the grades of each student's courses and calculating their average using the AVG aggregate function. We group the GRADES records by student ID in order to calculate the correct averages.

4. List all students with overdue balances

```
1   SELECT STUDENT_ID, 'Outstanding balance is: ', BALANCE
2   FROM FINANCIAL_INFORMATION
3   WHERE BALANCE > 0;
```

This query lists the STUDENT_ID and BALANCE from the FINANCIAL_INFORMATION table. This is only displayed when BALANCE > 0.

5. Find the average GPA of all students (pg 61 of Topic5-SQL lecture)

```
1   SELECT 'Average GPA is ', AVG(GRADES.GPA_ACHIEVED)
2   FROM GRADES;
```

This query lists the average GPA_ACHIEVED from the GRADES table while using the AVG aggregate function.

6. List the number of students in a course (ie CPS510) (pg 64 of Topic5-SQL lecture)

```
SELECT COURSE_CODE, COUNT(STUDENT_ID) as Number_Enrolled
FROM STUDENT_ENROLLMENTS
GROUP BY COURSE_CODE;
```

This query lists the COURSE_CODE, and counts the number of STUDET_ID labelled as Number_Enrolled. These values are from the STUDENT_ENROLLMENTS table and are grouped by COURSE_CODE.

7. List Student # in a specific section of a course

```
SELECT DISTINCT STUDENT_ENROLLMENTS.STUDENT_ID, USERS.FULL_NAME, COURSE_CODE, SECTION_ID
FROM STUDENT_ENROLLMENTS
JOIN USERS ON USERS.USER_ID = STUDENT_ENROLLMENTS.STUDENT_ID
WHERE COURSE_CODE = 'COE318'
    AND SECTION_ID = '1'
ORDER BY STUDENT_ENROLLMENTS.STUDENT_ID ASC;
```

The picture above depicts our query that displays a list of student numbers in a specific section of a course. We first select the values that we want to display and made them distinct. The tables that were used are STUDENT_ENROLLMENTS and USERS. In this case, we only wanted the specific course code “COE318” as their section_ID “1”. They are then ordered by their student IDs.

8. List professors in a department

```
SELECT DISTINCT u.FULL_NAME, d.DEPARTMENT_NAME
FROM USERS u
JOIN DEPARTMENTS d ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
WHERE ROLE = 'instructor'
GROUP BY d.DEPARTMENT_NAME, u.FULL_NAME, u.DEPARTMENT_ID ;

```

The picture above depicts our query that displays a list of professors in a department. We first select the values that we want to display, such as FULL_NAME and DEPARTMENT_NAME. The tables that were used are USERS and DEPARTMENTS. In this case, we only wanted “instructors” as their role. They are then grouped together as shown above.

9. List student ID grades in a term

```
SELECT s.STUDENT_ID, t.TERM_NAME, AVG(g.GPA_ACHIEVED) as "Average GPA Achieved"
FROM STUDENT_ENROLLMENTS s
JOIN TERMS t ON t.TERM_ID = s.TERM_ID
JOIN GRADES g ON s.STUDENT_ID = g.STUDENT_ID
JOIN ACADEMIC_LETTER l ON l.GRADE_ID = g.GRADE_ID
GROUP BY t.TERM_NAME, s.STUDENT_ID, g.GPA_ACHIEVED
```

The picture above depicts our query that displays a student's ID, with their grade in that respective term. We first select the values that we want to display and find the AVG of the GPA in that term. The tables that were used are STUDENT_ENROLLMENTS, TERMS, GRADES and ACADEMIC_LETTER. They are then grouped together as shown above.

10. List the instructors and course code in a course section

```
SELECT c.INSTRUCTOR_ID, c.COURSE_CODE, c.SECTION_ID
FROM COURSE_SECTIONS c
```

This query display the INSTRUCTOR ID, COURSE_CODE, and SECTION_ID from the COURSE_SECTIONS table.

TA Recommendations

- No recommendations for this assignment

Documentation of A4b

Complex Queries

1. AllPeopleInASpecificDepartment

```
SELECT DISTINCT us.FULL_NAME, de.DEPARTMENT_NAME
FROM USERS us
JOIN DEPARTMENTS de ON us.DEPARTMENT_ID = de.DEPARTMENT_ID
WHERE de.DEPARTMENT_ID = '1'
GROUP BY de.DEPARTMENT_NAME, us.FULL_NAME, us.DEPARTMENT_ID ;
```

This query lists all the people that belong to a specific department. For example, this query shows all the people listed under the Engineering department.

2. StudentsTaking2SpecificCoursesTogether

```
SELECT stu.USER_ID, stu.FULL_NAME
FROM USERS stu
WHERE EXISTS
(SELECT e1.STUDENT_ID
FROM COURSE_SECTIONS c1, COURSE_SECTIONS c2, STUDENT_ENROLLMENTS e1, STUDENT_ENROLLMENTS e2
WHERE c1.INSTRUCTOR_ID = 100000064
      AND c1.COURSE_CODE ='BLG143'
      AND e1.STUDENT_ID = e2.STUDENT_ID
      AND e1.COURSE_CODE = 'BLG143'
      AND c2.INSTRUCTOR_ID = 100000064
      AND c1.TERM_ID = c2.TERM_ID
      AND c2.COURSE_CODE = 'CPS109'
      AND c1.INSTRUCTOR_ID = c2.INSTRUCTOR_ID
      AND e2.COURSE_CODE = 'CPS109'
      AND stu.USER_ID = e1.STUDENT_ID);
```

This query lists the student ids of students taking 2 courses from the same professor during the same term. A lot of comparisons are implemented to check that the instructor/course information matches together.

3. List Student ID grades in a given term

```
SELECT s.STUDENT_ID, t.TERM_NAME, AVG(g.GPA_ACHIEVED) as "Average GPA Achieved"
FROM STUDENT_ENROLLMENTS s
JOIN TERMS t ON t.TERM_ID = s.TERM_ID
JOIN GRADES g ON s.STUDENT_ID = g.STUDENT_ID
JOIN ACADEMIC_LETTER l ON l.GRADE_ID = g.GRADE_ID
GROUP BY t.TERM_NAME, s.STUDENT_ID, g.GPA_ACHIEVED
```

The picture above depicts our query that displays a student's ID, with their grade in that

respective term. We first select the values that we want to display and find the AVG of the GPA in that term. The tables that were used are STUDENT_ENROLLMENTS, TERMS, GRADES and ACADEMIC_LETTER. They are then grouped together as shown above.

4. List Student # in a specific section of a course

```
SELECT DISTINCT STUDENT_ENROLLMENTS.STUDENT_ID, USERS.FULL_NAME, COURSE_CODE, SECTION_ID
FROM STUDENT_ENROLLMENTS
JOIN USERS ON USERS.USER_ID = STUDENT_ENROLLMENTS.STUDENT_ID
WHERE COURSE_CODE = 'COE318'
    AND SECTION_ID = '1'
ORDER BY STUDENT_ENROLLMENTS.STUDENT_ID ASC;
```

The picture above depicts our query that displays a list of student numbers in a specific section of a course. We first select the values that we want to display and made them distinct. The tables that were used are STUDENT_ENROLLMENTS and USERS. In this case, we only wanted the specific course code “COE318” as their section_ID “1”. They are then ordered by their student IDs.

5. List professors in a department

```
SELECT DISTINCT u.FULL_NAME, d.DEPARTMENT_NAME
FROM USERS u
JOIN DEPARTMENTS d ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
WHERE ROLE = 'instructor'
GROUP BY d.DEPARTMENT_NAME, u.FULL_NAME, u.DEPARTMENT_ID ;

```

The picture above depicts our query which displays a list of professors in a department. We first select the values that we want to display, such as FULL_NAME and DEPARTMENT_NAME. The tables that were used are USERS and DEPARTMENTS. In this case, we only wanted “instructors” as their role. They are then grouped together as shown above.

Views

1. Lists the number of people in a department

```
CREATE OR REPLACE VIEW NUMBER_IN_DEPARTMENT AS
(
    SELECT d.DEPARTMENT_NAME, COUNT(u.USER_ID) AS NUMBER_OF_PEOPLE
    FROM DEPARTMENTS d
    JOIN USERS u ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
    GROUP BY d.DEPARTMENT_NAME, u.USER_ID

)
ORDER BY d.DEPARTMENT_NAME ASC, u.USER_ID ASC
WITH READ ONLY;
```

This view demonstrates the number of people in a department using the COUNT function. This uses the table DEPARTMENTS and the table USERS. It's then sorted by ascending in DEPARTMENT_NAME and USER_ID. It's also READ ONLY, as a table view.

2. Lists the programs that are offered

```
CREATE OR REPLACE VIEW PROGRAM_OFFERINGS AS
    (
        SELECT p.PROGRAM_ID, p.PROGRAM_NAME, d.DEPARTMENT_NAME
        FROM PROGRAMS p
        JOIN DEPARTMENTS d ON p.DEPARTMENT_ID = d.DEPARTMENT_ID
    )
ORDER BY p.PROGRAM_NAME ASC;
WITH READ ONLY
```

This view demonstrates the programs that are offered to the viewer. It selects PROGRAM_ID, PROGRAM_NAME, and DEPARTMENT_NAME from both the PROGRAMS table and the DEPARTMENTS table. It is also READ ONLY, as a table view.

3. Lists the Course Sections with available spots

```
CREATE OR REPLACE VIEW COURSE_SECTIONS_LIST AS
(
    SELECT cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME AS TERM, u.FULL_NAME AS INSTRUCTOR, COUNT(se.STUDENT_ID) AS TOTAL_ENROLLED, cs.CAPACITY
    FROM COURSE_SECTIONS cs
    JOIN COURSES c ON c.COURSE_CODE = cs.COURSE_CODE
    JOIN USERS u ON u.USER_ID = cs.INSTRUCTOR_ID
    JOIN TERMS t ON t.TERM_ID = cs.TERM_ID
    LEFT JOIN STUDENT_ENROLLMENTS se ON se.COURSE_CODE = cs.COURSE_CODE AND se.SECTION_ID = cs.SECTION_ID
    GROUP BY cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME, u.FULL_NAME, cs.CAPACITY
    HAVING COUNT(se.STUDENT_ID) < cs.CAPACITY

)
ORDER BY cs.COURSE_CODE ASC, cs.SECTION_ID ASC
WITH READ ONLY;
```

This view demonstrates the Course Sections using the COURSE_CODE, SECTION_ID, COURSE_NAME, TERM_NAME, FULL_NAME and COUNTs the STUDENT_ID with their capacity. It uses the COURSE_SECTIONS, COURSES, USERS, TERMS, and STUDENT_ENROLLMENTS table, and the use of GROUP BY and HAVING to count the number of spots taken in each section and eliminate sections that are full. It is then labelled READ ONLY.

TA Recommendations

- No recommendations for this assignment

Documentation of A5

menu.sh

```
#!/bin/sh
MainMenu()
{
    while [ "$CHOICE" != "START" ]
    do
        clear
        echo
"=============================================================="
        echo "| Oracle All Inclusive Tool
|"
        echo "| Main Menu - Select Desired Operation(s):
|"
        echo "| <CTRL-Z Anytime to Enter Interactive CMD Prompt>
|"
        echo
"--------------------------------------------------------------"
        echo " $IS_SELECTEDM M) View Manual"
        echo " "
        echo " $IS_SELECTED1 1) Drop Tables"
        echo " $IS_SELECTED2 2) Create Tables"
        echo " $IS_SELECTED3 3) Populate Tables"
        echo " $IS_SELECTED4 4) Query Tables"
        echo " "
        echo " $IS_SELECTEDX X) Force/Stop/Kill Oracle DB"
        echo " "
        echo " $IS_SELECTEDE E) End/Exit"
        echo "Choose: "
        read CHOICE
        if [ "$CHOICE" == "0" ]
        then
            echo "Nothing Here"
        elif [ "$CHOICE" == "1" ]
        then
            bash drop_tables.sh
            bash drop_views.sh
            Pause
        elif [ "$CHOICE" == "2" ]
        then
```

```
            bash create_tables.sh
            bash create_views.sh
            Pause
        elif [ "$CHOICE" == "3" ]
        then
            bash populate_tables.sh
            Pause
        elif [ "$CHOICE" == "4" ]
        then
            bash queries.sh
            Pause
        elif [ "$CHOICE" == "E" ]
        then
            exit
        fi
        read -p "Press enter to continue..."
    done
}
#--COMMENTS BLOCK--
# Main Program
#--COMMENTS BLOCK--
ProgramStart()
{
    StartMessage
    while [ 1 ]
    do
        MainMenu
    done
}
ProgramStart
```

create_tables.sh

```
#!/bin/sh
echo "Creating tables..."
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
CREATE TABLE "DEPARTMENTS" (
  "DEPARTMENT_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1
MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
```

```
  "DEPARTMENT_NAME" VARCHAR2(128) NOT NULL ENABLE,
  PRIMARY KEY ("DEPARTMENT_ID") USING INDEX ENABLE
);
CREATE TABLE "USERS" (
  "USER_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 100000000,
  "USERNAME" VARCHAR2(25) NOT NULL ENABLE,
  "PASSWORD" VARCHAR2(64) NOT NULL ENABLE,
  "ROLE" VARCHAR2(25) NOT NULL ENABLE,
  "ENROLLMENT_DATE" DATE NOT NULL ENABLE,
  "PHONE_NUMBER" NUMBER(13, 0) NOT NULL ENABLE,
  "EMAIL_ADDRESS" VARCHAR2(254) NOT NULL ENABLE,
  "FULL_NAME" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_STREET_ADDRESS" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_CITY" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_PROVINCE" CHAR(2) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER,
  PRIMARY KEY ("USER_ID") USING INDEX ENABLE,
  UNIQUE ("PHONE_NUMBER") USING INDEX ENABLE,
  UNIQUE ("EMAIL_ADDRESS") USING INDEX ENABLE,
  CONSTRAINT "USER_ROLES" CHECK ( "ROLE" IN ('student', 'instructor',
'admin')) ENABLE
);
CREATE TABLE "COURSES" (
  "COURSE_CODE" VARCHAR2(128),
  "HOUR_UNITS" NUMBER NOT NULL ENABLE,
  "COURSE_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("COURSE_CODE") USING INDEX ENABLE
);
CREATE TABLE "PROGRAMS" (
  "DEPARTMENT_ID" NUMBER,
  "PROGRAM_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "PROGRAM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  PRIMARY KEY ("PROGRAM_ID") USING INDEX ENABLE
);
CREATE TABLE "TERMS" (
  "TERM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
```

```
  "TERM_NAME" VARCHAR2(256) NOT NULL ENABLE,
  "START_DATE" DATE NOT NULL ENABLE,
  "END_DATE" DATE NOT NULL ENABLE,
  CONSTRAINT "CK_TERMS_DATERANGE" CHECK ("START_DATE"<="END_DATE") ENABLE,
  PRIMARY KEY ("TERM_ID") USING INDEX ENABLE
);
CREATE TABLE "FINANCIAL_INFORMATION" (
  "ITEM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "LAST_ACTIVITY_DATE" DATE NOT NULL ENABLE,
  "AMOUNT_INCLUDING_TAX" NUMBER NOT NULL ENABLE,
  "BALANCE" NUMBER NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "ITEM_NAME" VARCHAR2(30) NOT NULL ENABLE,
  PRIMARY KEY ("ITEM_ID") USING INDEX ENABLE
);
CREATE TABLE "COURSE_SECTIONS" (
  "COURSE_CODE" VARCHAR2(128),
  "INSTRUCTOR_ID" NUMBER,
  "TERM_ID" NUMBER,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  "CAPACITY" NUMBER NOT NULL ENABLE,
  CONSTRAINT "COURSE_SECTIONS_KEYS" PRIMARY KEY ("COURSE_CODE", "TERM_ID",
"SECTION_ID") USING INDEX ENABLE,
  CONSTRAINT "COURSE_SECTIONS_CODE_NOT_NULL" CHECK ( "COURSE_CODE" IS NOT
NULL) ENABLE
);
CREATE TABLE "GRADES" (
  "GRADE_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  PRIMARY KEY ("GRADE_ID") USING INDEX ENABLE
);
CREATE TABLE "STUDENT_ENROLLMENTS" (
```

```
  "TRANSACTION_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1
MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "COURSE_CODE" VARCHAR2(128) NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("TRANSACTION_ID") USING INDEX ENABLE
);
CREATE TABLE "ACADEMIC_LETTER" (
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" VARCHAR2(25) NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID") USING INDEX ENABLE
);
ALTER TABLE "COURSES" ADD FOREIGN KEY ("DEPARTMENT_ID") REFERENCES
"DEPARTMENTS" ("DEPARTMENT_ID") ENABLE;
ALTER TABLE "PROGRAMS" ADD FOREIGN KEY ("DEPARTMENT_ID") REFERENCES
"DEPARTMENTS" ("DEPARTMENT_ID") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES"
("COURSE_CODE") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS"
("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES
"COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("INSTRUCTOR_ID") REFERENCES
"USERS" ("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("TERM_ID") REFERENCES
"TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("TERM_ID") REFERENCES
"TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("STUDENT_ID")
REFERENCES "USERS" ("USER_ID") ENABLE;
ALTER TABLE "STUDENT_ENROLLMENTS" ADD CONSTRAINT
"STUDENT_ENROLLMENTS_STUDENT" FOREIGN KEY ("STUDENT_ID") REFERENCES
"USERS" ("USER_ID") ENABLE;
ALTER TABLE "STUDENT_ENROLLMENTS" ADD CONSTRAINT
"STUDENT_ENROLLMENTS_REFS" FOREIGN KEY ("COURSE_CODE", "TERM_ID",
```

```
"SECTION_ID") REFERENCES "COURSE_SECTIONS" ("COURSE_CODE", "TERM_ID",
"SECTION_ID") ON DELETE CASCADE ENABLE;
ALTER TABLE "USERS" ADD CONSTRAINT "USERS_DEPARTMENT" FOREIGN KEY
("DEPARTMENT_ID") REFERENCES "DEPARTMENTS" ("DEPARTMENT_ID") ON DELETE SET
NULL ENABLE;
exit;
EOF
echo "Done!"
```

create_views.sh

```
#!/bin/sh
echo "Creating views..."
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
PROMPT Creating COURSE_SECTIONS_LIST...;
CREATE OR REPLACE VIEW COURSE_SECTIONS_LIST AS
(
    SELECT cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME AS
TERM, u.FULL_NAME AS INSTRUCTOR, COUNT(se.STUDENT_ID) AS TOTAL_ENROLLED,
cs.CAPACITY
    FROM COURSE_SECTIONS cs
    JOIN COURSES c ON c.COURSE_CODE = cs.COURSE_CODE
    JOIN USERS u ON u.USER_ID = cs.INSTRUCTOR_ID
    JOIN TERMS t ON t.TERM_ID = cs.TERM_ID
    LEFT JOIN STUDENT_ENROLLMENTS se ON se.COURSE_CODE = cs.COURSE_CODE
AND se.SECTION_ID = cs.SECTION_ID
    GROUP BY cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME,
u.FULL_NAME, cs.CAPACITY
    HAVING COUNT(se.STUDENT_ID) < cs.CAPACITY
)
ORDER BY cs.COURSE_CODE ASC, cs.SECTION_ID ASC
WITH READ ONLY;
PROMPT Creating NUMBER_IN_DEPARTMENT...;
CREATE OR REPLACE VIEW NUMBER_IN_DEPARTMENT AS
(
    SELECT d.DEPARTMENT_NAME, COUNT(u.USER_ID) AS NUMBER_OF_PEOPLE
    FROM DEPARTMENTS d
    JOIN USERS u ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
    GROUP BY d.DEPARTMENT_NAME, u.USER_ID
```

```
)
ORDER BY d.DEPARTMENT_NAME ASC, u.USER_ID ASC
WITH READ ONLY;
PROMPT Creating PROGRAM_OFFERINGS...;
CREATE OR REPLACE VIEW PROGRAM_OFFERINGS AS
    (
        SELECT p.PROGRAM_ID, p.PROGRAM_NAME, d.DEPARTMENT_NAME
        FROM PROGRAMS p
        JOIN DEPARTMENTS d ON p.DEPARTMENT_ID = d.DEPARTMENT_ID
    )
ORDER BY p.PROGRAM_NAME ASC
WITH READ ONLY;
exit;
EOF
echo "Done!"
```

drop_tables.sh

```
#!/bin/sh
echo "Dropping tables..."
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
DROP TABLE DEPARTMENTS CASCADE CONSTRAINTS;
DROP TABLE USERS CASCADE CONSTRAINTS;
DROP TABLE COURSES CASCADE CONSTRAINTS;
DROP TABLE PROGRAMS CASCADE CONSTRAINTS;
DROP TABLE TERMS CASCADE CONSTRAINTS;
DROP TABLE FINANCIAL_INFORMATION CASCADE CONSTRAINTS;
DROP TABLE COURSE_SECTIONS CASCADE CONSTRAINTS;
DROP TABLE GRADES CASCADE CONSTRAINTS;
DROP TABLE STUDENT_ENROLLMENTS CASCADE CONSTRAINTS;
DROP TABLE ACADEMIC_LETTER CASCADE CONSTRAINTS;
exit;
EOF
echo "Done!"
```

drop_views.sh

```
#!/bin/sh
echo "Dropping views..."
```

```
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
DROP VIEW COURSE_SECTIONS_LIST;
DROP VIEW NUMBER_IN_DEPARTMENT;
DROP VIEW PROGRAM_OFFERINGS;
exit;
EOF
echo "Done!"
```

populate_tables.sh

```
#!/bin/sh
echo "Populating tables..."
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
SET DEFINE OFF;
ALTER SESSION SET NLS_DATE_FORMAT = 'MM/DD/YYYY';
PROMPT Populating DEPARTMENTS...;
INSERT INTO DEPARTMENTS
VALUES(1,'Faculty of Engineering & Architectural Science');
INSERT INTO DEPARTMENTS
VALUES(21,'Faculty of Arts');
INSERT INTO DEPARTMENTS
VALUES(2,'Faculty of Law');
INSERT INTO DEPARTMENTS
VALUES(22,'Faculty of Science');
INSERT INTO DEPARTMENTS
VALUES(3,'Faculty of Community Services');

PROMPT Populating USERS...;
INSERT INTO USERS
VALUES(100000060,'alex.joel','alex.joel.11','student','08/07/2021',1234567
890,'a.joel@example.com', 'Alex Joel', '8 Example Road', 'Markham', 'ON',
22);
INSERT INTO USERS
VALUES(100000081,'ab.cd','ab.cd','admin','09/09/2009',1238765432,'ab.cd@ex
ample.com', 'Ab Cade', '11 mango Avenue', 'London', 'ON', 21);
INSERT INTO USERS
```

```
VALUES(100000040,'jason.green','jason.green','instructor','08/05/2015',647
1201598,'jason.green@abc.ca', 'Jason Green', '11 Random Avenue',
'Toronto', 'ON', 1);
INSERT INTO USERS
VALUES(100000064,'ker.hen','kermitTheFrog','instructor','01/01/2020',41649
11444,'kermit.henson@gmail.com', 'Kermit Henson', '15 Sesame Street', 'New
York City', 'NY', 22);
INSERT INTO USERS
VALUES(100000004,'quan.ding','3RzPb@@W8lry','student','11/19/2022',4169671
111,'quan.ding@gmail.com', 'Quandale Dingle', '25 Dale Street', 'Toronto',
'ON', 1);

PROMPT Populating PROGRAMS...;
INSERT INTO PROGRAMS
VALUES(2,'Ethics, Society and Law',2);
INSERT INTO PROGRAMS
VALUES(1,'Computer Engineering',1);

PROMPT Populating TERMS...;
INSERT INTO TERMS
VALUES(1,'Fall 2022','09/12/2022','12/16/2022');

PROMPT Populating FINANCIAL_INFORMATION...;
INSERT INTO FINANCIAL_INFORMATION
VALUES(1,'09/05/2022',5500,5500,1,100000004,
'TuitionFall');

PROMPT Populating COURSES...;
INSERT INTO COURSES
VALUES('COE328',1,'Digital Systems',1);
INSERT INTO COURSES
VALUES('CPS510',1,'Database Systems I',1);
INSERT INTO COURSES
VALUES('BLG144',50,'Biology II',22);
INSERT INTO COURSES
VALUES('MEC511',1,'Thermodynamics and Fluids',1);
INSERT INTO COURSES
VALUES('BLG143',50,'Biology I',22);
INSERT INTO COURSES
VALUES('CHY103',40,'General Chemistry I',22);
```

```
INSERT INTO COURSES
VALUES('CPS109',60,'Computer Science I',22);
INSERT INTO COURSES
VALUES('COE318',1,'Software Systems',1);

PROMPT Populating COURSE_SECTIONS...;
INSERT INTO COURSE_SECTIONS
VALUES ('COE318', 100000040, 1, 1, 60);
INSERT INTO COURSE_SECTIONS
VALUES ('BLG143', 100000064, 1, 1, 25);
INSERT INTO COURSE_SECTIONS
VALUES ('CPS109', 100000064, 1, 1, 30);
INSERT INTO COURSE_SECTIONS
VALUES ('CPS109', 100000040, 1, 2, 30);
INSERT INTO COURSE_SECTIONS
VALUES ('CHY103', 100000040, 1, 1, 30);
INSERT INTO COURSE_SECTIONS
VALUES ('BLG143', 100000064, 1, 2, 25);

PROMPT Populating STUDENT_ENROLLMENTS...;
INSERT INTO STUDENT_ENROLLMENTS (STUDENT_ID, COURSE_CODE, TERM_ID,
SECTION_ID)
VALUES (100000060,'CPS109',1,1);
INSERT INTO STUDENT_ENROLLMENTS (STUDENT_ID, COURSE_CODE, TERM_ID,
SECTION_ID)
VALUES (100000060,'CHY103',1,1);
INSERT INTO STUDENT_ENROLLMENTS (STUDENT_ID, COURSE_CODE, TERM_ID,
SECTION_ID)
VALUES (100000004,'COE318',1,1);
INSERT INTO STUDENT_ENROLLMENTS (STUDENT_ID, COURSE_CODE, TERM_ID,
SECTION_ID)
VALUES (100000060,'BLG143',1,2);

PROMPT Populating GRADES...;
INSERT INTO GRADES
VALUES(5 ,'BLG143', 100000060, 1, 'C', 2);
INSERT INTO GRADES
VALUES(21 ,'COE328', 100000004, 1, 'A', 4);
INSERT INTO GRADES
VALUES(41 ,'COE318', 100000004, 1, 'C', 2);
```

```
PROMPT Populating ACADEMIC_LETTER...;
INSERT INTO ACADEMIC_LETTER
VALUES(1, 21, '09/13/2021', 'COMPLETED');

exit;
EOF
echo "Done!"
```

queries.sh

```
#!/bin/sh
echo "Querying tables..."
sqlplus64
"[username]/[password]@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c
.scs.ryerson.ca)(Port=1521))(CONNECT_DATA=(SID=orcl12c)))" <<EOF
PROMPT Showing program offerings...;
SELECT * FROM PROGRAM_OFFERINGS;

PROMPT Showing number of people by department...;
SELECT * FROM NUMBER_IN_DEPARTMENT;

PROMPT Showing students with outstanding balances...;
SELECT STUDENT_ID, 'Outstanding balance is: ', BALANCE
    FROM FINANCIAL_INFORMATION
    WHERE BALANCE > 0;

PROMPT Showing open course sections...;
SELECT * FROM COURSE_SECTIONS_LIST;

PROMPT Showing total number of students in each course...;
SELECT COURSE_CODE, COUNT(STUDENT_ID) as Number_Enrolled
    FROM STUDENT_ENROLLMENTS
    GROUP BY COURSE_CODE;

PROMPT Showing students from a specific course section (COE318, Section
1)...;
SELECT DISTINCT STUDENT_ENROLLMENTS.STUDENT_ID, USERS.FULL_NAME,
COURSE_CODE, SECTION_ID
    FROM STUDENT_ENROLLMENTS
    JOIN USERS ON USERS.USER_ID = STUDENT_ENROLLMENTS.STUDENT_ID
```

```
    WHERE COURSE_CODE = 'COE318'
        AND SECTION_ID = '1';
ORDER BY STUDENT_ENROLLMENTS.STUDENT_ID ASC;

PROMPT Showing average GPA of each student...;
SELECT GRADES.STUDENT_ID, 'overall GPA is: ', AVG(GRADES.GPA_ACHIEVED)
    FROM GRADES
    GROUP BY GRADES.STUDENT_ID;

PROMPT Showing term GPAs per student...;
SELECT s.STUDENT_ID, t.TERM_NAME, AVG(g.GPA_ACHIEVED) as "Average GPA
Achieved"
    FROM STUDENT_ENROLLMENTS s
    JOIN TERMS t ON t.TERM_ID = s.TERM_ID
    JOIN GRADES g ON s.STUDENT_ID = g.STUDENT_ID
    JOIN ACADEMIC_LETTER l ON l.GRADE_ID = g.GRADE_ID
    GROUP BY t.TERM_NAME, s.STUDENT_ID, g.GPA_ACHIEVED;

PROMPT Showing offered programs...;
SELECT s.STUDENT_ID, t.TERM_NAME, AVG(g.GPA_ACHIEVED) as "Average GPA
Achieved"
    SELECT p.PROGRAM_ID, p.PROGRAM_NAME, d.DEPARTMENT_NAME
    FROM PROGRAMS p
    JOIN DEPARTMENTS d ON p.DEPARTMENT_ID = d.DEPARTMENT_ID

PROMPT Showing number in department...;
SELECT d.DEPARTMENT_NAME, COUNT(u.USER_ID) AS NUMBER_OF_PEOPLE
    FROM DEPARTMENTS d
    JOIN USERS u ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
    GROUP BY d.DEPARTMENT_NAME
ORDER BY d.DEPARTMENT_NAME ASC, u.USER_ID ASC

PROMPT Showing students that take the same course together ...;
SELECT stu.USER_ID, stu.FULL_NAME
FROM USERS stu
WHERE EXISTS
(SELECT e1.STUDENT_ID
FROM COURSE_SECTIONS c1, COURSE_SECTIONS c2, STUDENT_ENROLLMENTS e1,
STUDENT_ENROLLMENTS e2
WHERE c1.INSTRUCTOR_ID = 100000064
```

```
      AND c1.COURSE_CODE ='BLG143'
      AND e1.STUDENT_ID = e2.STUDENT_ID
      AND e1.COURSE_CODE = 'BLG143'
      AND c2.INSTRUCTOR_ID = 100000064
      AND c1.TERM_ID = c2.TERM_ID
      AND c2.COURSE_CODE = 'CPS109'
      AND c1.INSTRUCTOR_ID = c2.INSTRUCTOR_ID
      AND e2.COURSE_CODE = 'CPS109'
      AND stu.USER_ID = e1.STUDENT_ID);

PROMPT Showing instructors in a department...;
SELECT DISTINCT u.FULL_NAME, d.DEPARTMENT_NAME
FROM USERS u
JOIN DEPARTMENTS d ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
WHERE ROLE = 'instructor'
GROUP BY d.DEPARTMENT_NAME, u.FULL_NAME, u.DEPARTMENT_ID ;

PROMPT Showing people in a specific department...;
SELECT DISTINCT us.FULL_NAME, de.DEPARTMENT_NAME
FROM USERS us
JOIN DEPARTMENTS de ON us.DEPARTMENT_ID = de.DEPARTMENT_ID
WHERE de.DEPARTMENT_ID = '1'
GROUP BY de.DEPARTMENT_NAME, us.FULL_NAME, us.DEPARTMENT_ID ;

PROMPT Showing average gpa of all students...;
SELECT 'Average GPA is ', AVG(GRADES.GPA_ACHIEVED)
FROM GRADES;

PROMPT Showing instructor course sections...;
SELECT c.INSTRUCTOR_ID, c.COURSE_CODE, c.SECTION_ID
FROM COURSE_SECTIONS c

PROMPT Showing all the student's names...;
SELECT FULL_NAME
FROM USERS
WHERE ROLE = 'student';

PROMPT Showing department users count...;
SELECT d.DEPARTMENT_ID, d.DEPARTMENT_NAME, COUNT(u.USER_ID) AS
"TOTAL_USERS"
```

```
FROM USERS u
JOIN DEPARTMENTS d ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
GROUP BY d.DEPARTMENT_ID, d.DEPARTMENT_NAME, u.DEPARTMENT_ID
ORDER BY d.DEPARTMENT_ID;

exit;
EOF
echo "Done!"
```

TA Recommendations

- No recommendations for this assignment
- Unix shell worked perfectly

Documentation of A6

Functional Dependencies

The following functional dependencies are identified.

Users

```
CREATE TABLE "USERS" (
  "USER_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 100000000,
  "USERNAME" VARCHAR2(25) UNIQUE NOT NULL ENABLE,
  "PASSWORD" VARCHAR2(64) NOT NULL ENABLE,
  "ROLE" VARCHAR2(25) NOT NULL ENABLE,
  "ENROLLMENT_DATE" DATE NOT NULL ENABLE,
  "PHONE_NUMBER" NUMBER(13, 0) UNIQUE NOT NULL ENABLE,
  "EMAIL_ADDRESS" VARCHAR2(254) UNIQUE NOT NULL ENABLE,
  "FULL_NAME" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_STREET_ADDRESS" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_CITY" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_PROVINCE" CHAR(2) NOT NULL ENABLE,
  "PROGRAM_ID" NUMBER,
  PRIMARY KEY ("USER_ID") USING INDEX ENABLE,
  UNIQUE ("PHONE_NUMBER") USING INDEX ENABLE,
  UNIQUE ("EMAIL_ADDRESS") USING INDEX ENABLE,
  CONSTRAINT "USER_ROLES" CHECK ( "ROLE" IN ('student', 'instructor', 'admin')) ENABLE
);
```

```
ALTER TABLE "USERS" ADD CONSTRAINT "USERS_PROGRAM" FOREIGN KEY ("PROGRAM_ID") REFERENCES "PROGRAMS" ("PROGRAM_ID") ON DELETE SET NULL ENABLE;
```

Users(User_Id, Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id)

{User_Id} → Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id
{Username} → User_Id
{Phone_Number} → User_Id
{Email_Address} → User_Id

Originally, there was a Department_Id column. It was no longer needed due to being transitively dependent on Program_Id (Program_Id → Department_Id from Programs table), so the column was removed. Now, this latest version of the Users table is in 3NF because there are no transitive dependencies and all non-primary key attributes are functionally dependent on the primary key.

Grades

```
CREATE TABLE "GRADES" (
  "GRADE_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  PRIMARY KEY ("GRADE_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "GRADES" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Grades(Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)

{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Academic_Letter

```
CREATE TABLE ACADEMIC_LETTER (
    RECORD_ID NUMBER GENERATED ALWAYS AS IDENTITY,
    GRADE_ID NUMBER REFERENCES GRADES(GRADE_ID),
    DATE_TAKEN DATE NOT NULL,
    STATUS VARCHAR2(32) NOT NULL,
    PRIMARY KEY(RECORD_ID)
);
```

Academic_Letter (Record_ID, Grade_ID, Date_Taken, Status)

{Record_ID} → Grade_ID, Date_Taken, Status

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Terms

```
CREATE TABLE TERMS (
    TERM_ID      NUMBER GENERATED ALWAYS AS IDENTITY,
    TERM_NAME    VARCHAR2(128) NOT NULL,
    START_DATE   DATE NOT NULL,
    END_DATE     DATE NOT NULL,
    PRIMARY KEY(TERM_ID),
    CONSTRAINT CK_Terms_DateRange CHECK(START_DATE <= END_DATE)
);
```

Terms (Term_ID, Term_Name, Start_Date, End_Date)

{Term_ID} → Term_Name, Start_Date, End_Date

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Student_Enrollements

```
CREATE TABLE "STUDENT_ENROLLMENTS" (
    TRANSACTION_ID NUMBER GENERATED ALWAYS AS IDENTITY,
    STUDENT_ID NUMBER REFERENCES USERS(USER_ID) NOT NULL,
    COURSE_CODE VARCHAR2(128) REFERENCES COURSES(COURSE_CODE) NOT NULL,
    SECTION_ID NUMBER REFERENCES COURSE_SECTIONS(SECTION_ID) NOT NULL,
    TERM_ID NUMBER REFERENCES TERMS(TERM_ID) NOT NULL,
    PRIMARY KEY(TRANSACTION_ID)
);
```

Student_Enrollments(Transaction_ID, Student_ID, Course_Code, Section_ID, Term_ID)

{Transaction_ID} → Course_Code, Section_ID, Term_ID, Student_ID

Many to Many relationship, where no functional dependencies hold between Students and Courses.

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Programs

```
CREATE TABLE  "PROGRAMS"
   (
    DEPARTMENTID    NUMBER REFERENCES DEPARTMENTS(DEPARTMENTID),
    PROGRAMNAME     VARCHAR2(128) NOT NULL,
    PROGRAMID       NUMBER GENERATED ALWAYS AS IDENTITY,
    PRIMARY KEY(PROGRAMID)
   );

```

Programs(Program_ID, Department_ID, Program_Name)
{Program_ID} → Department_ID, Program_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Course

```
CREATE TABLE COURSE (
    COURSE_CODE      VARCHAR2(128) PRIMARY KEY,
    HOUR_UNITS       NUMBER NOT NULL,
    COURSE_NAME      VARCHAR(128) NOT NULL,
    DEPARTMENT_ID    NUMBER NOT NULL REFERENCES DEPARTMENTS(DEPARTMENT_ID)
)
```

Course(Course_Code,Hour_Units,Course_Name,Department_ID)
{Course_Code} → Hour_Units, Course_Name, Department_ID
It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Course Sections

```
CREATE TABLE "COURSE_SECTIONS" (
  "COURSE_CODE" VARCHAR2(128),
  "INSTRUCTOR_ID" NUMBER,
  "TERM_ID" NUMBER,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  "CAPACITY" NUMBER NOT NULL ENABLE,
  CONSTRAINT "COURSE_SECTIONS_KEYS" PRIMARY KEY ("COURSE_CODE", "TERM_ID", "SECTION_ID") USING INDEX ENABLE,
  CONSTRAINT "COURSE_SECTIONS_CODE_NOT_NULL" CHECK ( "COURSE_CODE" IS NOT NULL) ENABLE
);
```

```
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("INSTRUCTOR_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
```

Course_Sections(Section_ID, Course_Code, Term_ID, Instructor_ID, Capacity)

{Section_ID, Course_Code, Term_ID} → Instructor_ID, Capacity

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Departments

```
CREATE TABLE DEPARTMENTS (
    DEPARTMENT_ID    NUMBER GENERATED ALWAYS AS IDENTITY,
    DEPARTMENT_NAME  VARCHAR2(128) NOT NULL,
    PRIMARY KEY(DEPARTMENT_ID)
);
```

Departments(Department_ID, Department_Name)

{Department_ID} → Department_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Financial_Information

```
CREATE TABLE "FINANCIAL_INFORMATION" (
  "ITEM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "LAST_ACTIVITY_DATE" DATE NOT NULL ENABLE,
  "AMOUNT_INCLUDING_TAX" NUMBER NOT NULL ENABLE,
  "BALANCE" NUMBER NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "ITEM_NAME" VARCHAR2(30) NOT NULL ENABLE,
  PRIMARY KEY ("ITEM_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Financial_Information(Item_Id, Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name)

{Item_Id} → Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

TA Recommendations

- Before, {Transaction_ID} → Course_Code, Section_ID, Term_ID
- Recommendation, {Transaction_ID} → Course_Code, Section_ID, Term_ID, Student_ID
- Reason: Student_ID depends on the determinant Transaction_ID

Documentation of A7

3NF Normalization

The following functional dependencies are analyzed and decomposed if needed.
Users

```
CREATE TABLE "USERS" (
  "USER_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 100000000,
  "USERNAME" VARCHAR2(25) UNIQUE NOT NULL ENABLE,
  "PASSWORD" VARCHAR2(64) NOT NULL ENABLE,
  "ROLE" VARCHAR2(25) NOT NULL ENABLE,
  "ENROLLMENT_DATE" DATE NOT NULL ENABLE,
  "PHONE_NUMBER" NUMBER(13, 0) NOT NULL ENABLE,
  "EMAIL_ADDRESS" VARCHAR2(254) UNIQUE NOT NULL ENABLE,
  "FULL_NAME" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_STREET_ADDRESS" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_CITY" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_PROVINCE" CHAR(2) NOT NULL ENABLE,
  "PROGRAM_ID" NUMBER,
  PRIMARY KEY ("USER_ID") USING INDEX ENABLE,
  UNIQUE ("PHONE_NUMBER") USING INDEX ENABLE,
  UNIQUE ("EMAIL_ADDRESS") USING INDEX ENABLE,
  CONSTRAINT "USER_ROLES" CHECK ( "ROLE" IN ('student', 'instructor', 'admin')) ENABLE
);
```

```
ALTER TABLE "USERS" ADD CONSTRAINT "USERS_PROGRAM" FOREIGN KEY ("PROGRAM_ID") REFERENCES "PROGRAMS" ("PROGRAM_ID") ON DELETE SET NULL ENABLE;
```

Users(User_Id, Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id)

{User_Id} → Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id
{Username} → User_Id
{Email_Address} → User_Id

(Note: We made the decision to exclude phone numbers as a unique attribute.)

Example of Decomposition

Grades & Academic_Letter

```
CREATE TABLE "GRADES & ACADEMIC_LETTER" (
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID", "GRADE_ID") USING INDEX ENABLE
);
```

Grades & Academic_Letter (Grade_ID, Record_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status)

Functional Dependencies:
{Grade_Id, Record_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved
{Record_ID} → Grade_ID, Date_Taken, Status

The partial dependency that can be shown is {Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved.

Candidate keys:
$\{Record\_ID\}^+$ = {Record_ID, Grade_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → CK
$\{Grade\_Id\}^+$ = {Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} ≠ Grades & Academic_Letter → Not CK
$\{Grade\_Id, Record\_ID\}^+$ → {Grade_Id, Record_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → This is redundant, so not a candidate key.

The table above violates 2NF, as there is redundancy for {Grade_Id} → …. Decompose Grades & Academic_Letter by making {Grade_Id} → … its own table. (The same results can be derived by making {Record_ID} → … its own table as well.)

Grades(Grade_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

Academic_Letter(Record_ID, Grade_ID, Date_Taken, Status)
{Record_ID} → Grade_ID, Date_Taken, Status

The table is now normalized using Bernstein's Algorithm, resulting in the Grades and Academic_Letter tables below.

Grades

```
CREATE TABLE "GRADES" (
  "GRADE_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  PRIMARY KEY ("GRADE_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "GRADES" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Grades(Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)

{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Academic_Letter

```
CREATE TABLE "ACADEMIC_LETTER" (
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID") USING INDEX ENABLE
);
```

Academic_Letter (Record_ID, Grade_ID, Date_Taken, Status)

{Record_ID} → Grade_ID, Date_Taken, Status

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Terms

```
CREATE TABLE "TERMS" (
  "TERM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "TERM_NAME" VARCHAR2(256) NOT NULL ENABLE,
  "START_DATE" DATE NOT NULL ENABLE,
  "END_DATE" DATE NOT NULL ENABLE,
  CONSTRAINT "CK_TERMS_DATERANGE" CHECK ("START_DATE"<="END_DATE") ENABLE,
  PRIMARY KEY ("TERM_ID") USING INDEX ENABLE
);
```

Terms (Term_ID, Term_Name, Start_Date, End_Date)

{Term_ID} → Term_Name, Start_Date, End_Date

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Student_Enrollements

```
CREATE TABLE "STUDENT_ENROLLMENTS" (
  "TRANSACTION_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "COURSE_CODE" VARCHAR2(128) NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("TRANSACTION_ID") USING INDEX ENABLE
);
```

Student_Enrollments(Transaction_ID, Student_ID, Course_Code, Section_ID, Term_ID)

{Transaction_ID} → Course_Code, Section_ID, Term_ID, Student_ID

Many to Many relationships, where no functional dependencies hold between Students and Courses.

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Programs

```
CREATE TABLE "PROGRAMS" (
  "PROGRAM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "PROGRAM_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER,
  PRIMARY KEY ("PROGRAM_ID") USING INDEX ENABLE
);
```

Programs(Program_ID, Department_ID, Program_Name)

{Program_ID} → Department_ID, Program_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Course

```
CREATE TABLE "COURSES" (
    "COURSE_CODE" VARCHAR2(128),
    "HOUR_UNITS" NUMBER NOT NULL ENABLE,
    "COURSE_NAME" VARCHAR2(128) NOT NULL ENABLE,
    "DEPARTMENT_ID" NUMBER NOT NULL ENABLE,
    PRIMARY KEY ("COURSE_CODE") USING INDEX ENABLE
);
```

Course(Course_Code,Hour_Units,Course_Name,Department_ID)

{Course_Code} → Hour_Units, Course_Name, Department_ID

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Course Sections

```
CREATE TABLE "COURSE_SECTIONS" (
  "COURSE_CODE" VARCHAR2(128),
  "INSTRUCTOR_ID" NUMBER,
  "TERM_ID" NUMBER,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  "CAPACITY" NUMBER NOT NULL ENABLE,
  CONSTRAINT "COURSE_SECTIONS_KEYS" PRIMARY KEY ("COURSE_CODE", "TERM_ID", "SECTION_ID") USING INDEX ENABLE,
  CONSTRAINT "COURSE_SECTIONS_CODE_NOT_NULL" CHECK ( "COURSE_CODE" IS NOT NULL) ENABLE
);
```

```
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("INSTRUCTOR_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
```

Course_Sections(Section_ID, Course_Code, Term_ID, Instructor_ID, Capacity)

{Section_ID, Course_Code, Term_ID} → Instructor_ID, Capacity

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Departments

```
CREATE TABLE "DEPARTMENTS" (
    "DEPARTMENT_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
    "DEPARTMENT_NAME" VARCHAR2(128) NOT NULL ENABLE,
    PRIMARY KEY ("DEPARTMENT_ID") USING INDEX ENABLE
  );
```

Departments(Department_ID, Department_Name)

{Department_ID} → Department_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

Financial_Information

```
CREATE TABLE "FINANCIAL_INFORMATION" (
  "ITEM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "LAST_ACTIVITY_DATE" DATE NOT NULL ENABLE,
  "AMOUNT_INCLUDING_TAX" NUMBER NOT NULL ENABLE,
  "BALANCE" NUMBER NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "ITEM_NAME" VARCHAR2(30) NOT NULL ENABLE,
  PRIMARY KEY ("ITEM_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Financial_Information(Item_Id, Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name)

{Item_Id} → Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name

It's 3NF because all non-primary key attributes are functionally dependent on the primary key.

TA Recommendations

- No recommendations for this assignment

Documentation of A8

BCNF Normalization

The following functional dependencies are analyzed and decomposed if needed.
Users

```
CREATE TABLE "USERS" (
  "USER_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 100000000,
  "USERNAME" VARCHAR2(25) UNIQUE NOT NULL ENABLE,
  "PASSWORD" VARCHAR2(64) NOT NULL ENABLE,
  "ROLE" VARCHAR2(25) NOT NULL ENABLE,
  "ENROLLMENT_DATE" DATE NOT NULL ENABLE,
  "PHONE_NUMBER" NUMBER(13, 0) NOT NULL ENABLE,
  "EMAIL_ADDRESS" VARCHAR2(254) UNIQUE NOT NULL ENABLE,
  "FULL_NAME" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_STREET_ADDRESS" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_CITY" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_PROVINCE" CHAR(2) NOT NULL ENABLE,
  "PROGRAM_ID" NUMBER,
  PRIMARY KEY ("USER_ID") USING INDEX ENABLE,
  UNIQUE ("PHONE_NUMBER") USING INDEX ENABLE,
  UNIQUE ("EMAIL_ADDRESS") USING INDEX ENABLE,
  CONSTRAINT "USER_ROLES" CHECK ( "ROLE" IN ('student', 'instructor', 'admin')) ENABLE
);
```

```
ALTER TABLE "USERS" ADD CONSTRAINT "USERS_PROGRAM" FOREIGN KEY ("PROGRAM_ID") REFERENCES "PROGRAMS" ("PROGRAM_ID") ON DELETE SET NULL ENABLE;
```

Users(User_Id, Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id)

{User_Id} → Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id
{Username} → User_Id
{Email_Address} → User_Id

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key and no non-prime attribute can functionally determine any of the prime attributes (User_Id, Username, Email_Address).

Example of Decomposition

Grades & Academic_Letter

```
CREATE TABLE "GRADES & ACADEMIC_LETTER" (
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID", "GRADE_ID") USING INDEX ENABLE
);
```

Grades & Academic_Letter (Grade_ID, Record_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status)

Functional Dependencies:
{Grade_Id, Record_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved
{Record_ID} → Grade_ID, Date_Taken, Status

The partial dependency that can be shown is {Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved.

Candidate keys:
$\{Record\_ID\}^+$ = {Record_ID, Grade_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → CK
$\{Grade\_Id\}^+$ = {Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} ≠ Grades & Academic_Letter → Not CK
$\{Grade\_Id, Record\_ID\}^+$ → {Grade_Id, Record_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → This is redundant, so not a candidate key.

The table above violates 2NF, as there is redundancy for {Grade_Id} → …. Decompose Grades & Academic_Letter by making {Grade_Id} → … its own table. (The same results can be derived by making {Record_ID} → … its own table as well.)

Grades(Grade_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

Academic_Letter(Record_ID, Grade_ID, Date_Taken, Status)
{Record_ID} → Grade_ID, Date_Taken, Status

These decomposed tables now follow BCNF as they are in BCNF and all dependencies rely on the super key. (See Grades and Academic_Letter tables.)

The table is now normalized using Bernstein's Algorithm, resulting in the Grades and Academic_Letter tables below.

Grades

```
CREATE TABLE "GRADES" (
  "GRADE_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  PRIMARY KEY ("GRADE_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "GRADES" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Grades(Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)

{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Academic_Letter

```
CREATE TABLE "ACADEMIC_LETTER" (
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID") USING INDEX ENABLE
);
```

Academic_Letter (Record_ID, Grade_ID, Date_Taken, Status)

{Record_ID} → Grade_ID, Date_Taken, Status

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Terms

```
CREATE TABLE "TERMS" (
  "TERM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "TERM_NAME" VARCHAR2(256) NOT NULL ENABLE,
  "START_DATE" DATE NOT NULL ENABLE,
  "END_DATE" DATE NOT NULL ENABLE,
  CONSTRAINT "CK_TERMS_DATERANGE" CHECK ("START_DATE"<="END_DATE") ENABLE,
  PRIMARY KEY ("TERM_ID") USING INDEX ENABLE
);
```

Terms (Term_ID, Term_Name, Start_Date, End_Date)

{Term_ID} → Term_Name, Start_Date, End_Date

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Student_Enrollements

```
CREATE TABLE "STUDENT_ENROLLMENTS" (
  "TRANSACTION_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "COURSE_CODE" VARCHAR2(128) NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("TRANSACTION_ID") USING INDEX ENABLE
);
```

Student_Enrollments(Transaction_ID, Student_ID, Course_Code, Section_ID, Term_ID)

{Transaction_ID} → Course_Code, Section_ID, Term_ID, Student_ID

Many to Many relationships, where no functional dependencies hold between Students and Courses.

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Programs

```
CREATE TABLE "PROGRAMS" (
  "PROGRAM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "PROGRAM_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER,
  PRIMARY KEY ("PROGRAM_ID") USING INDEX ENABLE
);
```

Programs(Program_ID, Department_ID, Program_Name)

{Program_ID} → Department_ID, Program_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Course

```
CREATE TABLE "COURSES" (
  "COURSE_CODE" VARCHAR2(128),
  "HOUR_UNITS" NUMBER NOT NULL ENABLE,
  "COURSE_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("COURSE_CODE") USING INDEX ENABLE
);
```

Course(Course_Code,Hour_Units,Course_Name,Department_ID)

{Course_Code} → Hour_Units, Course_Name, Department_ID

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Course Sections

```
CREATE TABLE "COURSE_SECTIONS" (
  "COURSE_CODE" VARCHAR2(128),
  "INSTRUCTOR_ID" NUMBER,
  "TERM_ID" NUMBER,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  "CAPACITY" NUMBER NOT NULL ENABLE,
  CONSTRAINT "COURSE_SECTIONS_KEYS" PRIMARY KEY ("COURSE_CODE", "TERM_ID", "SECTION_ID") USING INDEX ENABLE,
  CONSTRAINT "COURSE_SECTIONS_CODE_NOT_NULL" CHECK ( "COURSE_CODE" IS NOT NULL) ENABLE
);
```

```
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("INSTRUCTOR_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
```

Course_Sections(Section_ID, Course_Code, Term_ID, Instructor_ID, Capacity)

{Section_ID, Course_Code, Term_ID} → Instructor_ID, Capacity

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Departments

```
CREATE TABLE "DEPARTMENTS" (
    "DEPARTMENT_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
    "DEPARTMENT_NAME" VARCHAR2(128) NOT NULL ENABLE,
    PRIMARY KEY ("DEPARTMENT_ID") USING INDEX ENABLE
  );
```

Departments(Department_ID, Department_Name)

{Department_ID} → Department_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Financial_Information

```
CREATE TABLE "FINANCIAL_INFORMATION" (
  "ITEM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "LAST_ACTIVITY_DATE" DATE NOT NULL ENABLE,
  "AMOUNT_INCLUDING_TAX" NUMBER NOT NULL ENABLE,
  "BALANCE" NUMBER NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "ITEM_NAME" VARCHAR2(30) NOT NULL ENABLE,
  PRIMARY KEY ("ITEM_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Financial_Information(Item_Id, Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name)

{Item_Id} → Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

TA Recommendations

- No recommendations for this assignment

Documentation of A9

How to access our database

https://apex.oracle.com/pls/apex/r/ramss/better-ramss

There is no code shown in the oracle database but our SQL queries are listed below. To login into the database, please use the following login credentials. This application is a web GUI developed on Oracle Apex.

Note: Do NOT put any actual sensitive information in this database! The current password is exposed in plain text for demo purposes only!

USER_ID	USERNAME	PASSWORD	ROLE
100000060	alex.joel	alex.joel.11	student
100000081	ab.cd	ab.cd	admin
100000040	jason.green	jason.green	instructor
100000064	ker.hen	kermitTheFrog	instructor
100000004	quan.ding	3RzPb@@W8lry	student

3NF/BCNF

The following functional dependencies are analyzed and decomposed if needed.

Users

```
CREATE TABLE "USERS" (
  "USER_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 100000000,
  "USERNAME" VARCHAR2(25) UNIQUE NOT NULL ENABLE,
  "PASSWORD" VARCHAR2(64) NOT NULL ENABLE,
  "ROLE" VARCHAR2(25) NOT NULL ENABLE,
  "ENROLLMENT_DATE" DATE NOT NULL ENABLE,
  "PHONE_NUMBER" NUMBER(13, 0) NOT NULL ENABLE,
  "EMAIL_ADDRESS" VARCHAR2(254) UNIQUE NOT NULL ENABLE,
  "FULL_NAME" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_STREET_ADDRESS" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_CITY" VARCHAR2(30) NOT NULL ENABLE,
  "HOME_PROVINCE" CHAR(2) NOT NULL ENABLE,
  "PROGRAM_ID" NUMBER,
  PRIMARY KEY ("USER_ID") USING INDEX ENABLE,
  UNIQUE ("PHONE_NUMBER") USING INDEX ENABLE,
  UNIQUE ("EMAIL_ADDRESS") USING INDEX ENABLE,
  CONSTRAINT "USER_ROLES" CHECK ( "ROLE" IN ('student', 'instructor', 'admin')) ENABLE
);
```

```
ALTER TABLE "USERS" ADD CONSTRAINT "USERS_PROGRAM" FOREIGN KEY ("PROGRAM_ID") REFERENCES "PROGRAMS" ("PROGRAM_ID") ON DELETE SET NULL ENABLE;
```

Users(User_Id, Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id)

{User_Id} → Username, Password, Role, Enrollment_Date, Phone_Number, Email_Address, Full_Name, Home_Street_Address, Home_City, Home_Province, Program_Id
{Username} → User_Id
{Email_Address} → User_Id

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key and no non-prime attribute can functionally determine any of the prime attributes (User_Id, Username, Email_Address).

Example of Decomposition

Grades & Academic_Letter

```
CREATE TABLE "GRADES & ACADEMIC_LETTER" (
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID", "GRADE_ID") USING INDEX ENABLE
);
```

Grades & Academic_Letter (Grade_ID, Record_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status)

Functional Dependencies:
{Grade_Id, Record_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved, Date_Taken, Status
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved
{Record_ID} → Grade_ID, Date_Taken, Status

The partial dependency that can be shown is {Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved.

Candidate keys:
$\{Record\_ID\}^+$ = {Record_ID, Grade_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → CK
$\{Grade\_Id\}^+$ = {Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} ≠ Grades & Academic_Letter → Not CK
$\{Grade\_Id, Record\_ID\}^+$ → {Grade_Id, Record_ID, Date_Taken, Status, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved} = Grades & Academic_Letter → This is redundant, so not a candidate key.

The table above violates 2NF, as there is redundancy for {Grade_Id} → …. Decompose Grades & Academic_Letter by making {Grade_Id} → … its own table. (The same results can be derived by making {Record_ID} → … its own table as well.)

Grades(Grade_ID, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)
{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

Academic_Letter(Record_ID, Grade_ID, Date_Taken, Status)
{Record_ID} → Grade_ID, Date_Taken, Status

These decomposed tables now follow BCNF as they are in BCNF and all dependencies rely on the super key. (See Grades and Academic_Letter tables.)

The table is now normalized using Bernstein's Algorithm, resulting in the Grades and Academic_Letter tables below.

Grades

```
CREATE TABLE "GRADES" (
  "GRADE_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "COURSE_CODE" VARCHAR2(128),
  "STUDENT_ID" NUMBER,
  "CREDIT_UNITS" NUMBER,
  "GRADE" CHAR(2),
  "GPA_ACHIEVED" NUMBER(3, 2) NOT NULL ENABLE,
  PRIMARY KEY ("GRADE_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "GRADES" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "GRADES" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Grades(Grade_Id, Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved)

{Grade_Id} → Course_Code, Student_Id, Credit_Units, Grade, GPA_Achieved

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Academic_Letter

```
CREATE TABLE "ACADEMIC_LETTER" (
  "RECORD_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "GRADE_ID" VARCHAR2(5),
  "DATE_TAKEN" DATE NOT NULL ENABLE,
  "STATUS" VARCHAR2(32) NOT NULL ENABLE,
  PRIMARY KEY ("RECORD_ID") USING INDEX ENABLE
);
```

Academic_Letter (Record_ID, Grade_ID, Date_Taken, Status)

{Record_ID} → Grade_ID, Date_Taken, Status

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Terms

```
CREATE TABLE "TERMS" (
  "TERM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "TERM_NAME" VARCHAR2(256) NOT NULL ENABLE,
  "START_DATE" DATE NOT NULL ENABLE,
  "END_DATE" DATE NOT NULL ENABLE,
  CONSTRAINT "CK_TERMS_DATERANGE" CHECK ("START_DATE"<="END_DATE") ENABLE,
  PRIMARY KEY ("TERM_ID") USING INDEX ENABLE
);
```

Terms (Term_ID, Term_Name, Start_Date, End_Date)

{Term_ID} → Term_Name, Start_Date, End_Date

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Student_Enrollements

```
CREATE TABLE "STUDENT_ENROLLMENTS" (
  "TRANSACTION_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "COURSE_CODE" VARCHAR2(128) NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  PRIMARY KEY ("TRANSACTION_ID") USING INDEX ENABLE
);
```

Student_Enrollments(Transaction_ID, Student_ID, Course_Code, Section_ID, Term_ID)

{Transaction_ID} → Course_Code, Section_ID, Term_ID, Student_ID

Many to Many relationships, where no functional dependencies hold between Students and Courses.

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Programs

```
CREATE TABLE "PROGRAMS" (
  "PROGRAM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "PROGRAM_NAME" VARCHAR2(128) NOT NULL ENABLE,
  "DEPARTMENT_ID" NUMBER,
  PRIMARY KEY ("PROGRAM_ID") USING INDEX ENABLE
);
```

Programs(Program_ID, Department_ID, Program_Name)

{Program_ID} → Department_ID, Program_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Course

```
CREATE TABLE "COURSES" (
    "COURSE_CODE" VARCHAR2(128),
    "HOUR_UNITS" NUMBER NOT NULL ENABLE,
    "COURSE_NAME" VARCHAR2(128) NOT NULL ENABLE,
    "DEPARTMENT_ID" NUMBER NOT NULL ENABLE,
    PRIMARY KEY ("COURSE_CODE") USING INDEX ENABLE
  );
```

Course(Course_Code,Hour_Units,Course_Name,Department_ID)

{Course_Code} → Hour_Units, Course_Name, Department_ID

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Course Sections

```
CREATE TABLE "COURSE_SECTIONS" (
  "COURSE_CODE" VARCHAR2(128),
  "INSTRUCTOR_ID" NUMBER,
  "TERM_ID" NUMBER,
  "SECTION_ID" NUMBER NOT NULL ENABLE,
  "CAPACITY" NUMBER NOT NULL ENABLE,
  CONSTRAINT "COURSE_SECTIONS_KEYS" PRIMARY KEY ("COURSE_CODE", "TERM_ID", "SECTION_ID") USING INDEX ENABLE,
  CONSTRAINT "COURSE_SECTIONS_CODE_NOT_NULL" CHECK ( "COURSE_CODE" IS NOT NULL) ENABLE
);
```

```
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("COURSE_CODE") REFERENCES "COURSES" ("COURSE_CODE") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("INSTRUCTOR_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
ALTER TABLE "COURSE_SECTIONS" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
```

Course_Sections(Section_ID, Course_Code, Term_ID, Instructor_ID, Capacity)

{Section_ID, Course_Code, Term_ID} → Instructor_ID, Capacity

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Departments

```
CREATE TABLE "DEPARTMENTS" (
    "DEPARTMENT_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
    "DEPARTMENT_NAME" VARCHAR2(128) NOT NULL ENABLE,
    PRIMARY KEY ("DEPARTMENT_ID") USING INDEX ENABLE
  );
```

Departments(Department_ID, Department_Name)

{Department_ID} → Department_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

Financial_Information

```
CREATE TABLE "FINANCIAL_INFORMATION" (
  "ITEM_ID" NUMBER GENERATED BY DEFAULT AS IDENTITY MINVALUE 1 MAXVALUE 9999999999999999999999999999 INCREMENT BY 1 START WITH 1,
  "LAST_ACTIVITY_DATE" DATE NOT NULL ENABLE,
  "AMOUNT_INCLUDING_TAX" NUMBER NOT NULL ENABLE,
  "BALANCE" NUMBER NOT NULL ENABLE,
  "TERM_ID" NUMBER NOT NULL ENABLE,
  "STUDENT_ID" NUMBER NOT NULL ENABLE,
  "ITEM_NAME" VARCHAR2(30) NOT NULL ENABLE,
  PRIMARY KEY ("ITEM_ID") USING INDEX ENABLE
);
```

```
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("TERM_ID") REFERENCES "TERMS" ("TERM_ID") ENABLE;
ALTER TABLE "FINANCIAL_INFORMATION" ADD FOREIGN KEY ("STUDENT_ID") REFERENCES "USERS" ("USER_ID") ENABLE;
```

Financial_Information(Item_Id, Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name)

{Item_Id} → Last_Activity_Date, Amount_Including_Tax, Balance, Term_Id, Student_Id, Item_Name

It's BCNF because all non-primary key attributes are functionally dependent on the primary (super) key.

TA Recommendations

- No recommendations for this assignment

Relational Algebra (A10)

Note: τ_A means order the relations by the attributes in A.

AverageGPAofAllStudents

```
1   SELECT 'Average GPA is ', AVG(GRADES.GPA_ACHIEVED)
2   FROM GRADES;
```

$$\Pi_{GRADES}(F_{AVG(GPA\_ACHIEVED)}(GRADES))$$

AllNamesUsers

```
1   SELECT FULL_NAME
2   FROM USERS;
```

$$\Pi_{FULL\_NAME}(USERS)$$

AllStudentNames

```
1   SELECT FULL_NAME
2   FROM USERS
3   WHERE ROLE = 'student';
```

$$\Pi_{FULL\_NAME}(\sigma_{ROLE = \text{“student”}}(USERS))$$

AllStudentGPA

```
1   SELECT GRADES.STUDENT_ID, 'overall GPA is: ', AVG(GRADES.GPA_ACHIEVED)
2   FROM GRADES
3   GROUP BY GRADES.STUDENT_ID
```

$$\Pi_{STUDENT\_ID}(F_{AVG(GPA\_ACHIEVED)}(GRADES))$$

OutstandingBalance

```
1   SELECT STUDENT_ID, 'Outstanding balance is: ', BALANCE
2   FROM FINANCIAL_INFORMATION
3   WHERE BALANCE > 0;
```

$$\Pi_{STUDENT\_ID}(\sigma_{BALANCE > 0}(FINANCIAL\_INFORMATION))$$

StudentsFromSpecificSection

```sql
SELECT DISTINCT STUDENT_ENROLLMENTS.STUDENT_ID, USERS.FULL_NAME, COURSE_CODE, SECTION_ID
FROM STUDENT_ENROLLMENTS
JOIN USERS ON USERS.USER_ID = STUDENT_ENROLLMENTS.STUDENT_ID
WHERE COURSE_CODE = 'COE318'
    AND SECTION_ID = '1'
ORDER BY STUDENT_ENROLLMENTS.STUDENT_ID ASC;
```

$$\Pi_{\text{STUDENT\_ID,COURSE\_CODE,SECTION\_ID}}(\sigma_{\text{COURSE\_CODE = “COE318” AND SECTION\_ID = ‘1’}}(\text{STUDENT\_ENROLLMENTS})) \gtrless_{\text{USERS.USER\_ID = STUDENT\_ENROLLMENTS.STUDENT\_ID}}$$

DepartmentUsersCount

```sql
CREATE OR REPLACE FORCE EDITIONABLE VIEW "NUMBER_IN_DEPARTMENT" ("DEPARTMENT_ID", "DEPARTMENT_NAME", "NUMBER_OF_PEOPLE") AS
  (
    SELECT d.DEPARTMENT_ID, d.DEPARTMENT_NAME, COUNT(u.USER_ID) AS NUMBER_OF_PEOPLE
    FROM DEPARTMENTS d
    JOIN PROGRAMS p ON p.DEPARTMENT_ID = d.DEPARTMENT_ID
    JOIN USERS u ON p.PROGRAM_ID = u.PROGRAM_ID
    GROUP BY d.DEPARTMENT_ID, d.DEPARTMENT_NAME, u.USER_ID

)
ORDER BY d.DEPARTMENT_ID ASC
WITH READ ONLY
```

$$\tau_{\text{d.department\_id}}(\Pi_{\text{d.department\_name, COUNT(user\_id)} \rightarrow \text{number\_of\_people}}(_{\text{department\_name, user\_id}}F_{\text{COUNT(user\_id)}}(\rho_d(\text{departments}) \gtrless_{\text{p.department\_id = d.department\_id}} \rho_p(\text{programs}) \gtrless_{\text{p.program\_id = u.program\_id}} \rho_u(\text{users}))))$$

InstructorCourseSection

```sql
SELECT c.INSTRUCTOR_ID, c.COURSE_CODE, c.SECTION_ID
FROM COURSE_SECTIONS c
```

$$\Pi_{\text{c.instructor\_id, c.course\_code, c.section\_id}}(\rho_c(\text{course\_sections}))$$

CourseSectionList

```sql
CREATE OR REPLACE VIEW COURSE_SECTIONS_LIST AS
(
    SELECT cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME AS TERM, u.FULL_NAME AS INSTRUCTOR, COUNT(se.STUDENT_ID) AS TOTAL_ENROLLED, cs.CAPACITY
    FROM COURSE_SECTIONS cs
    JOIN COURSES c ON c.COURSE_CODE = cs.COURSE_CODE
    JOIN USERS u ON u.USER_ID = cs.INSTRUCTOR_ID
    JOIN TERMS t ON t.TERM_ID = cs.TERM_ID
    LEFT JOIN STUDENT_ENROLLMENTS se ON se.COURSE_CODE = cs.COURSE_CODE AND se.SECTION_ID = cs.SECTION_ID
    GROUP BY cs.COURSE_CODE, cs.SECTION_ID, c.COURSE_NAME, t.TERM_NAME, u.FULL_NAME, cs.CAPACITY
    HAVING COUNT(se.STUDENT_ID) < cs.CAPACITY

)
ORDER BY cs.COURSE_CODE ASC, cs.SECTION_ID ASC
WITH READ ONLY;
```

$$\Pi_{\text{cs.course\_code, cs.section\_id, c.course\_name, t.term\_name} \rightarrow \text{term, t.term\_id} \rightarrow \text{term\_id, u.full\_name} \rightarrow \text{instructor, COUNT (student\_id)} \rightarrow \text{total\_enrolled, cs.capacity}}$$

$$(\sigma_{\text{COUNT(student\_id)} < \text{cs.capacity}}$$

$$(_{\text{course\_code, section\_id, course\_name, term\_name, term\_id, full\_name, capacity}}F_{\text{COUNT (student\_id)}}$$

$$(\rho_{cs}(\text{course\_sections}) ><_{\text{c.course\_code = cs.course\_code}}$$

$$\rho_{c}(\text{courses}) ><_{\text{u.user\_id = cs.instructor\_id}}$$

$$\rho_{u}(\text{users}) ><_{\text{t.term\_id = cs.term\_id}}$$

$$\rho_{t}(\text{terms}) ><_{\text{se.course\_code = cs.course\_code AND se.section\_id = cs.section\_id}}$$

$$\rho_{se}(\text{student\_enrollments})$$

$$)$$

$$)$$

$$)$$

ProgramOfferings

```sql
CREATE OR REPLACE VIEW PROGRAM_OFFERINGS AS
    (
        SELECT p.PROGRAM_ID, p.PROGRAM_NAME, d.DEPARTMENT_NAME
        FROM PROGRAMS p
        JOIN DEPARTMENTS d ON p.DEPARTMENT_ID = d.DEPARTMENT_ID
    )
ORDER BY p.PROGRAM_NAME ASC
WITH READ ONLY;
```

$$\tau_{p.program\_name}(\Pi_{p.program\_id,\ p.program\_name,\ d.department\_name}(\rho_p(programs) >\!\!<_{p.department\_id = d.department\_id}$$

$$\rho_d(departments)))$$

AllPeopleInASpecificDepartment

```sql
SELECT DISTINCT us.FULL_NAME, de.DEPARTMENT_NAME
FROM USERS us
JOIN DEPARTMENTS de ON us.DEPARTMENT_ID = de.DEPARTMENT_ID
WHERE de.DEPARTMENT_ID = '1'
GROUP BY de.DEPARTMENT_NAME, us.FULL_NAME, us.DEPARTMENT_ID ;
```

$$\Pi_{us.full\_name,\ de.department\_name}(\rho_{us}(USERS) >\!\!<_{us.department\_id = de.department\_id} (\rho_{de}(DEPARTMENTS)$$

$$(\sigma_{DEPARTMENT\_ID = \text{“1”}})))$$

InstructorsInADepartment

```sql
SELECT DISTINCT u.FULL_NAME, d.DEPARTMENT_NAME
FROM USERS u
JOIN DEPARTMENTS d ON u.DEPARTMENT_ID = d.DEPARTMENT_ID
WHERE ROLE = 'instructor'
GROUP BY d.DEPARTMENT_NAME, u.FULL_NAME, u.DEPARTMENT_ID ;
```

$$\Pi_{u.full\_name,\ d.department\_name}(\rho_u(USERS) >\!\!<_{u.department\_id = d.department\_id} (\rho_d(DEPARTMENTS)$$

$$(\sigma_{ROLE = \text{“instructor”}})))$$

StudentsTaking2SpecificCoursesTogether

```sql
SELECT DISTINCT e1.STUDENT_ID, stu.FULL_NAME
FROM COURSE_SECTIONS c1, COURSE_SECTIONS c2, STUDENT_ENROLLMENTS e1, STUDENT_ENROLLMENTS e2, USERS stu
WHERE c1.INSTRUCTOR_ID = 100000064
    AND c1.COURSE_CODE ='BLG143'
    AND e1.STUDENT_ID = e2.STUDENT_ID
    AND e1.COURSE_CODE = 'BLG143'
    AND c2.INSTRUCTOR_ID = 100000064
    AND c1.TERM_ID = c2.TERM_ID
    AND c2.COURSE_CODE = 'CPS109'
    AND c1.INSTRUCTOR_ID = c2.INSTRUCTOR_ID
    AND e2.COURSE_CODE = 'CPS109'
    AND stu.USER_ID = e1.STUDENT_ID;
```

$$
\begin{aligned}
&\delta(\\
&\Pi_{e1.student\_id,\ stu.full\_name}(\\
&\sigma_{c1.instructor\_id = 100000064 \text{ AND } c1.course\_code = \text{"BLG143"} \text{ AND } e1.student\_id = e2.student\_id \text{ AND } e1.course\_code = \text{"BLG143"} \text{ AND } \\
&c2.instructor\_id = 100000064 \text{ AND } c1.term\_id = c2.term\_id \text{ AND } c2.course\_code = \text{"CPS109"} \text{ AND } c1.instructor\_id = c2.instructor\_id \text{ AND } e2.course\_code = \\
&\text{"CPS109"} \text{ AND } stu.user\_id = e1.student\_id \\
&(\rho_{c1}(course\_sections) \times \\
&\rho_{c2}(course\_sections) \times \\
&\rho_{e1}(student\_enrollments) \times \\
&\rho_{e2}(student\_enrollments) \times \\
&\rho_{stu}(users)) \\
&) \\
&)
\end{aligned}
$$

GradesInATerm

```sql
SELECT s.STUDENT_ID, t.TERM_NAME, AVG(g.GPA_ACHIEVED) as "Average GPA Achieved"
FROM STUDENT_ENROLLMENTS s
JOIN TERMS t ON t.TERM_ID = s.TERM_ID
JOIN GRADES g ON s.STUDENT_ID = g.STUDENT_ID
JOIN ACADEMIC_LETTER l ON l.GRADE_ID = g.GRADE_ID
GROUP BY t.TERM_NAME, s.STUDENT_ID, g.GPA_ACHIEVED
```

$$\Pi_{\text{s.student\_id, t.term\_name, AVG (gpa\_achieved)} \rightarrow \text{gpa\_achieved}}($$

$$_{\text{term\_name, student\_id, gpa\_achieved}}F_{\text{AVG (gpa\_achieved)}}$$

$$(\rho_s(\text{student\_enrollments}) ><_{\text{t.term\_id = s.term\_id}}$$

$$\rho_t(\text{terms}) ><_{\text{s.student\_id = g.student\_id}}$$

$$\rho_g(\text{grades}) ><_{\text{l.grade\_id = g.grade\_id}}$$

$$\rho_l(\text{academic\_letter}))$$

)

TotalStudentsRegisteredInACourse

```sql
SELECT COURSE_CODE, COUNT(STUDENT_ID) as Number_Enrolled
FROM STUDENT_ENROLLMENTS
GROUP BY COURSE_CODE;
```

$$\Pi_{\text{COURSE\_CODE}}({}_{\text{COURSE\_CODE}}F_{\text{COUNT(STUDENT\_ID)-->Number\_Enrolled}}(\text{STUDENT\_ENROLLMENTS}))$$

Conclusion

Our application is called the Student Administration System (Better RAMMS), an online self-service administrative platform that allows students and staff in the university to view or update courses, grades, academic letters, personal and financial information. Using three different users for this application, the first user is a student who is able to view their courses, grades, academic letter, and personal and financial information. The second user is an instructor who updates the end-of-the-semester GPA. Lastly, the system administrator will update the platform to keep track of all the students' information and courses.

The application is a GUI with the help of Oracle Apex. Our GUI uses the login credentials of three different users. Through the application, the three different users will receive three different views of the application.

USER_ID	USERNAME	PASSWORD	ROLE
100000060	alex.joel	alex.joel.11	student
100000081	ab.cd	ab.cd	admin
100000040	jason.green	jason.green	instructor
100000064	ker.hen	kermitTheFrog	instructor
100000004	quan.ding	3RzPb@@W8lry	student