UNIVERSITY OF
GOTHENBURG

CHALMERS

# Summary and explanation of the code

## 1 TABLES.SQL

The code in **tables.sql** creates multiple tables in a database with various columns and constraints. Each CREATE TABLE statement defines a table with a given name and a set of columns, each of which has a data type and potentially other constraints.

```
CREATE TABLE Programs(
    name TEXT PRIMARY KEY,
    abbreviation TEXT NOT NULL UNIQUE
);
```

The ***Programs*** table has two columns, ***name*** and ***abbreviation***. The ***name*** column is the **PRIMARY KEY** for the table, which means that it uniquely identifies each row in the table. The ***abbreviation*** column is marked as NOT NULL, which means that it cannot contain NULL values. It is also marked as **UNIQUE**, which means that no two rows in the table can have the same value in the abbreviation column.

```
CREATE TABLE Students (
        idnr CHAR(10) PRIMARY KEY,
        name TEXT NOT NULL,
        login TEXT NOT NULL UNIQUE,
        program TEXT NOT NULL,
        CONSTRAINT check_idnr CHECK(idnr~'^[0-9]{10}$'),
        FOREIGN KEY(program) REFERENCES Programs,
        UNIQUE(idnr, program)
);
```

The ***Students*** table has columns for the student's ID number (***idnr***), name (***name***), login (***login***), and program (***program***). The ***idnr*** column is defined as a CHAR(10) data type, which means it can store up to 10 characters. This column is also marked as the **PRIMARY KEY**, which means that it is a unique identifier for each row in the table. The login column is marked as **UNIQUE**, which means that it cannot have duplicate values. Additionally, the ***idnr*** and ***program*** columns are marked as **UNIQUE** together, which means that each combination of values for these two columns must be unique in the table. Finally, the program column has a **FOREIGN KEY** constraint that references the ***Programs*** table, which means that it must have a value that exists in the ***Programs*** table.

```
CREATE TABLE Branches (
        name TEXT,
        program TEXT,
        PRIMARY KEY(name,program),
        FOREIGN KEY(program) REFERENCES Programs
);
```

The ***Branches*** table has columns for the name (***name***) and program (***program***) of a branch. The ***name*** and ***program*** columns are marked as the **PRIMARY KEY**, which means that each combination of values for these two columns must be unique in the table. The program column also has a **FOREIGN KEY** constraint that references the ***Programs*** table, which means that it must have a value that exists in the ***Programs*** table.

```
CREATE TABLE Departments (
    name TEXT PRIMARY KEY,
    abbreviation TEXT NOT NULL UNIQUE
);
```

The ***Departments*** table has columns for the name (***name***) and abbreviation (***abbreviation***) of a department. The name column is marked as the **PRIMARY KEY**, which means that it is a unique identifier for each row in the table. The abbreviation column is marked as **UNIQUE**, which means that it cannot have duplicate values.

```
CREATE TABLE Courses (
        code CHAR(6) PRIMARY KEY,
        name TEXT NOT NULL,
        credits DECIMAL NOT NULL,
        department TEXT NOT NULL,
        CONSTRAINT check_more_than_zero CHECK(credits > 0),
        FOREIGN KEY(department) REFERENCES Departments,
        CONSTRAINT check_course_format CHECK(code~'^[A-Z]{3}[0-9]{3}$')
);
```

The ***Courses*** table has columns for the course code (***code***), name (***name***), number of credits (***credits***), and department (***department***). The code column is defined as a CHAR(6) data type and is marked as the **PRIMARY KEY**, which means that it is a unique identifier for each row in the table. The credits column is defined as a DECIMAL data type, which can store decimal values. This column has a **CHECK** constraint that ensures that the value is greater than 0. The department column has a **FOREIGN KEY** constraint that references the ***Departments*** table, which means that it must have a value that exists in the ***Departments*** table.

```
CREATE TABLE Prerequisite (
    prerequisiteCourse CHAR(6) NOT NULL,
    course CHAR(6) NOT NULL,
    PRIMARY KEY(prerequisiteCourse, course),
    FOREIGN KEY(prerequisiteCourse) REFERENCES Courses(code),
    FOREIGN KEY(course) REFERENCES Courses(code)
);
```

The ***Prerequisite*** table has columns for the prerequisite course code (***prerequisiteCourse***) and course code (***course***). The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. Both columns also have **FOREIGN KEY** constraints that reference the code column in the ***Courses*** table, which means that they must have values that exist in the ***Courses*** table.

```
CREATE TABLE LimitedCourses (
        code CHAR(6),
        capacity SMALLINT NOT NULL,
        CONSTRAINT more_than_zero CHECK(capacity > 0),
        PRIMARY KEY(code),
        FOREIGN KEY(code) REFERENCES Courses
);
```

The ***LimitedCourses*** table has columns for the course code (***code***) and capacity (***capacity***). The code column has a **FOREIGN KEY** constraint that references the ***Courses*** table, which means that it must have a value that exists in the ***Courses*** table. The capacity column is defined as a SMALLINT data type, which can store small integer values. It has a **CHECK** constraint that ensures that the value is greater than 0.

```
CREATE TABLE HostedBy (
        department TEXT,
        program TEXT,
        PRIMARY KEY (department, program),
        FOREIGN KEY(department) REFERENCES Departments(name),
        FOREIGN KEY(program) REFERENCES Programs
);
```

The ***HostedBy*** table has columns for the department (***department***) and program (***program***) that a course is hosted by. The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The department column has a **FOREIGN KEY** constraint that references the name column in the ***Departments*** table, while the program column has a **FOREIGN KEY** constraint that references the **Programs** table. This means that both columns must have values that exist in their respective tables.

```
CREATE TABLE StudentBranches (
        student CHAR(10),
        branch TEXT NOT NULL,
        program TEXT NOT NULL,
        PRIMARY KEY(student, branch, program),
        FOREIGN KEY(student) REFERENCES Students,
        FOREIGN KEY(branch, program) REFERENCES Branches(name, program),
        FOREIGN KEY(student, program) REFERENCES Students(idnr, program)
);
```

The ***StudentBranches*** table has columns for the student ID number (***student***), branch name (***branch***), and program (program). The combination of values for these three columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The student column has a **FOREIGN KEY** constraint that references the ***idnr*** column in the ***Students*** table, while the branch and program columns have a **FOREIGN KEY** constraint that references the combination of the ***name*** and ***program*** columns in the ***Branches*** table. This means that all three columns must have values that exist in their respective tables.

```
CREATE TABLE Classifications (
        name TEXT PRIMARY KEY
);
```

The ***Classifications*** table has a single column, ***name***, which is defined as a TEXT data type and is marked as the **PRIMARY KEY** for the table.

```
CREATE TABLE Classified (
        course CHAR(6),
        classification TEXT,
        PRIMARY KEY(course, classification),
        FOREIGN KEY(course) REFERENCES Courses(code),
        FOREIGN KEY(classification) REFERENCES Classifications(name)
);
```

The ***Classified*** table has columns for the course code (***course***) and classification name (***classification***). The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The course column has a **FOREIGN KEY** constraint that references the ***code*** column in the ***Courses*** table, while the ***classification*** column has a **FOREIGN KEY** constraint that references the ***name*** column in the ***Classifications*** table. This means that both columns must have values that exist in their respective tables.

```
CREATE TABLE MandatoryProgram (
        course CHAR(6),
        program TEXT,
        PRIMARY KEY(course, program),
        FOREIGN KEY(course) REFERENCES Courses(code),
        FOREIGN KEY(program) REFERENCES Programs
);
```

The ***MandatoryProgram*** table has columns for the course code (***course***) and program (***program***). The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The course column has a **FOREIGN KEY** constraint that references the ***code*** column in the ***Courses*** table, while the program column has a **FOREIGN KEY** constraint that references the ***Programs*** table. This means that both columns must have values that exist in their respective tables.

```
CREATE TABLE MandatoryBranch (
        course CHAR(6),
        branch TEXT,
        program TEXT,
        PRIMARY KEY(course, branch, program),
        FOREIGN KEY(course) REFERENCES Courses(code),
        FOREIGN KEY(branch, program) REFERENCES Branches(name, program)
);
```

***MandatoryBranch*** table has columns for the course code (***course***), branch name (***branch***), and program (***program***). The combination of values for these three columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The ***course*** column has a **FOREIGN KEY** constraint that references the code column in the ***Courses*** table, while the ***branch*** and ***program*** columns have a **FOREIGN KEY** constraint that references the combination of the ***name*** and ***program*** columns in the ***Branches*** table. This means that all three columns must have values that exist in their respective tables.

```
CREATE TABLE RecommendedBranch (
        course CHAR(6),
        branch TEXT,
        program TEXT,
        PRIMARY KEY(course, branch, program),
        FOREIGN KEY(course) REFERENCES Courses(code),
        FOREIGN KEY(branch, program) REFERENCES Branches(name, program)
);
```

The ***RecommendedBranch*** table has columns for the course code (***course***), branch name (***branch***), and program (***program***). The combination of values for these three columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The course column has a **FOREIGN KEY** constraint that references the ***code*** column in the ***Courses*** table, while the ***branch*** and ***program*** columns have a **FOREIGN KEY** constraint that references the combination of the ***name*** and ***program*** columns in the ***Branches*** table. This means that all three columns must have values that exist in their respective tables.

```
CREATE TABLE Registered (
        student CHAR(10),
        course CHAR(6),
        PRIMARY KEY(student, course),
        FOREIGN KEY(student) REFERENCES Students(idnr),
        FOREIGN KEY(course) REFERENCES Courses(code)
);
```

The ***Registered*** table has columns for the student ID (***student***) and course code (***course***). The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The ***student*** column has a **FOREIGN KEY** constraint that references the ***idnr*** column in the ***Students*** table, while the ***course*** column has a **FOREIGN KEY** constraint that references the ***code*** column in the ***Courses*** table. This means that both columns must have values that exist in their respective tables.

```
CREATE TABLE Taken(
        student CHAR(10),
        course CHAR(6),
        grade CHAR(1) NOT NULL,
        CONSTRAINT valid_grade CHECK(grade IN ('U', '3', '4', '5')),
        PRIMARY KEY(student, course),
        FOREIGN KEY(student) REFERENCES Students(idnr),
        FOREIGN KEY(course) REFERENCES Courses(code)
);
```

The ***Taken*** table has columns for the student ID (***student***) and course code (***course***). The combination of values for these two columns is marked as the **PRIMARY KEY**, which means that each combination of values must be unique in the table. The ***student*** column has a **FOREIGN KEY** constraint that references the ***idnr*** column in the ***Students*** table, while the ***course*** column has a **FOREIGN KEY** constraint that references the ***code*** column in the ***Courses*** table. This means that both columns must have values that exist in their respective tables.

```
CREATE TABLE WaitingList (
        student CHAR(10),
        limitedCourse CHAR(6),
        position SERIAL NOT NULL,
        CONSTRAINT status_more_than_zero CHECK(position > 0),
        UNIQUE(limitedCourse, position),
        PRIMARY KEY(student, limitedCourse),
        FOREIGN KEY(student) REFERENCES Students(idnr),
        FOREIGN KEY(limitedCourse) REFERENCES LimitedCourses(code)
);
```

The ***WaitingList*** table has four columns: ***student***, ***limitedCourse***, ***position***, and ***status_more_than_zero***. The ***student*** and ***limitedCourse*** columns are defined as CHAR(10) and CHAR(6) data types, respectively. These columns are marked as **FOREIGN KEYS** that reference the ***idnr*** and ***code*** columns of the ***Students*** and ***LimitedCourses*** tables, respectively. This means that the values in these columns must match values in the ***idnr*** and ***code*** columns of the referenced tables. The position column is defined as a **SERIAL** data type, which is an auto-incrementing integer data type. It is marked as NOT NULL, which means that it cannot contain NULL values.

The ***status_more_than_zero*** column is a **CONSTRAINT** that ensures that the position column has a value that is greater than zero. The ***WaitingList*** table has a **UNIQUE** constraint that ensures that the combination of values in the ***limitedCourse*** and ***position*** columns are unique across the table. It also has a **PRIMARY KEY** constraint that is defined on the ***student*** and ***limitedCourse*** columns, which means that the combination of values in these columns must be unique across the table.

## 2 VIEWS.SQL

```
CREATE VIEW BasicInformation AS
    SELECT stu.idnr, name, login, stu.program, branch
        FROM Students AS stu LEFT OUTER JOIN StudentBranches AS StuB
        ON (stu.idnr = StuB.student);
```

The ***BasicInformation*** view provides a view of the basic information about students, including their ID numbers, names, logins, programs, and branches. This view is created by performing a left outer join between the Students and StudentBranches tables on the student ID number. This means that the view includes all rows from the Students table, along with any matching rows from the StudentBranches table. If a student does not have a matching row in the StudentBranches table, the branch column in the view will contain NULL values.

```
CREATE VIEW FinishedCourses AS
    SELECT t.student, t.course, t.grade, c.credits
    FROM Taken AS t, Courses AS c
    WHERE (c.code = t.course);
```

The ***FinishedCourses*** view provides a view of the courses that students have completed, including the student ID, course code, grade, and number of credits. This view is created by performing a join between the Taken and Courses tables on the course code. This means that the view includes all rows from the Taken table that have a matching course code in the Courses table.

```
CREATE VIEW PassedCourses AS
    SELECT student, course, credits
    FROM FinishedCourses
    WHERE (grade<>'U');
```

The ***PassedCourses*** view provides a view of the courses that students have passed, which are defined as courses that have a grade other than 'U'. This view is created by selecting all rows from the FinishedCourses view where the grade is not equal to 'U'.

```
CREATE VIEW Registrations AS
    SELECT student, course, COALESCE(status, 'registered') AS status
    FROM (SELECT student, position, limitedCourse AS course, 'waiting' AS status
    FROM WaitingList) AS waiting NATURAL
        FULL OUTER JOIN Registered AS registerd;
```

The ***Registrations*** view provides a view of the courses that students are registered for or are waiting for. This view is created by performing a natural full outer join between the Registered and WaitingList tables. A natural full outer join combines all rows from both tables, matching rows with the same values in the join columns, and filling in NULL values for missing columns in each table. In this case, the view includes all rows from the Registered and WaitingList tables, with NULL values in the status column for rows that do not have a matching row in the other table.

```
CREATE VIEW MandatoryProgramCourses AS
    SELECT binfo.idnr AS student, mandatory.course
    FROM BasicInformation AS binfo, MandatoryProgram AS mandatory
    WHERE (binfo.program = mandatory.program);
```

The ***MandatoryProgramCourses*** view provides a view of the mandatory courses for each program. This view is created by performing an inner join between the BasicInformation and MandatoryProgram views on the program. This means that the view includes only rows where the program in the BasicInformation view matches the program in the MandatoryProgram view.

```
CREATE VIEW MandatoryBranchCourses AS
    SELECT binfo.idnr AS student, mandatory.course
    FROM BasicInformation AS binfo, MandatoryBranch AS mandatory
    WHERE binfo.branch = mandatory.branch AND binfo.program = mandatory.program;
```

The ***MandatoryBranchCourses*** view provides a view of the mandatory courses for each branch of each program. This view is created by performing an inner join between the BasicInformation and MandatoryBranch views on the branch and program. This means that the view includes only rows where the branch and program in the BasicInformation view match the branch and program in the MandatoryBranch view.

```
CREATE VIEW RecommendedBranchCourses AS
    SELECT binfo.idnr AS student, recommended.course
    FROM BasicInformation AS binfo, RecommendedBranch AS recommended
    WHERE binfo.branch = recommended.branch AND binfo.program = recommended.program;
```

The ***RecommendedBranchCourses*** view provides a view of the recommended courses for each branch of each program. This view is created by performing an inner join between the BasicInformation and RecommendedBranch views on the branch and program. This means that the view includes only rows where the branch and program in the BasicInformation view match the branch and program in the RecommendedBranch view.

```
CREATE VIEW UnreadMandatory AS
    SELECT student, course FROM MandatoryProgramCourses UNION
    SELECT student, course FROM MandatoryBranchCourses
    EXCEPT (SELECT student, course FROM PassedCourses);
```

The ***UnreadMandatory*** view provides a view of the courses that students have not yet completed, but are mandatory for their program or branch. This view is created by taking the union of the MandatoryProgramCourses and MandatoryBranchCourses views and removing any rows that are already present in the PassedCourses view. This means that the view includes only rows that are in the MandatoryProgramCourses or MandatoryBranchCourses views, but are not in the PassedCourses view.

```
CREATE VIEW ClassCredits As
    SELECT student, course, credits, classification AS class
    FROM PassedCourses AS fc
    LEFT JOIN Classified AS c USING(course);
```

The ***ClassCredits*** view provides a view of the courses that students have passed, along with their classification (e.g. math, research, seminar) and the number of credits they received for the course. This view is created by performing a left outer join between the PassedCourses and Classified views on the course. This means that the view includes all rows from the PassedCourses view, along with any matching rows from the Classified view. If a course does not have a matching row in the Classified view, the classification and class columns in the view will contain NULL values.

```sql
CREATE VIEW PathToGraduationHelper AS
WITH
    -- student
    col0 AS (SELECT idnr AS student FROM students),

    -- total credits
    col1 AS (SELECT student,SUM(credits) AS totalCredits
             FROM PassedCourses GROUP BY student),

    -- mandatory left
    col2 AS (SELECT student,COUNT(course) AS mandatoryLeft
             FROM UnreadMandatory GROUP BY student),

    -- math credits
    col3 AS (SELECT student,SUM(credits) AS credits
             FROM ClassCredits
             WHERE class = 'math'
             GROUP BY student),

    -- research credits
    col4 AS (SELECT student,SUM(credits) AS credits
             FROM ClassCredits
             WHERE class = 'research'
             GROUP BY student),

    -- seminar courses
    col5 AS (SELECT student,COUNT(*) AS course
             FROM ClassCredits
             WHERE class = 'seminar'
             GROUP BY student),

    -- recommended courses
    col6 AS (SELECT student, SUM(credits) AS credits
            FROM PassedCourses AS pc
                RIGHT JOIN RecommendedBranchCourses
                USING (course, student)
                GROUP BY student)

-- Use COALESCE to replace null values with 0.
SELECT col0.student,
       COALESCE(col1.totalCredits,0)   AS totalCredits,
       COALESCE(col2.mandatoryLeft,0)  AS mandatoryLeft,
       COALESCE(col3.credits,0)        AS mathCredits,
       COALESCE(col4.credits,0)        AS researchCredits,
       COALESCE(col5.course,0)         AS seminarCourses,
       COALESCE(col6.credits,0)        AS recommendedcredits
    FROM col0
    -- Use a chain of (left) outer joins to combine them.
    LEFT OUTER JOIN col1 ON (col0.student=col1.student)
    LEFT OUTER JOIN col2 ON (col0.student=col2.student)
    LEFT OUTER JOIN col3 ON (col0.student=col3.student)
    LEFT OUTER JOIN col4 ON (col0.student=col4.student)
    LEFT OUTER JOIN col5 ON (col0.student=col5.student)
    LEFT OUTER JOIN col6 ON (col0.student=col6.student);
```

The ***PathToGraduationHelper*** view provides a view of various information about each student's progress towards graduation. This view is created using a WITH clause, which defines several subqueries (called "common table expressions" or CTEs) that are used to compute the information for each student. The final SELECT statement then combines the information from these CTEs into a single view.

The first CTE, called **col0**, selects the ID numbers of all students from the Students table. This CTE is used as the base for the other CTEs, as it provides a list of all students that we want to include in the view.

The second CTE, called **col1**, computes the total number of credits that each student has passed by summing the credits for each course in the PassedCourses view for each student. This CTE is created by performing a GROUP BY on the student column in the PassedCourses view, which groups the rows by student and computes the sum of the credits for each group.

The third CTE, called **col2**, computes the number of mandatory courses that each student has left by counting the number of courses in the UnreadMandatory view for each student. This CTE is created by performing a GROUP BY on the student column in the UnreadMandatory view, which groups the rows by student and counts the number of courses in each group.

The fourth CTE, called **col3**, computes the number of math credits that each student has earned by summing the credits for each course in the ClassCredits view that has a classification of "math" for each student. This CTE is created by performing a GROUP BY on the student column in the ClassCredits view, which groups the rows by student and computes the sum of the credits for each group where the classification is "math".

The fifth CTE, called **col4**, computes the number of research credits that each student has earned by summing the credits for each course in the ClassCredits view that has a classification of "research" for each student. This CTE is created by performing a GROUP BY on the student column in the ClassCredits view, which groups the rows by student and computes the sum of the credits for each group where the classification is "research".

The sixth CTE, called **col5**, computes the number of seminar courses that each student has taken by counting the number of courses in the ClassCredits view that have a classification of "seminar" for each student. This CTE is created by performing a GROUP BY on the student column in the ClassCredits view, which groups the rows by student and counts the number of courses in each group where the classification is "seminar".

The seventh CTE, called **col6**, computes the number of recommended credits that each student has earned by summing the credits for each course in the PassedCourses view that is also present in the RecommendedBranchCourses view for each student. This CTE is created by performing a RIGHT OUTER JOIN between the PassedCourses and RecommendedBranchCourses views on the course and student columns. This means that the view includes all rows from the PassedCourses view, along with any matching rows from the RecommendedBranchCourses view. The CTE then performs a GROUP BY on the student column, which groups the rows by student and computes the sum of the credits for each group.

The final SELECT statement in the view combines the information from these CTEs into a single view by performing a series of left outer joins between the CTEs on the student column. This means that the view includes all rows from the col0 CTE, along with any matching rows from the other CTEs. If a student does not have a matching row in one of the other CTEs, the columns from that CTE will contain NULL (0 because of **COALESCE**) values in the view.

```
CREATE VIEW PathToGraduation AS
    SELECT student,
           totalCredits,
           mandatoryLeft,
           mathCredits,
           researchCredits,
           seminarCourses,
                 mandatoryLeft = 0              -- mandatory courses left
                 AND mathCredits >= 20          -- math credits
                 AND researchCredits >= 10      -- research credits
                 AND seminarCourses  >= 1       -- seminar curses
                 AND recommendedcredits >= 10   -- recommended credits
           AS qualified
        FROM PathToGraduationHelper;
```

The ***PathToGraduation*** view has a final SELECT statement that combines the information from the subqueries of the helper into a single view. The view includes the student ID, total number of credits the student has passed, the number of mandatory courses the student has left, the number of math credits the student has earned, the number of research credits the student has earned, the number of seminar courses the student has taken, and the number of recommended credits the student has earned. This information can be used to track each student's progress towards graduation and make decisions about what courses they should take next.

```
CREATE VIEW CourseQueuePositions AS
    SELECT limitedCourse AS course, student, position AS place
    FROM WaitingList;
```

The first view, named ***CourseQueuePositions***, defines a view that combines rows from the WaitingList table. This view is useful for tracking the position of each student in the waiting list for a given course.

```
CREATE FUNCTION on_register() RETURNS trigger AS
    $$ BEGIN
    -- Student should not already be registered
    -- Student should not already be in waiting list
    IF (SELECT COUNT(*) FROM Registrations WHERE course = NEW.course AND student =
    NEW.student) > 0 THEN
        RAISE EXCEPTION USING message='Student is already registered', errcode='23505';
    END IF;

    -- If student does not forfill all requisits, raise error
    -- prereq \ passed = Ö
    IF (SELECT COUNT(*) FROM
            ((SELECT prerequisiteCourse AS course FROM Prerequisite WHERE course =
            NEW.course) EXCEPT
            (SELECT course FROM PassedCourses WHERE student =  NEW.student)) AS foo) >
            0 THEN
        RAISE EXCEPTION USING message='The student does not meet the prerequisites of
        this course', errcode='23001';
    END IF;

    -- If student has passed the course, raise error
    IF (SELECT COUNT(*) FROM PassedCourses WHERE student = NEW.student AND course =
    NEW.course) > 0 THEN
        RAISE EXCEPTION 'Student has already taken the course';
    END IF;

    -- If course is full, add to waiting list
    IF (SELECT COUNT(*) FROM LimitedCourses WHERE code = NEW.course) > 0 AND
        (SELECT COUNT(*) FROM Registered WHERE course = NEW.course) >=
        (SELECT capacity FROM LimitedCourses WHERE code = NEW.course) THEN
        INSERT INTO WaitingList (student, limitedCourse, position) VALUES (NEW.student,
        NEW.course,
            (SELECT COALESCE(MAX(position), 0) + 1 FROM WaitingList WHERE limitedCourse
            = NEW.course));
    ELSE
        INSERT INTO Registered (student, course) VALUES (NEW.student, NEW.course);
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER on_register_trigger
```

```
    INSTEAD OF INSERT ON Registrations
    FOR EACH ROW EXECUTE PROCEDURE on_register();
```

The first trigger function, named ***on_register***, is defined to be called instead of an INSERT operation on the Registrations view. This function is executed whenever a new row is added to the Registrations view. The function first checks if the student is already registered for the course. If so, it raises an exception indicating that the student is already registered.

Next, the function checks if the student has fulfilled the prerequisites for the course by comparing the prerequisite courses for the given course with the courses that the student has passed. If the student has not fulfilled the prerequisites, the function raises an exception indicating that the student does not meet the prerequisites of the course.

The function then checks if the student has already passed the course. If so, it raises an exception indicating that the student has already passed the course.

If the course is limited and the number of registered students for the course is equal to or greater than the capacity of the course, the function adds the student to the waiting list for the course. Otherwise, it adds the student to the Registered table. In either case, the function returns the new row that was added to the view.

```
CREATE FUNCTION on_unregister() RETURNS trigger AS
    $$
    DECLARE deletedPosition INT;

    BEGIN
    -- if student is in registered -> remove from registered
    IF (SELECT COUNT(*) FROM Registered WHERE OLD.student = student AND OLD.course =
    course) = 1 THEN
        DELETE FROM Registered WHERE OLD.student = student AND OLD.course = course;
        -- check if course is limited
        IF (SELECT COUNT(*) FROM LimitedCourses WHERE OLD.course = code) != 0 THEN
            -- check if course has capacity
            IF (SELECT COUNT(*) FROM Registered WHERE OLD.course = course) < (SELECT
            capacity FROM LimitedCourses WHERE OLD.course = code) THEN
                -- check if people is in waiting list for course
                IF (SELECT COUNT(*) FROM WaitingList WHERE OLD.course = limitedCourse)
                > 0 THEN
                    -- insert first student in waiting list into registered
                    INSERT INTO Registered VALUES((SELECT student FROM WaitingList
                    WHERE OLD.course = limitedCourse AND position = 1), OLD.course);
                    -- delete first student from waiting list
                    DELETE FROM WaitingList WHERE OLD.course = limitedCourse AND
                    position = 1;
                    -- update all positions of waiting list
                    UPDATE WaitingList SET position = position - 1 WHERE OLD.course =
                    limitedCourse;
                END IF;
            END IF;
        END IF;
    -- if student in waiting list -> remove from waiting list
    ELSIF (SELECT COUNT(*) FROM WaitingList WHERE OLD.student = student AND OLD.course
    = limitedCourse) = 1 THEN
        -- save position from deleted student

        deletedPosition := (SELECT position FROM WaitingList WHERE OLD.student =
        student AND OLD.course = limitedCourse);
        DELETE FROM WaitingList WHERE OLD.student = student AND OLD.course =
        limitedCourse;
        UPDATE WaitingList SET position = position - 1 WHERE OLD.course = limitedCourse
        AND position > deletedPosition;
    END IF;

    RETURN OLD;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER on_unregister_trigger
    INSTEAD OF DELETE ON Registrations
    FOR EACH ROW EXECUTE PROCEDURE on_unregister();
```

The second trigger function, named ***on_unregister***, is defined to be called instead of a DELETE operation on the Registrations view. This function is executed whenever a row is deleted from the Registrations view. The function first checks if the student is registered for the course by checking the Registered table. If the student is registered, the function removes the row from the Registered table and checks if the course is limited and if there is room in the course. If the course has capacity and there are students on the waiting list for that course, the first student on the waiting list is registered for the course and removed from the waiting list.

# 4 PortalConnection.java

```
public String register(String student, String courseCode){
        try(PreparedStatement st = conn.prepareStatement(
                "INSERT INTO Registrations (student, course) VALUES(?, ?)"
        );) {
            st.setString(1, student);
            st.setString(2, courseCode);
            st.execute();

            if(st.getUpdateCount() == 1){
                return "{'success': true}";
            } else {
                return "{'success': false, 'error': 'developers fucked up'}";
            }
        } catch (SQLException throwables){
            return "{\"success\":false, \"error\":\"" + throwables.getMessage().
            replaceAll("\n", " ")
                    + "\"}";
        }
    }
```

The code represents a method named ***register*** that is used to register a student to a course in a database. The method takes a ***student*** and a ***courseCode*** as input and returns a string indicating the success or failure of the operation.

The method begins by preparing a SQL statement that will be used to insert a new registration record into the ***Registrations*** table in the database. The ***student*** and ***courseCode*** input parameters are used to specify the values to be inserted into the student and course columns, respectively.

The SQL statement is executed and the number of updated rows is checked. If one row was updated, the method returns a string indicating that the operation was successful. Otherwise, it returns a string indicating that the operation failed and that the developers messed up.

If an exception occurs during the execution of the SQL statement, the method catches the exception and returns a string indicating that the operation failed and providing the error message from the exception. The error message is sanitized by removing any newline characters and replacing them with spaces.

```
public String unregister(String student, String courseCode){
        try(PreparedStatement st = conn.prepareStatement(
                "DELETE FROM Registrations WHERE student=? AND course=?"
        );) {
            st.setString(1, student);
            st.setString(2, courseCode);

            st.execute();

            if (st.getUpdateCount() == 1) {
```

```
            return "{'success': true}";
        } else {
            return "{'success': false, 'error': 'developers fucked up'}";
        }
    } catch (SQLException throwables){
        return "{\"success\":false, \"error\":\"" +
        throwables.getMessage().replaceAll("\n", " ")
                + "\"}";
    }
}
```

The above code is a function in Java that removes a student from the ***Registrations*** table. It does this by using a *PreparedStatement* object that represents the *SQL DELETE* query. The *DELETE* query removes a row from the ***Registrations*** table that has the specified ***student*** and ***course*** fields. The function takes the student and course values as input parameters and sets them as values in the query using the *setString* method. The query is executed using the execute method and the function returns a JSON string indicating whether the operation was successful or not. If an exception occurs during the execution of the query, the function catches the exception and returns a JSON string with an error message.

```
public String getInfo(String student) throws SQLException{

    try{PreparedStatement stInfo = conn.prepareStatement(
        // replace this with something more useful
            "SELECT jsonb_build_object('student', student, 'name', name, 'login',
            login, 'program', program,"
                    + "                             'branch', branch, 'seminarCourses',
                    seminarCourses, 'mathCredits', mathCredits,"
                    + "                             'researchCredits', researchCredits,
                    'totalCredits', totalCredits,"
                    + "                             'canGraduate',canGraduate) AS jsondata
                    FROM"
                    + "     (SELECT idnr as student, name, login, program, branch FROM
                    BasicInformation) AS basicInfo NATURAL JOIN"
                    + "     (SELECT student, seminarCourses, mathCredits, researchCredits,
                    totalCredits, qualified AS canGraduate FROM PathToGraduation)
                    AS gradInfo"
                    + "     WHERE student=?;"
        );
        stInfo.setString(1, student);

        PreparedStatement stFinished = conn.prepareStatement(
                "SELECT jsonb_build_object('course', name, 'code', course, 'credits',
                credits, 'grade', grade) AS jsondata"
                        + "     FROM FinishedCourses WHERE student=?;"
        );

        stFinished.setString(1, student);

        PreparedStatement stRegistered = conn.prepareStatement(
                "SELECT jsonb_build_object('course', name, 'code', course, 'status',
                status, 'position', position) AS jsondata "
                        + "FROM Registrations AS r LEFT JOIN Courses AS c ON r.course
                        = c.code "
                        + "WHERE student = ?;"
        );
        stRegistered.setString(1, student);

        ResultSet rsInfo = stInfo.executeQuery();
        ResultSet rsFinished = stFinished.executeQuery();
        ResultSet rsRegistered = stRegistered.executeQuery();

        if (rsInfo.next()) {
            JSONObject json = new JSONObject(rsInfo.getString("jsondata"));
            json.put("finished", new JSONArray());
            while (rsFinished.next()) {
                json.append("finished", new JSONObject(rsFinished.getString("jsondata")));
            }

            json.put("registered", new JSONArray());
            while (rsRegistered.next()) {
                json.append("registered", new JSONObject(
                rsRegistered.getString("jsondata")));
            }

            return json.toString();
        } else {
            return "{\"student\":\"does not exist :(\"}";
        }
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        return "{'success':false, error: 'View logs to see error'}";
    }
```

The ***getInfo*** method is a function that accepts a student parameter and returns a JSON string containing information about the student. It does this by executing three prepared SQL statements: *stInfo*, *stFinished*, and *stRegistered*. These statements retrieve information about the student's basic information, finished courses, and registered courses, respectively. The results of these queries are then used to construct the final JSON object, which is returned as a string. The method also includes error handling, in case any of the SQL statements throw an exception.

```
System.out.println("----TEST2----");
          System.out.println(
                  (new JSONObject(c.register("6666666666", "CCC111")).getBoolean
                  ("success")));
          pause();
```

The code is printing the result of calling the ***register*** method from the *c* object. The register method takes in two arguments, "6666666666" and "CCC111", and returns a JSON string containing a key "success" with a corresponding Boolean value. The code is then printing the value of this "success" key by creating a new *JSONObject* from the return value of register and calling the *getBoolean* method on it with the key "success".