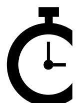
Programació amb bases de dades

PID_00261969

Ignasi Lorente Puchades

Temps mínim de dedicació recomanat: 3 hores



Universitat
Oberta
de Catalunya

Ignasi Lorente Puchades

Enginyer superior en Informàtica per la Universitat Oberta de Catalunya (UOC). Exerceix de cap de projectes d'aplicacions web en l'àmbit públic i privat. Professor col·laborador del Grau Multimèdia a la Universitat Oberta de Catalunya (UOC) i professor associat a la Universitat Pompeu Fabra (UPF).

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats per la professora: Àngels Rius Gavidia (2019)

Primera edició: febrer 2019
© Ignasi Lorente Puchades
Tots els drets reservats
© d'aquesta edició, FUOC, 2019
Av. Tibidabo, 39-43, 08035 Barcelona
Disseny: Manel Andreu
Realització editorial: Oberta UOC Publishing, SL

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció 5

Objectius 6

1. Estructura d'un SGBD relacional 7
1.1. Usuaris de la base de dades 8
1.2. Els esquemes d'una base de dades 9

2. Estructures de programació en SQL 13
2.1. Vistes en SQL 14
2.1.1. Sintaxi 15
2.1.2. Avantatges en l'ús de les vistes 17
2.2. Els procediments emmagatzemats 17
2.2.1. Sintaxi 18
2.2.2. Paràmetres d'entrada i de sortida 20
2.2.3. Variables dins del codi 20
2.2.4. Sentències condicionals 21
2.2.5. Sentències iteratives i ús de cursors de dades 23
2.3. Els disparadors (*triggers*) 25
2.3.1. Sintaxi 26

3. Concurrència i transaccions 30
3.1. Nivell de concurrència 32
3.2. Sintaxi 32
3.3. Responsabilitats de l'SGBD i del desenvolupador 33

Resum 34

Activitats 35

Glossari 37

Introducció

En aquest mòdul didàctic aprendrem a accedir a una base de dades relacional des d'un programa d'aplicació mitjançant el llenguatge estàndard d'accés a bases de dades relacionals, el llenguatge SQL (de l'anglès *structured query language*). Concretament veurem com fer l'accés a la base de dades des de l'entorn web.

Ampliarem els coneixements vistos sobre SQL. En primer lloc, veurem quines sentències de l'SQL permeten descriure la base de dades i definir-ne els usuaris, assignar permisos d'accés i fer consultes. També veurem quines són les diferents estructures de programació que l'SQL ofereix al desenvolupador per facilitar-li l'accés a les dades des del programa. Finalment, estudiarem de quina manera s'executen aquestes operacions, tenint en compte el nombre d'usuaris que vulguin accedir a les mateixes dades simultàniament.

Abans de començar a estudiar l'SQL convé tenir clara la notació que farem servir:

- Tota sentència SQL acaba amb punt i coma.
- Cada sentència pot estar formada per una o més clàusules.
- Hi ha clàusules opcionals i obligatòries.
- Una clàusula opcional s'indica entre claudàtors.
- Una clàusula que permeti triar entre diverses opcions s'escriu separant cada opció per una barra vertical. Per exemple *A*|*B*|*C* on A, B i C són opcions.
- Les paraules reservades s'escriuen en majúscules.
- Els valors a informar per l'usuari s'indiquen entre els símbols menor (<) i major (>).

Objectius

En els materials didàctics d'aquesta unitat trobareu les eines indispensables per a assolir els objectius següents:

1. Entendre l'estructura d'organització d'una base de dades.

2. Comprendre la necessitat de generar codi d'aplicació per a accedir a la base de dades des d'un entorn web, tenint en compte el sistema de gestor de bases de dades i l'entorn de programació triats.

3. Conèixer els mecanismes que ofereix una base de dades per tal d'encapsular dades amb vistes, permetent l'accés a les dades de manera anàloga en les taules.

4. Conèixer els mecanismes que ofereix una base de dades per tal de combinar diferents operacions SQL: disparadors i procediments emmagatzemats.

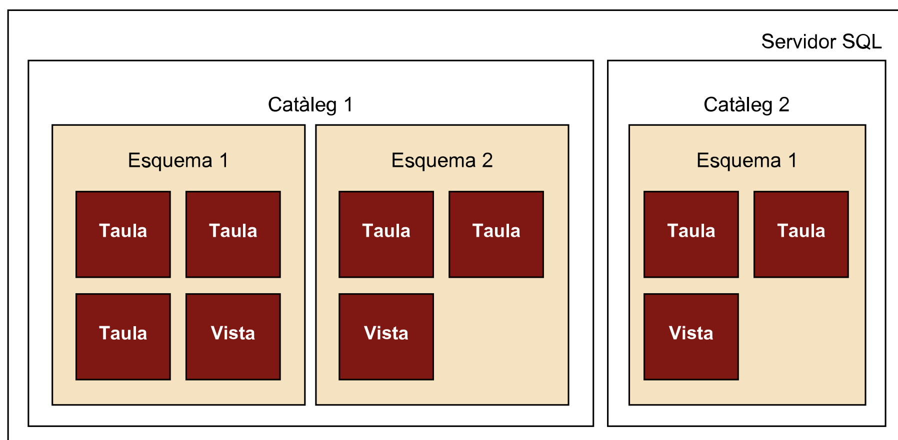
5. Saber com es gestionen les transaccions i com es facilita la concurrència d'accés a les dades des d'un SGBD.

1. Estructura d'un SGBD relacional

Fins ara hem vist que en una base de dades relacional els elements que permeten emmagatzemar la informació són les taules. Tot i això, hem de tenir en compte que un SGBD podrà gestionar més d'una base de dades, i que caldrà veure com s'organitzaran les taules i les vistes dins d'un SGBD per tal de diferenciar-les d'altres taules i de vistes d'altres bases de dades.

Un SGBD que suporti SQL estarà format per una sèrie d'elements estructurats de manera jeràrquica:

- Servidor: sistema informàtic que gestionarà els diferents catàlegs de dades.
- Catàleg: component que conté un conjunt d'esquemes de bases de dades.
- Esquema: component que permet agrupar un conjunt de taules, vistes i altres elements del model lògic d'una base de dades que són propietat d'un usuari.



D'acord amb això veiem que les bases de dades venen descrites pels seus esquemes i que diversos esquemes poden formar part d'un mateix catàleg. Aquests catàlegs són propietat d'algun usuari del servidor i pot donar accés a altres usuaris. Hi ha, però, un usuari que gestiona el conjunt de catàlegs i té el control total de la informació continguda en el servidor: l'administrador de la base de dades.

Si bé el llenguatge SQL ofereix sentències per a la creació de la base de dades, les seves taules i vistes, cadascun dels diferents SGBD té accions pròpies per a la creació i gestió del servidor, així com dels catàlegs.

1.1. Usuaris de la base de dades

En una base de dades hi ha diferents rols i usuaris amb privilegis, que els permeten accedir a l'estructura de la base de dades. A més, els usuaris accedeixen a les dades en funció dels permisos que tenen. Així doncs, aquests permisos determinen la visibilitat dels diferents objectes de la base de dades.

L'**usuari administrador** (DBA, en anglès) és l'usuari que gestionarà els diferents elements de la base de dades, així com els altres usuaris.

Aquest rol serà de vital importància tant per a la instal·lació de la base de dades com per al seu manteniment en el dia a dia, tant si és controlant l'espai d'emmagatzematge com la modificació de l'estructura de la base de dades, a criteri dels desenvolupadors.

Serà l'usuari administrador l'encarregat de fer còpies de seguretat i, en cas que calgui, restaurar una versió estable i generar els reports necessaris per tal de tenir sempre informat el propietari de la base de dades.

No estarà entre els seus rols establir el disseny de la base de dades, que serà una de les atribucions de l'usuari desenvolupador, ja que és qui tindrà el coneixement dels requisits funcionals de l'aplicació a desenvolupar.

L'**usuari desenvolupador** de la base de dades és qui defineix l'estructura de la base de dades, d'acord amb els requisits de l'aplicació a què es vol donar servei.

A més, l'usuari administrador serà l'encarregat de dotar o de revocar els permisos (GRANTS) als diferents usuaris de la base de dades.

Aquesta és la sentència d'assignació de privilegis:

```
GRANT <privilegi>
ON <objecte>
TO {<user> | PUBLIC | <rol>}
[ WITH GRANT OPTION ];
```

Veiem alguns dels privilegis que es poden assignar als usuaris de la base de dades:

- **CREATE object**: permet als usuaris crear o modificar l'objecte especificat en el seu esquema.
- **CREATE ANY object**: permet als usuaris crear o modificar l'objecte especificat en qualsevol esquema.
- **INSERT**: permet als usuaris inserir files en una taula.
- **SELECT**: permet als usuaris seleccionar dades d'un objecte de base de dades.
- **UPDATE**: permet a l'usuari actualitzar el contingut de les columnes de les files d'una taula.
- **DELETE**: permet eliminar files d'una base d'una taula.
- **EXECUTE**: permet a l'usuari cridar procediments emmagatzemats o funcions.

Privilegi CREATE

Aquest privilegi permetrà a l'usuari o rol especificat executar sentències CREATE, ALTER o DROP sobre objectes de la base de dades.

Per tal de revocar privilegis, utilitzarem la sentència REVOKE, amb la sintaxi següent:

```
REVOKE <privilegi>
ON <objecte>
FROM {<user> | <PUBLIC> | <rol>};
```

Tanmateix, serà l'usuari administrador l'encarregat de crear i de mantenir els diferents rols de la base de dades, i els rols assignats a cada usuari.

La sintaxi per a crear rols serà:

```
CREATE ROLE <rol>
[IDENTIFIED BY <password>];
```

Per exemple, per a crear un rol anomenat «desenvolupador» amb la contrasenya com a «pwd», el codi serà el següent:

```
CREATE ROLE desenvolupador
IDENTIFIED BY pwd;
```

Tot seguit, si es vol assignar el privilegi CREATE TABLE al rol «desenvolupador», i assignar aquest rol a un usuari «user1», utilitzarem el codi següent:

```
GRANT CREATE TABLE TO desenvolupador;
GRANT desenvolupador TO user1;
```

1.2. Els esquemes d'una base de dades

Un esquema és la descripció d'un objecte d'una base de dades que ens permet definir-ne l'estructura; és a dir, el conjunt de taules, vistes, índexs, tipus, seqüències i regles d'integritat que serveixen per assegurar la coherència de les dades emmagatzemades, entre altres objectes.

Per a crear un esquema, utilitzarem la sentència CREATE SCHEMA, que té la següent sintaxi:

```
CREATE SCHEMA <schema_name> [AUTHORIZATION <owner_name>]
```

```
[DEFAULT CHARACTER SET <char_set_name>]
[PATH <schema_name[, ...]>]
[ ANSI CREATE <statements [...]> ]
[ ANSI GRANT <statements [...]> ];
```

Com veiem en aquesta instrucció, es pot indicar en diferents clàusules: el nom del propietari de l'esquema (owner_name), la codificació dels caràcters amb què es treballa (char_set_name) —en cas que se'n vulgui una diferent de la instal·lada en el servidor— i el fitxer físic en el qual s'emmagatzema la informació de l'esquema (schema_name). També és possible indicar quines components (taules, vistes, índexs...) es volen crear, així com assignar permisos a diferents usuaris de la base de dades, sigui per llegir, escriure, etc.

Finalment, amb la clàusula GRANT podrem assignar permisos de lectura, d'escriptura, d'accés, etc. als diferents usuaris de la base de dades.

En la creació de l'esquema és important adonar-se que cal que s'informin les sentències de creació d'objectes de la base de dades com les taules, vistes, etc. D'aquesta manera, la creació i la modificació dels elements de l'esquema es podrà fer *a posteriori*.

Com que en un catàleg hi ha més d'un esquema de bases de dades, cal seleccionar-lo per a operar amb els elements. Això ho farem amb la clàusula SET SCHEMA.

```
SET SCHEMA <schema_name>
```

Per exemple, si volem crear un esquema anomenat «UNIVERSITAT» per a desar informació d'estudiants, podem executar les sentències SQL següents:

```
CREATE SCHEMA UNIVERSITAT AUTHORIZATION 'admin';
SET SCHEMA UNIVERSITAT;
CREATE TABLE Estudiant (
   id INT,
   nom VARCHAR (255),
   cognoms VARCHAR (255)
);
```

En aquest exemple veiem com es crea un esquema de nom «UNIVERSITAT», que pertany a l'usuari «admin». Tot seguit, se selecciona com l'esquema de treball («UNIVERSITAT») i, finalment, es crea una taula («ESTUDIANT»).

L'esquema, com qualsevol altre element d'SGBD descrit per alguna sentència de definició en SQL, no és inalterable i es podrà modificar i esborrar (sempre que l'usuari tingui permisos) amb les sentències ALTER SCHEMA i DROP SCHEMA.

En el cas de la modificació de l'esquema, la sintaxi serà:

```
ALTER SCHEMA <schema_name> [RENAME TO <new_schema_name>] [OWNER TO <new_user_name>];
```

Com veiem, amb la sentència ALTER SCHEMA, podrem modificar tant el nom de l'esquema com el nom de l'usuari que n'és propietari.

Per a eliminar un esquema, la sintaxi serà:

```
DROP SCHEMA <schema name> [RESTRICT | CASCADE};
```

En aquest cas, trobarem dues opcions, RESTRICT i CASCADE, que ens impediran, respectivament, eliminar l'esquema només si és buit o eliminar-lo en qualsevol cas, destruint-ne també el contingut.

Tal com hem vist, tot esquema té un nom i pot tenir un propietari, uns usuaris amb permisos d'accés, unes taules, etc. Aquesta informació es desarà en una estructura de dades a banda de l'esquema de la base de dades, la qual contindrà únicament dades orientades a la gestió.

Així doncs, a més dels esquemes creats pels diferents usuaris del servidor SQL, que seran accessibles als usuaris que en tinguin permís, cal un esquema que serveixi per organitzar i mantenir aquesta metainformació, a la qual només podrà accedir l'administrador de la base de dades.

Aquestes sentències ens serviran per a connectar amb un servidor i un esquema, des de la línia de comandes, per tal de poder executar operacions. Cada SGBD tindrà una implementació diferent de les sentències de creació i de connexió en un esquema propi d'un servidor.

Els diferents llenguatges de programació d'alt nivell ens oferiran idealment una única interfície que permetrà accedir i executar sentències en una base de dades independentment de l'SGBD.

Per a assolir aquest objectiu, habitualment cada llenguatge de programació proporcionarà uns connectors (*drivers*) que permetran que aquestes tasques siguin transparents per al desenvolupador. Hi ha d'altres llenguatges de programació, com PHP, que tindran una biblioteca pròpia per a cada base de dades. No obstant això, el llenguatge PHP també ofereix una biblioteca pròpia, anomenada PDO (PHP Data Object), que permet connectar amb qualsevol base de dades a partir d'un connector.

Habitualment, la connexió a una base de dades es fa mitjançant un URL, en el qual s'especifica el protocol de connexió, l'SGBD a què es vol connectar, l'adreça IP (o nom de domini), el port d'accés i el nom de l'esquema o de la base de dades amb què es vol connectar. En aquest URL es pot indicar, també, les credencials d'accés de l'usuari.

L'URL serà de la forma

```
jdbc:[<subprotocol>]:[<node>]/[<databaseName>]
```

On el subprotocol farà referència al tipus d'SGBD o especificació (Oracle, MySQL, ODBC), i el node farà referència a l'adreça del servidor.

Per exemple, si volem connectar a una base de dades Oracle des de Java, executarem una instrucció de l'estil:

```
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","user","password");
```

En cas que es desitgi connectar des de Java amb una base de dades MySQL, la instrucció serà de l'estil:

```
Connection con = DriverManager.getConnection
("jdbc:mysql://localhost/mydb?user=user&password=password");
```

Si, per contra, volem connectar a una base de dades Oracle des de PHP, executarem una instrucció de l'estil:

```
$con = oci_connect('user', 'password', 'localhost/xe');
```

En cas que es desitgi connectar des de PHP amb una base de dades MySQL, la instrucció serà de l'estil:

```
$con = mysqli_connect("localhost", "user", "password", "mydb");
```

En cas que es desitgi connectar des de PHP a partir d'un connector, la instrucció serà de l'estil:

```
$con = new PDO('mysql:dbname=mydb;host=localhost', 'user', 'password');
```

Una vegada connectats a la base de dades, un llenguatge de programació d'alt nivell permetrà realitzar operacions simples desant la sortida de cada operació per a utilitzar-la com a paràmetre de l'operació següent o, fins i tot, fer altres tipus d'operacions sobre el resultat per a utilitzar-lo com a entrada d'altres operacions.

Així, doncs, les instruccions que permetran el control del flux en els llenguatges de programació possibilitaran la concatenació de sentències SQL en bucles, sentències condicionals, etc.

2. Estructures de programació en SQL

L'SQL ens ofereix diferents formes d'encapsular les dades per a facilitar-ne l'accés, ja sigui mitjançant la generació de subconjunts de dades o altres mecanismes que permetin la combinació i concatenació de sentències de manera transparent al desenvolupador.

Primer veurem les vistes, un objecte de la base de dades que facilita l'accés a subconjunts de dades com si estiguessin emmagatzemats en una taula. Després, els procediments emmagatzemats i/o disparadors, també coneguts aquests darrers com a *triggers*. Ambdós son mecanismes de programació que utilitzen estructures algorísmiques per al control de flux. Difereixen en la forma com s'activa la seva execució, ja sigui sota demanda de l'usuari o de forma autònoma en produir-se certes circumstàncies especificades prèviament pel programador.

Com que la sintaxi de les sentències SQL per a les vistes, les funcions i els procediments emmagatzemats depèn de cada sistema gestor de base de dades, n'haurem de triar un per anar estudiant i provant els exemples.

El nostre SGBD de referència serà MySQL i la base de dades de la qual partirem i que anirem ampliant a mesura que necessitem serà BD_ESTUDIANT, formada per les taules següents:

- ESTUDIANT (codiMatricula, nom, cognoms, estudis, percentatge)
- ESTUDIANT_DE_GRAU (codiMatricula, nom, cognoms, percentatge)

L'extensió de totes dues taules és:

ESTUDIANT

codiMatricula	nom	cognoms	estudis	percentatge	notaLletra
1	Joan	Pi Dot	Grau	0	NULL
2	Laura	Sentís Aguilar	Màster	0	NULL
3	Roc	Sánchez Gómez	Grau	0	NULL
4	Joana	Sauler Sunyer	Grau	0	NULL

ESTUDIANTS_DE_GRAU

codiMatricula	nom	cognoms	percentatge	notaFinal
1	Joan	Pi Dot	0	9
3	Roc	Sánchez Gómez	0	8
4	Joana	Sauler Sunyer	0	7

I les instruccions del DDL de creació i del DML per a inserir les dades són les següents:

```
CREATE TABLE ESTUDIANT
(
   codiMatricula int,
   nom varchar(50),
   cognoms varchar(50),
   estudis varchar(50),
   percentatge numèric(5,2),
   notaLletra char(2),
   primary key (codiMatricula)
);

INSERT INTO ESTUDIANT VALUES (1,"Joan","Pi Dot","Grau", 0, null);
INSERT INTO ESTUDIANT VALUES (2,"Laura","Sentís Aguilar","Màster", 0, null);
INSERT INTO ESTUDIANT VALUES (3,"Roc","Sánchez Gómez","Grau", 0, null);
INSERT INTO ESTUDIANT VALUES (4,"Joana","Sauler Sunyer","Grau", 0, null);

CREATE TABLE ESTUDIANT_DE_GRAU AS
SELECT codiMatricula, nom, cognoms, percentatge
FROM ESTUDIANT
WHERE estudis = "Grau";

ALTER TABLE ESTUDIANT_DE_GRAU ADD notaFinal integer;
UPDATE ESTUDIANT_DE_GRAU SET notaFinal = 9 WHERE codiMatricula = 1;
UPDATE ESTUDIANT_DE_GRAU SET notaFinal = 8 WHERE codiMatricula = 3;
UPDATE ESTUDIANT_DE_GRAU SET notaFinal = 7 WHERE codiMatricula = 4;
```

2.1. Vistes en SQL

Fins ara hem considerat que l'accés a la informació emmagatzemada en una base de dades es fa a partir de consultes contra les taules que la formen.

Tot i això, hi ha una estructura, anomenada *vista*, que ens permetrà accedir a la informació present en una o més taules com si estigués emmagatzemada en una taula virtual.

Una **vista** és un conjunt de dades, resultat d'una consulta, al qual es pot accedir com si fos una taula.

Les vistes no seran part de l'esquema físic de la base de dades, i el seu contingut es calcularà sempre en el moment en què s'hi accedeixi. Les vistes no existeixen realment com un conjunt de valors emmagatzemats en la BD, sinó que es comporten com a taules fictícies, també anomenades derivades (no materialitzables) i que es construeixen a partir de valors reals emmagatzemats en la BD.

Quan s'actualitza una taula, tant si és inserint noves tuples o modificant les seves dades, la nova informació ha de reflectir-se en les taules que formen part de la consulta que serveix per a definir-la.

Per aquest motiu, només es podran actualitzar les vistes que compleixin diverses condicions:

- La vista ha d'incloure totes les claus primàries de les taules que utilitza la consulta que la defineix.

- La vista ha d'incloure tots els camps, que no poden tenir valor nul, de les taules que utilitza la consulta que la defineix.

En aquest cas, és possible fer el que s'anomena *reverse-mapping*, un mapeig entre vista-taules i taules-vista, és a dir un mapeig en ambdós sentits. En cas contrari, no es podrà fer l'operació, ja que es violarien les regles d'integritat del model relacional.

2.1.1. Sintaxi

Per a crear una vista, utilitzarem la sintaxi bàsica següent:

```
CREATE VIEW <nom_vista> [(<columnes>)]
AS <expressió_consulta>
[WITH CHECK OPTION];
```

L'expressió de consulta serà una sentència SELECT, que es pot fer sobre una o més taules.

A partir de la base de dades d'exemple BD_ESTUDIANT, es vol crear una vista per a obtenir les notes dels estudiants de grau, ja que és una consulta que es fa molt sovint.

```
CREATE VIEW NotesEstudiants (matrícula, nota, qualificació) as
(
   SELECT e.codiMatricula, eg.notaFinal, e.notaLletra
   FROM ESTUDIANT e INNER JOIN ESTUDIANT_DE_GRAU eg
ON e.codiMatricula = eg.codiMatricula
);
```

Observem que és una vista que accedeix a columnes de diferents taules.

Si l'executem mitjançant aquesta consulta SQL:

```
SELECT * FROM NotesEstudiants;
```

Obtindríem l'extensió següent:

EXTENSIÓ DE LA VISTA:

matrícula	nota	qualificació
1	9	Null
3	8	Null
4	7	Null

Per a esborrar una vista, utilitzarem la sentència DROP:

```
DROP VIEW <nom_vista> {RESTRICT|CASCADE};
```

Si utilitzem l'opció RESTRICT, la vista no s'esborrarà si està referenciada, per exemple, per una altra vista. En canvi, si posem l'opció CASCADE, tot el que referenciï la vista s'esborrarà.

Si volguéssim esborrar la vista que hem creat, hauríem d'executar:

```
DROP VIEW NotesEstudiants;
```

Els resultats de les vistes poden actualitzar les taules involucrades sempre que la vista contingui la clau primària, però, per fer-ho, cal definir la vista amb la clàusula WITH CHECK OPTION.

Creem la vista actualitzable que retorna les notes dels estudiants de grau.

```
CREATE VIEW NotesEstudiants (matrícula, nota, qualificació) as
(
   SELECT e.codiMatricula, eg.notaFinal, e.notaLletra
   FROM ESTUDIANT e INNER JOIN ESTUDIANT_DE_GRAU eg
WHERE e.codiMatricula = eg.codiMatricula
) WITH CHECK OPTION;
```

Imaginem que volem modificar la nota d'un estudiant de grau del qual coneixem el número de matrícula. Per a fer-ho, executaríem les sentències següents:

```
UPDATE NotesEstudiants
SET nota = 8
WHERE matrícula = 1;

SELECT * FROM  NotesEstudiants;
```

L'extensió de la vista es veuria modificada així:

EXTENSIÓ DE LA VISTA:

matrícula	nota	qualificació
1	8	Null
3	8	Null
4	7	Null

Finalment, si es vol modificar l'estructura d'una vista, utilitzarem la sentència ALTER VIEW:

```
ALTER VIEW <nom_vista> [(columnes)]
AS <expressió_consulta>
[WITH [RESTRICT|CASCADE] CHECK OPTION];
```

2.1.2. Avantatges en l'ús de les vistes

Les vistes poden proporcionar certs avantatges sobre les taules:

- Poden representar un subconjunt de les dades contingudes en una taula, limitant el grau d'exposició de les dades i permetent que un usuari determinat tingui permís per a consultar la vista, mentre que es denega l'accés a la resta de la taula base.

- Permeten unir i simplificar múltiples taules en una única taula virtual. A més, permeten crear subconjunts que simplifiquen la complexitat de les dades.

- Poden actuar com a taules agregades, en les quals el motor de base de dades agrega dades (suma, mitjana, etc.) i presenta els resultats calculats com a part de les dades.

- Ocupen molt poc espai de disc, de manera que la base de dades només conté la definició d'una vista i no una còpia de totes les dades que presenta.

2.2. Els procediments emmagatzemats

Els procediments emmagatzemats permetran aplicar la metodologia algorítmica en les consultes realitzades sobre una base de dades.

Tot i que hem vist que és possible realitzar algunes consultes complexes combinant sentències SQL i/o fent servir subconsultes, hi ha certes limitacions per a combinar sentències SQL. Per exemple, podem fer insercions de dades a par-

tir dades recuperades en una consulta, però no podem crear una taula amb noms de columna obtinguts com a resultat d'una consulta. Per a fer-ho, cal un pas intermedi per a emmagatzemar els valors recuperats en la consulta en variables per tal que després es puguin fer servir.

Per a combinar diverses sentències SQL com si fos una sola, podem fer servir els procediments emmagatzemats.

Un **procediment emmagatzemat** és una funció definida per un usuari de la base de dades que proporciona un servei determinat. El procediment s'emmagatzemarà en la base de dades i es tractarà com un objecte més.

Els procediments emmagatzemats es podran executar sota petició expressa d'un usuari des de línia de comandes, o bé des d'una aplicació que accedeixi a la base de dades.

L'ús de procediments emmagatzemats ens permetrà agrupar operacions que tenen un objectiu algorítmic comú i, d'aquesta manera, simplificar el desenvolupament d'aplicacions. A més, com que funciona com una caixa negra, el desenvolupador que invoqui el procediment no necessitarà conèixer quines operacions es duen a terme dins del procediment.

Una altra avantatja dels procediments emmagatzemats és que es guarden precompilats en la base de dades pel que milloren el rendiment de l'aplicació que els invoca.

2.2.1. Sintaxi

Per a crear un procediment, utilitzarem la sintaxi bàsica següent:

```
CREATE {PROCEDURE | FUNCTION} <nom_procediment>
AS
<sentències_sql>
END;
```

I aquesta serà la manera com s'invocarà:

```
CALL <nom_procediment>;
```

Crearem un procediment per a crear una nova taula amb les dades d'estudiants de grau, ESTUDIANT_DE_GRAU, concretament amb el nom i cognom de cadascun d'ells.

```
CREATE TABLE ESTUDIANTS_DE_GRAU AS
SELECT nom, cognoms
FROM ESTUDIANTS
WHERE tipusEstudi = 'Grau';
```

Observem que esta consulta ens permet crear una nova taula a partir de valors de columnes existents en una altra taula. No obstant, cal adonar-nos que no podríem crear una taula amb un número de columnes variable o amb noms de columna obtinguts a partir de la taula origen.

Si es vol eliminar el procediment, utilitzarem la sentència DROP:

```
DROP PROCEDURE <nom_procediment>;
```

Finalment, si es vol modificar un procediment, tindrem l'opció d'eliminar el procediment per tal de crear-lo de nou, o bé utilitzar la sentència ALTER:

```
ALTER {PROCEDURE | FUNCTION} <nom_procediment>
[( {<parameter_name datatype> }[, ...] )]
[NAME <new_object_name>]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT][DYNAMIC RESULT SETS INT]
[CASCADE | RESTRICT]
```

Tot i això, haurem de tenir en compte que els processos emmagatzemats tindran una sintaxi final diferent per a cada SGBD de manera que, en canviar d'una base de dades a una altra, s'hauran de reescriure per a adaptar-los a la nova sintaxi.

Diferenciarem els procediments de les funcions perquè, com en d'altres llenguatges d'alt nivell, les funcions retornaran un únic valor, mentre que els procediments no.

Cada SGBD, a més, tindrà una sèrie de característiques de diferent complexitat, de manera que ens permetrà realitzar operacions més o menys sofisticades amb més o menys codi.

La sentència completa serà:

```
CREATE {PROCEDURE | FUNCTION} <proc_name>
([{[IN | OUT | INOUT] [<parameter_name>] datatype [AS LOCATOR] [RESULT]} [, ...]] )
[ RETURNS datatype [AS LOCATOR]
LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS INT]<sentències_sql>
```

2.2.2. Paràmetres d'entrada i de sortida

Com podem veure en la definició, els procediments emmagatzemats podran acceptar paràmetres. Aquests paràmetres podran ser d'entrada de valors, de retorn d'un resultat o tenir una funció doble i ser paràmetres tant d'entrada com de sortida.

Cada paràmetre tindrà un nom i se li associarà un tipus de dada. Els tipus de dades dels paràmetres d'entrada han de coincidir amb els que s'han fet servir en definir les columnes de les taules.

Serà possible, a més, definir un valor per defecte per als paràmetres amb la sentència DECLARE PARAMETER.

Si volem retornar un únic valor, estarem parlant d'una funció en lloc d'un procediment i indicarem el tipus de dada de retorn mitjançant la claúsula RETURNS.

Ara crearem un procediment emmagatzemat per a calcular el percentatge de la nota d'un estudiant de grau a partir del seu codi de matrícula i sabent que, per defecte, fa sis assignatures.

```
CREATE PROCEDURE ValoracioEstudiants (in estudiant int, in nota int)
begin
DECLARE numAssignatures int default 6;
UPDATE ESTUDIANT_DE_GRAU
SET percentatge = nota/numAssignatures
WHERE codiMatricula = estudiant;
end;
```

Per a invocar-lo, el cridaríem així:

```
CALL ValoracioEstudiants (1,8);
```

Podríem comprovar el resultat de dur a terme aquest procediment tot executant la sentència:

```
SELECT * FROM estudiant_de_grau;
```

codiMatricula	nom	cognoms	percentatge	notaFinal
1	Joan	Pi Dot	1,33	8
3	Roc	Sánchez Gómez	0	8
4	Joana	Sauler Sunyer	0	7

2.2.3. Variables dins del codi

Una de les característiques principals dels procediments és que ens permetran fer servir variables, definides per l'usuari, per a emmagatzemar informació que utilitzarem en operacions posteriors dins del codi del procediment.

Per a definir variables dins d'un procediment emmagatzemat farem servir la clàusula DECLARE. Aquesta clàusula servirà per a definir variables locals i declarar paràmetres de rutines (procediments i funcions) que permetin comunicar cada rutina amb les crides corresponents.

```
DECLARE <var_name> [, <var_name>] ... <type> [NOT NULL] [DEFAULT <value>]
```

Les variables tindran un àmbit local i no tindran visibilitat fora del procediment. Tant els valors com l'espai reservat per a les variables es buidaran i descartaran un cop es finalitzi l'execució del procediment.

Per donar un valor predeterminat a una variable disposem de la cláusula DEFAULT. El valor es pot especificar com una expressió i no ha de ser constant. Si no s'especifica un valor per a la clàusula DEFAULT, el valor inicial serà NULL.

En cas que s'assigni un valor NULL a una variable indicada com a NOT NULL, l'execució del procediment retornarà un error i se'n finalitzarà l'execució.

Cada variable haurà de tenir associat un nom i un tipus de dades, que seran sempre els mateixos tipus de dades que els utilitzats per a la definició dels camps en la creació de taules.

Per a l'assignació de valors a variables trobarem diferents possibilitats, com l'assignació directa, tant si és d'una constant o el resultat d'un altre procediment, o bé l'assignació a partir de la sentència SELECT ... INTO.

Crearem una funció que retorni la mitjana de tots els percentatges dels estudiants de grau. Fixeu-vos que, com que és una funció, es fa servir la clàusula RETURNS i s'indica el tipus de dada que retorna.

```
CREATE FUNCTION MitjanaPercentatgesEstudiantsGrau() RETURNS numeric(5,2) DETERMINISTIC
BEGIN
   DECLARE mitjaPercentatges numeric(5,2);
   SELECT AVG(percentatge) INTO mitjaPercentatges
   FROM ESTUDIANT_DE_GRAU;
   RETURN (mitjaPercentatges);
END;
```

Crida a la funció i resultat:

```
select MitjanaPercentatgesEstudiantsGrau();
```

MitjaPercentatgesEstudiantsGrau()
0,44

2.2.4. Sentències condicionals

Com en la majoria dels llenguatges estructurats, es podran utilitzar sentències condicionals que ens permetran alterar el flux de l'execució del procediment.

Per a especificar les condicions, es farà servir l'estructura IF ... ELSE utilitzant la sintaxi bàsica següent:

```
IF ( <expressió_booleana> )
BEGIN
   <sentències_sql>
END
ELSE
BEGIN
   <sentències_sql>
END
```

Les condicions IF es podran niar com si fossin una expressió SQL simple. No hi ha límit al nombre de nivells niats.

Si la condició avaluada no se satisfà i l'expressió booleana retorna FALSE, executarà la consulta de declaració SQL de l'ELSE.

Com en les clàusules WHERE, es podran especificar les condicions utilitzant operadors lògics AND i OR, o operadors de comparació com ara =, <, >, <=, >=, <>. També es podran utilitzar predicats propis de l'SQL com BETWEEN, IN, IS NULL, IS NOT NULL o LIKE, així com d'altres consultes SQL que retornin una relació d'un únic camp de tipus booleà.

Si l'estructura IF ... ELSE només conté una consulta SQL, no cal que s'especifiqui la clàusula BEGIN ... END. En cas contrari, caldrà incloure-la obligatòriament per tal de permetre executar tota la consulta SQL.

Volem crear una funció per a obtenir la nota d'un estudiant de grau en format qualitatiu. A partir del codi de matrícula d'un estudiant de grau, s'obtindrà la nota en format qualitatiu d'acord amb aquestes correspondències: de 0 a 2 li correspon D; de 3 a 4, C–; de 5 a 6, C+; de 7 a 8, B, i de 9 a 10, A.

```
CREATE FUNCTION CalculNotaLletra (estudiant integer) RETURNS VARCHAR(2) DETERMINISTIC
BEGIN
   DECLARE notaLletra varchar(2);
   DECLARE quants integer;
   DECLARE nota integer;

   SELECT COUNT(*) INTO quants
   FROM ESTUDIANT_DE_GRAU
   WHERE codiMatricula = estudiant;

   IF (quants = 1) THEN
      SELECT notaFinal INTO nota
      FROM ESTUDIANT_DE_GRAU
      WHERE codiMatricula = estudiant;

      IF (nota <=2) THEN
         SET notaLletra = "D";
      ELSE
         IF (nota <=4) THEN
            SET notaLletra = "C-";
         ELSE
            IF (nota <=6) THEN
               SET notaLletra = "C+";
            ELSE
```

```
                IF (nota <=8) THEN
                   SET notaLletra = "B";
                ELSE
                   SET notaLletra = "A";
                END IF;
             END IF;
          END IF;
       END IF;
    ELSE
       SET notaLletra=" ";
    END IF;
    RETURN (notaLletra);
END;
```

Per a invocar la funció caldria executar el següent, suposant que ens interessa per a l'estudiant de codi 3:

```
SELECT calculNotalletra (3);
```

I el resultat seria:

CalculNotaLletra(3)
B

2.2.5. Sentències iteratives i ús de cursors de dades

Abans de veure quines són les sentències iteratives que podem fer servir quan creem un procediment o una funció per a repetir sentències SQL, caldrà saber què és un cursor i com es fa servir.

Un cursor és el mecanisme que tenim per a solucionar l'anomenat problema *impedance missmatch*. El problema al qual ens referim es produeix pel fet que les consultes SQL retornen conjunts de tuples i els programes d'aplicació tracten les files una per una. Per aquesta raó calia inventar un sistema que permetés recollir totes les files retornades pel codi SQL i gestionar-les perquè el programa les tracti, una per una, mitjançant un conjunt d'iteracions. La fila que es tracta en cada moment es diu que és la que està en la finestra del cursor. Por exemple, el cursor pot servir per a actualitzar (UPDATE) registres o files amb valors diferents obtinguts a partir d'una consulta SQL.

En les diferents implementacions de l'estàndard SQL podrem trobar més d'un tipus de sentències que permeten la iteració, com ara WHILE o FOR. En el primer cas, l'expressió condicional permetrà la sortida del bucle quan la condició d'iteració no es compleixi; en el segon cas, en canvi, iterarem el conjunt de sentències SQL tants cops com s'indiqui en la definició del bucle. Si es coneix el nombre d'iteracions que cal fer, el més pràctic és emprar la sentència FOR, la sintaxi de la qual és:

```
FOR <llista_variables>
IN <expressió_SQL> LOOP
      <sentència_SQL>
END LOOP;
```

Per a veure algun exemple d'utilització de bucle i com es fa servir un cursor, crearem un procediment que actualitzarà la nota de tots els estudiants de grau utilitzant el procediment per a calcular la lletra corresponent a la nota i que hem vist a l'apartat anterior.

```
CREATE PROCEDURE NotaLletra ()
BEGIN

   DECLARE codi integer;
   DECLARE lletra char(2);
   DECLARE final integer DEFAULT 0;
   DECLARE curEstudiants CURSOR FOR
SELECT codiMatricula FROM ESTUDIANT_DE_GRAU;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET final = 1;

   OPEN curEstudiants;

   WHILE NOT final DO

      FETCH curEstudiants INTO codi;

      SELECT calculNotaLletra(codi) INTO lletra;

      UPDATE ESTUDIANT
      SET notaLletra = lletra
      WHERE codiMatricula = codi;

   END WHILE;

   CLOSE curEstudiants;

END;
```

En aquest procediment es declara un cursor (curEstudiants) que permetrà obtenir tots els codis de matrícula dels estudiants de grau. Un cop definit, ja es pot fer servir i caldrà obrir-lo mitjançant la sentència SQL OPEN <Cursor>. A partir d'aquí, amb l'estructura iterativa WHILE (mentre quedin codis d'estudiants) s'obté el codi en curs utilitzant la sentència SQL FETCH <nom_cursor> INTO <variable>. Aquesta sentència diposita el codi de matrícula de la fila que està a la finestra del cursor en una variable, el que permetrà utilitzar-la tot seguit per a invocar el procediment CalculNotaLletra i obtenir el valor de la lletra amb la qual s'actualitzarà el camp nomLletra de la taula ESTUDIANT. Quan no quedin més codis de matrícula per a processar, cal tancar el cursor amb la sentència CLOSE <cursor>. Fixeu-vos que, un cop declarat el cursor, indiquem la condició per a continuar obtenint la següent fila del cursor, la qual es recuperarà amb el següent FETCH.

Així és com faríem la crida al procediment NotaLletra i en comprovaríem el resultat:

```
CALL NotaLletra();
SELECT * FROM ESTUDIANT;
```

codiMatricula	nom	cognoms	estudis	percentatge	notaLletra
1	Joan	Pi Dot	Grau	0	B
2	Laura	Sentís Aguilar	Màster	0	NULL
3	Roc	Sánchez Gómez	Grau	0	B
4	Joana	Sauler Sunyer	Grau	0	B

Justificació de l'ús dels cursors de dades

La millor opció no sempre és fer servir un cursor de dades per a iterar elements d'un conjunt de dades. Això és així perquè els sistemes de gestió de bases de dades incorporen mecanismes optimitzats per a fer certs tractaments sobre diverses files o tuples. Com a regla podem dir que cal emprar els cursors quan és necessari un tractament diferent per a cada fila o tupla. Si el tractament a fer per a cada fila de dades és el mateix, no cal utilitzar un cursor. Per exemple, si volguéssim posar a 'N' totes les notes en forma de lletra dels

estudiants de grau, no caldria un cursor. Ho podríem fer amb la sentència SQL UPDATE, tal com es mostra a continuació:

```
UPDATE ESTUDIANT
SET notaLletra = 'N';
WHERE codiMatricula IN (
   SELECT codiMatricula
   FROM ESTUDIANT_DE_GRAU
);
```

En canvi, si la nota qualitativa en forma de lletra fos diferent per a cada estudiant de grau, aleshores caldria un cursor que per a cada estudiant de grau permetés calcular-ne la qualificació textual i actualitzar aquesta dada.

2.3. Els disparadors (*triggers*)

Si bé els procediments emmagatzemats són blocs de codi que executarem sempre de manera conscient, pot ser que, per tal de mantenir una certa lògica de programació, necessitem tenir procediments que s'executin de manera transparent al desenvolupador quan la nostra base de dades entra en un cert estat o es donen una sèrie de condicions.

Imaginem el cas, ja conegut, de l'eliminació en cascada. Per tal de mantenir la integritat referencial, quan es dona l'ordre d'eliminar un registre d'una taula de la base de dades es dispara l'execució d'una sèrie d'operacions que tindran com a objectiu eliminar els registres de les taules que hi feien referència en la clau primària del registre eliminat.

Els **disparadors**, *triggers* en anglès, seran uns objectes de la base de dades que s'executaran de manera automàtica quan es produeixi un esdeveniment determinat.

Per exemple, imaginem una base de dades que modela el departament de vendes d'una empresa. Suposem, també, que en el moment en què un venedor aconsegueix vendre una quantitat determinada de productes d'un cert tipus, és a dir, si a finals d'any sobrepassa una determinada xifra de vendes, aquest venedor rebrà un premi.

Necessitem algun mecanisme que examini la xifra acumulada de vendes cada cop que s'insereixi una venda nova a la base de dades, de manera que si supera el llindar establert notifiqui que hi ha premi.

Per a implementar situacions com la descrita en l'exemple definirem un disparador que s'executarà sempre que es donin les condicions adequades.

Aquest disparador estarà accessible per a tots els objectes de la base de dades i la seva execució serà autònoma, de manera que no caldrà establir mecanismes de *polling* i, a més, serà transparent al desenvolupador, de manera que aquest no necessitarà ser conscient de la seva existència.

Polling

Acció síncrona repetitiva que té per objectiu comprovar que es compleix una certa acció o que un sistema assoleix un estat en particular.

Aquest darrer fet és, també, la contraindicació dels disparadors. Si bé el desenvolupador no necessita tenir consciència de l'existència dels disparadors, és possible que aquest fet provoqui que es realitzin noves operacions –tant si són dins de la base de dades com en les aplicacions que hi accedeixen– que invalidin o col·lideixin amb els canvis realitzats pel disparador.

Així, doncs, farem servir els disparadors en ocasions concretes:

- Implementació d'una regla de negoci, coneguda, part *core* del programa, i degudament documentada.

- Manteniment automàtic de taules d'auditoria d'activitat a la base de dades. Aquesta tasca serà aliena al negoci i no interferirà amb d'altres desenvolupaments que es puguin realitzar. Les tasques d'auditoria seran freqüents quan es treballi amb dades sensibles i es vulgui afegir una traçabilitat dels canvis transparent al negoci de l'aplicació.

- Manteniment automàtic de columnes derivades. Tot i que el model relacional recomana no utilitzar columnes derivades, és possible que la seva existència faciliti l'accés a certa informació. L'ús de disparadors permetrà mantenir les dades actualitzades de manera transparent per a l'usuari que hi accedirà.

- Comprovació de restriccions d'integritat definides pel dissenyador de la base de dades i no assumibles per la integritat referencial bàsica. D'aquesta manera, el disparador cancel·larà l'operació abans que es modifiquin les dades. Seguint l'exemple anterior, en el cas que un venedor superi una certa quantitat en els premis acumulats durant l'any, es pot fer saltar una regla d'integritat que no permeti superar-la.

- Reparació automàtica de restriccions d'integritat. En cas que no s'hagin implementat els *triggers* que validaran les regles d'integritat referencial definides pel dissenyador de la base de dades, es podran definir disparadors que, un cop infringida la regla d'integritat, realitzaran les operacions necessàries per tal de redreçar les dades per tal que tornin a ser vàlides.

2.3.1. Sintaxi

La sintaxi per a la creació d'un disparador serà diferent segons sigui l'SGBD, implementant més o menys opcions. Per exemple, en MySQL, la sintaxi serà la següent:

```
CREATE [DEFINER = usuari] TRIGGER <nom_disparador>
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON <nom_taula> FOR EACH ROW
[{FOLLOWS | PRECEDES} <nom_altre_disparador>]
```

```
<sentències_sql>
```

Com es veu en la definició, els disparadors s’activen només en tres situacions: en inserir dades en una taula (INSERT), en esborrar-les (DELETE) o en modificar valors continguts en taules (UPDATE).

Per a esborrar un disparador utilitzarem la sentència DROP.

```
DROP TRIGGER <nom_disparador> ON <nom_taula>;
```

Si volem modificar el contingut d’un disparador, el procediment a seguir serà eliminar-lo i, tot seguit, tornar a crear-lo.

Podrà succeir que hi hagi més d'un disparador assignat a més d'un esdeveniment associat a una taula o, fins i tot, en una mateixa operació sobre un camp d'una taula.

Cada SGBD implementarà un ordre d’execució propi. En el cas de MySQL, els disparadors associats a una mateixa taula i esdeveniment s’executaran successivament pel seu ordre de creació.

Es podrà modificar aquest flux a partir dels modificadors FOLLOWS i PRECEDES de la clàusula FOR EACH ROW.

Els disparadors, d’altra banda, es poden activar abans (BEFORE) o després (AFTER) d’haver processat els esdeveniments que donen peu a executar-los. Aquests esdeveniments estaran sempre fixats a una taula, de manera que si volem afegir una mateixa acció automàtica en diferents taules haurem de crear tants disparadors com taules es vulguin monitorar.

D’aquesta manera, per exemple, podrem preveure si s’incomplirà una regla d’integritat abans que la base de dades assoleixi un estat incoherent o bé podrem actualitzar una taula d’auditoria després que s’hagi aplicat una modificació d’una dada.

Las accions que s'activaran després d'executar un disparador només podran donar-se un únic cop per cada esdeveniment. Tot i això, es podran encadenar disparadors indicant si l’execució s’ha de fer abans (PRECEDES) o després (FOLLOWS) que s’hagi executat un altre disparador.

El disparador, en activar-se, adoptarà els privilegis propis de l’usuari indicat com a DEFINER, no de l’usuari que executa les accions que provoquen l’inici del disparador.

En la definició de les sentències, i només en el cas dels disparadors de tipus FOR EACH ROW, podrem fer referència als valors d'abans i després de l'acció amb les variables de sistema:

- **NEW**: és una variable de tipus compost que conté la fila després de l'execució d'una sentència de modificació (UPDATE) o de la fila que cal inserir (INSERT).
- **OLD**: és una variable de tipus compost que conté la fila abans de l'execució d'una sentència de modificació (UPDATE) o de la fila que cal esborrar (DELETE).

Finalment, podrà haver-hi situacions en què els disparadors s'executin en cascada, perquè un disparador realitza una acció que inicia un altre disparador, etc.

Aquest tipus de situacions poden portar a bucles infinits i cal tenir molta cura en dissenyar les situacions en què es crearan els disparadors.

Tot seguit mostrem un parell d'exemples de disparadors en què s'il·lustra la utilització de les clàusules AFTER i BEFORE.

Quan canviï la nota d'un estudiant de grau, volem que se'n calculi de manera automàtica la nota qualitativa i es modifiqui convenientment la columna notaLletra a la taula ESTUDIANT. Farem servir el procediment emmagatzemat CalculNotaLletra que ja hem definit anteriorment.

```
CREATE TRIGGER actualitzaNotaLletra AFTER UPDATE ON ESTUDIANT_DE_GRAU
FOR EACH ROW
BEGIN
   DECLARE nota char(2);
   SELECT calculNotaLletra (NEW.codiMatricula) INTO nota;
   UPDATE ESTUDIANT
   SET notaLletra = nota
   WHERE codiMatricula = NEW.codiMatricula;
END;
```

Si voleu provar-ne el funcionament, podeu executar:

```
UPDATE ESTUDIANT_DE_GRAU
SET notaFinal = 4
WHERE codiMatricula = 1;

SELECT * FROM ESTUDIANT WHERE codiMatricula = 1;
```

codiMatricula	nom	cognoms	estudis	percentatge	notaLletra
1	Joan	Pi Dot	Grau	0	C–

Volem que de manera automàtica es comprovi que la nova nota que es vol posar a un estudiant de grau estigui dins del rang de notes possibles; és a dir, entre 0 i 10. En cas contrari, no es podrà modificar la nota.

```
CREATE TRIGGER controlNotaFinal BEFORE UPDATE ON ESTUDIANT_DE_GRAU
FOR EACH ROW
BEGIN
   IF NEW.notaFinal < 0 or NEW.notaFinal > 10 THEN
      SET NEW.notaFinal = OLD.notaFinal;
   END IF;
END;
```

Podeu provar el resultat executant aquesta sentència:

```
SELECT * FROM ESTUDIANT_DE_GRAU WHERE codiMatricula = 1;
```

codiMatricula	nom	cognoms	percentatge	notaFinal
1	Joan	Pi Dot	1,33	4

Intenteu modificar la nota final de l'estudiant (amb codi de matrícula 1) canviant-la de 4 a 100. Veureu que no és possible, que queda com estava, ja que s'ha executat el disparador ControlNotaFinal que hem creat.

```
UPDATE ESTUDIANT_DE_GRAU
SET notaFinal = 100
WHERE codiMatricula = 1;

SELECT * FROM ESTUDIANT_DE_GRAU WHERE codiMatricula = 1;
```

codiMatricula	nom	cognoms	percentatge	notaFinal
1	Joan	Pi Dot	1,33	4

3. Concurrència i transaccions

Sovint ens trobarem que hi haurà diferents usuaris i aplicacions que accediran de manera simultània a les dades. L'SGBD haurà de ser capaç de permetre aquest tipus d'accés i assegurar que les dades recuperades són vàlides i coherents. Serà part de les tasques de l'SGBD mantenir la integritat referencial en casos d'accés concurrent i modificació simultània de la informació.

Tal com hem vist quan parlàvem dels disparadors (cas d'una acció que provoca l'execució de més d'un disparador) caldrà establir una sèrie de mecanismes que ens permetin assegurar la validesa de les dades en tot moment, dividint les accions a realitzar en petits blocs indivisibles que s'aniran executant de manera sincronitzada.

Per aquest motiu, introduirem el concepte de transacció, que representa la unitat de treball que serà necessària per a controlar la concurrència i la recuperació, si cal, de les dades.

> Una **transacció** és un conjunt d'operacions sobre la base de dades que s'han d'executar a la vegada, com si fos una de sola.

La utilització de les transaccions ens permetrà protegir les dades i les aplicacions de les anomalies provocades per l'accés simultani a la base de dades per part d'usuaris i d'altres aplicacions.

Les transaccions ens permetran, a més, dotar l'usuari de la base de dades de la sensació que només ell té accés a la base de dades, de manera que les seves accions sempre generaran el resultat esperat, sense errors derivats de l'ús d'un conjunt de dades incoherents.

Per exemple, reprenem el cas de la base de dades que modela el departament de vendes d'una empresa. Podrà succeir que dos o més venedors realitzaran una venda d'un tipus de producte al mateix temps, de manera que se superi la quantitat disponible en estoc.

En aquest cas, sense control de concurrència i de transaccions, podria passar que, sobre un estoc de 30 unitats de producte, el venedor A processa una comanda de 25, i el venedor B processa una comanda de 15. Les accions que es realitzaran sobre la base de dades seran:

1) El venedor A consulta l'estoc per a validar que es pot realitzar la comanda (30 > 25).

2) El resultat és afirmatiu i es desa el valor de l'estoc (30) per a iniciar l'operació.

3) El venedor B consulta l'estoc per a validar que es pot realitzar la comanda (30 > 15).

4) El resultat és afirmatiu i es desa el valor de l'estoc (30) per a iniciar l'operació.

5) El venedor A modifica el valor de l'estoc: 30 – 25 → 5.

6) El venedor B modifica el valor de l'estoc: 30 – 15 → 15.

El resultat és que el valor final de l'estoc és 15, quan en realitat s'ha sobrepassat la quantitat de producte existent i no s'hauria d'haver permès al venedor B realitzar l'operació.

Un altre cas que ens permetrà controlar l'ús de transaccions és el de pèrdua de connexió amb la base de dades enmig d'un conjunt d'operacions, de manera que les dades quedarien en un estat intermedi i no serien coherents amb l'estat real.

Considerem el cas que, en el procés de venda, una vegada s'ha modificat l'estoc, es passa a incrementar el valor dels ingressos per venda d'aquest venedor derivats de la venda del producte.

Si, un cop modificat l'estoc, es perd la connexió amb la base de dades, la venda no s'acabarà de processar i retornarà un error, i caldria tornar a processar des del principi comprovant l'estoc, etc.

El valor de l'estoc, en aquest moment, no serà correcte i aquesta incoherència en les dades impedirà el correcte funcionament de l'aplicació.

En definitiva, l'SGBD haurà d'aportar mecanismes de recuperació que evitin la pèrdua de dades existents, així com reaccionar davant de la incoherència en l'actualització de les dades.

Per aquest motiu, es definiran quatre propietats que hauran d'implementar tots els SGBD per tal d'oferir la capacitat de treballar amb transaccions, conegudes pel nom ACID:

1) Atomicitat: el conjunt d'operacions que constituiran una transacció es considerarà com una unitat d'execució atòmica i indivisible. Si no s'executen totes les operacions que constitueixen la transacció, caldrà recuperar l'estat previ a l'inici.

Un cop es finalitza l'execució de les operacions de la transacció, s'executarà la sentència COMMIT, que consolidarà els canvis fets en la base de dades.

Si hi ha cap problema en l'execució i cal avortar-la, es realitzarà una operació de ROLLBACK, que es podrà executar tant de manera explícita com implícita.

2) Consistència: l'execució d'una transacció preservarà, per damunt de tot, la consistència i la coherència de la base de dades, mantenint la integritat referencial, de domini, etc.

3) Isolament: una transacció s'executarà sense interferència de cap altra acció que es pugui executar de manera concurrent.

4) Definitivitat: el resultat d'una transacció serà definitiu i, només en finalitzar-se i un cop s'ha executat la sentència COMMIT, es podran modificar les taules i les dades involucrades en les operacions que formen la transacció.

3.1. Nivell de concurrència

Amb l'isolament de les transaccions, pot ocórrer que el conjunt d'operacions que la formin sigui massa extens i bloquegi tant les taules que seran modificades com les dades, i aquest fet afecti el rendiment de les aplicacions que hi accedeixin.

El **nivell de concurrència** indicarà com s'aprofitaran els recursos disponibles, segons es permeti l'encavalcament de certes operacions dins de diferents transaccions.

Continuant amb l'exemple anterior, es poden plantejar dues transaccions: una per a consultar el total de productes per tipus prèvia modificació de la taula i l'altra per a calcular el nou valor d'estoc a modificar. Atès que són operacions diferents es pot plantejar d'encavalcar de manera que es puguin dur a terme de forma concurrent ambdues transaccions.

L'SGBD haurà de ser capaç de detectar aquests possibles encavalcaments per tal d'optimitzar l'ús dels recursos disponibles i afavorir la velocitat d'execució de les operacions.

3.2. Sintaxi

Les transaccions no es definiran com a blocs de codi tancats, sinó que només s'indicaran marcant l'inici i el final de les operacions que es consideraran com a part de la transacció.

Una vegada activada la transacció, totes les operacions que es realitzin a continuació en formaran part fins que se n'indiqui la finalització de manera explícita.

Nota

En altres SGBD la instrucció que indica inici de transacció pot ser BEGIN TRANSACTON o START TRANSACTION.

Per a indicar l'inici d'una transacció, utilitzarem la sintaxi següent:

```
SET TRANSACTION <mode_d_acces>;
```

El mode d'accés podrà ser de dos tipus: READ ONLY, en cas que només es vulgui accedir a les bases de dades per a realitzar consultes, i READ WRITE, en cas que alguna de les operacions incloses modifiqui les dades.

Per a indicar la finalització d'una transacció, utilitzarem la sintaxi següent:

```
{COMMIT | ROLLBACK} [WORK];
```

Com hem vist prèviament, la sentència COMMIT consolidarà els canvis realitzats durant la transacció, mentre que ROLLBACK revertirà els canvis realitzats durant la transacció.

La paraula reservada WORK serà només indicativa i, a més, serà opcional.

3.3. Responsabilitats de l'SGBD i del desenvolupador

Com ja hem vist, per a evitar els impactes en el rendiment de la base de dades, serà important planificar correctament l'ordre de les operacions que s'executaran dins d'una transacció.

Així, doncs, per part del desenvolupador caldrà determinar si una certa operació haurà de formar part d'una transacció o no, o si una transacció es pot dividir en diferents transaccions per a afavorir la concurrència.

Aquesta tasca serà part de les responsabilitats del desenvolupador, que haurà de garantir que les transaccions tindran una durada mínima, garantint també la coherència i la consistència de les dades.

L'SGBD haurà de garantir, definint un pla d'accés a partir de l'estudi de les operacions que formen les diferents transaccions, que les peticions d'escriptura i de lectura s'executin en concurrència de manera òptima.

Així mateix, serà part de les responsabilitats de l'SGBD garantir que es compleixen totes les regles d'integritat definides en la base de dades, tant si és com a validació prèvia al tancament de la transacció amb un COMMIT, com si és immediatament després de l'execució de cadascuna de les operacions.

Òbviament, en el cas d'error o de violació d'una regla d'integritat, l'SGBD haurà de ser capaç de retornar a l'estat previ a l'inici de la transacció en què estava la base de dades.

Resum

En aquest mòdul didàctic hem vist com s'organitza un servidor de base de dades, representat com un conjunt de catàlegs i, dins d'aquests, un conjunt d'esquemes. En aquests esquemes hi trobarem altres elements de la base de dades com poden ser les taules o vistes.

També hem vist diferents objectes de la base de dades que ofereix el sistema gestor per a facilitar l'accés a les dades. Per començar, les vistes, que permeten accedir a subconjunts de dades de una o més taules com si estiguessin emmagatzemades en una única taula i en quins casos es modificable aquest conjunt de dades.

Després, hem vist altres objectes que s'emmagatzemen en l'esquema de bases de dades i que a més, permeten encapsular l'accés a dades fent-lo transparent al desenvolupador d'aplicacions: els procediments emmagatzemats i els disparadors.

Aquests objectes ens permetran definir algoritmes i conjunts d'operacions que s'executaran de manera explícita. En el cas dels procediments i funcions, o de manera implícita, en el cas dels disparadors, en el moment en què es donin una sèrie de condicions establertes.

Finalment, hem vist com treballaria l'SGBD en cas que hi hagi un accés concurrent a les taules i dades, per mitjà de transaccions, per tal de garantir la coherència i la integritat de la informació.

Activitats

1. Determineu si les afirmacions següents són vertaderes o falses i argumenteu breument la vostra resposta.

a) Segons l'SQL estàndard, una BD es crea amb la sentència CREATE DATABASE <nom_bd>.

b) L'esquema d'una base de dades conté informació sobre les taules i altres components lògics dels esquemes dels usuaris.

c) Segons l'SQL estàndard, quan s'inicia una sessió sempre cal explicitar l'inici d'una transacció amb la sentència SET TRANSACTION <mode_transacció>.

2. A partir d'aquestes taules:

```
CREATE TABLE SOCI (
   nSoci char(10) primary key,
   sexe char(1) not null,
   check (sexe='D' or sexe='H'));

CREATE TABLE CLUB (
   nClub char(20) primary key);

CREATE TABLE SOCI_CLUB (
   nSoci char(10) not null references SOCI,
   nClub char(20) not null references CLUB,
   primary key(nsoci, nclub));

CREATE TABLE clubs_amb_mes_de_5_socis (
   nClub char(20)
   primary key references CLUB);
```

i de les restriccions d'integritat següents:

```
RI1: Un club no pot tenir més de vint socis.
RI2: Un club ha de tenir més dones que no pas homes.
```

implementeu un procediment emmagatzemat anomenat AssignarIndividual que, donat un soci i un club, insereixi l'assignació a la taula SOCI_CLUB.

A més, si el club passa a tenir més de cinc socis, l'ha de donar d'alta a clubs_amb_mes_de_cinc_socis. El procediment ha d'informar dels errors per mitjà d'excepcions, i proporcionar els missatges d'error següents:

```
'Soci ja assignat a aquest club'
'El soci o el club no existeixen'
'El club té més de vint socis'
'El club té menys dones que homes'
'Error intern'
```

3. A partir de les taules creades amb les sentències indicades a continuació, definiu un disparador que implementi la regla de negoci: «quan la modificació de l'estoc d'un producte el deixi per sota del punt de comanda, cal inserir-hi una petició pendent, si no s'havia fet prèviament».

```
CREATE TABLE PRODUCTE(
   nProd INTEGER,
```

```
    estoc INTEGER NOT NULL,
    puntComanda INTEGER NOT NULL,
    qtatDemanar INTEGER NOT NULL,
    PRIMARY KEY(nProd));

CREATE TABLE PETICIO_PENDENT(
    nProd INTEGER,
    qtt INTEGER NOT NULL,
    data DATE,
    PRIMARY KEY (nProd));
```

4. A partir de les taules creades en l'exercici anterior, definiu una vista que implementi la regla de negoci: «obtenir l'identificador de producte i estoc sempre que l'estoc sigui menor que el valor de punt de comanda».

Es poden inserir dades en aquesta vista? Quines són les condicions que cal avaluar abans de la inserció?

Glossari

ACID *m* Acrònim format pels mots *atomicitat*, *consistència*, *isolament* i *definitivitat*, que indica les propietats que ha de tenir tota transacció.

administrador de la base de dades *m* Usuari de la base de dades que gestionarà els objectes, l'estructura i els privilegis d'accés de la resta d'usuaris.

cancel·lació d'una transacció *f* Finalització d'una transacció sense que se'n confirmin les actualitzacions fetes en la BD.

catàleg *m* Component de l'entorn SQL que conté un conjunt d'esquemes, un dels quals és l'esquema d'informació, que té tota la informació dels esquemes dels usuaris (noms de taules, columnes, restriccions, definicions de vistes, etc.).

confirmació d'una transacció *f* Finalització d'una transacció que fa que els canvis realitzats esdevinguin definitius en la BD.

connexió *m* Associació que es crea entre un client i un servidor.

control de concurrència *m* Conjunt de tècniques que utilitza un SGBD per a evitar que hi hagi interferències entre transaccions que s'executen concurrentment.

DBA *m* Sigles en anglès de l'usuari administrador de la base de dades.

disparador *m* Acció o procediment emmagatzemat que s'executa automàticament quan es duu a terme una operació d'inserció, d'esborrament o de modificació en alguna taula de la base de dades.

esquema *m* Element que agrupa un conjunt de components lògics (taules, vistes, procediments emmagatzemats, restriccions, etc.).

nivell de concurrència *m* Grau d'aprofitament dels recursos de procés, disponibles segons l'encavalcament d'execució de les transaccions, que accedeixen concurrentment a la BD i aconsegueixen confirmar.

procediment emmagatzemat *m* Acció o funció definida per un usuari que proporciona un servei determinat. Un cop creat, el procediment es desa en la base de dades i es tracta com un objecte més.

rol *m* Conjunt de privilegis que té assignat un usuari de la base de dades, definit per l'usuari administrador.

servidor *m* Element superior de la jerarquia de components de l'entorn SQL que conté un conjunt de catàlegs.

sessió *f* Conjunt de sentències SQL que s'executen mentre hi ha una connexió activa en un servidor.

transacció *f* Seqüència d'operacions de lectura i d'actualització de la BD que compleix les propietats ACID.

vista *f* Conjunt de dades, resultat d'una consulta, al qual es pot accedir igual que a una taula.