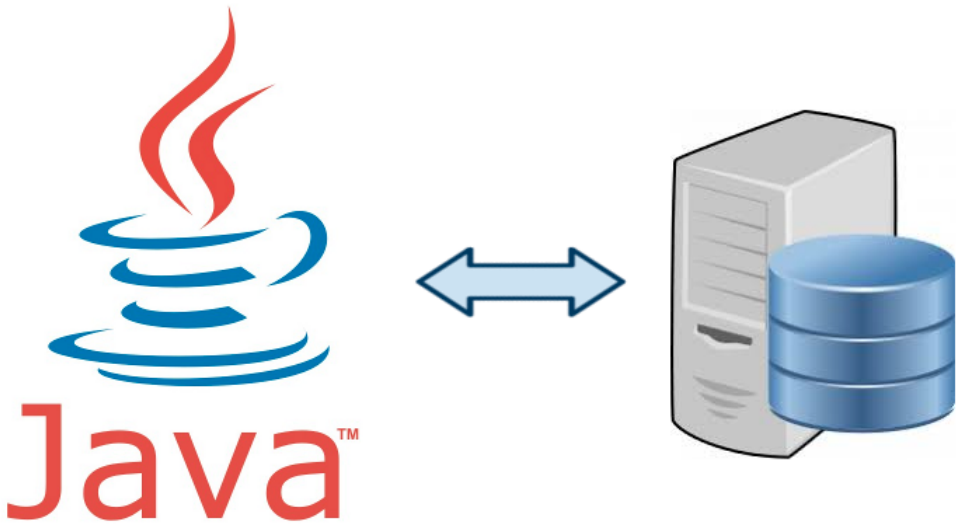
Accés a Dades

Tema 4: Bases de Dades Relacionals



1 - Les Bases de Dades Relacionals

Els Sistemes Gestors de Bases de Dades Relacionals es basen en el Model Relacional (com el seu nom indica), és a dir en taules interrelacionades entre elles, i han demostrat la seua solidesa per a guardar la informació al llarg dels anys. Recordem que el primer producte comercial seriós d'aquest tipus, Oracle, va aparéixer en 1979. Alguna cosa tindrà, per tant, per a haver aguantat tants anys. Segurament és perquè es tracta d'una tecnologia senzilla però molt eficient. I que a més ha sabut adaptar-se a la major part de sistemes de dades que les empreses reclamaven i a un cost prou assequible.

Però no tot són avantatges en el Model Relacional. Té limitacions importants, per exemple a l'hora de representar informació poc estructurada, o per a estructures excessivament dinàmiques i complexes. Fins i tot semblava que anava a substituir-se aquest model per altres, com el Model Orientat a Objectes. Però aquesta substitució o canvi no acaba d'arribar. La principal raó la trobem en la solidesa i maduresa que tenen els Sistemes Gestors de Bases de Dades Relacionals.

De fet, molts autors apunten cap a una evolució dels Sistemes Relacionals incorporant eines i tecnologies que els apropen al model Orientat a Objectes (com veurem en el Tema 6) més que no cap a la seua desaparició i substitució.

Per tant, s'imposa el seu estudi.

2 - El desfasament Objecte-Relacional

Quan necessitem explicar o plasmar una realitat complexa, en compte d'intentar guardar-la directament, és molt convenient utilitzar un model conceptual més proper a nosaltres que es comporte de forma similar a la realitat. Es tracta de plasmar els aspectes essencials i, a la vegada, alleugerir els detalls insignificants per tal de poder rebaixar la complexitat, i representar-lo d'una manera propera a nosaltres.

La utilització de models conceptuals durant la implementació d'aplicacions informàtiques és d'una importància extrema per poder portar a bon termini qualsevol projecte d'informatització.

El problema és que els models conceptuals són representacions mentals creats a base d'un procés d'abstracció. I no hi ha una única forma de plasmar-los o representar-los. Moltes vegades fem servir aproximacions esquemàtiques que poden estar molt prop de la representació mental, però fins i tot les representacions esquemàtiques són difícilment representables en la memòria d'un ordinador.

- En el cas dels Sistemes Gestors de Bases de Dades Relacionals, primer intentem representar-lo per mig del **Model Entitat-Relació** (que seria el model conceptual que ens permet fer l'aproximació esquemàtica), i després traduim aquest al Model Relacional (a les taules).
- Com veurem després, el **Model Orientat a Objectes** intenta representar la realitat per mig d'objectes i les interactuacions que poden haver entre ells. Per tant és un altre model conceptual per a intentar representar la mateixa realitat.

Ens trobem per tant davant de dues maneres de representar la informació, i inevitablement hi haurà un desfasament. Perquè en les nostres aplicacions utilitzarem objectes, mentre que en el Model Relacional (on volem guardar la informació) s'utilitzen taules.

Intentarem explicar aquest desfasament amb uns exemples.

El Model Relacional

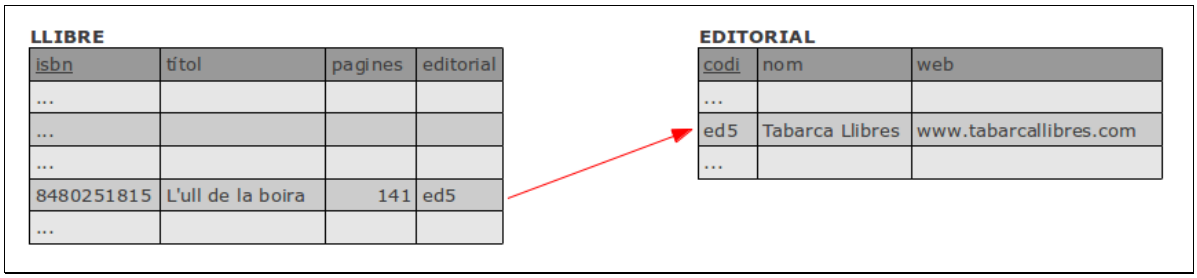
El Model Relacional es basa en les taules. En una taula tindrem en les columnes els distints atributs o característiques que ens volem representar, i en les files els distints individus d'aquesta taula.

EDITORIAL

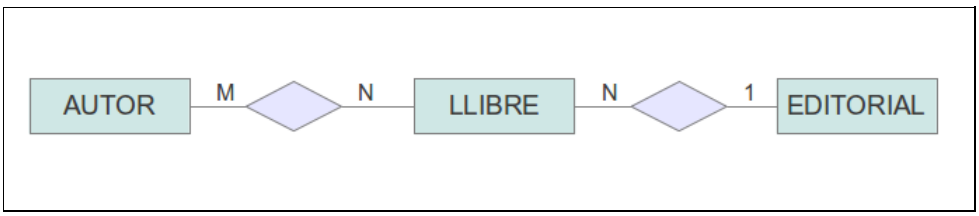
codi	nom	web
...		
ed5	Tabarca Llibres	www.tabarcallibres.com
...		

En totes les taules considerem que tenim una clau principal, en l'exemple **codi**, que identifica unívocament l'individu, en aquest cas l'editorial.

Les diferents taules poden estar relacionades. Si per exemple ens guardem també els llibres, veuríem que les editorials i els llibres estan relacionades, concretament un llibre l'ha editat una editorial. Per a marcar aquest fet el Model Relacional utilitza les **claus externes** (***foreign keys***). Una clau externa és un camp que en una altra taula és clau principal (la que identifica unívocament). D'aquesta manera, en la taula **LLIBRE** posarem un camp (per exemple **editorial**) el contingut del qual serà el codi, qui identifica en l'altra taula, i d'aquesta manera representarem perfectament l'editorial a la qual pertany el llibre.

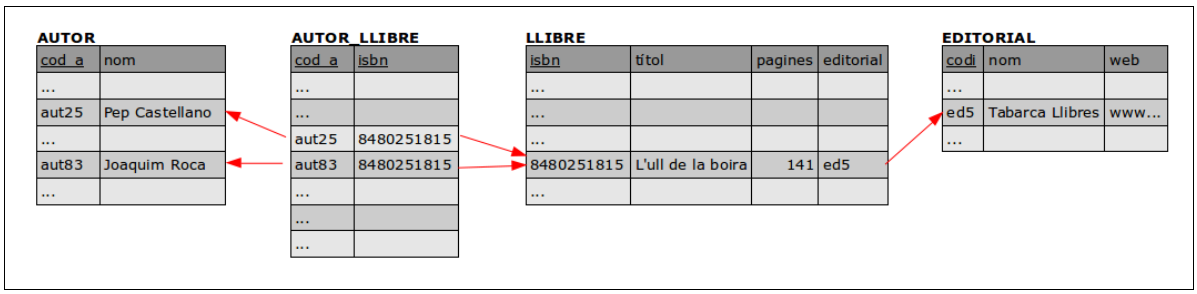


Completem un poc més l'exemple anterior, registrant també els autors dels llibres. Recordem que un llibre pot tenir més d'un autor i un autor haver fet més d'un llibre. Com comentàvem anteriorment, el primer que s'hauria de fer per tenir un bon disseny és l'esquema en el **Model Entitat-Relació**, i després traduir-lo al Model Relacional.

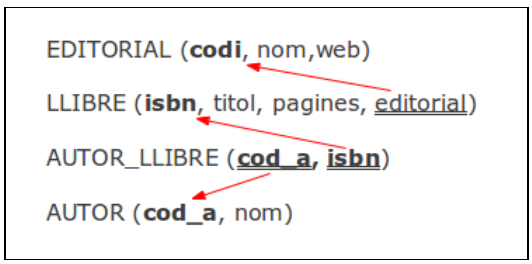


Que vol dir que una editorial pot tenir molts llibres editats (però un llibre només està editat per una editorial). I un llibre pot ser escrit per més d'un autor, el qual a la seua vegada pot escriure molts llibres.

La traducció al Model Relacional ens donaria no 3 taules, sinó 4. A més de les taules **EDITORIAL**, **LLIBRE** i **AUTOR**, ens fa falta una altra taula, anomenada per exemple **AUTOR_LLIBRE**, que és el resultat de la relació M:N entre **AUTOR** i **LLIBRE**.



Aquest seria l'esquema del Model Relacional:



on hem indicat les claus principals en negreta, i les claus externes amb un doble subratllat i una fletxa apuntant a la clau principal de l'altra taula. Observeu que la clau principal de **AUTOR_LLIBRE** és **cod_a + isbn**. A més cadascuna d'elles és també clau externa que apunta a la taula corresponent.

El Model Relacional també permet definir un conjunt de regles i limitacions en els valors de les dades i en les accions a realitzar, amb l'objectiu d'assegurar la consistència de les dades. Així, és possible indicar què s'ha de fer amb els registres d'una taula que es troben vinculats al registre d'una segona taula en el moment d'eliminar-lo d'aquesta segona taula. Per exemple, si eliminem l'autor **aut83** de la taula d'autors, què fem amb les files de la taula AUTOR_LLIBRE que tenen aquest autor? Doncs tres serien les possibilitats:

- **NO ACTION** , és a dir, no fer l'acció (no esborrar aut83)
- **CASCADE** , és a dir, esborrar també les files de AUTOR_LLIBRE en les que estiga aut83.

- **SET NULL** , és a dir posar a nul el camp en la taula vinculada, però en el nostre exemple aquest ultim cas no és possible, perquè **cod_a** forma part de la clau principal, i per definició cap camp de la clau principal pot agafar el valor nul.

En el cas d'esborrar un autor, a priori sembla que l'opció més correcta seria la primera, és a dir, no poder esborrar un autor del qual tenim algun llibre (i per tant tenim alguna fila en la taula AUTOR_LLIBRE amb aquest autor). Però en cada cas s'ha de mirar quina és l'opció més adequada. Així per exemple, segurament si esborrem un llibre, podríem perfectament esborrar automàticament de la taula AUTOR-LLIBRE les files d'aquest llibre (però no esborrar de la taula AUTOR, clar).

El Model Relacional també permet altres restriccions, com per exemple:

- definir el rang o conjunt de valors possibles que un camp d'una taula podrà agafar (el que s'anomena com a **domini**),
- definir un camp com a no nul (per exemple podríem exigir que el camp **nom** de la taula **AUTOR** no puga agafar el valor nul)
- assegurar la no repetició de determinats camps en diferents registres d'una mateixa taula (per exemple que no es puga repetir el camp nom de la taula EDITORIAL, ja que suposaria tenir dues editorials que es diuen exactament igual).

Model Orientat a Objectes

El **Model Orientat a Objectes** és un altre model conceptual que té un punt de vista diferent al Model Entitat-Relació.

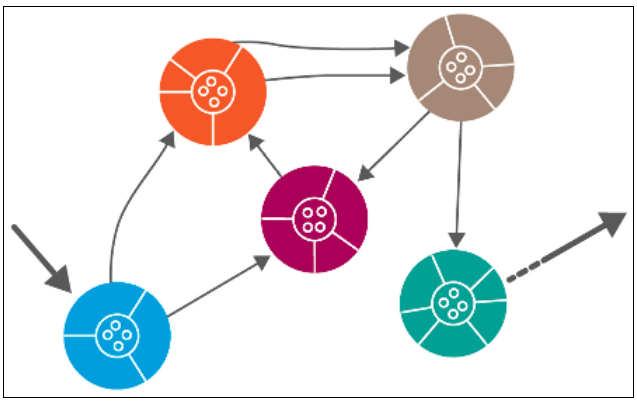
Els **objectes** poden representar qualsevol element del model conceptual, una entitat, una característica, un procés, una acció, una relació… En els objectes no únicament intentarem representar les característiques importants (dades), sinó que també vodrem fer referència al comportament o la funcionalitat que tindran en el moment de materialitzar-se durant l'execució de les aplicacions (codi). És a dir, que en un objecte es guarden tant les dades com les operacions (mètodes) que fem amb aquestes dades, tot junt.

La importància de centrar el model en els objectes és múltiple. Ente els avantages estan:

- En referència a les dades, els objectes actuen com estructures jeràrquiques, de manera que la informació queda sempre perfectament contextualitzada dins els objectes. Així, no té sentit referir-nos a una variable solta. Per exemple, en una aplicació d'una biblioteca el número de pàgines d'un llibre estarà sempre associat (i contingut) a un objecte llibre; igual que el títol, isbn i la resta de dades significatives. Podríem pensar que és igual que en les taules, on els camps o atributs pertanyen a una taula. La diferència, però, es troba en el fet que en el Model Relacional només es manté aquesta relació dins la taula, mentre que en el Model Orientat a Objectes aquesta relació s'estén a tota l'aplicació incorporant-la en el propi codi d'execució.

- En referència al comportament o la funcionalitat, els objectes delimiten les accions a realitzar sobre les seues dades i sobre la resta d'objectes, definint les regles del joc del que es pot fer durant l'execució de les aplicacions.

El següent dibuix s'intenta explicar aquesta manera de funcionar:



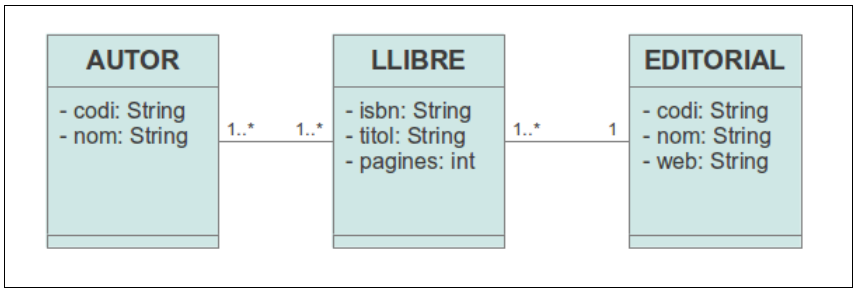
Les figures circulars representen objectes de diferents tipus segons el color. En el centre dels objectes, les petites

figures representen les dades encapsulades, inaccessibles de forma directa. Les corones circulars exteriors representen els mètodes, els quals a petició d'altres objectes (fletxes) poden consultar o manipular l'estat de l'objecte. Però no poden tocar directament les dades, sempre ha de ser a través dels mètodes que ofereix l'objecte.

Els valors del conjunt de dades que conformen un objecte en un moment determinat s'anomena també **estat**, perquè permet descriure l'evolució de qualsevol objecte durant l'execució d'una aplicació, des del moment de la seua creació fins que siguen eliminats de la memòria. Per tant, els estats dels objectes aniran variant al llarg de l'aplicació.

Malauradament, el Model conceptual Orientat a Objectes és un model eminentment dinàmic que **no contempla**, a priori, la **persistència dels seus objectes**. Per tant s'ha d'aconseguir poder guardar de forma permanent els estats dels objectes. S'haurà de portar des d'on estan guardats els objectes al principi de l'aplicació, inicialitzant-los en l'estat en què es trobaven quan es van guardar. També s'haurà d'anar guardant periòdicament l'estat dels objectes a mida que es vagen produint els canvis, de manera que hi haja sempre una correspondència entre els objectes en memòria i els seus estats emmagatzemats.

Si pensem en objectes, l'exemple anterior de la biblioteca podria quedar així (no hem posat els mètodes de cada classe, per centrar-nos en les dades):



Cosa que ens duria a les següents classes:

```
Autor {
        String cod_a;
        String nom;
}

Editorial {
        String cod_ed;
        String nom;
        String web;
}

Llibre {
        String isbn;
        Editorial editorial;
        ArrayList<Autor> autors;
        int pagines;
}
```

En aquest exemple, per simplificar, únicament hem plasmat les relacions entre classes a la classe **Llibre**, posant la referència a l'objecte **Editorial** al qual pertany el llibre, i un **ArrayList d'Autors**, on estarà la llista dels autors del llibre. En realitat, per a una major comoditat a l'hora de programar, el més lògic seria posar també en **Editorial** un **ArrayList<Llibre>** que arreplegue tots els llibres de l'editorial, i en **Autor** posaríem un **ArrayList<Llibre>** amb tots els llibres de l'autor. Però no les hem posades per fer l'exemple més senzill.

El Desfasament Objecte-Relacional

Si ens plantegem guardar els **objectes** d'una aplicació en un SGBD **Relacional**, el principal problema que trobarem és que es tracta de conceptes diferents, els objectes i les taules, i estan centrats en aspectes també diferents. El Model Entitat-Relació (que tindrà una traducció directa al Model Relacional) es troba fortament centrat en les dades i en l'estructura que cal donar a aquestes dades per poder guardar-les i recuperar-les. En canvi el Model Orientat a Objectes es troba centrat en els objectes, entesos com a agrupacions de dades i també com a un conjunt de processos de canvi, que afecten aquestes dades.

El Model Relacional necessitarà sempre certa quantitat d'informació extra destinada a mantenir les relacions i la coherència de les dades. Les claus externes són l'exemple més clar. Es tracta d'informació afegida en alguns registres per tal de vincular-los a uns altres. Així per exemple, per a saber el llibre de quina editorial és, afegim la clau externa que apunte a la taula Editorial, que serà un camp en la taula Llibre on es guardarà la clau principal de l'editorial a la qual pertany (per això es diu clau externa).

La vinculació entre objectes, en canvi, s'aconsegueix de forma estructural. No es necessiten dades extres, sinó que la mateixa estructura de dades defineix la vinculació, la visibilitat, l'accés, etc. Per exemple, per a guardar l'editorial d'un llibre, no guardem la seua clau principal com una clau externa en el llibre, sinó que guardem una referència a la mateixa editorial, a l'objecte editorial.

Aquestes diferències constitueixen el que en el món de la programació es coneix com a ***desfasament objecte-relacional***. Aquest desfasament ens obliga, quan decidim treballar conjuntament amb els dos models, un SGBD Relacional per a guardar les dades i un llenguatge Orientat a Objectes com per exemple Java, a codificar implementacions extres que funcionen a mode d'adaptadors. És a dir que hem de ***convertir*** o ***transformar*** els objectes en taules i a l'inrevés.

Mirem l'exemple comentat tant en el Model Relacional com en l'Orientat a Objectes: la biblioteca. Hi ha evidents diferències en guardar les coses d'una o altra manera.

Les diferències més clares són:

- La manera d'indicar l'editorial del llibre, en objectes és posar una referència a l'editorial, mentre que en el Model Relacional posem una clau externa. Quan anem a guardar les dades, s'haurà de substituir l'objecte editorial per la clau principal de l'editorial. I quan recuperem, serà el procés invers, hauríem d'utilitzar la clau externa per agafar l'editorial i guardar aquest objecte en l'objecte llibre.
- Més complicada és la manera d'indicar els autors del llibre. Com pot haver més d'un autor per llibre, en objectes posem un ArrayList de tipus Autor, mentre que en el Model Relacional hem de posar una nova taula. Guardar les dades d'un llibre suposarà en la pràctica guardar en dues taules, LLIBRE i AUTOR_LLIBRE. Les dades de LLIBRE es poden guardar directament, i les de AUTOR_LLIBRE suposarà un bucle per a recórrer l'ArrayList i fer una operació d'escriptura (INSERT) en AUTOR_LLIBRE per cada autor de la llista. A l'inrevès també haurem de fer un bucle, recorrent totes les files dels autors del llibre en qüestió, per a anar inserint en l'ArrayList.

També haurem de tenir present que el Model Relacional disposa d'un conjunt de llenguatges (DDL, DCL, SQL, etc.) adequats per explotar al màxim els SGBD tenint en compte les característiques relacionals, i així poder crear les taules (CREATE TABLE), inserir les dades (INSERT) i recuperar-les (SELECT), mentre que en la programació Orientada a Objectes es treballa bàsicament amb llenguatges de programació imperatius. Haurem d'incorporar per tant les sentències de SQL (i DDL, ...) dins del llenguatge amfitrió orientat a objectes (per exemple Java).

Un altre exemple de desfasament el trobem també en els resultats recuperats des d'un SGBD. Aquestos s'obtenen sempre en un format tabular i, per tant, caldrà implementar utilitats que transformen les seqüències de dades simples en estats dels objectes de l'aplicació, com havíem vist en l'exemple de la biblioteca.

3 - Connexió a les BD: Connectors

Deixem de moment de banda el desfasament Objecte-Relacional i centrem-nos ara en l'accés a Base de Dades Relacionals des dels llenguatges de programació. Ho raonarem en general i ho aplicarem a Java.

Des de la dècada dels 80 que existeixen a ple rendiment les Bases de Dades Relacionals. Quasi tots els Sistemes Gestors de Bases de Dades (excepte els més xicotets com Access o Base de LibreOffice) utilitzen l'arquitectura client-servidor. Això vol dir que hi ha un ordinador central on està instal·lat el Sistema Gestor de Bases de Dades Relacional que actua com a servidor, i hi haurà molts clients que es connectaran al servidor fent peticions sobre la Base de Dades.

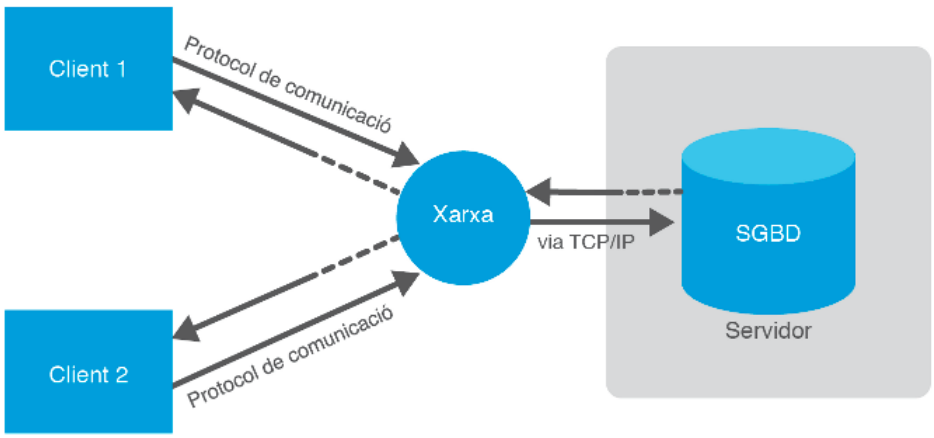
Els Sistemes Gestors de Bases de Dades inicialment disposaven de llenguatges de programació propis per a poder fer els accesos des dels clients. Era molt consistent, però a base de ser molt poc operatiu:

- L'empresa desenvolupadora del SGBD havien de mantenir un llenguatge de programació, que resultava necessàriament molt costós, si no volien que quedatra desfasat.
- Les empreses usuàries del SGBD, que es connectaven com a clients, es trobaven molt lligades al servidor per haver d'utilitzar el llenguatge de programació per accedir al servidor, cosa que no sempre s'ajustava a les seues necessitats. A més, el plantejar-se canviar de servidor, volia dir que s'havien de re-fer tots els programes, i per tant una tasca de moltíssima envergadura.

Per a poder ser més operatius, calia desvincular els llenguatges de programació dels Sistemes Gestors de Bases de Dades utilitzant uns estàndars de connexió.

3.1 - ODBC

A mida que les teories de dades relacionals anaven agafant força i les xarxes guanyaven adeptes gràcies a l'increment de l'eficiència a preus realment competitius, van començar a implementar-se uns Sistemes Gestors de Bases de Dades basats en la tecnologia client-servidor, que van triomfar.



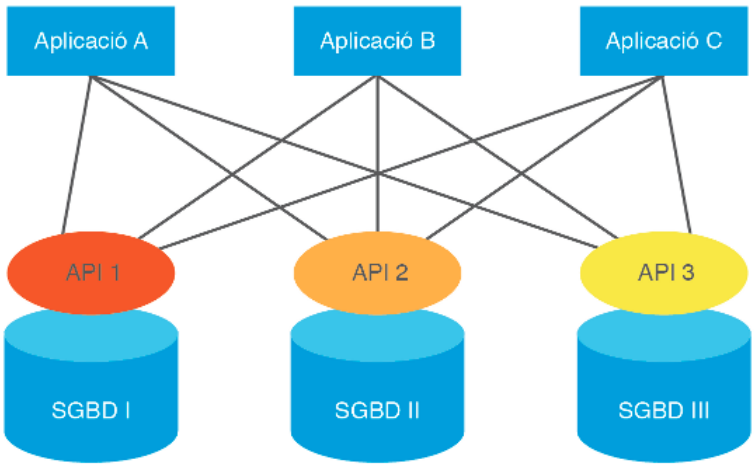
La tecnologia client-servidor va permetre aïllar les dades i els programes específics d'accés a aquestes dades, del desenvolupament de l'aplicació. La raó principal d'aquesta divisió va ser segurament possibilitar l'accés remot a les dades de qualsevol ordinador connectat a la xarxa. El cert, però, és que aquest fet va empènyer els sistemes de bases de dades a desenvolupar-se d'una forma aïllada i a crear protocols i llenguatges específics per poder-se comunicar remotament amb les aplicacions que corrien en els clients. Per dir-lo d'alguna manera, havien de desenvolupar els servidors i també els clients, per a poder connectar-se amb el servidor.

A poc a poc, el software al voltant de les bases de dades va créixer espectacularment intentant donar resposta a un màxim de demandes a través de sistemes altament configurables. És el que avui dia es coneix com a *middleware* o capa intermèdia de persistència. És a dir, el conjunt d'aplicacions, utilitats, biblioteques, protocols i llenguatges, situats tant a la part servidor com a la part client, que permeten connectar-se remotament a una base de dades per configurar-la o explotar-ne les seues dades.

L'arribada dels estàndards

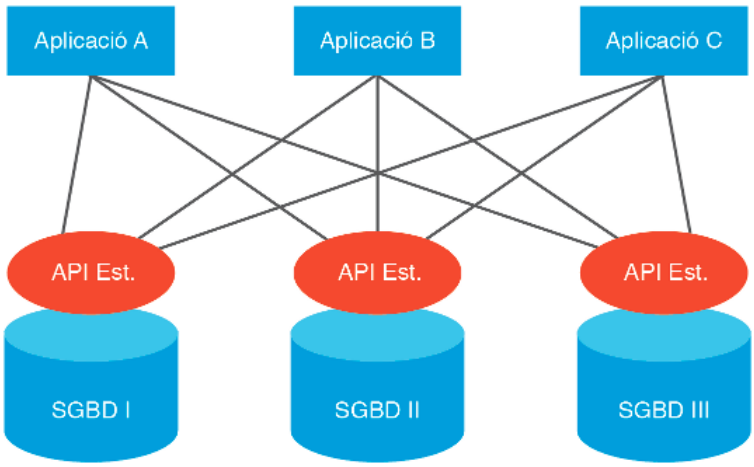
Inicialment, cada empresa desenvolupadora d'un SGBD implementava les seues solucions específiques per al seu sistema, però prompte es van donar compte que col·laborant conjuntament podien treure'n major rendiment i avançar molt més ràpidament.

Sostenint-se en el Model Relacional i en algunes implementacions primerenques de les empreses IBM i Oracle, es va desenvolupar el llenguatge de consulta de dades anomenat **SQL** (***Structured Query Language***). Va ser un gran pas endavant, perquè s'uniformava la manera d'accedir a la BD, però les aplicacions necessitaven API amb funcions que permeteren fer crides des del llenguatge de desenvolupament per enviar les consultes SQL.

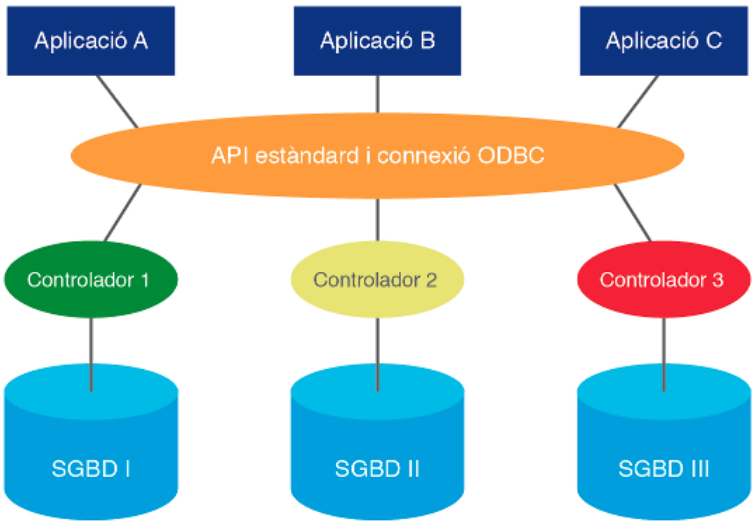


Cada SGBD té la seua pròpia connexió i el seu propi API.

El grup anomenat *SQL Access Group*, en el qual participaven prestigioses empreses del sector com Oracle, Informix, Ingres, DEC, Sun o HP, va definir un API universal amb independència del llenguatge de desenvolupament i la Base de Dades a connectar.



El 1992, Microsoft i Simba implementen l'**ODBC** (***Open Data Base Connectivity***), un API basat en la definició del *SQL Acces Group*, que s'integra en el sistema operatiu de Windows i que permet afegir múltiples **connectors** (o **controladors** o ***Drivers***) a diverses Bases de Dades Relacionals (que utilitzen SQL) de forma molt senzilla i transparent, ja que els connectors són autoinstal·lables i totalment configurables des de les mateixes eines del Sistema Operatiu. D'aquesta manera, tenint instal·lat ODBC (i en Windows acabarà venint instal·lat per defecte), les aplicacions es connectaran a través d'ODBC a qualsevol dels SGBD del qual tinguem instal·lat el connector. Per a connectar a una Base de Dades o una altra, només hem de canviar de connector, sense haver de canviar la pròpia aplicació.



L'arribada de l'ODBC va representar un pas sense precedents en el camí cap a la interoperabilitat entre bases de dades i llenguatges de programació. La majoria d'empreses desenvolupadores de Sistemes Gestors de Bases de Dades van proporcionar els *drivers* de connectivitat, i els llenguatges de programació més importants van desenvolupar biblioteques específiques per suportar l'API ODBC.

La situació actual

Actualment, ODBC continua sent una adequada manera de connectar als SGDB Relacionals. El seu desenvolupament segueix liderat per Microsoft, però existeixen versions per a altres Sistemes Operatius com UNIX/LINUX o MAC. Els llenguatges més populars de desenvolupament mantenen actualitzades les biblioteques de comunicació amb les successives versions que han anat apareixent i la majoria de SGBD disposen d'un controlador ODBC bàsic. I per tant la connexió queda garantida.

Actualment, l'ODBC s'estructura en tres nivells. El primer, anomenat *core API*, és el nivell més bàsic corresponent a l'especificació original (basada en el *SQL Access Group*). El *Level 1 API* i el *Level 2 API* afegeixen funcionalitats avançades, com cridades a procediments guardats en el Sistema Gestor de Bases de Dades, aspectes de seguretat d'accés, definició de tipus estructurats, etc.

En realitat, l'ODBC és una especificació de baix nivell, és a dir, de funcions bàsiques que possibiliten la connexió, que asseguren l'atomicitat de les peticions, el retorn d'informació, el capsulament del llenguatge de consulta SQL o l'obtenció de dades aconseguides en resposta a un petició.

La funcionalitat de baix nivell fa que es puga adaptar a moltes aplicacions; això sí, a costa d'un considerable nombre de línies de codi necessàries per adaptar-se a la lògica de cada aplicació. És per això que sobre la base de l'ODBC han sorgit altres alternatives de persistència de més alt nivell. Per exemple, Microsoft ha desenvolupat OLE DB o ADO.NET. Aquest últim possibilita ja els objectes per a qualsevol tipus d'aplicació basada en la plataforma .NET.

3.2 - JDBC

Pràcticament de forma simultània a ODBC, l'empresa Sun Microsystems, l'any 1997 va treure a la llum **JDBC** (***Java DataBase Connectivity***), un API connector de bases de dades, implementat específicament per a utilitzar amb el llenguatge Java. Es tracta d'un API molt similar a ODBC quant a la manera de funcionar:

- Tindrem l'API JDBC que utilitzaran les aplicacions que vulguen connectar a les Bases de Dades, i que garatitza una uniformitat, siga quina siga la o les Bases de Dasdes a connectar
- Ens farà falta un **controlador** o ***driver*** per cada Base de Dades a la qual vulguem connectar

Però com comentàvem, està adaptat específicament per a Java. És a dir, la funcionalitat es troba encapsulada en classes (ja que Java és un llenguatge totalment orientat a objectes) i a més, no depèn de cap plataforma especifica, d'acord amb la característica multiplataforma defensada per Java.

Aquest connector serà l'API que estudiarem en detall en aquesta unitat, ja que Java no disposa de cap biblioteca específica ODBC. Les raons esgrimides per *Sun* són que ODBC no es pot fer servir directament en Java ja que està implementat en C i no és orientat a objectes.

Però, per a no perdre la potencialitat de les connexions ODBC, que recordem que ens permetia connectar a qualsevol Base de Dades Relacional, Sun Microsystem ha optat per incorporar de sèrie un *driver* especial que actua d'adaptador entre l'especificació JDBC i l'especificació ODBC. Aquest controlador s'acostuma a anomenar també **pont JDBC-ODBC** (*bridge* en anglès). Utilitzant aquest *driver* podrem enllaçar qualsevol aplicació Java amb qualsevol connexió ODBC.

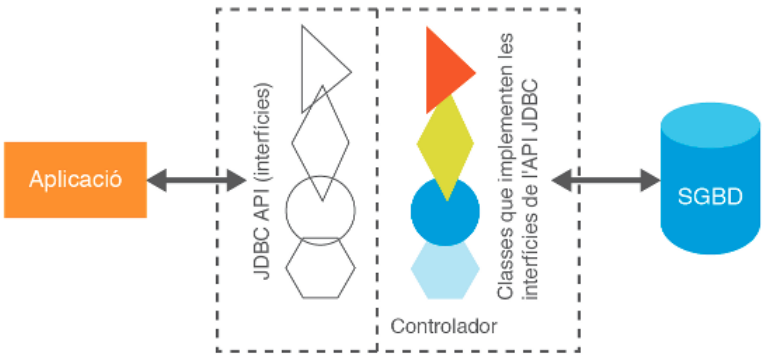
Actualment, la gran majoria d'SGBD disposen de *drivers* JDBC, però en cas d'haver de treballar amb un sistema que no en tinga, si disposa de controlador ODBC, podrem fer servir el pont JDBC-ODBC per aconseguir la connexió des de Java.

Arquitectura JDBC

- La biblioteca estàndard **JDBC** conté un gran nombre d'interfícies *sense* les classes que les implementen.
- Els controladors o drivers dels SGBD concrets (i que els proporciona el fabricant del SGBD) són els que han d'implementar les interfícies anteriors i així accedir a les seues dades amb les particularitats que puga tenir el SGBD en concret.

Des de les aplicacions s'utilitzaran les interfícies de JDBC, i d'aquesta manera, el controlador utilitzat serà totalment transparent a l'aplicació.

D'aquesta manera s'aconsegueix independitzar l'aplicació dels controladors permetent la substitució del controlador original per qualsevol altre compatible JDBC sense pràcticament necessitat d'haver de modificar el codi de l'aplicació.



D'una banda trobem les interfícies definides a l'estàndard (les figures amb fons transparent). Es tracta de l'API amb el que l'aplicació treballarà de forma directa.

De l'altra banda trobem les classes específiques del controlador (driver) que interaccionen amb el SGBD i que implementen les interfícies de l'estàndard JDBC. Són les figures amb fons de diferents colors

És important destacar que JDBC no exigeix cap instal·lació, ni cap canvi substancial en el codi a l'hora de fer servir un o altre controlador. I fins i tot podrem utilitzar més d'un controlador per a poder connectar des de la mateixa aplicació a més d'un SGBD. Això és possible perquè:

- Java permet carregar en memòria qualsevol classe a partir del seu nom, i així carregar el o els controladors que necessitem. Es fa amb la sentència **Class.forName** ("nom_de_la_Classe").
- La classe **DriverManager** (de l'API JDBC) és capaç de seleccionar el driver adequat d'entre tots els que estiguen carregats en memòria, sense necessitat d'indicar-li el *driver* específic que cal fer servir.

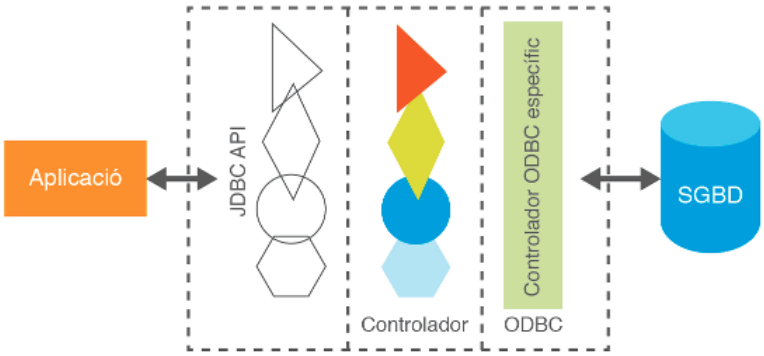
Tipus de controladors

JDBC distingeix quatre tipus de controladors:

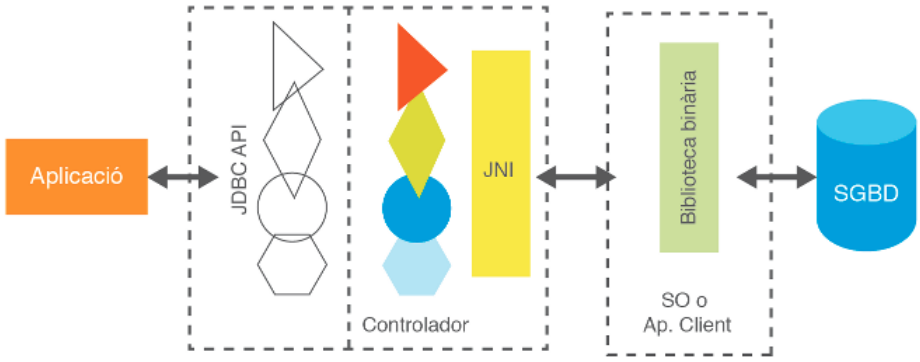
1. Tipus I. Controladors **pont** (*bridge driver*) com JDBC-ODBC. Es caracteritzen per fer servir una tecnologia externa a JDBC i actuar d'adaptador entre les especificacions de l'API JDBC i la tecnologia concreta utilitzada. El més conegut és el controlador **pont JDBC-ODBC**, però n'hi ha d'altres, com JDBC-OLE DB. La seua principal raó de ser és la de permetre utilitzar l'atra tecnologia (ODBC) que està molt estesa i assegurar així la connexió amb pràcticament qualsevol font de dades. En cada client haurem de tenir:

- Haurà de tenir instal·lada una utilitat de gestió i configuració de fonts de dades ODBC (o de la tecnologia utilitzada)
- Haurà de tenir instal·lat el *driver* ODBC específic del'SGBD al qual es vol accedir
- A través de la primera utilitat crear un DSN (Data Source Name) que utilitze el driver del SGBD amb les dades de connexió al SGBD

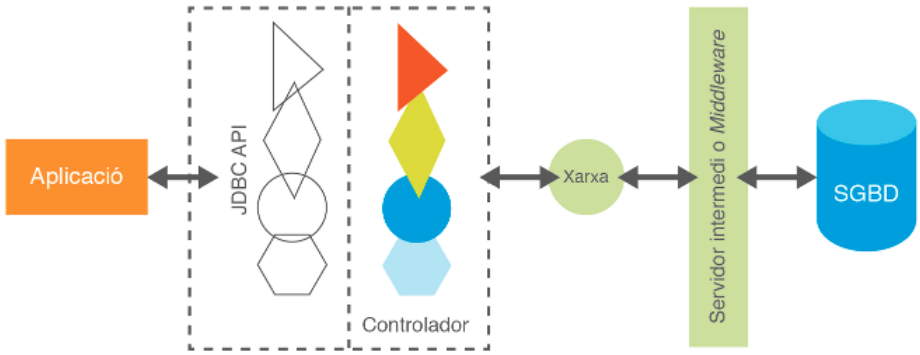
Com que la connexió és a través de ODBC (no directament) pot donar problemes de rendiment i, per tant, s'aconsella fer servir aquest tipus de controlador només com a última alternativa.



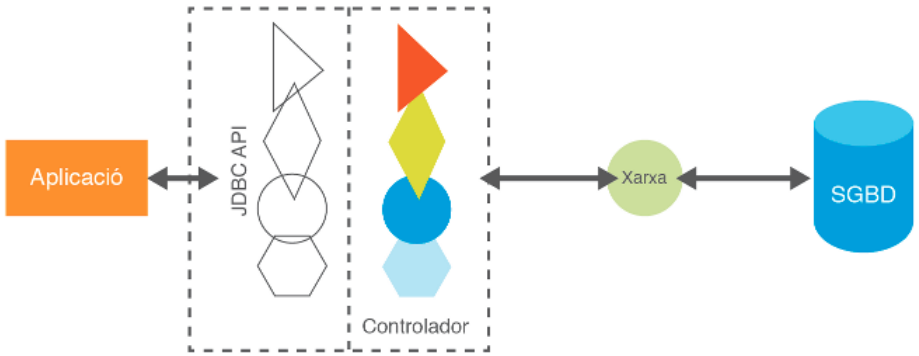
2. Tipus II. Controladors de **Java amb API parcialment nadiu** (*Native-API partly Java driver*). S'anomenen també simplement *nadius*. Com el seu nom indica, estan formats d'una part codificada en Java i una altra part que usa biblioteques binàries instal·lades en el sistema operatiu. Aquest tipus de controladors existeixen perquè alguns Sistemes Gestors de Bases de Dades tenen entre les seues utilitats de sèrie connectors propis. Solen ser connectors propietaris que no segueixen cap estàndard, ja que acostumen a ser anteriors a ODBC o JDBC, però es mantenen perquè solen estar molt optimitzats i són molt eficients. Utilitzant una tecnologia Java anomenada JNI és possible implementar classes, els mètodes de les quals invoquen funcions de biblioteques binàries instal·lades en les sistema operatiu. Els controladors de tipus II utilitzen aquesta tecnologia per crear les classes implementadores de l'API JDBC. En alguns casos pot requerir una instal·lació extra de certes utilitats a la part client, exigides pel connector nadiu del sistema gestor.



3. Tipus III. Controladors de **Java via protocol de xarxa**. Es tracta d'un controlador escrit totalment en Java que tradueix les cridades JDBC a un protocol de xarxa contra un servidor intermedi (anomenat normalment *Middleware*) que pot estar connectat a diversos SGBD. Aquest tipus de *driver* presenta l'avantatge que utilitza un protocol independent dels SGBD i, per tant, el canvi de font de dades es pot fer de manera totalment transparent als clients. Això el converteix en un sistema molt flexible, encara que per contra, es necessitarà instal·lar, en algun lloc accessible de la xarxa, un servidor intermedi connectat a tots els SGBD que calga. Aquest tipus de controladors són molt útils quan hi ha un número molt gran de clients, ja que els canvis d'SGBD no requeriran cap canvi en els clients, ni tan sols la incorporació d'una nova biblioteca.



4. Tipus IV. Controladors de tipus **Java pur** o **Java 100%**. S'anomenen també controladors de *protocol nadiu*. Són controladors escrits totalment en Java. Les peticions al Sistema Gestor de Bases de Dades es fan sempre a través del protocol de xarxa que utilitza el propi SGBD i, per tant, no es necessita ni codi nadiu en el client (com en el cas del tipus II) ni servidor intermedi (com en el cas del tipus III) per connectar amb la font de dades. Es tracta, doncs, d'un *driver* que no requereix cap tipus d'instal·lació ni requeriment, la qual cosa el fa ser una alternativa molt ben considerada que en els últims temps ha acabat imposant-se. De fet, la majoria de fabricants han acabat creant un controlador de tipus IV, tot i que segueixen mantenint també els dels altres tipus.



Els tipus desitjables són el tipus III i sobretot el **tipus IV**, ja que d'aquesta manera no ens fa falta instal·lar res per a poder connectar.

En aquest tema intentarem connectar a 4 Sistemes Gestors de Bases de Dades diferents: **PostgreSQL**, **Oracle**, **MySQL** i **SQLite**

Els drivers o controladors (tots de tiups IV) us els podeu baixar sense problemes. Són fàcils de trobar, únicament amb un buscador posar **jdbc** i el Sistema Gestor de Bases de Dades (per exemple **jdbc postgresql**). De tota manera, per més comoditat, teniu **una carpeta en el curs de Moodle amb tots els drivers** que ens fan falta.

4 - Perspectiva de Desenvolupament de Bases de Dades en Eclipse

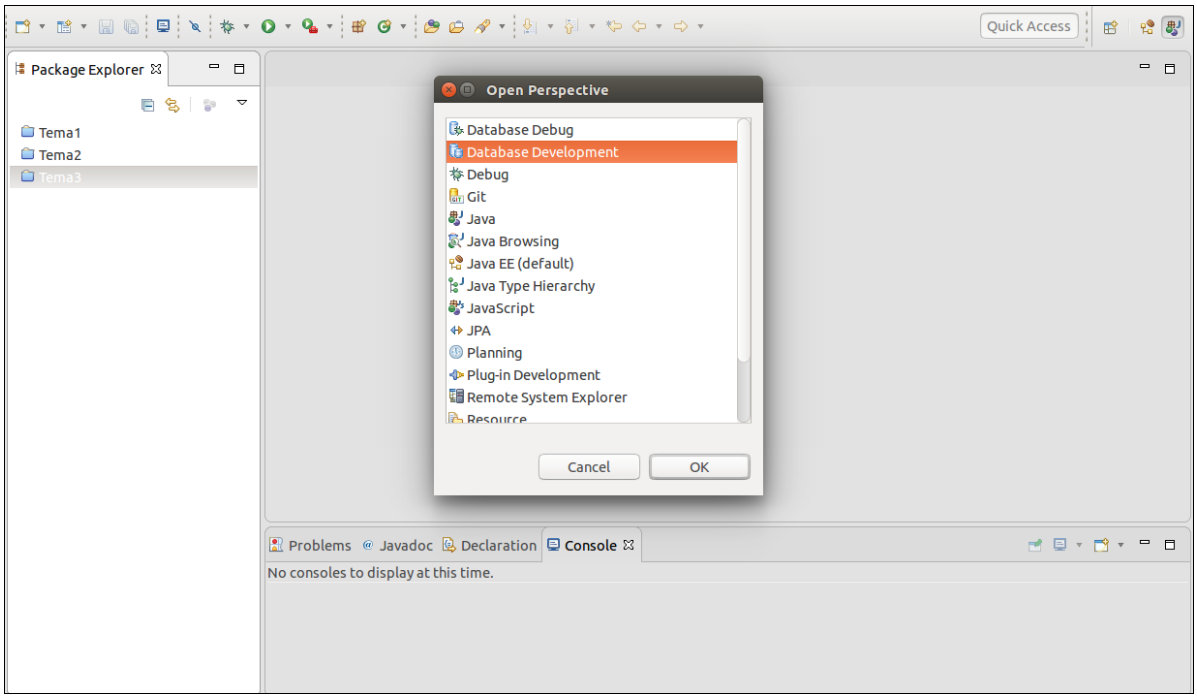
Abans de començar a treballar la persistència en diferents Bases de Dades Relacionals, ens configurarem Eclipse per a poder treballar còmodament amb elles.

Des dels programes Java podríem connectar ja mateix amb les diferents BD amb els connectors JDBC, però abans ens muntarem una perspectiva per a poder treballar, veure les taules amb les dades i fins i tot administrar aquestes Bases de Dades. Utilitzarem els connectors (drivers) JDBC, però des d'un entorn, una perspectiva més còmoda. Si no tinguérem aquesta perspectiva que ens permet "veure i tocar" les taules, o bé ahuríem de saber perfectíssimament l'estructura de totes les taules de la Base de Dades, o tenir una eina pròpia del SGBD per poder veure les taules. Com que són molts els SGBD que anem a connectar ens convé la perspectiva

La perspectiva és **Desenvolupament de Bases de Dades** (**Database Development**). Depenent de la instal·lació d'Eclipse que tinguem, aquesta perspectiva estaria ja disponible. En concret, en la que us vaig recomanar instal·lar (**Eclipse IDE for Java EE Developers**) ja està instal·lada. La manera d'accedir és:

Window --> Perspective --> Open Perspective --> Other

I triem la perpectiva **Database Development**.



Si tinguérem una versió d'Eclipse que no té aquesta perspectiva, la podrem instal·lar. Millor dit, instal·laríem les eines (tools) que ens permeten tenir-la.

El següent vídeo explica el procés d'instal·lació i canvi en la perspectiva (recordeu que si ja teníeu la perspectiva no us cal fer açò)

Connexió a PostgreSQL

Com a primer exemple de connexió, intentarem connectar a PostgreSQL. Farà falta especificar:

- Primer el controlador (driver) de PostgreSQL. Triarem un de tipus IV (en el vídeo es veu el procés de baixada del driver; l'enllaç és aquest JDBC42 Postgresql Driver, Version 9.4.1211), però recordeu que també teniu el driver més actual en el curs de Moodle)
- Després les dades de connexió, que seran:
  - Servidor: **89.36.214.106**
  - Base de dades: **geo_ad**
  - Usuari: **geo_ad**
  - Contrasenya: **geo_ad**

De la configuració anterior podem observar la **URL** de connexió que va a continuació. És important, perquè després quan connectem des dels nostres programes Java, haurem d'especificar-la també. Haurem de substituir *servidor* per l'adreça IP o el nom del servidor, la *base_de_dades* per la base de dades a la qual ens volem connectar.

URL de PostgreSQL

```
jdbc:postgresql://servidor:5432/base_de_dades
```

Connexió a Oracle

Per a poder connectar a Oracle, utilitzarem un usuari genèric que està en quasi totes les instàncies d'Oracle anomenat SCOTT. El servidor està en una altra màquina que la resta de SGBD, per tant pareu atenció a l'adreça del servidor. Farà falta especificar:

- Primer el controlador (driver) de Oracle (el podeu baixar des d'ací ojdbc6.jar, però us faria falta donar-vos d'alta de forma gratuïta; per més comoditat el teniu a l'aula virtual)
- Després les dades de connexió, que seran:
  - Servidor: **94.177.240.173**
  - Instància: **orcl**

- Base de Dades: **scott**
- Usuari: **scott**
- Contrasenya: **tiger**

Aquest vídeo comenta el procés (suposa que ja ens hem baixat el driver JDBC apropiat):

I la cadena de connexió que ens quedarà serà

URL d'Oracle

```
jdbc:oracle:thin:@servidor:1521:instancia
```

Connexió a MySQL

Repetim el procés per a MySQL. El servidor està en la mateixa màquina que PostgreSQL (89.36.214.106). L'usuari amb que connectarem ara es diu **factura**. Haurem d'especificar:

- Primer el controlador (driver) de MySQL (el podeu baixar des d'ací https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.44.tar.gz, però per a més comoditat el teniu disponible a l'aula virtual)
- Després les dades de connexió, que seran:
  - Servidor: **89.36.214.106**

- Base de Dades: **factura**
- Usuari: **factura**
- Contrasenya: **factura**

URL de MySQL

```
jdbc:mysql://servidor:3306/base_de_dades
```

Connexió a SQLite

SQLite és un SGBD molt diferent als anteriors. En tots els anteriors s'utilitza l'arquitectura client-servidor. I per tant el servidor de Base de Dades no té per què estar en la mateixa màquina.

Però en moltes ocasions ens pot venir bé un SGBD molt més xicotet i senzill que guarde la BD en la mateixa màquina, que siga monousuari i que pese molt poc.

SQLite és un SGBD multiplataforma (podrem fer-lo rodar en qualsevol plataforma) que és molt lleuger. Admet la major part de l'estàndar SQL-92. I guarda una Base de Dades en un únic fitxer. Una altra característica és que des de Java, amb el driver JDBC podrem accedir a les Bases de Dades SQLite sense que faça falta cap motor de Base de Dades.

Per tant és ideal com a Base de Dades que puguem copiar a les nostres aplicacions.

També intentarem connectar des de la perspectiva. Però ara no caldrà especificar on està el servidor ni quin usuari s'ha de connectar. Haurem de dir el fitxer amb la ruta on està (o estarà) la Base de Dades. Per tant només haurem d'especificar:

- Primer el controlador (driver) de SQLite (sqlite-jdbc-3.14.2.1.jar)
- Després les dades del fitxer: ruta i nom. La ruta ha d'existir. Si el fitxer no existeix, el crearà.

URL de SQLite

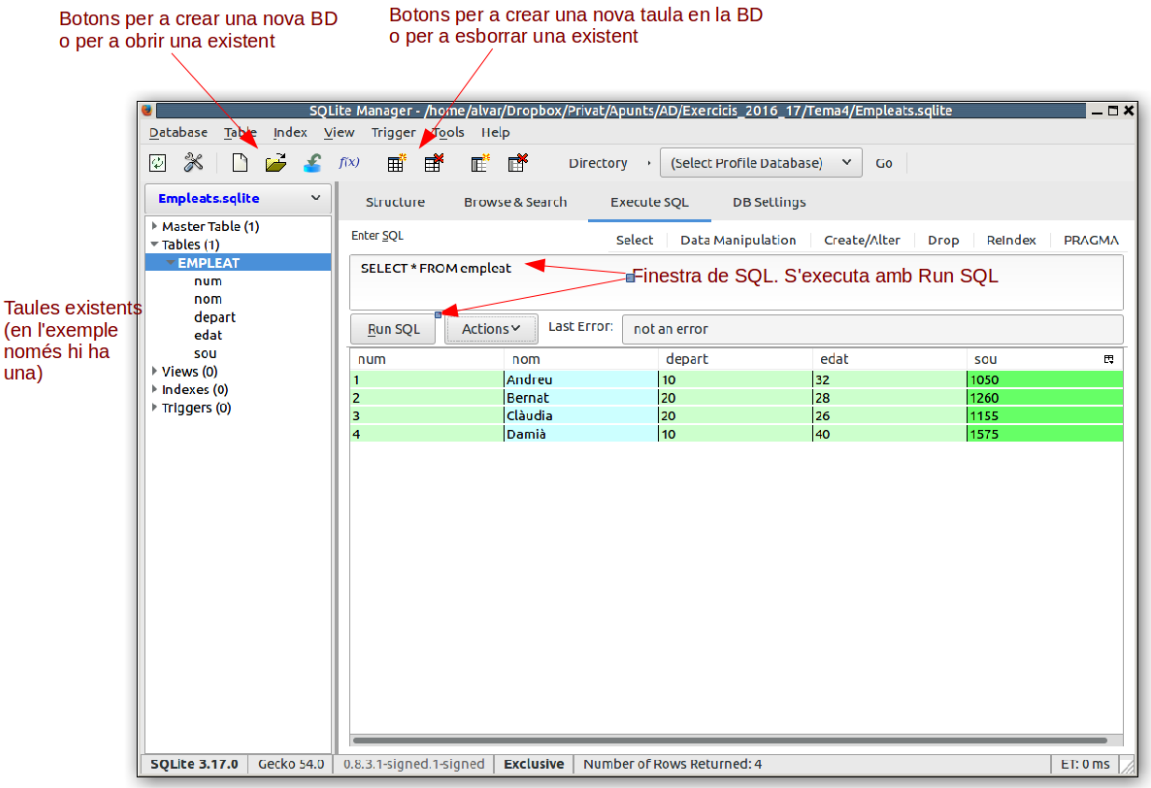
```
jdbc:sqlite:ruta_del_fitxer_sqlite
```

Com hem comentat, SQLite és més senzill que els altres SGBD. Concretament, els tipus de dades que utilitza són:

- **INTEGER**. El valor és un enter amb signe, que ocupa 1, 2, 3, 4, 6, o 8 bytes depenent de la grandària del valor.
- **REAL**. El valor és un número real en coma flotant que ocupa 8 bytes (doble precisió).
- **TEXT**. El valor és una cadena de caràcters que pot estar codificada en UTF-8, UTF-16BE o UTF-16LE, depenent de la codificació de la Base de Dades. No ens preocuparem de la codificació interna.
- **BLOB**. Per a guardar dades binàries que es guardaran exactament com entren, sense mirar el format.

Des de la perspectiva de Bases de Dades podrem visualitzar les taules i fins i tot crear-les, amb SQL.

Com que les Bases de Dades SQLite les haurem d'administrar del tot, si resulta un poc incòmoda l'administració des de la perspectiva d'Eclipse, podem utilitzar alguna eina que ens ho facilite. Segurament la més còmoda és un **plugin** per a **Firefox** anomenat **SQLite Manager**. És molt fàcil d'instal·lar (només heu de buscar **Firefox SQLite Manager**, i segur que la primera opció serà la d'instal·lar el plugin). Una vegada instal·lat segur que haureu de reiniciar Firefox si ja l'estàveu utilitzant. La seua utilització és intuïtiva. En aquesta imatge estem veient una BD ja creada, que crearem i utilitzarem més avant:

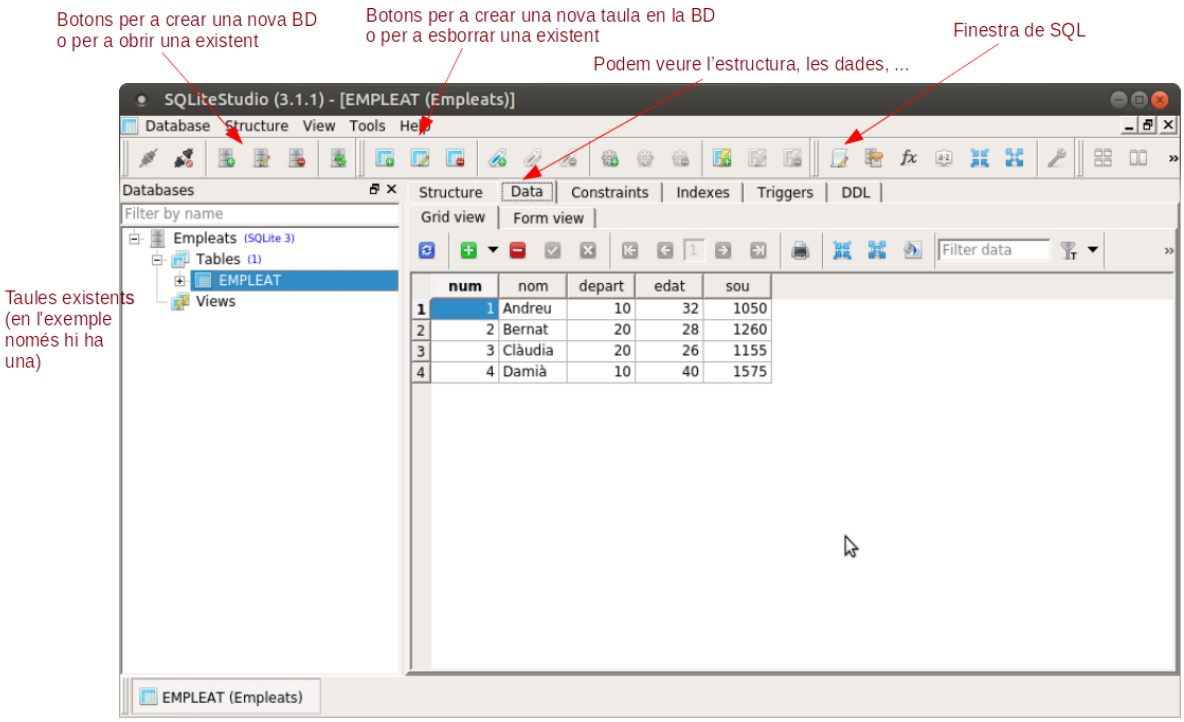


Lamentablement, no funciona aquest plugin de moment en l'última versió de Firefox. Segurament ho arreglaran prompte.

Hi ha una altra alternativa, que és la d'utilitzar el **SQLiteStudio**, un programa en versió portable que no cal instal·lar, únicament descomprimir siga quina siga la plataforma utilitzada. El podem baixar de la següent adreça:

https://sqlitestudio.pl/index.rvt

Aquest és el seu aspecte, que com veieu resulta molt còmode:


Botons per a crear una nova BD
o per a obrir una existent
Botons per a crear una nova taula en la BD
o per a esborrar una existent
Finestra de SQL
Podem veure l'estructura, les dades, ...
Taules existents
(en l'exemple
només hi ha
una)
SQLiteStudio (3.1.1) - [EMPLEAT (Empleats)]
Database Structure View Tools Help
Databases
Filter by name
Empleats (SQLite 3)
Tables (1)
EMPLEAT
Views
Structure Data Constraints Indexes Triggers DDL
Grid view Form view
Filter data
num nom depart edat sou
1 Andreu 10 32 1050
2 Bernat 20 28 1260
3 Clàudia 20 26 1155
4 Damià 10 40 1575
EMPLEAT (Empleats)

5 - Iniciació a l'API JDBC

Ara veurem els elements bàsics de l'API JDBC que permeten a les aplicacions Java comunicar-se amb un SGBD fent servir el llenguatge SQL. Cal que disposeu del connector **JDBC** dels 4 SGDB que anem a utilitzar: **PostgreSQL**, **Oracle**, **MySQL** i **SQLite**. I també que els afegiu a les biblioteques del vostre projecte. També serà necessari que habiliteu una connexió per consultar la Base de Dades des de la perspectiva **Database Development** (com les que hem creat en el punt anterior).

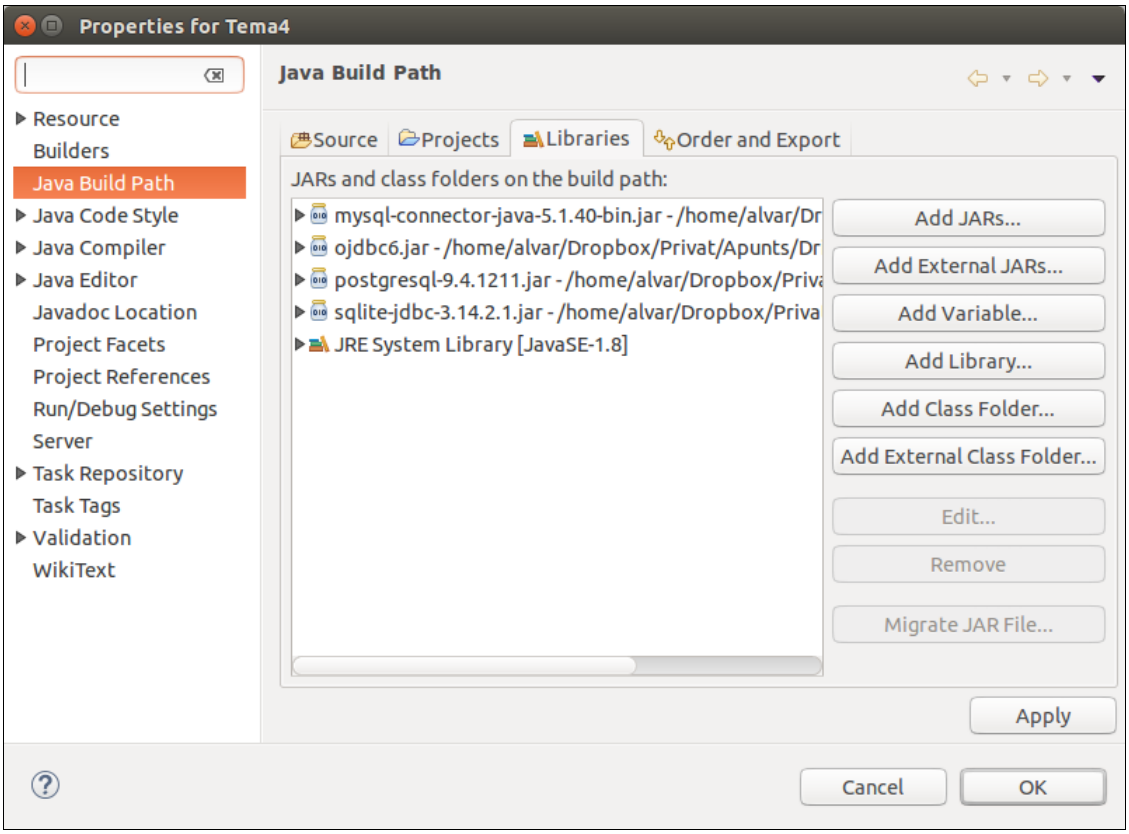
Per tal de poder practicar la connexió i accedir a les dades, tenim una Bases de Dades de prova en cada SGBD:

- PosgreSQL: **geo_ad** (contrasenya: **geo_ad**)
- Oracle: **scott** (contrasenya: **tiger**)
- MySQL: **factura** (contrasenya: **factura**)
- SQLite: **proveta.sqlite**, però en realitat crearem les Bases de Dades que ens facen falta

Crearem un projecte nou anomenat **Tema4**, per exemple, i li afegirem els **controladors JDBC** de **PostgesSQL**, **Oracle**, **MySQL** i **SQLite** com a biblioteca del projecte. Per a separar les proves dels exercicis, ens creem un **paquet** anomenat **Exemples**.

Els drivers o controladors us els podeu baixar sense problemes. Són fàcils de trobar, únicament amb un buscador posar **jdbc** i el Sistema Gestor de Bases de Dades (per exemple **jdbc postgresql**). De tota manera, per més comoditat, teniu **una carpeta en el curs de Moodle amb tots els drivers** que ens fan falta

En la següent imatge es veu com hem incorporat els drivers al projecte (es fa en les **propietats** del projecte, secció **Java Build Path**, pestanya **Libraries**, amb el botó **Add Extetrnals JARs...**)



En els punts següents anem a comentar cada cosa per separat, pas a pas. De tota manera, posem la llista inicial de classes que utilitzarem per a poder connectar i accedir a la Base de Dades:

Classe	Descripció

Driver	Permet connectar a una Base de Dades
DriverManager	Permet gestionar tots els drivers instal·lats al sistema
Connection	Representa una connexió amb una BD. En una aplicació pot haver més d'una connexió
Statement	Permet executar sentències SQL sense paràmetres
PreparedStatement	Permet executar sentències SQL amb paràmetres
ResultSet	Conté les files resultants d'executar una sentència SELECT
DatabaseMetadata	Proporciona informació d'una BD, com per exemple les taules que conté
ResultSetMetadata	Proporciona informació sobre un ResultSet: número de columnes, noms de les columnes, tipus, ...

5.1 - Càrrega de controladors

Generalment, en una aplicació es pot tenir un número tan elevat de classes que la màquina virtual no pot tenir-les carregades totes en memòria. A mida que va sent necessari, la màquina virtual s'encarregarà de carregar en memòria les classes que es necessiten. Normalment, la màquina virtual descobreix la localització exacta de la ruta on es troba la classe a carregar analitzant les sentències ***import***. En general és una forma molt útil i eficient de detectar la ubicació de les classes d'una aplicació.

El problema que tenim ara és que per poder utilitzar l'*import* **necessitem conèixer a priori la classe que farem servir**. A més, una vegada escrita la sentencia *import*, si algun dia necessitem reanomenar la classe o decidim utilitzar una classe equivalent d'un altre paquet, necessitarem reescriure el codi canviant la sentència **import** i recompilant-lo de nou per fer efectius els canvis.

Afortunadament, Java disposa d'una altra sentència per localitzar els fitxers compilats en el moment de la càrrega. Es el mètode **Class.forName**, que accepta com a paràmetre una cadena de text amb el nom complet de la classe a carregar (paquet i nom de classe). Qualsevol controlador JDBC disposa d'una classe especial anomenada generalment **Driver**, encarregada d'establir la connexió amb el nostre SGBD. En realitat el nom i el paquet de la classe depenen de cada fabricant i, per tant, caldrà consultar la documentació per conèixer el nom de la classe.

En el cas de **PostgreSQL**, per indicar la classe a carregar, escriurem**:**

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

Per a **Oracle**:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Per a **MySQL**:

```
Class.forName("com.mysql.jdbc.Driver")
```

I per a **SQLite**:

```
Class.forName("org.sqlite.JDBC");
```

Encara que no sempre és necessari posar aquesta sentència, ens anirà bé per assegurar-nos que accedim al driver. En cas de posar-la, cal escriure-la sempre abans de començar a utilitzar l'API JDBC.

El mètode **forName** de la classe **Class** pot llençar una excepció en cas que no es trobe la classe. En aquest exemple hem fet ***throws*** sobre l'error, però una altra possibilitat seria capturar l'error (amb **try ... catch**) per saber si s'ha escrit el nom correctament o que ens hem oblidat d'afegir el controlador al projecte.

```
public class carregaControladors {

    public static void main(String[] args) throws ClassNotFoundException {
                Class.forName("org.postgresql.Driver");
                System.out.println("Ja s'ha carregat el controlador");
    }
}
```

5.2 - Establiment de la connexió

Un objecte de la classe **Connection** (de l'API JDBC) representarà una connexió a la Base de Dades d'un determinat SGBD. Haurem de tenir el controlador del SGBD inclós en el projecte (en el classpath), i millor si el tenim carregat en memòria amb la sentència **Class.forName('*nom de la classe del driver*')**.

L'objecte **Connection** l'obtindrem a partir del **DriverManager**, que havíem comentat que és capaç de gestionar tots els drivers carregats en memòria (i amb **Class.forName** ens asseguràvem que estaven en memòria). El mètode que utilitzarem del **DriverManager** és el **getConnection(*url*,*usuari*,*password*)**, on li passarem les dades de connexió a la Base de Dades d'aquesta manera:

- **URL**: cadena de connexió seguint el protocol JDBC. Comença sempre per **jdbc**, el nom del SGBD (separat per dos punts), i la manera d'arribar a la BD, també separat per 2 punts. Aquesta manera d'arribar a la BD dependrà del controlador del SGBD, però d'alguna manera harem d'especificar el *servidor*, el *port* de connexió i el nom de la BD o esquema a connectar.
- **Usuari** i **contrasenya**: encara que en alguns SGBD (com per exemple SQLite) no seran necessaris.

Aquestes són les ***url*** que utilitzarem:

- **PostgreSQL**: per connectar-nos al servidor situal a l'adreça **89.36.214.106**, que escolta el port per defecte (**5432**), i a la Base de Dades anomenada **geo_ad**, la cadena serà:

```
jdbc:postgresql://89.36.214.106:5432/geo_ad
```

- **Oracle**: el tenim en un altre servidor, en l'adreça **94.177.240.173**; el port per defecte d'Oracle és el **1521**, i hem d'especificar la instància (la macro Base de Dades d'Oracle) que es diu **orcl** ; observeu que en la cadena no especifiquem l'esquema, que seria l'equivalent a la mini Base de Dades on volem connectar (en Oracle un usuari es connecta sempre al seu esquema). La cadena de connexió serà:

```
jdbc:oracle:thin:@94.177.240.173:1521:orcl
```

- **MySQL**: el servidor és el mateix de PostgreSQL, el port per defecte és 3306, i si volem connectar a la Base de Dades **geo**:

```
jdbc:mysql://89.36.214.106:3306/factura
```

- **SQLite**: no haurem d'especificar ni servidor ni port (ni posteriorment usuarini contrasenya); únicament el nom del fitxer amb la ruta. Si volem connectar a la Base de Dades situada en el directori **/home/usuari/BD_SQLite**, i anomenada **proveta.sqlite**:

```
jdbc:sqlite:/home/usuari/BD_SQLite/proveta.sqlite
```

Mirem quatre exemple de connexió, un per a cada Base de Dades de prova que tenim en els diferents SGBD que ens hem plantejat connectar. Observeu com només hem canviat la classe que carreguem amb **Class.forName** i la **url**, a banda de l'usuari i contrasenya, clar (en SQLite no hi haurà). I el millor de tot és que una vegada feta la connexió, farem el mateix tractament siga quin siga el SGBD al qual ens hem connectat, com veurem amb posterioritat.

- **PostgreSQL**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class provaConnexioPostgreSQL {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:postgresql://89.36.214.106:5432/geo_ad";
        String usuari = "geo_ad";
        String password = "geo_ad";

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

```
        Connection con = DriverManager.getConnection(url, usuari, password);
        System.out.println("Connexió completada");
        con.close();
    }
}
```

- **Oracle**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class provaConnexioOracle {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:oracle:thin:@94.177.240.173:1521:orcl";
        String usuari = "scott";
        String password = "tiger";

        Class.forName("oracle.jdbc.driver.OracleDriver");

        Connection con = DriverManager.getConnection(url, usuari, password);
        System.out.println("Connexió completada");
        con.close();
    }
}
```

- **MySQL**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class provaConnexioMySQL {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:mysql://89.36.214.106:3306/factura";
        String usuari = "factura";
        String password = "factura";

        Class.forName("com.mysql.jdbc.Driver");

        Connection con = DriverManager.getConnection(url, usuari, password);
        System.out.println("Connexió completada");
        con.close();
    }
}
```

- **SQLite**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class provaConnexioSQLite {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:sqlite:proveta.sqlite";

        Class.forName("org.sqlite.JDBC");

        Connection con = DriverManager.getConnection(url);
        System.out.println("Connexió completada");
        con.close();
    }
}
```

Com que no hem posat ruta, crearà el fitxer **proveta.sqlite** en el directori actiu, és a dir, en l'arrel del projecte. També podríem posar la ruta d'aquesta manera:

```
String url = "jdbc:sqlite:/home/usuari/BD_SQLite/proveta.sqlite";
```

en aquest cas haureu de cuidar que existesca la ruta del fitxer, sinó es produirà una SQLException

Observem les següents qüestions en els quatre programes equivalents anteriors:

- Com ja hem comentat alguna vegada, la sentència **Class.forName()** no seria necessària en moltes aplicacions. Però ens assegura que hem carregat el driver, i per tant el **DriverManager** el sabrà gestionar.
- El **DriverManager** és capaç de trobar el driver adequat a través de la **url** proporcionada (sobretot si el driver està carregat en memòria), i és qui ens proporciona l'objecte **Connection** per mig del mètode **getConnection()**. Hi ha una altra manera d'obtenir el **Connection** per mig de l'objecte **Driver**, com veurem al final d'aquest pregunta, però també serà passant indirectament pel **DriverManager**.
- Si no es troba la classe del driver (per no tenir-lo en les llibreries del projecte, o haver escrit malament el seu nom) es produirà l'excepció ClassNotFoundException. És convenient tractar-la amb **try ... catch**, encara que en els exemples anteriors s'ha optat per fer **throws** per a simplificar-los.
- Si no es pot establir la connexió per alguna raó es produirà l'excepció **SQLException**. Igual que en al cas anterior, és convenient tractar-la amb **try ... catch**.
- L'objecte Connection mantindrà una connexió amb la Base de Dades des del moment de la creació fins el moment de tancar-la amb **close()**. És molt important tancar la connexió, no solament per alliberar la memòria del nostre ordinador (que en tancar l'aplicació s'alliberaria), sinó sobretot per **tancar la sessió oberta en el Servidor de Bases de Dades**.
- En el cas de SQLite només s'ha d'especificar la ***url***. No hem dit ni usuari ni contrasenya, que no tenen sentit en aquest SGBD monousuari.

Una manera de connectar alternativa a les anteriors és utilitzant l'objecte **Driver**. La classe **java.sql.Driver** pertany a l'API **JDBC**, però no és instanciable, i tan sols és una interfície, per a que les classes **Driver** dels contenidors hereten d'ella i implementen la manera exacta d'accedir al SGBD corresponent. Com no és instanciable (no podem fer **new Driver()**) la manera de crear-lo és a través del métode **getDriver()** del **DriverManager**, que seleccionarà el driver adequat a partir de la **url**. Ja només restaran definir algunes *propietats,* com l'usuari i la contrasenya, i obtenir el **Connection** per mig del mètode **connect()**

La manera de connectar a través d'un objecte **Driver** és més llarga, però més completa ja que es podrien especificar més coses. I potser ajude a entendre el muntatge dels controladors dels diferents SGBD en Java.

```
import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class provaConnexioPostgreSQLAmbDriver {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url="jdbc:postgresql://89.36.214.106:5432/geo_ad";
        String usuari="geo_ad";
        String password="geo_ad";

        Driver driver = DriverManager.getDriver(url);

        Properties properties = new Properties();
        properties.setProperty("user", usuari);
        properties.setProperty("password", password);

        Connection con = driver.connect(url, properties);
        System.out.println("Connexió completada a través de Driver");
        con.close();
    }
}
```

En aquest exemple s'ha optat per no carregar el driver amb **Class.forName()**. Segurament la major part de vegades funcionarà, però podria ser que no funcionara en un moment determinat. Com ja hem comentat, el **Class.forName()** ens assegura que estarà carregat en memòria.

5.3 - Peticions bàsiques

Per escriure sentències SQL, JDBC disposa dels objectes **Statement**. Es tracta d'objectes que s'han de crear a partir de ***Connection***, els quals poden enviar sentències SQL al SGBD connectat per a que s'executen amb el mètode **executeQuery** o **executeUpdate**.

Hi ha una variant del **Statement**, anomenada **PreparedStatement** que ens dóna més versatilitat per a posar paràmetres i executar la sentència d'una altra manera. El veurem en la pregunta 6.5.

La diferència entre els dos mètodes que executen sentències SQL és:

- El mètode **executeQuery** serveix per executar sentències de les quals s'espera **que tornen dades**, és a dir, són consultes **SELECT**.
- En canvi, el mètode **executeUpdate** serveix específicament per a sentències que no retornen dades. Serviran per a modificar la Base de Dades connectada (**INSERT**, **DELETE**, **UPDATE**, fins i tot **CREATE TABLE**) .

Sentències que no retornen dades

Les executem amb el mètode **executeUpdate**. Seran **totes** les sentències SQL **excepte el SELECT**, que és la de consulta. És a dir, ens servirà per les següents sentències:

- Sentències que canvien les estructures internes de la BD on es guarden les dades (instruccions conegudes amb les sigles **DDL**, de l'anglès *Data Definition Language*), com per exemple **CREATE TABLE**, **CREATE VIEW**, **ALTER TABLE**, **DROP TABLE**, ...,
- Sentències per atorgar permisos als usuaris existents o crear-ne de nous (subgrup d'instruccions conegudes com a **DCL** o *Data Control Language*), com per exemple **GRANT**.
- I també les sentències per a modificar les dades guardades fent servir les instruccions **INSERT**, **UPDATE** i **DELETE**.

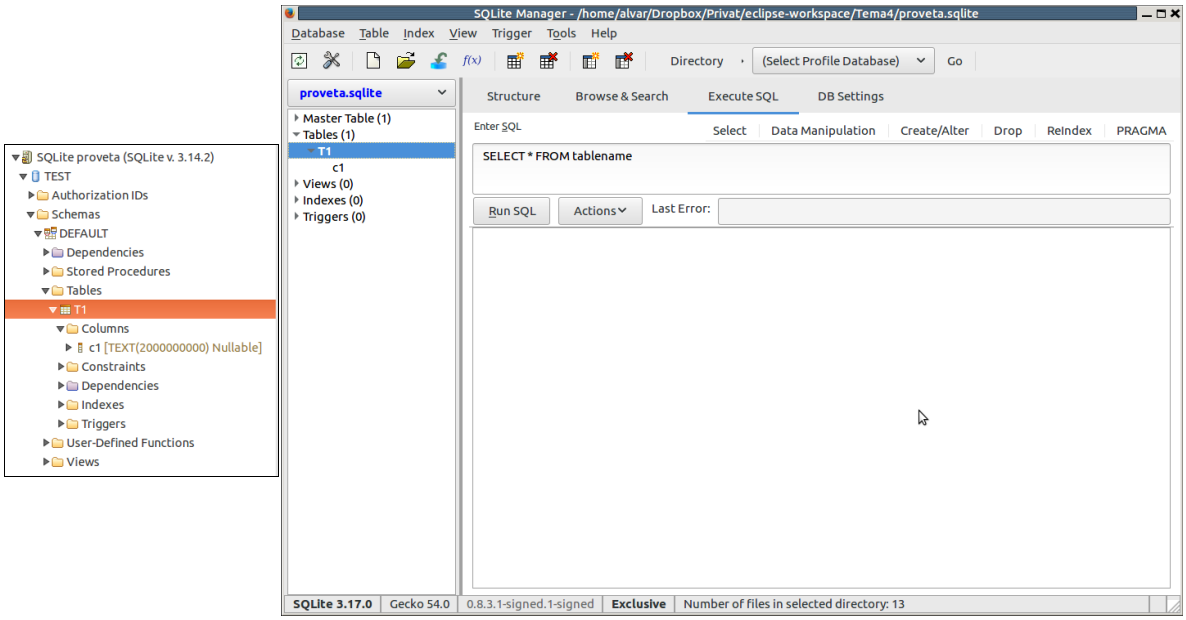
Encara que es tracta de sentències molt dispars, des del punt de vista de la comunicació amb el SGBD es comporten de manera molt similar, seguint el patró següent:

1. Instanciació del **Statement** a partir d'una connexió activa.
2. Execució d'una sentència SQL passada per paràmetre al mètode **executeUpdate**.
3. Tancament de l'objecte **Statement** instanciat.

Mirem aquest exemple, en qual crearem una taula molt senzilla en la Base de Dades SQLite **proveta.sqlite**.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class provaSQLiteCreacioTaula {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:sqlite:proveta.sqlite";

        Class.forName("org.sqlite.JDBC");

        Connection con = DriverManager.getConnection(url);

        Statement st = con.createStatement();
        st.executeUpdate("CREATE TABLE T1 (c1 TEXT)");
        st.close();

        con.close();
    }
}
```

Des de la perspectiva de Bases de Dades (esquerra) podrem comprovar que ara ja existeix la taula, igual que des del SQLite Manager de Firefox (dreta):



Sentències que retornen dades

Les executem amb el mètode **executeQuery**. Servirà per a la sentència **SELECT**, que és la de consulta.

Les dades que ens torne aquesta sentència les haurem de guardar en un objecte de la classe **ResultSet**, és a dir conjunt de resultat. Per tant, l'execució de les consultes tindrà un forma semblant a la següent:

```
ResultSet rs = st.executeQuery(sentenciaSQL);
```

L'objecte **ResultSet** conté el resultat de la consulta organitzat per files, de manera que en cada moment es pot consultar **una fila**. Per a anar visitant totes les files d'una a una, anirem cridant el mètode **next()** de l'objecte **ResultSet**, ja que cada vegada que s'execute **next** s'avançarà a la següent fila. Immediatament després d'una execució, el **ResultSet** es troba posicionat just abans de la primera fila, per tant per accedir a la primera fila caldrà executar **next** una vegada. Quan les files s'acaben, el mètode **next** retornarà fals.

Des de cada fila es podrà accedir al valor de les seues columnes fent servir uns quants mètodes **get** disponibles segons el tipus de dades a retornar i passant per paràmetre el número de columna que desitgem obtenir. El nom dels mètodes comença per **get** seguit del **nom del tipus de dades**. Així, si volem recuperar la segona columna, sabent que és una dada de tipus String caldrà executar:

```
rs.getString(2);
```

Les columnes es comencen a comptar a partir del valor **1** (**no zero**). La major part dels SGDB suporten la possibilitat de passar per paràmetre el nom de la columna, però no tots, així que normalment s'opta pel paràmetre numèric.

En el següent exemple, mostrem el contingut de les dues primeres columnes de la taula **INSTITUT** de la Base de Dades **geo_ad** de **PostgreSQL**, que resulten ser el codi numèric de l'Institut i el seu nom:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class consultaPostgreSQL {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:postgresql://89.36.214.106:5432/geo_ad";
        String usuari = "geo_ad";
        String password = "geo_ad";

        Connection con = DriverManager.getConnection(url, usuari, password);

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

```
        ResultSet rs = st.executeQuery("SELECT * FROM institut");
        while (rs.next()){
            System.out.print(rs.getInt(1) + "\t");
            System.out.println(rs.getString(2));
        }
        st.close();

        con.close();
    }
}
```

I ara mirem un altre programa per a accedir a les dues primeres columnes de la taula **EMP** de l'usuari **SCOTT** en **ORACLE**, on estaran el número de l'empleat i el seu nom. Observeu com només hem canviat la cadena de connexió, usuari i contrasenya, a banda de la sentència SQL, ja que ara estem accedint a una altra taula:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class consultaOracle {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:oracle:thin:@94.177.240.173:1521:orcl";
        String usuari = "scott";
        String password = "tiger";

        Connection con = DriverManager.getConnection(url, usuari, password);

        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT * FROM emp");
        while (rs.next()){
            System.out.print(rs.getInt(1) + "\t");
            System.out.println(rs.getString(2));
        }
        st.close();

        con.close();
    }
}
```

En aquest exemple, on accedim a **MySQL**, accedirem a una altra taula:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class consultaMySQL {

    public static void main(String[] args) throws ClassNotFoundException,
SQLException  {
        String url = "jdbc:mysql://89.36.214.106:3306/factura";
        String usuari = "factura";
        String password = "factura";

        Connection con = DriverManager.getConnection(url, usuari, password);

        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT * FROM poble");
        while (rs.next()){
            System.out.print(rs.getInt(1) + "\t");
            System.out.println(rs.getString(2));
        }
        st.close();

        con.close();
    }
}
```

No reutilització de Statement ni ResultSet

És un error prou habitual per inesperat el fet d'intentar reutilitzar un mateix **ResultSet** per a arreplegar més d'una consulta. . I el mateix amb el **Statement**. Bé siga per una mala implementació o un bug o el que siga, el comportament pot ser imprevisible. I per tant no val la pena arriscar-se.

Us aconselle que si en una aplicació teniu més d'una consulta de les que retornen dades, **utilitzeu un Statement i un ResultSet diferents per a cadascuna**.

No hi ha problema en utilitzar el mateix Statement per a moltes consultes de les que no retornen dades.

Assegurar l'alliberament de recursos

Les instàncies de **Connection** i les de **Statement** guarden, en memòria, molta informació relacionada amb les execucions realitzades. A més, mentre continuen actives mantenen en el SGBD una sessió oberta, que suposarà un conjunt important de recursos oberts, destinats a servir de forma eficient les peticions dels clients. És important tancar aquestos objectes per a alliberar recursos tant del client com del servidor.

Si en un mateix mètode hem de tancar un objecte **Statement** i el **Connection** a partir del qual l'hem creat, s'haurà de tancar primer el **Statement** i després el **Connection**. Si ho fem al revés, quan intentem tancar el **Statement** ens saltarà una excepció de tipus **SQLException**, ja que el tancament de la connexió l'hauria deixat inaccessible.

A més de respectar l'orde, caldrà assegurar l'alliberament dels recursos situant les operacions de tancament dins un bloc ***finally***. D'aquesta manera, encara que es produesquen errors, no es deixaran d'executar les instruccions de tancament.

Cal tenir en compte encara un detall més quan siga necessari realitzar el tancament de diversos objectes a la vegada. En aquest cas, encara que les situàrem una darrera l'altra, totes les instruccions de tancament dins el bloc *finally*, no seria prou garantia per assegurar l'execució de tots els tancaments, ja que, si mentre es produeix el tancament d'un dels objectes es llança una excepció, els objectes invocats en una posició posterior a la del que s'ha produït l'error no es tancaran.

La solució d'aquest problema passa per evitar el llançament de qualsevol excepció durant el procés de tancament. Una possible forma és encapsular cada tancament entre sentències ***try-catch*** dins del ***finally***

```
try{
   //sentències que poden llançar una excepció
   ...
} catch (SQLException ex) {
   // captura i tractament de l'excepció
   ...
}finally{
   try {
      stm1.close();
   } catch (SQLException ex) {...}

   try {
      stm2.close();
   } catch (SQLException ex) {...}

   ...

   try {
      con.close();
   } catch (SQLException ex) {...}
}
```

De vegades, l'error en un tancament es produeix perquè l'objecte mai ha arribat a instanciar-se i, per tant, la variable presenta un valor *null*, o perquè ja ha estat tancat amb anterioritat. Ambdós casos són previsibles, i es pot evitar l'error fent servir una instrucció condicional que evite tancar-lo quan ja estava tancat.

```
...
try {
   // Assegurem que la connexió està instanciada i oberta
   if (con!=nul && !con.isClosed() {
      // tanquem la connexió
      con.close();
   }
} catch (SQLException ex) { ... }
```

5.4 - Exemple

A continuació posarem un exemple molt senzill, el dels empleats, en el qual primer crearem la taula per a guardar les dades, després introduirem les dades, les modificarem, i per últim les consultarem. D'aquesta manera podrem veure tots els exemples de sentències SQL.

Les dades són les mateixes que en altres ocasions:

Num	Nom	Depart	Edat	Sou
1	Andreu	10	32	1000.00
2	Bernat	20	28	1200.00
3	Claudia	10	26	1100.00
4	Damià	10	40	1500.00

La clau principal serà el camp num de tipus enter. El nom serà de text, el departament i l'edat també enters, mentre que el sou serà real.

Per a no interferir entre tots, utilitzarem la Base de Dades SQLite, on cadascú guardarà en un fitxer seu les dades. Haureu de cuidar la ruta, que siga on voleu guardar la vostra Base de Dades. Si no poseu res en la ruta, es guardarà en el directori actiu, que és l'arrel del projecte.

Creació de la taula

La sentència SQL que crea la taula en una Base de Dades SQLite serà així:

```
CREATE TABLE EMPLEATS (
   num INTEGER CONSTRAINT cp_emp PRIMARY KEY,
   nom TEXT,
   depart INTEGER,
   edat INTEGER,
   sou REAL );
```

Hem d'observar que el mètode del **Statement** a utilitzar és **executeUpdate()**, ja que la sentència de creació no torna res (no és un SELECT)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class exempleEmpleatsCrearTaula {

    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        String sentSQL = null;

        try {
            Class.forName("org.sqlite.JDBC");

            String url = "jdbc:sqlite:Empleats.sqlite";
            con = DriverManager.getConnection(url);

            st = con.createStatement();

            sentSQL = "CREATE TABLE EMPLEAT(" +
                    "num INTEGER CONSTRAINT cp_emp PRIMARY KEY, " +
                    "nom TEXT, " +
                    "depart INTEGER, " +
                    "edat INTEGER, " +
                    "sou REAL " +
                ")";

            st.executeUpdate(sentSQL);
            st.close();

        } catch (SQLException ex) {
            System.out.println("Error " + ex.getMessage());
```

```
        } catch (ClassNotFoundException ex) {
            System.out.println("No s'ha trobat el controlador JDBC (" +
ex.getMessage() + ")");
        } finally {
            try {
                if (st != null && !st.isClosed()) {
                    st.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Statement per
alguna raó");
            }
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Connection per
alguna raó");
            }
        }
    }

}
```

Si voleu veure el resultat, podeu anar a la perspectiva Database Development, però recordeu que heu de desconnectar després, per no interferir.

Inserció de dades

També volem introduir les dades que es poden veure a la taula anterior. Crearem un **Statetement** que reutilitzarem per anar escrivint totes les sentències **INSERT**. Recordeu que no és problema la reutilització si gastem el **executeUpdate()**.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class exempleEmpleatsInserirTaula {

    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        String sentSQL = null;

        try {
            Class.forName("org.sqlite.JDBC");

            String url = "jdbc:sqlite:Empleats.sqlite";
            con = DriverManager.getConnection(url);

            st = con.createStatement();

            sentSQL = "INSERT INTO EMPLEAT VALUES (1,'Andreu',10,32,1000.0)";
            st.executeUpdate(sentSQL);

            sentSQL = "INSERT INTO EMPLEAT VALUES (2,'Bernat',20,28,1200.0)";
            st.executeUpdate(sentSQL);

            sentSQL = "INSERT INTO EMPLEAT VALUES (3,'Clàudia',10,26,1100.0)";
            st.executeUpdate(sentSQL);

            sentSQL = "INSERT INTO EMPLEAT VALUES (4,'Damià',10,40,1500.0)";
            st.executeUpdate(sentSQL);

        } catch (SQLException ex) {
            System.out.println("Error " + ex.getMessage());
        } catch (ClassNotFoundException ex) {
            System.out.println("No s'ha trobat el controlador JDBC (" +
ex.getMessage() + ")");

        } finally {
            try {
                if (st != null && !st.isClosed()) {
                    st.close();
                }
            } catch (SQLException ex) {
```

```
                System.out.println("No s'ha pogut tancar el Statement per
alguna raó");
            }
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Connection per
alguna raó");
            }
        }
    }
}
```

Ara sí que és un bon moment per a consultar la taula des de la perspectiva Database Development. Recordeu que heu de desconnectar després de consultar la taula.

Modificació de dades

Ara modificarem les dades. Senzillament augmentem el sou de tots els empleats en un 5%. I també modifiquem el departament de l'empleat 3, posant-li el departament 20.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class exempleEmpleatsModificarTaula {

    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        String sentSQL = null;

        try {
            Class.forName("org.sqlite.JDBC");

            String url = "jdbc:sqlite:Empleats.sqlite";
            con = DriverManager.getConnection(url);

            st = con.createStatement();

            sentSQL = "UPDATE EMPLEAT SET sou = sou * 1.05";
            st.executeUpdate(sentSQL);

            sentSQL = "UPDATE EMPLEAT SET depart=20 WHERE num = 3";
            st.executeUpdate(sentSQL);

        } catch (SQLException ex) {
            System.out.println("Error " + ex.getMessage());
        } catch (ClassNotFoundException ex) {
            System.out.println("No s'ha trobat el controlador JDBC (" +
ex.getMessage() + ")");

        } finally {
            try {
                if (st != null && !st.isClosed()) {
                    st.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Statement per
alguna raó");
            }
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Connection per
alguna raó");
            }
        }
    }
}
```

Consultar les dades

Vegem de quina manera podem mostrar per pantalla tots els empleats del que cobren més de 1.100€. Ara el mètode que utilitzarem és **executeQuery()**, ja que aquesta consulta sí que torna dades.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class exempleEmpleatsConsultarTaula {

    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        String sentenciaSQL = null;
        ResultSet rs = null;

        try {
            Class.forName("org.sqlite.JDBC");

            String url = "jdbc:sqlite:Empleats.sqlite";
            con = DriverManager.getConnection(url);

            st = con.createStatement();

            sentenciaSQL = "SELECT * FROM EMPLEAT WHERE sou > 1100";
            rs = st.executeQuery(sentenciaSQL);

            System.out.println("Núm. \tNom \tDep \tEdat \tSou");
            System.out.println("----------------------------------------");

            while (rs.next()) {
                System.out.print(rs.getInt(1) + "\t");
                System.out.print(rs.getString(2) + "\t");
                System.out.print(rs.getInt(3) + "\t");
                System.out.print(rs.getInt(4) + "\t");
                System.out.println(rs.getDouble(5));
            }

        } catch (SQLException ex) {
            System.out.println("Error " + ex.getMessage());
        } catch (ClassNotFoundException ex) {
            System.out.println("No s'ha trobat el controlador JDBC (" +
ex.getMessage() + ")");

        } finally {
            try {
                if(rs!=null && !rs.isClosed()){
                    rs.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el ResultSet per
alguna raó");
            }
            try {
                if (st != null && !st.isClosed()) {
                    st.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Statement per
alguna raó");
            }
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException ex) {
                System.out.println("No s'ha pogut tancar el Connection per
alguna raó");
            }
        }
    }
}
```

Podeu observar com es pot usar un bucle **while** per obtenir el valor de totes les files retornades. També podeu veure els diferents mètodes que retornen les dades de cada columna en funció del tipus: **getInt()**, **getString()**, **getDouble()**,

...

Finalment, cal indicar que els objectes **ResultSet** també s'han de tancar de la mateixa manera que els **Statements** o les connexions. Els **ResultSet** són els primers que caldrà tancar.

6 - JDBC avançat

JDBC disposa d'una alta funcionalitat i estructures que poden ajudar-nos a incrementar la qualitat de les aplicacions que construïm.

Volem construir aplicacions flexibles, robustes i eficients. Necessitarem, doncs, un bon tractament d'errors que trasllade quan faça falta la informació adequada a l'usuari o reconduint el flux de l'execució cap a processos que interpreten i compensen els errors.

L'eficiència és també una característica important de la qualitat. En general, els Sistemes Gestors de Bases de Dades disposen de mecanismes automàtics per potenciar l'eficiència de les peticions, com ara l'ús de memòria caché d'accés ràpid, la creació d'índexs automàtics, etc. Aquestos automatismes responen a determinats patrons a l'hora de fer les peticions. Per això JDBC preveu altres formes, diferents a les estudiades fins ara, per realitzar peticions que milloren el rendiment.

6.1 - Accés a les Meta Dades

Normalment quan accedim a una Base de Dades des d'un programa Java, coneixerem l'estructura d'aquesta Base de Dades, és a dir, les taules que té i cada taula quins camps i de quin tipus són. I fins i tot les claus externes entre les taules.

Però podria passar que no coneguérem aquesta estructura. Això no hauria de ser un problema per accedir a la Base de Dades, ja que aquesta és autodescriptiva, és a dir, que hem de poder consultar a la Base de Dades per a que ens diga la seua estructura.

L'objecte que ens dóna aquesta informació és el **DatabaseMetaData**. Disposa de molts mètodes que ens proporcionen gran quantitat d'informació.

I també ens dóna informació un altre objecte, **ResultSetMetaData**, que obtindrà la informació a partir d'un **ResultSet**. Veurem els dos per separat.

DatabaseMetaData

És l'adequat quan volem veure les taules i vistes que tenim en la Base de Dades, així com l'estructura de cadascuna d'elles.

Mirem primer els mètodes més interessants que té:

Valor tornat	Nom del mètode	Descripció
String	**getDatabaseProductName()**	Torna el nom del SGBD
String	**getDriverName()**	Torna el driver JDBC utilitzat
String	**getURL()**	Torna la URL de la connexió
String	**getUserName()**	Torna el nom de l'usuari que s'ha connectat
ResultSet	**getTables(cataleg, esquema, nom, tipus)**	Torna informació de totes les taules que coincideixen amb els patrons o criteris. Si posem null a tots els paràmetres ens tornarà totes les taules i vistes
ResultSet	**getColumns(cataleg, esquema, nom, nom_columna)**	Torna informació de les columnes de la taula especificada en el tercer paràmetre (que està en el catàleg i en l'esquema, si els posem). El quart paràmetre servirà per a seleccionar les columnes que coincidesquen amb el patró (null per a totes)
ResultSet	**getPrimaryKeys(cataleg, esquema, taula)**	Torna la llista de camps que formen la clau principal
ResultSet	**getImportedKeys(cataleg, esquema, taula)**	Torna una llista amb les claus externes definides en aquesta taula
ResultSet	**getExportedKeys(cataleg, esquema, taula)**	Torna una llista amb les claus externes que apunten a aquesta taula

Per poder comprovar tot l'anterior, farem un programa que ens done informació de la Base de Dades, una relació de les taules i vistes que té.

Posteriorment acceptarà un número, i traurà la informació de la taula corresponent amb aquest format: columnes, tipus, clau principal, claus externes.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.*;
import java.util.ArrayList;
```

```
public class MetaData {

    public static void main(String[] args)
            throws SQLException, ClassNotFoundException,
NumberFormatException, IOException {
        Class.forName("org.postgresql.Driver");
        Connection con =
DriverManager.getConnection("jdbc:postgresql://89.36.214.106:5432/geo_ad",
"geo_ad", "geo_ad");

        DatabaseMetaData dbmd = con.getMetaData();
        System.out.println("Informació general");
        System.out.println("------------------");
        System.out.println("SGBD: " + dbmd.getDatabaseProductName());
        System.out.println("Driver: " + dbmd.getDriverName());
        System.out.println("URL: " + dbmd.getURL());
        System.out.println("Usuari: " + dbmd.getUserName());
        System.out.println();
        System.out.println("Llistat de taules:");
        System.out.println(String.format("%-6s %-7s %-7s %-10s
%-10s","Número","Catàleg","Esquema","Nom","Tipus"));
        System.out.println("-----------------------------------------------");
        ResultSet ll = dbmd.getTables(null, "public", null, null);
        int compt = 1;
        ArrayList<String> taules = new ArrayList<String>();
        while (ll.next()) {
            System.out.println(String.format("%-6d %-7s %-7s %-10s %-10s",
(compt++),ll.getString(1),ll.getString(2),ll.getString(3),ll.getString(4)));
            taules.add(ll.getString(3));
        }
        System.out.println();
        System.out.println("Introdueix un número per veure l'estructura de la
taula (0 per acabar): ");
        BufferedReader ent = new BufferedReader(new
InputStreamReader(System.in));
        int opcio = Integer.parseInt(ent.readLine());

        while (opcio != 0) {
            if (opcio < compt && opcio > 0) {
                ResultSet taula = dbmd.getTables(null, "public",
taules.get(opcio - 1), null);
                if (taula.next()){
                    if (taula.getString(4).equals("TABLE")){
                        ResultSet rs = dbmd.getColumns(null, "public",
taules.get(opcio - 1), null);
                        System.out.println("Estructura de la taula " +
taules.get(opcio - 1));
                        System.out.println("----------------------------");
                        while (rs.next())
                            System.out.println(rs.getString(4) + " (" +
rs.getString(6) + ")");
                        System.out.println("----------------------------");

                        rs = dbmd.getPrimaryKeys(null, "public",
taules.get(opcio - 1));
                        System.out.print("Clau principal: ");
                        while (rs.next())
                            System.out.print(rs.getString(4) + " ");
                        System.out.println();

                        rs = dbmd.getImportedKeys(null, "public",
taules.get(opcio - 1));
                        System.out.println("Claus externes: ");
                        while (rs.next()) {
                            System.out.println(rs.getString(8) + " apunta a "
+ rs.getString(3));
                        }
                        rs.close();
                    }
                }
                taula.close();

            }
            System.out.println();
            System.out.println("Introdueix un número per veure l'estructura de
la taula (0 per acabar): ");
            opcio = Integer.parseInt(ent.readLine());
        }
        ll.close();
        con.close();
    }
}
```

ResultSetMetaData

Una vegada executada una sentència SELECT de SQL que ja tenim el resultat en un ResultSet, podem accedir també a meta dades d'aquest ResultSet, obtenint per exemple el número de columnes, o el tipus de les columnes. Ho obtenim per mig del **ResulSetMetaData**.

Només veurem 3 mètodes

Valor tornat	Nom del mètode	Descripció
int	**getColumnCount()**	Torna el número de columnes del ResultSet
String	**getColumnName(index)**	Torna el nom de la columna (la primera columna és la 1)
String	**getColumnTypeName(index)**	Torna el tipus de la columna

El següent exemple és una modificació del programa anterior, on ara traurem tot el contingut d'una taula per mig de la sentència **SELECT * FROM *taula***. Intentarem donar-li un aspecte tabular, però sense patir molt per l'aspecte.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

public class ResultSetMetaData {
    public static void main(String[] args)
            throws SQLException, ClassNotFoundException,
NumberFormatException, IOException {
        Class.forName("org.postgresql.Driver");
        Connection con =
DriverManager.getConnection("jdbc:postgresql://89.36.214.106:5432/geo_ad",
"geo_ad","geo_ad");

        DatabaseMetaData dbmd = con.getMetaData();
        System.out.println("Llistat de taules:");
        System.out.println(String.format("%-6s %-7s %-7s %-10s %-10s",
"Número", "Catàleg", "Esquema", "Nom", "Tipus"));
        System.out.println("-----------------------------------------");
        ResultSet ll = dbmd.getTables(null, "public", null, null);
        int compt = 1;
        ArrayList<String> taules = new ArrayList<String>();
        while (ll.next()) {
            System.out.println(String.format("%-6d %-7s %-7s %-10s %-10s",
(compt++), ll.getString(1), ll.getString(2),ll.getString(3),
ll.getString(4)));
            taules.add(ll.getString(3));
        }
        System.out.println();
        System.out.println("Introdueix un número per veure el contingut de la
taula (0 per acabar): ");
        BufferedReader ent = new BufferedReader(new
InputStreamReader(System.in));
        int opcio = Integer.parseInt(ent.readLine());

        while (opcio != 0) {
            if (opcio < compt && opcio > 0) {
                ResultSet taula = dbmd.getTables(null, "public",
taules.get(opcio - 1), null);
                if (taula.next()) {
                    if (taula.getString(4).equals("TABLE")) {
                        ResultSet rs =
con.createStatement().executeQuery("SELECT * FROM " + taules.get(opcio - 1));
                        System.out.println("Contingut de la taula " +
taules.get(opcio - 1));
                        System.out.println("----------------------------");

                        java.sql.ResultSetMetaData rsmd = rs.getMetaData();
                        for (int i = 1; i <= rsmd.getColumnCount(); i++)
                            System.out.print(String.format("%-
20.20s",rsmd.getColumnName(i)));
                        System.out.println();

 System.out.println("----------------------------------------");
```

```
                        while (rs.next()) {
                            for (int i = 1; i <= rsmd.getColumnCount(); i++)
                                System.out.print(String.format("%-20.20s
",rs.getString(i)));
                            System.out.println();
                        }
                        rs.close();
                    }
                }
                taula.close();
            }
            System.out.println();
            System.out.println("Introdueix un número per veure el contingut de
la taula (0 per acabar): ");
            opcio = Integer.parseInt(ent.readLine());
        }
        ll.close();
        con.close();
    }
}
```

6.2 - ResultSets que poden avançar cap avant i cap arrere

Fins el moment, tots els **ResultSet** que hem creat podien avançar únicament cap avant, fent un recorregut seqüencial de les dades.

- Inicialment el punter que apunta a l'estructura està situat abans de la primera fila
- En fer **next()** se situa a la següent fila (la primera vegada se situa en la primera fila)
- Quan estiguem situats en l'última fima, si fem next() se situara després de l'última, i ho indicarà tornant **false**

Però aquesta manera de funcionar, que es diu **TYPE_FORWARD_ONLY** , que és l'opció per defecte per a obrir un ResultSet, no és l'única. Aquestos són els tipus de ResultSet que hi ha:

- **TYPE_FORWARD_ONLY**: és el tipus utilitzat fins el moment, i és el tipus per defecte. El ResultSet només pot avançar cap avant.
- **TYPE_SCROLL_INSENSITIVE**: El ResultSet pot avançar cap avant i cap arrere, fins i tot pot anar a una posició absoluta (directament, sense passar per les anteriors). El ResultSet no és sensible als canvis fets en les dades que han proporcionat aquest resultat, és a dir, ens mostra les dades que hi havia en el moment d'executar la consulta, sense poder mostrar possibles canvis posteriors.
- **TYPE_SCROLL_SENSITIVE**: Igual que l'anterior en quant al moviment. Però ara sí que és capaç de mostrar possibles canvies fets en les dades originals posteriors al moment de l'execució de la consulta, mentre el ResultSet estiga obert.

I per una altra banda el **ResultSet** pot tenir la possibilitat d'**actualitzar** les dades originals. Des d'aquest punt de vista hi ha 2 possibles tipus:

- **CONCUR_READ_ONLY**: El ResultSet només és de lectura i no es pot actualitzar
- **CONCUR_UPDATABLE**: El ResultSet es pot actualitzar (i els canvis es reflectiran en les dades originals) utilitzant el mateix ResultSet

L'opció per defecte és **CONCUR_READ_ONLY**

Realment on declararem els tipus és en la creació del **STATEMENT** a partir del qual crearem la sentència que omplirà el **ResultSet**. És lògic, ja que aquestes maneres de funcionar s'han de preparar abans d'executar la sentència. Fins ara no havíem posat paràmetres en la creació del **Statement**, cosa que fa que es cree amb les opcions per defecte. Si volem altres opcions:

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

```
Statement st =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
```

La primera qüestió que hem de comentar és que el SGBD ha de ser capaç de suportar els tipus que no són per defecte, i no sempre és així. Mentre que **PostgreSQL**, **Oracle** i **MySQL** sí que són capaços de suportar tots els tipus anteriors, **SQLite** no ho pot fer i tan sols suporta **TYPE_FORWARD_ONLY** i **CONCUR_READ_ONLY**, com era d'esperar.

Deixant de banda la possibilitat de reflectir els possibles canvis de les dades originals (**SENSITIVE**), i de poder actualitzar-les (**CONCUR_UPDATABLE**), anem a estudiar els possibles moviments dins del **ResultSet**.

- **next**: Avança una fila cap avant. Torna *true* si s'ha pogut posicionar, i *false* si no s'ha pogut posicionar, per estar després de l'última fila.
- **previous**: Avança una fila cap arrere. Torna *true* si s'ha pogut posicionar, i *false* si no s'ha pogut posicionar, per estar abans de la primera fila.
- **first**: Se situa en la primera fila. Torna *true* si s'ha pogut posicionar, i *false* si no s'ha pogut posicionar, per no contenir el ResultSet cap fila.
- **last:**: Se situa en l'última fila. Torna *true* si s'ha pogut posicionar, i *false* si no s'ha pogut posicionar, per no contenir el ResultSet cap fila.
- **beforeFirst**: Se situa al principi del RecordSet, abans de la primera fila. Si no hi havia cap fila, no fa res.

- **afterLast**: Se situa al final del RecordSet, després de l'última fila. Si no hi havia cap fila, no fa res.
- **relative(int *files*)**: Meneja el cursor respecte de la posició actual, tantes files com s'indica en el paràmetre (si el paràmetre és negatiu, anirà cap arrere).
- **absolute(int *fila*)**: Situa el cursor en la fila especificada en el paràmetre (1 és la primera)

6.3 - Tractament d'errors en aplicacions JDBC

L'execució de sentències SQL està sotmesa a molts de factors que poden provocar algun error. Pot passar que la connexió falle, que el controlador no siga l'adequat, que les sentències tinguen errades, que el SGBD no suporte la sentència, i un llarg etcètera de possibilitats.

> **Nota**
> Podeu trobar informació referida als codis de **SQLSTATE** en la wikipedia, en aquest enllaç. El codi SQLSTATE està format per cinc caràcters. Els dos primers indiquen la tipologia de l'error i els tres últims el concreten.

Els errors SQL es troben molt ben definits a l'especificació estàndard, la qual descriu el valor de la variable anomenada **SQLSTATE**, que identifica l'estat d'una sentència SQL immediatament després de la seua execució. Quan JDBC detecta que després d'una execució el valor d'aquesta variable es correspon a un error, dispara una excepció de tipus **SQLException** la qual, a més de contenir un missatge clarificador, incorpora el valor del **SQLSATE**. Podem recuperar aquest valor amb el mètode **getSQLState()**.

L'ús de ***try-catch*** ens permetrà capturar específicament excepcions **SQLException** o derivades. Una vegada capturades, utilitzarem el codi SQLSTATE per decidir com cal actuar.

Un altre mètode molt útil és **getMessage()** que torna una cadena amb l'error produït. Pot servir perfectament per a la resta d'errors no tractats, ja que amb aquest missatge sempre donarem una pista, encara que no personalitzada com abans.

Imaginem, per exemple, que en intentar connectar amb un SGBD capturem una excepció SQL amb el valor **SQLState** igual a *28000*. Si consulteu aquest codi a la pàgina que us indiquem en la nota de dalt veureu que el valor *28000* correspon a un error en l'autenticació. En canvi, si el codi rebut haguera estat *08001* significaria que JDBC està trobant problemes de xarxa a l'hora de connectar, ja siguen deguts a una desconnexió física, o simplement a un *host* o adreça IP desconegut.

> **Nota**
> Tant PostgreSQL com Oracle defineixen millor l'error d'autenticació. En el cas de PostgreSQL és el 28P01, i en el cas d'Oracle el 72000. Per tant hauríem de substituir per aquestos valors en el programa posterior si volem connectar a ells.La taula d'errors de PostgreSQL la podeu trobar en aquest enllaç. Mentre que la d'Oracle en aquest altre enllaç.

No cal informar detalladament l'usuari de tots i cadascun dels possibles errors, però sí que cal decidir quins errors requeriran un tractament específic i quins no. Segurament no seria mala idea, si detectem un **SQLState** de valor *08001*, aconsellar l'usuari que abans de trucar al servei tècnic revise les connexions de xarxa o s'assegure que el SGBD es troba en marxa.

D'altra banda, la detecció precisa del **SQLState** ens pot també permetre realitzar accions per reconduir l'error. Imaginem, per exemple, que per raons de seguretat l'administrador del SGBD va canviant de contrasenya. L'administrador tria una contrasenya a l'atzar d'entre un conjunt de tres o quatre prefixades. Per tal de no haver d'estar contínuament configurant la nostra aplicació cada vegada que canvie la contrasenya, podem implementar una utilitat que accepte un conjunt de tres o quatre contrasenyes de manera que puga anar provant d'una en una quan reba un error d'autenticació.

Per a la resta d'errors, podem avisar a l'usuari de l'error que s'ha produït, o podem utilitzar una altra tècnica, que és utilitzar enregistradors. Els enregistradors (*loggers*) van guardant automàticament en un fitxer les coses que van succeint.

Vegem un possible exemple on posem en pràctica totes les consideracions que acabem de comentar. Està fet sobre **MySQL**, ja que com hem comentat abans, PostgreSQL utilitza ara un altre codi d'error per a la contrasenya invàlida :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class tractamentErrorConnexio {

    public static void main(String[] args) {
        boolean connectat = false;
        Connection con = null;
        System.out.println("tractamentErrorConnexio()");

        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:mysql://89.36.214.106:3306/geo";

            String usuari = "geo";
            String [] contrasenyes = {"geo0","geo1","geo"};

            for (int i = 0; !connectat && i < contrasenyes.length; i++) {
                try {
                    con = DriverManager.getConnection(url, usuari,
contrasenyes[i]);
                    connectat = true;
                } catch (SQLException ex) {
                    if (!ex.getSQLState().equals("28000")) {
                        // NO és un error d'autenticació
                        throw ex;
                    }
                }
            }
            if (connectat)
                System.out.println("Connexió efectuada correctament");
            else
                System.out.println("Error en la contrasenya");
        } catch (SQLException ex) {
            if (ex.getSQLState().equals("08001")) {
                System.out.println("S'ha detectat un problema de connexió.
Reviseu els cables de xarxa i assegureu-vos que el SGBD està operatiu."
                                + " Si continua sense connectar, aviseu el
servei tècnic");

            } else {
                System.out.println("S'ha produït un error inesperat. Truqueu
al servei tècnic indicant el següent codi d'error SQL:"
                                + ex.getSQLState());
            }
        } catch (ClassNotFoundException ex) {
            System.out.println("No s'ha trobat el controlador JDBC ("
                    + ex.getMessage() + "). Truqueu al servei tècnic");
        } finally {
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException ex) {

Logger.getLogger(tractamentErrorConnexio.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
    }
}
```

6.4 - Transaccions

Una transacció és un conjunt de sentències SQL d'actualització (INSERT, DELETE, UPDATE) que o bé s'executen totes o bé no s'executa ninguna.

La manera de confirmar les sentències és amb **COMMIT**, i la manera de rebutjar-les totes és **ROLLBACK** (quedant l'estat com estava abans de començar la transacció). En qualsevol dels dos casos, després d'executar les sedntències de control **COMMIT** o **ROLLBACK**, començarà una transacció nova.

JDBC trasllada també aquest metodologia al seu API. Per defecte, les connexions JDBC consideren que cada objecte **Statement** és en si mateix una transacció. Abans de cada execució es demana l'inici d'una transacció i al final, si l'execució té èxit, s'envia un **commit** i si no té èxit, un **rollback**. Per això diem que la connexió actua en mode **autocommit**.

Però hi ha una altra manera de funcionar. Els **Statements** poden treballar sense automatitzar el **commit** després de cada execució. Caldrà canviar la connexió de mode. Per canviar-la, executarem el mètode **setAutoCommit(false)**.

A partir d'aleshores es consideraran instruccions d'una mateixa transacció totes les sentències executades entre dues cridades del mètode **commit** o **rollback** (equivalents JDBC de les instruccions **COMMIT** i **ROLLBACK** de SQL).

A continuació mostrem una implementació genèrica que espera per paràmetre un *array* de sentències SQL que pertanyen a una mateixa transacció. La funció implementada executa sentència per sentència havent prèviament desactivat el mode **autocommit**. Si totes les sentències s'executen amb èxit es cridarà al mètode **commit**; en cas contrari es cridarà al mètode **rollback**. A la implementació hem suposat que la connexió ja existeix i es troba activa.

Aquest exemple seria part d'un programa més extens. No el podreu executar, només es mostra a nivell il·lustratiu.

```
public void executa(String[] sentenciesSql) throws SQLException{
   boolean autocommit=true;
   Statement stm = null;
   try {
      autocommit = con.getAutoCommit();
      con.setAutoCommit(false);
      stm = con.createStatement();
      for(String sent: sentenciesSql){
         stm.executeUpdate(sent);
      }
      con.commit();
      con.setAutoCommit(autocommit);
   } catch (SQLException ex) {
      con.rollback();
      throw ex;
   }finally{
      try {
         if(stm!=null && !stm.isClosed()){
            stm.close();
         }
      } catch (SQLException ex) {
         Logger.getLogger(ProvesBasiques.class.getName()).log(Level.SEVERE,
         null, ex);
      }
   }
}
```

En tractar-se d'una connexió ja existent que s'utilitza en diversos mètodes, desactivarem sempre el mode **autocommit** a l'inici, però guardant-nos l'estat previ en la variable **autocommit**. Al final restituirem aquest valor guardat.

La major part de SGBD permeten utilitzar transaccions explícites amb qualsevol instrucció SQL, fins i tot en sentències DDL (*data definition language*) com CREATE TABLE, etc . Les sentències de definició modifiquen directament l'estructura de les dades i, per tant, cal anar molt en compte perquè poden provocar danys importants, pèrdues de dades existents, etc.

Però hi ha alguns SGBD com Oracle que no suporten la revocació de sentències DDL i en cas d'executar **rollback**, obtindrem un error indicant que les sentències DDL no es poden revocar. Per preveure casos com aquest caldrà capturar l'error i avisar l'usuari que no s'ha pogut restaurar la base de dades i que caldria revisar-la manualment.

6.5 - Millora de rendiment

Un altre aspecte important que mesura la qualitat de les aplicacions és l'eficiència amb la qual s'aconsegueix comunicar amb el SGBD. Per optimitzar la connexió és important reconèixer quins processos poden actuar de coll d'ampolla.

En primer lloc, analitzarem la petició de connexió a un SGBD perquè es tracta d'un procés costós però inevitable que cal considerar.

En segon lloc, estudiarem les sentències predefinides, perquè el seu ús facilita la creació de *dades clau* i índexs temporals de manera que siga possible anticipar-se a la demanda o disposar de les dades de forma molt més ràpida.

Temps de vida d'una connexió

L'establiment d'una connexió és un procediment molt lent, tant a la part client com a la part servidor.

- A la part client, **DriverManager** ha de descobrir el controlador correcte d'entre tots els que haja de gestionar. La majoria de vegades les aplicacions treballaran només amb un únic controlador, però cal tenir en compte que **DriverManager** no coneix a priori quina URL de connexió correspon a cada controlador, i per saber-ho envia una petició de connexió a cada controlador que tinga registrat, el controlador que no li retorna error serà el correcte.
- A la banda servidor, es crearà un context específic i s'habilitaran un conjunt de recursos per cada client connectat. És a dir, que durant la petició de connexió el SGDB ha de gastar un temps considerable abans de deixar operativa la comunicació client-servidor.

Aquesta elevat cost de temps concentrat en el moment de la petició de connexió fa que ens plantegem si val la pena obrir i tancar la connexió cada vegada que ens toque executar una sentència SQL, o obrir una connexió al principi de l'aplicació que tancaríem en finalitzar. Lamentablement no hi ha una única resposta, sinó que depèn de la freqüència d'ús de la connexió i el número de connexions contra el mateix SGBD.

Com en tot, es tracta de trobar el punt d'equilibri. Si el número de clients, i per tant de connexions, és baix i la freqüència d'ús és alta, serà preferible mantenir les connexions obertes molt de temps. Per contra, si el número de connexions és molt alt i la freqüència d'ús baixa, el que serà preferible serà obrir i tancar la connexió cada vegada que es necessite. I també hi haurà una multitud de casos en què la solució consistirà a mantenir les connexions obertes però no permanentment. Es pot donar un temps de vida a cada connexió, o bé tancar-les després de restar inactiva una quantitat determinada de temps, o es pot fer servir el criteri de mantenir un número màxim de connexions obertes, tancant les més antigues o les més inactives quan se sobrepasse el límit.

Sentències predefinides

JDBC disposa d'un objecte derivat del **Statement** que s'anomena **PreparedStatement**., a la qual se li passa la sentència SQL en el moment de crear-lo, no en el moment d'executar la sentència (com passava amb **Statement**). I a més aquesta sentència pot admetre paràmetres, cosa que ens pot anar molt bé en determinades ocasions.

Siga com siga, **PreparedStatement** presenta avantatges sobre el seu antecessor **Statement** quan ens toque treballar amb sentències que s'hagen d'executar diverses vegades. La raó és que qualsevol sentència SQL, quan s'envia al SGBD serà compilada abans de ser executada.

- Utilitzant un objecte **Statement**, cada vegada que fem una execució d'una sentència, ja siga via **executeUpdate** o bé via **executeQuery**, el SGBD la compilarà, ja que li arribarà en forma de cadena de caràcters.
- En canvi, al **PreparedStament** la sentència mai varia i per tant es pot compilar i guardar dins del mateix objecte, de manera que les següents vegades que s'execute no caldrà compilar-la. Això reduirà sensiblement el temps d'execució.

En alguns sistemes gestors, a més, fer servir **PreparedStatements** pot arribar a suposar més avantatges, ja que utilitzen la seqüència de bytes de la sentència per detectar si es tracta d'una sentència nova o ja s'ha servit amb anterioritat. D'aquesta manera es propicia que el sistema guarde les respostes en la memòria caché, de manera que

es puguen lliurar de forma més ràpida.

La principal diferència dels objectes **PreparedStatement** en relació als **Statement**, és que en els primers se'ls passa la sentència SQL predefinida en el moment de crear-lo. Com que la sentència queda predefinida, ni els mètodes **executeUpdate** ni **executeQuery** requeriran cap paràmetre. És a dir, justet al revès que en el **Statement**.

```
...
sentenciaSQL = "INSERT INTO PAIS VALUES('Marroc')";
PreparedStatement stm = con.prepareStatement(sentenciaSql);
stm.executeUpdate();
...
```

Els **paràmetres** de la sentència es marcaran amb el símbol d'interrogant (?) i s'identificaran per la posició que ocupen a la sentència, començant a comptar des de l'esquerra i a partir del número 1. El valor dels paràmetres s'assignarà fent servir el mètode específic, d'acord amb el tipus de dades a assignar. El nom començarà per **set** i continuarà amb el nom del tipus de dades (exemples: **setString**, **setInt**, **setLong**, **setBoolean**…). Tots aquestos mètodes segueixen la mateixa sintaxi:

```
setXXXX(<posicioALaSentenciaSQL>, <valor>);
```

Veiem un exemple, en el qual suposem que tenim les dades a introduir en un objecte **article**:

```
...
PreparedStatement st = con.prepareStatement("INSERT INTO ARTICLE (id,
descripcio) VALUES(?, ?)");
st.setLong(1, article.getId());
st.setString(2, article.getDescripcio());
st.executeUpdate();
...
```

Exercicis

Els següents exercicis, posa'ls tots en un paquet del projecte **Tema4** anomenat **Exercicis**

Exercici 4_1

Crea una classe executable (amb main) anomenada **CreaTaulesRuta** que cree les taules necessàries per a guardar les dades de les rutes en una Base de Dades SQLite anomenada **Rutes.sqlite**.

Han de ser 2 taules:

- **RUTES**: que contindrà tota la informació del conjunt de la ruta. La clau principal s'anomenarà **num_r** (entera). També guardarà el nom de la ruta (**nom_r**), desnivell (**desn**) i desnivell acumulat (**desn_ac**). Els tipus d'aquestos tres camps últims seran de text, enter i enter respectivament.
- **PUNTS**: que contindrà la informació dels punts individuals de les rutes. Contindrà els camps **num_r** (número de ruta: enter) , **num_p** (número de punt: enter), **nom_p** (nom del punt: text) , **latitud** (número real) i **longitud** (número real). La clau principal serà la combinació **num_r + num_p** . Tindrà una **clau externa** (**num_r**) que apuntarà a la clau principal de **RUTES**.

Adjunta tot el projecte, i també la Base de Dades **Rutes.sqlite** (normalment estarà dins del projecte)

Exercici 4_2

Crea una altra classe executable anomenada **PassarRutesObjSQLite** que passe les dades del fitxer **Rutes.obj** a les taules de **Rutes.sqlite**.

Per a major comoditat copiat el fitxer **Rutes.obj** que vam fer al projecte **Tema3** dins d'aquest projecte, a més de les classes **Coordenades.java**, **PuntGeo.java** i **Ruta.java** dins del paquet. Si el paquet s'anomena igual en els dos projectes (jo us havia suggerit **Exercicis**) no hauria d'haver cap problema.

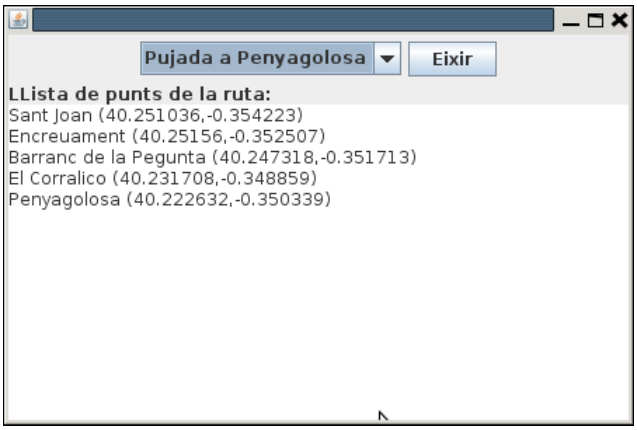
Concretament en el trasvassament d'informació:

- Per a cada **ruta** s'ha de considerar el **número de ruta**, que s'anirà incrementant, i a més serà la **clau principal**. La primera ruta ha de ser la número 1. A banda han d'anar la resta de camps.
- Per a cada **punt** s'ha de guardar el **número de ruta** i el **número de punt** (s'aniràincrementant des de 1 per a cada ruta). A banda han d'anar la resta de camps.
- T'aconselle vivament que abans d'executar les sentències SQL d'inserció, les tragues per pantalla, per veure si la sintaxi és correcta. Quan totes siguen correctes, pots substituir l'eixida per l'execució de les sentències.
- Per últim intenta capturar els errors per a traure missatges comprensibles.

Adjunta tot el projecte, i també la Base de Dades **Rutes.sqlite**, que normalment estarà situada dins del projecte

Exercici 4_3

Crea un classe anomenada **Vis_Rutes_SQLite**, que mostre les rutes amb un aspecte similar a l'**Exercici 3_4**, però accedint ara a les dades guardades en **Rutes.sqlite**:



Podríem mantenir la connexió fins que s'aprete el botó d'**Eixir**, que és quan la tancaríem.

Ací teniu l'esquelet dels programes.

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.util.ArrayList;

import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;


public class Vis_Rutes_SQLite_Pantalla extends JFrame implements
ActionListener{

    JComboBox combo;
    JButton eixir = new JButton("Eixir");
    JTextArea area = new JTextArea();
    Connection con;

    public void iniciar(){
        // sentències per a fer la connexió


        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());

        JPanel panell1 = new JPanel(new FlowLayout());
        JPanel panell2 = new JPanel(new BorderLayout());
        this.add(panell1,BorderLayout.NORTH);
        this.add(panell2,BorderLayout.CENTER);

        ArrayList<String> llista_rutes = new ArrayList<String>();
        // sentències per a omplir l'ArrayList amb el nom de les rutes


        combo = new JComboBox(llista_rutes.toArray());

        panell1.add(combo);
        panell1.add(eixir);

        panell2.add(new JLabel("LLista de punts de la
ruta:"),BorderLayout.NORTH);
        panell2.add(area,BorderLayout.CENTER);

        this.setVisible(true);
        combo.addActionListener(this);
        eixir.addActionListener(this);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == combo){
```

```
            //accions quan s'ha seleccionat un element del combobox, i
que han de consistir en omplir el JTextArea

        }

        if (e.getSource() == eixir){
            //accions quan s'ha apretat el botó d'eixir

        }
    }
}
```

```
public class Vis_Rutes_SQLite {

    public static void main(String[] args) {
        Vis_Rutes_SQLite_Pantalla finestra = new
Vis_Rutes_SQLite_Pantalla();
        finestra.iniciar();
    }
}
```

Exercici 4_4

En aquest exercici anem a practicar la creació d'una classe que encapsule pràcticament tot el tractament de la Base de Dades, per a fer transparent el desfasament Objecte-Relacional. Per tant aquesta classe ha de ser capaç de llegir de les taules convertint a objectes, i també guardar la informació dels objectes en les taules. S'anomenarà **GestionarRutesBD.**

En un paquet nou del mateix projecte, anomenat **util.bd** haureu de fer la classe **GestionarRutesBD**, que és la que encapsularà tot. Internament només tindrà una propietat, la de connexió. Tindrà els següents mètodes:

- Constructor **public GestionarRutesBD()**: establirà la connexió amb la Base de Dades **Rutes.sqlite** (per comoditat en el directori del projecte). Si no existeixen les taules **RUTES** i **PUNTS** les haurà de crear (podeu utilitzar la sentència **CREATE TABLE IF NOT EXISTS** ...)
- **public void close()**: tancarà la connexió.
- **public void inserir(Ruta r)**: Inserirà en la BD les dades corresponents a la ruta passada per paràmetre (inicialment s'aconsella únicament "imprimir" les sentències, per veure si són correctes). El **num_r** ha de ser el posterior a l'última existent, per exemple amb la consulta **SELECT MAX(num_r) FROM RUTES**
- **public Ruta buscar(int i)**: torna la ruta amb el número passat com a paràmetre.
- **public ArrayList<Ruta> llistat():** torna un ArrayList de Ruta amb totes les rutes de la Base de Dades.
- **public void esborrar(int i)**: esborra la ruta amb el número passat com a paràmetre (recordeu que els punts de la ruta també s'han d'esborrar)

Per a provar-lo podeu incorporar l'executable (que té main) **UtilitzarRutesBD**. Hauríeu de comentar les línies que no s'utilitzen en cada moment. Per exemple, si ja heu aconseguit inserir la ruta, i continueu provant el programa per als mètodes **buscar()** i **llistat()**, comenteu la línia **gRutes.inserir(r);** per a no inserir-la més vegades.

```
public class UtilitzarRutesBD {

    public static void main(String[] args) throws
FileNotFoundException, IOException, ClassNotFoundException {
        // Creació del gestionador
        GestionarRutesBD gRutes = new GestionarRutesBD();


        // Inserció d'una nova Ruta
        String[] noms = {"Les Useres","Les Torrocelles","Lloma
Bernat","Xodos (Molí)","El Marinet","Sant Joan"};
        double[] latituds =
{40.158126,40.196046,40.219210,40.248003,40.250977,40.251221};
        double[] longituds = {-0.166962,-0.227611,-0.263560,-0.296690,-
0.316947,-0.354052};

        Ruta r = new Ruta();
        r.setNom("Pelegrins de Les Useres");
        r.setDesnivell(896);
```

```
        r.setDesnivellAcumulat(1738);
        for (int i=0;i<6;i++){
            r.addPunt(noms[i], latituds[i], longituds[i]);
        }

        gRutes.inserir(r);

        // Llistat de totes les rutes
        ArrayList<Ruta> llista = gRutes.llistat();
        for (int i=0;i<llista.size();i++){
            llista.get(i).mostraRuta();
        }

        // Buscar una ruta determinada
        gRutes.buscar(2).mostraRuta();

        gRutes.close();
    }
}
```

Exercici 4_5 (voluntari)

Crea el següent mètode en la classe **GestionarRutesBD**:

public void guardar(Ruta r)

El que ha de fer aquest mètode és:

- Si no existeix la ruta, la inserirem
- Si ja existeix la ruta, la modificarem. Heu de parar especial atenció als punts. Potser el més còmode siga esborrar els punts de la ruta i tornar a crear-los

Considerarem que la ruta existeix si hi ha una amb el mateix nom.

Per a provar-lo, podeu utilitzar aquest programa principal.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;

import util.bd.GestionarRutesBD;
import util.bd.Ruta;
import util.bd.PuntGeo;

public class UtilitzarRutesBD2{

    public static void main(String[] args) throws
FileNotFoundException, IOException, ClassNotFoundException {
        // Creació del gestionador
        GestionarRutesBD gRutes = new GestionarRutesBD();

        Ruta r = gRutes.buscar(1);
        r.mostraRuta();
        r.setDesnivellAcumulat(606);
        gRutes.guardar(r);

        r = gRutes.buscar(2);
        r.mostraRuta();
        r.getLlistaDePunts().add(0, new PuntGeo ("Plaça M.Agustina",
39.988507,-0.034533));
        gRutes.guardar(r);

        System.out.println("Després de modificar");
        r=gRutes.buscar(1);
        r.mostraRuta();
        r=gRutes.buscar(2);
        r.mostraRuta();

        gRutes.close();
    }
}
```