Lab2 - Restricciones en tablas

Pepe Calderón, Fernando Sola, Daniel Ayala, Inma Hernández, Margarita Cruz, Carlos Arévalo, David Ruiz

Escuela Técnica Superior de
Ingeniería Informática

Índice

1 Objetivo

El objetivo de esta práctica es aprender a implementar restricciones de integridad en las tablas de una base de datos. El alumno aprenderá a:

- Añadir columnas adicionales a tablas existentes con ALTER TABLE.
- Implementar restricciones CHECK para validar rangos y valores permitidos.
- Definir restricciones UNIQUE para garantizar unicidad de valores.
- Implementar tipos ENUM para atributos con valores predefinidos.
- Aplicar reglas de negocio mediante restricciones de base de datos.

En el laboratorio anterior creamos el esquema relacional básico de la base de datos GradesDB con sus tablas, claves primarias y claves ajenas. En este laboratorio completaremos el diseño añadiendo las restricciones de integridad que implementan las reglas de negocio documentadas en el laboratorio L0 (Requisitos).

2

Preparación del entorno

Abre HeidiSQL y conéctate con el usuario `iissi_user` a la base de datos `GradesDB`. Abre el archivo `createdb.sql` que creaste en el laboratorio anterior. Trabajaremos añadiendo código al final de este archivo, después de la creación de todas las tablas.

3

Control de versiones

Continuaremos trabajando con el repositorio `GradesDB` creado en L1. Tras completar cada sección de restricciones, haz commit:

```
git add -A
git commit -m "Añadidas restricciones CHECK/UNIQUE para [nombre_tabla]"
```

4

Restricciones de integridad: conceptos

Las restricciones de integridad garantizan que los datos almacenados en la base de datos cumplan ciertas reglas de negocio. SQL proporciona varios mecanismos:

- **CHECK**: Valida que un valor cumpla una expresión booleana (rangos, patrones, etc.)
- **UNIQUE**: Garantiza que no haya valores duplicados (claves alternativas)
- **ENUM**: Define un conjunto cerrado de valores permitidos para un atributo
- **NOT NULL**: Impide valores nulos (ya lo usamos en L1)

Estas restricciones se pueden definir al crear la tabla (inline) o añadirse posteriormente con `ALTER TABLE`.

5

Restricciones en la tabla `people`

La tabla `people` almacena información básica de todas las personas del sistema (tanto estudiantes como profesores). Implementaremos restricciones para validar la edad (RN12) y el formato del DNI (RN14), además de garantizar unicidad.

```
ALTER TABLE people
    ADD CONSTRAINT rn12_people_age CHECK (age BETWEEN 16 AND 70),
    ADD CONSTRAINT rn14_people_dni CHECK (dni REGEXP '^[0-9]{8}[A-Za-z]$'),
    ADD CONSTRAINT rn_uq_people_dni UNIQUE (dni),
    ADD CONSTRAINT rn_uq_people_email UNIQUE (email);
```

Observaciones:

- `CHECK (age BETWEEN 16 AND 70)` valida que la edad esté en el rango permitido.
- `REGEXP '^[0-9]{8}[A-Za-z]$'` verifica el formato del DNI:
  - `^` inicio de cadena
  - `[0-9]{8}` exactamente 8 dígitos
  - `[A-Za-z]` una letra (mayúscula o minúscula)
  - `$` fin de cadena
- El nombre de la constraint (`rn12_people_age`) facilita identificar qué regla de negocio se viola si ocurre un error.

Validación:

Ejecuta el script `createdb.sql` completo en HeidiSQL (F9) para aplicar estas restricciones. Luego prueba a violarlas:

Prueba 1: Edad fuera de rango

```
INSERT INTO people (dni, first_name, last_name, age, email, role,
password_hash)
VALUES ('99999999Z', 'Juan', 'Test', 15, 'juan@test.es', 'student',
'hash123');
```

Error esperado: `Check constraint 'rn12_people_age' is violated`

Prueba 2: DNI duplicado

Primero inserta una persona válida:

```
INSERT INTO people (dni, first_name, last_name, age, email, role,
password_hash)
VALUES ('88888888A', 'Ana', 'Test', 25, 'ana@test.es', 'student',
'hash456');
```

Ahora intenta insertar otra con el mismo DNI:

```
INSERT INTO people (dni, first_name, last_name, age, email, role,
password_hash)
VALUES ('88888888A', 'Pedro', 'Test', 30, 'pedro@test.es', 'student',
'hash789');
```

Error esperado: `Duplicate entry '88888888A' for key 'rn_uq_people_dni'`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla people"`. {: .notice–info}

6 Restricciones en la tabla `professors`

```
ALTER TABLE professors
    ADD CONSTRAINT rn20_professors_category CHECK (
        category IN ('Ayudante','AyudanteDoctor','Titular','Catedrático')
    );
```

Observaciones:

- `IN (lista)` valida que el valor esté en la lista de valores permitidos.
- Implementa el tipo enumerado `CategoríaProfesor` del modelo conceptual.
- Si se intenta insertar un valor no válido, se rechazará la operación.

Validación:

Prueba a insertar un profesor con una categoría no válida:

```
-- Primero necesitamos una persona en la tabla people
INSERT INTO people (person_id, dni, first_name, last_name, age, email,
role, password_hash)
VALUES (1000, '11111111A', 'Carlos', 'Profesor', 45, 'carlos@us.es',
'professor', 'hash');

-- Ahora intentamos crear el profesor con categoría inválida
INSERT INTO professors (professor_id, category)
VALUES (1000, 'Adjunto');
```

Error esperado: `Check constraint 'rn20_professors_category' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla professors"`.
{: .notice–info}

7

Restricciones en la tabla `students`

```
ALTER TABLE students
    ADD CONSTRAINT rn19_students_access_method CHECK (
        access_method IN
('Selectividad','Ciclo','Mayor','Titulado','Extranjero')
    );
```

Observaciones:

- Implementa el tipo enumerado `MétodoAcceso` del modelo conceptual.
- Define los cinco métodos de acceso válidos al centro universitario.

Validación:

Prueba a insertar un estudiante con método de acceso no válido:

```
-- Primero necesitamos una persona
INSERT INTO people (person_id, dni, first_name, last_name, age, email,
role, password_hash)
VALUES (2000, '22222222B', 'María', 'Estudiante', 20, 'maria@us.es',
'student', 'hash');

-- Intentamos crear estudiante con método inválido
INSERT INTO students (student_id, access_method)
VALUES (2000, 'Erasmus');
```

Error esperado: `Check constraint 'rn19_students_access_method' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla students"`. {: .notice–info}

8

Restricciones en la tabla `degrees`

```
ALTER TABLE degrees
    ADD CONSTRAINT rn13_degree_duration CHECK (duration_years BETWEEN 3 AND
6),
    ADD CONSTRAINT rn_uq_degrees_name UNIQUE (degree_name);
```

Observaciones:

- RN13 valida que la duración esté entre 3 y 6 años según la normativa universitaria.
- No puede haber dos grados con el mismo nombre.
- Evita duplicados accidentales.

Validación:

Prueba a insertar un grado con duración inválida:

```
INSERT INTO degrees (degree_name, duration_years)
VALUES ('Grado en Pruebas', 2);
```

Error esperado: `Check constraint 'rn13_degree_duration' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla degrees"`. {: .notice–info}

9

Restricciones en la tabla `subjects`

```
ALTER TABLE subjects
    ADD CONSTRAINT rn10_subjects_credits CHECK (credits IN (6, 12)),
    ADD CONSTRAINT rn16_subjects_course CHECK (course BETWEEN 1 AND 6),
    ADD CONSTRAINT ck_subjects_type CHECK (
        subject_type IN ('Formación Básica','Obligatoria','Optativa')
    ),
    ADD CONSTRAINT rn_uq_subjects_name UNIQUE (subject_name),
    ADD CONSTRAINT rn_uq_subjects_acronym UNIQUE (acronym);
```

Observaciones:

- RN10 solo permite asignaturas de 6 o 12 créditos (sistema ECTS).
- RN16 valida que el curso esté entre 1 y 6 (máxima duración de un grado).
- El constraint del tipo implementa el enumerado `TipoAsignatura` del diagrama conceptual.
- Tanto el nombre completo como el acrónimo deben ser únicos.

Validación:

Prueba a insertar una asignatura con créditos inválidos:

```
INSERT INTO subjects (degree_id, subject_name, acronym, credits, course,
subject_type)
VALUES (1, 'Asignatura Test', 'TST', 8, 1, 'Obligatoria');
```

Error esperado: `Check constraint 'rn10_subjects_credits' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla subjects"`. {: .notice–info}

10

Restricciones en la tabla `groups`

```
ALTER TABLE groups
    ADD CONSTRAINT rn15_groups_year CHECK (academic_year BETWEEN 2000 AND
2100),
    ADD CONSTRAINT ck_groups_activity CHECK (
        activity IN ('Teoría','Laboratorio')
    ),
    ADD CONSTRAINT rn_uq_groups_name UNIQUE (subject_id, group_name,
academic_year);
```

Observaciones:

- RN15 establece un rango temporal razonable para los años académicos.
- El constraint de actividad implementa el enumerado `TipoActividad` del modelo conceptual.
- La clave alternativa compuesta permite grupos con el mismo nombre en diferentes asignaturas o años:
  - ✅ IISSI-1, T1, 2024
  - ✅ IISSI-1, T1, 2025 (mismo grupo, diferente año)
  - ✅ ADDA, T1, 2024 (mismo nombre, diferente asignatura)
  - ✗ IISSI-1, T1, 2024 (duplicado exacto)

Validación:

Prueba a insertar un grupo con año académico fuera de rango:

```
INSERT INTO groups (subject_id, group_name, activity, academic_year)
VALUES (1, 'T1', 'Teoría', 1999);
```

Error esperado: `Check constraint 'rn15_groups_year' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla groups"`. {: .notice–info}

11

Restricciones en la tabla `teaching_loads`

```
ALTER TABLE teaching_loads
    ADD CONSTRAINT rn21_teaching_loads_credits CHECK (credits > 0);
```

Observaciones:

- RN21 garantiza que las cargas docentes sean positivas.
- Un profesor debe impartir al menos algún crédito en un grupo.

Validación:

Prueba a insertar una carga docente con créditos cero o negativos:

```
INSERT INTO teaching_loads (professor_id, group_id, credits)
VALUES (1, 1, 0);
```

Error esperado: `Check constraint 'rn21_teaching_loads_credits' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricc`
{: .notice–info}

12

Restricciones en la tabla `grades`

```
ALTER TABLE grades
    ADD CONSTRAINT rn11_grades_value CHECK (grade_value BETWEEN 0 AND 10),
    ADD CONSTRAINT rn08_grades_with_honors CHECK (
        with_honors = 0 OR grade_value >= 9
    ),
    ADD CONSTRAINT rn18_grades_exam_call CHECK (
        exam_call IN ('Primera','Segunda','Tercera','Extraordinaria')
    );
```

Observaciones:

- RN11 valida que las notas estén en el rango estándar (0 a 10).
- RN08 implementa la regla: si `with_honors = 1`, entonces `grade_value >= 9`.
  - La expresión `with_honors = 0 OR grade_value >= 9` se cumple si:
    - No es matrícula de honor (with_honors = 0), o
    - Si es matrícula de honor, la nota es >= 9
- RN18 implementa el enumerado `Convocatoria` del modelo conceptual.

Validación:

Prueba a insertar una nota con matrícula de honor pero valor inferior a 9:

```
INSERT INTO grades (student_id, group_id, grade_value, exam_call,
with_honors)
VALUES (1, 1, 8.5, 'Primera', 1);
```

Error esperado: `Check constraint 'rn08_grades_with_honors' is violated`

Recuerda: al finalizar este apartado, haz commit: `git add -A && git commit -m "Añadidas restricciones para tabla grades"`. {: .notice–info}

13

Modificación y borrado de datos: UPDATE y DELETE

Hasta ahora hemos trabajado con `INSERT` para añadir datos. SQL también proporciona `UPDATE` para modifica filas existentes y `DELETE` para eliminar filas.

13.1. Actualizar datos con UPDATE

La sintaxis básica es:

```
UPDATE nombre_tabla
SET columna1 = valor1, columna2 = valor2, ...
WHERE condición;
```

Si omites la cláusula `WHERE` , se modificarán **TODAS** las filas de la tabla.

Ejemplo 1: Cambiar el email de una persona

```
UPDATE people
SET email = 'nuevo.email@us.es'
WHERE person_id = 1;
```

Ejemplo 2: Incrementar la duración de todos los grados en 1 año

```sql
UPDATE degrees
SET duration_years = duration_years + 1;
```

Ejemplo 3: Cambiar la categoría de varios profesores

```sql
UPDATE professors
SET category = 'Titular'
WHERE category = 'AyudanteDoctor';
```

13.2. Eliminar datos con DELETE

La sintaxis básica es:

```sql
DELETE FROM nombre_tabla
WHERE condición;
```

Si omites la cláusula `WHERE`, se borrarán **TODAS** las filas de la tabla.

Ejemplo 1: Borrar una persona específica

```sql
DELETE FROM people
WHERE person_id = 100;
```

Ejemplo 2: Borrar todos los grupos de teoría

```sql
DELETE FROM groups
WHERE activity = 'Teoría';
```

Ejemplo 3: Borrar notas de suspenso

```sql
DELETE FROM grades
WHERE grade_value < 5;
```

14

Comportamiento de borrado en claves ajenas

Cuando intentamos borrar una fila que es referenciada por claves ajenas en otras tablas, pueden surgir conflictos. Por ejemplo, si intentamos borrar una asignatura que tiene grupos asociados, ¿qué debe suceder con esos grupos?

SQL proporciona varias opciones para manejar este comportamiento mediante la cláusula `ON DELETE`:

Opción	Comportamiento
`RESTRICT`	**Impide el borrado** si existen referencias no se permite el borrado.
`CASCADE`	**Borra en cascada**: elimina automáticamente las filas que referencian la fila borrada.
`SET NULL`	**Establece a NULL** la clave ajena. Solo posible si la columna permite `NULL`.
`SET DEFAULT`	**Establece al valor por defecto** especificado con `DEFAULT`.

14.1. Ejemplo práctico: Borrado en cascada para subjects → groups

En nuestro modelo, existe una relación de **composición** entre `subjects` y `groups`: cuando se elimina una asignatura, también deben eliminarse todos sus grupos (no tiene sentido mantener grupos sin asignatura).

Para implementar esto, modificamos la declaración de la clave ajena en `groups`:

```
FOREIGN KEY (subject_id) REFERENCES subjects(subject_id) ON DELETE CASCADE
```

Comportamiento resultante:

- Si borramos una asignatura: `DELETE FROM subjects WHERE subject_id = 5;`
- Automáticamente se borrarán todos los grupos asociados: `groups.subject_id = 5`
- También se borrarán las matrículas en esos grupos (`group_enrollments`)
- Y las cargas docentes (`teaching_loads`)
- Y las notas (`grades`)

14.2. Ejemplo: RESTRICT (comportamiento por defecto)

Si intentamos borrar un grado que tiene asignaturas:

```
DELETE FROM degrees WHERE degree_id = 1;
```

Error: `Cannot delete or update a parent row: a foreign key constraint fails`

Para poder borrarlo, primero habría que borrar todas las asignaturas de ese grado, o cambiar la clave ajena a `ON DELETE CASCADE`.

14.3. Ejemplo: SET NULL

Si una relación es opcional (multiplicidad 0..1), podemos usar `SET NULL`:

```
FOREIGN KEY (degree_id) REFERENCES degrees(degree_id) ON DELETE SET NULL
```

En este caso, si se borra el grado, las asignaturas quedarían con `degree_id = NULL` (sin grado asignado). En nuestro modelo, `degree_id` es `NOT NULL`, por lo que no podemos usar `SET NULL`. Solo `RESTRICT` o `CASCADE` son válidos.

14.4. Modificar createDB.sql para añadir ON DELETE CASCADE

Actualiza la tabla `groups` en tu archivo `createDB.sql`:

Cambio a realizar:

```
-- ANTES
FOREIGN KEY (subject_id) REFERENCES subjects(subject_id)

-- DESPUÉS
FOREIGN KEY (subject_id) REFERENCES subjects(subject_id) ON DELETE CASCADE
```

Guarda los cambios y ejecuta nuevamente el script completo para recrear la base de datos con la nueva configuración.

```
git add -A
git commit -m "Añadido ON DELETE CASCADE para subjects->groups"
```

15

Push final

Una vez completado todo el laboratorio, incluyendo todas las restricciones y comportamientos de borrado, realiza el push final al repositorio remoto:

```
git push
```

Verifica que todos los cambios se han subido correctamente accediendo a GitHub y comprobando que el archivo `createDB.sql` está actualizado con:

> **Info**
>
> Versión PDF disponible