# Programació mitjançant l'SQL

David Fíguls i Massot

PID_00171648

UOC
Universitat Oberta de Catalunya

*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. Necessitat de l'SQL en les aplicacions**.......... 7
1.1. Biblioteques d'accés a bases de dades .......... 7
1.2. Base de dades de referència .......... 8
1.3. Conceptes comuns .......... 8
1.4. Tecnologies d'accés a bases de dades .......... 10
1.4.1. L'SQL hostatjat .......... 11
1.4.2. SQL/CLI .......... 13
1.4.3. ODBC .......... 13
1.4.4. OLE DB .......... 13
1.4.5. DAO / RDO / ADO / ADO.NET .......... 14
1.4.6. JDBC .......... 14
1.5. Comparativa de les tecnologies d'accés a bases de dades .......... 14
1.5.1. Nivell de programació .......... 14
1.5.2. Flexibilitat i eficiència de sentències SQL .......... 15
1.5.3. Portabilitat a nivell de sistema de gestió de bases de dades .......... 15
1.5.4. Llenguatges de programació suportats .......... 16

**2. API JDBC i *drivers***.......... 17
2.1. *Drivers* JDBC .......... 18
2.2. Tipus de *drivers*.......... 19
2.2.1. Pont JDBC-ODBC (tipus 1) .......... 19
2.2.2. Pont JDBC-libSGBD (tipus 2) .......... 20
2.2.3. Directe JDBC-SGBD (tipus 4) .......... 21
2.2.4. JDBC-Intermediari (tipus 3) .......... 21
2.3. Comparativa dels quatre tipus de *drivers*.......... 22

**3. Programació amb JDBC**.......... 25
3.1. Connexió a una base de dades .......... 25
3.2. Consultes i modificacions bàsiques .......... 27
3.2.1. Execució de sentències SQL: la classe `Statement`.......... 27
3.2.2. Tractament de resultats: la classe `ResultSet`.......... 29
3.3. Consultes i modificacions avançades .......... 31
3.3.1. `ResultSet` amb llibertat de moviments .......... 32
3.3.2. Modificacions per mitjà d'un `ResultSet`.......... 32
3.4. Consultes i modificacions amb `PreparedStatement`.......... 34
3.5. Consideracions addicionals sobre consultes i modificacions .......... 35

3.5.1. Tractament de valors nuls .......... 35
3.5.2. Combinacions de taules .......... 38
3.5.3. Consultes niuades .......... 39
3.5.4. Consultes recursives .......... 40
3.5.5. Claus primàries autogenerades .......... 41
3.5.6. SQL injection .......... 42
3.6. Procediments emmagatzemats .......... 44
3.6.1. Paràmetres d'entrada .......... 44
3.6.2. Paràmetres de sortida .......... 44
3.6.3. Paràmetres d'entrada i sortida .......... 45
3.7. Gestió d'errors .......... 45
3.7.1. Tipus d'error i com es reporten .......... 45
3.7.2. Obtenció d'informació sobre una `SQLException` .......... 46
3.7.3. Gestió de l'error des dels nostres programes .......... 47
3.8. Gestió de transaccions .......... 50
3.8.1. Durada d'una transacció .......... 52

**Resum** .......... 53

**Activitats** .......... 55

**Glossari** .......... 56

**Bibliografia** .......... 57

# Introducció

En aquest mòdul estudiarem diverses tècniques per a operar amb bases de dades (BD) relacionals des de les nostres aplicacions. És el que s'anomena *SQL programat* o *SQL immers*.

Hem vist la manera d'interaccionar amb una BD mitjançant la creació i execució de sentències SQL. Aquest mètode de treball es coneix com a *SQL interactiu* o *SQL directe* i és utilitzat, principalment, pels administradors de BD.

La major part dels usuaris d'una BD no tenen coneixements de l'SQL i, per accedir a la BD, ho fan mitjançant aplicacions desenvolupades a mida. Aquestes aplicacions contenen sentències SQL que en temps d'execució s'envien a la BD, es processen i, si hi ha resultats, es retornen a l'aplicació. Aquest segon mètode de treball es coneix com a *SQL programat*.

La comunicació entre l'aplicació i la BD, lluny de ser irrellevant, sovint condiciona de ple el rendiment general de l'aplicació. Una utilització incorrecta de les tècniques de l'SQL programat pot alentir innecessàriament o, fins i tot, bloquejar la millor BD i les aplicacions que en depenen. Així, l'objectiu final del mòdul és aportar els coneixements necessaris per a aconseguir una comunicació eficient entre l'aplicació i la BD.

Podem utilitzar SQL programat des d'un ampli ventall de llenguatges de programació i sistemes gestors de bases de dades (SGBD). En aquest mòdul presentem les solucions més habituals i ens centrarem en el llenguatge de programació Java i en l'SGBD PostgreSQL.

# Objectius

Els materials didàctics que integren aquest mòdul permetran a l'estudiant assolir els objectius següents:

**1.** Presentar les tècniques principals de l'SQL programat: l'SQL hostatjat, l'SQL/CLI, ODBC, OLE DB, ADO i JDBC.

**2.** Comprendre les diferències més importants que hi ha entre les diverses tècniques de l'SQL programat, i els avantatges i inconvenients principals de cadascuna.

**3.** Aprendre les diverses estratègies de treball que ens ofereix el JDBC.

**4.** Combinar correctament els mecanismes que ens ofereix el JDBC per a desenvolupar aplicacions que operin eficientment amb una base de dades.

# 1. Necessitat de l'SQL en les aplicacions

Totes les aplicacions manipulen dades mitjançant variables de tipus bàsics o estructures de dades més o menys complexes. A la fi de l'execució de l'aplicació totes aquestes dades s'esborren, si no és que utilitzem algun tipus d'emmagatzematge permanent. En alguns casos n'hi ha prou de guardar les dades en fitxers, però, majoritàriament, caldrà utilitzar una base de dades (BD).

Si el volum de dades que manipula l'aplicació és prou elevat, la utilització de fitxers és contraproduent, ja que alenteix l'execució o complica la programació. Aquest motiu ja seria suficient per a fer-nos optar per una BD, si bé no és l'únic.

Quan enllacem una aplicació amb una BD podem aprofitar qualsevol de les funcionalitats pròpies dels sistemes de gestió de bases de dades (SGBD) que hem anat estudiant al llarg del curs.

La clau de volta és l'elecció d'un llenguatge de programació i d'un SGBD que facin possible la interacció entre l'aplicació i la BD. Afortunadament podem trobar un ampli ventall de solucions que tenen un tret en comú: utilitzen SQL per a comunicar l'aplicació i la BD (per això, el nom de SQL programat).

Al llarg d'aquest apartat farem una repassada a les solucions principals:

- SQL hostatjat (en C i Java)
- SQL/CLI
- ODBC
- OLE DB
- ADO
- JDBC

## 1.1. Biblioteques d'accés a bases de dades

Totes les solucions tindran, com a element comú, la utilització d'una biblioteca encarregada d'enllaçar les aplicacions amb les BD. L'SQL hostatjat és la solució que més amaga la utilització d'una biblioteca, però, en darrer terme, fa crides a funcions d'una biblioteca emprant una sintaxi diferent.

Un objectiu bàsic de totes les solucions és la portabilitat. És a dir, la possibilitat de portar el codi d'un SGBD a un altre amb el mínim de canvis. A aquest efecte i al llarg del temps s'han anat definint diferents estàndards. Paral·lelament, els fabricants de SGBD han anat publicant biblioteques per accedir a les seves BD seguint els diversos estàndards.

Tots els estàndards permeten fer pràcticament el mateix. La diferència no està en el que permeten fer sinó, més aviat, en la manera de dur-ho a terme. L'evolució que han fet ha estat en la línia de simplificar la feina de programació.

### 1.2. Base de dades de referència

Al llarg del mòdul anirem veient diversos exemples de programes que utilitzen una BD per a gestionar els missatges de diferents fòrums. La BD tindrà les taules següents:

**1) Usuaris**. Hi guardarem les dades dels diferents usuaris. Columnes: nom d'usuari (clau primària), contrasenya, nom i cognoms.
**2) Fòrums**. Punt d'entrada de cada fòrum. Columnes: codi fòrum (clau primària), nom.
**3) Missatge**. Cadascun dels missatges enviats. Columnes: codi missatge (clau primària), codi fòrum (identifica el fòrum al qual pertany el missatge), autor (identifica l'usuari que ha escrit el missatge), títol, text, fil (identifica el missatge al qual dóna resposta).
**4) Lectures**. Registre de les lectures que els usuaris fan dels missatges. Columnes: codi missatge, nom d'usuari (aquestes dues primeres columnes defineixen la clau primària), data i hora.

USUARIS (nom_usuari, contrasenya, nom, cognoms)
FORUMS (codi_forum, nom)
MISSATGES (codi_missatge, codi_forum, autor, titol, text, fil)
{codi_forum} és clau forana que referencia FORUMS(codi_forum)
{autor} és clau forana que referencia USUARIS(nom_usuari)
{fil} és clau forana que referencia MISSATGES(codi_missatge)
LECTURES (codi_missatge, nom_usuari, data_hora)
{codi_missatge} és clau forana que referencia MISSATGES(codi_missatge)
{nom_usuari} és clau forana que referencia USUARIS(nom_usuari)

### 1.3. Conceptes comuns

Malgrat l'absència d'una solució única de l'SQL programat, totes les alternatives tenen un comú divisor que estudiem a continuació:

**1) Connexió i desconnexió**. Partint de la base que les aplicacions accedeixen a una BD per fer-hi unes quantes operacions i no una de sola, totes les solucions opten per separar la connexió o desconnexió de les operacions. La raó és amortitzar el temps que es tarda a posar-se en contacte amb la BD entre les diverses operacions. Hem de tenir en compte que aquest temps pot ser significatiu especialment si la BD està en una màquina diferent de la de l'aplicació.

L'estratègia habitual per a programar aplicacions ha estat, durant molt de temps, iniciar una connexió en arrancar l'aplicació client i mantenir-la fins al final.

Amb l'aparició de les aplicacions web aquesta estratègia ha canviat radicalment. El nombre d'usuaris (clients) que poden executar l'aplicació simultàniament fa inviable mantenir una connexió independent per a cadascun.

Tenint present que un usuari només utilitza la connexió puntualment en el moment de sol·licitar una pàgina, podem pensar en dues solucions:

- Reaprofitar connexions entre clients diferents.
- Obrir i tancar connexions en cada pàgina.

De les dues solucions, la segona és més simple però menys eficient a causa de la lentitud de la connexió. En aquest sentit, els darrers anys han sortit biblioteques que permeten fer connexions més lleugeres i ràpides, optimitzades per a aplicacions web.

Sigui com sigui, durant la connexió a una BD cal especificar uns quants paràmetres, entre els quals els més importants són:

- *host* on localitzar l'SGBD.
- *port* on escolta.
- nom de la BD a la qual ens volem connectar (recordem que un SGBD pot gestionar més d'una BD).
- usuari i contrasenya amb el qual ens volem connectar.

**2) Sentències SQL estàtiques o dinàmiques**. Un cop connectats a la BD, podem interaccionar-hi mitjançant sentències SQL. Abans de l'execució d'una sentència SQL, l'SGBD ha d'analitzar-ne el text, verificar que és sintàcticament correcte, transformar-lo, optimitzar-lo, etc. Segons el moment en què es realitza aquest procés distingim entre:

**a) SQL estàtic**. Efectuem aquest procés en temps de compilació. En principi, és una estratègia més ràpida, però més rígida, ja que hem de tenir definides totes les sentències que executarà l'aplicació.

**b) SQL dinàmic**. Duem a terme aquest procés en temps d'execució, just en el moment d'efectuar cadascuna de les sentències SQL que conté l'aplicació. En principi, alenteix l'execució de l'aplicació, però permet la creació de sentències SQL en temps d'execució.

**3) Sentències SQL, paràmetres i variables**. Fins i tot en l'SQL estàtic, la major part de les sentències tenen una part variable que es defineix en temps d'execució. Ens referim a valors de columnes, utilitzats en sentències de modificació o de consulta, que no impedeixen que l'anàlisi de la sentència es faci en temps de compilació. Per exemple, en la nostra BD de referència:

**a)** Per afegir un nou missatge hem de fer un `INSERT` i indicar-ne els valors de les columnes. Una aplicació demanaria les dades a l'usuari, les emmagatzemaria temporalment en variables i, d'alguna manera, aquestes variables es vincularien amb la sentència SQL.

**b)** Per obtenir els missatges d'un fòrum hem de fer un `SELECT` i indicar, a la clàusula `WHERE`, el codi del fòrum del qual volem els missatges. Una aplicació demanaria a l'usuari el codi del fòrum, l'emmagatzemaria en una variable, i es vincularia amb la sentència SQL.

**4) Sentències SQL amb resultats, iteracions i variables**. Les sentències SQL consultores (`SELECT`) retornen resultats que, d'alguna manera, s'han d'emmagatzemar en variables perquè l'aplicació les pugui tractar posteriorment. Aquests resultats poden ser un valor, un conjunt de valors (corresponents a una fila d'una sentència `SELECT`) o un conjunt de files (totes les dades provinents d'una sentència `SELECT`).

Les sentències que retornen un conjunt de files sovint es tracten d'una manera iterativa, accedint als valors d'una fila en cada iteració. A aquest efecte tindrem operacions per a consultar els valors de la fila actual, avançar a la fila següent i detectar el final. Els SGBD ofereixen una estructura de control per poder fer efectius aquests recorreguts. És el que es coneix amb el nom de *cursor*.

**5) Transaccions**. Podem definir transaccions agrupant diferents sentències SQL que, conceptualment, constitueixin una unitat indivisible d'execució. Les operacions disponibles són les habituals: iniciar transacció, confirmar transacció o desfer transacció.

**6) Tractament d'errors**. Totes les solucions ofereixen alguna via per capturar els errors que es poden produir en executar una sentència SQL. La manera de fer-ho, però, canvia molt depenent de la solució que triem.

### 1.4. Tecnologies d'accés a bases de dades

A continuació presentem les solucions de l'SQL programat més habituals: l'SQL hostatjat, l'SQL/CLI, l'ODBC, l'OLE DB, l'ADO i el JDBC. Entre totes aquestes, l'SQL hostatjat és la més antiga i la que es desmarca més de les altres perquè:

- Utilitza un sintaxi pròpia, allunyada de la sintaxi dels llenguatges de programació. Les altres usen crides a funcions.
- No permet l'execució de sentències SQL dinàmiques. Les altres sí.

- Empra un tipus de variables especials per a comunicar-se amb la BD. Les altres fan servir els tipus de variables habituals que ofereixen els llenguatges de programació.

### 1.4.1. L'SQL hostatjat

Abans de l'aparició de l'SQL hostatjat, la comunicació amb BD s'havia de fer per mitjà de funcions de biblioteques subministrades per cadascun dels fabricants de SGBD. El treball directe amb aquestes biblioteques no era prou àgil i va motivar l'aparició de l'SQL hostatjat per tal de:

- Simplificar l'enviament i la recepció de dades de les sentències SQL.
- Verificar sintàcticament les sentències SQL en temps de compilació.
- Permetre la portabilitat del codi entre diferents SGBD (a aquest efecte, l'SQL hostatjat va ser inclòs dins l'estàndard SQL).

La característica més visible de l'SQL hostatjat és la introducció de sentències SQL enmig del codi de l'aplicació, utilitzant una sintaxi al marge de la pròpia del llenguatge de programació amfitrió. Aquest codi híbrid no pot ser compilat normalment i requerirà un tractament previ: una precompilació.

**Exemple**

Vegem un exemple escrit en el llenguatge de programació amfitrió C que obre i tanca una connexió a la base de dades de referència (anomenada `bdMail`) amb l'usuari `jmarti` i la contrasenya `1234`.

```
int main() {
  EXEC SQL CONNECT TO bdMail USER jmarti/1234;
  EXEC SQL DISCONNECT;
}
```

Durant la precompilació s'analitzen les sentències SQL i se substitueixen per crides a funcions d'una biblioteca encarregades de l'accés a la BD. A partir d'aquest moment, el codi ja es pot compilar normalment. El resultat de la precompilació de l'exemple anterior seria:

```
/* Processed by ecpg (4.5.0) */
/* These include files are added by the preprocessor */
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
/* End of automatic include section */

#line 1 "connect.pgc"
int main() {
    {
    ECPGconnect(__LINE__, 0, "bdMail" , "jMarti" ,
      "1234" , NULL, 0);
    }
#line 3 "connect.pgc"
    { ECPGdisconnect(__LINE__, "CURRENT");}
#line 7 "connect.pgc"
}
```

Atès que les sentències SQL s'analitzen en temps de compilació, l'SQL hostatjat només permet l'SQL estàtic (tot i que en alguna versió és possible un cert grau de dinamisme).

Malgrat que la tècnica de SQL hostatjat es podria aplicar a qualsevol llenguatge de programació, a la pràctica, l'elecció està molt restringida. Cal que l'SGBD ofereixi una biblioteca i un precompilador per al llenguatge de programació que ens interessi. A continuació veurem dos casos concrets:

- **L'SQL hostatjat en C**. El llenguatge més habitual per a utilitzar l'SQL hostatjat és C. Pot sobtar que el llenguatge escollit sigui C, en comptes d'un llenguatge més actual, però no oblidem que C és el llenguatge amb el qual es codifiquen els principals sistemes operatius i SGBD.

- **L'SQL hostatjat en Java: l'SQLJ**. Java ofereix la possibilitat d'accedir a una BD mitjançant l'SQL estàtic (anomenat *SQLJ*). Malgrat tot, pocs SGBD ofereixen el precompilador per fer-ho possible.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex002SelectMissatgesUsuari.pgc.*

**Exemple**

El següent exemple mostra els missatges d'un usuari. Està escrit en l'SQL hostatjat en C i, d'entrada, podem apreciar com es combinen les línies C i de l'SQL. Per a recórrer els missatges d'un usuari utilitzem un cursor que ens permet moure'ns a la fila següent i accedir als valors de les seves columnes (ho aconseguim amb l'operació `FETCH...INTO`). El vincle entre les sentències SQL i el codi C són les variables pont, declarades en la secció `DECLARE SECTION` i, en l'exemple, assignades cada cop que fem un `FETCH...INTO` (fixem-nos amb els dos punts : que precedeixen les variables pont). Finalment, per detectar el final del recorregut hem de consultar si hi ha cap error (`sqlca.sqlstate`). Quan l'error ens indica que no hi ha més files per llegir (error amb codi "02000") acabem el recorregut.

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
//Declaracio de variables pont
EXEC SQL BEGIN DECLARE SECTION;
  char usuari[20];
  char password[20];
  int codi_missatge;
  char titol[40];
  char text[250];
  char autor[10];
EXEC SQL END DECLARE SECTION;

//Connectem a la base de dades de referencia
//amb usuari i contrasenya.
printf("Usuari:"); scanf("%s",usuari);
printf("Password:"); scanf("%s",password);
EXEC SQL CONNECT TO bdMail@localhost
  USER :usuari USING :password;

//cursor per a poder llegir les files de la consulta
printf("Autor:"); scanf("%s",autor);
EXEC SQL DECLARE cMissatges CURSOR FOR
  SELECT codi_missatge,títol,text FROM Missatges
  WHERE autor=:autor;

EXEC SQL OPEN cMissatges;
//Obtenim la primera fila de la consulta
EXEC SQL FETCH cMissatges
      INTO :codi_missatge, :titol, :text;
bool final=strcmp(sqlca.sqlstate,"02000")==0;
```

```
    while (!final) {
      //Mostrem les variables pont
      printf("%d -- %s -- %s\n",codi_missatge,titol,text);

      //Obtenim la fila seguent
      EXEC SQL FETCH cMissatges
          INTO :codi_missatge, :titol, :text;
      final=strcmp(sqlca.sqlstate,"02000")==0;
    }
    printf("final\n");
    EXEC SQL CLOSE cMissatges;
    EXEC SQL DISCONNECT;
```

### 1.4.2. SQL/CLI

El grup format per *X/open* i l'SQL *access group* (SAG) va desenvolupar l'especificació d'una interfície per a permetre l'SQL programat, anomenada *call level interface* (CLI). Tenien per objectiu incrementar la portabilitat d'aplicacions gràcies a la definició d'una interfície a partir de la qual es pogués accedir a qualsevol SGBD. La major part de l'especificació de la interfície va ser afegida a l'estàndard SQL:1992 l'any 1995.

### 1.4.3. ODBC

Microsoft va desenvolupar la interfície *open database connectivity* (ODBC) basant-se en una versió preliminar de la interfície SQL/CLI. Tot i que al principi va ser desenvolupat pel sistema operatiu de Microsoft, posteriorment s'ha estès a altres sistemes operatius.

L'aportació d'ODBC va ser la definició de l'entorn que permet carregar dinàmicament els *drivers* d'un SGBD concret a partir del nom de la BD.

Això dóna pas a crear aplicacions i compilar-les sense associar-les a cap SGBD concret, només al nom d'una BD. En temps d'execució, el *driver manager* té la funció de carregar el *driver* necessari per a accedir a la BD i associar-lo a l'aplicació.

### 1.4.4. OLE DB

La segona interfície d'accés a dades proposada per Microsoft és *object linking and embedding database* (OLE DB). Mentre que ODBC permetia l'accés a BD relacionals, OLE DB està dissenyada per a accedir a qualsevol origen de dades, com ara BD orientades a objectes, fulls de càlcul, correu, etc.

Podem trobar *drivers* OLE DB, anomenats *proveïdors*, per a accedir a un ampli ventall d'orígens de dades.

#### 1.4.5. DAO / RDO / ADO / ADO.NET

Després de la definició de les interfícies ODBC i OLE DB, Microsoft ha anat publicant diferents biblioteques orientades a objectes. La finalitat d'aquestes biblioteques és simplificar la utilització de les interfícies des de l'antic VisualBasic (el que va ser llenguatge de programació estrella de Microsoft) i actualment des de .NET.

Inicialment va aparèixer *data access objects* (DAO) que permetia accedir a BD Access o, per mitjà d'ODBC, a qualsevol altre SGBD. Era una interfície simple i orientada a aplicacions petites.

La segona biblioteca va ser *remote data objects* (RDO), que permetia accedir a pràcticament totes les funcions d'ODBC, de manera que augmentava el potencial del DAO. Anava orientat a aplicacions grans.

Més recent és la biblioteca *activeX data objects* (ADO), que simplifica el model d'objectes de les anteriors i amplia l'accés a dades per mitjà d'OLE DB. La darrera biblioteca de Microsoft és ADO.NET, que seria la versió d'ADO per a ser utilitzada des de llenguatges de programació .NET.

#### 1.4.6. JDBC

El llenguatge de programació Java utilitza la biblioteca anomenada *Java data base connectivity* (JDBC) per a accedir a dades. Per mitjà de JDBC podem accedir a BD i altres orígens de dades com ara fulls de càlcul, fitxers de text, etc. Seria l'equivalent d'ADO.NET en l'entorn Java i, per tant, serà el nostre centre d'interès al llarg d'aquest mòdul didàctic.

### 1.5. Comparativa de les tecnologies d'accés a bases de dades

Davant de totes aquestes alternatives, cal fer un punt i apart per a reflexionar sobre els avantatges i els inconvenients de cada solució i, sobretot, treure conclusions que ens guiïn en l'elecció de la solució òptima en cada situació. Centrarem la comparativa des de perspectives diferents:

- Nivell de programació
- Flexibilitat i eficiència de sentències SQL
- Portabilitat
- Llenguatges de programació que suporta.

#### 1.5.1. Nivell de programació

Les diferents solucions exposades utilitzen tècniques de programació clarament diferenciades:

- L'SQL hostatjat: usa una sintaxi diferent i requereix un precompilador que ofereixen diversos SGBD per a alguns llenguatges de programació.

- Biblioteca de funcions SQL/CLI, ODBC, OLE DB: contenen funcions simples i, per tant, eficients d'executar, però de programació feixuga. Normalment s'utilitzen des de llenguatge C.

- Biblioteca orientada a objectes DAO, RDO, ADO, JDBC: biblioteques que estan per sobre de les anteriors i que en simplifiquen la utilització. Són orientades a objectes, amb funcions de nivell superior i que permeten una programació més agradable, però poden no ser tan eficients com les anteriors.

### 1.5.2. Flexibilitat i eficiència de sentències SQL

L'SQL hostatjat és l'única solució que permet l'SQL estàtic. La resta permeten l'SQL dinàmic, però poden simular l'SQL estàtic mitjançant procediments emmagatzemats.

Simplificant-ho, podem considerar que l'SQL hostatjat és la solució menys flexible, però més eficient. La realitat és més complexa i, en alguns casos, utilitzant l'SQL dinàmic es pot aconseguir una eficiència similar a l'SQL estàtic.

### 1.5.3. Portabilitat a nivell de sistema de gestió de bases de dades

Tot i que totes les solucions que hem vist tenen per objectiu la portabilitat, aquesta no s'aconsegueix de la mateixa manera.

Tant l'SQL hostatjat com l'SQL/CLI han de ser compilats amb la biblioteca d'accés a SGBD. Tot i que podem trobar biblioteques per a la gran majoria de SGBD, hem de fer l'elecció en el moment de compilar l'aplicació i, per tant, abans de distribuir-la. Canviar de SGBD comportaria la recompilació de l'aplicació.

A partir d'ODBC, totes les solucions es compilen utilitzant una interfície. En temps d'execució es carreguen els *drivers* adequats segons el que determini l'aplicació i això, sovint, queda definit en un fitxer de configuració de l'aplicació. Per tant, per a canviar SGBD, només cal modificar el fitxer de configuració de l'aplicació, sense recompilar res.

En realitat, cap de les solucions no garanteix una portabilitat total. Les aplicacions sovint utilitzen extensions SQL pròpies d'un SGBD que són incompatibles amb altres SGBD.

Addicionalment, per a executar aplicacions desenvolupades amb ODBC o posteriors, cal tenir instal·lats els *drivers* per a l'SGBD que escollim. Aquests *drivers* han d'estar instal·lats en tots els ordinadors on s'ha d'executar l'aplicació i, per tant, en condiciona la distribució.

### 1.5.4. Llenguatges de programació suportats

L'SQL hostatjat es podria aplicar a qualsevol llenguatge, però necessita un precompilador específic per a cadascun. En la pràctica són pocs els llenguatges que ofereixen l'SQL hostatjat, però l'aparició de l'SQLJ demostra que aquesta solució, malgrat els anys d'existència, continua essent vigent.

Les solucions basades en biblioteques poden ser utilitzades des de qualsevol llenguatge de programació capaç d'enllaçar-se amb la biblioteca. Tot és possible, però en la pràctica algunes combinacions són complicades.

Entre d'altres, algunes de les combinacions més habituals són:

- SQL/CLI, ODBC, OLE DB: C, C++
- DAO, RDO, ADO: VBasic 6.0
- ADO.NET: .NET
- JDBC, SQLJ: Java

Com que les aplicacions que necessiten accés a BD són majoritàriament aplicacions de gestió, les combinacions més freqüents són les dues darreres. Els llenguatges Java i .NET són les opcions més interessants per a desenvolupar tant aplicacions de gestió tradicionals com aplicacions web.

# 2. API JDBC i *drivers*

En aquest apartat iniciem el tema principal d'aquest mòdul didàctic fent una introducció a l'*aplication program interface* de JDBC (API JDBC), la biblioteca de classes i interfícies que ens permetrà accedir a BD des d'aplicacions escrites en Java.

Per a accedir a BD, JDBC segueix el mateix patró utilitzat en totes les solucions vistes en l'apartat anterior:

- Permet connectar-nos a una BD i, mitjançant sentències SQL, consultar-ne el contingut o modificar-lo.
- Defineix una interfície comuna a tots els SGBD, tal com proposava l'SQL/CLI.
- Permet escollir l'SGBD en temps d'execució (no en temps de compilació), seguint la proposta d'ODBC.

En el moment d'escriure aquest material, l'especificació actual de l'API JDBC és la 4.0. Malgrat això, els conceptes bàsics per a programar utilitzant JDBC ja estaven definits en la versió 1.0, i s'han mantingut al llarg de les versions. Així doncs, aquest material està basat, principalment, en la versió 1.0 de JDBC i, puntualment, en canvis introduïts en versions posteriors.

Vegem ara un resum de les versions de JDBC i els canvis que han anat introduint:

**1) JDBC 1.0 (JDK 1.1).** Definit en el *package* `java.sql` conté les classes principals: `DriverManager` i `Driver` (per a gestionar els *drivers*), `Connection` (un objecte per a cadascuna de les BD amb les quals connectem), `Statement` (un objecte per a definir cadascuna de les sentències SQL que volem executar en la BD), `ResultSet` (rep els resultats d'una consulta i ofereix mètodes per a poder-los consultar amb desplaçaments de cursor endavant). Així mateix, defineix els tipus de dades (enters, reals, cadenes, binari i data i hora) que es poden passar entre l'aplicació i la BD.

També inclou altres classes com ara `PreparedStatement` (alternativa per a executar sentències SQL), definició de transaccions, `StoredProcedures` (execució de procediments emmagatzemats) i `SQLException` i `SQLWarning` (per al tractament d'errors).

**2) JDBC 2.0 (JDK 1.2)**. Introdueix millores a la classe `ResultSet`. S'amplien els tipus de desplaçament de cursor i permet, també, desplaçaments endarrere i a una posició concreta. Així mateix, ofereix la possibilitat de fer actualitzacions de la fila apuntada pel cursor mitjançant mètodes (sense utilitzar sentències SQL).

Permet fer actualitzacions en la BD en *batch*, i incorpora els tipus de dades previstos en l'SQL:1999 (`BLOB`, `CLOB`, `ARRAY`...).

Incorpora un nou *package* d'extensió estàndard anomenat `javax.sql`. Conté la classe `RowSet` (variant de `ResultSet` per a integrar en components `JavaBeans`), *Java Naming Directory Interface* (JNDI), que permet connectar-nos a una BD a partir d'un nom (sense haver de codificar ni el *driver* ni la *url* de la BD), *pool* de connexions (conjunts de connexions permanentment obertes que es comparteixen entre diferents clients) i transaccions distribuïdes.

**3) JDBC 3.0 (JDK 1.4)**. Implementa l'estàndard SQL:1999, possibilita l'accés a metadades de la BD, amplia els mètodes per a tipus de dades SQL:1999, incorpora nous tipus de dades (`DATALINK` i `BOOLEAN`), facilita la consulta de columnes autoincrementades després de fer una inserció, permet obtenir múltiples `ResultSets` per a cada `Statement`, millora els *pools* de connexions, incorpora *pools* de `PreparedStatements` i permet transaccions amb `savePoints` (punts de recuperació intermedis).

**4) JDBC 4.0 (JDK 1.6)**. Simplifica la càrrega de *drivers*, millora els `ResultSets` (`RowSet`, `WebRowSet`), incorpora novetats SQL:2003 (`NCHAR`, `NVARCHAR`, `LONGNVARCHAR`, `NCLOB`) per poder treballar amb textos de qualsevol alfabet, dóna suport per a excepcions encadenades i facilita el treball amb tipus de dades XML.

**PostgreSQL i JDBC 4.0**

A tall d'exemple, mentre s'elabora aquest material, el *driver* de PostgreSQL ofereix una implementació raonable del JDBC 3.0, i alguna extensió pròpia. La versió JDBC 4.0 també s'ofereix però no és completa.

## 2.1. *Drivers* JDBC

L'API JDBC és format per dos *packages*, `java.sql` i `javax.sql`. Aquests *packages* contenen les classes i les interfícies necessàries per a accedir a diferents orígens de dades, però no és complet. Manquen les classes que implementen les interfícies JDBC per a accedir a cadascun dels SGBD amb els quals volem treballar. És el que anomenem *drivers JDBC*. Entre d'altres, al *package* `java.sql` es defineix la interfície `Driver`, que és la interfície principal que un *driver* ha d'implementar.

Els *drivers* s'hauran d'instal·lar en tots els equips on vulguem executar aplicacions que utilitzin JDBC. Com que els *drivers* són implementacions fetes en Java, com a tals, seran portables a qualsevol plataforma que accepti Java i, per tant, no hi hauríem de tenir cap problema. Més endavant ja veurem que, malauradament, això no sempre és així.

Per exemple, per a executar una aplicació que usi PostgreSQL caldrà que instal·lem els *drivers* JDBC de PostgreSQL. Sense aquests *drivers* podríem compilar l'aplicació, però es generaria un error en temps d'execució (recordem que, seguint el principi d'ODBC, els *drivers* es carregen en temps d'execució).

Per a instal·lar manualment un *driver* en un equip, cal descarregar els fitxers necessaris (normalment un fitxer `.jar`). Tot seguit cal modificar el `CLASSPATH` perquè inclogui els fitxers que hem descarregat.

## 2.2. Tipus de *drivers*

Podem trobar *drivers* JDBC per a accedir a un ampli ventall de SGBD. Ens pot sorprendre trobar més d'un *driver* per accedir a un mateix SGBD, especialment perquè tots permeten fer el mateix. Quin sentit pot tenir?

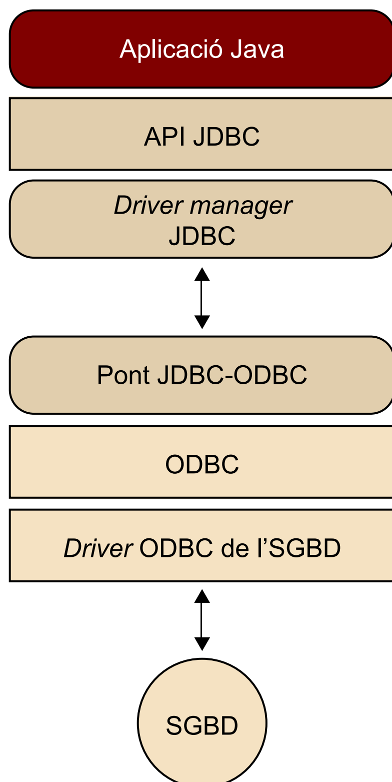
Els diferents *drivers* que permeten accedir a un mateix SGBD ofereixen les mateixes classes i mètodes, però utilitzen camins diferents per a comunicar-se amb la BD. Podem trobar quatre tipus d'accés diferents:

- **Tipus 1**. Per mitjà d'ODBC, mitjançant un pont JDBC-ODBC.
- **Tipus 2**. Per mitjà d'una biblioteca del SGBD (que anomenem *libSGBD*), mitjançant un pont JDBC-libSGBD.
- **Tipus 3**. De JDBC a SGBD passant per un intermediari. Aquest intermediari acostuma a fer servir *drivers* de tipus 1, 2 o 4 per a comunicar-se amb SGBD.
- **Tipus 4**. Directe de JDBC a SGBD.

### 2.2.1. Pont JDBC-ODBC (tipus 1)

Aquest *driver* accedeix a la BD per mitjà d'un *driver* ODBC. És l'únic que està inclòs a l'API JDBC i, per tant, no requereix la instal·lació de cap *driver* JDBC addicional.

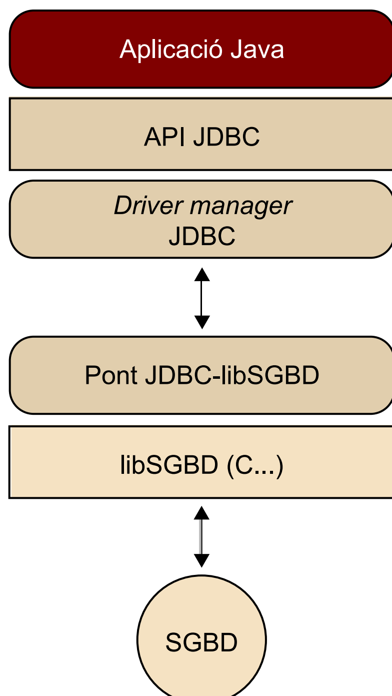
*Driver* JDBC tipus 1




### 2.2.2. Pont JDBC-libSGBD (tipus 2)

En aquest cas, accedim a la BD per mitjà d'una biblioteca específica per a cada SGBD, normalment, escrita en llenguatge C. Els *drivers* JDBC només fan de pont entre Java i aquesta biblioteca.

*Driver* JDBC tipus 2



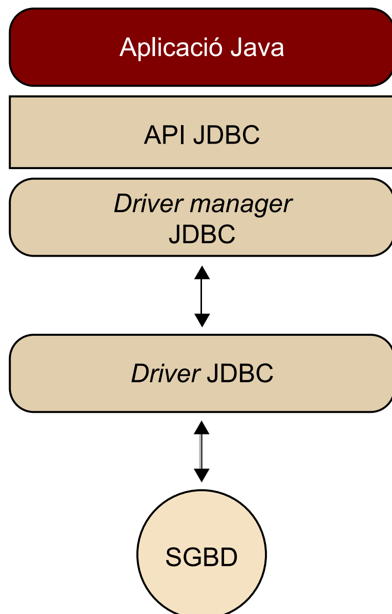
### 2.2.3. Directe JDBC-SGBD (tipus 4)

Accedim a la BD, exclusivament, mitjançant un *driver* JDBC que es comunica d'una manera directa amb l'SGBD per la xarxa.

**Nota**

Els *drivers* JDBC de PostgreSQL només s'ofereixen de tipus 4.

*Driver* JDBC tipus 4

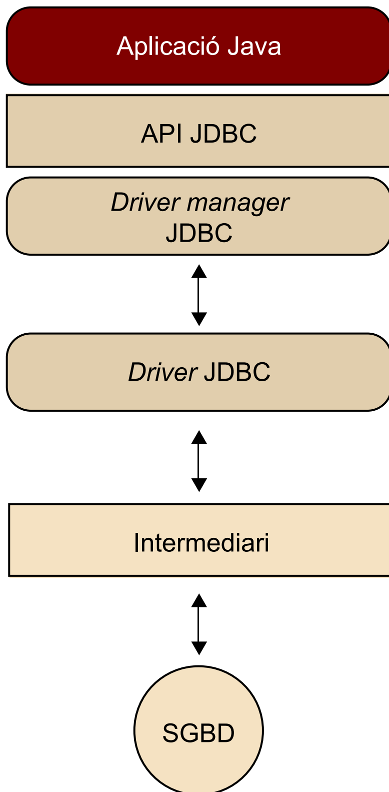



### 2.2.4. JDBC-Intermediari (tipus 3)

Ens comuniquem, exclusivament, amb un intermediari per la xarxa. Aquest intermediari s'encarrega de comunicar-se amb l'SGBD i, normalment, tant l'intermediari com l'SGBD estan instal·lats en el mateix equip o en la mateixa xarxa local.

L'intermediari pot estar implementat en qualsevol llenguatge de programació. Si està implementat en Java (que no és imprescindible) normalment utilitzarà *drivers* JDBC de tipus 1, 2 o 4 per a accedir a l'SGBD concret.

*Driver* JDBC tipus 3




### 2.3. Comparativa dels quatre tipus de *drivers*

Analitzem els quatre tipus de *drivers* des de punts de vista diferents:

**1) Portabilitat**. Els *drivers* de tipus 1 i 2 van aparèixer en els orígens de JDBC amb la intenció d'aprofitar els *drivers* ODBC i les biblioteques d'accés a l'SGBD que ja hi havia implementades. Això garantia la introducció de JDBC sense cap esforç per part dels fabricants de SGBD, però en limitava la portabilitat.

Per a poder utilitzar *drivers* de tipus 1 o 2 cal que l'equip on s'executa l'aplicació tingui instal·lats *drivers* ODBC o biblioteques d'accés a SGBD i, tant els uns com els altres, són dependents de la plataforma. Aquest fet els desautoritza i els restringeix a casos experimentals o d'inexistència de *drivers* JDBC.

Els *drivers* de tipus 3 i 4 també van aparèixer en els orígens de JDBC, tot i que inicialment hi havia pocs SGBD que els oferissin. D'una manera gradual, els fabricants de SGBD han anat publicant *drivers* de tipus 3 i 4 i, actualment, són les opcions recomanades.

A diferència dels *drivers* de tipus 1 i 2, els de tipus 3 i 4 garanteixen la portabilitat d'una aplicació a totes les plataformes que suporten Java. En aquest cas, els *drivers* estan escrits íntegrament en Java i, per tant, són plenament portables.

Addicionalment, els *drivers* de tipus 3 permeten ocultar l'SGBD als clients (ja que aquests es comuniquen exclusivament amb l'intermediari). Per tant, podem portar l'aplicació d'un SGBD a un altre modificant, tan sols, el codi de l'intermediari. Les aplicacions clients resten inalterades.

**2) Tipus i versions de JDBC implementades**. No tots els fabricants de SGBD ofereixen *drivers* de tots els tipus i els que s'ofereixen no sempre implementen la darrera versió de JDBC.

Els *drivers* de tipus 1 queden una mica al marge d'aquesta incertesa. Recordem que estan basats en *drivers* ODBC i que, per tant, estan limitats a les funcionalitats d'ODBC. Per aquesta raó només implementen JDBC2.0.

Per a la resta de casos, cal esbrinar quins tipus de *driver* ofereix el fabricant de SGBD que ens interessa, i quina versió JDBC implementa. A més, podem trobar *drivers* que implementin parcialment alguna versió.

En resum, tot i que en principi els *drivers* de tipus 3 i 4 garanteixen portabilitat, a la pràctica podem tenir algunes sorpreses.

**3) Nivell de seguretat**. Les aplicacions no sempre s'executen en el mateix equip de la BD. Normalment s'han de connectar a la BD per mitjà de la xarxa. En escenaris simples es connecten per la xarxa local, però en d'altres mitjançant Internet.

En escenaris d'Internet es considera que els *drivers* de tipus 1, 2 i 4 tenen un nivell de seguretat baix. Per tal que les aplicacions es puguin comunicar amb l'SGBD, cal exposar l'SGBD a Internet, de manera que s'obre la porta a possibles atacs.

Els *drivers* de tipus 3 permeten que l'aplicació es connecti a un intermediari i no a l'SGBD. Això possibilita ocultar l'SGBD d'Internet i, per tant, augmentar el nivell de seguretat (sempre que l'intermediari estigui implementat correctament!).

Així, a nivell de seguretat, n'hi ha prou d'utilitzar *drivers* de tipus 4 en escenaris de xarxa local (o de tipus 1 o 2, si no hi ha alternativa). En escenaris d'Internet es recomana l'ús de *drivers* de tipus 3.

En conclusió, els *drivers* de tipus 3 ofereixen les millors prestacions, tant de portabilitat, de seguretat com fins i tot d'eficiència. En contrapartida, l'oferta és escassa (sovint cal implementar l'intermediari) i tenen una arquitectura més complexa (necessiten que l'intermediari s'estigui executant en un servidor).

Els *drivers* de tipus 4 tenen un nivell de prestacions correcte i, a diferència dels de tipus 3, podem trobar una àmplia oferta en el mercat. Permeten obtenir bons resultats sense massa complicacions.

Els *drivers* de tipus 1 i 2 serien l'alternativa en cas de no trobar *drivers* de tipus 3 o 4, i en situacions experimentals.

# 3. Programació amb JDBC

Ara ens centrarem en el vessant més pràctic del mòdul didàctic. Suposem que volem desenvolupar una aplicació en Java i que ja hem decidit quin SGBD i *driver* utilitzarem. Vegem com s'usa l'API JDBC, és a dir, com es programa mitjançant les classes d'aquesta biblioteca.

Començarem veient com s'estableix connexió amb una BD. A continuació aprendrem la manera d'enviar consultes, tractar els resultats, fer modificacions i treballar amb procediments emmagatzemats. Finalment veurem la manera de gestionar les transaccions i el tractament d'errors.

## 3.1. Connexió a una base de dades

El primer pas per treballar amb una BD és la connexió. Per a poder-nos connectar cal seguir quatre passos:

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex001CreacioBDMail.java*.

**1) Importar els *packages* necessaris**. Tal com s'ha comentat, l'API JDBC és formada per dos *packages*, `java.sql` i `javax.sql`. El primer conté les classes i interfícies essencials (inclou les classes `Driver`, `Connection`, `Statement`, `ResultSet`, `PreparedStatement`, `CallableStatement`, principalment) i l'haurem d'importar sempre. El segon és l'extensió estàndard i conté classes més especialitzades que s'escapen dels objectius d'aquest mòdul didàctic (`RowSet`, `DataSource` i `PooledConnection`, entre d'altres).

**2) Carregar el *driver* adequat**. La manera tradicional de carregar un *driver* és forçant la càrrega del *driver* a partir del seu nom, utilitzant el mètode `forName` de la classe `Class`. Per exemple, per carregar el *driver* del PostgreSQL faríem el següent:

```
Class.forName( "org.postgresql.Driver" );
```

Aquest nom identifica la classe `Driver` del *package* `org.postgresql` (recordem que és la classe que implementa la interfície `java.sql.Driver`). Si volem utilitzar un altre *driver* haurem de fer una mica d'investigació per esbrinar el nom del *package* i el nom de la classe que implementa la interfície `java.sql.Driver`.

Carregar una classe a partir del seu nom pot fallar si el *classloader* (objecte responsable de carregar les classes necessàries per a l'execució d'un programa) no és capaç de trobar cap classe amb aquest nom. Per tant, ens hem d'assegurar

que el `CLASSPATH` apunti al fitxer (normalment `.jar`) que conté el *driver*. En cas de no trobar-la, es genera una excepció de tipus `ClassNotFoundException`.

En la versió JDBC 4.0 es proposa delegar la responsabilitat de carregar el *driver* al `DriverManager`, que serà l'encarregat de buscar el *driver* en els directoris o fitxers `jar` definits al `CLASSPATH` quan siguin necessaris.

3) **Obrir la connexió**. Tot i que el concepte d'obrir connexió sigui simple, amaga una certa complexitat quan hem d'indicar la BD que volem obrir.

```
String dbURL="jdbc:postgresql:bdMail";
Connection conn = DriverManager.getConnection( dbURL,
  "usuari","contrasenya");
```

L'encarregat d'obrir una connexió amb una BD és el `DriverManager`, per mitjà del mètode `getConnection` i requereix tres paràmetres:

**a)** El primer és l'anomenat `url` i identifica la BD a la qual ens volem connectar. És una cadena formada per tres parts: la primera sempre és `jdbc` i la resta, variables.

```
jdbc:<subprotocol>:<subname>
```

El `subprotocol` és el nom del *driver* que utilitzarem per a connectar-nos. Un altre cop, cal fer una mica d'investigació.

El `subname` serveix per a identificar la BD pròpiament. El seu format depèn del *driver* que emprem i, per tant, no té un format estàndard. Aquest és un tercer punt d'investigació.

En els casos més explícits, identifica el servidor (on hi ha l'SGBD), el port on escolta l'SGBD i el nom de la BD. Si no indiquem servidor s'entén que és el mateix ordinador (`localhost`) i, si tampoc no ho fem amb el port, s'entén que és el port per defecte de l'SGBD.

En l'exemple anterior ens estaríem connectant a una BD PostgreSQL que hi ha en el mateix ordinador, escoltant el port per defecte i que s'anomena `bdMail`.

**b)** El segon i tercer paràmetres corresponen al nom d'usuari de la BD i a la contrasenya corresponent. Cal assegurar-nos que ens connectem amb un usuari que tingui suficients privilegis per a executar les sentències SQL que vinguin a continuació.

4) **Tancar la connexió**. Sens dubte, és l'operació més senzilla de les vistes fins ara. Simplement cal cridar el mètode `close` de la connexió que volem tancar.

```
conn.close();
```

Si no tanquem la connexió ho farà el *garbage collector* quan destrueixi l'objecte connexió.

En tot cas, en aplicacions client és molt important tancar les connexions quan ja no les volem utilitzar; així aconseguim que el servidor alliberi recursos i que els pugui dedicar a un altre client.

Arribats en aquest punt ens podem plantejar si els passos que anem seguint són elegants. Hem indicat el nom del *driver* a l'hora de carregar-lo, i l'hem de tornar a indicar a l'hora de connectar-nos. És necessària aquesta redundància? En la versió JDBC 4.0 es delega la càrrega de *drivers* al `DriverManager`, de manera que només cal indicar el *driver* en el moment de crear la connexió.

També ens podem plantejar si és convenient indicar l'equip, el port, el nom de la BD, el nom d'usuari i la contrasenya en el codi font. Ho és? No massa, especialment si volem distribuir l'aplicació sense haver-la de recompilar cada cop! De fet, a partir de JDBC 2.0 ja es proposa utilitzar una interfície anomenada `DataSource` per a desvincular el codi font dels detalls de connexió.

## 3.2. Consultes i modificacions bàsiques

Entrem ara a la part més interessant de la programació en JDBC o, com a mínim, la que ens ocuparà més temps. Veurem la manera de fer consultes a la BD i de modificar-ne les dades per mitjà de sentències SQL.

Començarem enviant consultes o modificacions mitjançant la classe `Statement` i tractant els resultats de les consultes amb la classe `ResultSet`. Amb això tindrem una idea bàsica de comunicació entre aplicació i BD.

**Classe `Statement`**

Realment, la classe `Statement` no envia consultes o modificacions. Són les instàncies d'aquesta classe, és a dir, els objectes, que realment envien consultes o modificacions. Apliquem aquest abús de llenguatge aquí i a la resta del material per millorar la claredat del text.

### 3.2.1. Execució de sentències SQL: la classe `Statement`

El procés per a fer una consulta o una modificació arrenca d'allà mateix. Hem de crear un objecte de tipus `Statement` que contindrà la sentència SQL de consulta (`SELECT`), modificació de dades (`INSERT`, `UPDATE` i `DELETE`) o modificació de l'estructura de BD (`CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`, etc.).

La responsabilitat de crear nous objectes de tipus `Statement` és de la connexió a la qual volem enviar la sentència SQL.

```
Statement st = conn.createStatement();
```

Un cop tenim l'objecte de tipus `Statement` creat, utilitzarem el mètode `executeQuery` per a les consultes i `executeUpdate` per a les modificacions. En el cas de les consultes, el mètode `executeQuery` retorna un objecte `ResultSet` que ens permetrà accedir a les dades consultades. Ho tractarem en el subapartat següent.

```
ResultSet rs;
rs = st.executeQuery("SELECT * FROM Usuaris");
```

En el cas de modificacions, el mètode `executeUpdate` retorna un enter. Aquest enter indica el nombre de files afectades. En el cas de modificar l'estructura de la BD (execució de sentències SQL de tipus DDL), retorna un 0.

```
st.executeUpdate("DROP TABLE Usuaris");
st.executeUpdate("CREATE TABLE Usuaris ("+
  "nom_usuari VARCHAR(10) PRIMARY KEY, "+
  "contrasenya VARCHAR(10), nom VARCHAR(20), "+
  "cognoms VARCHAR(40))");
st.executeUpdate("INSERT INTO Usuaris "+
  "(nom_usuari,contrasenya,nom,cognoms) VALUES "+
  "('mPalau','1234','Manel','Palau Roca')");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex001CreacioBDMail.java*.

Les tres sentències SQL d'aquest exemple sempre fan el mateix. En alguns casos això ja és suficient (per exemple, quan creem la taula `Usuaris`), però normalment no serà així.

La darrera sentència de l'exemple afegeix l'usuari Manel a la taula `Usuaris`, però és un cas poc habitual. Normalment, quan s'afegeixen files en una BD, els valors de les columnes els introdueix l'usuari de l'aplicació en temps d'execució o es carreguen des d'un fitxer. En tot cas, són valors que no es coneixen en temps de compilació.

Per aconseguir executar una sentència SQL amb valors canviants concatenarem les parts fixes de la sentència amb les parts canviants (que se substituiran per variables). El codi queda una mica il·legible, però amb el temps ens hi acabarem acostumant.

```
String nomUsuari,contrasenya,nom,cognoms;
...
st.executeUpdate("INSERT INTO Usuaris "+
  "(nom_usuari,contrasenya,nom,cognoms) VALUES "+
  "('"+nomUsuari+"','"+contrasenya+"','"+nom+"','"+
  cognoms+"')");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex004InsertStatementVars.java*.

Podem aplicar aquest mateix patró per fer consultes SQL utilitzant la clàusula WHERE. Per exemple, ens pot interessar consultar els missatges del fòrum que l'usuari de l'aplicació seleccioni. Fixem-nos que la part canviant correspon a una columna numèrica i que, per tant, no està envoltada de cometes simples!

**Nota**

Fixeu-vos que les cometes simples per a definir textos en SQL es mantenen com a part fixa de la instrucció!

```
int codiForum;
...
ResultSet rs = st.executeQuery("SELECT * "+
   "FROM Missatges WHERE codi_forum="+codiForum);
```

En el cas de columnes de tipus data hem de fer una atenció especial a la interpretació que en farà l'SGBD. Depenent de la configuració local, l'SGBD pot interpretar les dates en format dd/mm/yyyy o mm/dd/yyyy. Per a assegurar la interpretació correcta usarem funcions de l'SGBD per a fer la conversió. En el cas de PostgreSQL, per exemple, utilitzarem la funció to_timestamp.

```
//La interpretació dependrà de la config. de l'SGBD
st.executeUpdate("INSERT INTO Lectures "+
  "VALUES(1,'mPalau','3/4/2010 16:19')");

//assegurem que l'SGBD ho interpreti correctament!
st.executeUpdate("INSERT INTO Lectures "+
  "VALUES(1,'cMas', to_timestamp('3/4/2010 16:19'"+
  ",'DD/MM/YYYY HH24:MI'))");
```

Per definir sentències SQL dinàmiques amb paràmetres de tipus temps haurem de tenir en compte el format de les dates en Java. Per exemple, el mètode toString de la classe Timestamp retorna un data en format YYYY-MM-DD hh:mm::ss. El codi necessari seria:

```
st.executeUpdate("INSERT INTO Lectures "+
  "VALUES("+codiMissatge+",'"+nomUsuari+"',"+
  "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");
```

### 3.2.2. Tractament de resultats: la classe ResultSet

Com ja hem dit, la classe ResultSet ens permetrà accedir als resultats de les consultes. Aquest accés, però, no és lliure i ens haurem de cenyir a les restriccions següents:

- Simultàniament només podem accedir a una sola fila. Per poder accedir a totes les files haurem de fer un recorregut i, en cada iteració, accedir a una de les files.
- El recorregut, per defecte, només pot anar endavant.
- Durant el recorregut, d'entrada, només podem consultar les files. No les podem modificar.

Per tant, la classe ResultSet ens oferirà mètodes per poder fer un recorregut per les files de la consulta i, en cada iteració, consultar el valor de les columnes de la fila actual tenint en compte que:

- Podem consultar el valor de les columnes a partir del nom corresponent o a partir d'un enter que representa la posició de la columna dins de la taula (començant per 1).

- Disposem de mètodes diferents per a cada tipus de dades de les columnes que es vol consultar.

| Tipus estàndard SQL | Mètode getTipus |
|---|---|
| CHAR | getString |
| VARCHAR | getString |
| SMALLINT | getShort |
| INTEGER | getInt |
| FLOAT | getFloat / getDouble |
| DOUBLE | getDouble |
| DECIMAL | getBigDecimal |
| DATE | getDate |
| ESTAFI | getTime |
| MONEY | getDouble |

**Exemple**

En l'exemple següent consultem les dades dels usuaris (emmagatzemades a la taula Usuaris de la nostra BD de referència), fem un recorregut i, en cada iteració, mostrem la columna número 1 (que correspon a la columna nom_usuari) i els cognoms.

```
rs = st.executeQuery("SELECT * FROM Usuaris");
while (rs.next()) {
  System.out.print(rs.getString(1)+"--"+
            rs.getString("cognoms"));
}
rs.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex005ConsultaStatement.java*.

S'intueix que darrere d'un ResultSet hi ha un cursor que apunta a la fila actual. Quan es crea un objecte ResultSet, el cursor apunta a la posició anterior a la primera fila i, cada cop que executem el mètode next, el cursor avança. Quan el mètode next no troba cap més fila, retorna fals i el recorregut finalitza. Però, en realitat, les coses no són ben bé així.

Cada cop que JDBC demana dades a la BD (el què es coneix com a *fetch*) no rep una fila i prou. Per qüestions de rendiment, la BD envia unes quantes files. A més, el nombre de files que s'envien depèn de cada *driver*.

En el cas del *driver* JDBC de PostgreSQL, per defecte, s'envien totes les dades de la consulta de cop. Mentre no són tractades, aquestes dades es guarden a l'equip client en una memòria cau. Hem de vigilar que el volum de dades de la consulta no sigui massa gran, sinó podem exhaurir la memòria de l'equip client!

En relació amb aquest tema, hem de tenir cura de fer consultes de les dades que ens siguin estrictament necessàries. En el cas anterior tenim un exemple clar de consulta ineficient. No té cap sentit fer una consulta de totes les columnes de la taula `Usuaris` si després només en mostrem dues. El mateix codi refinat seria:

```
rs = st.executeQuery("SELECT nom_usuari,cognoms "+
        "FROM Usuaris");
while (rs.next()) {
  System.out.print(rs.getString(1)+"--"+
        rs.getString("cognoms"));
}
rs.close();
```

**Consultes ineficients**

Per a millorar la llegibilitat dels exemples, en aquest material hem optat per utilitzar consultes que fan servir *. Heu de tenir present que en un cas pràctic heu d'evitar-ho perquè, en general, poden ser ineficients.

### 3.3. Consultes i modificacions avançades

Tal com hem vist, en la versió JDBC 2.0 es va introduir la possibilitat de fer recorreguts millorats (endavant, endarrere i desplaçaments directes a qualsevol posició) i també la possibilitat de modificar les files mitjançant mètodes (sense utilitzar l'SQL).

Per a permetre aquestes noves funcionalitats, cal crear l'objecte `Statement` amb el mateix mètode `createStatement`, però amb dos paràmetres que determinen el tipus de moviment i el tipus d'operacions permeses (vegeu la taula següent).

| | |
|---|---|
| **Tipus de moviment** | `TYPE_FORWARD_ONLY`: és el tipus de moviment assignat per defecte. Només permet fer un sol recorregut cap endavant. |
| | `TYPE_SCROLL_INSENSITIVE`: permet llibertat de moviments (endavant, endarrere) tants cops com calgui. |
| | `TYPE_SCROLL_SENSITIVE`: com l'anterior, però reflecteix els canvis que es van produint en la BD mentre està actiu. S'entén que aquests canvis normalment són generats per l'execució simultània d'altres aplicacions en altres equips. |
| **Tipus d'operacions admeses** | `CONCUR_READ_ONLY`: només permet fer consultes i és l'opció per defecte. |
| | `CONCUR_UPDATABLE`: permet fer consultes i modificacions. |

En total tenim sis combinacions possibles, però en la pràctica no tots els *drivers* les permeten. Per exemple, el *driver* de PostgreSQL (i no és l'únic) no possibilita el tipus de moviment sensitiu i, per tant, ofereix quatre combinacions. Per a solucionar aquesta limitació es recomana repetir les consultes cada cop que es vulguin tenir les dades actualitzades.

### 3.3.1. `ResultSet` amb llibertat de moviments

Per a treballar amb `ResultSet` lliures de moviments cal triar l'opció `TYPE_SCROLL_INSENSITIVE` o `TYPE_SCROLL_SENSITIVE` (si el *driver* ho permet). Així, podrem fer recorreguts:

- **Endavant**: amb el mètode `beforeFirst()` i `next()`. El primer no és necessari, ja que és la posició inicial del `ResultSet` i el segon ja l'hem vist.

- **Endarrere**: amb els mètodes `afterLast()` i `previous()`.

**Exemple**

L'exemple següent fa un recorregut endarrere per les files de la taula `Usuaris`. En cada iteració mostra la clau primària i els cognoms dels usuaris.

```
st=conn.createStatement(
  ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_READ_ONLY);

rs = st.executeQuery("SELECT * FROM Usuaris");
rs.afterLast();
while (rs.previous())
  System.out.println(rs.getString(1)+"--"+
    rs.getString("cognoms"));
rs.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex006ConsultaStatementScrollReves.java.*

- **Aleatoris**: amb els mètodes `first()`, `last()`, `absolute(int n)` i `relative(int n)`. Permeten moure'ns a la primera, a la darrera, a una posició concreta o a una de relativa, respectivament. Si intentem moure'ns fora de rang genera un error.

**Exemple**

A continuació tenim un exemple per a veure com s'utilitzen els diversos mètodes de posicionament.

```
st=conn.createStatement(
  ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
rs = st.executeQuery("SELECT * FROM Usuaris");
rs.last();
...
rs.relative(-1);
...
rs.first();
...
rs.absolute(3);
...
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex007ConsultaStatementScrollAleatori.java.*

### 3.3.2. Modificacions per mitjà d'un `ResultSet`

Amb l'opció `CONCUR_UPDATABLE` tenim la possibilitat d'actualitzar la BD a mesura que anem recorrent el `ResultSet`, i sense utilitzar sentències SQL.

En la pràctica, però, poden aparèixer problemes. El motiu de fons és que l'SGBD ha de poder propagar automàticament la modificació des de les dades llegides (i que es troben en el ResultSet) cap a les taules que emmagatzemen aquestes dades. Aquesta propagació automàtica dels canvis, en general, només es pot fer quan la sentència SELECT que està lligada al ResultSet està basada en una única taula i, entre les dades llegides, s'inclou la clau primària.

Un cop superada aquesta problemàtica, els canvis possibles són modificació, inserció i esborraments.

**Modificació**

Un cop situats a la fila que volem modificar, disposem de mètodes per a canviar el valor de les columnes. Són els mètodes updateXXX i són antagònics als getXXX.

Després de modificar les columnes d'una fila cal enviar els canvis a la BD amb el mètode updateRow(). Si ens movem de fila sense executar aquest mètode, els canvis normalment es perdran (depèn del *driver* que utilitzem). Per contra, si volem descartar els canvis fets, tenim el mètode cancelRowUpdates(). L'exemple següent afegeix un "." al final de cada nom d'usuari.

```
st = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_UPDATABLE);
rs = st.executeQuery("SELECT * FROM Usuaris");
while (rs.next())
{
  rs.updateString("nom",rs.getString("nom")+".");
  rs.updateRow();
}
rs.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex008ResultSetUpdatable.java*.

També podem fer modificacions sobre un ResultSet amb llibertat de moviments.

**Exemple**

L'exemple següent se situa a la darrera fila i obté el número de fila amb getRow(). Així sabem el nombre total de files consultades. A continuació se situa sobre una posició aleatòria entre la primera i la darrera fila i afegeix un "." al final del nom.

```
st = conn.createStatement(
      ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
rs = st.executeQuery("SELECT * FROM Usuaris");
rs.last();
int nFiles=rs.getRow();
int pos=(int)(Math.random()*nFiles)+1;
rs.absolute(pos);
rs.updateString("nom",rs.getString("nom")+".");
rs.updateRow();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex009ResultSetUpdatableAleatori.java*.

#### Inserció

Per poder inserir de primer ens hem de moure a una nova fila mitjançant el mètode `moveToInsertRow()`. A partir d'aquí, assignem el valor de les diverses columnes amb `updateXXX` i, per afegir cridem `insertRow()`.

```
rs.moveToInsertRow();
rs.updateString("nom_usuari","root");
rs.updateString("contrasenya","super");
rs.updateString("nom","administrador");
rs.updateString("cognoms","");
rs.insertRow();
```

#### Esborraments

Els esborraments són el cas més simple. Només cal situar-se a la fila adequada i cridar el mètode `deleteRow()`. L'exemple següent se situa a la darrera fila i l'esborra.

```
//Eliminem la darrera fila
rs.last();
rs.deleteRow();
```

### 3.4. Consultes i modificacions amb `PreparedStatement`

La classe `PreparedStatement` és una alternativa a la classe `Statement`. Permeten fer el mateix, consultes i modificacions a la BD, però canvia la manera de fer-ho.

D'entrada, un `PreparedStatement` no es pot reutilitzar per a executar una segona sentència SQL. Cada sentència anirà lligada a un objecte `PreparedStatement` diferent.

Les sentències SQL que definim en un `PreparedStatement` normalment són incompletes, en el sentit que poden tenir alguns valors indefinits. Utilitzarem un `?` per a cada valor indefinit de la sentència.

```
"SELECT * FROM USUARIS WHERE nom_usuari=?"
```

Una sentència incompleta no es pot executar, però sí que es pot enviar a l'SGBD i fer alguns dels passos necessaris per a dur-la a terme (ho podem entendre com una precompilació). L'objectiu és clar. Tan bon punt sapiguem els valors que completen la sentència, aquesta es podrà executar ràpidament, ja que una part de la feina ja s'haurà fet amb antelació.

L'escenari ideal dels `PreparedStatement` és la situació en què una sentència s'ha d'executar repetidament, canviant només alguns valors; per exemple, si volem afegir unes quantes files a una taula. S'entén que el rendiment és superior que si utilitzem `Statement` independents, ja que l'SGBD analitza un sol cop la sentència SQL i l'executa tantes vegades com calgui.

Si una sentència només s'ha d'executar un sol cop, utilitzar `PreparedStatements` deixa de ser tan beneficiós i el rendiment s'equipara a la utilització de `Statement` independents.

**Vegeu també**

Més endavant, en el subapartat 3.5.6, veurem l'SQL Injection i altres beneficis derivats d'utilitzar `PreparedStatements`.

Per a assignar els valors d'un `PreparedStatement` tenim un seguit de mètodes `setXXX`, de funcionament idèntic a l'`updateXXX` dels `ResultSets` modificables amb l'excepció que els paràmetres només es poden indexar a partir de la posició i no amb el nom de la columna.

```
INSERT INTO FORUMS (codi_forum,nom) VALUES (paràmetre1,paràmetre2)
```

**Exemple**

Vegem com podem afegir uns quants fòrums a la BD de referència. Per a aquest exemple, suposem que tenim una classe `Forum` implementada amb els mètodes suficients. Fixem-nos, també, en el mètode `clearParameters` que esborra el valor dels paràmetres i ens prepara per a l'execució següent.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex010PreparedStatement Insert.java*.

```
Forum[] dades={ new Forum(3,"PostgreSQL"),
  new Forum(4,"ODBC"), new Forum(5,"Oracle") };

sql="INSERT INTO FORUMS(codi_forum,nom) VALUES (?,?)";
PreparedStatement pst = conn.prepareStatement(sql);
for(Forum f:dades)
{
  pst.clearParameters();
  pst.setInt(1,f.getCodiForum());
  pst.setString(2,f.getNom());
  pst.executeUpdate();
}
```

## 3.5. Consideracions addicionals sobre consultes i modificacions

Fins ara hem après la manera de consultar o modificar dades d'una BD mitjançant sentències SQL. Ara ens dedicarem a veure la manera de resoldre situacions particulars que apareixen amb més o menys freqüència.

### 3.5.1. Tractament de valors nuls

En els exemples que hem vist no hem fet cap consideració especial per als valors nuls, i de tant en tant cal fer-ne. Totes les columnes que no tenen la restricció `NOT NULL` ocasionalment poden contenir un valor nul.

Hi ha quatre situacions en què poden aparèixer complicacions degudes als valors nuls: la lectura de valors nuls d'un `ResultSet`, l'ús de valors nuls en sentències SQL dinàmiques, l'ús de valors nuls en la clàusula `WHERE` i la modificació d'un `ResultSet` amb valors nuls.

### Lectura de valors nuls d'un `ResultSet`

Cada cop que fem una consulta a la BD, la nostra aplicació ha de vigilar de no rebre valors nuls i protegir-se si s'escau.

En la majoria de casos no cal usar cap mètode especial per a detectar si un valor és nul. Els mètodes `getString`, `getBigDecimal`, `getBytes`, `getDate`, `getTime`, `getTimestamp`, `getAsciiStream`, `getCharacterStream`, `getUnicodeStream`, `getBinaryStream`, `getObject`, `getArray`, `getBlob`, `getClob` i `getRef` retornen un objecte. Si el valor consultat en la BD és nul, l'objecte Java també ho serà.

Cal posar una atenció especial en els mètodes que no retornen objectes. En cas de valor nul, els mètodes `getByte`, `getShort`, `getInt`, `getLong`, `getFloat` i `getDouble` retornen un 0. I el mètode `getBoolean` retorna fals. En aquests casos hem d'utilitzar un mètode addicional, `wasNull()` per a esbrinar si el 0 i el valor fals corresponen realment a un 0 i un fals, respectivament, o a un valor nul.

**Exemple**

El codi següent obté els missatges dels fòrums que inicien nous fils, és a dir, els que tenen el fil nul.

```
rs=st.executeQuery("SELECT * FROM Missatges");
while (rs.next())
{
  int fil=rs.getInt("fil");
  if (rs.wasNull()) {
    System.out.println(rs.getInt("codi_forum")+
      "--"+rs.getString("titol"));
  }
}
rs.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex011Nulls.java*.

### Ús de valors nuls en sentències SQL dinàmiques

Per a afegir un missatge en un fòrum, podríem tenir un codi semblant a aquest:

**Vegeu també**

En el subapartat 3.2.1 ja hem vist la manera de definir una sentència SQL amb valors canviants concatenant cadenes (que contenen la part fixa) amb variables.

```
int codiForum, ordre, fil;
String autor,titol,text;
...
st.executeUpdate("INSERT INTO Missatges "+
  "VALUES("+codiMissatge+","+codiForum+",'"+autor+
  "','"+titol+"','"+text+"',"+fil+")");
```

Aquest codi és correcte, però no permet tractar valors nuls. Per exemple:

- No permet obrir un fil nou. O, dit d'una altra manera, cap valor de la variable fil no es convertirà en un NULL en la BD. Aquest problema és comú a tots els tipus bàsics.

- No permet deixar el text nul. Fins i tot fent que el text valgués "NULL", aquest NULL quedaria tancat entre cometes simples i, per tant, la BD l'interpretaria com un text que conté literalment NULL.

La solució al problema dels tipus bàsics passa o bé per treballar amb els seus objectes equivalents o bé per determinar un valor que representi el nul. Per exemple, el fil ha d'identificar un missatge i, per tant, serà un valor enter >0. Podríem reservar el –1 per identificar els valors nuls.

Per solucionar el segon problema hauríem de treure les cometes simples de la part estàtica de la sentència SQL, i afegir-la quan l'objecte fos necessari.

```
st.executeUpdate("INSERT INTO Missatges VALUES("+
  codiMissatge+","+codiForum+",'"+autor+"','"+
  titol+"',"+
  (text==null?"NULL":"'"+text+"'")+","+
  (fil==-1?"NULL":fil)+")");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *es012NullsStatement.java.*

### Ús de valors nuls en la clàusula WHERE

La sentència patró que es defineix en un PreparedStatment està oberta a la possibilitat de valors nuls, però només funcionarà si els nuls no estan en les condicions WHERE de la consulta SQL.

Els nuls que apareixen en les condicions WHERE tenen una sintaxi diferent, xxx IS NULL, en comptes de xxx=valor. Aquest canvi de sintaxi impedeix tenir un PreparedStatement que sigui vàlid per a condicions amb valors nuls i valors no nuls.

Per a la resta de casos, és a l'hora de definir els valors que haurem de tenir en compte els valors nuls. Quan els valors siguin objectes, podem continuar emprant el mètode setXXX, però en els tipus bàsics serà imprescindible la utilització del mètode setNull.

```
...
pst.clearParameters();
pst.setInt(1,codiMissatge);
pst.setInt(2,codiForum);
pst.setString(3,autor);
pst.setString(4,titol);
pst.setString(5,text);
if (fil==-1) pst.setNull(6,java.sql.Types.INTEGER);
else pst.setInt(6,fil);
pst.executeUpdate();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex013NullsPreparedStatement.*

#### Modificació d'un `ResultSet` amb valors nuls

En aquest darrer cas torna a passar el mateix. Tenim un mètode especial per a assignar nuls, que serà imprescindible per als tipus bàsics, i opcional, per a la resta.

```
rs.moveToInsertRow();
rs.updateInt(1,codiMissatge);
rs.updateInt(2,codiForum);
rs.updateString(3,autor);
rs.updateString(4,titol);
rs.updateString(5,text);
if (fil==-1) rs.updateNull(6);
else rs.updateInt(6,fil);
rs.insertRow();
rs.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex014NullsUpdatableResultSet*.

### 3.5.2. Combinacions de taules

Fins ara només hem vist exemples amb consultes simples basades en una sola taula; tanmateix, aquesta situació no és la més habitual. Una gran part de les consultes que una aplicació fa a una BD estan basades en combinacions (en anglès, *joins*) entre més d'una taula.

Amb combinacions tot continua funcionant tal com ja hem explicat, però hem de tenir present que:

- Possiblement els `ResultSets` modificables no funcionaran.

- En cas de noms de columnes coincidents en taules diferents, és interessant utilitzar àlies. Hi ha *drivers* que permeten solucionar el problema especificant `taula.columna`, però d'altres no. Per exemple, el *driver* de PostgreSQL no ho possibilita i, per tant, haurem de definir un àlies a les columnes repetides.

Podem trobar-ne un cas a la BD d'exemple amb les columnes nom de la taula `Fòrums` i `Usuaris`. Per a solucionar l'ambigüitat, cal fer la consulta amb àlies.

```
rs=st.executeQuery("SELECT Forums.nom as nom_forum, "+
  "Usuaris.nom as nom_usu,* "+
  "FROM Missatges,Forums,Usuaris "+
  "WHERE Missatges.codi_forum=Forums.codi_forum "+
  "AND Missatges.autor=Usuaris.nom_usuari");
while (rs.next())
  System.out.println(rs.getString("nom_forum")+
    "--"+rs.getInt("codi_missatge")+"--"+
    rs.getString("titol")+"--"+
    rs.getString("nom_usu"));
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex015ConsultaJoinAlias.java*.

### 3.5.3. Consultes niuades

Les consultes niuades són consultes que s'executen mentre recorrem les files d'una altra consulta.

En alguns casos, aquest tipus de consultes es poden evitar mitjançant combinacions, amb la qual cosa s'aconsegueix un rendiment més bo. En d'altres, la lògica de la consulta pot ser prou complicada per a no poder-se expressar en una única consulta SQL i fer imprescindible la utilització de consultes niuades.

L'únic detall que hem de tenir en compte és que un `Statement` només pot tenir un `ResultSet` actiu. Per tant, cada consulta niuada haurà de tenir un `Statement` independent.

**Exemple**

L'exemple següent mostra cada usuari i, per a cada usuari, els missatges corresponents. Per a il·lustrar la situació, sense fer servir un exemple excessivament complicat, s'ha resolt amb una consulta niuada, però també es podria solucionar amb una sola consulta que combinés les dades de les taules `Usuaris` i `Missatges`. En una situació real, però, hauríem de triar la solució amb combinacions, ja que té un rendiment més bo.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex016ConsultaNiuada Statement.java*.

```
st1 = conn.createStatement();
st2 = conn.createStatement();
rsUsuaris = st1.executeQuery("SELECT * FROM Usuaris");
while (rsUsuaris.next())
{
  String nomUsuari=rsUsuaris.getString("nom_usuari");
    rsMissatges=st2.executeQuery("SELECT * FROM "+
    "Missatges WHERE autor='"+nomUsuari+"'");
  while (rsMissatges.next())
  {
    System.out.println(
      rsUsuaris.getString("nom")+" "+
      rsUsuaris.getString("cognoms")+"--"+
      rsMissatges.getString("titol"));
  }
  rsMissatges.close();
}
rsUsuaris.close();
st1.close();
st2.close();
```

Tot i que l'exemple és correcte, les consultes niuades són una bona ocasió per a posar en pràctica els `PreparedStatement`. Efectivament, la consulta interior sempre té el mateix patró, i s'executa repetidament.

```
st = conn.createStatement();
pst = conn.prepareStatement("SELECT * FROM "+
      "Missatges WHERE autor=?");
rsUsuaris = st.executeQuery("SELECT * FROM Usuaris");
while (rsUsuaris.next())
{
  String nomUsuari=rsUsuaris.getString("nom_usuari");
  pst.setString(1,nomUsuari);
  ResultSet rsMissatges=pst.executeQuery();
  while (rsMissatges.next())
  {
    System.out.println(
      rsUsuaris.getString("nom")+" "+
      rsUsuaris.getString("cognoms")+"--"+
      rsMissatges.getString("titol"));
  }
  rsMissatges.close();
}

rsUsuaris.close();
st.close();
pst.close();
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex017ConsultaNiuada PreparedStatement.java*.

### 3.5.4. Consultes recursives

A conseqüència d'una interrelació recursiva entre taules, poden aparèixer consultes recursives. Seria un cas de consulta niuada o, més aviat, molt niuada.

En la BD d'exemple tenim una interrelació recursiva en la taula `Missatges`. De cada missatge en sabem el fil, és a dir, el missatge al qual respon. Gràcies a aquesta interrelació podem fer una llista dels missatges d'un fòrum per ordre de fil, no per ordre de redacció.

La solució més elegant seria utilitzar un únic `PreparedStatement` que es va executant d'una manera recursiva: en un primer nivell consultem els missatges que inicien fils (els que tenen fil nul); a continuació, i per cada missatge que inicia fil, consultem les seves respostes, i després les respostes de les respostes... Tanmateix, aquesta solució no és possible perquè:

- Un `PreparedStatement` no permet nuls a la condició `WHERE`, i la primera consulta demana els missatges el fil dels quals és nul.

- Tal com plantegem la consulta, hem de visitar les respostes del primer fil abans de finalitzar el recorregut dels fils inicials. Això significa que necessitem tants `ResultSet` simultanis com profunditat tingui la recursivitat. I recordem que un `Statement`, i per extensió un `PreparedStatement`, només permet un sol `ResultSet` simultani.

Resumint, la solució passa per crear un nou `Statement` a cada crida recursiva. En l'exemple considerem que `fil=-1` significa que el fil és nul. Hem afegit un paràmetre `tab` per tabular la sortida per pantalla.

```
void llistaRec(Connection conn, int codiForum,
    int fil, String tab) throws Exception
{
  Statement st = conn.createStatement();
  ResultSet rs=st.executeQuery("SELECT * FROM "+
    "Missatges WHERE fil"+
    (fil==-1?" IS NULL":"="+fil));
  while (rs.next())
  {
    System.out.println(tab+rs.getString("titol")+
      "--"+rs.getString("autor")+
      "--"+rs.getInt("codi_missatge"));
    llistaRec(conn,codiForum,
      rs.getInt("codi_missatge"), tab+" ");
  }
  rs.close();
  st.close();
}
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex018ConsultaRecursiva.java.*

### 3.5.5. Claus primàries autogenerades

D'una manera o d'una altra, els SGBD permeten la definició de columnes autogenerades. Són columnes el valor de les quals s'assigna automàticament cada cop que s'hi insereix una nova fila. Els valors autogenerats per aquestes columnes normalment són valors numèrics i correlatius.

Les columnes autogenerades s'utilitzen en les claus primàries, però tenen un problema a l'hora de codificar les aplicacions. En el moment en què s'insereix una nova fila l'aplicació envia les dades de les columnes, sense definir el valor de la clau primària. L'SGBD rep aquestes dades, calcula la nova clau primària i l'assigna.

Si l'aplicació no necessita saber el valor de la clau primària autogenerada no hi ha cap problema. Però, si no, com ho fa l'aplicació per saber quin ha estat el valor assignat?

JDBC ofereix un mecanisme que simplifica la feina. En el moment d'executar la sentència `INSERT` amb el mètode `executeUpdate` de la classe `Statement`, hi afegim un segon paràmetre `RETURN_GENERATED_KEYS` per indicar que ens retorni els valors de les columnes autogenerades (que normalment serà la clau primària). Rebem els valors autogenerats per mitjà d'un `ResultSet` que obtenim amb el mètode `getGeneratedKeys`, també de la classe `Statement`.

**Exemple**

En l'exemple següent creem una taula `Grups` amb una clau primària entera autogenerada. És important tenir present que la sintaxi per a definir columnes autogenerades depèn de l'SGBD. En l'exemple hem utilitzat la sintaxi de PostgreSQL. Cal destacar, també, que de valors autogenerats, només n'hi haurà un i que, per això, podem recórrer el `ResultSet` sense iterar.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex019Autoincrement.java*.

```
st.executeUpdate("CREATE TABLE Grups " +
  "(codi_grup SERIAL PRIMARY KEY, nom VARCHAR(20))");
String nom=...
st.executeUpdate("INSERT INTO Grups ('"+nom+"')",
    Statement.RETURN_GENERATED_KEYS);
ResultSet rs=st.getGeneratedKeys();
if (rs.next()) System.out.println(rs.getInt(1));
rs.close();
```

Malauradament, el *driver* del PostgreSQL no suporta aquesta funcionalitat i, per tant, l'exemple anterior generaria un error en temps d'execució. Per aconseguir el mateix efecte, PostgreSQL proposa una extensió de l'SQL estàndard afegint la clàusula `RETURNING` al final de la sentència `INSERT`.

**Exemple**

Aquest exemple mostra el codi necessari per a obtenir el valor de la clau primària autogenarada de la taula `Grups` en el cas de PostgreSQL.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex020AutoIncrementOK.java*.

```
ResultSet rs=st.executeQuery("INSERT INTO Grups " +
  "(nom) VALUES ('"+nom+"') RETURNING codiGrup");
if (rs.next()) System.out.println(rs.getInt(1));
else System.out.println("ERROR");
rs.close();
```

### 3.5.6. SQL injection

Les instruccions SQL enviades a la BD per mitjà de `Statement` poden tenir un problema de seguretat conegut per SQL *injection*. A continuació us en proposem un exemple per entendre la base del problema.

És molt habitual que les aplicacions identifiquin els usuaris abans de començar a treballar. Gràcies a això, poden oferir més o menys operacions als usuaris, segons el rol que tenen.

Un sistema d'identificació habitual és demanar el nom d'usuari i la contrasenya. Un cop l'hem entrat, el sistema valida que l'usuari i la contrasenya són correctes. I, per a això, el més raonable és fer una consulta a la BD que contingui la taula amb les dades dels usuaris (com la de la nostra BD de referència).

Hi ha moltes maneres de fer aquesta validació; per exemple, fent una consulta com aquesta:

```
String nomUsuari,contrasenya;
//llegim nomUsuari i contrasenya
...
ResultSet rs = st.executeQuery("SELECT * "+
  "FROM Usuaris WHERE nom_usuari='"+nomUsuari+
  "' AND contrasenya='"+contrasenya+"'");
if (rs.next()) System.out.println("CORRECTE");
else System.out.println("ERROR");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex021SQLInjection.java.*

**Seguretat de les contrasenyes**

L'exemple de la contrasenya exhibeix una mala pràctica de seguretat molt habitual: emmagatzemar contrasenyes planes. És preferible emmagatzemar la contrasenya a la BD fent servir alguna tècnica d'encriptació de dades.

Com qui no vol la cosa, amb un codi com aquest tenim un forat de seguretat important. Fixem-nos en el que passa si un usuari introdueix el nom següent i qualsevol contrasenya:

```
' OR 1=1 OR 'A'='
```

Això no és el nom de cap usuari i, malgrat tot, el nostre sistema dóna com a resultat `CORRECTE`. El problema rau en la construcció de la sentència SQL concatenant parts estàtiques amb el contingut de variables. Si alguna variable, en comptes de contenir un valor, té el codi SQL adequat, permet modificar la sentència SQL que s'està executant!

Aquest problema té solució, tant si es reforça el codi com si s'utilitza `PreparedStatements`:

- Les solucions en codi es basen a escapar els apòstrofs, és a dir, marcar els apòstrofs dels valors de tipus text perquè l'SGBD no els confongui amb els apòstrofs de la sentència SQL. Cada SGBD ofereix alguna alternativa per escapar apòstrofs, com, per exemple, `\'` o `''`.

- Les sentències executades amb `PreparedStatement` i els `ResultSet` modificables estan protegits de SQL *Injection*, ja que el contingut dels valors no interfereix en la sentència que s'està executant.

Per exemple, PostgreSQL escapa els apòstrofs duplicant-los tots. L'exemple anterior es podria corregir utilitzant el mètode `replace` de classe `String`.

```
//Llegim nomUsuari i contrasenya
...

//Escapem els apòstrofs
contrasenya=contrasenya.replace("'","''");

//Consultem si el nomUsuari i la contrasenya són vàlids
ResultSet rs = st.executeQuery("SELECT * FROM "+
  "Usuaris WHERE nom_usuari='"+nomUsuari+"' AND "+
  "contrasenya='"+contrasenya+"'");
if (rs.next()) System.out.println("CORRECTE");
else System.out.println("ERROR");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex022EscapaApostrof.java.*

### 3.6. Procediments emmagatzemats

JDBC ofereix l'objecte `CallableStatement` per poder executar procediments emmagatzemats. La sintaxi utilitzada s'aparta de l'SQL i s'anomena *escape syntax*. A més a més, tindrem mètodes per a definir els paràmetres d'entrada i recollir els de sortida. La forma general d'una crida serà:

```
{? = call nomProcedimentEmmagatzemat(?, ?, ...)}
```

en què els `?` corresponen als paràmetres d'entrada, de sortida, i d'entrada i sortida.

**Exemple**

El següent és un exemple d'un procediment emmagatzemat que rep un paràmetre d'entrada (import) i retorna l'import amb IVA.

```
CREATE FUNCTION ambIva(import FLOAT) RETURNS FLOAT AS $$
BEGIN
  RETURN import * 1.18;
END;
$$ LANGUAGE plpgsql;
```

La creació del `CallableStatement` per a aquest procediment emmagatzemat seria:

```
CallableStatement cst =
    conn.prepareCall("{?=call ambIva(?)}");
```

Els procediments emmagatzemats també es poden crear des de JDBC, mitjançant un `Statement`, però és una pràctica poc habitual. La diferència principal és que tota la sentència quedarà definida en una sola línia de text.

```
st.executeUpdate("CREATE OR REPLACE FUNCTION "+
  "ambIva(import FLOAT) RETURNS FLOAT AS $$ "+
  "BEGIN RETURN import * 1.18; END; "+
  "$$ LANGUAGE plpgsql;");
```

#### 3.6.1. Paràmetres d'entrada

Per a passar paràmetres d'entrada utilitzarem mètodes `setXXX`, un cop creat el `CallableStatement`. El funcionament d'aquests mètodes és idèntic als mètodes `setXXX` dels `ResultSet` modificables.

#### 3.6.2. Paràmetres de sortida

Els paràmetres de sortida s'han de registrar abans de fer la crida al procediment emmagatzemat. Durant el registre indiquem el tipus de paràmetre que rebrem. Després d'executar el procediment emmagatzemat es poden recollir els resultats amb els mètodes `getXXX`.

```
CallableStatement cst =
  conn.prepareCall("{?=call ambIva(?)}");
cst.registerOutParameter(1, java.sql.Types.REAL);
cst.setFloat(2,10.0f);
cst.execute();
System.out.println(cst.getFloat(1));
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex023StoredProcedureIN_OUT*.

### 3.6.3. Paràmetres d'entrada i sortida

Per aconseguir paràmetres d'entrada i sortida farem una combinació dels dos anteriors. Utilitzarem el mètode `setXXX` per a donar el valor d'entrada i, abans d'executar el procediment emmagatzemat, també el registrarem. Un cop acabat, obtindrem els paràmetres de sortida amb `getXXX`.

```
st.executeUpdate("CREATE OR REPLACE FUNCTION "+
  "ambIva2(INOUT import FLOAT) AS $$ "+
  "BEGIN import:= import*1.18; END; "+
  "$$ LANGUAGE plpgsql;");

cst = conn.prepareCall("{call ambIva2(?)}");
cst.registerOutParameter(1, java.sql.Types.REAL);
cst.setFloat(1,10.0f);
cst.execute();
System.out.println(cst.getFloat(1));
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex024StoredProcedureINOUT*.

## 3.7. Gestió d'errors

Totes les aplicacions estan exposades a errors durant l'execució, però, si a més, estan connectades a una BD, les probabilitats augmenten. Podem trobar-ne la raó pel fet de cooperar amb un sistema complex (l'SGBD), al qual ens connectem per xarxa i que, simultàniament, dóna servei a altres aplicacions. Per si no fos prou, les sentències que enviem sovint són dinàmiques i, per tant, susceptibles d'errors només detectables en temps d'execució.

### 3.7.1. Tipus d'error i com es reporten

JDBC exposa els errors mitjançant excepcions facilitant d'una manera considerable el desenvolupament d'aplicacions. Com és habitual en Java, es defineix una jerarquia d'excepcions que deriven d'`Exception`. Al capdamunt d'aquesta jerarquia hi ha la classe `SQLException`.

Una gran part de les excepcions que genera JDBC són o deriven de `SQLException`, tot i que podrem explotar poc la jerarquia. Les excepcions que ens interessarà capturar són sobretot `SQLException` i, puntualment, alguna excepció derivada. Així doncs, rarament podrem tractar les excepcions a nivell de blocs `try-catch` i ho haurem de fer a partir de la informació que ens facilitaran els objectes `SQLException`.

### 3.7.2. Obtenció d'informació sobre una `SQLException`

Els objectes `SQLException` exposen el tipus d'error generat mitjançant tres mètodes:

- `getMessage()`: retorna una cadena amb informació textual de l'error. És el mètode estàndard heretat de la classe `Exception`.
- `getSQLState()`: retorna una cadena de cinc caràcters amb un codi d'error definit a XOPEN SQLState o a SQL:2003. Haurem d'investigar quin estàndard segueix l'SGBD que utilitzem i els codis dels errors que ens interessa detectar.
- `getErrorCode()`: retorna un enter que correspon al codi d'error. El significat d'aquest enter dependrà de l'SGBD al qual estem connectats.

```
int codiMissatge;
String nomUsuari;
...
try
{
  st.executeUpdate("INSERT INTO Lectures "+
    "VALUES("+codiMissatge+",'"+nomUsuari+"',"+
    "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");
}
catch (SQLException e)
{
  System.out.println("Missatge:"+e.getMessage());
  System.out.println("SQLState:"+e.getSQLState());
  System.out.println("ErrorCode:"+e.getErrorCode());
}
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex025SQLExceptionInfo.java*.

En aquest exemple, la sentència `INSERT` pot generar tres tipus d'error. Si l'SGBD utilitzat és el PostgreSQL, els resultats que podríem obtenir són:

**1)** La clau forana `codi_missatge` no identifica cap missatge.

```
ERROR: insert or update on table "lectures" violates foreign key
constraint "lectures_missatges_fkey"
  Detail: Key (codi_missatge)=(455) is not present in table
  "missatges".

SQLState:23503
ErrorCode:0
```

**2)** La clau forana `nom_usuari` no identifica cap usuari.

```
ERROR: insert or update on table "lectures" violates foreign key
constraint "lectures_usuaris_fkey"
  Detail: Key (nom_usuari)=(sadf) is not present in table "usuaris".

SQLState:23503
ErrorCode:0
```

**3)** Aquest usuari ja havia llegit el missatge: clau primària duplicada.

```
ERROR: duplicate key value violates unique constraint "lectures_pkey"

SQLState:23505
ErrorCode:0
```

**Enllaç d'interès**

Podeu trobar la llista de codis SQLState de PostgreSQL en l'annex A de la documentació: http://www.postgresql.org/docs/8.1/interactive/errcodes-appendix.html.

Podem comprovar que el *driver* de PostgreSQL no retorna `ErrorCode`, només `SQLState`. Això no presenta cap problema, ja que des de l'estàndard SQL:1992 es va proposar reportar els errors per mitjà de codis `SQLState`. Durant uns quants anys s'han mantingut les dues versions de codis, però la versió actual del *driver* JDBC de PostgreSQL ha prescindit de l'`ErrorCode`.

### 3.7.3. Gestió de l'error des dels nostres programes

Analitzant els `SQLState` obtinguts en els tres exemples anteriors podem distingir entre **errors provocats per claus foranes** o **errors provocats per claus primàries**. Tanmateix, per a discriminar quina de les claus foranes ha provocat l'error no ens queda més remei que analitzar el text obtingut a `getMessage`.

Ara ja podem definir el patró que seguirem per al tractament d'errors:

**1)** Definir el bloc `try-catch` del conjunt de sentències SQL de les quals volem tractar els errors. Com més gran sigui el conjunt, més difícil ens serà determinar-ne l'error exacte. Si amb un sol bloc no podem tractar bé els errors, n'haurem de definir més d'un, sovint, niuats.

**2)** Capturar els diferents tipus d'errors a partir de la classe d'excepció generada. Això ens permetrà diferenciar entre errors de *drivers*, connexió, etc., però no entre el tipus d'error SQL.

**3)** Per discriminar el tipus d'error SQL ens basarem en el codi `SQLState`.

**4)** Per acabar de determinar la causa exacta de l'error haurem d'analitzar la cadena obtinguda amb el mètode `getMessage` buscant alguna paraula clau que ens ajudi.

Un cop detectada la causa exacta de l'error, l'aplicació haurà de decidir entre reintentar l'operació, abandonar-la silenciosament (és a dir, ignorar-la) o aturar-la amb una excepció. Exceptuant el reintent, en les altres situacions és molt important tancar tots els objectes `ResultSet`, `Statement`, etc. que puguem tenir oberts. Així permetem que l'SGBD alliberi recursos ràpidament i que pugui atendre altres usuaris.

**Exemple per registrar la lectura d'un missatge per part d'un usuari**

```
Class.forName( "org.postgresql.Driver" );
String dbURL="jdbc:postgresql:bdMail";
Connection conn = DriverManager.getConnection(
      dbURL, "usuari","usuari");
Statement st = conn.createStatement();
st.executeUpdate("INSERT INTO Lectures VALUES("+
  codiMissatge+",'"+nomUsuari+"',"+
  "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex026TractamentException.java*.

Les diverses línies d'aquest codi poden generar tipus d'excepcions diferents:

- `Class.forName(...)`:
  - `ClassNotFoundException`: no trobem el *driver*. No està instal·lat o no hem definit el `CLASSPATH`.

- `DriverManager.getConnection(...)`:
  - `SQLException`: hi ha un error connectant amb la BD: l'url, el port, l'usuari, la contrasenya, etc. són incorrectes.

- `Connection.createStatement()`
  - `SQLException`: la connexió està tancada, o els paràmetres per a definir el tipus de `ResultSet` són incorrectes.

  - `SQLFeatureNotSupportedException`: el *driver* no suporta aquesta operació, normalment deguda al tipus de `ResultSet` demanat.

- `Statement.executeUpdate(...)`
  - `SQLException`: error en la instrucció SQL. Serà la situació més habitual.

  - `SQLFeatureNotSupportedException`: en cas que el *driver* no suporti alguna funcionalitat. Per exemple, en el cas de PostgreSQL, la impossibilitat de retornar les columnes autogenerades.

Si ens interessa tractar totes les excepcions d'una manera particular, no en farem prou amb un sol bloc `try-catch`. Ara bé, exceptuant els errors provocats per `executeUpdate`, la resta normalment no s'haurien de produir. Són errors del sistema més que de l'aplicació i, si es produeixen, poca cosa podrem fer per solucionar-los.

Una opció correcta és no fer res d'especial en els errors de sistema. L'exemple anterior, tal com està, seria insuficient, ja que, en cas d'error, no es tancarien els objectes oberts. En l'exemple següent mostrem una possible estructuració del codi per garantir que, tant en cas d'èxit com d'error, tots els objectes oberts quedaran tancats. La utilització del bloc `finally` simplifica la codificació. Així

mateix, dins del bloc `finally`, tots els mètodes estan protegits amb un bloc `try-catch` per garantir que, tot i que es produeixi un error, el bloc `finally` continua executant-se fins al final.

```
try
{
  Class.forName( "org.postgresql.Driver" );
  String dbURL="jdbc:postgresql:bdMail";
  conn = DriverManager.getConnection( dbURL,
          "usuari","usuari");
  st = conn.createStatement();
  //...
}
catch (Exception e)
{
  throw e;
}
finally
{
  //tanquem el que calgui
  if (st!=null)
  {
    try st.close();
    catch(SQLException e)
    {
      //...tractem l'error
    }
  }
  if (conn!=null)
  {
    try conn.close();
    catch(SQLException e)
    {
    //...tractem l'error
    }
  }
}
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex026TractamentException.java.*

Els errors provocats per l'execució de sentències SQL sí que els tractarem. Si el problema és provocat per dades errònies que l'usuari ha entrat, és molt interessant donar un missatge d'error fàcil d'entendre per l'usuari final. I, sobretot, donar una segona oportunitat.

**Exemple**

En aquest exemple llegim el codi d'un missatge, el nom d'un usuari i afegim una fila a la taula `Lectures` per reflectir que l'usuari en qüestió ha llegit el missatge en la data i l'hora actuals.

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex026TractamentException.java.*

```
boolean correcte=false;
while (!correcte)
{
try
  {
    //llegim les dades
    System.out.print("CodiMissatge:");
    codiMissatge=Integer.parseInt(cons.readLine());
    System.out.print("NomUsuari:");
    nomUsuari=cons.readLine();
    ts = new Timestamp(System.currentTimeMillis());

    st.executeUpdate("INSERT INTO Lectures "+
       "VALUES("+codiMissatge+",'"+nomUsuari+"',"+
       "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");

    correcte=true;
  }
  catch (SQLException e)
  {
    if(e.getSQLState().equals(FK_ERROR))
      if(e.getMessage().indexOf(FK_MISSATGES)!=-1)
        out.println("codi de Missatge no existeix");
      else if(e.getMessage().indexOf(FK_USUARIS)!=-1)
        out.println("nom de l'usuari no existeix");
      else out.println("Inesperat:"+e.getMessage());
    else if (e.getSQLState().equals(PK_ERROR))
      out.println("Error: missatge ja llegit");
    else out.println("Inesperat:"+e.getMessage());
  }
  catch (Exception e)
  {
    out.println("Error llegint les dades");
  }
}
```

## 3.8. Gestió de transaccions

Cada connexió JDBC pot mantenir una sola transacció activa. Quan creem una connexió, per defecte, es treballa en mode *autocommit*. Això significa que cada sentència SQL que executem constitueix una transacció que s'inicia abans de dur a terme la sentència i que, si no es genera cap error, acaba en *commit* en finalitzar (per això, el nom *autocommit*).

Treballar en mode *autocommit* és suficient quan cada transacció només incorpora una sentència SQL o quan sabem que no poden aparèixer problemes de concurrència (per exemple, quan estem segurs que només nosaltres estem treballant amb la BD o quan sabem que totes les operacions que s'executen sobre la BD són operacions de només lectura). Quan hàgim de treballar amb transaccions que incorporen més d'una sentència SQL, o quan tinguem dubtes en relació amb les operacions que s'executen d'una manera concurrent amb les operacions que duen a terme les nostres aplicacions, és recomanable treballar amb el mode *autocommit* desactivat.

Amb el mode *autocommit* desactivat, les transaccions s'acaben explícitament cridant els mètodes `commit()` o `rollback()` i comencen implícitament en executar la primera sentència després d'un *commit* o un *rollback*. Això permet incloure diferents sentències SQL dins d'una transacció, tirar-les totes endarrere amb un *rollback* o acceptar-les totes amb un *commit*.

```
conn.setAutoCommit(false);
...
try
{
  ... sentències SQL que formen la transacció
  conn.commit();
}
catch (SQLException e)
{
  conn.rollback();
}
```

En la nostra BD de referència tenim una taula per a registrar els missatges que han llegit els usuaris. Cada cop que un usuari llegeix un missatge haurem d'actualitzar la taula `Lectures` amb un codi semblant a aquest.

```
st.executeUpdate("INSERT INTO Lectures "+
  "VALUES("+codiMissatge+",'"+nomUsuari+"',"+
  "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");
rs=st.executeQuery("SELECT * FROM Missatges "+
  "WHERE codi_missatge="+codiMissatge);

if (rs.next())
  System.out.println(rs.getString("titol")+
    "--"+rs.getString("text"));
```

Aquest codi sembla correcte i, de fet, ho és sempre que no es produeixi cap error. Ara bé, si mai es produeix un error després d'haver inserit la fila i abans de llegir el missatge, deixarem la BD en un estat incorrecte. Per recuperar l'estat hauríem de tractar l'error i fer un `DELETE` de la fila, però aquest `DELETE` també podria fallar i aleshores es complicaria la recuperació de l'estat.

La solució òptima per a aquest escenari és la utilització de transaccions. En l'exemple següent definim una transacció que comprèn les operacions de registrar la lectura del missatge i llegir el missatge en si. Si aconseguim registrar la lectura, llegir el missatge i mostrar-lo per pantalla, finalitzem la transacció amb *commit*. En cas contrari, desfem la transacció i tornem a l'estat inicial.

```
conn.setAutoCommit(false);
...
try
{
  st.executeUpdate("INSERT INTO Lectures "+
    "VALUES("+codiMissatge+",'"+nomUsuari+"',"+
    "to_timestamp('"+ts+"','YYYY-MM-DD HH24:MI'))");
  rs=st.executeQuery("SELECT * FROM Missatges "+
      "WHERE codi_missatge="+codiMissatge);
  if (rs.next()) System.out.println(
    rs.getString("titol")+ "--"+rs.getString("text"));
  else throw new SQLException();
  rs.close();
  conn.commit();
}
catch (SQLException e)
{
  conn.rollback();
}
```

**Nota**

Teniu disponible el codi de l'exemple al fitxer *ex027MissatgesLlegits Transaccio.java*.

### 3.8.1. Durada d'una transacció

Quan treballem amb el mode *autocommit* desactivat, cal tenir present que la durada d'una transacció convé que sigui tan curta com sigui possible. O, més ben dit, interessa que no sigui més llarga del que és estrictament necessari.

Les sentències SQL que s'executen durant una transacció poden afectar diverses files de taules de la BD diferents. Depenent de la gestió interna que fa l'SGBD, aquestes files o taules poden quedar bloquejades. No és fins al final de la transacció que l'SGBD allibera les files o taules del bloqueig. Mentrestant, sentències SQL que s'estiguin executant concurrentment i que afectin alguna fila o taula bloquejades poden quedar en pausa i, fins que no s'acaba la transacció, no continuen l'execució.

És a dir, com més llarga sigui l'execució d'una transacció, més probabilitats hi ha d'aturar temporalment l'execució d'altres aplicacions que accedeixin simultàniament a la BD. Per això és tan important que les transaccions durin estrictament el temps necessari.

# Resum

En aquest mòdul didàctic hem presentat les tecnologies principals de SQL programat que incorporen la majoria dels SGBD relacionals del mercat:

- L'SQL hostatjat, que permet introduir sentències SQL enmig del codi i requereix una precompilació abans de ser compilat.
- L'SQL/CLI, interfície estàndard que permet accedir a la BD a partir de funcions.
- ODBC, evolució de l'SQL/CLI que permet carregar el *driver* en temps d'execució.
- OLE DB: evolució d'ODBC que permet accedir a altres orígens de dades, no solament a SGBD.
- DAO, RDO, ADO, ADO.NET, evolució d'OLE DB orientada a objectes. Per a sistemes operatius Microsoft.
- JDBC: evolució d'OLE DB orientada a objectes i oberta a qualsevol plataforma en l'entorn de programació Java.

Ens hem centrat en JDBC i n'hem vist les diferents versions i els diversos tipus de *drivers* (implementacions de l'API JDBC dels diferents SGBD). També hem estudiat l'estructura bàsica d'una aplicació: connexió, execució de sentències SQL, tractament dels resultats i desconnexió.

Així mateix, hem presentat possibles alternatives tant per a l'execució de sentències SQL com per al tractament dels resultats. Hem vist la possibilitat de fer recorreguts endarrere o aleatoris, d'actualitzar la BD sense sentències SQL, de definir sentències SQL parametritzades, d'executar procediments emmagatzemats, de gestionar els errors i de treballar amb transaccions.

Addicionalment, hem plantejat una solució a situacions habituals com són els valors nuls, les combinacions, les claus autogenerades, les consultes niuades, les consultes recursives i la prevenció d'atacs mitjançant SQL *injection*.

# Activitats

**1.** Us proposem un seguit d'activitats basades en els exemples Java del mòdul. Veureu que s'ha utilitzat l'SGBD PostgreSQL en tots els exemples i recomanem fer servir els *drivers* de tipus 4 que podeu trobar a http://jdbc.postgresql.org/download.html.

Abans de començar, recordeu-vos de crear una BD en el PostgreSQL i un usuari amb privilegis suficients. Els exemples són fets amb una BD anomenada *bdMail* i un usuari amb nom "usuari" i contrasenya "1234".

# Glossari

**API** *f* (*aplication program interface*) Interfície d'aplicació.

**BD** *f* Sigla corresponent a *base de dades*.

**cursor** *m* Estructura de SQL que permet que una aplicació sigui capaç de recórrer les files del resultat d'una consulta de SQL.

**JDBC** *f* Marca comercial de Sun Microsystems. Defineix un API estàndard per a manipulació de SQL des de Java. S'accepta com l'acrònim de *Java database connectivity*.

**JDK** *m* (*Java Development Kit*). Intèrpret i entorn per a desenvolupaments Java realitzat per *Sun Microsystems*.

**programa *driver*** *m* Programa d'implementació particular del JDBC.

**SAG-X/Open** *m* *Access group X/Open*.

**SQL** *m* *Structured query language*.

**SQL/CLI** *m* Tècnica de SQL programat que segueix un enfocament interpretat i que permet a l'aplicació l'accés a les BD gestionades per l'SGBD mitjançant crides a diferents subrutines que són disponibles en biblioteques.

**SQL hostatjat** *m* Tècnica de SQL programat que segueix un enfocament compilat i que permet incorporar directament les sentències del llenguatge SQL (denominades *sentències de SQL hostatjat*) dins de l'aplicació, barrejades amb les sentències pròpies del llenguatge de programació (anomenat *llenguatge amfitrió*).

**variable pont amb SQL** *f* Mecanisme que ofereix l'SQL hostatjat per a poder referenciar les variables de l'aplicació en les sentències de SQL. Serveix tant per a passar dades a una sentència de SQL hostatjat com per a recollir les dades que són resultat d'una sentència de SQL hostatjat.

# Bibliografia

**Date, C. J.; Darwen, H.** (1996). *A Guide to the SQL Standard* (4a. ed.). Massachusetts: Addison-Wesley.

**ECPG-Embedded SQL in C** (2010). [Documentació en línia:] http://developer.postgresql.org/pgdocs/postgres/ecpg.html

**JDBC driver de PostgreSQL** (2010). [Documentació en línia:] http://jdbc.postgresql.org

**PostgreSQL** (2009). [Documentació en línia:] http://www.postgresql.org/docs

**Sun Microsystems, Inc.** (2010). *JDBC*. [Documentació en línia:] http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/index.html