BASES DE DATOS
APUNTES ASIGNATURA – Curso 2021 - 2022
Juan Diego Sanz Guevara

6.5.2. Uso de consultas pre-compiladas

6.6. INSERCIÓN DE DATOS Y MODULARIDAD

6.6.1. Manejo de conexión en un programa

6.6.2. Ejecución de sentencias DDL desde JDBC

6.6.3. Inserción de datos usando JDBC

1. INTRODUCCIÓN A LAS BASES DE DATOS

1.1. INTRODUCCIÓN E HISTORIA DE LAS BASES DE DATOS

Las bases de datos se pueden de definir de distintas formas, entre ellas:

- Un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.
- Una colección de datos que contiene información relevante para una empresa.
- Colección de datos agrupados y soportados en algún medio físico, con una organización en la que figuran no solamente los datos, sino también las relaciones entre los mismos.

Asociado a una base de datos, existe siempre un **SGBD** (**S**istema de **G**estión de **B**ases de **D**atos), que significa:

- Colección de programas que permite a los usuarios crear y mantener una BD.
- Software de propósito general que facilita los procesos de definición, construcción, manipulación y compartición de bases de datos entre varios usuarios y aplicaciones.

Entre las principales características de los sistemas de base de datos podemos mencionar:

- Independencia lógica y física de los datos
- Redundancia mínima
- Acceso concurrente por parte de múltiples usuarios
- Integridad de los datos
- Consultas complejas optimizadas
- Seguridad de acceso y auditoría
- Respaldo y recuperación
- Acceso a través de lenguajes de programación estándar

En cuanto a la historia, las bases de datos surgen de la necesidad de conservar la información mñas allá de los que existe en la memoria RAM.

Las bases de datos basadas en archivos eran datos guardados en texto plano, fáciles de guardar pero muy difíciles de consultar y por su necesidad de mejora nacen las bases de datos relacionales. Su inventor, **Edgar Codd**, dejó ciertas reglas para asegurarse de que toda la filosofía de las bases de datos no se perdiera, estandarizando el proceso, Codd **inventó el algebra relacional.** (Consultar 12 reglas y mandamientos de Edgar Codd).

Aquí van algunos años importantes en el mundo de las bases de datos:

- **1967 - 1969:** La CODASYL hace la primera especificación para una base de datos en red.
    - Aparecen el DDL y el DML.
- **1970**: Edgar Codd (IBM Research Laboratory) propone el **Modelo Relacional.**
- **1976:** Peter Chen define el Modelo Entidad – Relación
- **1979:** Surge Oracle Database.
- **1986:** Estandarización de SQL.
- **1980's:** Bases de datos orientadas a objetos.

- **1990's:** Aparecen múltiples SGBD (Access, MySQL, Postgres, …)
- **2000's:** Bases de datos NoSQL.

1.2. ARQUITECTURA ANSI/SPARC

¿Cómo se organiza una BD?

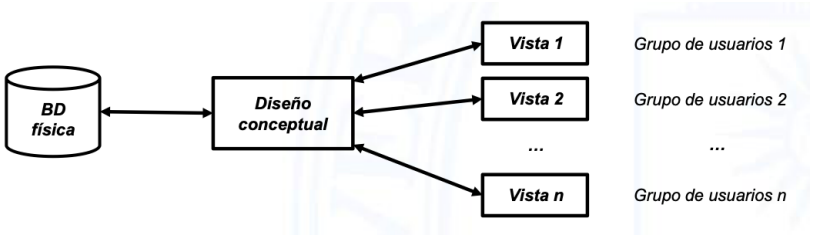
Imagina una oficina de correos, ¿Cómo interacciona la persona que recoge el paquete?, ¿cómo se organiza la búsqueda de los paquetes?, ¿cómo se organiza la ubicación física de los paquetes?

Uno de los objetivos más importantes del SGBD es proporcionar una versión abstracta de los datos (esconder detalles de cómo se almacenan y mantienen).

La arquitectura ANSI/SPARC es un estándar de diseño abstracto para un SGBD (que no es un estándar formal)

La arquitectura se organiza en tes niveles:

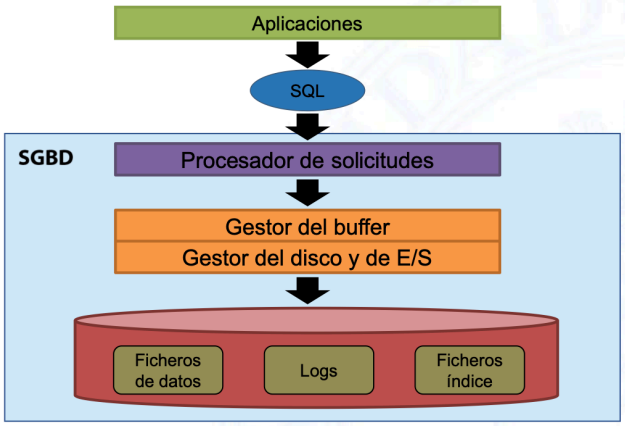
- **Físico o interno**, que describe cómo se almacenan realmente lo datos
    - Es el más cercano a la máquina
    - Dependiente del gestor de bases que se use
    - Define la estructura física de almacenamiento de toda la BD.
    - Tipos de registro almacenados
    - Secuencia física de los registros
    - Estructuras de almacenamiento y de acceso
- **Conceptual**, que describe los datos y sus relaciones
    - Da soporte a la independencia entre datos y aplicaciones
    - Oculta detalles físicos
    - Visión completa de todos los requerimientos y elementos de interés para la organización
    - Incluye restricciones semánticas sobre los datos
    - Define la estructura lógica de toda la BD: Entidades, tipos de datos, relaciones, restricciones…
- **Vista de usuario o externo,** permite al usuario visualizar los datos que le son relevantes y restringe qué datos son autorizados para su acceso (seguridad).
    - El más cercano a los usuarios finales
    - Descrito mediante esquemas externos
    - Porción de la BD a la que el usuario accede
    - Pueden existir múltiples vistas del mismo esquema conceptual



Un SGBD debe proporcionar:

- Lenguaje de definición de datos (DDL, Data Definition Language), que permite describir el modelo conceptual como modelo de datos y es la traducción del modelo ER a tablas.
- Lenguaje de manipulación de datos (DML, Data Manipulation Language), que permite recuperar y actualizar un conjunto de registros.
- Procedimiento d administración, a veces llamado lenguaje de control de datos (DCL, Data Control Language)

La arquitectura de un Sistema de Gestión de Bases de Datos es como la siguiente:



2. MODELO RELACIONAL

Los datos son **números** y **textos** concretos que pueden ser encontrados en un programa, mientras que **información** no es más que el conocimiento extraído de esos datos (se requiere de experiencia experta para ser generados). Los datos son inalterables, mientras que la información de estos puede variar con el tiempo.

Los datos deben ser:

- Accesibles
- Almacenados de forma segura
- Accesibles en distintos niveles (un usuario no puede acceder a los mismos datos que un administrador, esto es **seguridad**).

El proceso para diseñar una base de datos está basado en cuatro pasos:

Paso 1: Análisis de requisitos: Especificación experta de las necesidades de nuestras bases de datos.

Paso 2: Diseño conceptual: Diagrama E – R (Entidad – Relación).

Paso 3: Diseño Lógico: Modelo Relacional.

Paso 4: Modelo físico: Implementación de la base de datos.

2.1. MODELO RELACIONAL

La **relación** es la estructura fundamental del modelo de relación. Esta relación representa las **entidades** (algo similar a un objeto en programación orientada a objetos, que representa algo en el mundo real o incluso algo abstracto) y **atributos** (que son las características que los hacen ser una entidad) de nuestro dominio/ámbito de trabajo. A su vez encontramos dos tipos de relaciones: la **básica** y la **derivada**.

- Las relaciones básicas son aquellas que almacenan datos y cuya implementación se llama tabla (como R o S)
    - Una tabla bidimensional define un conjunto de columnas o **atributos**. Está compuesta por unas filas llamadas "registros" o "tuplas" (en otras palabras una tabla bidimensional es una matriz).
- Las relaciones derivadas pueden ser o bien "consultas" o bien "vistas". No son más que combinaciones de otras relaciones usando operadores. (Las obtenidas de un operador, como el resultado de hacer la unión interna entre dos relaciones).

2.1.1. Propiedades de las relaciones y atributos

- Relaciones (tablas):
    - Sean $A = \{a, b, c\}$ y $B = \{1, 2\}$ dos conjuntos de valores, una relación R de orden n definida sobre A y B, es un subconjunto del producto cartesiano A x B.
    - Al estar formada por el producto cartesiano de 2 elementos, se dice que la relación es de **grado u orden 2**
    - Ejemplo de relaciones de orden 2 sobre A y B
        - $R_1 = \{(a, 1), (a, 2), (c, 2)\}$
        - $R_2 = \{(a, 1), (b, 1)\}$

    - Cada relación tiene un nombre único en una base de datos. (Sólo podemos tener una relación "R" por ejemplo).
    - En una relación no puede a ver dos o más atributos con el mismo nombre. (En nuestra relación "R" no podemos tener un atributo "A" dos o más veces).
    - El orden de los atributos no es relevante.
    - El cardinal de la relación es el número de tuplas de dicha relación.
- Atributos:
    - En el ejemplo anterior, A y b son atributos de R_1 y R_2
    - Atributos distintos pueden tener mismos dominios
    - El conjunto de atributos de un problema es su **universo** T, que es siempre finito.
- Tuplas
    - No hay tuplas duplicadas en una tabla.
    - El orden de las tuplas no es relevante.

2.1.2. Dominios

Un dominio es un conjunto de valores que un atributo puede tener. Algunos dominio básicos dentro del modelo relacional son:

- Integer
- Real
- Date
- Datetime
- Time
- Character
- String

En un dominio encontramos un valor especial el cual, o es desconocido o su valor es "no valor". A este valor especial se le conoce como "NULL". Hay que tener en cuenta que:

- NULL no es 0
- NULL no es String vacío

Es frecuente usar tablas para representar a las relaciones de forma visual:

- El número de columnas es el grado de la relación
- El número de filas es su cardinalidad
- Cada columna está unida al nombre de su atributo

Ejemplos:

- Dom(A) = {a, b, c}, Dom(B) = $\mathbb{N}$, Dom(C) = {x y}
- R = {(a, 5, y), (c, 10, x), (b, 1, y)}
- La relación R se representa así:

A	B	C
a	5	y
c	10	X
b	1	y

2.1.3. Identificación de atributos

- **Clave candidata:** El conjunto de atributos de una relación que puede identificar unívocamente una tupla.
- **Clave primaria:** La clave candidata seleccionada como identificador de la tabla ()esta clave no puede ser nula).
- **Clave sustituta:** Otras claves candidatas distintas a las claves primarias.
- **Clave extranjera:** Un conjunto de atributos que sirve como clave primaria de otra tabla. (no deben ser nulas).

2.2. ÁLGEBRA RELACIONAL

Existen métodos y operaciones que nos permiten crear nuevas relaciones en base a antiguas relaciones existentes. El Álgebra relacional no es más que una notación matemática que nos permite anotar consultas usando operadores especiales sobre las relaciones. Hay dos tipos de operadores en el Álgebra relacional:

- **Operadores básicos o primitivos**
  - Proyección: π
  - Selección: σ
  - Unión: ∪
  - Diferencia: -
  - Producto cartesiano: X
  - Asignación: :=
  - Renombrado: ρ
- **Operadores extendidos o derivados**
  - Natural Join: ⊳⊲
  - Join condicional: $\rhd\lhd_{cond}$
  - División: ÷
  - Intersección: ∩
  - Full outer join: ><
  - Left outer join: >⊲
  - Right outer join: ⊳<

2.2.1. Operadores básicos

- **Proyección (π):** La "proyección" es usada para proyectar los datos de columna requeridos de una relación. Por defecto, la proyección elimina los datos duplicados. Un ejemplo es:

Estudiante

Nombre	Ciudad
Pepe	Pozuelo
María	Alcorcón

Localidad

Ciudad	Provincia
Pozuelo	Madrid
Alcorcón	Madrid

Calificación

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$\pi_{Provincia}(Localidad)$

Provincia
Madrid

- **Seleccionador (σ):** La "selección" es usada para seleccionar las tuplas requeridas de una relación. El operador de selección sólo filtra las tuplas, no las muestra. Para seleccionar los datos se usa:

Estudiante

Nombre	Ciudad
Pepe	Pozuelo
María	Alcorcón

Localidad

Ciudad	Provincia
Pozuelo	Madrid
Alcorcón	Madrid

Calificación

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$\sigma_{Nota>5}(Calificación)$

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8

$\sigma_{Nota>5 \wedge Nombre="Pepe"}(Calificación)$

Nombre	Asignatura	Nota
Pepe	BD	7.5

- **Unión (**U**):** El operador unión actúa igual que la operación unión en la teoría de conjuntos (la unión de dos conjuntos es igual a la creación de un tercer conjunto que contenga tanto los elementos del conjunto R como del conjunto S (si un elemento está presente en ambos conjuntos no se duplica)).
    - **Existe una restricción** para la unión de dos relaciones: Ambas relaciones deben tener el mismo conjunto de atributos

Calificación1

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8

Calificación2

Nombre	Asignatura	Nota
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$Calificación1 \cup Calificación2$

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

- **Diferencia (-):** R-S contendrá todas las tuplas de R que no se encuentren en el conjunto S. También es igual que la operación diferencia en teoría de conjuntos. **Restricción**: Al igual que

en la unión, ambas relaciones deben tener el mismo conjunto de atributos. Los conjuntos no permiten elementos duplicados.

Calificación1

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8

Calificación2

Nombre	Asignatura	Nota
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$Calificación1 - Calificación2$

Nombre	Asignatura	Nota
Pepe	BD	7.5

- **Producto cartesiano (X):** Es igual que el producto cartesiano en teoría de conjuntos. El producto cartesiano de dos relaciones R y S, R X S, es otra relación, definida sobre la unión de cabeceras de R y S cuyas tuplas de forman concatenando cada tupa de R con cada una de las de S en todas las formas posibles

Estudiante

Nombre	Ciudad
Pepe	Pozuelo
María	Alcorcón

Calificación

Nombre	Asignatura	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$Estudiante \times Calificación$

E.Nombre	Ciudad	C.Nombre	Asignatura	Nota
Pepe	Pozuelo	Pepe	BD	7.5
Pepe	Pozuelo	María	Ing. Soft.	5.8
Pepe	Pozuelo	Pepe	Ing. Soft.	5.0
María	Alcorcón	Pepe	BD	7.5
María	Alcorcón	María	Ing. Soft.	5.8
María	Alcorcón	Pepe	Ing. Soft.	5.0

- **Asignación (:=):** Asigna una relación que es resultado de operaciones algebraicas a una nueva relación.

$$NotasPepe := \sigma_{Nombre="Pepe"}(Calificación)$$

NotasPepe

Nombre	Asignatura	Nota
Pepe	BD	7.5
Pepe	Ing. Soft.	5.0

- **Renombrado (ρ):** Renombra una relación y/o sus atributos

$$\rho_{Notas(Estudiante,Curso,Nota)}(Calificación)$$

Notas

Estudiante	Curso	Nota
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

2.2.2. Operadores derivados

- **Natural join (⊳⊲):** Si dos relaciones R y S tienen un atributo común A, se realiza el producto cartesiano R x S y se seleccionan las filas con igual valor de A.
    - Equivalente : $\sigma_{R.A=S.A}$ (R x S).

Estudiante	
Nombre	**Ciudad**
Pepe	Pozuelo
María	Alcorcón
Lucía	Madrid

Calificación		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0
Juan	BD	9.4

Estudiante ⋈ Calificación

E.Nombre	Ciudad	C.Nombre	Asignatura	Nota
Pepe	Pozuelo	Pepe	BD	7.5
Pepe	Pozuelo	Pepe	Ing. Soft.	5.0
María	Alcorcón	María	Ing. Soft.	5.8

- **Join condicional (⊳⊲$_{cond}$):** Igual que el natural join, pero cambiando la condición sobre los atributos comunes por la especificada.

Calificación		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

NotaAlfabética	
Nota	**Descripción**
5.0	Aprobado
7.0	Notable
9.0	Sobresaliente

Calificación $\bowtie_{C.Nota \geq N.Nota}$ *NotaAlfabética*

Nombre	Asignatura	C.Nota	N.Nota	Descripción
Pepe	BD	7.5	5.0	Aprobado
Pepe	BD	7.5	7.0	Notable
María	Ing. Soft.	5.8	5.0	Aprobado
Pepe	Ing. Soft.	5.0	5.0	Aprobado

- **División (÷):** $R \div S = \{t \mid \forall t_s \in S, \exists t_r \in R \text{ tal que } t_t(A, B) = t \wedge t_r(C, D) = t_s\}$

Calificación			
Nombre	**Asignatura**	**Convocatoria**	**Nota**
Pepe	BD	1	6.2
Juan	Ing. Soft.	1	3.5
Juan	Ing. Soft.	2	5.0
Pepe	Concurrencia	1	5.0
María	Ing. Soft.	1	5.0
María	BD	1	3.5
María	BD	2	9.5

Primera	
Convocatoria	**Nota**
1	5.0

Segunda	
Convocatoria	**Nota**
1	3.5
2	5.0

Calificación ÷ Primera

Nombre	Asignatura
Pepe	Concurrencia
María	Ing. Soft.

Calificación ÷ Segunda

Nombre	Asignatura
Juan	Ing. Soft.

- **Intersección (∩):** Es igual que la intersección en teoría de conjuntos. Contiene las tuplas que estén en ambas relaciones

Calificación1		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8

Calificación2		
Nombre	**Asignatura**	**Nota**
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0

$Calificación1 \cap Calificación2$		
Nombre	**Asignatura**	**Nota**
María	Ing. Soft.	5.8

- **Full outer join (><):** Como el natural join, pero incluye los elementos que están en una sola de las relaciones (poniendo NULL sobre los elementos correspondientes de la otra relación)

Estudiante	
Nombre	**Ciudad**
Pepe	Pozuelo
María	Alcorcón
Lucía	Madrid

Calificación		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0
Juan	BD	9.4

$Estudiante \bowtie Calificación$				
E.Nombre	**Ciudad**	**C.Nombre**	**Asignatura**	**Nota**
Pepe	Pozuelo	Pepe	BD	7.5
Pepe	Pozuelo	Pepe	Ing. Soft.	5.0
María	Alcorcón	María	Ing. Soft.	5.8
Lucía	Madrid	NULL	NULL	NULL
NULL	NULL	Juan	BD	9.4

- **Left outer join (>⋉):** Como el natural join, pero incluye también los elementos que están en la relación de la izquierda (poniendo NULL en los elementos correspondientes de la otra relación)

Estudiante	
Nombre	**Ciudad**
Pepe	Pozuelo
María	Alcorcón
Lucía	Madrid

Calificación		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0
Juan	BD	9.4

$Estudiante \rtimes Calificación$				
E.Nombre	**Ciudad**	**C.Nombre**	**Asignatura**	**Nota**
Pepe	Pozuelo	Pepe	BD	7.5
Pepe	Pozuelo	Pepe	Ing. Soft.	5.0
María	Alcorcón	María	Ing. Soft.	5.8
Lucía	Madrid	NULL	NULL	NULL

- **Right outer join (⊳<):** Como el natural join, incluye también los elementos que están en la relación de la derecha (poniendo NULL en los elementos correspondientes de la otra relación)

Estudiante	
Nombre	**Ciudad**
Pepe	Pozuelo
María	Alcorcón
Lucía	Madrid

Calificación		
Nombre	**Asignatura**	**Nota**
Pepe	BD	7.5
María	Ing. Soft.	5.8
Pepe	Ing. Soft.	5.0
Juan	BD	9.4

$Estudiante \ltimes Calificación$				
E.Nombre	**Ciudad**	**C.Nombre**	**Asignatura**	**Nota**
Pepe	Pozuelo	Pepe	BD	7.5
Pepe	Pozuelo	Pepe	Ing. Soft.	5.0
María	Alcorcón	María	Ing. Soft.	5.8
NULL	NULL	Juan	BD	9.4

2.3. PRINCIPIOS DEL DISEÑO DE UNA BASE DE DATOS

2.3.1. Consistencia en los datos

Una base de datos será inconsistente si tiene el mismo dato asociado a valores de diferentes relaciones. (Un ejemplo de inconsistencia sería que un mismo estudiante tuviera dos fechas de cumpleaños distintas en dos relaciones distintas). Para evitar esta inconsistencia debemos minimizar la duplicidad de estos datos

2.3.2. Dependencia funcional

- Un descriptor X es un subconjunto de T
- Si X e Y son ambos descriptores en un cierto contexto y se tiene que dado el valor de X está unívocamente determinado el de Y:
    - Entre X e Y tiene lugar la **dependencia funcional** $X \rightarrow Y$
    - Se lee como "X implica Y" o bien, "Y depende funcionalmente de X".
- Dada una relación, se denomina **esquema** al par formado por el conjunto de atributos sobre el que está definida y el conjunto de dependencias entre ellos.
    - Es frecuente llamar **relación** tanto a una relación concreta como al esquema.
    - El par R(T, L), donde T es el **universo** de atributos, y L **conjunto de dependencias** entre ellos, se denomina **relación universal**
- Si $X \rightarrow Y$ y no existe ningún subconjunto propio $Z \subset X$ tal que $Z \rightarrow Y$, entonces la dependencia $X \rightarrow Y$ **es total.** En caso contrario, es parcial.
- Un ejemplo de Dependencia funcional es:

    Película (nombre_pelicula, año, director).

    Dependencia funcional: {nombre_pelicula} → {año, director}

    Con el subconjunto {nombre_pelicula} creamos una dependencia funcional con los atributos "año" y "director" de manera que "x" nombre de película, los atributos "año" y "director" serán siempre los mismos.
- Otro ejemplo de Dependencia funcional sería tener un atributo llamado "fecha_nacimiento" y tener otro llamado "signo_horoscopo". El signo está sujeto a la fecha de nacimiento, por tanto, decimos que hay dependencia funcional de fecha de nacimiento a Signo del horóscopo.
- Cuando $a \rightarrow b$ (a implica b) no necesariamente $b \rightarrow a$ (es decir, que sea recíproco).

    Persona (dni, nombre_completo, fecha_nacimiento, signo_horoscopo)

    {fecha_nacimiento} → {signo_horoscopo}

    La fecha de nacimiento implica el símbolo del horóscopo. En cambio, el símbolo del horóscopo no implica la fecha de nacimiento
- Cuando $a \rightarrow b$ y $b \rightarrow a$, decimos que existe una dependencia funcional mutua ($a \longleftrightarrow b$)

    {dni} → {nombre_completo, fecha_nacimiento}

    {nombre_completo, fecha_nacimiento} → {dni}

Si sabes el DNI, sabes el nombre completo y la fecha de nacimiento. Y lo mismo en viceversa.

- Se dice que "b" tiene una dependencia funcional completa de "a" cuando ($a \rightarrow b$) y b no es consecuencia de un subconjunto de "a". Para entender esto, tenemos el siguiente ejemplo:

  Persona (dni, nombre_completo, fecha_nacimiento, signo_horoscopo)

  {dni, fecha_nacimiento} → {signo_horoscopo}

  {fecha_nacimiento} → {signo_horoscopo}

  En este caso el conjunto {signo_horoscopo} no tiene dependencia funcional completa con el conjunto {dni, fecha_nacimiento}, puesto que "b" ({signo_horoscopo}) presenta dependencia funcional con el subconjunto {fecha_nacimiento} c {dni, fecha_nacimiento}

- Si "a" es un atributo simple y ($a \rightarrow b$), entonces existe una dependencia funcional completa de "a" a "b". Esto es fácil de entender puesto que, si "a" sólo tiene un atributo, entonces "b" no podrá presentar dependencia funcional respecto a un subconjunto de "a".

2.3.3. Axiomas de Armstrong

- **Primarios:**
  - **Reflexividad:** Si $Y \subseteq X$, entonces, $X \rightarrow Y$. En particular, siempre ocurre que $X \rightarrow X$
  - **Aumentividad:** Si $X \rightarrow Y$, entonces $XZ \rightarrow YZ$, para cualquier otro descriptor Z
  - **Transitividad:** Si $X \rightarrow Y$ e $Y \rightarrow Z$, entonces $X \rightarrow Z$
- **Derivados:**
  - **Proyectividad:** Si $X \rightarrow YZ$, entonces $X \rightarrow Y$ y $X \rightarrow Z$
  - **Aditividad:** Si $X \rightarrow Y$ y $Z \rightarrow W$, entonces $XZ \rightarrow YW$
  - **Unión:** Si $X \rightarrow Y$ y $X \rightarrow Z$, entonces $X \rightarrow YZ$

2.4. REFINAMIENTO DEL MODDELO RELACIONAL

Para diseñar un modelo relacional necesitamos inferir/deducir un conjunto de T atributos de interés y L dependencias funcionales a través del concomimiento de un experto. Sabiendo esto, podemos definir el modelo relacional como R (T, L).

Para evitar la inserción y eliminación de una anomalía necesitamos refinar el modelo relacional, con el final de obtener un modelo normalizado.

2.4.1. Cierre de un conjunto de dependencias

Teniendo una R (T, L) con un subconjunto de atributos X perteneciente a los atributos T ($X \subset T$), definimos X^+ **(Cierre de X)** como un superconjunto de X ($X \subseteq X^+$) tal que $X \rightarrow X^+$ (X implica X^+)

El cierre es el máximo conjunto de atributos en T que pueden inferirse/depender de X. Para **calcular el cierre** usamos un proceso iterativo basado en los **Axiomas de Armstrong**. El proceso iterativo acabará cuando el resultado de la iteración n-1 sea el mismo de la iteración **n**.

Ejemplo:

Dado $T = \{A, B, C, D, E, F\}$

L = { $A \rightarrow BC$, (1)

$C \rightarrow D$, (2)

$AC \rightarrow E$, (3)

$AB \rightarrow C$, (4)

$F \rightarrow A\}$ (5)

$X = ABE$

Para calcular $X^+ = ABE^+$ hacemos los siguientes pasos:

1) $ABE \rightarrow \{A, B, E\}$ (Propiedad proyectiva)
2) $ABE \rightarrow \{A, B, C, E\}$ (Aplicando (1)). Esto puede tener dos explicaciones:
    a. Por la propiedad proyectiva sabemos que ABE -> A y una de las dependencias funcionales dice: $A \rightarrow BC$, la cual puede dividirse en $A \rightarrow B$ y en $A \rightarrow C$ (Propiedad proyectiva). Por transitividad afirmamos que $ABE \rightarrow C$. Por tanto, utilizando la propiedad de **aditividad**, $ABE \rightarrow \{A, B, C, E\}$.
    b. Por la propiedad de "**aditividad**" y usando la primera dependencia funcional ($A \rightarrow BC$) podemos concluir que $ABC + A \rightarrow \{A, B, E\} + \{B, C\}$ que simplificando obtenemos $ABC \rightarrow \{A, B, C, E\}$
3) $ABE \rightarrow \{A, B, C, D, E\}$ (Aplicando (2)).
    a. Empleando la transitividad en la segunda dependencia, deducimos que $ABE \rightarrow D$ y por aditividad concluimos que $ABE \rightarrow \{A, B, C, D, E\}$
4) No hay ninguna manera de poder incluir ningún atributo más a través de las dependencias funcionales (ya que el atributo F se encuentra a la izquierda del →). Por ende, concluimos la clausura de ABE, dando como resultado: $ABE^+ = \{A, B, C, D, E\}$

2.4.2. Propiedad Unión sin perdidas (LossLess Join) (LJ)

Un esquema/tabla R (T, L) puede descomponerse haciendo proyecciones en diferentes sub-esquemas. La notación para esto es la siguiente: R (T, L) = {R_1, R_2, R_3, …, R_k}. R será un LossLess Join descompuesta si podemos reconstruir R a través de uniones naturales (natural joins).

$$\mathrm{R} = \pi_{\mathrm{T1}}(\mathrm{R}) \bowtie \pi_{\mathrm{T2}}(\mathrm{R}) \bowtie \cdots \bowtie \pi_{\mathrm{Tn}}(\mathrm{R})$$

Esta propiedad nos permite comprobar si la normalización que hemos hecho (al dividir en sub-esquemas) es correcta, pues si al hacer la LJ nos da un resultado distinto a R entonces no hemos hecho bien la normalización. (No se han perdido datos ni atributos)

2.4.3. Simplificación de dependencias funcionales

Dada una relación R (T, L) vamos a obtener un L' (Es decir, una simplificación de las dependencias funcionales de L, también llamado un recubrimiento no redundante) quitando todas las redundancias que aparezcan en él, basándonos en los **Axiomas de Armstrong.**

M es recubrimiento no redundante del conjunto de dependencias L si:

- $L^+ = M^+$
- Toda dependencia de M tiene implicados simples.
- En todos los implicantes de M no hay atributos superfluos (aquel que no aporta información adicional o relevante)
- En M no hay dependencias redundantes.

Ejemplo:

Si $L = \{AB \rightarrow CDE\}$, entonces el recubrimiento no redundante de L es M = $\{AB \rightarrow C,\ AB \rightarrow D,\ AB \rightarrow E\}$.

2.4.4. Algoritmo para obtener el recubrimiento no redundante

1) **Obtener implicados simples;** Aplicar el axioma de proyectividad hasta que todo implicado conste de un atributo.
2) **Eliminar atributos superfluos en implicantes:** Un atributo en un implicante compuesto es superfluo si, al suprimir la dependencia en cuestión, el cierre del implicante eliminando dicho atributo contiene al atributo eliminado.
3) **Eliminar dependencias redundantes:** Una dependencia es redundante si al suprimirla el cierre del implicante respecto de lo que queda, contiene al atributo implicado.

***** Sólo un reminder de que está prohibido simplificar sin más la parte del implicante.

Ejemplo: Calcular el recubrimiento no redundante de L:

Dado $T = \{A, B, C, D, E, F\}$

L = { $A \rightarrow BC$, (1)

$C \rightarrow D$, (2)

$AC \rightarrow E$, (3)

$AB \rightarrow C$, (4)

$F \rightarrow A\}$ (5)

$X = ABE$

Paso 1: Obtener implicados simples.

L1 = { $A \rightarrow B$,

$A \rightarrow C$,

$C \rightarrow D$,

$AC \rightarrow E$,

$AB \rightarrow C$,

$F \rightarrow A\}$

Paso 2: Eliminar atributos superfluos implicantes

¿Es redundante C en $AC \rightarrow E$?, pues puede tener como posibles simplificaciones $A \rightarrow E$ o $C \rightarrow E$. Para saber con cuál nos quedamos, empezaremos haciendo el cierre de A y después el cierre de C:

$L_{1-(AC \rightarrow E)}$ = { $A \rightarrow B$,

$A \rightarrow C$,

$C \rightarrow D$,

$AB \rightarrow C$,

$F \rightarrow A\}$

$\boldsymbol{A^{+}_{L_{1-(AC \rightarrow E)}} = ABCD.}$

Como $\boldsymbol{C} \in \boldsymbol{A^{+}_{L_{1-(AC \rightarrow E)}}}$, entonces C es redundante en $AC \rightarrow E$ y se tiene L_2

L_2 = { $A \rightarrow B$,

$A \rightarrow C$,

$C \rightarrow D$,

$\boldsymbol{A \rightarrow E}$,

$AB \rightarrow C$,

$F \rightarrow A\}$

¿Es redundante A en $AB \rightarrow C$?, pues puede tener como posibles simplificaciones $A \rightarrow C$ o $B \rightarrow C$

$L_{2-(AB \rightarrow C)}$ ={ $A \rightarrow B$,

$A \rightarrow C$,

$C \rightarrow D$,

$A \rightarrow E$

$F \rightarrow A\}$

$\boldsymbol{B^{+}_{L_{2-(AB \rightarrow C)}}}$ **= B.**

Como $\boldsymbol{A} \notin \boldsymbol{B^{+}_{L_{2-(AB \rightarrow C)}}}$, entonces A no es redundante en $AB \rightarrow C$.

¿Es redundante B en $AB \rightarrow C$?

$\boldsymbol{A^{+}_{L_{2-(AB \rightarrow C)}} = ABCDE}$

Como $\boldsymbol{B} \in \boldsymbol{A^{+}_{L_{2-(AB \rightarrow C)}}}$, entonces B es redundante en $AB \rightarrow C$ y se tiene L_3

L_3 = { $A \rightarrow B$,

$A \rightarrow C$,

$C \rightarrow D$,

$A \rightarrow E$,

$\boldsymbol{A \rightarrow C}$,

$F \rightarrow A\}$

Paso 3: Eliminar dependencias redundantes

Además de ver a simple vista que simplemente podríamos eliminar la quinta dependencia porque se repite en la segunda, vamos a hacerlo así:

¿Es redundante $A \rightarrow B$?

$L_{3-(A \rightarrow B)}$ = { $A \rightarrow C$,

$C \rightarrow D$,

$A \rightarrow E$,

$\boldsymbol{A \rightarrow C}$,

$F \rightarrow A$}

$\boldsymbol{A^{+}_{L_{3-(A \rightarrow B)}} = ACDE}$

Como $\boldsymbol{B \notin A^{+}_{L_{3-(A \rightarrow B)}}}$, entonces $A \rightarrow B$ no es redundante.

¿Es redundante $A \rightarrow C$?

$\boldsymbol{A^{+}_{L_{3-(A \rightarrow B)}} = ACDE}$

Como $\boldsymbol{C \in A^{+}_{L_{3-(A \rightarrow C)}}}$, entonces $A \rightarrow B$ es redundante y se obtiene L_4

L_4 = { $A \rightarrow B$,

$C \rightarrow D$,

$A \rightarrow E$,

$A \rightarrow C$,

$F \rightarrow A$}

Puede comprobarse que ninguna otra dependencia es redundante, por lo que L_4 es el recubrimiento no redundante de L.

2.4.5. Remover dependencias funcionales

Por último, debemos eliminar las dependencias funcionales redundantes, las cuales pueden ser deducidas a partir de otras dependencias. (Podemos remover partes del implicante que sean redundantes o la dependencia funcional al completo). El proceso para remover dependencias funcionales redundantes es ir dependencia a dependencia estudiando según sus cierres si el implicante sigue estando, eliminando esa dependencia.

2.4.6. Calcular las claves de un esquema R (T, L)

Denominamos una clave K de un esquema R (T, L) como el subconjunto de atributos de K que verifica:

- $K \subseteq T$
- $K \rightarrow T$
- No existe K', $K' \subseteq K$, con $K' \rightarrow T$.

Para calcular las claves, debemos seguir un algoritmo:

2.4.7. Algoritmo para el cálculo de una clave de un esquema R (T, L):

- L debe ser no redundante
- Se definen los siguientes subconjuntos de atributos: **I, D, ID y N**
    - **I** es el conjunto de atributos en T que aparecen en L sólo como implicantes (parte izquierda)
    - **D** es el conjunto de atributos en T que aparecen en L sólo como implicados (parte derecha)
    - **ID** es el conjunto de atributos en T que aparecen a ambos lados de dependencias en L.
    - **N** es el conjunto de atributos en T que no aparecen en ninguna dependencia de L.
- $\boldsymbol{Z = I \cup N}$ **es el núcleo** o intersección de todas las claves.
- Si $\boldsymbol{Z^+ = T}$**,** entonces Z es clave única.
- En caso contrario, se añaden a Z atributos procedentes de ID hasta que el cierre sea T.

Ejemplo:

Calcular una clave de R (T, L):

T = $\boldsymbol{\{A, B, C, D, E, F\}}$

L_4 = { $\boldsymbol{A \rightarrow B}$**,**

$\boldsymbol{C \rightarrow D}$**,**

$\boldsymbol{A \rightarrow E}$**,**

$\boldsymbol{A \rightarrow C}$**,**

$\boldsymbol{F \rightarrow A\}}$

Ya tenemos que L es no redundante.

I = $\{F\}$

D = $\{B, D, E\}$

ID = $\{A, C\}$

N = $\{\}$

Z = $I \cup N = \{F\}$

$Z^+ = F^+ = ABCDEF = T$

Dado que Z^+ = T, entonces **F es clave única de R (T, L).**

Ejemplo 2:

Calcular una clave de R (T, L):

T = $\boldsymbol{\{A, B, C, D, E, F, G\}}$

L_4 = { $\boldsymbol{AB \rightarrow C}$**,**

$\boldsymbol{CD \rightarrow E}$**,**

$\boldsymbol{C \rightarrow A}$,

$\boldsymbol{DE \rightarrow F}\}$

Tenemos que L es no redundante

I = $\{B, D\}$

D = $\{F\}$

ID = $\{A, C, E\}$

N = $\{G\}$

Z = $I \cup N = \{B, D, G\}$

$Z^+ = Z^+ = (BDG)^+ = BDG \neq T$

Dado que $Z^+ \neq$ T, entonces BDG no es clave. Probamos a añadir elementos de ID:

- $ABDG^+ = ABCDEFG = T$
- Dado que $ABDG^+ = T$, entonces $ABDG$ es clave candidata (no tiene por qué ser única).
- $BCDG^+ = ABCDEFG = T$, luego $BCDG$ también es clave candidata

2.5. NORMALIZACIÓN EN EL MODELO RELACIONAL

Es una operación que permite **obtener un modelo relacional** basado en un **modelo inicial** y una **serie de reglas**. Nos permite pasar nuestras relaciones iniciales con sus dependencias funcionales a una forma normalizada, donde no se repiten las partes implicadas (las partes que están a la derecha de las dependencias funcionales).

Para lograr esta normalización debemos seguir una serie de reglas para obtener el modelo relacional procedente de un diagrama Entidad-Relación.

La normalización nos permite **evitar redundancias, reducir el coste de actualizar los datos** (pues el dato solo se encuentra en un sitio) y **proteger la integridad de la base de datos.**

*Recordatorio de que en una relación el dominio de los atributos debe respetarse. Un ejemplo para entenderlo:

C_Departamento	N_Departamento	C_Trabajador	N_Trabajador	Anios
AB123	Administración	123456	Luis López	12
AB123	Administración	234567	Pedro Pérez	10
AB456	Ventas	345678	Juan Juárez	14
CD123	Marketing	456789	Raúl Rodríguez	16
CD123	Marketing	567890	Samuel Sota	15
YZ890	Mantenimiento	NULL	NULL	NULL

En esta tabla el dominio del atributo "años" es del valor entero. No podemos poner, por ejemplo, un nombre como "administración", debemos respetar ese dominio. Ahora, si nosotros queremos actualizar el nombre de un elemento del atributo "Departamento" tendríamos que recorrer la tabla hasta encontrar el elemento que queremos actualizar. Pero puede darse el caso de que ese elemento esté repetido, lo que hace más complicado actualizar el elemento que nosotros queremos. Para eso hacemos uso de la normalización. Encontramos cuatro formas de normalizar:

2.5.1. Formas de normalización

1) Primera Forma Normal (1FN)

Una relación se encuentra en 1FN cuando todos sus atributos son **atómicos**. Un atributo es atómico si el dominio de sus elementos no puede ser subdividido (Por ejemplo, en la tabla anterior N_Trabajador no será atributo atómico puesto que puede dividirse en nombre y apellido del trabajador). Otras condiciones que debe cumplirse para ser 1FN son:

- La relación tiene una **clave primaria única.**
- La clave primaria no debe tomar valores **NULL**
- **Todas las tuplas** deben tener el **mismo número de columnas**
- Los atributos no clave dependen de un atributo clave (dependencia funcional).
- El orden de los atributos y de las tuplas es irrelevante.

Esta **1FN** nos permite eliminar elementos duplicados de una base de datos.

2) Segunda Forma Normal (2FN)

Para que una relación esté en 2FN debe encontrarse primero en 1FN y que sus atributos no claves **dependan completamente** de la clave primaria.

Ejemplo:

¿Están estas dependencias funcionales en 2FN?

1) {nif, id_proyecto} → horas_trabajo. *
2) {nif, id_ proyecto} → nombre_empleado. **

* Podría estar en 2FN si entendemos "horas_trabajo" como las horas invertidas en un proyecto, pues necesitaríamos tanto el nif como el id del proyecto. Sin embargo, si se refiere a las horas trabajadas de un empleado, únicamente necesitaríamos el nif (y por tanto no sería un 2FN puesto que no existiría dependencia funcional completa).

** No puede estar en 2FN porque podemos hallar la dependencia funcional: {nif} → nombre_empleado.

3) Tercera Forma Normal (3FN)

Una relación está en **tercera forma normal** si, está en 2FN y además no existe ninguna **dependencia funcional transitiva** en sus atributos no claves. Es decir, si no dependen de ninguna otra cosa excepto de su clave.

* (**Transitividad:** Si $X \rightarrow Y$ e $Y \rightarrow Z$, entonces $X \rightarrow Z$)

Ejemplo: Dada la relación: Proyecto (id_proyecto, nombre_empleado, nombre_departamento)

id_proyecto → nombre_empleado

nombre_empleado → nombre_departamento

Tenemos nombre_empleado que depende completamente de nuestra clave primaria, pero claramente una dependencia funcional transitiva en nuestros atributos no claves. Por tanto, no está en 3FN.

4) Forma Normal de Boyce-Codd (FNBC)

La FNBC no es más que la generalización fuerte de la 3FN. Para que una relación esté en FNBC, para toda dependencia funcional A → B, debe ser una clave **candidata** o **primaria**

* Toda relación en FNBC está en 3FN, pero no toda relación 3FN está en FBNC

* Esta forma es suficiente para evitar inconsistencias

Ejemplo:

C_Departamento	N_Departamento	C_Trabajador	N_Trabajador	Anios
AB123	Administración	123456	Luis López	12
AB123	Administración	234567	Pedro Pérez	10
AB456	Ventas	345678	Juan Juárez	14
CD123	Marketing	456789	Raúl Rodríguez	16
CD123	Marketing	567890	Samuel Sota	15
YZ890	Mantenimiento	NULL	NULL	NULL

* Esta tabla para empezar no está ni en 1FN (porque hay elementos NULL y un atributo puede subdividirse (N_Trabajador)). Pero además **no podría** estar tampoco en FNBC porque C_Departamento y N_Departamento no son claves (ni candidatas, ni primarias) de la relación, pues no podemos identificar una fila completa con un elemento de esos atributos. (Por ejemplo, con el N_Trabajador "Luis López", sí puedes identificar los demás atributos de la tabla, por tanto, ese atributo sí sería una clave).

5) Cuarta Forma Normal (4FN)

Una relación estará en 4FN si previamente está en 3FN y si no hay dos o más datos independientes que pueden identificar (por ejemplo, A → B y A → C, de forma que B y C sean independientes).

6) Quinta Forma Normal (5FN)

Para que una relación esté en 5FN debe estar antes en 4FN y no se puede dividir en varias relaciones sin perder parte de la información.

2.5.2. El algoritmo de Ullman

El algoritmo de Ullman permite descomponer un esquema R (T, L) en subesquemas R_1, R_2, ..., R_n, de forma que:

- Cada R_i está en 3FN
- Se preservan las dependencias: $(\cup\ L_i)^+ = L^+$
- Es lossless join (puede recomponerse la relación original haciendo el join de las resultantes): $(\bowtie\ R_i)$ = R

Pasos del algoritmo de Ullman sobre un esquema R (T, L):

1) Obtener el recubrimiento no redundante de L
2) Calcular una clave
3) Agrupar las dependencias con igual implicante en una misma relación
4) Borrar los subesquemas que sean proyección de otro
5) Si ningún subesquema contiene a la clave, añadir un nuevo subesquema R_0 con los atributos de la clave

Ejemplo:

T = {A, B, C, D, E, F, G, H}

L = { $\boldsymbol{A \rightarrow E}$,

$\boldsymbol{BE \rightarrow D}$,

$\boldsymbol{AD \rightarrow BE}$,

$\boldsymbol{BDH \rightarrow E}$,

$\boldsymbol{AC \rightarrow E}$,

$\boldsymbol{F \rightarrow A}$,

$\boldsymbol{E \rightarrow B}$,

$\boldsymbol{D \rightarrow H}$

$\boldsymbol{BG \rightarrow F}$.

$\boldsymbol{CD \rightarrow A}$}

1) Simplificamos las dependencias y eliminamos redundancias

En primer lugar, cambiamos las dependencias funcionales a la forma $X \rightarrow A$, (con sólo un atributo al lado derecho):

L_2 = { $A \rightarrow E$,

$BE \rightarrow D$,

$AD \rightarrow B$,

$AD \rightarrow E$,

$BDH \rightarrow E$,

$AC \rightarrow E$,

$F \rightarrow A$,

$E \rightarrow B$,

$D \rightarrow H$

$BG \rightarrow F$.

$CD \rightarrow A$}

A continuación, comprobamos si los atributos de la parte izquierda son redundantes.

- El atributo B es redundante en $BE \rightarrow D$, pues tenemos que $E \rightarrow B$
- El atributo D es redundante en $AD \rightarrow B$, pues tenemos que $A \rightarrow E$ y $E \rightarrow B$
- El atributo D es redundante en $AD \rightarrow E$, pues tenemos que $A \rightarrow E$ y por tanto toda la dependencia funcional $AD \rightarrow E$ es redundante. Pues acabará siendo $A \rightarrow E$ que ya existe.
- El atributo C es redundante en $AC \rightarrow E$, pues tenemos que $A \rightarrow E$ y por tanto la dependencia funcional $AC \rightarrow E$ es redundante. Por el mismo motivo de más arriba.

Con esto obtenemos:

L_3 = { $A \rightarrow E$,

$E \rightarrow D$,

$A \rightarrow B$,

$BDH \rightarrow E$,

$F \rightarrow A$,

$E \rightarrow B$,

$D \rightarrow H$

$BG \rightarrow F$.

$CD \rightarrow A\}$

Para la dependencia funcional $BDH \rightarrow E$, podemos obtener:

- $BD \rightarrow E$, El cierre BD^+ = {B, D, H, E}, con esto obtenemos $BD \rightarrow E$.
- $BH \rightarrow E$, El cierre BH^+ = {B, H}
- $DH \rightarrow E$, El cierre DH^+ = {D, H}

Que habría sido lo mismo que decir que

- El atributo H es redundante en $BDH \rightarrow E$, pues tenemos $D \rightarrow H$

Por tanto, obtenemos

L_4 = { $A \rightarrow E$,

$E \rightarrow D$,

$A \rightarrow B$,

$BD \rightarrow E$,

$F \rightarrow A$,

$E \rightarrow B$,

$D \rightarrow H$

$BG \rightarrow F$.

$CD \rightarrow A\}$

Continuamos eliminando dependencias redundantes

L_4 = { $A \rightarrow E$,	(1)	$A^+ \setminus 1$ = {A, B}.	No es redundante
$E \rightarrow D$,	(2)	$E^+ \setminus 2$ = {E, B}	No es redundante
$A \rightarrow B$,	(3)	$A^+ \setminus 3$ = {A, B, D, E, H}	Como incluye a B, SÍ es redundante
$BD \rightarrow E$,	(4)	$BD^+ \setminus 4$ = {B, D H}	No es redundante
$F \rightarrow A$,	(5)	$F^+ \setminus 5$ = {F}	No es redundante
$E \rightarrow B$,	(6)	$E^+ \setminus 6$ = {E, D, H}	No es redundante
$D \rightarrow H$	(7)	$D^+ \setminus 7$ = {D}	No es redundante
$BG \rightarrow F$.	(8)	$BG^+ \setminus 8$ = {B, G}	No es redundante
$CD \rightarrow A\}$	(9)	$CD^+ \setminus 9$ = {C, D, H}	No es redundante

Por tanto, obtenemos:

L_5 = { $A \rightarrow E$,

$E \rightarrow D$,

$BD \rightarrow E$,

$F \rightarrow A$,

$E \rightarrow B$,

$D \rightarrow H$

$BG \rightarrow F$.

$CD \rightarrow A\}$

2) Creamos sub-esquemas R agrupando las dependencias por implicante:

$R_1 = \{A, E\}, \quad A \rightarrow E$

$R_2 = \{B, D, E\}, \quad E \rightarrow D, E \rightarrow B$

~~$R_3 = \{B, D, E\}, \quad BD \rightarrow E$~~

$R_4 = \{A, F\}, \quad F \rightarrow A$

$R_5 = \{D, H\}, \quad D \rightarrow H$

$R_6 = \{B, F, G\}, \quad BG \rightarrow F$

$R_7 = \{A, C, D\}, \quad CD \rightarrow A$

Con esto, ya obtenemos todo en forma 3FN

3) Calculamos la clave K

- I = {C, G}
- D = {H}
- ID = {A, B, D, E, F}
- N = {}
- K = {C, G}

Para comprobar si K es clave tenemos que comprobar si su cierre engloba todos los atributos de T. Como no los engloba debemos añadir a K un elemento de ID para ver si esa clave sí engloba a todos los elementos. (añadimos B)

BCG^+ = {A, B, C, D, E, F, G, H}. Por tanto, K = BCG

Una vez averiguada la clave, debemos comprobar si esta se encuentra en alguno de los sub-esquemas creados antes. Como no se encuentra (deben estar las tres letras juntas), creamos un R_0 que contenga la clave:

R_0 = K = {B, C, G}

Con esto ya hemos descompuesto la relación dejándola en 3FN. Ahora debemos comprobar si la relación se encuentra también en BNFC. Para ello todos los implicantes de las dependencias funcionales deben ser una clave primaria o candidata. Si nos fijamos en L_5 (o en los subesquemas)

(L_5 = { $A \rightarrow E$,

$E \rightarrow D$,

$BD \rightarrow E$,

$F \rightarrow A$,

$E \rightarrow B$,

$D \rightarrow H$

$BG \rightarrow F$.

$CD \rightarrow A\}$)

Una vez hagamos la última descomposición y creemos todos los sub-esquemas (para comprobar si están bien creados hacer Natural Join entre todos), R estará en BNFC

3. DISEÑO CONCEPTUAL

3.1. MODELO Y DIAGRAMAS E/R

3.2. PASO A TABLAS DEL MODELO E/R

4. CREACIÓN Y UTILIZACIÓN DE BASES DE DATOS

4.1. ADMINISTRACIÓN DE OBJETOS DE LA BASE DE DATOS

Cuando hablamos de bases de datos debemos identificar correctamente los diferentes conceptos alrededor de este término.

En primer lugar, SGDB (DBMS en inglés) significa Sistema Gestor de Bases de Datos. Cuando hablamos de MySQL, por ejemplo, estamos hablando de un SGBD, **no de una base de datos.**

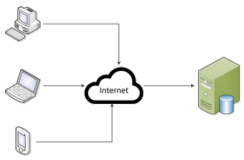
Otros SGBD bastante famosos son:

- Oracle
- PostgreSQL
- SQL Server
- BD2

El funcionamiento **básico** de un SGBD es **funcionar como un servidor**

No obstante, existen SGBD que no funcionan modo servidor, sino que funcionan en modo empotrado / embebido (como por ej: Apache Derby)

Que un sistema funcione como un servidor implica que se conectan a él clientes



No obstante, un servidor y el propio cliente podrían estar en la misma máquina (es el funcionamiento que se verá en esta asignatura). Esto implica que la conexión se realiza a través de la máquina local (IP: 127.0.0.1)

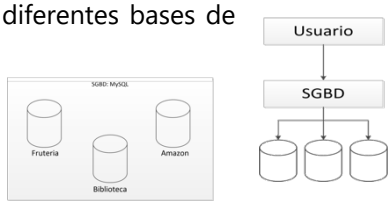
Clientes:

Un cliente es cualquier máquina o software de una máquina que se conecta al servidor creado por el SGBD. Estos clientes podrían ser:

- Aplicaciones Web (Amazon, eBay, …)
- Aplicaciones escritorio (Gestión biblioteca, gestión frutería, …)
- Aplicaciones de administración del SGBD

Bases de datos:

Como se comentaba previamente, el SGBD es el sistema que gestiona las diferentes bases de datos. La base de datos es el lugar donde se guarda la información.



Por tanto, en un SGBD (por ej: MySQL) podemos tener varias bases de datos:

Cuando un usuario accede a un SGBD, luego puede acceder a una base de datos en concreto.

Conexión:

La conexión al SGBD por parte de un cliente debe indicar:

- **Credenciales de usuario** (user y password).
- **Base de datos a la que quiere acceder**.

Usuarios:

El usuario y la contraseña permiten identificar si ese usuario tiene acceso al SGBD y en concreto a la base de datos especificada.

Se podrían crear dos usuarios diferentes para que accedan a la misma base de datos.

El objetivo de esto podría ser el siguiente supuesto:

Supongamos que tenemos una aplicación web de una frutería. Queremos por una parte que se pueda consultar las frutas a través de esa web. Así mismo, tenemos otra interfaz para administrar la base de datos (actualizar productos, etc…).

Podríamos tener dos usuarios:

- Acceso web: Usuario con privilegios sobre la base de datos SOLO para leer.
- Administrador: Usuario con privilegios sobre la misma BD pero puede modificar.

Clientes:

Finalmente, a nivel de clientes, existen clientes de administración de un SGBD que pemiten, con acceso de superusuario (root) controlar todas las bases de datos.

En MySQL esto se puede acceder con:

- MySQL Command Line Tool
- MySQL Workbench

4.1.1. Instalación del servidor de bases de datos

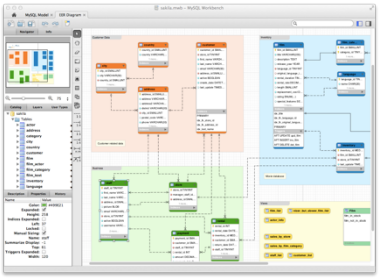
MySQL Workbench es una herramienta que permite varias funcionalidades, destacando:

- Acceso, gestión y administración de bases de datos.
- Diseño de bases de datos.

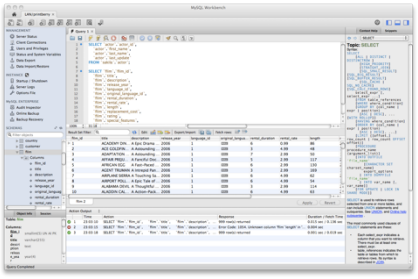
La herramienta puede ser descargada directamente desde la página de MySQL:

https://www.mysql.com/products/workbench/

La visión de editor de diagramas E-R es:



La visión de gestión de BD es:



En google aparece cómo instalarlo en Linux, mac o Windows

4.1.2. Acceso a bases de datos desde una aplicación cliente

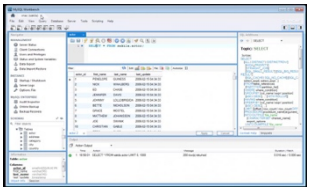
En el siguiente video se muestra un pequeño tutorial sobre el uso de MySQL Workbench y varias de sus funcionalidades.

https://www.youtube.com/watch?v=Myg70dM21Tc

4.2. GESTIÓN DE NIVELES DE ACCESO A LA BASE DE DATOS

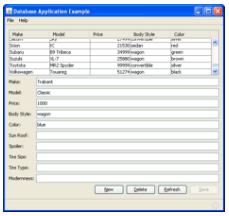
Una vez tenemos un DBMS operativo con un esquema de base de datos funcionando (con sus tablas y datos) tenemos varias opciones para interactuar contra él.

- Acceso mediante Workbench o similar



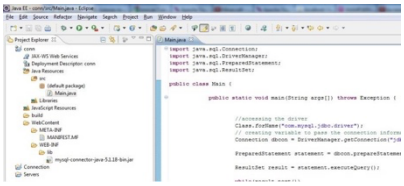
  - Entorno clásico para DBAs
  - Debemos definir las operaciones a realizar (DML y/o DDL).
  - Orientado a usuario experto en DB. Acceso directo a la DB. Poco usable para usuarios no expertos.

- Acceso por aplicaciones ya desarrolladas



  - Aplicaciones que usarán usuarios no expertos
  - Se despreocupa de todo: Ni si quiera tiene que saber que es una DB y qué hay detrás.
  - Interfaz amigable para usuarios no expertos.
  - Inserción, consulta y borrado de datos sin necesitar saber SQL o estructura lógica del esquema de BD.

- Acceso programático



  - Necesario conocimiento de SQL y lenguaje a desarrollar
  - Según el lenguaje y DBMS, usaremos un driver u otro.
  - Debemos controlar además del propio acceso a la BD el manejo de los datos, las sentencias a ejecutar (evitar SQL Injection), etc.
  - Orientado a programadores. Desarrollo de aplicaciones para usuarios no expertos.

Una solución planteada por el SQL Access Group (SAG) para resolver esta dependencia de DBMS fue ODBC

ODBC:

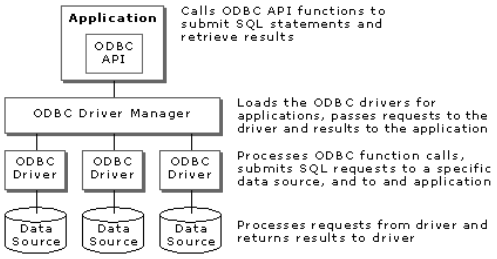
ODBC significa Open DataBase Connectivity

Su objetivo era permitir acceder a cualquier dato desde cualquier aplicación, sin importar que DBMS se usara.

Esto se lograba con una capa intermedia llamada nivel de Interfaz de Cliente SQL entre la aplicación y el DBMS.

El objetivo era traducir las consultas de la aplicación en comandos que el DBMS entienda.

El requisito es que tanto la aplicación desarrollada como el DBMS fueran compatibles con ODBC.



SQL es:

- Un lenguaje de **Definición de Datos (Data Definition Language, DDL)**
    - Tareas de definición de las estructuras que almacenarán los datos (**CREATE, DROP, ALTER, TRUNCATE)**
- Un lenguaje de **Manipulación de Datos (Data Manipulation Languaje, DML)**
    - Tareas de consulta o manipulación de datos (**SELECT, INSERT, DELETE y UPDATE**)
- Un lenguaje de **Control de Datos (Data Control Languaje, DCL)**
    - Controlar el acceso a los datos contenidos en la Base de Datos (**GRANT, REVOLKE**)

4.3. LENGUAJE DE DEFINICIÓN DE DATOS

4.3.1. Sentencias SQL de creación, modificación y borrado de esquemas y tablas

4.4. LENGUAJE DE MANIPULACIÓN DE DATOS

4.4.1. Sentencias SQL de inserción, consulta, edición y borrado de datos + subconsultas

4.5. LENGUAJE DE CONSULTA DE DATOS

4.5.1. Gestión de usuarios

4.5.1.1. Acceso usuarios del Sistema Gestor MySQL

Control de acceso bd:

Nivel 1: Comprobación de la conexión:

- Desde dónde se conecta el usuario
- Nombre del usuario
- Consulta a tabla "user" (host, user, password). (Base de Datos de Gestion mysql)

Nivel 2: Comprobación de privilegios:

- Por cada petición en la conexión se comprueba si hay privilegios para efectuarla.
- Consulta de tablas db, tables_priv, columns_priv, proces_priv. (Base de Datos de Gestión mysql)

4.5.1.2. Crear usuarios del sistema

Crear usuarios:

CREATE USER 'user1'@'localhost' **IDENTIFIED BY** 'user1';

GRANT ALL ON *.* **TO** 'user1'@''localhost' ';

SHOW GRANTS FOR 'user1'@''localhost' ';

Proyectar usuarios del Sistema:

SELECT * FROM mysql.user;

Cambiar la contraseña de un usuario:

ALTER USER 'user1'@'localhost' **IDENTIFIED BY** 'contraseña';

Eliminar un usuario:

DROP USER 'user1'@'localhost'

4.5.2. Sentencias SQL de consulta de datos

4.5.2.1. Otorgar privilegios al usuario (GRANT)

GRANT ALL [PRIVILEGES] **ON** *.* **TO** 'user1'@'localhost' **IDENTIFIED BY** 'contraseña' **WITH GRANT OPTION** {**REQUIRE** (...)};

Comando que permite 'conceder' privilegios a un usuario:

- **ALL**: se conceden **todos los privilegios** a este usuario. Los posibles privilegios: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP,REFERENCES, INDEX, ALTER, CREATE_TMP_TABLE, LOCK_TABLES, CREATE_VIEW,SHOW_VIEW, CREATE_ROUTINE, ALTER_ROUTINE, EXECUTE y GRANT.
- **ON**: los **objetos** a los que se aplican los privilegios, el formato es **base_de_datos.tabla**, *.* Otros ejemplos: ventas.*, contabilidad.polizas,
- **TO**: el **usuario** al que se le conceden los privilegios, el formato es **'usuario'@'servidor'.** Otros ejemplos: 'user1'@'%', 'sergio'@'192.168.10.132'
- **IDENTIFIED BY:** la **contraseña** se indica en esta parte y se escribe en texto plano.
- **WITH GRANT OPTION:** esta última parte es opcional, e indica que el usuario en cuestión puede a la vez **otorgar privilegios a otros usuarios**.
- **REQUIRE**: Opciones de **seguridad** en el acceso relacionadas con SSL.

4.5.2.2. Revocación de privilegios de usuario (REVOKE)

REVOKE ALL PRIVILEGES, **GRANT OPTION FROM** 'user1'@'localhost';

SHOW GRANTS FOR 'user1'@'localhost';

SELECT * FROM mysql.user;

4.5.3. Permisos sobre los objetos de la base de datos

Niveles Privilegios

- **Global**
- **De Base de Datos**
- **De Tabla**
- **De Columna**
- **De Rutina**

Privilegios Nivel Global

- Los permisos globales se aplican a todas las bases de datos de un servidor dado.
- Estos privilegios son almacenados en la tabla "mysql.user"

GRANT ALL ON *.* y **REVOKE ALL ON** *.* otorgan y quitan solo permisos globales.

GRANT ALL ON *.* **TO** 'user1'@'localhost' **WITH GRANT OPTION**;

REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user1'@'localhost';

Privilegios Nivel de Base de Datos (I)

- Los permisos de base de datos se aplican a todos los objetos en una base de datos dada.
- Estos permisos se almacenan en las tablas "mysql.db"

GRANT ALL ON db_name.* y **REVOKE ALL ON** db_name.* otorgan y quitan solo permisos de bases de datos.

GRANT ALL ON sakila.* **TO** 'user1'@'localhost';

REVOKE ALL ON sakila.* **FROM** 'user1'@'localhost';

Privilegios Nivel de Tabla

- Los permisos de tabla se aplican a todas las columnas en una tabla dada.
- Estos permisos se almacenan en la tabla "mysql.tables_priv"

GRANT ALL ON db_name.tbl_name y **REVOKE ALL ON** db_name.tbl_name otorgan y quitan permisos solo de tabla.

GRANT ALL ON sakila.* **TO** 'user1'@'localhost';

REVOKE ALL ON sakila.* **FROM** 'user1'@'localhost';

Privilegios Nivel de Columna

- Los permisos de columna se aplican a columnas en una tabla dada.
- Estos permisos se almacenan en las tablas “mysql.columns_priv” y “mysql.tables_priv”.
- Usando REVOKE, debe especificar las mismas columnas que se otorgaron los permisos

GRANT INSERT (title, interpret) **ON** sakila.cds **TO** ‘user1’@’localhost’;

REVOKE INSERT (title, interpret) **ON** sakila.cds **FROM** ‘user1’@’localhost’;

5. PROGRAMACIÓN DENTRO DE LA BASE DE DATOS

5.1. PROCEDIMIENTOS ALMACENADOS

Un procedimiento almacenado es un conjunto de sentencias SQL que se pueden almacenar en el servidor.

Puede ser visto como un pequeño programa que tenemos en la base de datos y que podemos llamar para realizar una serie de operaciones.

Trasladan el cómputo para transformas los datos al servidor.

Abstraen al usuario de la realización de consultas complejas y la transformación de datos.

Para crear los procedimientos almacenados:

- Dentro del código del procedimiento hay sentencias SQL, que usan el ";" como delimitador.
- Es necesario cambiar el delimitador de la sentencia que crea el procedimiento (es habitual usar $$).

DELIMITER $$

CREATE PROCEDURE mostrar_cervezas()

BEGIN

SELECT * FROM cerveza;

END $$

DELIMETER;

Para mostrar todos los procedimientos almacenados, se usa el comando **SHOW PROCEDURE STATUS;**

SHOW PROCEDURE STATUS [LIKE 'pattern' **| WHERE** search_condition**]**

Para hacer uso de un procedimiento almacenado, se usa el comando **CALL**:

CALL mostrar_cervezas();

Para borrar un procedimiento almacenado, se usa el comando DROP PROCEDURE:

DROP PROCEDURE mostrar_cervezas;

Un procedimiento almacenado puede tener tres clases de parámetros:

- **IN**: Parámetro de entrada. Los cambios dentro del procedimiento sobre los parámetros de este tipo, no se verán reflejados fuera del procedimiento.
- **OUT**: Parámetro de salida. Se puede cambiar el valor del parámetro dentro del procedimiento y dichos cambios se verán reflejados en el exterior. Sin embargo, el valor anterior no puede accederse dentro del procedimiento.
- **INOUT**: Parámetro de entrada y de salida. El procedimiento puede acceder en el procedimiento se verán reflejados en el exterior

Procedimiento que muestre la información de las cervezas de un fabricante dado el indentificador de éste:

```
DELIMITER $$
CREATE PROCEDURE mostrar_cervezas (
IN ID_Fab INT
)
BEGIN
SELECT * FROM cerveza WHERE ID_Fabricante = ID_Fab;
END $$
DELIMITER;
```

Para almacenar el resultado de un SELECT en una variable puede usarse la cláusula INTO:

```
DELIMETER $$
CREATE PROCEDURE obtener_n_cervezas (
OUT n_cervezas INT
)
BEGIN
SELECT COUNT (*) INTO n_cervezas FROM cerveza;
END $$
DELIMETER;
```

Para llamar a este nuevo procedimiento necesitamos una variable de usuario.

- Permiten almacenar un valor y hacer referencia a él más tarde.
- Posibilitan pasar valores de una sentencia a otra.
- Son específicas de cada conexión (se liberan al terminarla).
- Todas deben comenzar con @ (No así las variables locales).

```
CALL obtener_n_cervezas (@n_beeers);
SELECT @n_beers;
```

Es posible definir el valor de una variable y definir nuevas variables de usuario con SET:

```
DELIMETER $$
CREATE PROCEDURE increment (
INOUT n INT
)
BEGIN
SET n = n +1;
END $$
DELIMETER;
```

SET @valor = 1;

CALL increment(@valor);

SELECT @valor;

En el caso de variables locales, es necesario declararlas previamente en el procedimiento, la sintaxis es:

DECLARE nombre tipo [**DEFAULT** valor];

DELIMETER $$

CREATE PROCEDURE obtener_n_mas_1_cervezas (**OUT** n **INT**)

BEGIN

DECLARE n_cervezas **INT**;

SELECT COUNT (*) **INTO** n_cervezas **FROM** cerveza;

SET n = n_cervezas + 1;

END $$

DELIMETER;

Es posible crear bloques condicionales:

IF condición **THEN** sentencias

[**ELSEIF** condición **THEN** sentencias]

[**ELSE** sentencias]

END IF;

CASE variable

WHEN valor1 **THEN** sentencias

WHEN valor2 **THEN** sentencias

[**ELSE** sentencias]

END CASE;

También es posible crear bucles:

WHILE condición **DO** sentencias<;

END WHILE;

REPEAT sentencias **UNTIL** condición

END REPEAT;

Programar un procedimiento llamado get_nombre_socio que, dado un identificador de un socio, retorne su nombre. Si es socio no existe, debe retornar una carrera que contenga “El socio no existe’

```
CREATE PROCEDURE get_nombre_socio (IN id INT, OUT nombre_ocio VARCHAR(100))
BEGIN
DECLARE socio_existe INT;
SELECT COUNT(*) INTO socio_existe FROM socio WHERE ID_socio = id;
IF socio_existe = 1 THEN
SELECT nombre INTO nombre_socio FROM socio WHERE ID_socio = id;
ELSE
SET nombre_socio = 'El socio no existe';
END IF;
END $$
```

Programar un procedimiento llamado get_dir_socio que, dado un identificador de un socio, retorne su dirección. Si es socio no existe, debe retornar una carrera que contenga "El socio no existe'. Si el socio no tiene ninguna dirección anotada, debe retornar "Dirección no asignada".

```
CREATE PROCEDURE get_dir_socio (IN id INT, OUT dir_ocio VARCHAR(100))
BEGIN
DECLARE socio_existe INT;
SELECT COUNT(*) INTO socio_existe FROM socio WHERE ID_socio = id;
IF socio_existe = 1 THEN
SELECT direccion INTO dir_socio FROM socio WHERE ID_socio = id;
ELSE
SET dir_socio = 'El socio no existe';
END IF;
END $$
```

Programar un procedimiento llamado borra_socios que, dados dos enteros, elimine los socios cuyo identificador esté entre ambos números y sea par (ambos inclusive).).

```
CREATE PROCEDURE borra_socios (IN id INT, OUT final INT))
BEGIN
    DECLARE i INT DEFAULT inicio;
    WHILE <= final DO
        IF MOD(I, 2) = THEN
            DELETE FROM socio WHERE ID_Socio = i;
        END IF;
        SET I = i +1;
    END WHILE
END $$
```

5.2. TRIGGERS

Los triggers (disparadores) son elementos que están asociados con tablas y se almacenan en la BD (como los procedimientos y las funciones).

Son elementos asociados a eventos

- Se ejecutan cuando se produce un evento / acción sobre la tabla a la que están asociados.
- Están asociados a operaciones de modificación de datos de la tabla: INSERT, DELETE, UPDATE.

Ejemplo: Trigger que se activa antes de hacer la inserción de una persona.

- Éste comprobará si el DNI que nos dan tiene letra, y si no la tiene, llamará a la función de calcular letra, para ponérsela, de forma que se agregue siempre el DNI con la letra

Al crear un trigger, además de darle un nombre, hay que especificar:

- Si se ejecuta antes o después de la sentencia que lo activa.
- Qué evento lo activa (inserción, borrado o actualización de datos)
- La tabla a la que está asociado.

CREATE TRIGGER nombre

{**BEFORE** | **AFTER**} {**INSERT** | **DELETE** | **UPDATE**}

ON table **FOR EACH ROW**

BEGIN

sentencias;

END $$

Hay algunas **restricciones**:

- No puede haber dos triggers referidos al mismo momento y evento.
- Sólo pueden ejecutarse comandos del DML.
- No debe usarse la propia tabla en el código.
- No puede referenciarse la tabla a la que está asociado (Sí es posible para realizar una consulta).
    - **OLD** hace una referencia a la tupla que se va a **borrar o actualizar**
        - Es de sólo lectura
    - **NEW** hace referencia a los nuevos valores a **insertar o actualizar**.
        - Puede modificarse con **SET** si se hace en el momento **BEFORE**.

Para eliminar un trigger:

DROP TRIGGER nombre;

Programar un trigger que compruebe que, al insertar que un bar vende una cerveza, su precio no sea negativo. (Si lo es, debe insertarse NULL.)

```
CREATE TRIGGER comprueba_precio
BEFORE INSERT ON vende FOR EACH ROW
BEGIN
    IF NEW precio < 0 THEN
        SET NEW precio = NULL;
    END IF;
END $$
```

No queremos tener cervezas que hayan dejado de gustar a todos los socios. Programar un trigger que, al borrar un gusto, si no hay socios a los que les siga gustando la cerveza involucrada, borre también la cerveza,

```
CREATE TRIGGER borra_cerveza_impopular
ALTER DELETE ON gusta FOR EACH ROW
BEGIN
    DECLARE n INT;
    SELECT COUNT(*) INTO n FROM gusta WHERE ID_cerveza = OLD.ID_cerveza;
    IF n = 0 THEN
        DELETE FROM cerveza WHERE ID_cerveza = OLD.ID_cerveza;
    END IF;
END $$
```

Añadir un campo "veces_modificado" a la tabla vende, que indicará el número de veces que se ha modificado el precio de una cerveza en un bar.

```
ALTER TABLE vende
ADD COLUMN veces_modificado INT DEFAULT 0;
```

Programar un trigger que haga que, cada vez que se modifique un precio, se incrementa dicho contador. Actualizar el precio de la cerveza 4 en el bar con licencia "ES-b2" a 1€ para hacer la prueba.

```
CREATE TRIGGER modificación_precio
BEFORE UPDATE ON vende FOR EACH NOW
BEGIN
    IF OLD precio <> NEW precio THEN
        SET NEW.veces_modificado = OLD.veces_modificado + 1;
    END IF;
END $$
```

UPDATE vende

SET precio = 1;

WHERE ID_cerveza == 4 AND licencia = ‘ES-b2’;

6. ACCESO PROGRAMÁTICO A BASES DE DATOS USANDO JDBC

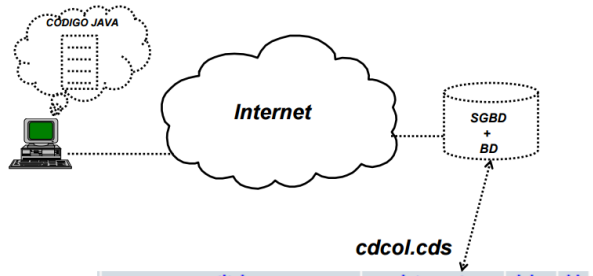
6.1. INTRODUCCIÓN

Dependiendo del lenguaje de programación tendremos una serie de conectores disponibles para trabajar con bases de datos.

6.1.1. JDBC:

Java **D**ata**B**ase **C**onnectivity

JDBC es un API que permite ejecutar operaciones sobre bases de datos en Java.

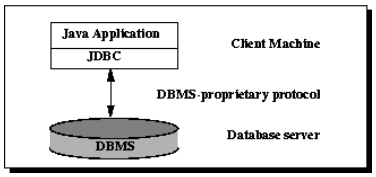


titel	interpret	jahr	id
Beauty	Ryuichi Sakamoto	1990	1
Goodbye Country (Hello Nightclub)	Groove Armada	2001	4
Glee	Bran Van 3000	1997	5

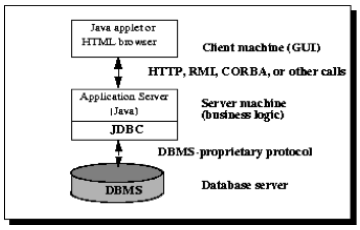
Como cualquier programa Java es independiente del SO o de la base de datos a la que se accede (driver-dependent).

Utiliza SQL como lenguaje de acceso y manipulación.

Arquitectura de 2 capas:



Arquitectura de 3 capas:



La conexión a un DBMS en particular se realiza junto con el driver/biblioteca de conexión apropiado.

La mayoría de los DBMS tienen su propia implementación de driver JDBC:

- Oracle: http://www.oracle.com/technetwork/database/features/jdbc/index- 091264.html
- MySQL: http://www.oracle.com/technetwork/database/features/jdbc/index- 091264.html
- PostgreSQL: http://www.oracle.com/technetwork/database/features/jdbc/index- 091264.html
- SQL Server: http://www.oracle.com/technetwork/database/features/jdbc/index- 091264.html

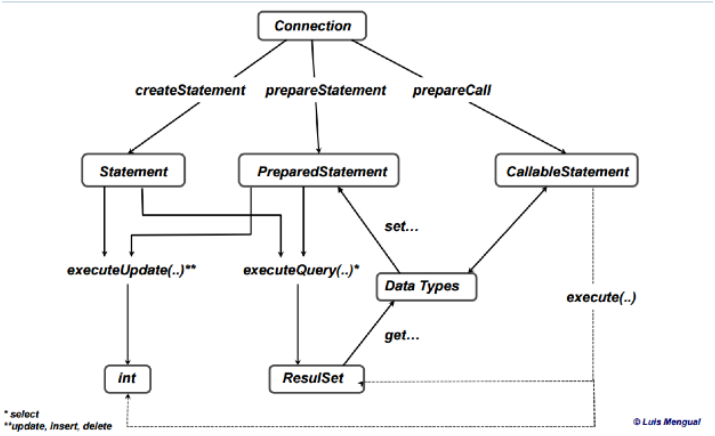
Componentes básicos de JDBC:

API: Proporciona el acceso programático a través de determinadas clases y métodos. Puede ejecutar sentencias SQL, obtener resultados y propagar cambios. El API JDBC es parte de la plataforma Java (SE EE). Paquetes java.sql y javax.sql (la mayoría de lo que se usa viene de java.sql).

6.1.2. Clases e Interfaces JDBC:

Componentes básicos de JDBC:

Clase	Descripción
DriverManager	*Para cargar un driver*
Connection	*Para establecer conexiones con las bases de datos*
Statement	*Para ejecutar sentencias SQL y enviarlas a las BBDD*
ResultSet	*Para almacenar el resultado de la consulta*



Tipos de datos SQL / Java

SQL	→	**Java**
INTEGER o INT		int
SMALLINT		shot
FLOAT		float
DOUBLE		double
CHARACTER(n) o CHAR(n)		String
VARCHAR(n)		String
BOOLEAN		boolean
DATE		java.sql.Date
TIME		java.sql.Time
TIMESTAMP		java.sql.Timestamp
BLOB		java.sql.Blob

6.2. CONEXIÓN A UNA BASE DE DATOS USANDO JDBC

6.2.1. Incorporar driver JDBC en Proyecto de consola

El fichero físico que contiene el driver (JAR/ZIP) debe estar en el CLASSPATH de la app.

Instalación del Driver:

En el siguiente vídeo se muestra el proceso de instalación de driver y prueba:

https://www.youtube.com/watch?v=foqBso6MHX8

6.2.2. Fases de comunicación con JDBC

1. Cargar el driver
2. Establecer conexión con la BBDD (clase Connection)
3. Crear y ejecutar una sentencia SQL (clases Statement / PreparedStatement, métodos executeQuery / executeUpdate)
4. Gestionar el resultado (clase ResultSet)
5. Liberar recursos

```
import java.sql.*;

public class EJ2 {

    private String userName;
    private String password;
    private String driver;
    private String dbName;
    private String url;

    private Connection conn = null;

    private void preparedConnection() throws Exception {
        this.url = "jdbc:mysql://localhost:3306/";
        this.driver = "com.mysql.jdbc.Driver";
        this.dbName = "Personas";
        this.userName = "root";
        this.password = "47291606 H ache";

        //Fase 1: El driver se carga con este método. (de la clase Class())
        //Simplemente carga el driver, si no funciona es porque falta cargar el JAR
de JDBC.
        Class.forName(driver);

        //Fase 2: La conexion la establecemos a traves de este metodo. (de la clase
DriverManager())
        conn = DriverManager.getConnection(url + dbName, userName ,password);
        if (!conn.isClosed()) {
            System.out.println("Conectado a la base de datos!");
        }
    }
```

```java
    public void run() throws Exception {
        this.preparedConnection();
        //this.executeSelect();
        //this.executeUpdate();
        //this.executeInsert();
        //this.executeDelete();
    }

    private void executeSelect() throws Exception {
        //Fase 3: Crear y ejecuar una sentencia SQL
        PreparedStatement  st = conn.prepareStatement("SELECT * FROM PERSONAS WHERE
id = ?");

        st.setInt(1, 2);
        ResultSet rs = st.executeQuery(); //Fase 4: Gestionar el resultado

        int numRegs = 0;

        while (rs.next()) {
            System.out.println("ID: " + rs.getString("id"));
            System.out.println("Name: " + rs.getString("name"));
            System.out.println("Phone: " + rs.getString("phone"));
            System.out.println("Email: " + rs.getString("email"));
            System.out.println("Contry: " + rs.getString("country"));
            System.out.println("-----------------------------------------------
");
            numRegs++;
        }

        //Fase 5: Liberar recursos
        st.close();
        System.out.println("Número de registros:" + numRegs);
    }

    private void getUserByID (int id) throws Exception {
        PreparedStatement  st = conn.prepareStatement("SELECT * FROM PERSONAS WHERE
id = ?");

        st.setInt(1, id);
        ResultSet rs = st.executeQuery();

        while (rs.next()) {
            System.out.println("ID: " + rs.getString("id"));
            System.out.println("Name: " + rs.getString("name"));
            System.out.println("Phone: " + rs.getString("phone"));
            System.out.println("Email: " + rs.getString("email"));
            System.out.println("Contry: " + rs.getString("country"));
            System.out.println("-----------------------------------------------
");
        }
        st.close();
```

```
    }

    private void executeUpdate() throws Exception {
        getUserByID(7);

        PreparedStatement  st = conn.prepareStatement("UPDATE PERSONAS SET name = ?
WHERE id = ?");
        st.setString(1, "Alejandro");
        st.setInt(2, 7);

        int count = st.executeUpdate();

        System.out.println("Rows updated: " + count);
        st.close();
        getUserByID(7);
    }

    private int getCuantos() throws Exception {
        PreparedStatement  st = conn.prepareStatement("SELECT COUNT(*) FROM
PERSONAS");

        ResultSet rs = st.executeQuery();

        int v = -1;
        while (rs.next()) {
            v = rs.getInt("COUNT(*)");
        }

        st.close();

        return v;
    }

    private void getUserByName (String n) throws Exception {
        PreparedStatement  st = conn.prepareStatement("SELECT * FROM PERSONAS WHERE
name = ?");

        st.setString(1, n);
        ResultSet rs = st.executeQuery();

        while (rs.next()) {
            System.out.println("ID: " + rs.getString("id"));
            System.out.println("Name: " + rs.getString("name"));
            System.out.println("Phone: " + rs.getString("phone"));
            System.out.println("Email: " + rs.getString("email"));
            System.out.println("Contry: " + rs.getString("country"));
            System.out.println("-------------------------------------------------
");
        }
        st.close();
    }
```

```
    private void executeInsert() throws Exception {
        int cuantos = getCuantos();
        System.out.println("Registros en la BD: " + cuantos);

        PreparedStatement  st = conn.prepareStatement("INSERT INTO PERSONAS (name,
phone, email, country) VALUES (?,?,?,?)");

        st.setString(1, "Pepito");
        st.setString(2, "666 666 666");
        st.setString(3, "pepito@gmail.com");
        st.setString(4, "Spain");

        int count = st.executeUpdate();
        System.out.println("Filas anhanidas: " + count);

        cuantos = getCuantos();
        System.out.println("Registros en la BD: " + cuantos);

        getUserByName("Pepito");
    }

    private void executeDelete() throws Exception {
        int cuantos = getCuantos();
        System.out.println("Registros en la BD: " + cuantos);

        PreparedStatement  st = conn.prepareStatement("DELETE FROM PERSONAS WHERE
country = ?");

        st.setString(1, "South Korea");

        int count = st.executeUpdate();
        System.out.println("Filas borradas: " + count);

        cuantos = getCuantos();
        System.out.println("Registros en la BD: " + cuantos);
    }
}
```

¿Debemos cerrar la conexión (conn.close) tras cada consulta? ¿Por qué?

No, es ineficiente si se van a ejecutar varias cosas.

Posibles errores:

Errores más comunes:

- Driver no cargado (no se encuentra en classpath).
- Fallo de conexión.
- No existe la base de datos.

- Error de sintaxis en sentencia SQL.
- Violación reglas integridad referencial.

¿Qué / Cómo debemos comprobar para solucionar cada error?

Cualquier de esos errores producirá una excepción.

En función del tipo de excepción o de los datos que nos de la misma (códigos de error, por ejemplo) podemos saber el error concreto.

PreparedStatement:

Cada vez que enviamos una consulta al SGBD, éste:

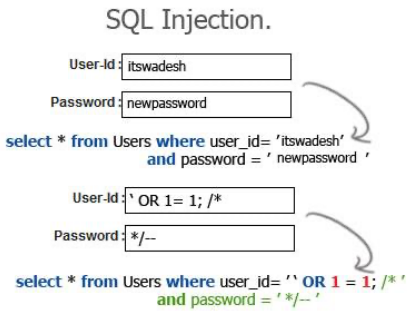
- La analiza sintácticamente (Query Processor)
- Construye un plan para ejecutarla (Query Optimizer)

Si tenemos un bucle donde repetidamente lanzamos la misma query con diferentes parámetros es ineficiente usar la clase Statement.

Es mejor usar para estas situaciones PreparedStatement.

Además, evita los ataques por inyección de código SQL en Java.

SQL Injection:



Un ejemplo en código en: https://www.youtube.com/watch?v=foqBso6MHX8

Arriba hemos visto las implementaciones de los métodos Insert, Update y Delete.

A modo de resumen, a continuación, explicaremos qué clases y métodos usar:

ExecuteQuery → Select.

executeUpdate → Insert, Update, Delete.

PreparedStatement → Cuando se realiza una operación varias veces.

6.3. EJECUCIÓN DE UNA CONSULTA (STATEMENT Y RESULTSET)

Para poder recuperar datos de la base de datos necesitamos construir una instrucción de consulta.

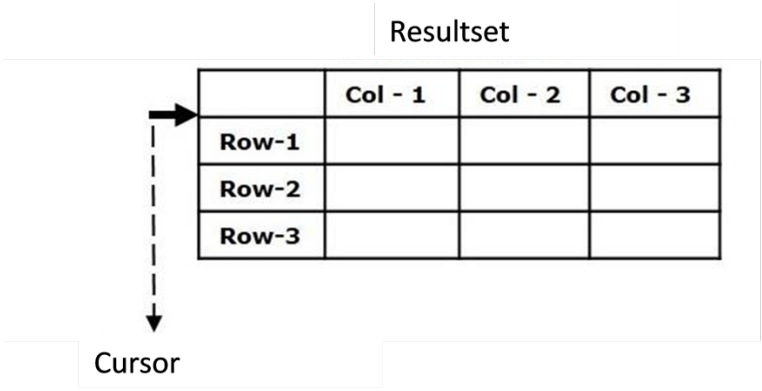
La primera opción que vamos a ver es la ejecución de una sentencia estática, que se definirá en una variable tipo String. La clase de sentencia 'Statement' se creará desde la conexión creada anteriormente.

Al ejecutar la consulta (executeQuery) usando el objeto statement, si la consulta es correcta, obtendremos como resultado una tabla. Recordemos que el resultado de aplicar operaciones de álgebra relacional es una tabla derivada, es decir, una tabla de datos que no están guardados en la base de datos como tabla física. El resultado de la consulta se recibirá dentro de un objeto Resultset.

El método para ejecutar consultas puede lanzar varias excepciones, dependiendo del error que haya ocurrido (Error en la sentencia SQL, error en la conexión, etc).

El objeto ResultSet encapsula la tabla de datos resultado. La tabla resultada tendrá tantas columnas como las solicitadas (proyectadas), en la selección SELECT de las consultas, y con el nombre indicado en la misma (nombre de la columna o el alias puesto en la consulta).

Para acceder a los valores devueltos, se dispone de un cursor. Al recibir el resultado, el cursor se encuentra antes de la primera fila de resultados. La clase ofrece métodos para saber si el cursor ha llegado al final de la tabla de resultados y avanzar dentro de la tabla.



Además, el objeto de resultados ofrece métodos para obtener el valor de una columna de la tabla (por número o nombre de columna), y algunos métodos adicionales, como por ejemplo para saber cuántas filas ha devuelto la consulta.

6.4. MANEJO DE LOS RESULTADOS (RESULTSET Y METADA)

Dentro de cada tabla de resultados (ResultSet) existe un enlace a los metadatos del conjunto de resultados obtenidos de cada consulta.

Esos metadatos nos proporcionan datos descriptivos de los resultados obtenidos, como el número de columnas, nombre y tipos de cada columna, por ejemplo:

En concreto, el objeto de la clase de metadatos (ResultSetMetaData) se puede obtener de la siguiente manera:

ResultSetMetaData m = rs.getMetaData();

Con el objeto de metadatos se puede obtener la siguiente información:

- Obtener el número de columnas: m.getColumnCount();
- Obtener el número de columnas: m.getTableName();

Para cada columna:

- Obtener su nombre = m.getColumnName(i);
- Obtener su etiqueta = m.getColumnLabel(i);

- Obtener su tipo = m.getColumnType(i);
- Obtener su tipo = m.getColumnDisplaySize(i);

Hay que tener en cuenta que la implementación en cada SGBD es distinta, así como para las distintas versiones de un mismo driver JDBC. Hay versiones que los metadatos no devuelven toda la información sobre la consulta, al no estar algunos métodos implementados.

6.5. MANEJO DE LOS RESULTADOS (CURSORES EN RESULTSET) Y CONSULTAS CON PARÁMETROS

6.5.1. Manejo de cursores en los resultados en Java

Una de las nuevas características del API JDBC 2.0 es la habilidad de mover el cursor en una hoja de resultados (ResultSet) tanto hacia atrás como hacia adelante. También hay métodos que nos permiten mover el cursor a una fila particular y comprobar la posición del cursor. La hoja de resultados Scrollable hace posible crear una herramienta GUI (Interface Gráfico de Usuario) para navegar a través de ella, lo que probablemente será uno de los principales usos de esta característica. Otro uso será movernos a una fila para actualizarla.

Por ejemplo:

- Comprobar si el cursor está sin inicializar – resultSet.isBeforeList();
- Comprobar si el cursor está en la primera fila – resultSet.isFirst();
- Comprobar si el cursor está en la última fila de resultados – resultSet.isLast();
- Comprobar si el cursor está al final, es decir, ya se ha procesado la última fila – resultSet.isAfterLast();
- Obtener el número de línea en el que se encuentra el cursor resultSet.getRow();
- Avanzar el cursor – resultSet.next();
- Retroceder el cursor – resultSet.previous();
- Llevar el cursor antes de la primera posición – resultSet.beforeFirst();
- Llevar el cursor a la primera posición – resultSet.first();
- Llevar el cursor a la última posición - resultSet.last();
- Llevar el cursor a una posición dada - resultSet.absolute(i);
    - Si la i es positiva irá a la posición i desde el principio.
    - Si la i es 0, el cursor se coloca antes de la primera fila.
    - Si la i es negativa irá a la posición i desde el final.

Antes de poder aprovechar estas ventajas, necesitamos crear un objeto ResultSet Scrollable.

String sqlQuery = "SELECT * FROM actor;";

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, .CONCUR_READ_ONLY);

ResultSet srs = stmt.executeQuery(sqlQuery);

Las distintas implementaciones de conectores y drivers JDBC implementan los cursores de diferente manera. Por defecto, los cursores son de sólo lectura y, al menos, permiten recorrer el conjunto de datos

hacia delante, no todas las implementaciones de los drivers JDBC permiten retroceder el cursor en el conjunto de resultados.

Este código es similar al utilizado anteriormente, excepto en que añade dos argumentos al método createStatement.

El primer argumento es una de las tres constantes añadidas al API ResultSet para indicar el tipo de un objeto ResultSet: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, y TYPE_SCROLL_SENSITIVE.

El segundo argumento es una de las dos constantes de ResultSet para especificar si la hoja de resultados es de sólo lectura o actualizable:CONCUR_READ_ONLY y CONCUR_UPDATABLE.

Lo que debemos recordar aquí es que si especificamos un tipo, también debemos especificar si es de sólo lectura o actualizable. También, debemos especificar primero el tipo, y como ambos parámetros son int, el compilador no comprobará si los hemos intercambiado.

Respecto al acceso a los datos (RSConcurrency) los cursores pueden ser:

- ResultSet.CONCUR_READ_ONLY – Crea un cursor de sólo lectura. Es el modo por defecto.
- ResultSet.CONCUR_UPDATABLE – Crea un result set actualizable / modificable. En la actualidad no se recomienda utilizar de esta manera los datos.

Además, respecto al modo de recorrer los resultados:

- TYPE_FORWARD_ONLY – Sólo permite recorrer los resultados hacia adelante (next)
- TYPE_SCROLL_INSENSITIVE – El cursor se puede mover hacia adelante por los resultados y hacia atrás. El ResultSet además no se ve afectado por los cambios hechos en la base de datos durante la utilización del conjunto de resultados.
- TYPE_SCROLL_SENSITIVE - El ResultSet se puede mover hacia adelante por los resultados y hacia atrás. Además el conjunto de datos se queda enlazado con las tablas de la base de datos, de tal manera que los cambios realizados en la base de datos mientras el ResultSet esté abierto se reflejan en el objeto de java.

Una vez que tengamos un objeto ResultSet desplazable, srs en el ejemplo anterior, podemos utilizarlo para mover el cursor sobre la hoja de resultados. Recuerda que cuando creabamos un objeto ResultSet anteriormente, tenía el cursor posicionado antes de la primera fila. Incluso aunque una hoja de resultados se seleccione desplazable, el cursor también se posiciona inicialmente delante de la primera fila. En el API JDBC 1.0, la única forma de mover el cursor era llamar al método next. Este método todavía es apropiado si queremos acceder a las filas una a una, yendo de la primera fila a la última, pero ahora tenemos muchas más formas para mover el cursor.

La contrapartida del método next, que mueve el cursor una fila hacia delante (hacia el final de la hoja de resultados), es el nuevo método previous, que mueve el cursor una fila hacia atrás (hacia el inicio de la hoja de resultados). Ambos métodos devuelven false cuando el cursor se sale de la hoja de resultados (posición antes de la primera o después de la última fila), lo que hace posible utilizarlos en un bucle while. Ye hemos utilizado un método next en un bucle while, pero para refrescar la memoria, aquí tenemos un ejemplo que mueve el cursor a la primera fila y luego a la siguiente cada vez que pasa por el bucle while. El bucle termina cuando alcanza la última fila, haciendo que el método next devuelva false. El siguiente fragmento de código imprime los valores de cada fila de srs, con cinco espacios en blanco entre el nombre y el precio.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);

ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

```
while (srs.next()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

La salida se podría parecer a esto.

```
Colombian     7.99
French_Roast     8.99
Espresso     9.99
Colombian_Decaf     8.99
French_Roast_Decaf     9.99
```

Al igual que en el fragmento anterior, podemos procesar todas las filas de srshacia atrás, pero para hacer esto, el cursor debe estar detrás de la última fila. Se puede mover el cursor explícitamente a esa posicón con el método afterLast. Luego el método previous mueve el cursor desde la posicón detrás de la última fila a la última fila, y luego a la fila anterior en cada iteracción del bucle while. El bucle termina cuando el cursor alcanza la posición anterior a la primera fila, cuando el método previous devuelve false.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

La salida se podría parecer a esto.

```
French_Roast_Decaf     9.99
Colombian_Decaf     8.99
Espresso     9.99
French_Roast     8.99
Colombian     7.99
```

Se puede mover el cursor a una fila particular en un objeto ResultSet. Los métodos first, last, beforeFirst, y afterLast mueven el cursor a la fila indicada en sus nombres. El método absolute moverá el cursor al número de fila indicado en su argumento. Si el número es positivo, el cursor se mueve al número dado desde el principio, por eso llamar a absolute(1) pone el cursor en la primera fila. Si el número es negativo, mueve el cursor al número dado desde el final, por eso llamar a absolute(-1) pone el cursor en la última fila. La siguiente línea de código mueve el cursor a la cuarta fila de srs.

```
srs.absolute(4);
```

Si srs tuviera 500 filas, la siguiente línea de código movería el cursor a la fila 497.

```
srs.absolute(-4);
```

Tres métodos mueven el cursor a una posición relativa a su posición actual. Como hemos podido ver, el método next mueve el cursor a la fila siguiente, y el método previous lo mueve a la fila anterior. Con el método relative, se puede especificar cuántas filas se moverá desde la fila actual y también la dirección en la que se moverá. Un número positivo mueve el cursor hacia adelante el número de filas dado; un número negativo mueve el cursor hacia atrás el número de filas dado. Por ejemplo, en el siguiente fragmente de código, el cursor se mueve a la cuarta fila, luego a la primera y por último a la tercera.

```
srs.absolute(4); // cursor está en la cuarta fila

. . .

srs.relative(-3); // cursor está en la primera fila

. . .

srs.relative(2); // cursor está en la tercera fila
```

El método getRow permite comprobar el número de fila donde está el cursor. Por ejemplo, se puede utilizar getRow para verificar la posición actual del cursor en el ejemplo anterior.

```
srs.absolute(4);

int rowNum = srs.getRow(); // rowNum debería ser 4

srs.relative(-3);

int rowNum = srs.getRow(); // rowNum debería ser 1

srs.relative(2);

int rowNum = srs.getRow(); // rowNum debería ser 3
```

Existen cuatro métodos adicionales que permiten verificar si el cursor se encuentra en una posición particular. La posición se indica en sus nombres:isFirst, isLast, isBeforeFirst, isAfterLast. Todos estos métodos devuelven un boolean y por lo tanto pueden ser utilizados en una sentencia condicional. Por ejemplo, el siguiente fragmento de código comprueba si el cursor está después de la última fila antes de llamar al método previous en un bucle while. Si el método isAfterLast devuelve false, el cursor no estará después de la última fila, por eso se llama al método afterLast. Esto garantiza que el cursor estára después de la última fila antes de utilizar el método previous en el bucle while para cubrir todas las filas de srs.

```
if (srs.isAfterLast() == false) {
        srs.afterLast();
}
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

En la siguiente página, veremos cómo utilizar otros dos métodos de ResultSetpara mover el cursor, moveToInsertRow y moveToCurrentRow. También veremos ejemplos que ilustran por qué podríamos querer mover el cursor a ciertas posiciones.

¿Cómo definimos estos tipos de cursores para su uso?

A la hora de definir la sentencia utilizada para ejecutar queries se puede definir el tipo de cursor de la siguiente manera:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

¿Podemos usar cualquier tipo de cursor?

Como ya se ha comentado anteriormente, depende de la implementación del driver JDBC, por lo tanto depende de la versión y del sistema al que nos estemos conectando, es decir qué Sistema Gestor de Bases de Datos (SGBD) y qué versión de éste se encuentra en el servidor.

Las propiedades de la conexión nos permiten ver el tipo de cursor del que se dispone la conexión a la base de datos, mediante el uso de los metadatos del driver JDBC. La clase que nos indican los metadatos de la conexión a la base de datos es DatabaseMetaData:

```
DatabaseMetaData metadata = conn.getMetaData();
```

Mediante el uso del método metadata.supportsResultSetType(type) y metadata.supportsResultSetConcurrency(type, concurrency);.

Por ejemplo:

```
boolean isScrollSensitive = metadata.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);
```

Con los metadatos de la base de datos se puede obtener más información, como la versión del SGBD, si soporta las partes del estándar SQL como el GROUP BY, qué versiones de JDBC son compatibles con el servidor o consultar variables de configuración.

Prueba de conceptos vistos hasta ahora:

- Comprobar si nuestro driver y servidor soportan los tipos de cursor cursor TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_INSENSITIVE, y TYPE_FORWARD_ONLY
- Obtener la lista de actores cuyo nombre es 'DAN' e imprimirlo en orden inverso al que se recibe en el resultset.
- Contar el número de filas resultado de una consulta. Por ejemplo: DESCRIBE actor;

Ejercicio:

Cargar todos los actores de la tabla 'actor' de la base de datos sakila, luego hay que pedir al usuario repetidamente un número entero por teclado.

- Si el número está entre 1 y el número de filas cargadas en el resultado imprime por la consola: "El nombre del actor en la fila X es ...".
- Si el número es superior al número de filas, indicar que no existe la fila seleccionada.
- Si el número es menor que 1 terminar del programa.

6.5.2. Uso de consultas pre-compiladas

En ejemplos anteriores hemos visto como realizar consultas con la clase Statement.

Esta clase permite ejecutar sentencias almacenadas en un String. Las consultas vistas hasta ahora consultas (query) de tipo SELECT, pero veremos cómo se pueden ejecutar otro tipos de sentencias.

Ahora vamos a ver un tipo de Statement (implementación del interfaz Statement) que se debe usar cuando se añaden parámetros recibidos por el usuario. Esta clase es el PreparedStatement. Esta clase es la que usan los frameworks de persistencia como spring hibernate de forma transparente por nosotros.

La forma de usarla es parecida a Statement, pero sólo puede haber una consulta, a la que se le pueden cambiar los parámetros. Su uso sería:

```
String query = "select * from actor where first_name= ? ";

PreparedStatement sent= conexion.prepareStatement(query);

sent.setString(1, "ED");

ResultSet rs = sent.executeQuery();
```

Esta clase es la forma más habitual de realizar las consultas parametrizadas por dos razones:

Es segura frente a posibles ataques de inyección de código malicioso.

Imaginemos que en nuestro código, como vimos el otro día, usamos un Statement, y como consulta base el siguiente String: "SELECT * FROM actor WHERE actor_id=".

Si pedimos a un usuario que nos de el id del actor que queremos consultar.

Pedimos a nuestro gran amigo y compañero de grado que pruebe nuestro programa y lo ejecuta pasándonos como parámetro:

"1; DROP TABLE actor; DROP DATABASE sakila"

Al ejecutar la consulta podríamos permitir que algún usuario malintencionado borrara una tabla, la base de datos. O incluso podría ser peor y tomar control de la base de datos usando algo del tipo:

"1; CREATE USER hacker GRANT all ON *.*"

Dado que la instrucción (PreparedStatement) contiene una única consulta, ya que se da al realizar la construcción del objeto, la forma en la que el servidor ejecuta la consulta va a ser siempre la misma, por lo que la planificación de la ejecución de la consulta se puede realizar a priori. Es decir, la ejecución de la consulta sólo se realiza una vez, lo que aumenta la velocidad en sucesivas ejecuciones. En contraposición al uso de un Statement, que al poder ejecutar distintas consultas, el SGBD debe analizar la sentencia SQL que viene en cada ejecución y planificar su ejecución.

Si volvemos a mirar el código necesario para ejecutar una consulta usando un objeto PreparedSatatement, podemos observar que los parámetros de la consulta se ponen dentro del String usando un símbolo de interrogación ?, y con el objeto de tipo String conteniendo la consulta que tendrá tantas interrogaciones como parámetros sean necesarios se construye el objeto de tipo PreparedStatement.

Luego en este objeto antes de ejecutarlo y obtener el objeto con la tabla de resultados ResultSet, se deben añadir los valores de los parámetros. Los parámetros como el resto de índices en JDBC empiezan desde 1.

En nuestro ejemplo vamos a buscar los actores que se llamen ED. Por lo que la interrogación irá en la cláusula WHERE. Se debe saber, que los parámetros van a ser literales o constantes de un tipo de datos

concreto. Es decir, los parámetros serán sustituidos por un entero, String, o incluso imagen, pero no son válidos en nombre de columna o tabla. En la cadena de texto de la consulta la interrogación no está encapsulada por comillas simples aunque el parámetro sea texto o fecha, JDBC se encarga de añadirlos posteriormente en la ejecución.

Habrá que añadir tantos valores de parámetros en el PreparedStament como interrogaciones existan en la consulta. Para ello usaremos el método set del tipo correspondiente, y el índice adecuado para el parámetro.

El uso del ResultSet obtenido no varía respecto a lo visto anteriormente.

Vamos a probar el ejemplo:

```
String query= "select * from actor where first_name= ? ";
PreparedStatement stmt= conexion.prepareStatement(query);
stmt.setString(1, "ED");
ResultSet rs = stmt.executeQuery();
```

6.6. INSERCIÓN DE DATOS Y MODULARIDAD

6.6.1. Manejo de conexión en un programa

Las conexiones a bases de datos son un recurso del ordenador 'caro', por ello debemos poder manejar el número de conexiones que se realizan simultáneamente desde nuestro programa.

Por ello debemos desarrollar nuestro código de forma modular, y centralizar el acceso a conexiones en una sola clase. Para ello crearemos una clase que manejará los accesos a la base de datos. En el caso de una aplicación monohilo como las que vamos a desarrollar en este curso será suficiente que dicha clase tenga una única conexión y conceda acceso a dicha conexión cuando una sub-tarea lo requiera.

En programas más grandes, y sobretodo en programas multihilo, que requieren realizar muchas consultas a los datos, por ejemplo un servidor con múltiples clientes, se suelen usar pools de conexiones.

Un pool de conexiones es un conjunto limitado de conexiones, donde las conexiones pueden ser reutilizadas para diferentes tareas. Este pool es administrado por una clase que va asignando las conexiones a medida que los procesos van solicitando la ejecución de consultas.

Algunos driver JDBC (como la de Oracle) ofrecen la clase PoolDataSource que permite administrar varias conexiones, otros como el de MySQL no la ofreces, pero existen librerías como HikariCP que implementan estos pool de conexiones sobre JDBC.

Ejemplo:

Para ella vamos a crear una clase con un atributo estático (único, de clase) que manejará dicha conexión. Debemos crear varios métodos de clase:

- Un método para configurar la conexión, recibirá el servidor, base de datos, usuario y contraseña de la conexión.
- Un método para solicitar la conexión, debemos crear la conexión y marcar que la conexión está ocupada.
- Un método para liberar la conexión, debe cerrar la conexión y marcarla como libre.

El desarrollo de un pool de conexiones incluye el control de concurrencia y tener una lista de conexiones disponibles.

6.6.2. Ejecución de sentencias DDL desde JDBC

Para la ejecución se sentencias DDL, de manipulación de la estructura de la base de datos, que no devuelven ningún resultset podemos usar el método: executeUpdate.

Este método está disponible en la clase Statement, y por lo tanto, también está implementado en la **subclase PreparedStatement** que vimos en la clase anterior.

Para las sentencias **CREATE** y **ALTER** usaremos este método:

Ejemplo:

Vamos a crear una base de datos de prueba para conectarnos desde Java, la llamaremos test_bbdd_jdbc. Podemos crearla desde el cliente que tengamos (ej: MySQL Workbench).

Desde el proyecto Java en el que hemos creado la clase que maneja la conexión, vamos a crear otra clase que permita crear una tabla. La tabla a crear será la tabla 'clientes'. Con los siguientes campos:

- id_cliente, que es la clave primaria, un entero y autonumérico
- nombre, cadena de texto
- apellidos, cadena de texto
- ciudad, cadena de texto
- distrito, cadena de texto
- direccion, cadena de texto
- telefono_fijo, cadena de texto
- telefono_movil, cadena de texto

La creación de la tabla se realizará en un método estático, donde la consulta sea un String. El método devolverá un booleano, devolverá falso en caso de que haya un error a la hora de crear la tabla; devolverá falso en caso de que no se haya podido crear.

Por último, se ha de invocar al método de crear la tabla desde el programa principal

¿Qué pasará si llamamos dos veces?

Vamos a crear un método de comprobación para comprobar ver si la tabla existe ya en la base de datos:

Usaremos la consulta: SHOW TABLES LIKE 'actor', que debe devolver true o false dependiendo si la tabla existe o no.

6.6.3. Inserción de datos usando JDBC

El método **executeUpdate** puede usarse también para ejecutar sentencias **INSERT** y **UPDATE**.

Para este tipo de sentencias interesa además conocer que existe una sobrecarga del método executeUpdate, que nos permite especificar en el segundo parámetro si se quieren obtener las claves primarias de las filas creadas usando la constante Statement.RETURN_GENERATED_KEYS. Estas claves generadas las necesitaremos cuando creemos filas nuevas y la tabla contenga una clave primaria autonumérica. Este método como resultado devuelve el número de filas afectadas por la consulta.

Si posteriormente queremo revisar las claves primarias generadas por las sentencias SQL podemos obtenerlas como un ResultSet de la siguiente manera:

ResultSet rs = stmt.getGeneratedKeys();

Ejercicio

Se quiere insertar una fila con una sentencia INSERT en la tabla que creamos en el ejercicio anterior. Para ello vamos a implementar un programa que pida al usuario los campos necesarios.

Hay que recordar que si queremos que el SGBD genere automáticamente la clave primaria, no debemos incluir en la sentencia INSERT la columna de la clave primaria. Se quiere hacer usando la clase de manejo de conexión que se creó en el ejemplo anterior.

1) Se pide, implementar la inserción de una fila en la tabla, usando la clase Statement. En este caso se debe crear la sentencia en un String y concatenar los valores necesarios.

2) Implementar un método que introduzca una fila en la tabla clientes usando PreparedStatement. Hay que acordarse que se deben declarar los parámetros con símbolos de interrogación, y posteriormente introducir los parámetros mediante los métodos del PreparedStatement.