

# Ministerio de Producción

## Secretaría de Industria y Servicios

### Subsecretaría de Servicios Tecnológicos y Productivos

y

# Ministerio de Educación y Deportes

A través del

**inet** | Instituto Nacional de  
Educación Tecnológica

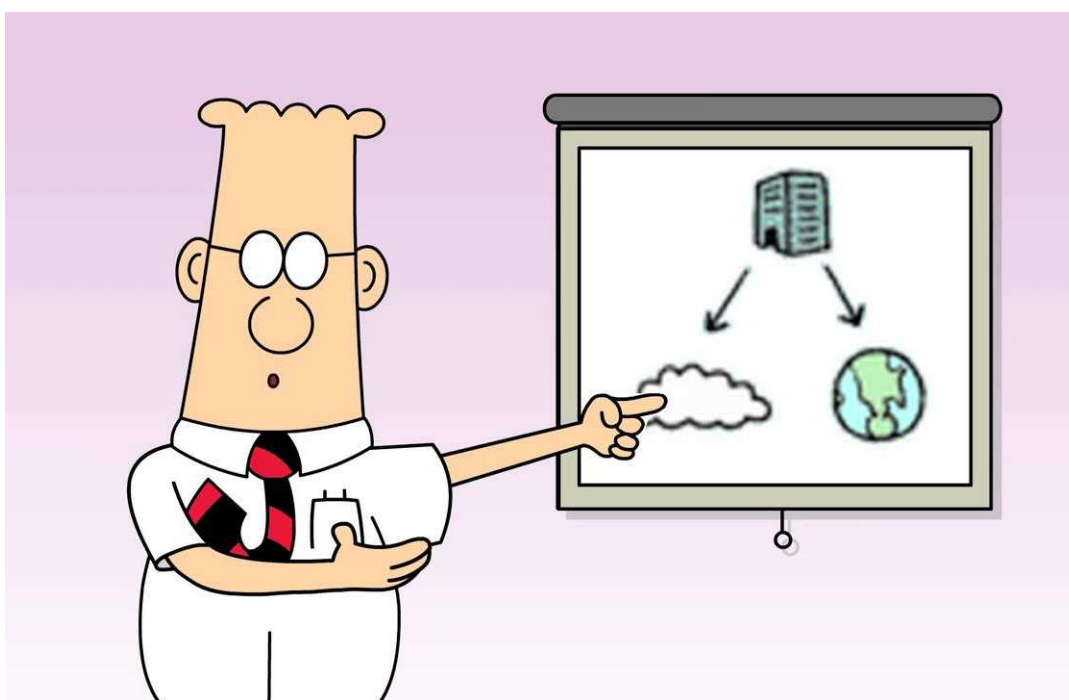


## Analistas del Conocimiento

### Dimensión Programador

# Apunte del Módulo

## Base de Datos



## Tabla de Contenidos

<b>TABLA DE CONTENIDOS .....</b>	<b>3</b>
<b>INTRODUCCIÓN .....</b>	<b>5</b>
<b>DBMS – DATA BASE MANAGEMENT SYSTEM .....</b>	<b>7</b>
CONCEPTO .....	7
PROPÓSITO DE LOS DBMS (DATABASE MANAGEMENT SYSTEM) .....	7
FUNCIONES DE LOS DBMS .....	10
<b>TECNOLOGÍA DE BASE DE DATOS VS. TECNOLOGÍA TRADICIONAL.....</b>	<b>11</b>
<b>SISTEMAS DE BASE DE DATOS DISTRIBUIDOS Y CENTRALIZADOS.....</b>	<b>14</b>
<b>CLASIFICACIÓN DE BASES DE DATOS .....</b>	<b>15</b>
BASES DE DATOS NOSQL.....	21
BASES DE DATOS PARA SOPORTE A LA TOMA DE DECISIONES .....	23
<b>MODELO ENTIDAD RELACIÓN .....</b>	<b>25</b>
CASO DE ESTUDIO .....	25
CONCEPTOS BÁSICOS DEL MODELO DE ENTIDAD RELACIÓN .....	26
ENTIDADES .....	26
ATRIBUTOS .....	27
REGLAS DE INTEGRIDAD .....	30
RELACIONES.....	31
<b>MODELO LÓGICO RELACIONAL.....</b>	<b>36</b>
ESTRUCTURA DE DATOS RELACIONAL.....	36
INTEGRIDAD EN LAS BASES DE DATOS RELACIONALES.....	38
REGLAS DE INTEGRIDAD PARA CLAVE PRIMARIA .....	39
REGLA DE INTEGRIDAD EN LAS RELACIONES.....	40
REGLAS DE INTEGRIDAD PARA CLAVES AJENAS .....	40
<b>MOTORES DE BASES DE DATOS RELACIONALES .....</b>	<b>41</b>
ALTERNATIVAS DE MOTORES DE BASES DE DATOS RELACIONALES .....	42
TRANSACCIONES .....	43
¿POR QUÉ ES NECESARIA LA RECUPERACIÓN? .....	44
ACID, LETRA POR LETRA EN LAS TRANSACCIONES.....	45
ESTADOS DE UNA TRANSACCIÓN Y OPERACIONES ADICIONALES .....	46
ÍNDICES .....	48
PROCEDIMIENTOS ALMACENADOS Y FUNCIONES .....	53
VISTAS .....	54
TRIGGERS O DISPARADORES .....	54
<b>LENGUAJE DE CONSULTA SQL .....</b>	<b>55</b>

<b>DEFINICIÓN DE DATOS .....</b>	<b>56</b>
<b>MANIPULACIÓN DE DATOS.....</b>	<b>62</b>
SELECT .....	62
CONSULTA MULTI-TABLAS. ....	65
SUBCONSULTAS .....	69
UPDATE .....	70
INSERT .....	70
DELETE.....	71
<b><u>SEGURIDAD .....</u></b>	<b><u>72</u></b>
ADMINISTRACIÓN DE CUENTAS DE USUARIO EN MySQL.....	73
CONEXIONES SEGURAS CON MySQL.....	74
<b><u>MODELOS DE PERSISTENCIA .....</u></b>	<b><u>75</u></b>
<b>ORM .....</b>	<b>76</b>
<b>HIBERNATE ORM .....</b>	<b>77</b>
EL ARCHIVO DE CONFIGURACIÓN .....	77
EL ARCHIVO DE MAPEO XML .....	81
MAPEO DE RELACIONES .....	85
OPERACIONES SOBRE LAS ENTIDADES .....	91
<b>FUENTES DE INFORMACIÓN .....</b>	<b>96</b>
<b>ÍNDICE DE FIGURAS .....</b>	<b>96</b>

## Introducción

Durante las últimas cuatro décadas del siglo veinte, el uso de las bases de datos creció en todas las empresas. En los primeros días, muy pocas personas interactuaban directamente con los sistemas de bases de datos, aunque sin darse cuenta interactuaban indirectamente con bases de datos—con informes impresos como los extractos de las tarjetas de crédito, o mediante agentes como los cajeros de los bancos y los agentes de reservas de las líneas aéreas. Después llegaron los cajeros automáticos y permitieron a los



usuarios interactuar directamente con las bases de datos. Pero a partir de la revolución de Internet a finales de los años noventa aumentó significativamente el acceso directo del usuario a las bases de datos. Las organizaciones convirtieron sus procesos, permitieron que sus operaciones se pudieran realizar por medio de la Web. Por ejemplo, cuando se accede a una librería en línea y se busca en una colección de libros o de música, se está accediendo a datos almacenados en una base de datos. Cuando se realiza un pedido en línea, el pedido se almacena en una base de datos. Cuando se accede al sitio Web de un banco y se consultan el estado de la cuenta y los movimientos, la información se recupera del sistema de información del banco, que utiliza bases de datos.

Cuando se accede a un sitio Web, puede que se recupere información personal de una base de datos para seleccionar los anuncios que se deben mostrar. Más aún, los datos sobre todos los accesos Web que realizan los usuarios dentro de un sistema pueden almacenarse en una base de datos. Así, aunque las interfaces de usuario ocultan los detalles del acceso a las bases de datos, y la mayoría de las personas, ni siquiera es consciente que están interactuando con una base de datos, el acceso a las bases de datos forma actualmente una parte esencial de la vida de todos.

**Una base de datos es una colección de datos relacionados.** Con la palabra datos nos referimos a los hechos conocidos que se pueden grabar y que tienen un significado implícito. Por ejemplo, los nombres, números de teléfono y direcciones de las personas que conoce. Puede tener todos estos datos grabados en un libro de direcciones indexado o los puede tener almacenados en el disco duro de un computador. Esta colección de datos con un significado implícito es una base de datos.

La definición anterior de base de datos es muy genérica; por ejemplo, podemos pensar que la colección de palabras que compone esta página de texto es una colección de datos relacionados y que, por tanto, constituye una base de datos. No obstante, el uso común del término base de datos es normalmente más restringido.

Una base de datos tiene las siguientes propiedades implícitas:

- Una base de datos representa algún **aspecto del mundo real**, lo que en ocasiones se denomina mini-mundo o universo de discurso. Los cambios introducidos en el mini-mundo se reflejan en la base de datos.
- Una base de datos es una **colección de datos lógicamente coherente** con algún tipo de significado inherente. No es correcto denominar base de datos a un surtido aleatorio de datos.

- Una base de datos se diseña, construye y rellena con datos para un **propósito específico**. Dispone de un grupo pretendido de usuarios y algunas aplicaciones preconcebidas en las que esos usuarios están interesados.

En otras palabras, una base de datos tiene algún origen del que se derivan los datos, algún grado de interacción con eventos del mundo real y un público que está activamente interesado en su contenido:

Los usuarios finales de una base de datos pueden efectuar transacciones comerciales (por ejemplo, un cliente que compra una cámara fotográfica) o se pueden producir eventos (por ejemplo, un empleado tiene un hijo) que provoquen un cambio en la información almacenada en la base de datos.

Con el objetivo de que una base de datos sea en todo momento precisa y fiable, debe ser un **reflejo exacto** del mini-mundo que representa; por consiguiente, en la base de datos deben reflejarse los cambios tan pronto como sea posible.

Una base de datos puede ser de **cualquier tamaño y complejidad**. Por ejemplo, la lista de nombres y direcciones a la que nos referíamos anteriormente puede constar de únicamente unos cuantos cientos de registros, cada uno de ellos con una estructura sencilla. Por el contrario, el catálogo computarizado de una gran biblioteca puede contener medio millón de entradas organizadas en diferentes categorías (por los apellidos del autor principal, por el tema, por el título del libro), y cada categoría ordenada alfabéticamente.

Amazon.com®, es un buen ejemplo de una gran base de datos comercial: contiene datos de más de 20 millones de libros, CDs, vídeos, DVDs, juegos, ropa y otros productos. La base de datos ocupa más de 2 terabytes (un terabyte equivale a  $10^{12}$  bytes de almacenamiento) y se almacena en 200 computadores diferentes (denominados servidores). Cada día acceden a Amazon.com aproximadamente 15 millones de visitantes que utilizan la base de datos para hacer compras. La base de datos se actualiza continuamente a medida que se añaden libros y otros productos nuevos al inventario, mientras que el stock se actualiza al tiempo que se tramitan las compras. Alrededor de 100 personas son las responsables de mantener actualizada la base de datos de Amazon.

Una base de datos se puede generar y mantener manualmente o estar automatizada. Por ejemplo, el catálogo de libros de una biblioteca es una base de datos que se puede crear y mantener de forma manual. Una base de datos automatizada se encuentra almacenada en un dispositivo de almacenamiento como un disco duro y se crea y mantiene a través de un grupo de aplicaciones específicamente diseñadas.

## DBMS – Data Base Management System

---

### Concepto

Una base de datos, por sí misma, no es más que una forma de almacenamiento de datos; por lo tanto, para poder acceder a dichos datos existe un conjunto de programas que facilitan la interacción con dichos datos.

El objetivo principal de este conjunto de programas denominado: Sistema de Administración de Base de Datos (DBMS, en inglés DataBase Management System) es proporcionar una forma de almacenar y recuperar la información de una base de datos de manera que sea práctica y eficiente.

Éstos sistemas de administración de BD (Base de Datos) están diseñados para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de administración de bases de datos deben garantizar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o de los intentos de acceso no autorizados. Si los datos van a ser compartidos entre diferentes usuarios, el sistema debe evitar posibles resultados anómalos.

Podemos definir un sistema de administración de datos **como una colección de programas que permite a los usuarios crear y mantener una base de datos**. El DBMS es un sistema de software de propósito general que facilita los procesos de definición, construcción, manipulación y compartición de bases de datos entre varios usuarios y aplicaciones. Definir una base de datos implica especificar los tipos de datos, estructuras y restricciones de los datos que se almacenarán en la misma.

### Propósito de los DBMS (DataBase Management System)

Las bases de datos y los sistemas de administración de bases de datos surgieron en respuesta a los primeros métodos de gestión informatizada de los datos comerciales. A modo de ejemplo de dichos métodos, típicos de los años sesenta, considérese parte de una entidad bancaria que, entre otros datos, guarda información sobre todos los clientes y todas las cuentas de ahorro. Una manera de guardar la información en la computadora es almacenarla en archivos del sistema operativo. Para permitir que los usuarios manipulen la información, el sistema tiene varios programas de aplicación que gestionan los archivos, incluyendo programas para:

- Efectuar cargos o abonos en las cuentas.
- Añadir cuentas nuevas.
- Calcular el saldo de las cuentas.
- Generar los extractos mensuales.

Estos programas de aplicación los han escrito programadores de sistemas en respuesta a las necesidades del banco.

Se añaden nuevos módulos al sistema según surgen las necesidades. Por ejemplo, supóngase que un banco decide ofrecer además de caja de ahorro, cuentas corrientes. En consecuencia, se crean nuevos archivos permanentes que contienen información acerca de todas las cuentas corrientes abiertas en el banco y puede que haya que escribir nuevos programas de aplicación

para afrontar situaciones que no se dan en las cajas de ahorro, como los descubiertos<sup>11</sup>. Así, con el paso del tiempo, se añaden más archivos y programas de aplicación al sistema.

Los sistemas operativos convencionales soportan este **sistema de procesamiento de archivos** típico. El sistema almacena los registros permanentes en varios archivos y necesita diferentes programas de aplicación para extraer y añadir a los archivos correspondientes. Antes de la aparición de las bases de datos, las organizaciones normalmente almacenaban la información en sistemas de este tipo.

Guardar la información de la organización en un sistema de procesamiento de archivos tiene una serie de inconvenientes importantes:

- **Redundancia e inconsistencia de los datos.** Debido a que los archivos y programas de aplicación los crean diferentes programadores en el transcurso de un período de tiempo, es probable que los diversos archivos tengan estructuras diferentes y que los programas estén escritos en varios lenguajes de programación diferentes. Además, puede ser que la información esté duplicada en varios lugares (archivos). Por ejemplo, la dirección y el número de teléfono de un cliente dado pueden aparecer en un archivo que contenga registros de cuentas de ahorros y en un archivo que contenga registros de cuentas corrientes. Esta redundancia conduce a costos de almacenamiento y de acceso más elevados. Además, puede dar lugar a la **inconsistencia de los datos**; es decir, puede ser que las diferentes copias de los mismos datos no coincidan. Por ejemplo, es posible que el cambio en la dirección de un cliente esté reflejado en los registros de las cuentas de ahorro, pero no en el resto del sistema.
- **Dificultad en el acceso a los datos.** Supóngase que uno de los empleados del banco necesita averiguar los nombres de todos los clientes que viven en domicilio con un mismo código postal. El empleado pide al departamento de procesamiento de datos que genere esa lista. Debido a que esta petición no fue prevista por los diseñadores del sistema original, no hay un programa de aplicación a mano para satisfacerla. Hay, sin embargo, un programa de aplicación que genera la lista de todos los clientes. El empleado del banco tiene ahora dos opciones: obtener la lista de *todos* los clientes y extraer manualmente la información que necesita, o bien pedir a un programador de sistemas que escriba el programa de aplicación necesario. Ambas alternativas son obviamente insatisfactorias.  
Supóngase que se escribe el programa y que, varios días más tarde, el mismo empleado necesita reducir esa lista para que incluya únicamente a aquellos clientes que tengan una cuenta con saldo igual o superior a \$10.000. Como se puede esperar, no existe ningún programa que genere tal lista. De nuevo, el empleado tiene que elegir entre dos opciones, ninguna de las cuales es satisfactoria. La cuestión aquí es que los entornos de procesamiento de archivos convencionales no permiten recuperar los datos necesarios de una forma práctica y eficiente. **Hacen falta sistemas de recuperación de datos más adecuados para el uso general.**
- **Aislamiento de datos.** Como los datos están dispersos en varios archivos, y los archivos pueden estar en diferentes formatos, es difícil escribir nuevos programas de aplicación para recuperar los datos correspondientes.

<sup>11</sup> Descubierta de una cuenta corriente: el término se refiere a la posibilidad de utilizar un determinado monto asignado en la cuenta, para hacer operaciones como extracciones de dinero o pago de servicios, sin tener saldo real en la cuenta.



- **Problemas de integridad.** Los valores de los datos almacenados en la base de datos deben satisfacer ciertos tipos de **restricciones de consistencia**. Por ejemplo, el saldo de ciertos tipos de cuentas bancarias no puede nunca ser inferior a una cantidad predeterminada (por ejemplo, \$25). Los desarrolladores hacen cumplir esas restricciones en el sistema añadiendo el código correspondiente en los diversos programas de aplicación. Sin embargo, cuando se añaden nuevas restricciones, es difícil cambiar los programas para hacer que se cumplan. El problema se complica cuando las restricciones implican diferentes elementos de datos de diferentes archivos.
- **Problemas de atomicidad.** Los sistemas informáticos, como cualquier otro dispositivo mecánico o eléctrico, está sujeto a fallos. En muchas aplicaciones es crucial asegurar que, si se produce algún fallo, los datos se restauren al estado consistente que existía antes del fallo. Considérese un programa para transferir \$10.000 desde una cuenta A a una Cuenta B. Si se produce un fallo del sistema durante la ejecución del programa, es posible que los \$10.000 fueran retirados de la cuenta A, pero no acreditados en la cuenta B, dando lugar a un estado inconsistente de la base de datos. Evidentemente, resulta esencial para la consistencia de la base de datos que tengan lugar tanto el débito (descuento del saldo de la cuenta A) como el crédito (carga en la Cuenta B), o que no tenga lugar ninguno. Es decir, la transferencia de fondos debe ser *atómica*—debe ocurrir en su totalidad o no ocurrir en absoluto. Resulta difícil asegurar la atomicidad en los sistemas convencionales de procesamiento de archivos.
- **Anomalías en el acceso concurrente.** Para aumentar el rendimiento global del sistema y obtener una respuesta más rápida, muchos sistemas permiten que varios usuarios actualicen los datos simultáneamente. En realidad, hoy en día, los principales sitios de comercio electrónico de Internet pueden tener millones de accesos diarios de compradores a sus datos. En tales entornos es posible la interacción de actualizaciones concurrentes y puede dar lugar a datos inconsistentes. Considérese una cuenta bancaria A, que contenga \$5000. Si dos clientes retiran fondos (por ejemplo, \$500 y \$1000, respectivamente) de la cuenta A aproximadamente al mismo tiempo, el resultado de las ejecuciones concurrentes puede dejar la cuenta en un estado incorrecto (o inconsistente). Supóngase que los programas que se ejecutan para cada retiro de dinero leen el saldo anterior, reducen su valor en el importe que se retira y luego escriben el resultado. Si los dos programas se ejecutan concurrentemente, pueden leer el valor \$500, y escribir después \$4500 y \$4000, respectivamente. Dependiendo de cuál escriba el valor en último lugar, la cuenta puede contener \$4500 o \$4000, en lugar del valor correcto, \$3500. Para protegerse contra esta posibilidad, el sistema debe mantener alguna forma de supervisión. Pero es difícil ofrecer supervisión, ya que muchos programas de aplicación diferentes que **no se han coordinado con anterioridad** pueden tener acceso a los datos.
- **Problemas de seguridad.** No todos los usuarios de un sistema de bases de datos deben poder acceder a todos los datos. Por ejemplo, en un sistema bancario, el personal de nóminas (Empleados de la empresa), sólo necesita ver la parte de la base de datos que contiene información acerca de los diferentes empleados del banco. No necesitan tener acceso a la información acerca de las cuentas de clientes. Pero, como los programas de aplicación se añaden al sistema de procesamiento de datos a medida que se requieren, es difícil hacer cumplir tales restricciones de seguridad.  
Estas dificultades, entre otras, motivaron el desarrollo de las bases de datos y los

sistemas de administración de las mismas.

## Funciones de los DBMS

- **Metadatos:** la **definición o información descriptiva** de una base de datos, relacionada con su estructura y los datos que contiene, también se almacena, en forma de catálogo o diccionario de la base de datos; es lo que se conoce como **metadatos**.
- **Construcción de la base de datos:** es el proceso consistente en almacenar los datos en algún medio de almacenamiento controlado por el **DBMS** (Sistema de Administración de Base de Datos).
- **Manipulación de una base de datos:** son aquellas funciones como la consulta de la base de datos para recuperar datos específicos, actualizar la base de datos para reflejar los cambios introducidos en el mini-mundo y generar informes a partir de los datos.
- **Compartir una base de datos:** permite que varios usuarios y programas accedan a la base de datos de forma simultánea.
- **Consultas:** una **aplicación** accede a la base de datos enviando **consultas o solicitudes** de datos al DBMS. Una **consulta** normalmente provoca la recuperación de algunos datos.
- **Transacciones:** una **transacción** puede provocar la lectura o la escritura de algunos datos en la base de datos.
- **Protección:** incluye la protección del sistema contra el funcionamiento defectuoso del hardware o el software (caídas) y la protección de la seguridad contra el acceso no autorizado o malintencionado.
- **Mantenimiento:** una gran base de datos típica puede tener un ciclo de vida de muchos años, por lo que el DBMS debe ser capaz de mantener el sistema de bases de datos permitiendo que el sistema evolucione según cambian los requisitos con el tiempo.

No es necesario utilizar software DBMS de propósito general para implementar una base de datos computarizada. Podríamos escribir nuestro propio conjunto de programas para crear y mantener la base de datos; en realidad, podríamos crear nuestro propio software DBMS de propósito especial. En cualquier caso (utilicemos o no un DBMS de propósito general), normalmente tenemos que implantar una cantidad considerable de software complejo. De hecho, la mayoría de los **DBMS son sistemas de software muy complejos**. El siguiente esquema muestra la estructura de un sistema que utiliza un DBMS.

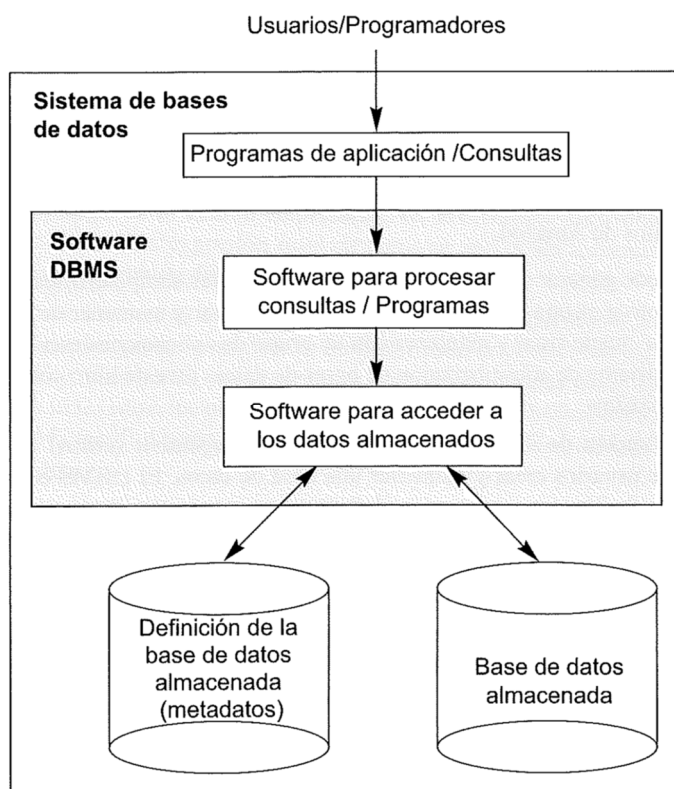


Figura 1: Estructura de un Sistema que utiliza DBMS

## Tecnología de Base de Datos vs. Tecnología Tradicional

Unas cuantas características distinguen la tecnología de bases de datos de la tecnología tradicional de programación con archivos. En el procesamiento tradicional de archivos, cada usuario define e implementa los archivos necesarios para una aplicación concreta como parte de la programación de esa aplicación. Por ejemplo, un usuario, la *oficina de notificación de calificaciones*, puede encargarse del mantenimiento de un archivo con los estudiantes y sus calificaciones. Los programas encargados de imprimir el certificado de estudios de un estudiante e introducir nuevas calificaciones en el archivo se implementan como parte de la aplicación.

Un segundo usuario, la *oficina de contabilidad*, puede encargarse del seguimiento de las cuotas de los estudiantes y sus pagos. Aunque ambos usuarios están interesados en datos relacionados con los estudiantes, cada uno mantiene archivos separados (y programas para la manipulación de esos archivos), porque cada uno requiere algunos datos que no están disponibles en los archivos del otro. Esta redundancia en la definición y el almacenamiento de datos da como resultado un derroche de espacio de almacenamiento y reiterados esfuerzos por mantener al día datos comunes.

Con las bases de datos se mantiene un único almacén de datos, que se define una sola vez, y al que acceden varios usuarios. En los sistemas de archivos cada aplicación tiene libertad para asignar un nombre independientemente a los elementos de datos. Por el contrario, en una base de datos, los nombres o etiquetas de los datos se definen una vez, y son utilizados para

consultas, transacciones y aplicaciones. Las principales características de las bases de datos frente a la de procesamiento de archivos son las siguientes:

- **Naturaleza auto descriptiva de un sistema de bases de datos.**

Una característica fundamental de la tecnología de bases de datos es que el sistema de bases de datos no sólo contiene la propia base de datos, sino también una completa definición o descripción de la estructura de la base de datos y sus restricciones. Esta definición se almacena en el catálogo DBMS, que contiene información como la estructura de cada archivo, el tipo y el formato de almacenamiento de cada elemento de datos, y distintas restricciones de los datos. La información almacenada en el catálogo se denomina, como vimos anteriormente: **metadatos** y describe la estructura de la base de datos principal. El software DBMS y los usuarios de la base de datos utilizan el catálogo cuando necesitan información sobre la estructura de la base de datos. Un paquete de software DBMS de propósito general no se escribe para una aplicación de base de datos específica. Por consiguiente, debe referirse al catálogo para conocer la estructura de los archivos de una base de datos específica, como el tipo y el formato de los datos a los que accederá. El software DBMS debe funcionar igual de bien con **cualquier cantidad de aplicaciones de bases de datos** (por ejemplo, la base de datos de una universidad, la base de datos de un banco o la base de datos de una empresa), siempre y cuando la definición de la base de datos esté almacenada en el catálogo.

- **Aislamiento entre programas y datos, y abstracción de datos.**

En el procesamiento de archivos tradicional, la estructura de los archivos de datos está incrustada en las aplicaciones, por lo que los cambios que se introducen en la estructura de un archivo pueden obligar a realizar *cambios en todos los programas* que acceden a ese archivo. Por el contrario, los programas que acceden a un DBMS no necesitan esos cambios en la mayoría de los casos. La estructura de los archivos de datos se almacena en el catálogo DBMS, independientemente de los programas de acceso. Llamaremos a esta propiedad **independencia programa-datos**.

- **Soporte de varias vistas de los datos.**

Normalmente una base de datos tiene muchos usuarios, cada uno de los cuales puede necesitar una perspectiva o vista diferente de la base de datos. Una **vista** puede ser un subconjunto de la base de datos o puede contener **datos virtuales** derivados de los archivos de la base de datos pero que no están explícitamente almacenados.

Algunos usuarios no tienen la necesidad de preocuparse por si los datos a los que se refieren están almacenados o son derivados. Un DBMS multiusuario cuyos usuarios tienen variedad de diferentes aplicaciones debe ofrecer facilidades para definir varias vistas.

- **Compartición de datos y procesamiento de transacciones multiusuario.**

Un DBMS multiusuario, como su nombre indica, debe permitir que varios usuarios puedan acceder a la base de datos al mismo tiempo. Esto es esencial, si los datos destinados a varias aplicaciones serán integrados y mantenidos en una sola base de datos. El DBMS debe incluir software de control de la concurrencia para que esos varios usuarios que intentan actualizar los mismos datos, en el mismo momento, lo hagan de un modo controlado para que el resultado de la actualización sea correcto. Por ejemplo, si varios agentes de viajes intentan reservar un asiento en un vuelo, el DBMS debe garantizar que en cada momento sólo un agente tiene acceso

a la asignación de ese asiento para un pasajero. Estos tipos de aplicaciones se denominan, por lo general, aplicaciones de procesamiento de transacciones en línea (OLTP, *online transaction processing*). Un papel fundamental del software DBMS multiusuario es garantizar que las transacciones concurrentes operan correcta y eficazmente.

El concepto de transacción es cada vez más importante para las aplicaciones de bases de datos. Una transacción es un **programa en ejecución o proceso** que incluye uno o más accesos a la base de datos, como la lectura o la actualización de los registros de la misma. Se supone que una transacción ejecuta un acceso lógicamente correcto a la base de datos si lo ejecutó íntegramente sin interferencia de otras transacciones. El DBMS debe implementar varias propiedades de transacción. La propiedad aislamiento garantiza que parezca que cada transacción se ejecuta de forma aislada de otras transacciones, aunque puedan estar ejecutándose cientos de transacciones al mismo tiempo. La propiedad de atomicidad garantiza que se ejecuten o todas o ninguna de las operaciones de bases de datos de una transacción.

## Sistemas de Base de Datos Distribuidos y Centralizados

Cuando una aplicación interactúa con una base de datos, se comunica con un servidor de base de datos, el cuál puede estar instalado localmente y ejecutándose en la misma computadora en donde se ejecuta la aplicación, o de lo contrario, puede estar instalado en otra máquina comúnmente llamada servidor de base de datos (porque presta servicio, de acuerdo a las peticiones que la aplicación realice). Cuando el servidor de base de datos se encuentra instalado en un servidor externo, la comunicación se logra utilizando una red, que requiere un protocolo<sup>2</sup> para poder comunicarse; el protocolo que generalmente se utiliza se denomina TCP/IP. Este protocolo queda fuera del alcance de este material, pero podemos explicarlo fácilmente si pensamos en Internet: hoy en día, todos estamos acostumbrados a utilizar computadoras con internet y buscar información en google, buscar noticias de otros países, conectarnos a las redes sociales o jugar en red con otras personas que están en otra parte del mundo, ¿Cómo se logra esto?



Cada computadora se comunica con otra computadora utilizando un número identificador para cada una, denominado IP, es simplemente como un documento de la computadora, que la identifica y es necesario para lograr comunicarse con otra, de esta manera si entramos a la página de google: [www.google.com](http://www.google.com), ¿Cómo hacemos para poder ver la página del buscador?, simplemente nuestra computadora se conecta por medio de la red con otra computadora que cumple la función de servidor, ya que responde a las peticiones de los usuarios: al entrar a [www.google.com.ar](http://www.google.com.ar), el servidor detecta que nosotros queremos acceder a la página del buscador y simplemente nos brinda la página.

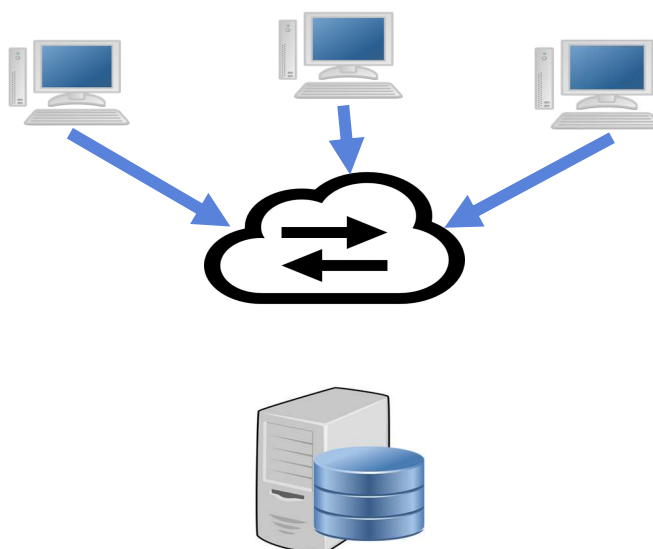


Figura 2: Esquema de un Sistema de Base de Datos Distribuida

<sup>2</sup> **Protocolo:** En informática y telecomunicación, un protocolo de comunicaciones es un sistema de reglas que permiten que dos o más entidades de un sistema de comunicación se comuniquen entre ellas para transmitir información por medio de cualquier tipo de variación de una magnitud física. Se trata de las reglas o el estándar que define la sintaxis, semántica y sincronización de la comunicación, así como también los posibles métodos de recuperación de errores. Los protocolos pueden ser implementados por hardware, por software, o por una combinación de ambos.

## Clasificación de Bases de Datos

---

En la actualidad, las empresas manejan una gran cantidad de datos que debe tener almacenados en una base de datos para poder consultarlos o modificarlos mediante un producto de software o sistema. Sin esta funcionalidad, resultaría imposible administrar en su totalidad los datos que usa la empresa y se perdería mucho tiempo y dinero.

Uno de los pasos cruciales en la construcción de una aplicación que maneje una base de datos, es sin duda, el diseño de la base de datos, en donde lo más importante es la correcta elección del **Modelo de Datos**.

No importa si nuestra base de datos tiene sólo 20 registros o miles, es importante asegurarnos de que está correctamente diseñada para que tenga eficiencia y pueda seguir siendo utilizada a lo largo del tiempo.

Son muchas las consideraciones a tener en cuenta al momento de hacer el diseño de la base de datos, entre las que se pueden mencionar:

- La velocidad de acceso
- El tamaño de la base de datos
- El tipo de los datos
- Facilidad de acceso a los datos
- Facilidad para extraer los datos requeridos

Un **modelo de datos** es un conjunto de conceptos utilizados para organizar los datos de interés y describir su estructura en forma comprensible para un sistema informático. Un modelo de datos es básicamente una "descripción" de algo conocido como contenedor de datos (donde se guarda la información), así como de los métodos para almacenar y recuperar información de esos contenedores. Los modelos de datos son abstracciones que permiten la implementación de un sistema eficiente de base de datos; por lo general se refieren a algoritmos y conceptos matemáticos asociados.

El diseño de bases de datos es el proceso por el que se determina la organización de una base de datos, incluidos su estructura, contenido y las aplicaciones que se han de desarrollar. Durante mucho tiempo, el diseño de bases de datos fue considerado una tarea para expertos: más un arte que una ciencia. Sin embargo, se ha progresado mucho en el diseño de bases de datos y éste se considera ahora una disciplina estable, con métodos y técnicas propios. Debido a la creciente aceptación de las bases de datos por parte de la industria y el gobierno en el plano comercial y a una variedad de aplicaciones científicas y técnicas, el diseño de bases de datos desempeña un papel central en el empleo de los recursos de datos en la mayoría de las organizaciones. El diseño de bases de datos ha pasado a constituir parte de la formación general de los informáticos, en el mismo nivel que la capacidad de construir algoritmos usando un lenguaje de programación convencional.

Un modelo de datos es entonces una serie de conceptos que puede utilizarse para describir un conjunto de datos y las operaciones para manipularlos. Hay dos tipos de modelos de datos: los **modelos conceptuales** y los **modelos lógicos**. Los *modelos conceptuales* se utilizan para representar la realidad a un alto nivel de abstracción. Mediante los modelos conceptuales se puede construir una descripción de la realidad fácil de entender. En los *modelos lógicos*, las descripciones de los datos tienen una correspondencia con la estructura física de la base de datos. En el diseño de bases de datos se usan primero los modelos conceptuales para lograr una

Módulo Base de Datos

Versión 1.1 – Liberada el 08/02/2017



descripción de alto nivel de la realidad, y luego se transforma el esquema conceptual en un esquema lógico. El motivo de realizar estas dos etapas es la dificultad de abstraer la estructura de una base de datos que presente cierta complejidad.

Un **esquema** es un conjunto de representaciones lingüísticas o gráficas que describen la estructura de los datos de interés. Los modelos conceptuales deben ser buenas herramientas para representar la realidad, por lo que deben poseer las siguientes cualidades:

- **Expresividad:** deben tener suficientes conceptos para expresar perfectamente la realidad.
- **Simplicidad:** deben ser simples para que los esquemas sean fáciles de entender.
- **Minimalidad:** cada elemento de la realidad está representado una sola vez en el esquema.
- **Formalidad:** todos los conceptos deben tener una interpretación única, precisa y bien definida.

En general, un modelo no es capaz de expresar todas las propiedades de una realidad determinada, por lo que hay que añadir aserciones que complementen el esquema. A medida que han ido evolucionando el software y el hardware, las posibilidades y las demandas de los usuarios han ido creciendo y paralelamente los modelos de datos fueron enriqueciéndose y salvando carencias de sus predecesores. Cronológicamente, podemos clasificarlos de la siguiente forma:

- Basados en sistemas de archivos convencionales
- Jerárquico
- De Red
- Relacional
- Orientado a Objetos
- Orientado a Documentos
- Multidimensional

Los modelos de datos primitivos se usaron durante la década de los 70, cuando aún no se utilizaban las técnicas de bases de datos. Los objetos se representaban como registros organizados en ficheros, y las relaciones mediante referencias explícitas a otros registros en algún campo del mismo. Los lenguajes de manipulación dependen por entero de la organización física de los datos y las operaciones básicas de lectura y escritura son específicas para la aplicación implementada.

Para garantizar, o al menos mejorar, las independencias de las aplicaciones frente a los datos aparecen los primeros sistemas de gestión de bases de datos basados en lo que ahora llamamos modelos de datos clásicos. Los primeros en aparecer fueron el jerárquico y la red, cuyos nombres muestran cual es la estructura de datos subyacente en los modelos. Los elementos siguen siendo representados por registros, pero las relaciones entre elementos se expresan, con ciertas limitaciones implícitas del modelo, mediante la estructura en que se basan.



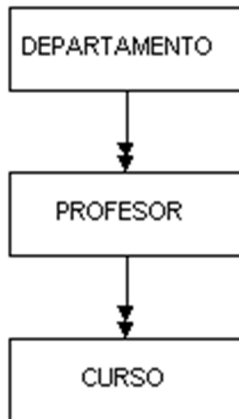
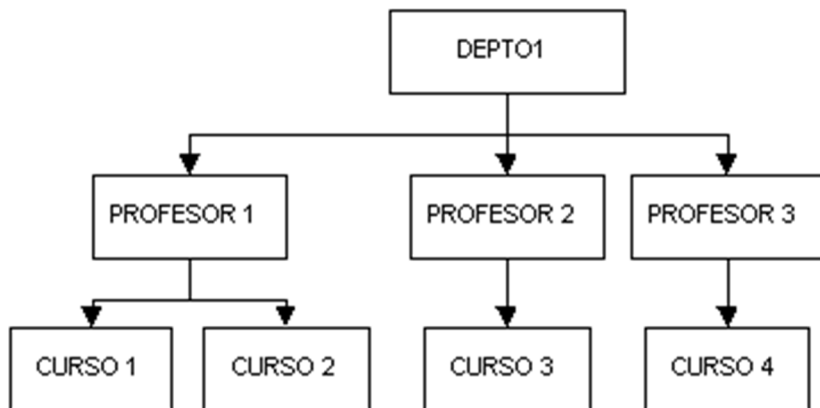
**Estructura lógica****Ejemplo de base de datos**

Fig. 3: El modelo de datos jerárquico

En el **modelo jerárquico** los datos se organizan en una forma similar a un árbol (visto al revés), en donde un *nodo padre* de datos puede tener varios *hijos*. El nodo que no tiene padres es llamado raíz, y a los nodos que no tienen hijos se los conoce como hojas.

Las bases de datos jerárquicas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de datos y datos muy compartidos permitiendo crear estructuras estables y de gran rendimiento. Una de las principales limitaciones de este modelo es su incapacidad de representar eficientemente la redundancia de datos.

Por otro lado, el **modelo de red** es ligeramente distinto del jerárquico; su diferencia fundamental es la modificación del concepto de nodo: se permite que un mismo nodo tenga varios padres (posibilidad no permitida en el modelo jerárquico). Representó una gran mejora con respecto al modelo jerárquico ya que ofrecía una solución eficiente al problema de redundancia de datos, no obstante, la dificultad que significa administrar los datos en una base de datos de red ha provocado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

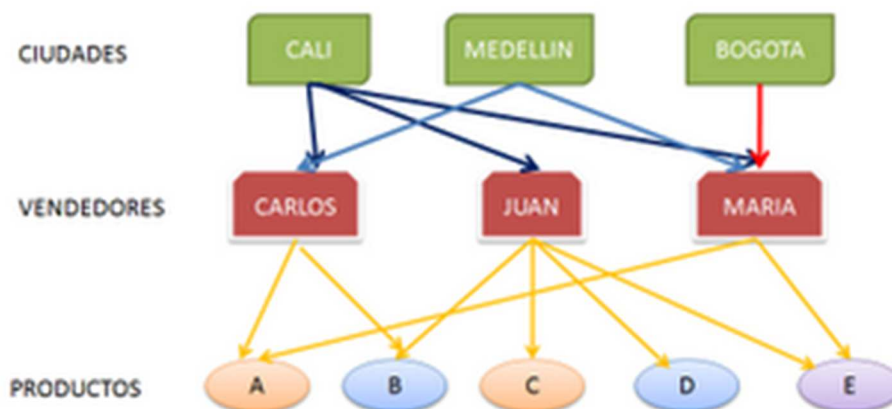


Fig. 4: El modelo de datos de red

El modelo más utilizado en la actualidad para modelar problemas reales y administrar datos dinámicamente es el **Modelo Relacional**. Tras ser postulados sus fundamentos en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California) no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos. Su idea fundamental es el uso de "relaciones", que podrían considerarse en forma lógica como conjuntos de datos llamados "tuplas". Pese a que ésta es la teoría de las bases de datos relacionales creadas por Edgar Frank Codd, la mayoría de las veces se conceptualiza de una manera más fácil de imaginar. Esto es pensando en cada relación como si fuese una tabla que está compuesta por registros (las filas de una tabla), que representarían las tuplas, y campos (las columnas de una tabla).

En este modelo, el lugar y la forma en que se almacenen los datos no tienen relevancia (a diferencia de otros modelos como el jerárquico y el de red). Esto tiene la considerable ventaja de que es más fácil de entender y de utilizar para un usuario no experto. Los datos pueden ser recuperados o almacenados mediante "consultas" que ofrecen una amplia flexibilidad y poder para administrar los datos.

**En este módulo se abordará el Modelo Relacional, con una Base de Datos Relacional, y su Administrador de Base de Datos y el lenguaje de consulta SQL Structured Query Language.**

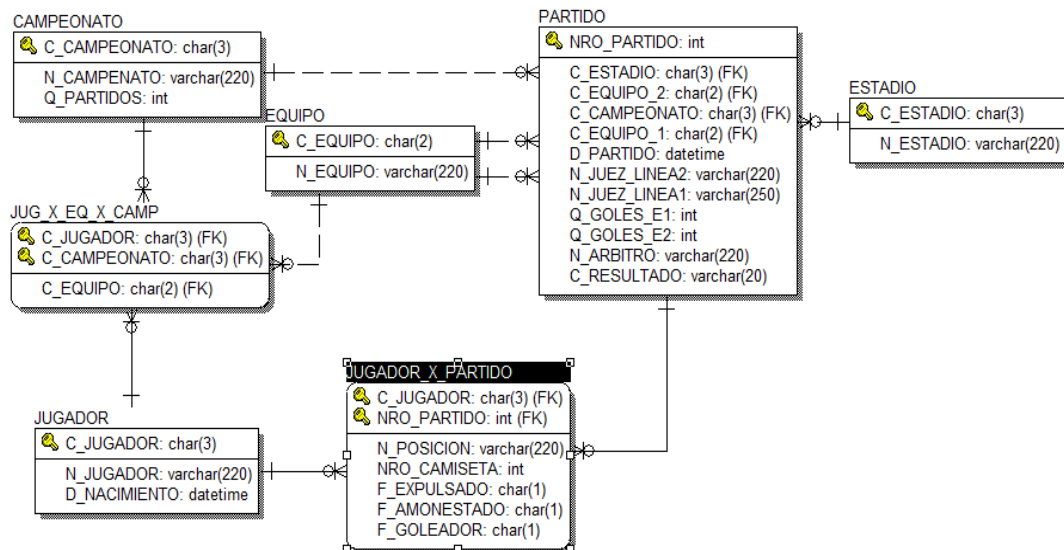


Fig. 5: El modelo de datos relacional

El primer obstáculo al que se enfrentan los programadores que usan el modelo relacional de datos es el limitado sistema de tipos soportado por el modelo relacional. Los dominios de aplicación complejos, como el diseño asistido por computadora y los sistemas de información geográfica, necesitan tipos de datos del mismo nivel de complejidad, como las estructuras de registros anidados, los atributos multi-valorados y la herencia, que los lenguajes de programación soportan. El **modelo de datos orientado a objetos** extiende el modelo de datos relacional ofreciendo un sistema de tipos más rico que incluye tipos de datos complejos y soporta el paradigma de orientación a objetos.

Una base de datos orientada a objetos incorpora todos los conceptos importantes del paradigma de objetos:

- **Encapsulamiento:** propiedad que permite ocultar los datos al resto de los objetos, impidiendo así accesos incorrectos o conflictos
- **Herencia:** propiedad a través de la cual los objetos heredan comportamiento dentro de una jerarquía de clases
- **Polimorfismo:** propiedad de una operación mediante la cual puede ser aplicada a distintos tipos de objetos

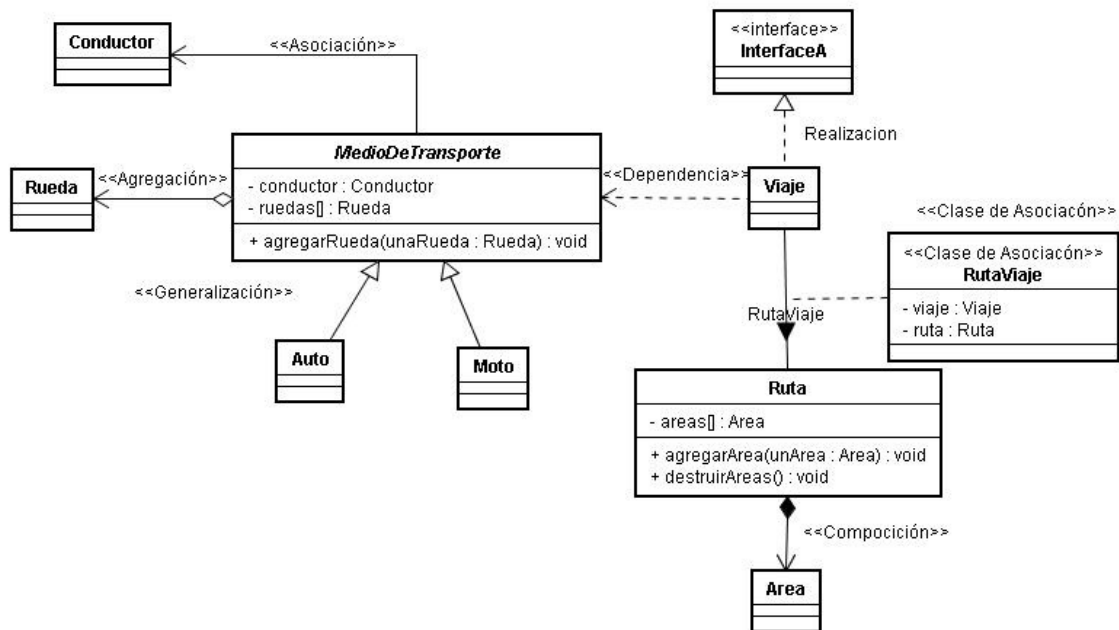


Fig. 6: El modelo de datos orientado a objetos

Los **modelos de datos orientado a documentos** permiten la *indexación a texto completo*, es decir que posibilita crear estructuras de datos similares a los árboles de búsqueda para, en líneas generales, realizar búsquedas más potentes.

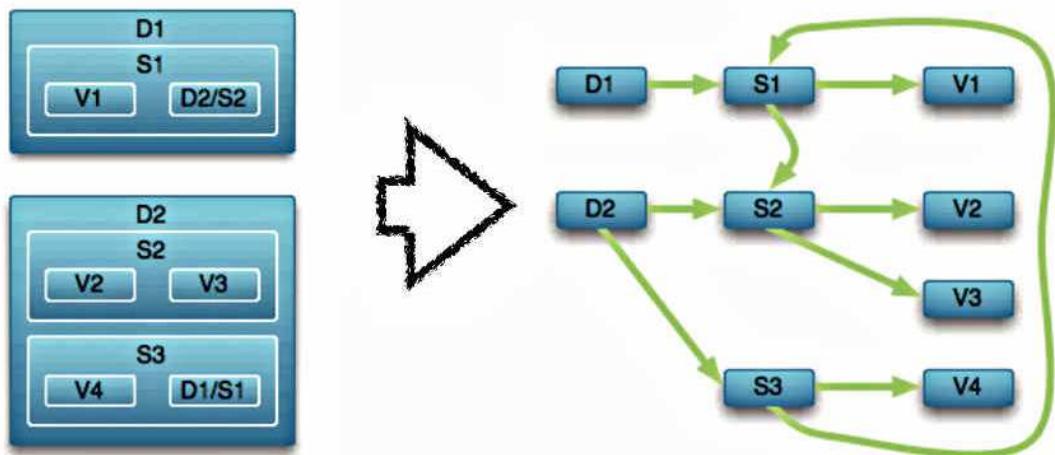


Fig. 7: El modelo de datos orientado a documentos

Los **modelos de datos multidimensionales** son ideados para desarrollar aplicaciones muy concretas, como creación de Cubos OLAP (On-Line Analytical Processing, Procesamiento Analítico en Línea). Básicamente no se diferencian demasiado de las bases de datos relacionales (una tabla en una base de datos relacional podría serlo también en una base de datos multidimensional), la diferencia se encuentra a nivel conceptual; en las bases de datos multidimensionales los campos o atributos de una tabla pueden ser de dos tipos, o bien representan dimensiones de la tabla, o bien representan métricas que se desea estudiar.

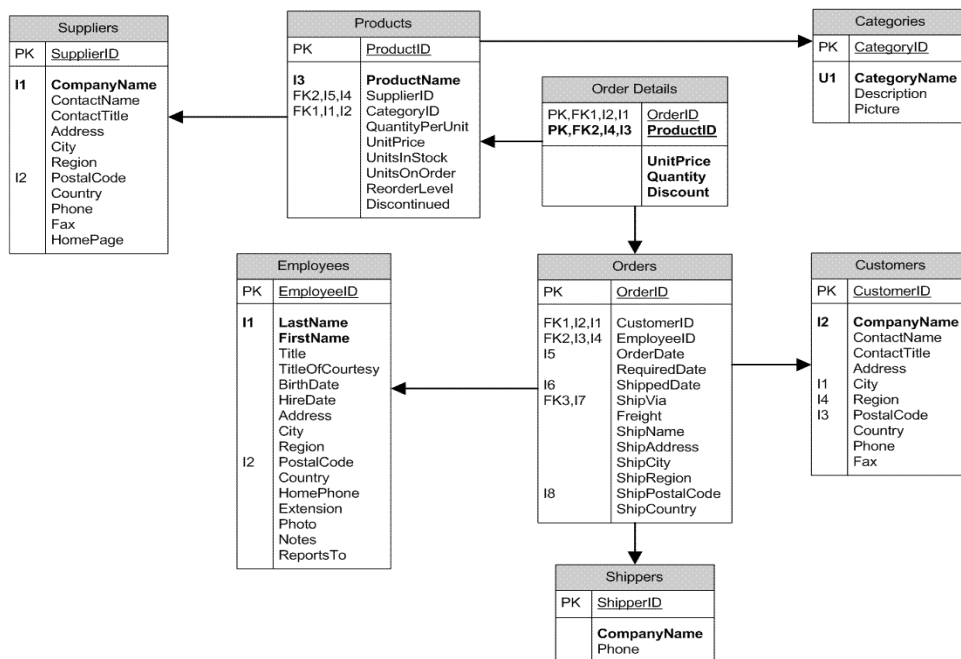


Fig. 8: El modelo de datos multidimensional

Las bases de datos multidimensionales suelen representarse gráficamente como cubos, según se muestra en la siguiente figura.

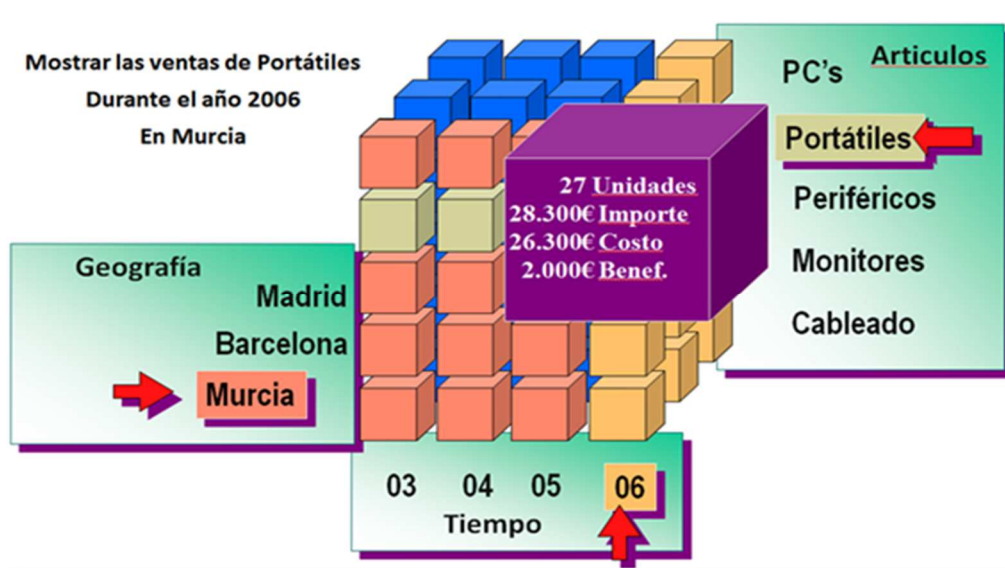


Fig. 9: El modelo de datos multidimensional como un cubo

## Bases de datos NoSQL

Conforme evoluciona la tecnología, se ha logrado aprender técnicas bastante comunes para normalizar las bases de datos relacionales en la medida de lo posible, escalarlas según crece la demanda, y utilizarlas como sistema de persistencia para almacenar información desde lenguajes de programación estructurados u orientados a objeto. La cuota de uso de software como SQLite, MySQL, PostgreSQL u Oracle, por poner cuatro ejemplos conocidos, es muy alta, encontrándose en la mayor parte de los desarrollos modernos.

Con la llegada de Internet y el software como servicio o SaaS<sup>3</sup>, los servicios en la nube y las startups<sup>4</sup> de éxito con millones de usuarios; también llegaron los problemas de alta escalabilidad: si bien los modelos relacionales se pueden adaptar para hacerlos escalar incluso en los entornos más difíciles, a menudo se hacen cada vez menos intuitivos a medida que aumenta la complejidad.

Los sistemas NO SQL intentan atacar este problema proponiendo una estructura de almacenamiento más versátil a costa de perder ciertas funcionalidades como las transacciones con operaciones en más de una colección de datos o la incapacidad de ejecutar el producto cartesiano de dos tablas teniendo que recurrir a la denormalización<sup>5</sup> de datos.

NO SQL es un término que describe las bases de datos no relacionales de alto desempeño. Las bases de datos NO SQL utilizan varios modelos de datos, incluidos los de documentos, gráficos, claves-valores y columnas. Las bases de datos NO SQL son famosas por la facilidad de desarrollo, el desempeño escalable, la alta disponibilidad y la resiliencia<sup>6</sup>. A continuación, se presenta una tabla comparativa entre las bases relacionales y las NO SQL.

	Base de datos relacional	Base de datos NO SQL
Modelo de datos	El modelo relacional normaliza los datos en estructuras tabulares conocidas como tablas, que están formadas por filas y columnas. Contienen un esquema que define estrictamente las tablas, columnas, índices, relaciones entre las tablas y otros	Las bases de datos no relacionales (NO SQL) no suelen contener un esquema. Se suele utilizar una clave de partición para recuperar valores, conjuntos de columnas o documentos JSON <sup>7</sup> o XML <sup>8</sup> semiestructurados, así

<sup>3</sup> **SaaS:** Software as a Service

<sup>4</sup> **Startup:** Una startup podría definirse como una empresa de nueva creación que presenta unas grandes posibilidades de crecimiento y, en ocasiones, un modelo de negocio escalable. Aunque el término de startup puede referirse a compañías de cualquier ámbito, normalmente suele utilizarse para aquellas que tienen un fuerte componente tecnológico y que están relacionadas con el mundo de Internet y las TICs (Tecnologías de Información y Comunicación). Debido a estas características las startups tecnológicas suelen diferenciarse de otras empresas jóvenes en sus posibilidades de crecimiento y necesidades de capital.

<sup>5</sup> **Denormalización:** La denormalización es el proceso de procurar optimizar el desempeño de una base de datos por medio de agregar datos redundantes.

<sup>6</sup> **Resiliencia:** Capacidad de adaptación frente a un agente perturbador o un estado o situación adversos.

<sup>7</sup> **JSON:** acrónimo de JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript, aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente.

<sup>8</sup> **XML:** acrónimo de eXtensible Markup Language, traducido como "Lenguaje de Marcado Extensible" o "Lenguaje de Marcas Extensible", utilizado para almacenar datos en forma legible. XML no ha nacido únicamente para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada

	elementos de las bases de datos. La normalización es el proceso que elimina redundancia en las bases de datos. Cada dato es ubicado en una celda, y el valor de ese dato debe ser atómico.	como otros documentos que contengan atributos de elementos relacionados.
Propiedades ACID	<p>Los sistemas de administración de bases de datos relacionales (RDBMS) tradicionales admiten un conjunto de propiedades definidas por el acrónimo ACID (por sus siglas en inglés): atomicidad, consistencia, aislamiento y durabilidad.</p> <ul style="list-style-type: none"> <li>• Atomicidad significa “todo o nada” – una transacción se ejecuta completamente o no se ejecuta en absoluto.</li> <li>• Consistencia quiere decir que una vez se ha ejecutado una transacción, los datos deben acoplarse al esquema de la base de datos.</li> <li>• Aislamiento requiere que las transacciones simultáneas se ejecuten por separado.</li> <li>• Durabilidad es la capacidad de recuperarse de un error inesperado del sistema o de un corte de energía y volver al último estado conocido.</li> </ul>	Habitualmente, las bases de datos No SQL intercambian algunas de las propiedades ACID de los sistemas de administración de bases de datos relacionales (RDBMS) tradicionales por un modelo de datos más flexible que se escala de forma horizontal. Estas características convierten las bases de datos No SQL en una elección excelente en las situaciones en las que los RDBMS detectan desafíos en la arquitectura a la hora de superar una combinación de cuellos de botella de desempeño, escalabilidad, complejidad operativa y el aumento de los costos de administración y soporte.
Desempeño	Normalmente, el desempeño depende del subsistema de disco. Es necesaria la optimización de consultas, índices y estructura de tabla para lograr el máximo desempeño.	El desempeño, por lo general, depende del tamaño del clúster de hardware <sup>9</sup> subyacente, la latencia de red <sup>10</sup> y la aplicación que efectúa la llamada.
Escalado	Lo más sencillo es ampliar la escala con un hardware más rápido. Se requieren inversiones adicionales para que las tablas relacionales abarquen un sistema distribuido.	Está diseñada para reducir la escala utilizando clústeres distribuidos de hardware de bajo costo para aumentar el desempeño sin que aumente la latencia.

Aunque conocer el tipo de modelo de datos es importante, también hay que tener en cuenta otros factores. Las bases de datos No SQL están pensadas para ser escalables y distribuidas. Y por ser distribuidas es importante tener en cuenta el teorema CAP, también llamado Teorema

---

entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

<sup>9</sup> **Cluster de Hardware:** El término clúster (del inglés cluster, "grupo" o "racimo") se aplica a los conjuntos o conglomerados de computadoras unidos entre sí normalmente por una red de alta velocidad y que se comportan como si fuesen una única computadora. La tecnología de clústeres ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y software para aplicaciones críticas, servidores web y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

<sup>10</sup> **Latencia de red:** En redes informáticas de datos se denomina latencia a la suma de retardos temporales dentro de una red. Un retardo es producido por la demora en la propagación y transmisión de paquetes dentro de la red.



de Brewer (ver representación gráfica al final de la página). Este teorema enuncia que es imposible para un sistema distribuido garantizar simultáneamente:

- **Consistencia:** al realizar una consulta o inserción siempre se tiene que recibir la misma información, con independencia del nodo o servidor que procese la petición.
- **Disponibilidad:** que todos los clientes puedan leer y escribir, aunque se haya caído uno de los nodos.
- **Tolerancia a particiones:** los sistemas distribuidos pueden estar divididos en particiones (generalmente de forma geográfica). Esta condición implica, que el sistema tiene que seguir funcionando, aunque existan fallos o caídas parciales que dividan el sistema.

Por tanto, además de pensar en el tipo de base de datos No SQL que mejor se adapta a nuestro modelo de datos, también tendremos que pensar en su funcionamiento. Así podremos conseguir que nuestra aplicación funcione de la mejor manera posible.

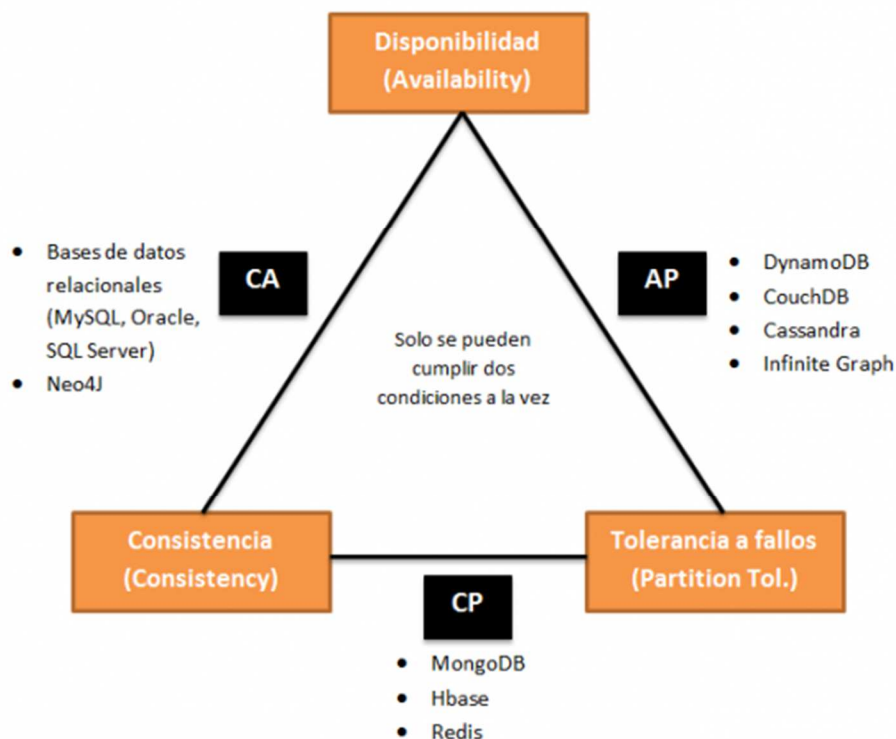


Fig. 10: El Teorema de CAP

## Bases de datos para soporte a la toma de decisiones

A medida que las empresas crecen lo mismo sucede con sus bases de datos y las fuentes de información. Surgen nuevos sistemas que brindan soporte a las distintas áreas de la organización que deben compartir datos para el desarrollo de sus funciones.

Un **data warehouse** es una tecnología de base de datos de sólo lectura que se caracteriza por integrar y depurar información de una o más fuentes distintas, para luego procesarla permitiendo su análisis desde infinidad de perspectivas y con grandes velocidades de respuesta. La ventaja principal de este tipo de bases de datos radica en la estructura de almacenamiento

de la información (modelos de tablas en estrella, en copo de nieve, cubos relacionales... etc). Este tipo de persistencia de la información es homogénea y fiable, y permite la consulta y el tratamiento jerarquizado de la misma (siempre en un entorno diferente a los sistemas operacionales).

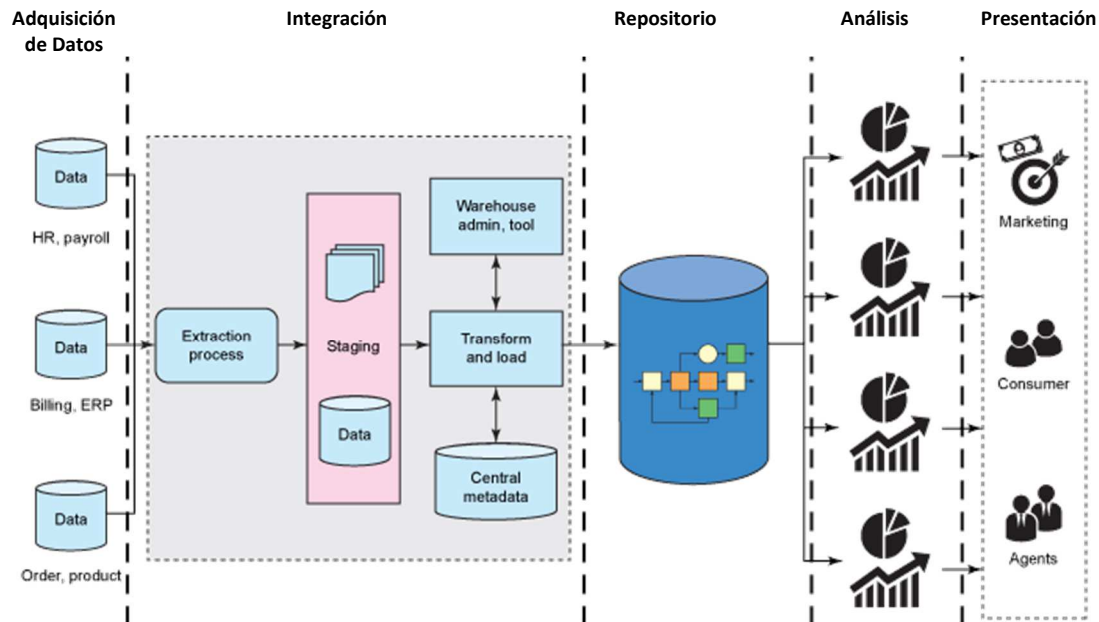


Fig. 11: Estructura de un data warehouse

Algunas características de los Data Warehouse's son:

- Proporciona información clave para la toma de decisiones en la empresa
- Mejora la calidad de las decisiones tomadas.
- Los datos son útiles a mediano y largo plazo.
- Si las fuentes de los datos y los objetivos están claros, los sistemas son sencillos de instalar.
- Útil para el almacenamiento de análisis y consultas históricas.
- Permite mayor flexibilidad y rapidez en el acceso a la información.
- Brinda una comunicación fiable entre todos los departamentos de la empresa.

Para que el Data Warehouse sea exitoso se necesita una limpieza continua, transformación e integración de los datos. Además, requiere sistemas, aplicaciones y almacenamiento específico. Es un trabajo constante que garantiza el éxito de los datos en el diagnóstico y las soluciones de inteligencia de negocios que serán implementadas en la compañía.

Finalmente, se puede decir que el principal beneficio del Data Warehouse para las empresas es que tiene la propiedad de eliminar los datos que interfieren con el análisis de la información y su entrega; sea cual sea el formato requerido, el Data Warehouse facilita el proceso de gestión.



## Modelo Entidad Relación

---

### Caso de Estudio

Antes de comenzar a desarrollar el tema de modelo de Entidad Relación, vamos a presentar nuevamente el caso de estudio utilizado en el módulo de Programación Orientada a Objetos: **“Sistema de Gestión de Ventas y Reservas de Entradas para un Complejo de Cines”**, así recordamos de que trata y damos continuidad al desarrollo del software que iniciamos antes.

- Un complejo de cines está integrado por varios **cines** ubicados principalmente en los centros comerciales de la ciudad.
- Cada cine cuenta con una cantidad de **salas**, que son las que exhiben las películas en las distintas **funciones** cinematográficas.
- La **programación** de las salas se renueva en forma semanal, existiendo la posibilidad de que algunas salas queden sin uso. Cabe mencionar que no todas las salas tienen la misma capacidad (cantidad de butacas).
- La **programación** es la que determina qué **películas** van a proyectarse y los **horarios** para cada función de cada una de las salas, para todos los cines. Esta programación se realiza en forma centralizada, desde la administración del Complejo, tomándose como base la información de las películas próximas a estrenar, que envía el INCAA (Instituto Nacional de Cines y Artes Audiovisuales). La programación implica el diseño de las funciones y sus horarios en forma anticipada, debiendo el responsable de la misma, habilitar cada función en el momento que desee permitir la reserva y/o venta de entradas para la misma.
- La que se le entrega al cliente representa el comprobante de venta y como tal debe cumplir con lo reglamentado en la Ley de Facturación vigente, debiendo contener como datos: nro. de venta, fecha de venta, número de función, sala en la que se proyecta la película, el nombre de la película, fecha y hora de la función, el precio, el tipo de entrada (si es mayor, menor, jubilado) y la calificación de la película, que según especificaciones de la Ley de Cine Nro. 17.741, debe ser informada tanto en la entrada como al inicio de la película. Es importante destacar que la entrada es válida únicamente para la fecha, hora y función indicadas en la misma.
- Los **tipos de entradas y los días y horarios** de proyección son los que determinan el **precio** de la entrada, que también pueden variar en cada cine del complejo. Las funciones admiten ciertos tipos de entradas y otros no, dependiendo de factores como: horarios, calificación de las películas, etc. Por ejemplo: si una película está calificada como para mayores de 16 años, para esa función no se pueden vender entradas de TIPO = MENOR. Cada función tiene asociado un tipo de función, que determina si la función es un pre-estreno, un estreno, una función normal.

## Conceptos básicos del Modelo de Entidad Relación

### Entidades

El elemento básico representado por el modelo entidad relación es una **entidad**, que es una cosa del mundo real con una existencia independiente.

- Una entidad puede ser un elemento con una existencia física (por ejemplo, una persona en particular, un coche, una casa o un empleado) o puede ser un elemento con una existencia conceptual (por ejemplo, una venta, un trabajo o un curso universitario).
- Cada entidad tiene atributos (propiedades particulares que la describen).

Una **entidad es una abstracción** de un conjunto de cosas del mundo real tal que:

- Las cosas de ese conjunto tienen las mismas características o comportamiento.
- Las cosas de ese conjunto están sujetas y conformes a las mismas reglas.

Las entidades que se tendrán en cuenta al modelar un sistema son aquellas que representan "cosas" de las que el sistema necesita almacenar ciertos datos.

Ejemplo:

El conjunto: {ruleman 8705, tornillo 5 mm, filtro de aire, etc.}  
forma la entidad => REPUESTO

Como vemos, el concepto de **entidad** se relaciona muy estrechamente con el concepto de **clases**, que fue introducido en el apunte de programación orientada a objetos. Esto se debe a que las bases de datos sirven para almacenar datos, que, en el caso de la programación orientada a objetos, los datos a almacenar serán los relacionados con los objetos generados en tiempo de ejecución en un producto de software. Por lo tanto, la *mayoría* de las entidades tendrán una correspondencia uno a uno con las clases identificadas en la etapa de requerimientos y análisis del proceso de construcción de software.

### Identificación de Entidades

Para identificar entidades vemos cuál es el sistema que se está analizando (identificando sus límites) y nos preguntamos:

**¿Cuáles son las cosas de este sistema de las que nos interesa tener sus datos?**

Para facilitar la identificación de las entidades, debemos tener en cuenta que existen cuatro categorías, de acuerdo a lo que ellas representen:

- ✓ **Cosas tangibles:** Artículo, Repuesto, Rodado.
- ✓ **Roles desempeñados por personas u organizaciones:** Cliente, Proveedor, Personal.
- ✓ **Incidentes:** Usado para representar la ocurrencia de un hecho (en un sistema de una compañía de seguros: Sinistros; en una empresa de transporte: Viajes).
- ✓ **Interacciones:** Representan alguna transacción (Compra, Pedido, Venta, Pago).

Es importante una buena elección del nombre dado a una entidad para la legibilidad y el entendimiento del modelo de datos. Luego de identificar las entidades, se analizan sus atributos.

## Atributos

Un atributo es una abstracción que identifica características, propiedades que posee una entidad. Los atributos de una entidad deben ser:

- **Completo:** capturar toda la información que interesa del objeto, desde el punto de vista del sistema.
- **Plenamente elaborados:** cada atributo captura un aspecto separado de la entidad.
- **Mutualmente independientes:** cada atributo debe tomar un valor independientemente de los valores asumidos por otros atributos.

Con respecto a nuestro caso de estudio, podemos identificar la entidad principal PELICULA y sus atributos, de la siguiente manera:

- Entidad: PELICULA
- Atributos: nombre, título original, año de estreno, disponible, duración, fecha de ingreso.

Podemos ver esta entidad representada en un diagrama de entidad-relación de la siguiente manera:

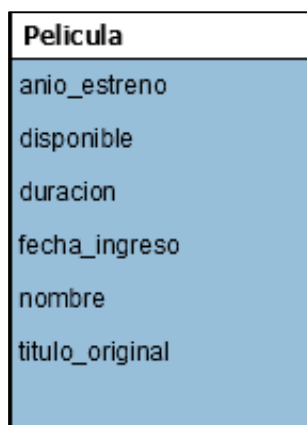


Fig. 12: Representación de una Entidad

Así podemos tener como ejemplo de entidades PELICULA, a dos instancias de película, con los siguientes atributos:

**Película 1:** {2013, true, 143, 11/07/2013, "El Gran Gatsby", "The Great Gatsby"}

**Película 2:** {2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes"}

A cada película de la entidad PELICULA, se las denomina genéricamente **instancias** de dicha entidad.

## Clasificación de Atributos

- **Atributos identificadores:** el o los atributos que permiten identificar unívocamente a una instancia de una entidad. Constituyen la "clave primaria".
- **Atributos descriptivos:** son las características intrínsecas de cada instancia de la entidad; como lo dice su nombre, describen a la entidad, representan sus propiedades.

- **Atributos referenciales:** son atributos que sirven para relacionar entidades entre sí. Se denominan REFERENCIALES ya que hacen referencia al ATRIBUTO IDENTIFICADOR de la entidad con que se relacionan. Estos conceptos serán profundizados a continuación:

### Atributo Identificador Único

Se denomina identificador a **uno o más atributos** que identifican unívocamente cada instancia de una entidad; es conocido también como "clave candidata". ***Es decir, nunca puede existir dos instancias de una entidad con el mismo valor de su atributo identificador.*** En una entidad puede haber varios atributos posibles para ser elegidos como "identificador".

Por ejemplo, en el caso de una entidad EMPLEADO podríamos tener dos atributos que cumplen con la condición de identificar unívocamente a cada instancia de la entidad: **documento y CUIT.**

En estos casos, para elegir el atributo identificador debemos tener en cuenta dos reglas:

- Que la clave sea **mínima**: Es decir elegir la alternativa en la que se necesiten menos atributos para conformar la clave.
- Elegir el atributo **más significativo** dentro del dominio del problema que se está modelando.

En el caso de ejemplo mencionado anteriormente, en la entidad Empleado se debe elegir un atributo para que sea identificador, en este caso conviene elegir el CUIT ya que ocasionalmente pueden existir dos personas con el mismo DNI, pero nunca con el mismo CUIT.

Una definición más formal podría ser la siguiente:

"El atributo A, o el conjunto de atributos, de una entidad, es un posible atributo identificador si y solo si satisface dos propiedades:

- **Unicidad:** en cualquier momento dado no existen 2 instancias con el mismo valor de A.
- **Minimidad:** Si A es compuesto (es decir el atributo identificador está formado por más de un atributo) no será posible eliminar ningún componente de A sin destruir la propiedad de unicidad.

Toda entidad tiene por lo menos un atributo como posible atributo identificador. El o los atributos identificadores se señalan con el símbolo "@"(arroba), o de lo contrario con la sigla PK (clave primaria).

Para mejorar el desempeño de la base de datos se recomienda utilizar identificadores numéricos; por lo tanto, si una entidad no posee un atributo identificador numérico, se debería agregar un atributo, comúnmente llamado id (abreviación de identificador) seguido por el nombre de la entidad, como se muestra en la entidad Película, donde podemos observar que si bien el nombre de la película no debe repetirse en el negocio, no es un atributo numérico; como consecuencia es más conveniente agregar uno, de la siguiente manera:

Película
@id_película
año_estreno
disponible
duracion
fecha_ingreso
nombre
titulo_original

Fig. 13: Representación de una Entidad con su atributo identificador

**Película 1:** {1, 2013, true, 143, 11/07/2013, "El Gran Gatsby", "The Great Gatsby"}

**Película 2:** {2, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes"}

Este atributo agregado id\_película, no es más que un número identificador que crece secuencialmente a medida que se agregan nuevas películas: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...n.

#### Atributo Referencial

Un atributo referencial se utiliza para poder establecer relaciones entre diferentes entidades de un modelo entidad- relación. Se dice que un atributo *j*, ó un conjunto de atributos, de una entidad B es un atributo referencial si y solo si satisface dos propiedades:

- Cada valor *j* es **nulo del todo o no nulo del todo**. – En caso de ser un atributo compuesto, formado por más de un atributo.
- Existe una entidad A con atributo identificador *j* tal que:
  - Cada valor no nulo de *j* es en la entidad B idéntico al valor *j* en alguna instancia de la entidad A. Es decir que si en B el atributo *j* tiene valor es porque existe ese mismo valor de *j* en la entidad A.

Podemos ver un ejemplo relacionado con el caso de estudio del Complejo de Cines, con las entidades: PELICULA y PAIS DE ORIGEN. Se utiliza el símbolo “#” (numeral) para señalar que un atributo es referencial, o de lo contrario con la sigla FK (clave foránea).

Esto se logra de la siguiente manera:

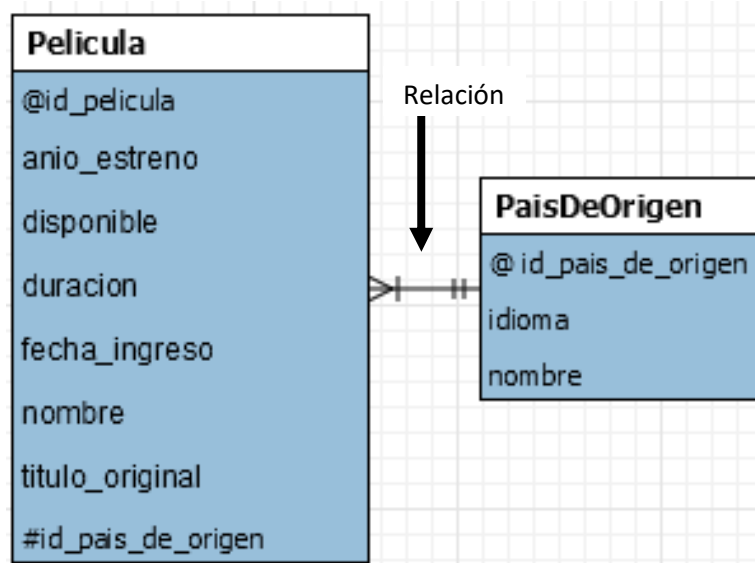


Fig. 14: Representación de Entidades y su forma de relacionarse, con su atributo referencial

Como podemos ver, la entidad Película se relaciona con la entidad PaisDeOrigen, donde país de origen tiene como atributo identificador, señalado con @, que es: id\_pais\_de\_origen.

La entidad Película tiene un atributo referencial, señalado con #, denominado: id\_pais\_de\_origen, que para cada instancia de película referenciará a un único país de origen. Por ejemplo:

Si tuviéramos las siguientes entidades en PaísDeOrigen:

País de Origen 1: {1, "Español", Argentina}

País de Origen 1: {2, "Inglés", Estados Unidos}

País de Origen 1: {3, "Francés", Francia}

Entonces la entidad película, tendría en el atributo referencial id\_pais\_de\_origen, el valor 1, que referencia a la instancia Argentina de la entidad PaisDeOrigen, como se ve a continuación:

**Película 2: { 1, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes", 1}**

### Reglas de Integridad

Existen dos reglas denominadas "Reglas de Integridad" que son generales y se aplican a todo el modelo de datos relacional y que son:

1. **Integridad de Entidades:** Ningún componente del atributo identificador en una entidad aceptará NULOS (nulo se considera que es inexistente, es decir, ausencia de valor).

#### Ejemplo:

**X Película: {NULL, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes", 1}**

**✓ Película: { 2, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes", 1}**

2. **Integridad Referencial:** Un modelo de datos no debe contener valores en sus atributos referenciales para los cuales no exista un valor concordante en el (ó los) atributos identificadores en la entidad objetivo pertinente.

**Ejemplo:**

✗ Película: { 2, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes", 4 }

No existe en la tabla PaisDeOrigen una instancia con id\_pais\_de\_origen = 4.

✓ Película: { 2, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes", 1 }

Es decir que si existe un valor para el (ó los) atributo referencial si o si debe existir su concordante de atributo identificador en la instancia con la cual se relaciona.

## Relaciones

- Una relación es la abstracción de un conjunto de asociaciones que existen entre las instancias de dos entidades, por ejemplo, existe una relación entre Película y PaisDeOrigen (Ver figura 14).
- Las relaciones tienen sentido bidireccional.
- Las relaciones existen ya que las entidades representan aspectos del mundo real y en este mundo los componentes no están aislados, sino que se relacionan entre sí; es por esto que es necesario que existan las relaciones entre las entidades.

## Cardinalidad y Opcionalidad de las relaciones

**Cardinalidad:** Indica para una instancia de una entidad A con cuántas instancias de la entidad B, se relaciona. Las posibilidades son: 0, 1 o muchos, que se representan con  $\bigcirc$  ; | o ; respectivamente

**Opcionalidad:** Indica para una instancia de una entidad A, si la relación con instancias de la entidad B, es opcional u obligatoria. Las posibilidades son: 0 o 1 que se representan con  $\bigcirc$  o |,  $\Leftarrow$  ; respectivamente. Veamos ejemplos:

- ✓ **Uno a uno:** por ejemplo, en el caso de estudio añadiremos las clases que se muestran a continuación para explicar esta relación:

Entonces, si llevamos estas clases a entidades, vemos que: un empleado puede tener o no un usuario y si ese usuario existe, es para un único empleado.

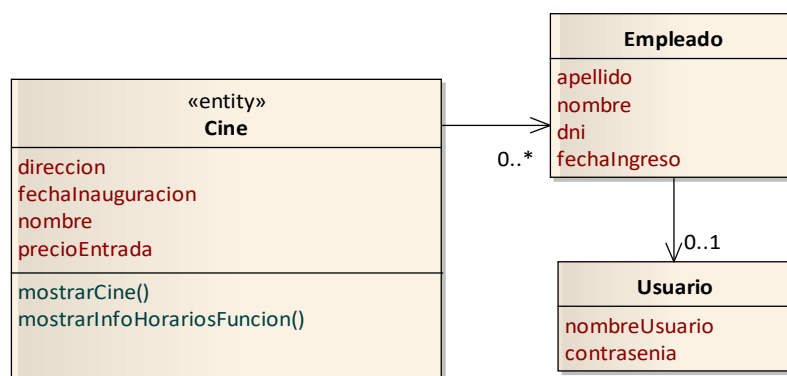


Fig. 15: Vista de Clases del Modelo de Dominio del Complejo de Cines

Como se ve en el diagrama de entidad relación siguiente, se utiliza para representar esta relación dos || en cada extremo de la relación para representar la cardinalidad; y un | y un ○, para representar la opcionalidad.

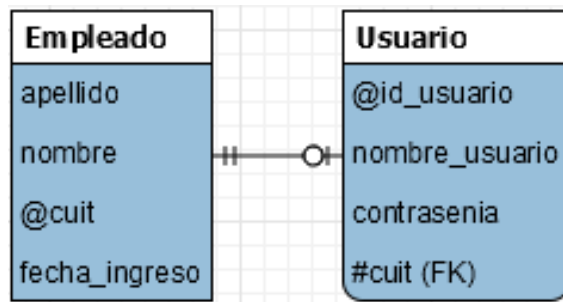


Fig. 16: Vista parcial del Modelo de Entidad Relación para representar la relación entre Empleado y Usuario

Esto se lee, de la siguiente manera:

**“Un empleado puede tener opcionalmente un único usuario o no tener ningún usuario asociado, un usuario está asociado de manera obligatoria a un único empleado”**

- ✓ **Uno a muchos**, por ejemplo, una película tiene un único género, pero un género como “Drama” puede estar asignado a muchas películas:

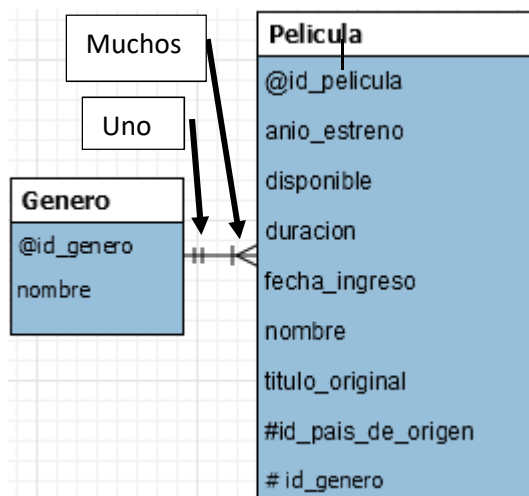


Fig. 17: Vista parcial del Modelo de Entidad Relación para representar la relación entre la entidad Película y la entidad Género

Esto se lee, de la siguiente manera:

**“Un género puede estar asignado a muchas películas, pero una película tiene un único género”**



- ✓ **Muchos a Uno**, por ejemplo, una película tiene muchas funciones asignadas, pero una función es para una única película.

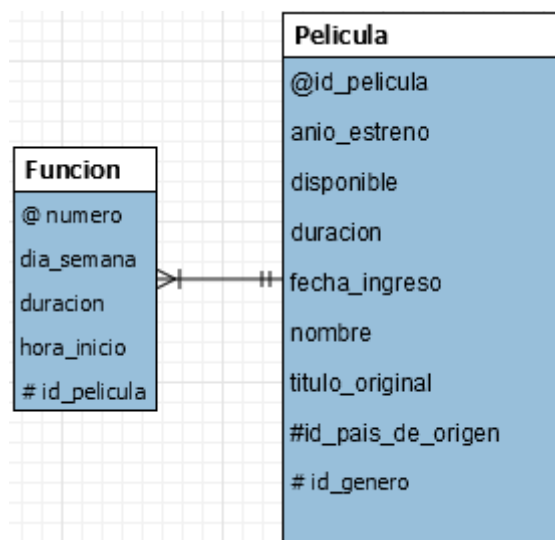


Fig. 18: Vista parcial del Modelo de Entidad Relación para representar la relación entre la entidad Película y la entidad Género

**ACLARACIÓN:** Para los casos anteriores (uno a muchos o muchos a uno) siempre el atributo referencial se debe colocar en la entidad donde está la cardinalidad MUCHOS. Esto se justifica debido a que los atributos deben tener valores atómicos en cada celda, es decir un único valor.

A continuación, se muestran 2 ejemplos para explicar lo mencionado anteriormente:

**Caso 1: Relación entre Función y Película, con Función referenciando a Película:** en este caso la función tiene una única película asociada porque se proyecta una película por función:

ENTIDAD FUNCIÓN				
@numero	dia_semana	duración	Hora_inicio	#id_pelicula
1	3	150	22:30	2
2	4	150	18:15	2

ENTIDAD PELICULA				
@id_pelicula	Nombre	duración	#id_pais_de_origen	#id_genero
2	Relatos Salvajes	122	1	1

\*Para simplificar, se muestra un conjunto reducido de atributos de película.

**Caso 2: Relación entre Función y Película, con Película referenciando a Función: en este caso como la película puede proyectarse en muchas funciones, deberíamos poner en la columna identificada como #id-funcion, más de un atributo referencias, rompiendo la propiedad de atomicidad de los atributos:**

ENTIDAD FUNCIÓN			
@numero	día_semana	duración	Hora_inico
1	3	150	22:30
2	4	150	18:15

ENTIDAD PELICULA					X
@id_pelicula	Nombre	duración	#id_pais_de_origen	#id_genero	#id_funcion
2	Relatos Salvajes	122	1	1	1,2

\*Para simplificar, se muestra un conjunto reducido de atributos de película.

- ✓ **Muchos a muchos**, por ejemplo, un personaje puede pertenecer a varias películas, por ejemplo: Harry Potter y la piedra filosofal, Harry Potter y la cámara secreta, Harry Potter y el prisionero de Azkaban... y una película puede tener varios personajes.

**En este caso, en donde ambas entidades tienen la cardinalidad muchos, nos encontramos con la limitación de no poder dónde colocar el atributo referencial en ninguna entidad, sin romper la unicidad.**

Cuando la relación es de muchos a muchos, la forma de solucionarlo y respetar con la propiedad de unicidad es por medio de la creación de una entidad intermedia, denominada *Entidad Asociativa*, que relacione un personaje con una película de la siguiente manera:

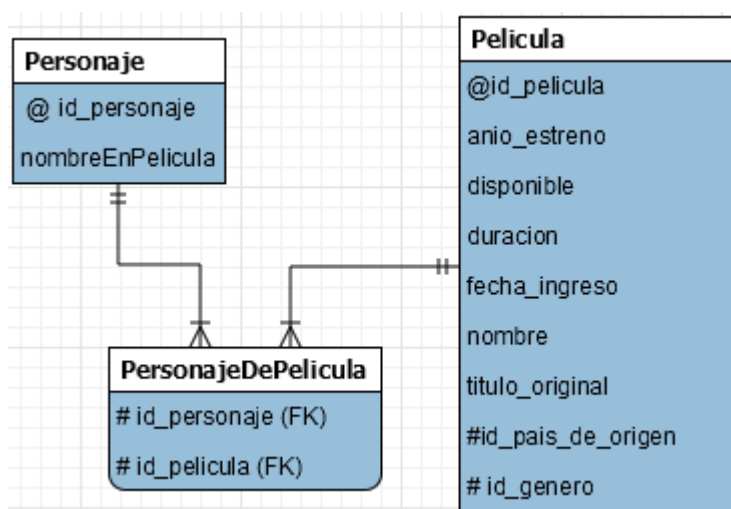


Fig. 19: Vista parcial del Modelo de Entidad Relación para representar la relación entre las entidades Película y Personaje, con su entidad asociativa

Como se puede ver en la figura 19, la tabla intermedia creada, *PersonajeDePelicula*, se encarga de establecer una relación uno a uno, entre un personaje y una película; de esta manera siguiendo con el ejemplo de Harry Potter, tendríamos:

Entidades de Personajes:

{1, "Harry Potter"}  
 {2, "Ron Weasley"},  
 {3, "Hermione Granger"}

**Instancias de la Entidad Pelicula:**





{5, 2001, true, 152, 03/03/2001, "Harry Potter y la piedra filosofal", "Harry Potter and the Philosopher's Stone", 1, 3}

{6, 2002, true, 152, 03/06/2002, "Harry Potter y la cámara secreta", "Harry Potter and the Chamber of Secrets", 1, 3}

**Instancias de la Entidad PersonajeDePelicula:**

{1,5},{ 2,5},{3,5},{1,6},{ 2,6},{3,6}

La siguiente tabla presenta un resumen de la Simbología usualmente adoptada en la construcción de Diagramas de Entidad - Relación para representar relaciones entre entidades:

Tipo de Cardinalidad / Opcionalidad	Simbología	Descripción de Finalidad y Observaciones
0 ó 1	 (0,1)	Ninguna o una única ocurrencia servirá de asociación entre dos entidades cualquiera. Si la relación existe, será una única ocurrencia de la Entidad.
1 y 1	 (1,1)	Una y solo una ocurrencia estará asociando una Entidad A con una Entidad B.
0 a N	 (0, N)	Ninguna, una o varias ocurrencias asocian dos entidades del modelo de datos. Si existe podrá ser efectivizado por una o varias ocurrencias de la Entidad.
1 a N	 (1,N)	Una o varias ocurrencias servirán siempre de asociaciones entre dos entidades cualquiera del modelo de Datos.

## Modelo Lógico Relacional

El Modelo Lógico Relacional es un modelo de **datos conceptual de alto nivel**. Este modelo y sus variaciones se utilizan con frecuencia para el diseño conceptual de las aplicaciones de base de datos, y muchas herramientas de diseño emplean estos conceptos. Describimos los conceptos básicos de la estructura de datos y las restricciones del modelo, así como su uso en el diseño de esquemas conceptuales para las aplicaciones que utilizan base de datos.

Este modelo es una forma de representar los datos (mediante tablas), y la manera para manipular esa representación (utilizando operadores).

En términos más precisos, el Modelo Relacional se ocupa de tres aspectos de los datos: su estructura, su integridad y su manipulación, las que se detallan a continuación.

### Estructura de Datos Relacional

El modelo relacional representa la base de datos como una colección de relaciones. Informalmente, cada una de estas relaciones se parece a una tabla de valores. Cuando una relación está pensada como una tabla de valores, cada fila representa una colección de valores relacionados. En la sección anterior se presentaron los conceptos de entidad y de relación como conceptos para el modelado de datos reales. En el modelo relacional, cada fila de la tabla representa un hecho que, por lo general, se corresponde con una instancia de la entidad. El nombre de la tabla y de las columnas se utiliza para ayudar a interpretar el significado de cada uno de los valores de las filas. Por ejemplo, la tabla de la figura 19 se llama ESTUDIANTE porque cada fila representa la información de un estudiante particular. Los nombres de columna (Nombre, Dni, Tel. Particular, Dirección,..., etc.) especifican el modo de interpretar los valores de cada fila en función de la columna en la que se encuentren. Todos los datos de una columna son del mismo tipo de dato.

Nombre	Dni	TlfParticular	Dirección	TlfTrabajo	Edad	Mnc
Benjamín Bayer	305-61-2435	921234567	Cercado, 3	NULL	19	3,21
Ana Ortiz	381-62-1245	923456987	El Prado, 29	NULL	18	2,89
Belén Durán	422-11-2320	NULL	La Suerte, 6	915698743	25	3,53
Marcelo Gómez	489-22-1100	922789632	Pase del río, 98	914563210	28	3,93
Bárbara Campos	533-69-1238	911000550	El Suspiro Verde, 56	NULL	19	3,25

Figura 20: Relación ESTUDIANTE

En la terminología formal del modelo relacional, una fila recibe el nombre de tupla, una cabecera de columna es un atributo y el nombre de la tabla es una relación. El tipo de dato que describe los valores que pueden aparecer en cada columna está representado por un dominio de posibles valores.

Ahora pasaremos a describir con más detalle todos estos términos:

- Una **relación** corresponde a lo que conocemos como **tabla**, que se utiliza para representar los datos que queremos almacenar en nuestra base de datos, por ejemplo, la tabla Estudiante, mencionada anteriormente, o la tabla Película. En el modelado de entidad relación es la Entidad.
- Un **atributo** corresponde a una **columna o campo**. El **número de atributos** se denomina **grado**.
- Una **tupla** corresponde a una **fila o registro de esa tabla**. En el modelado de entidad relación es lo que denominamos instancia de la Entidad. El **número de tuplas** de una tabla se denomina **cardinalidad**.
- La **clave primaria** es un **identificador único para la tabla**, es decir una columna o combinación de columnas con la siguiente propiedad:
  - *Nunca existen dos filas de la tabla con el mismo valor en esa columna o combinación de columnas.*
- Un **dominio** es una **colección de valores**, de los cuales uno o más atributos (columnas) obtienen sus valores reales. Conocido como fondo de valores legales.

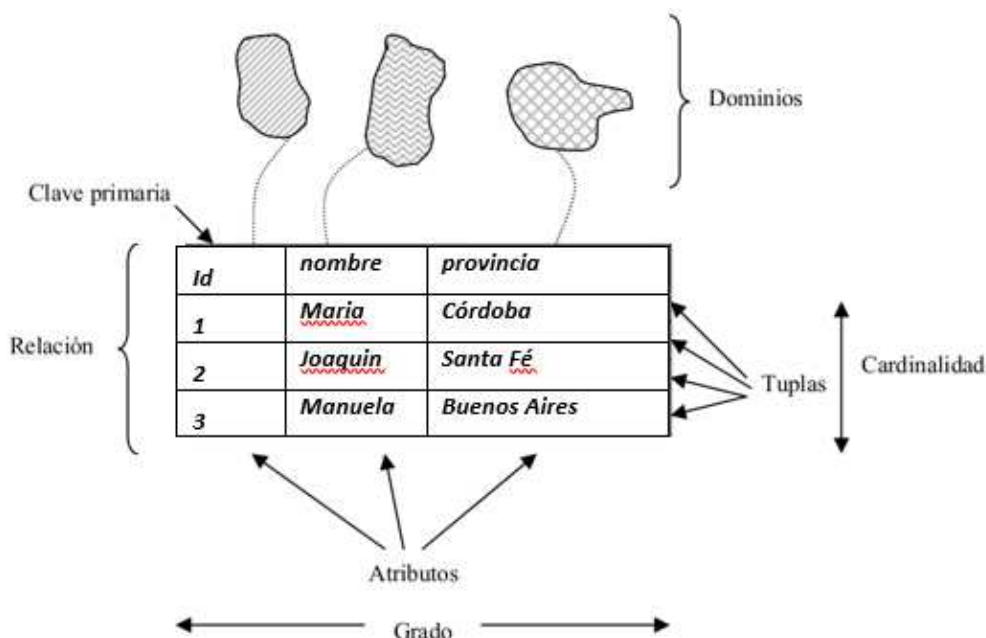


Figura 21: Conceptos vinculados a la Estructura Relacional

Para profundizar la comprensión de los conceptos anteriores, se plantea un ejemplo:

Una base de datos relacional consiste en un conjunto de tablas, a cada una de las cuales se le asigna un nombre exclusivo. Cada fila de la tabla representa una relación entre un conjunto de valores. Considérese la tabla de la siguiente figura:

número_cuenta	nombre_sucursal	saldo
C-101	Centro	500
C-102	Navacerrada	400
C-201	Galapagar	900
C-215	Becerril	700
C-217	Galapagar	750
C-222	Moralzarzal	700
C-305	Collado Mediano	350

Figura 22: Ejemplo de Relación

Tiene tres cabeceras de columna: número\_cuenta, nombre\_sucursal y saldo.

- Siguiendo la terminología del modelo relacional, se puede hacer referencia a estas cabeceras como atributos.
- Como vimos anteriormente, para cada atributo hay un conjunto de valores permitidos, denominado dominio de ese atributo. Para el atributo **nombre\_sucursal**, por ejemplo, el dominio es el conjunto de todos los nombres de sucursal:
  - Supóngase que D1 denota el conjunto de todos los números de cuenta, D2 el conjunto de todos los nombres de sucursal y D3 el conjunto de todos los saldos.
  - Todas las filas de cuenta deben consistir en una **tupla formada por tres valores** correspondientes a dichos dominios (**v1; v2; v3**), donde v1 es un número de cuenta (es decir, v1 está en el dominio D1), v2 es un nombre de sucursal (v2 en D2) y v3 es un saldo (v3 en D3).
- Como las tablas son, esencialmente, relaciones, se usarán los términos matemáticos **relación y tupla** en lugar de los términos **tabla y fila**.

A continuación, nos detendremos en el concepto de relaciones, ya que es la esencia de este Modelo Lógico Relacional:

- **Propiedades de las Relaciones:**
  - No existen tuplas repetidas.
  - Las tuplas no tienen que estar ordenadas, necesariamente, (de arriba hacia abajo).
  - Los atributos no tienen que estar ordenados, necesariamente, (de izquierda a derecha).
  - Todos los valores de los atributos son atómicos (esto significa que debe tener un único valor, por ejemplo si es un atributo que contendrá un número de teléfono, solo puede contener un número de teléfono).

## Integridad en las Bases de Datos Relacionales

El objetivo de las reglas de integridad es informar al sistema administrador de Bases de Datos (DBMS), de ciertas restricciones del mundo real, (por ejemplo, los pesos de las piezas no pueden ser negativos).

Se debe vigilar las operaciones de inserción y modificación y rechazar cualquier entrada que no cumpla con las especificaciones. La mayoría de las Bases de Datos están sujetas a tales reglas de integridad. Las reglas de integridad son específicas para cada Base de Datos.

El Modelo Relacional incluye dos reglas generales de integridad (generales porque se aplican a todas las bases de datos apegadas a este modelo). Estas dos reglas se refieren a las claves primarias y las claves ajenas o foráneas.

Algunas consideraciones introductorias:

- Las reglas de integridad se aplican a las relaciones base, que son las que supuestamente reflejan la realidad.
- Existen otras reglas específicas con respecto a la integridad de una base de datos en particular (dependientes de cada dominio del problema).

### Reglas de integridad para Clave Primaria

La clave primaria (PRIMARY KEY, en inglés) es sólo un **identificador único** para una relación, es decir, nunca existen dos filas de la relación con el mismo valor en esa columna o columnas. Decimos columnas, en plural ya que la clave primaria puede ser compuesta.

Puede ocurrir que la clave primaria sea la combinación de todos los atributos de la relación, la relación sería ese caso “toda clave”.

También es posible tener una relación con más de un identificador único, en ese caso la relación tiene claves candidatas. **Un atributo a (posiblemente compuesto) de la relación R es una clave candidata de R, si y solo si satisface las siguientes propiedades:**

- **Unicidad:** no existen dos tuplas de R con el mismo valor de **a**, en un momento dado.
- **Minimalidad:** si **a** es un atributo compuesto, no puedo eliminar un componente de **a** sin destruir la propiedad de unicidad.

En el caso de que existan varias claves candidatas, debemos escoger una y las demás serán claves Alternativas (claves candidata que no son clave primaria).

Por ejemplo, a partir de la relación cliente de la figura 23, el atributo id\_cliente de la relación cliente es suficiente para distinguir una tupla cliente de otra. Por tanto, id\_cliente es una clave candidata. De manera parecida, la combinación de nombre\_cliente e id\_cliente constituye una clave candidata para la relación cliente. El atributo nombre\_cliente de cliente no es una clave candidata, ya que es posible que varias personas se llamen igual. Además, podemos pensar que una combinación de los atributos nombre\_cliente y de calle\_cliente sea suficiente para distinguir entre los miembros de la relación cliente. Aunque los atributos id\_cliente y nombre\_cliente en conjunto pueden diferenciar las tuplas cliente, su combinación no forma una clave candidata, ya que el atributo id\_cliente por sí solo ya lo es.

Las claves (sean primarias, candidatas o superclaves<sup>11</sup>) son propiedades de toda la relación, no de cada una de las tuplas.

<sup>11</sup> **Superclave:** una superclave de una relación de esquema  $R(A_1, A_2, \dots, A_n)$  es un subconjunto de los atributos del esquema tal que no puede haber dos tuplas en la extensión de la relación que tengan la misma combinación de valores para los atributos del subconjunto. Toda relación tiene, por lo menos, una superclave, que es la formada por todos los atributos de su esquema. Esto se debe a la propiedad que cumple toda relación de no tener tuplas repetidas. Una superclave, por lo tanto, nos permite identificar todas las tuplas que contiene la relación.

<i>id_cliente</i>	<i>nombre_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>
19.283.746	González	Arenal, 12	La Granja
67.789.901	López	Mayor, 3	Peguerinos
18.273.609	Abril	Preciados, 123	Valsaín
32.112.312	Santos	Mayor, 100	Peguerinos
33.666.999	Rupérez	Ramblas, 175	León
01.928.374	Gómez	Carretas, 72	Cerceda

Figura 23: Relación Cliente

**Consideraciones:**

- Toda relación tendrá por fuerza una clave primaria.
- La clave primaria es la que tiene verdadera importancia; las demás, claves candidatas y alternativas, son sólo conceptos que surgen en el proceso de decisión.
- Al nombrar al atributo identificador, es conveniente hacerlo como clave primaria, porque hay muchos tipos de claves.

**¿Por qué son importantes las Claves Primarias?**

- Son importantes porque constituyen el **mecanismo de direccionamiento** a nivel de tuplas, básico en un sistema relacional.
- Es el único modo garantizado por el sistema para **localizar una tupla específica**.
- **Ningún** componente de la clave primaria de una relación base puede **aceptar nulos**. Se entiende por nulo a un valor o representación que, por convención, **no representa valor real del atributo aplicado**.

*Regla de integridad en las relaciones*

En una base de datos relacional, **nunca** registraremos información de algo que **no podamos identificar**. Consideraciones para esta regla:

- Para las claves primarias compuestas: cada valor individual de la clave primaria debe ser no nulo en su totalidad.
- Esta regla se aplica a las relaciones base únicamente.
- Se aplica sólo a las claves primarias.

En caso de que la clave primaria fuera compuesta, se utiliza la abreviación PK para cada atributo que forma parte de la clave primaria.

*Reglas de integridad para Claves Ajenas*

La clave ajena (FOREIGN KEY, en inglés) es un atributo (quizás compuesto) de una relación R2 cuyos valores deben concordar con los de la clave primaria de alguna otra relación R1 (no se requiere el caso inverso). Ejemplo: en la relación **Empleado** hace referencia al departamento en donde trabaja el empleado por medio de una clave foránea *id\_departamento* que apunta a la clave primaria *id\_departamento* de la relación **Departamento**.

La integridad referencial se formula en términos de estados de la base de datos. Cualquier estado que no satisfaga la regla será incorrecto. ¿Cómo pueden evitarse estos estados? La regla



no lo dice. La idea fundamental es la siguiente: para cada clave ajena es necesario responder tres preguntas:

1. ¿Puede aceptar nulos esa clave ajena? Ej. Empleado no asignado de momento a ningún departamento. La respuesta depende de las políticas vigentes en el mundo real representado por la base de datos.

2. ¿Qué deberá suceder si hay un intento de eliminar el objetivo de una referencia de clave ajena? Ej. Eliminar un departamento que tiene varios Empleados; existen tres posibilidades:

Restringida, si existen empleados.

Se propaga: se eliminan los empleados

Anula: se asignan nulos a la clave ajena de todos los empleados.

¿Qué deberá suceder si hay un intento de modificar la clave primaria del objetivo de una referencia? Modificar el número de departamento al cual pertenece un empleado.

## Motores de bases de datos relacionales

---

Como vimos anteriormente, el modelo relacional es actualmente, el principal modelo de datos para las aplicaciones comerciales de procesamiento de datos. Ha conseguido esa posición destacada debido a su simplicidad, lo cual facilita el trabajo del programador en comparación con modelos anteriores, como el de red y el jerárquico.

Una base de datos relacional consiste en un conjunto de **tablas**, a cada una de las cuales se le asigna un nombre exclusivo. Cada fila de la tabla representa una relación entre un conjunto de valores. De manera informal, cada tabla es un conjunto de instancias de entidades, y cada fila es una instancia. Dado que cada tabla es un conjunto de tales relaciones, hay una fuerte correspondencia entre el concepto de **tabla** y el concepto matemático de **relación**, del que toma su nombre el modelo de datos relacional.

Cuando se habla de bases de datos se debe diferenciar entre el **esquema de la base de datos**, que es el diseño lógico de la misma, y el **ejemplar de la base de datos**, que es una instantánea de los datos de la misma en un momento dado.

El concepto de relación se corresponde con el concepto de variable de los lenguajes de programación. El concepto de **esquema de la relación** se corresponde con el concepto de definición de tipos de los lenguajes de programación.

Resulta conveniente dar nombre a los esquemas de las relaciones, igual que se dan nombres a las definiciones de los tipos en los lenguajes de programación. En general, los esquemas de las relaciones consisten en una lista de los atributos y de sus dominios correspondientes.

El concepto de **ejemplar de la relación** se corresponde con el concepto de valor de una variable en los lenguajes de programación. El valor de una variable dada puede cambiar con el tiempo; de manera parecida, el contenido del ejemplar de una relación puede cambiar con el tiempo cuando la relación se actualiza. Sin embargo, se suele decir simplemente “relación” cuando realmente se quiere decir “ejemplar de la relación”.

## Lenguajes de consultas

Un lenguaje de consultas es un lenguaje en el que los usuarios solicitan información de la base de datos. Estos lenguajes suelen ser de un nivel superior que el de los lenguajes de programación habituales. **Los lenguajes de consultas pueden clasificarse como procedimentales o no procedimentales.**

En los **lenguajes procedimentales** el usuario indica al sistema que lleve a cabo una serie de operaciones en la base de datos para calcular el resultado deseado. En los **lenguajes no procedimentales** el usuario describe la información deseada sin dar un procedimiento concreto para obtener esa información.

La mayor parte de los sistemas comerciales de bases de datos relacionales ofrecen un lenguaje de consultas que incluye elementos de los enfoques procedimental y no procedimental. En la sección siguiente estudiaremos en profundidad el lenguaje de consulta estructurado o SQL (Structured Query Language). Existen varios lenguajes de consultas “puros”: el álgebra relacional es procedimental, mientras que el cálculo relacional de tuplas y el cálculo relacional de dominios no lo son. Estos lenguajes de consultas son rígidos y formales, y carecen del “azúcar sintáctico” de los lenguajes comerciales, pero ilustran las técnicas fundamentales para la extracción de datos de las bases de datos. Un lenguaje de manipulación de datos completo no sólo incluye un lenguaje de consultas, sino también un lenguaje para la modificación de las bases de datos. Este tipo de lenguajes incluye comandos para insertar y borrar tuplas, así como para modificar partes de las tuplas existentes.

## Alternativas de motores de bases de datos relacionales

En los negocios, se utilizan bases de datos relacionales para mantener los datos de sus aplicaciones y para asegurar que siempre se tendrá acceso a la información crítica de clientes, datos de productos, datos sociales y financieros particulares, tales como compras, cumplimientos beneficios y gastos. Estos sistemas pueden ir desde aplicaciones de escritorio que crean una base de datos pequeña en una máquina, hasta almacenes de datos de empresas de gran escala. Casi todas utilizan una forma de SQL como su lenguaje de consulta, y la mayoría adhiere al conjunto de propiedades ACID para asegurar la confiabilidad de las transacciones (atomicidad, consistencia, aislamiento y durabilidad).

Las bases de datos relacionales son elegidas generalmente debido a su simplicidad en comparación con las bases de datos NoSQL, tales como las bases de datos de objetos, documentales, o gráficas. Un producto para calificar y ser incluido en la categoría de base de datos relacional debe:

- Proveer almacenamiento de datos.
- Organizar los datos en un modelo relacional, formulado con tablas de filas y columnas.
- Permitir a todos los usuarios recuperar, editar, retornar y remover datos.

Los mejores productos de software de bases de datos relacionales están determinados por el nivel de satisfacción (basado en revisiones de los usuarios) y escala (basada en porción de mercado, tamaño del vendedor e impacto social).

<input type="checkbox"/>		Amazon RDS
<input type="checkbox"/>		DB2
<input type="checkbox"/>		HPE Vertica
<input type="checkbox"/>		Informix
<input type="checkbox"/>		MariaDB
<input type="checkbox"/>		Microsoft SQL
<input type="checkbox"/>		MySQL
<input type="checkbox"/>		Oracle Database 12c
<input type="checkbox"/>		PostgreSQL
<input type="checkbox"/>		SAP HANA
<input type="checkbox"/>		SQLite
<input type="checkbox"/>		Teradata Database

*Figura 24: Algunos Motores de Bases de Datos Comerciales*

**En el desarrollo de la parte práctica de este módulo trabajaremos con el motor de bases de datos MySQL.**

## Transacciones

Desde los inicios de la Informática se guardaron datos para ser recuperados y procesados más tarde, pero muchísimas veces esos datos eran inconsistentes y causaban problemas muy graves.

Veamos un ejemplo: Se tiene que grabar una venta y para ello se tienen dos tablas:

- Cabecera de ventas
- Detalles de ventas

Se grabó la cabecera y antes de que se grabaran todos los detalles ocurrió un corte en la energía eléctrica o se dañó la red o algún otro problema que impidió grabar todos los detalles de dicha venta. Eso implica que algo está muy mal: la cabecera no corresponde exactamente con los detalles. Y eso es inaceptable.

A veces, ese tipo de error puede ser detectado y corregido enseguida, pero a veces pueden pasar días, meses o inclusive años antes de ser detectado. Este tipo de problemas y muchos más causaban serios perjuicios a las empresas, motivo por el cual muchas personas se dedicaron a investigar cómo solucionarlos. A finales de los años 1970 Jim Gray definió las propiedades que debía tener una transacción confiable y en 1983 Andreas Reuter y Theo Härder inventaron la palabra ACID para describir a ese tipo de transacción. ACID es el acrónimo de Atomicity, Consistency, Isolation, Durability (en castellano: Atomicidad, Consistencia, Aislamiento, Durabilidad), conceptos ya mencionados con anterioridad.

El concepto de transacción proporciona un mecanismo para definir las **unidades lógicas del procesamiento de una base de datos**. Los sistemas de procesamiento de transacciones son sistemas con grandes bases de datos y cientos de usuarios concurrentes ejecutando transacciones de bases de datos. Entre estos sistemas podemos citar los de reservas en aerolíneas, bancos, procesamiento de tarjetas de crédito, mercado de acciones, etcétera. Estos

sistemas requieren una alta disponibilidad y una respuesta rápida para cientos de usuarios simultáneos.

*Una transacción es un programa en ejecución que constituye una unidad lógica del procesamiento de una base de datos. Una transacción incluye una o más operaciones de acceso a la base de datos (operaciones de inserción, eliminación, modificación o recuperación). Las operaciones con bases de datos que forman una transacción pueden estar incrustadas dentro de una aplicación o pueden especificarse interactivamente mediante un lenguaje de consulta de alto nivel como SQL.*

¿Por qué es necesaria la recuperación?

Siempre que se envía una transacción a un DBMS para su ejecución, el sistema es responsable de garantizar que todas las operaciones de la transacción se completen satisfactoriamente y que su efecto se grave permanentemente en la base de datos, o de que la transacción no afecte a la base de datos o a cualquier otra transacción. El DBMS no debe permitir que algunas operaciones de una transacción *T* se apliquen a la base de datos mientras otras no. Esto puede ocurrir si una transacción **falla** después de ejecutar algunas de sus operaciones, pero antes de ejecutar todas ellas.

Tipos de fallos:

Los fallos se clasifican generalmente como fallos de transacción del sistema y del medio. Hay varias razones posibles por las que una transacción puede fallar en medio de su ejecución:

1. **Un fallo de la computadora (caída del sistema).** Durante la ejecución de una transacción se produce un error del hardware, del software o de la red. Las caídas del hardware normalmente se deben a fallos en los medios (por ejemplo, un fallo de la memoria principal).
2. **Un error de la transacción o del sistema.** alguna operación de la transacción puede provocar que falle, como un desbordamiento de entero o una división por cero. El fallo de una transacción también puede deber a unos valores erróneos de los parámetros o debido a un error lógico de programación. Además, el usuario puede interrumpir la transacción durante su ejecución.
3. **Errores locales o condiciones de excepción detectados por la transacción.** Durante la ejecución de una transacción, se pueden dar ciertas condiciones que necesitan cancelar la transacción. Por ejemplo, puede que no se encuentren los datos para la transacción. Una condición de excepción, como un saldo de cuenta insuficiente en una base de datos bancaria, puede provocar que una transacción, como la retirada de fondos, sea cancelada. Esta excepción debe programarse en la propia transacción, en cuyo caso, no sería considerada un fallo.
4. **Control de la concurrencia.** El método de control de la concurrencia puede optar por abortar la transacción, para restablecerla más tarde, porque viola la serialización o porque varias transacciones se encuentran en estado de bloqueo.
5. **Fallo del disco.** Algunos bloques del disco pueden perder sus datos debido a un mal funcionamiento de la lectura o la escritura o porque se ha caído la cabeza de lectura/escritura del disco. Esto puede ocurrir durante una operación de lectura o escritura de la transacción.

6. **Problemas físicos y catástrofes.** Se refiere a una lista interminable de problemas que incluye fallos de alimentación o aire acondicionado, fuego, robo, sabotaje, sobrescritura de discos y cintas por error, y montaje de la cinta errónea por parte del operador.

Los fallos de los tipos 1, 2, 3 y 4 son más comunes que los de los tipos 5 o 6. Siempre que se produce un fallo de tipo 1 a 4, el sistema debe guardar información suficiente para recuperarse del fallo. El fallo de un disco u otros fallos catastróficos de tipo 5 o 6 no se dan con frecuencia; si ocurren, la recuperación es la tarea principal. El concepto de transacción es fundamental para muchas técnicas de control de la concurrencia y de recuperación ante fallos.

ACID, letra por letra en las transacciones

ACID también se aplica a las transacciones, las propiedades deben ser implementadas por el control de la concurrencia y los métodos de recuperación del DBMS. Las propiedades ACID son las siguientes:

#### **Atomicidad**

La *Atomicidad* requiere que cada transacción sea "todo o nada": si una parte de la transacción falla, todas las operaciones de la transacción fallan, y por lo tanto la base de datos no sufre cambios. Un sistema atómico tiene que garantizar la atomicidad en cualquier operación y situación, incluyendo fallas de alimentación eléctrica, errores y caídas del sistema.

#### **Consistencia**

La propiedad de *Consistencia* se asegura que cualquier transacción llevará a la base de datos de un estado válido a otro estado válido. Cualquier dato que se escriba en la base de datos tiene que ser válido de acuerdo a todas las reglas definidas.

#### **Aislamiento**

El *Aislamiento* ("Isolation" en inglés) se asegura que la ejecución concurrente de las transacciones resulte en un estado del sistema que se obtendría si estas transacciones fueran ejecutadas una atrás de otra. Cada transacción debe ejecutarse en aislamiento total; por ejemplo, si T1 y T2 se ejecutan concurrentemente, luego cada una debe mantenerse independiente de la otra.

#### **Durabilidad**

La *Durabilidad* significa que una vez que se confirmó una transacción quedará persistida, incluso ante eventos como pérdida de alimentación eléctrica, errores y caídas del sistema. Por ejemplo, en las bases de datos relacionales, una vez que se ejecuta un grupo de sentencias SQL, los resultados tienen que almacenarse inmediatamente (incluso si la base de datos se cae inmediatamente después).

## Estados de una transacción y operaciones adicionales

Una transacción es una unidad atómica de trabajo que se completa en su totalidad o no se lleva a cabo en absoluto. Para fines de recuperación, el sistema debe hacer el seguimiento de cuándo se inicia, termina y confirma o aborta una transacción. Por consiguiente, el gestor de recuperación hace un seguimiento de las siguientes operaciones:

- **BEGIN\_TRANSACTION.** Marca el inicio de la ejecución de una transacción.
- **READ o WRITE.** Especifican operaciones de lectura o escritura en los elementos de la base de datos que se ejecutan como parte de una transacción.
- **END\_TRANSACTION.** Especifica que las operaciones READ y WRITE de la transacción han terminado y marca el final de la ejecución de la transacción. Sin embargo, en este punto puede ser necesario comprobar si los cambios introducidos por la transacción pueden aplicarse de forma permanente a la base de datos (confirmados) o si la transacción se ha cancelado porque viola la serialización o por alguna otra razón.
- **COMMIT\_TRANSACTION.** Señala una *finalización satisfactoria* de la transacción, por lo que los cambios (actualizaciones) ejecutados por la transacción se pueden enviar con seguridad a la base de datos y no se desharán.
- **ROLLBACK (o ABORT).** Señala que la transacción *no ha terminado satisfactoriamente*, por lo que deben *deshacerse* los cambios o efectos que la transacción pudiera haber aplicado a la base de datos.

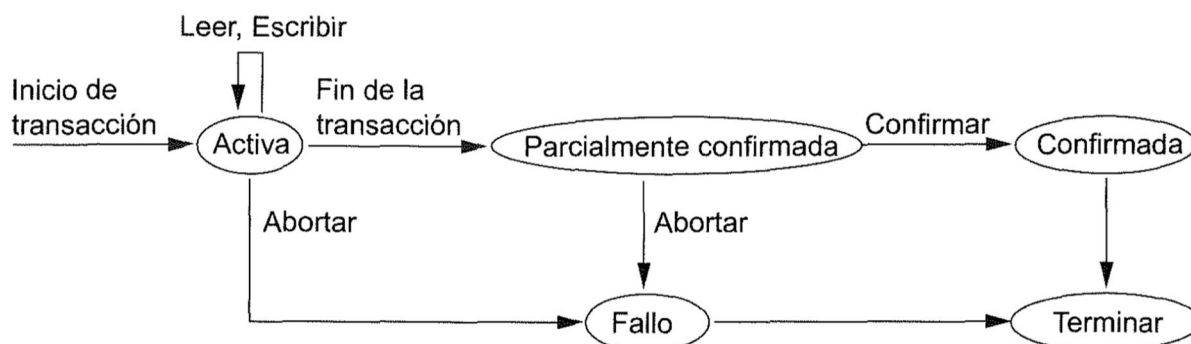


Fig. 25: Estados de una transacción y sus operaciones

Una transacción entra en estado activo inmediatamente después de iniciarse su ejecución; en este estado puede emitir operaciones READ y WRITE. Cuando la transacción termina, pasa al estado de parcialmente confirmada. En este punto, se necesitan algunos protocolos de recuperación para garantizar que un fallo del sistema no supondrá la imposibilidad de registrar los cambios de la transacción de forma permanente. Una vez que esta comprobación es satisfactoria, se dice que la transacción ha alcanzado su punto de confirmación y entra en el estado de confirmada. Una vez confirmada la transacción, ha concluido satisfactoriamente su ejecución y todos sus cambios deben grabarse permanentemente en la base de datos.

No obstante, una transacción puede entrar en el estado de fallo si falla una de las comprobaciones o si la transacción es cancelada durante su estado activo. Entonces es posible anular la transacción para deshacer el efecto de sus operaciones de escritura en la base de datos.

El estado terminado se alcanza cuando la transacción abandona el sistema. La información relativa a la transacción que se guarda en tablas del sistema mientras se está ejecutando, se elimina cuando la transacción termina. Las transacciones fallidas o canceladas pueden *restablecerse* más tarde (automáticamente o después de haber sido reenviadas por el usuario) como transacciones completamente nuevas.

Los programadores de SQL son los responsables de iniciar y finalizar las transacciones en puntos que exijan la coherencia lógica de los datos. El programador debe definir la secuencia de modificaciones de datos que los dejan en un estado coherente en relación con las reglas de negocios de la organización. Además, debe incluir estas instrucciones de modificación en una sola transacción de forma que el Motor de base de datos puede hacer cumplir la integridad física de la misma.

Es responsabilidad de un sistema de base de datos corporativo, como una instancia de Motor de base de datos, proporcionar los mecanismos que aseguren la integridad física de cada transacción. En términos generales, deberá proporcionar:

- Servicios de bloqueo que preservan el aislamiento de la transacción.
- Servicios de registro que aseguran la durabilidad de la transacción. Aunque se produzca un error en el hardware del servidor, el sistema operativo o la instancia de Motor de base de datos, la instancia utiliza registros de transacciones, al reiniciar, para revertir automáticamente las transacciones incompletas al punto en que se produjo el error del sistema.
- Características de administración de transacciones que exigen la atomicidad y coherencia de la transacción. Una vez iniciada una transacción, debe concluirse correctamente; en caso contrario, la instancia de Motor de base de datos deshacerá todas las modificaciones de datos realizadas desde que se inició la transacción.

Los sistemas de procesamiento de transacciones permiten normalmente la ejecución de varias transacciones concurrentemente. Permitir varias transacciones que actualizan concurrentemente los datos provoca complicaciones en la consistencia de los mismos. Asegurar la consistencia a pesar de la ejecución concurrente de las transacciones requiere un trabajo extra; es mucho más sencillo exigir que las transacciones se ejecuten **secuencialmente** — es decir, una a una, comenzado cada una sólo después que la anterior se haya completado. Sin embargo, existen dos buenas razones para permitir la concurrencia:

- **Productividad y utilización de recursos mejorados.** Una transacción consiste en varios pasos. Algunos implican operaciones de E/S; otros implican operaciones de CPU. La CPU y los discos pueden trabajar en paralelo en una computadora. Por tanto, las operaciones de E/S se pueden realizar en paralelo con el procesamiento de la CPU. Se puede entonces explotar el paralelismo de la CPU y del sistema de E/S para ejecutar varias transacciones en paralelo. Mientras una transacción ejecuta una lectura o una escritura en un disco, otra puede ejecutarse en la CPU mientras una tercera transacción ejecuta una lectura o una escritura en otro disco. Todo esto incrementa la **productividad (throughput)** del sistema—es decir, en el número de transacciones que puede ejecutar en un tiempo dado. Análogamente, la **utilización** del procesador y del disco aumenta

también; en otras palabras, el procesador y el disco están más desocupados o sin hacer ningún trabajo útil.

- **Tiempo de espera reducido.** Debe haber una mezcla de transacciones que se ejecutan en el sistema, algunas cortas y otras largas. Si las transacciones se ejecutan secuencialmente, la transacción corta debe esperar a que la transacción larga anterior se complete, lo cual puede llevar a un retardo impredecible en la ejecución de la transacción. Si las transacciones operan en partes diferentes de la base de datos es mejor hacer que se ejecuten concurrentemente, compartiendo los ciclos de la CPU y los accesos a disco entre ambas. La ejecución concurrente reduce los retardos impredecibles en la ejecución de las transacciones. Además, se reduce también el **tiempo medio de respuesta**: el tiempo medio desde que una transacción comienza hasta que se completa.

La razón para usar la ejecución concurrente en una base de datos es esencialmente la misma que para usar **multiprogramación** en un sistema operativo.

Cuando se ejecutan varias transacciones concurrentemente, la consistencia de la base de datos se puede destruir a pesar de que cada transacción individual sea correcta. El sistema de base de datos debe controlar la interacción entre las transacciones concurrentes para evitar que se destruya la consistencia de la base de datos. Esto se lleva a cabo a través de una serie de mecanismos denominados **esquemas de control de concurrencia**.

## Índices

Muchas consultas hacen referencia sólo a una pequeña parte de los registros de un archivo. Por ejemplo, la consulta “Buscar todas las cuentas de la sucursal Pamplona” o “Buscar el saldo del número de cuenta C-101” hace referencia solamente a una fracción de los registros de la relación **cuenta**. No es eficiente que el sistema tenga que leer cada registro y comprobar que el campo **nombre\_sucursal** contiene el nombre “Pamplona” o el valor C-101 del campo **número\_cuenta**. Lo más adecuado sería que el sistema fuese capaz de localizar directamente esos registros. Para facilitar estas formas de acceso se diseñan estructuras adicionales que se asocian con los archivos.

Un índice para un archivo del sistema funciona como el índice de un libro. Si se va a buscar un tema (especificado por una palabra o una frase) se puede buscar en el índice al final del libro, encontrar las páginas en las que aparece y después leerlas para encontrar la información buscada. Las palabras del índice están ordenadas alfabéticamente, lo cual facilita la búsqueda. Además, el índice es mucho más pequeño que el libro, con lo que se reduce aún más el esfuerzo necesario para encontrar las palabras en cuestión.

Los índices de los sistemas de bases de datos juegan el mismo papel que los índices de los libros en las bibliotecas. Por ejemplo, para recuperar un registro **cuenta** dado su número de cuenta, el sistema de bases de datos buscaría en un índice para encontrar el bloque de disco en que se localice el registro correspondiente, y entonces extraería ese bloque de disco para obtener el registro **cuenta**.

Almacenar una lista ordenada de números de cuenta no funcionaría bien en bases de datos muy grandes con millones de cuentas, ya que el propio índice sería muy grande; más aún, incluso



mantener ordenado el índice reduce el tiempo de búsqueda, por lo que encontrar una cuenta podría consumir mucho tiempo. En su lugar, se usan técnicas más sofisticadas de indexación. Algunas de estas técnicas se estudiarán más adelante.

Hay dos tipos básicos de índices:

- **Índices ordenados:** estos índices están basados en una disposición ordenada de los valores.
- **Índices asociativos:** estos índices están basados en una distribución uniforme de los valores a través de una serie de cajones (buckets). El valor asignado a cada cajón está determinado por una función, llamada función de asociación (hash function).

Se considerarán varias técnicas de indexación y asociación. Ninguna de ellas es la mejor. Sin embargo, para cada aplicación específica de bases de datos existe una técnica más apropiada. Cada una de ellas debe ser valorada según los siguientes criterios:

- **Tipos de acceso:** los tipos de acceso que se soportan eficazmente. Estos tipos podrían incluir la búsqueda de registros con un valor concreto en un atributo, o la búsqueda de los registros cuyos atributos contengan valores en un rango especificado.
- **Tiempo de acceso:** el tiempo que se tarda en hallar un determinado elemento de datos, o conjunto de elementos, usando la técnica en cuestión.
- **Tiempo de inserción:** el tiempo empleado en insertar un nuevo elemento de datos. Este valor incluye el tiempo utilizado en hallar el lugar apropiado donde insertar el nuevo elemento de datos, así como el tiempo empleado en actualizar la estructura del índice.
- **Tiempo de borrado:** el tiempo empleado en borrar un elemento de datos. Este valor incluye el tiempo utilizado en hallar el elemento a borrar, así como el tiempo empleado en actualizar la estructura del índice.
- **Espacio adicional requerido:** el espacio adicional ocupado por la estructura del índice. Como normalmente la cantidad necesaria de espacio adicional suele ser moderada, es razonable sacrificar el espacio para alcanzar un rendimiento mejor.

A menudo se desea tener más de un índice por archivo. Por ejemplo, se puede buscar un libro según el autor, la materia o el título. Los atributos o conjunto de atributos usados para buscar en un archivo se llaman **claves de búsqueda**. Hay que observar que esta definición de **clave** difiere de la usada en **clave primaria, clave candidata y superclave**. Este doble significado de **clave** está (desafortunadamente) muy extendido en la práctica. Usando el concepto de clave de búsqueda se determina que, si existen varios índices para un archivo, existirán varias claves de búsqueda.

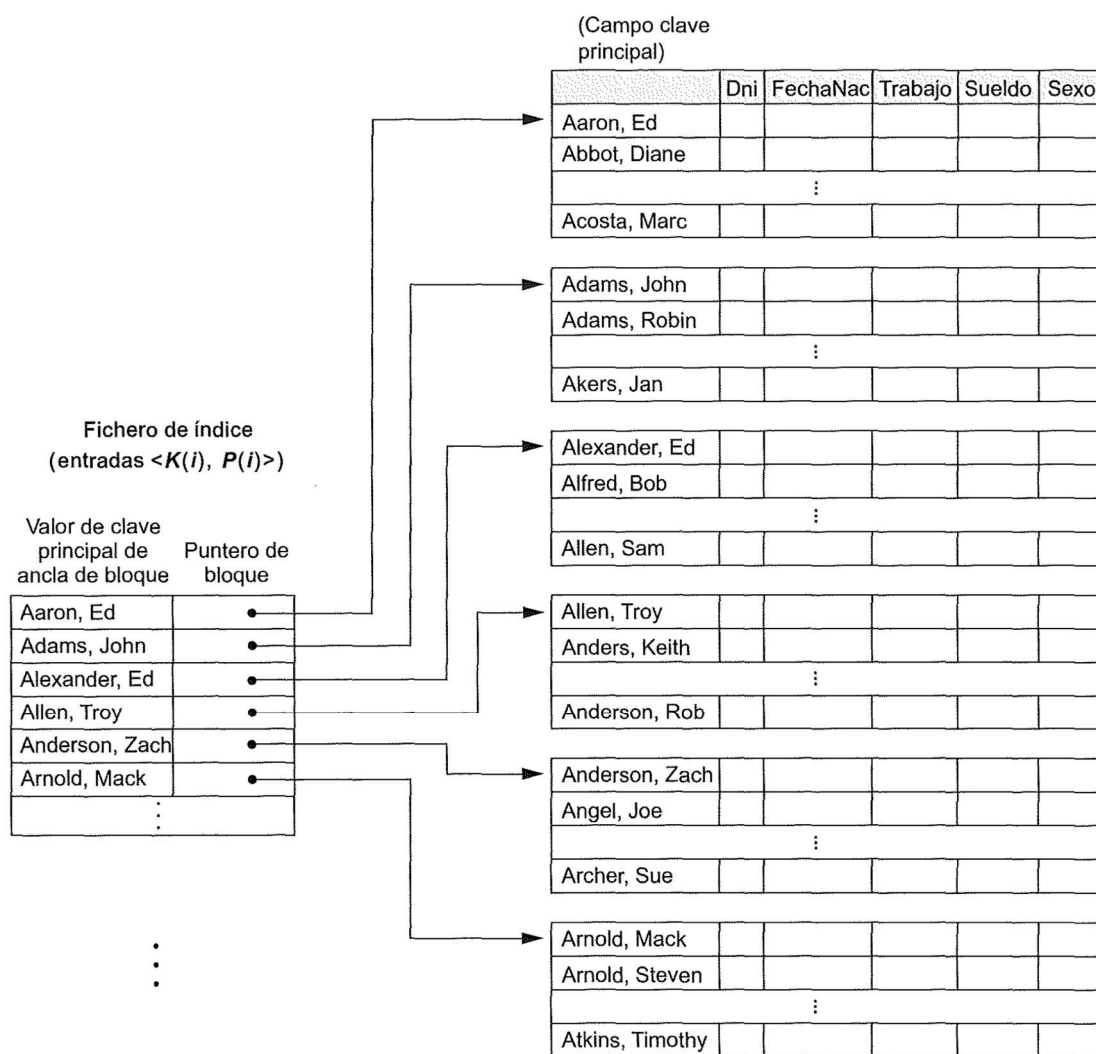


Fig. 26: Un ejemplo de estructura de índice para la tabla 'empleados'

En un fichero con una estructura de registro dada compuesta por varios campos (o atributos), normalmente se define una estructura de índice con un solo campo del fichero, que se conoce como **campo de indexación (o atributo de indexación)**. Normalmente, el índice almacena todos los valores del campo de índice, junto con una lista de punteros a todos los bloques de disco que contienen los registros con ese valor de campo. Los valores del índice están ordenados, de modo que podemos hacer una búsqueda binaria en el índice. El fichero de índice es mucho más pequeño que el fichero de datos, porque la búsqueda en el índice mediante una búsqueda binaria es razonablemente eficaz. La indexación multinivel anula la necesidad de una búsqueda binaria a expensas de crear índices al propio índice.

Hay varios tipos de índices ordenados. Un **índice principal o primario** se especifica en el *campo clave de ordenación* de un fichero ordenado de registros. Se utiliza un campo clave de ordenación para *ordenar físicamente* los registros del fichero en el disco, y cada registro tiene un *valor único* para ese campo. Si el campo de ordenación no es un campo clave (es decir, si varios registros del fichero tienen el mismo valor para el campo de ordenación), podemos utilizar otro tipo de índice, denominado **índice agrupado**. Un fichero puede tener como máximo un campo de ordenación física, por lo que puede tener a lo sumo un índice principal o un índice

agrupado, *pero no ambos*. Un tercer tipo de índice, el **índice secundario**, se puede especificar sobre cualquier campo no ordenado de un fichero. Un fichero puede tener varios índices secundarios además de su método de acceso principal.

En algunas ocasiones puede que el propio índice sea demasiado grande para un procesamiento eficiente. En la práctica no es excesivo tener un archivo con 100.000 registros, con 10 registros almacenados en cada bloque. Si se dispone de un registro índice por cada bloque, el índice tendrá 10.000 registros. Como los registros índices son más pequeños que los registros de datos, se puede suponer que caben 100 registros índices en un bloque. Por tanto, el índice ocuparía 100 bloques. Estos índices de mayor tamaño se almacenan como archivos secuenciales en disco.

Si un índice es lo bastante pequeño como para guardarlo en la memoria principal, el tiempo de búsqueda para encontrar una entrada será breve. Sin embargo, si el índice es tan grande que se debe guardar en disco, buscar una entrada implicará leer varios bloques de disco. Para localizar una entrada en el archivo índice se puede realizar una búsqueda binaria, pero aun así ésta conlleva un gran costo.

Para resolver este problema, el índice se trata como si fuese un archivo secuencial y se construye otro índice sobre el índice con agrupación. Para localizar un registro se usa en primer lugar una búsqueda binaria sobre el índice más externo para buscar el registro con el mayor valor de la clave de búsqueda que sea menor o igual al valor deseado. El puntero apunta a un bloque en el índice más interno. Hay que examinar este bloque hasta encontrar el registro con el mayor valor de la clave que sea menor o igual que el valor deseado. El puntero de este registro apunta al bloque del archivo que contiene el registro buscado. Si el archivo es extremadamente grande, incluso el índice exterior podría crecer demasiado para caber en la memoria principal. En este caso se podría crear todavía otro nivel más de indexación. De hecho, se podría repetir este proceso tantas veces como fuese necesario. Los índices con dos o más niveles se llaman **índices multinivel**.

Un diccionario normal es un ejemplo de un índice multinivel en el mundo ajeno a las bases de datos. La cabecera de cada página lista la primera palabra en el orden alfabético en esa página. Este índice es multinivel: las palabras en la parte superior de la página del índice del libro forman un índice sobre los contenidos de las páginas del diccionario.

Los índices multinivel están estrechamente relacionados con la estructura de árbol, tales como los árboles binarios usados para la indexación en memoria.

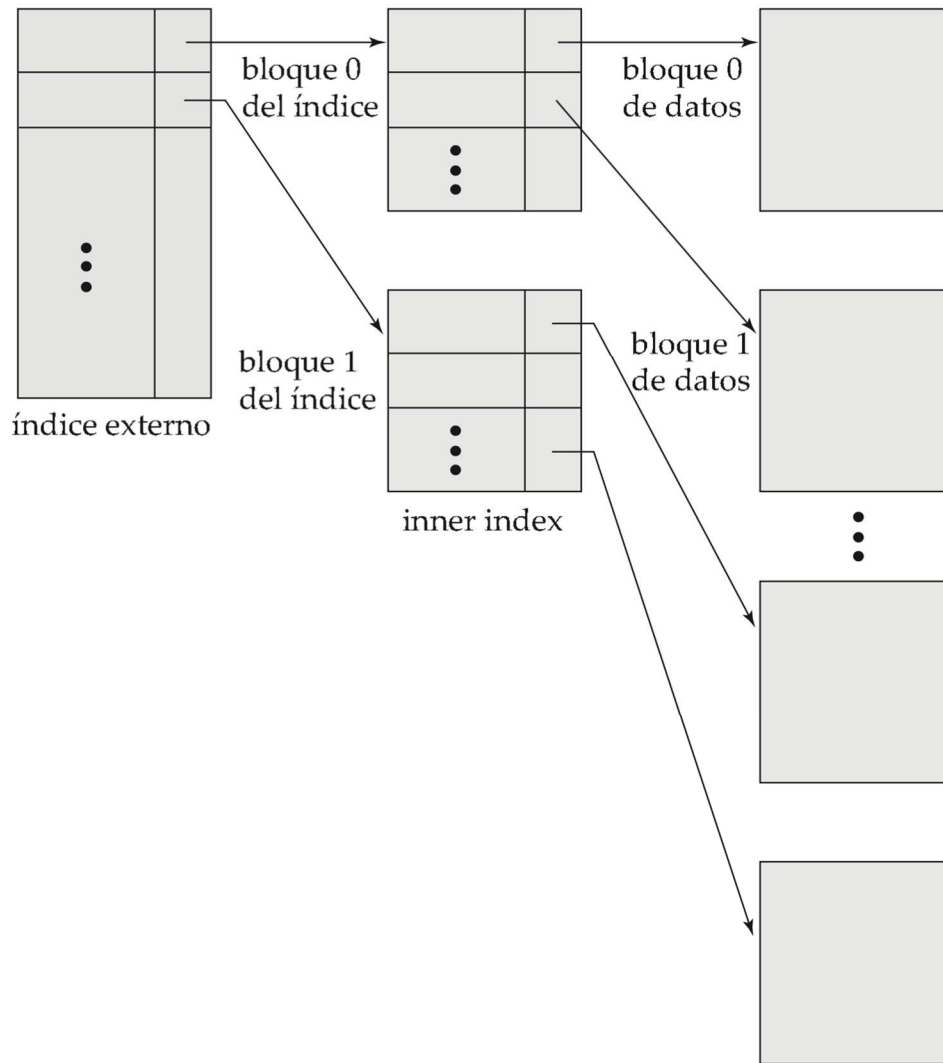


Fig.27: Representación de un índice de dos niveles

En resumen, muchas consultas solamente hacen referencia a una pequeña proporción de los registros de un archivo. Para reducir el gasto adicional en la búsqueda de estos registros se pueden construir *índices* para los archivos almacenados en la base de datos. Los archivos secuenciales indexados son uno de los esquemas de índice más antiguos usados en los sistemas de bases de datos. Para permitir una rápida recuperación de los registros según el orden de la clave de búsqueda, los registros se almacenan consecutivamente y los que no siguen el orden se encadenan entre sí. Para permitir un acceso aleatorio, se usan estructuras de índice.

## Procedimientos almacenados y funciones

En ocasiones es útil crear módulos de programa de bases de datos (procedimientos o funciones) que el DBMS almacena y ejecuta en el servidor de bases de datos. Es lo que históricamente se conoce como **procedimientos almacenados** (en inglés *stored procedures*) de bases de datos, aunque pueden ser funciones o procedimientos. El término utilizado en el estándar SQL para los procedimientos almacenados es **módulos almacenados persistentes**, porque el DBMS almacena persistentemente estos programas, algo parecido a los datos persistentes almacenados por el DBMS.

Los procedimientos almacenados son útiles en las siguientes circunstancias:

- Si varias aplicaciones necesitan un mismo programa de base de datos, este último se puede almacenar en el servidor e invocarlo desde esas aplicaciones. Esto reduce la duplicidad del esfuerzo y mejora la modularidad del software.
- La ejecución de un programa en el servidor puede reducir el costo derivado de la transferencia y la comunicación de datos entre el cliente y el servidor en ciertas situaciones.
- Estos procedimientos pueden mejorar la potencia de modelado de las vistas al permitir que los usuarios de bases de datos cuenten con tipos más complejos de datos derivados. Además, se pueden utilizar esos tipos para comprobar restricciones más complejas que quedan fuera de la especificación de aserciones y *triggers*.

En general, muchos DBMS's comerciales permiten escribir procedimientos y funciones almacenados en un lenguaje de programación de propósito general. De forma alternativa, un procedimiento almacenado puede estar compuesto por comandos SQL sencillos, como recuperaciones y actualizaciones. La forma general para declarar un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE <nombre del procedimiento> «parámetros»
<declaraciones locales>
<cuerpo del procedimiento>;
```

Los parámetros y las declaraciones locales son opcionales, y sólo se especifican si se necesitan. Para declarar una función, se necesita un tipo de devolución:

```
CREATE FUNCTION <nombre de la función> «parámetros»
RETURNS <tipo de devolución>
<declaraciones locales>
<cuerpo de la función>;
```

Si el procedimiento (o función) se escribe en un lenguaje de programación de propósito general, es típico especificar el lenguaje, así como el nombre del fichero donde se almacenará el código del programa. Por ejemplo, se puede utilizar el siguiente formato:

```
CREATE PROCEDURE <nombre del procedimiento> «parámetros»
LANGUAGE <nombre del lenguaje de programación>
EXTERNAL NAME <nombre de ruta del fichero>;
```

En general, cada parámetro debe tener un tipo de parámetro, uno de los tipos de datos de SQL. También debe tener un modo de parámetro, que es IN, OUT, o INOUT. Estos modos se corresponden con los parámetros cuyos valores son de sólo entrada, sólo salida, o de entrada y salida.

## Vistas

Una vista en terminología SQL es una tabla que deriva de otras tablas. Esas otras tablas pueden ser tablas base (tablas propias de la base de datos, almacenadas físicamente en la base de datos) o vistas definidas anteriormente. Una vista no existe necesariamente en formato físico; está considerada como una tabla virtual, en oposición a las tablas base, cuyas tuplas están realmente almacenadas en la base de datos. Esto limita las posibles operaciones de actualización que pueden aplicarse a las vistas, pero no ofrecen limitación alguna al consultar una vista.

Podemos pensar que una vista es una forma de especificar una tabla a la que nos referimos con frecuencia, aunque no exista físicamente.

## Triggers o Disparadores

En el curso de la ejecución de una aplicación de base de datos, pueden surgir ocasiones donde si se produce alguna acción específica, se desea que suceda luego otra acción, o tal vez que suceda una sucesión de acciones. Es decir, que a partir de una acción se desencadene la ejecución de las siguientes acciones. SQL proporciona el mecanismo para proporcionar esta capacidad, conocido como TRIGGER o DISPARADOR que es una acción o evento que genera que otro evento ocurra.

Una instrucción SQL desencadenadora causa otra instrucción SQL que será ejecutada. El disparo de un TRIGGER es útil en diversas situaciones. Un ejemplo es realizar una función de registro de LOGS. Ciertas acciones que son críticas para la integridad de una base de datos, como insertar, editar o eliminar una fila de la tabla, podría desencadenar la realización de una entrada en un registro de LOGS que sirven para documentar todas las acciones que suceden en el sistema.

De esta manera, por medio de diversas entradas en el registro de LOGS, pueden grabar no sólo las acciones ejecutadas en el sistema, sino también cuando fue ejecutada y por quién. Los TRIGGERS también pueden utilizarse para mantener una base de datos consistente. Por ejemplo, en una aplicación de gestión de pedidos, un pedido para un producto específico puede desencadenar una instrucción que cambia el estado del producto en la tabla de stock y pasar de estar disponibles a reservados. Del mismo modo, la eliminación de una fila en la tabla pedidos puede desencadenar una instrucción que cambia el estado del producto de reservado a disponible.

## Lenguaje de Consulta SQL

---

El nombre SQL significa Lenguaje de consulta estructurado (Structured Query Language).

SQL utiliza los términos tabla, fila y columna para los términos relación, tupla y atributo del modelo relacional formal, respectivamente. Utilizaremos todos estos términos indistintamente. El principal comando de SQL para definir datos es la sentencia CREATE, que se utiliza para crear esquemas, tablas (relaciones) y dominios (así como otras estructuras, como vistas, aserciones y triggers).

SQL es un lenguaje de bases de datos global: cuenta con sentencias para definir datos, consultas y actualizaciones.

Por tanto, se comporta como DDL y como DML. Además, dispone de características para definir vistas en la base de datos, especificar temas de seguridad y autorización, definir restricciones de integridad, y especificar controles de transacciones.

En esta sección nos detendremos a explicar el lenguaje de consulta denominado SQL. Definiremos la sintaxis referida al:

- ✓ Lenguaje de definición de datos: DDL (Data Definition Language).
- ✓ Lenguaje de manipulación de datos: DML (Data Manipulation Language).

### COMANDOS

Existen dos tipos de comandos SQL:

- ✓ **DDL:** permiten crear y definir nuevas bases de datos, campos e índices.
  - CREATE: Crea nuevas tablas, campos e índices.
  - DROP: Elimina tablas e índices.
  - ALTER: Modifica las tablas agregando campos o cambiando la definición de los campos.
- ✓ **DML:** permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.
  - SELECT: Consulta registros de la base de datos que satisfagan un criterio determinado.
  - INSERT: Carga lotes de datos en la base de datos en una única operación.
  - UPDATE: Modifica los valores de los campos y registros especificados.
  - DELETE: Elimina registros de una tabla de una base de datos.

A continuación, se procede a especificar las distintas sentencias que provee la base de datos MySQL para trabajar con una determinada base de datos.

En primer lugar, debemos aclarar la sintaxis utilizada para mostrar las sentencias:

{Alternativas}, entre llaves se colocarán las palabras que tienen opciones o alternativas en la sentencia a la que pertenecen.

[Opcional], entre corchetes se colocarán las palabras que son opcionales en la sentencia, es decir que pueden colocarse o pueden obviarse.

Comenzaremos con las sentencias del lenguaje referidas a la Definición de Datos y luego las sentencias necesarias para la Manipulación de Datos.

## Definición de Datos

### 1. CREATE DATABASE

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] nombre_base_datos
```

- Esta sentencia sirve para crear una base de datos con un nombre específico.
- Para poder crear una base de datos, el usuario que la crea debe tener privilegios de creación asignados.
- IF NOT EXISTS significa: SI NO EXISTE, por lo tanto, esto es útil para validar que la base de datos sea creada en caso de que no exista, si la base de datos existe y se ejecuta esta sentencia, se genera error.
- CREATE SCHEMA o CREATE DATABASE son sinónimos.
- Ejemplo:

```
CREATE DATABASE IF NOT EXISTS complejo_de_cine
```

### 2. CREATE TABLE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] nombre_de_tabla
nombre_de_columna tipo_de_dato [NOT NULL | NULL] [DEFAULT
valor_por_defecto][AUTO_INCREMENT] [UNIQUE | [PRIMARY] KEY
[CONSTRAINT [nombre_relación] FOREIGN KEY (nombre_columna) REFERENCES
nombre_de_tabla (nombre_columna)]
[ON DELETE opciones_de_referencia]
[ON UPDATE opciones_de_referencia]
```

Columna-  
definición

#### *tipo\_de\_dato:*

```
BIT[ (longitud) ]**
| TINYINT[ (longitud) ]
| SMALLINT[ (longitud) ]
| MEDIUMINT[ (longitud) ]
| INT[ (longitud) ]**
| INTEGER[ (longitud) ]
| BIGINT[ (longitud) ]**
| REAL[ (longitud,decimales) ]
| DOUBLE[ (longitud,decimales) ]**
| FLOAT[ (longitud,decimales) ]
| DECIMAL[ (longitud[ ,decimales] ) ]
| NUMERIC[ (longitud[ ,decimales] ) ]
| DATE **
| TIME **
| TIMESTAMP
| DATETIME **
| YEAR
| CHAR[ (longitud) ] [BINARY] **
| VARCHAR(longitud) [BINARY] **
| BINARY[ (longitud) ]
| VARBINARY(longitud)
```



```

TINYBLOB
BLOB
MEDIUMBLOB
LONGBLOB
TINYTEXT [BINARY]
TEXT [BINARY]
MEDIUMTEXT [BINARY]
LONGTEXT [BINARY]
ENUM(valor1, valor2, valor3,. . .)
SET(valor1, valor2, valor3,. . .)
JSON

```

**\*\*** Los tipos de datos con la anotación (\*\*) son los más utilizados.

#### *opciones\_de\_referencia:*

```
RESTRICT | CASCADE | SET NULL | NO ACTION
```

- Esta sentencia comienza creando la tabla, con la opción IF NOT EXISTS, de la misma manera que CREATE DATABASE, luego comienza a declarar cada una de las columnas de dicha tabla con sus tipos de datos.
- Las opciones NOT NULL | NULL, sirven para especificar si dicha columna puede aceptar valores nulos: NULL, o si no puede guardar valores nulos: NOT NULL.
- También se puede de manera opcional, indicar un valor por defecto para una columna. En dicho caso, si se inserta una fila en la tabla, sin asignarle un valor a una columna que tiene definido DEFAULT, se insertará la fila y la columna tendrá el valor por defecto definido.
- AUTO\_INCREMENT: se refiere a un valor AUTO INCREMENTAL; sirve para aquellas columnas con valores que numéricos enteros donde se necesita que dicho valor se incremente en uno por cada fila insertada en la tabla. Se utiliza muy frecuentemente en las claves primarias.
- UNIQUE: sirve para indicar que una columna en la tabla no puede tener valores repetidos, debe ser ÚNICA. No pueden existir dos filas en la tabla que tengan el mismo valor para un atributo definido como UNIQUE.
- PRIMARY KEY: sirve para especificar que una columna es clave primaria de una tabla. Si una columna es clave primaria implica que sus valores no deben repetirse (UNIQUE): podría ser una columna cuit de un Empleado, o sino una columna como id\_pelicula, que identifica las películas unívocamente, pero no es un atributo propio de la entidad película; en dicho caso se debería utilizar también la palabra AUTO\_INCREMENT para que el atributo crezca automáticamente.
- Como hemos visto, las tablas se relacionan entre ellas por medio de claves foráneas de la siguiente manera: se define una CONSTRAINT que significa RESTRICCIÓN y se le asigna un nombre para identificarla; dicho nombre funciona como clave que identifica unívocamente a cada CONSTRAINT que exista en la base de datos, por lo que toda CONSTRAINT debe tener un nombre no repetido; luego se indica con la palabra FOREIGN KEY, cuál es la columna de la tabla que funciona como clave foránea, indicando a continuación a cuál tabla y a cuál columna de dicha tabla hace referencia la clave foránea con la palabra REFERENCES. Cabe aclarar que la declaración de una CONSTRAINT con un nombre para indicar una relación entre tablas es opcional, ya que se puede establecer

una referencia con la palabra REFERENCES a continuación de la declaración de una columna y así indicar la relación sin asignarle un nombre, pero es buena práctica utilizar CONSTRAINTS, ya que el nombre de una restricción se utiliza para identificar una restricción particular en caso de que la restricción tenga que eliminarse más tarde y/o sustituirse por otra restricción.

- Con respecto a las opciones de referencia:

Estas opciones sirven para cuando se elimina o se actualiza una fila que afecta una relación entre tablas, es decir si por ejemplo tenemos dos tablas: Película y PaisDeOrigen, como ya hemos visto al principio del apunte, podemos tener que una película tiene una referencia a un país de origen de id\_pais\_de\_origen igual a 1.

En este caso existe una relación entre dichas tablas, para entender los conceptos a continuación vamos a establecer que la tabla que tiene la foreign key, es decir la tabla Película es la tabla secundaria y la tabla que es referenciada, es decir PaisDeOrigen es la tabla primaria.

Las opciones de referencia sirven para establecer que se hará en casos de que se elimine o se actualice una fila de la tabla primaria que está siendo referenciada por una fila de la tabla secundaria. Por ejemplo, si se elimina o si se actualiza el país de origen con id=1: "Argentina", en dicho caso que pasa con la película "Relatos Salvajes" que está referenciando a dicho país de origen.

- CASCADE: Eliminar o actualizar la fila de la tabla primaria, y automáticamente eliminar o actualizar las filas coincidentes en la tabla secundaria.
- SET NULL: Eliminar o actualizar la fila de la tabla primaria, y establecer la columna de clave externa (Foreign key) de la tabla secundaria a NULL. Si se especifica una SET NULL, hay que asegurarse que no se haya declarado la columna de la tabla secundaria como NOT NULL.
- RESTRICT: Rechaza la operación de eliminación o actualización en la tabla primaria.
- NO ACTION: Una palabra clave de SQL estándar. En MySQL, equivalente a RESTRICT.
- SET DEFAULT: Esta acción es reconocido por MySQL, pero no todos los motores de base de datos lo tienen, en este caso se establece que para una operación de eliminación o actualización en la tabla primaria se establecerá un valor por defecto para la tabla secundaria.

Para una ON DELETE / ON UPDATE que no se especifica ninguna opción de referencia, la acción por defecto es siempre RESTRICT.

- Ejemplo: Considerando las siguientes tablas, se mostrará la sentencia de creación de la tabla Película:

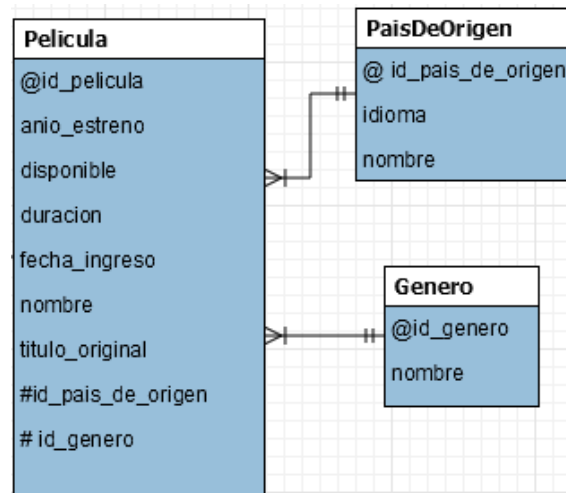


Fig.28: Vista parcial del Modelo de Entidad Relación del Complejo de Cines

```
CREATE TABLE IF NOT EXISTS `Pelicula` (
  `@id_pelicula` INT NOT NULL,
  `anio_estreno` YEAR NULL,
  `disponible` TINYINT(1) NOT NULL,
  `duracion` INT NOT NULL,
  `fecha_ingreso` DATE NULL,
  `nombre` VARCHAR(100) NOT NULL,
  `titulo_original` VARCHAR(100) NULL,
  `#id_pais_de_origen` INT NOT NULL,
  `#id_genero` INT NOT NULL,
  PRIMARY KEY (`@id_pelicula`),
  CONSTRAINT `Pais de Origen de Pelicula` FOREIGN KEY
  REFERENCES `PaisDeOrigen`
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
  CONSTRAINT `Genero de Pelicula` FOREIGN KEY
  REFERENCES `Genero`
  ON DELETE NO ACTION
  ON UPDATE NO ACTION)
```

### 3. ALTER TABLE

```
ALTER [IGNORE] TABLE nombre_tabla
  ADD [COLUMN] nombre_columna columna_definicion
    [FIRST | AFTER col_nombre]
  | ADD [COLUMN] (nombre_columna definicion_columna)
  | ADD {INDEX|KEY} [nombre_indice]
    [tipo_indice] (nombre_columna_indice) [indice_opcion] ...
  | ADD [CONSTRAINT] PRIMARY KEY
    [indice_tipo] (indice_col_nombre,...) [indice_opcion] ...
  | ADD [CONSTRAINT [symbol]]
    UNIQUE [INDEX|KEY] [indice_nombre]
    [indice_tipo] (indice_col_nombre,...) [indice_opcion] ...
  | ADD FULLTEXT [INDEX|KEY] [indice_nombre]
    (indice_col_nombre,...) [indice_opcion] ...
  | ADD SPATIAL [INDEX|KEY] [indice_nombre]
    (indice_col_nombre,...) [indice_option] ...
  | ADD [CONSTRAINT [nombre_relación] FOREIGN KEY (nombre_columna)
    REFERENCES nombre_de_tabla (nombre_columna)]
    [ON DELETE opciones_de_referencia]
    [ON UPDATE opciones_de_referencia]
  | ALTER [COLUMN] col_nombre {SET DEFAULT literal | DROP DEFAULT}
  | CHANGE [COLUMN] col_nombre_anterior col_nombre_nuevo
    columna_definicion
    [FIRST|AFTER col_nombre]
  | MODIFY [COLUMN] col_nombre columna_definicion
    [FIRST | AFTER col_nombre]
  | DROP [COLUMN] col_nombre
  | DROP PRIMARY KEY
  | DROP {INDEX|KEY} index_nombre
  | DROP FOREIGN KEY nombre_relación
  | DISABLE KEYS
  | ENABLE KEYS
  | RENAME [TO|AS] nombre_nuevo_tabla
  | RENAME {INDEX|KEY} indice_nombre_anterior TO indice_nombre_nuevo
  | ORDER BY col_nombre [, col_nombre] ...
```

- Para cambiar el nombre de una tabla de t1 a t2:

```
ALTER TABLE t1 RENAME t2;
```

- Para cambiar los tipos de datos de sus atributos, suponiendo que tenemos dos atributos **a** es INTEGER y **b** es CHAR con longitud 10: CHAR (10), cambiaremos **a** como TINYINT sin aceptar nulos y cambiaremos **b** por el nombre **c** y con longitud 20:

```
ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

- Para agregar una nueva columna **d**, de tipo DATETIME:

```
ALTER TABLE t2 ADD d DATETIME;
```

- Para agregar un nuevo índice en una columna **d**:

```
ALTER TABLE t2 ADD INDEX (d)
```

- Para eliminar la columna **c**:

```
ALTER TABLE t2 DROP COLUMN c;
```

- Para agregar una nueva columna auto incremental y clave primaria, de tipo entera llamada **e**:

```
ALTER TABLE t2 ADD e INT NOT NULL AUTO_INCREMENT,  
ADD PRIMARY KEY (e);
```

#### 4. DROP TABLE

```
DROP [TEMPORARY] TABLE [IF EXISTS]  
  
    nombre_tabla [, nombre_tabla] ...
```

- DROP TABLE remueve una o más tablas. Para poder eliminar tablas, el usuario que ejecuta la sentencia debe tener privilegios de DROP para cada tabla que quiera eliminar.
- Con este comando se elimina todos los datos de la tabla, tanto la definición como las filas de datos de la misma. Si no existe la tabla que se desea borrar MySQL devuelve error, por eso es útil utilizar la opción IF EXISTS, que significa si existe, en caso de que exista la tabla la elimina.

#### 5. CREATE INDEX

```
CREATE INDEX nombre_indice  
ON nombre_tabla (col_name [(length)] [ASC | DESC])  
[opcion_algoritmo | opcion_bloqueo] ...
```

## Manipulación de Datos

### SELECT

La sentencia que se muestra a continuación sirve para recuperar datos de la base de datos. Por medio de esta sentencia SELECT, obtengo como resultado el conjunto de filas, (con todos sus columnas o un subconjunto de ellas) de una tabla o más tablas. La forma básica de la sentencia SELECT es la siguiente:

```
SELECT <lista de atributos>
FROM <lista de tablas>
WHERE <condición>;
```

<lista de atributos> es una lista de los atributos cuyos valores serán recuperados por la consulta.

<lista de tablas> es una lista de las tablas necesarias para procesar la consulta.

<condición> es una expresión condicional (booleana) que se evalúa en las filas de la tabla, aquellas filas que cumplan con la condición booleana serán el resultado de la consulta.

Ejemplos de Selección simples:

- Selección sin condición:

```
SELECT nombre_columna,nombre_columna
FROM nombre_tabla;
```

- Selección de todas las columnas de una tabla, se simplifica colocando \* en lugar de colocar todos los nombres de las columnas:

```
SELECT * FROM nombre_tabla;
```

- Selección de determinadas columnas sin valores repetidos. La palabra reservada DISTINCT (distinto) sirve como filtro: selecciona todas las filas de la tabla y devuelve el valor de las columnas indicadas con una previa eliminación de aquellas filas que tuvieran el mismo valor en la o las columnas especificadas en la cláusula SELECT:

```
SELECT DISTINCT nombre_columna, nombre_columna
FROM nombre_tabla;
```

- Selección con utilización de operadores en la condición:

```
SELECT nombre_columna, nombre_columna
FROM nombre_tabla
WHERE nombre_columna operador valor;
```

```
SELECT nro, fecha, monto, cliente_nombre
FROM facturas
WHERE monto >200;
```

A continuación, se muestra una tabla con los posibles operadores :

Operador	Descripción
=	Igual
<>	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
BETWEEN	Entre - para comparar valores en un rango determinado.
LIKE	Parecido – se usa para buscar patrones.
IN	Para especificar múltiples posibles valores para una columna

- Selección con AND y OR en la condición:

```
SELECT * FROM Clientes
WHERE pais='Argentina'
AND provincia='Buenos Aires';
```

```
SELECT * FROM Customers
WHERE provincia='Córdoba'
OR provincia='Buenos Aires';
```

- Como en muchas aplicaciones de bases de datos se necesitan el agrupamiento y la agregación, SQL dispone de funciones que incorporan estos conceptos: **COUNT**, **SUM**, **MAX**, **MIN** y **AVG**.
  - La función COUNT devuelve el número de filas o valores especificados en una consulta.
  - Las funciones SUM, MAX, MIN y AVG se aplican a un conjunto o multiconjunto de valores numéricos y devuelven, respectivamente, la suma, el valor máximo, el valor mínimo y el promedio de esos valores.

- Estas funciones se pueden utilizar en la cláusula SELECT o en una cláusula HAVING (de la que hablaremos más tarde).
- Las funciones MAX y MIN también se pueden utilizar con atributos, como veremos en ejemplos posteriores.
- Por ejemplo, si quisiéramos visualizar la suma de los salarios de todos los empleados, el salario más alto, el salario más bajo y el sueldo medio.

```
SELECT SUM (sueldo), MAX (sueldo), MIN (sueldo), AVG  
(sueldo)  
FROM empleados
```

La selección de datos puede ser mucho más compleja como se muestra a continuación:

```
SELECT [ALL | DISTINCT | DISTINCTROW] lista_atributos  
FROM nombre_tabla  
[WHERE condición]  
[GROUP BY {nombre_columna | expresión}  
[ASC | DESC], ... [WITH ROLLUP]]  
[HAVING where_condicion]  
[ORDER BY {nombre_columna | expr | position}  
[ASC | DESC], ...]  
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

- En muchos casos queremos aplicar funciones a subgrupos de filas de una tabla, estando los subgrupos divididos en base a algunos valores de uno o más atributos. Por ejemplo, vamos a suponer que queremos saber el número de empleados que trabajan en cada proyecto. En estos casos, tenemos que dividir las filas de la tabla en subconjuntos no solapados de filas. Cada grupo (partición) estará compuesto por las filas que tienen el mismo valor para algún(os) atributo(s), denominado(s) atributo(s) de agrupamiento. Después podemos aplicar la función independientemente a cada grupo. La cláusula GROUP BY especifica los atributos de agrupamiento, que también deben aparecer en la cláusula SELECT, por ejemplo:  
Si quisiéramos saber, para todos proyectos de una empresa, el nro. de proyecto, el nombre del proyecto y el número de empleados que trabajan en cada proyecto.

```
SELECT nro_proyecto, nombre_proyecto, COUNT (*)  
FROM proyectos  
GROUP BY nro_proyecto, nombre_proyecto;
```

- SQL proporciona una cláusula HAVING, que puede aparecer en combinación con una cláusula GROUP BY, con este propósito. HAVING proporciona una condición en el grupo de filas asociado a cada valor de los atributos de agrupamiento. En el resultado de la consulta sólo aparecen los grupos que satisfacen la condición. HAVING siempre debe ir acompañado previamente de un GROUP BY, ya que HAVING sirve para expresar una condición que es evaluada por cada grupo; aquellos grupos que la cumplan aparecerán en el resultado de la consulta. Si quisiéramos saber, para todos proyectos de una empresa, el nro. de proyecto, el nombre del proyecto y el número de empleados que trabajan en cada proyecto y mostrar sólo aquellos proyectos que tengan más de 3 empleados trabajando:



```
SELECT nro_proyecto, nombre_proyecto, COUNT (*)
FROM proyectos
GROUP BY nro_proyecto, nombre_proyecto
HAVING COUNT (*) > 2;
```

- SQL permite ordenar las filas del resultado de una consulta por los valores de uno o más atributos, utilizando la cláusula ORDER BY. El orden predeterminado es el ascendente. Con la palabra clave DESC podemos ver el resultado ordenado descendientemente. La palabra clave ASC permite especificar explícitamente el orden ascendente.
- LIMIT: se puede utilizar para restringir el número de filas que puede retornar una sentencia SELECT. LIMIT, toma uno o dos argumentos numéricos, ambos deben ser no negativos y enteros. Con dos argumentos, el primer argumento especifica el desplazamiento de la primera fila a devolver, y el segundo especifica el número máximo de filas a devolver. El desplazamiento de la fila inicial es 0 (no 1):

Ejemplo:

```
SELECT * FROM nombre_tabla LIMIT 5,10;
Devuelve desde la fila 6 hasta la 15.
```

Con un argumento, el valor especifica el número de filas a devolver desde el principio del resultado:

Ejemplos:

```
SELECT * FROM nombre_tabla LIMIT 5;
Devuelve las primeras 5 filas.
```

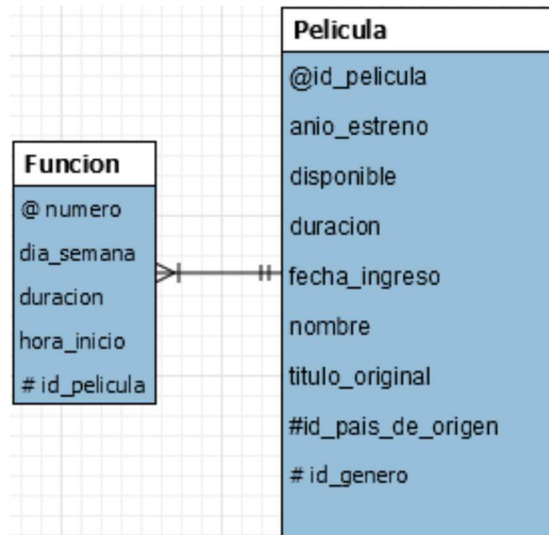
Para recuperar todas las filas de un cierto desplazamiento hasta el final del resultado, puede utilizar un gran número para el segundo parámetro.

Consulta multi-tablas.

En SQL existe la posibilidad de realizar consultas de datos de más de una tabla al mismo tiempo. La palabra utilizada para unir tablas es JOIN. Esta operación permite combinar filas de datos de dos o más tablas basándose en columnas comunes entre ellas.

Por ejemplo:

Si pensamos en las tablas Película y Función del Complejo de Cines:



Si quisiéramos recuperar los nombres de las películas con los datos de qué días de las semanas y a qué hora comienzan las funciones de cada película necesitamos consultar datos de más de una tabla a la vez. Para eso la sentencia debiera ser:

```
SELECT Película.nombre, Función.dia_semana, Función.hora_inicio
FROM Película INNER JOIN Funcion ON Funcion.id_pelicula =
Película.id_pelicula
```

Si las tablas Película y Función tienen estos datos:

ENTIDAD FUNCIÓN				
@numero	dia_semana	duración	Hora_inicio	#id_pelicula
1	3	150	22:30	2
2	4	150	18:15	2
3	1	160	17:40	3
4	4	160	19:10	3

ENTIDAD PELICULA				
@id_pelicula	Nombre	duración	#id_pais_de_origen	#id_genero
2	Relatos Salvajes	122	1	1
3	La era del Hielo	143	1	3

\*Para simplificar, se muestra un conjunto reducido de atributos de película.

De esta manera, los datos arrojados por la consulta anterior serían:

RESULTADOS		
nombre	dia_semana	hora_inicio
Relatos Salvajes	3	22:30
Relatos Salvajes	4	18:15

La era del Hielo	1	17:40
La era del Hielo	4	19:10

A continuación, se muestra la sintaxis de forma genérica a utilizar para recuperar datos de más de una tabla a la vez:

```
SELECT [ALL | DISTINCT | DISTINCT ROW] lista_atributos
FROM nombre_tabla_1 [INNER | CROSS] JOIN nombre_tabla_2 ON condicion

SELECT [ALL | DISTINCT | DISTINCT] lista_atributos
FROM nombre_tabla_1 {LEFT|RIGHT} JOIN nombre_tabla_2 ON condicion
```

*condicion: nombre\_tabla\_1.atributo = nombre\_tabla\_2.atributo*

Se debe aclarar que en el caso de las consultas multi-tablas como en el caso de Función y Película es útil utilizar abreviaciones de nombres de tablas para llamarlas de forma más simple y breve. También puede darse apodos a los nombres de columnas. Por ejemplo, la consulta de ejemplo anterior se puede escribir también de la siguiente manera:

```
SELECT pel.nombre, fun.hora_inicio, fun.dia_semana AS dia,
FROM Pelicula pel INNER JOIN Funcion fun
ON fun.id_pelicula = pel.id_pelicula
```

Abreviación del nombre de la tabla.

Renombrado de columna: en vez de dia\_semana, en los resultados saldrá dicha columna se llamará dia.

- Además de INNER JOIN que devuelve todas las filas que cumplen con la condición de igualdad de atributos expresada en la cláusula ON, existen además dos casos particulares que provee MySQL:
  - LEFT JOIN: Devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha. Si existe alguna fila de la tabla izquierda que no tiene coincidencia con ninguna fila de la tabla derecha, dicha fila es mostrada en el resultado, pero aparecerá el valor NULL en las columnas de la tabla derecha ya que no existe coincidencia.
  - RIGHT JOIN: Devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si existe alguna fila de la tabla derecha que no tiene coincidencia con ninguna fila de la tabla izquierda, dicha fila es mostrada en el resultado, pero aparecerá el valor NULL en las columnas de la tabla izquierda ya que no existe coincidencia.

- Por Ejemplo:  
Si las tablas Película y Función tienen estos datos:

ENTIDAD FUNCIÓN				
@numero	dia_semana	duración	Hora_inico	#id_película
1	3	150	22:30	2
2	4	150	18:15	2

ENTIDAD PELICULA				
@id_película	Nombre	duración	#id_pais_de_origen	#id_genero
2	Relatos Salvajes	122	1	1
3	La era del Hielo	143	1	3

\*Para simplificar, se muestra un conjunto reducido de atributos de película.

Si quisiéramos listar todos los nombres de películas y los días de semanas de sus funciones si es que tiene funciones asignadas.

Para asegurarnos de listar todas las películas ya sea que tengan o no tengan función asignada, podríamos utilizar LEFT o RIGHT JOIN de la siguiente manera:

```
SELECT pel.nombre, f.dia_semana
FROM Pelicula pel LEFT JOIN Funcion f
ON pel.id_película=f.id_película
```

```
SELECT p.nombre, f.dia_semana
FROM Funcion f RIGHT JOIN Pelicula p
ON f.id_película=p.id_película
```

De esta manera, los datos arrojados por cualquiera de las consultas anteriores serían:

RESULTADOS		
nombre	dia_semana	hora_inicio
Relatos Salvajes	3	22:30
Relatos Salvajes	4	18:15
La era del Hielo	Null	Null

- En MySQL, JOIN, CROSS JOIN, y INNER JOIN son equivalentes, pueden ser usadas indiferentemente.

## Subconsultas

Una subconsulta es una consulta anidada en una instrucción SELECT, INSERT, UPDATE o DELETE, o bien en otra subconsulta. Las subconsultas se pueden utilizar en cualquier parte en la que se permita una expresión. Se llama también subconsulta a una consulta o selección interna, mientras que la instrucción que contiene una subconsulta también es conocida como consulta o selección externa.

Una subconsulta anidada en la instrucción externa SELECT tiene los componentes siguientes:

- Una consulta SELECT normal, que incluye los componentes normales de la lista de selección.
- Una cláusula normal FROM que incluye uno o varios nombres de tablas o vistas.
- Una cláusula opcional WHERE.
- Una cláusula opcional GROUP BY.
- Una cláusula opcional HAVING.

Ejemplo:

```
SELECT * FROM clientes WHERE id IN (
  SELECT cliente_id FROM facturas WHERE fecha BETWEEN 2016-10-10 AND 2017-01-01
);
```

La consulta SELECT de una subconsulta se presenta siempre entre paréntesis. Una subconsulta puede anidarse en la cláusula WHERE o HAVING de una instrucción externa SELECT, INSERT, UPDATE o DELETE, o bien en otra subconsulta. Una subconsulta puede aparecer en cualquier parte en la que se pueda usar una expresión, si devuelve un solo valor.

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING. En una subconsulta, se utiliza una instrucción SELECT para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula WHERE o HAVING.

## Operadores para Subconsultas

Se puede utilizar el predicado ANY o SOME, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta.

```
SELECT * FROM tabla WHERE columna ANY ( [ SUBCONSULTA ] )
```

El predicado ALL se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta.

```
SELECT * FROM tabla WHERE columna ALL ( [ SUBCONSULTA ] )
```

El predicado IN se emplea para recuperar únicamente aquellos registros de la consulta principal para los que algunos registros de la subconsulta contienen un valor igual.

```
SELECT * FROM tabla WHERE columna IN ( [ SUBCONSULTA ] )
```

Inversamente se puede utilizar NOT IN para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual.

El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro.

```
SELECT * FROM tabla WHERE EXISTS ( [ SUBCONSULTA ] )
```

## UPDATE

El comando UPDATE se utiliza para modificar los valores de atributo de una o más filas seleccionadas. Una cláusula WHERE en el comando UPDATE selecciona las filas de una tabla que se van a modificar. No obstante, la actualización del valor de una clave principal puede propagarse a los valores de la foreign key de las filas de otras relaciones en caso de haberse especificado una opción de acción referencial en las restricciones de integridad referencial del DDL. Una cláusula SET adicional en el comando UPDATE especifica los atributos que se modificarán y sus nuevos valores.

- La sentencia UPDATE tiene la siguiente forma:

```
UPDATE nombre_tabla
SET col_nombre_1={valor1|DEFAULT} [ , col_nombre_2={valor2|DEFAULT} ]
[ WHERE condicion ]
[ ORDER BY ... ]
[ LIMIT cantidad_filas ]
```

- Ejemplo: Si quisiéramos editar una determinada película para cambiar su duración y definir que la misma está disponible para ser programada:

```
UPDATE Pelicula
SET disponible=1, duracion=160
WHERE id_pelicula=4
```

## INSERT

Se utiliza para agregar nuevas filas de datos a una tabla determinada.

- En su formato más sencillo, INSERT se utiliza para añadir una sola fila a una tabla. Debemos especificar el nombre de la tabla y una lista de valores para la fila. Los valores deben suministrarse en el mismo orden en el que se especificaron los atributos correspondientes en el comando CREATE TABLE. Por ejemplo, para añadir una fila nueva a la tabla película:

```
INSERT INTO pelicula
VALUES (2, 2014, true, 122, 01/08/2014, "Relatos Salvajes", "Relatos Salvajes",1)
```

- Una segunda forma de la sentencia INSERT permite especificar explícitamente los nombres de los atributos que se corresponden con los valores suministrados en el

comando INSERT. Esto resulta útil si la relación tiene muchos atributos y sólo vamos a asignar valores a unos cuantos en la fila nueva. Sin embargo, los valores deben incluir todos los atributos con la especificación NOT NULL y ningún valor predeterminado. Los atributos que permiten los valores NULL o DEFAULT son los que se pueden omitir. Por ejemplo, para introducir una fila para un nuevo EMPLEADO del que únicamente conocemos los atributos nombre, apellido, y dni, podemos utilizar:

```
INSERT INTO EMPLEADO (nombre, apellido, dni)
VALUES ('Ricardo', 'Roca', '653298653');
```

- De forma genérica podemos decir que la sintaxis de esta sentencia es:

```
INSERT INTO nombre_tabla (columna1, columna2, columna3,...)
VALUES (valor, valor2, valor3,...);
```

## DELETE

El comando DELETE elimina filas de una tabla. Incluye una cláusula WHERE, para seleccionar las filas que se van a eliminar.

- Las filas se eliminan explícitamente sólo de una tabla a la vez. Sin embargo, la eliminación se puede propagar a filas de otras tablas si se han especificado opciones de acciones referenciales en las restricciones de integridad referencial del DDL.
- En función del número de filas seleccionadas por la condición de la cláusula WHERE, ninguna, una o varias filas pueden ser eliminadas por un solo comando DELETE.
- La ausencia de una cláusula WHERE significa que se borrarán todas las filas de la relación; sin embargo, la tabla permanece en la base de datos, pero vacía. Debemos utilizar el comando DROP TABLE para eliminar la definición de la tabla.
- La forma de la sentencia DELETE es la siguiente:

```
DELETE FROM nombre_tabla
[WHERE condicion]
[ORDER BY ...]
[LIMIT cantidad_filas]
```

- Ejemplo: Si quisiéramos eliminar todas aquellas películas estrenadas en años anteriores a 1990 podríamos ejecutar la siguiente sentencia:

```
DELETE FROM Película
WHERE año_estreno <1990
```

## Seguridad

---

La seguridad en las bases de datos es un tema muy amplio que comprende muchos conceptos, entre los cuales se incluyen los siguientes:

- Aspectos legales y éticos en relación con el derecho de acceso a determinada información. Ciertos tipos de información están considerados como privados y no pueden ser accedidos legalmente por personas no autorizadas.
- Temas de políticas a nivel gubernamental, institucional o de empresa en relación con los tipos de información que no debería estar disponible públicamente (por ejemplo, la concesión de créditos o los informes médicos personales).
- Temas relativos al sistema, como los *niveles del sistema* en los que se deberían reforzar las distintas funciones de seguridad (por ejemplo, si una función de seguridad debería ser manipulada a nivel físico del hardware, a nivel del sistema operativo o a nivel del DBMS).
- La necesidad dentro de algunas organizaciones de identificar diferentes niveles de seguridad y de clasificar según éstos a los datos y a los usuarios: por ejemplo, alto secreto, secreto, confidencial y no clasificado.

Las amenazas a las bases de datos tienen como consecuencia la pérdida o la degradación de todos o de algunos de los siguientes objetivos de seguridad comúnmente aceptados: integridad, disponibilidad y confidencialidad.

**Pérdida de integridad.** La integridad de la base de datos tiene relación con el requisito a cumplir de que la información se encuentre protegida frente a modificaciones inadecuadas. La modificación de datos incluye la creación, inserción, modificación, cambio del estado de los datos y el borrado. La integridad se pierde si se realizan cambios no autorizados en los datos mediante acciones intencionadas o accidentales.

**Pérdida de disponibilidad.** La disponibilidad de la base de datos tiene relación con que los objetos estén disponibles para un usuario humano o para un programa que tenga los derechos correspondientes. Hemos analizado técnicas de alta disponibilidad en la sección anterior relacionada con las copias de seguridad.

**Pérdida de confidencialidad.** La confidencialidad de la base de datos tiene relación con la protección de los datos frente al acceso no autorizado. El acceso no autorizado, no previsto o no intencionado podría tener como resultado la pérdida de la confianza en la organización, el que ésta quede en entredicho o que sea objeto de acciones legales en su contra.

Para proteger las bases de datos contra estos tipos de amenazas, es habitual implementar medidas de control como el control de accesos y el cifrado de cifrado.

Un problema de seguridad habitual en los sistemas de computadores es prevenir que personas no autorizadas tengan acceso al sistema, bien para obtener información o bien para realizar cambios malintencionados en partes de la base de datos. El mecanismo de seguridad de un DBMS debe incluir medidas para restringir el acceso al sistema de base de datos en su totalidad. Esta función se denomina **control de acceso** y se gestiona mediante la creación de cuentas de usuario y contraseñas para controlar el proceso de entrada al DBMS.



El **cifrado de datos** se utiliza para proteger datos confidenciales como los números de las tarjetas de crédito y contraseñas. El cifrado se puede utilizar también para proporcionar protección adicional a partes confidenciales de la base de datos. Los datos se **codifican** utilizando algún algoritmo de codificación o cifrado. Un usuario no autorizado que acceda a datos codificados tendrá dificultades para descifrarlos, pero a los usuarios autorizados se les proporcionarán algoritmos de descodificación o descifrado (claves) para descifrar los datos. Un caso particular de las técnicas de cifrado es el *hashing* de contraseñas, donde el valor guardado en la base de datos corresponde a una cadena de bits que no puede ser descodificada (sólo de ida) y es utilizada para ser comparada con otras cadenas codificadas para validar si es igual o diferente y de esa forma otorgar o no el acceso a determinados recursos.

## Administración de cuentas de usuario en MySQL

MySQL tiene variedad de opciones para otorgar a determinados usuarios permisos entre tablas y bases de datos. Por defecto, el motor de base de datos crea un usuario con permisos para todas las tablas de la base de datos: este superusuario es el *root* y se recomienda limitar su uso a la gestión del DBMS y no usarlo en aplicaciones de producción.

Una cuenta MySQL se define en términos de un nombre de usuario y el equipo o equipos desde los que el usuario puede conectar al servidor. La cuenta también tiene una contraseña. Hay algunas consideraciones a tener en cuenta a la hora de gestionar las cuentas de usuario:

- Los nombres de usuarios en MySQL pueden tener como máximo 16 caracteres de longitud.
- Las contraseñas MySQL no están relacionadas con las contraseñas para ingresar en el sistema operativo. No hay una conexión necesaria entre la contraseña que se usa para iniciar sesión en una máquina Windows o Unix y la contraseña usada para acceder al servidor MySQL en la misma computadora.
- MySQL encripta contraseñas usando su propio algoritmo. Esta encriptación es diferente de la usada durante el proceso de logueo de Unix y es la misma que la implementada en la función `PASSWORD()`.

Para crear nuevos usuarios en MySQL usamos la siguiente sintaxis:

```
CREATE USER 'nombre_usuario'@'localhost' IDENTIFIED BY 'tu_contrasena';
```

Lamentablemente, a este punto el nuevo usuario no tiene permisos para hacer algo con las bases de datos. Por consecuencia si el usuario intenta identificarse (con la contraseña establecida), no será capaz de acceder a la consola de MySQL. Es importante destacar la palabra `'localhost'` luego del nombre de usuario, ya que este valor identifica desde donde podrá realizar conexiones a la base de datos el usuario creado: en este caso sólo desde la misma PC en la que se encuentre instalado MySQL, es decir el host local. Para aplicaciones que se conectan de forma remota a la base de datos es posible especificar la IP de la conexión, de esta forma sólo podrán realizarse conexiones al servidor MySQL desde la dirección especificada. Para permitir conexiones desde cualquier host o de otras computadoras con IP dinámica se utiliza el comodín `%`.

Por ello, lo primero que debemos hacer es proporcionarle el acceso requerido al usuario con la información que requiere.

```
GRANT [permiso] ON [nombre de bases de datos].[nombre de tabla] TO '[nombre de usuario]@'localhost';
```

Dentro de los permisos que podemos otorgar se encuentran los siguientes valores:

- ALL PRIVILEGES: esta opción otorga todos los permisos listados debajo.
- CREATE: permite crear nuevas tablas o bases de datos.
- DROP: permite eliminar tablas o bases de datos.
- DELETE: permite eliminar registros de tablas.
- INSERT: permite insertar registros en tablas.
- SELECT: permite leer registros en las tablas.
- UPDATE: permite actualizar registros seleccionados en tablas.
- GRANT OPTION: permite otorgar o remover privilegios a otros usuarios usuarios.

Un ejemplo de otorgamiento de permisos con GRANT OPTION es el siguiente:

```
GRANT SELECT ON cine.* TO 'martin'@'localhost' WITH GRANT OPTION;
```

Una vez que has finalizado con los permisos que deseas configurar para tus nuevos usuarios, hay que asegurarse siempre de refrescar todos los privilegios.

```
FLUSH PRIVILEGES;
```

Para remover un permiso, la estructura es casi idéntica a la que los asigna:

```
REVOKE [permiso] ON [nombre de base de datos].[nombre de tabla] FROM '[nombre de usuario]'@'localhost';
```

Así como se puede borrar bases de datos con DROP, también se puede usar el comando DROP para borrar usuarios:

```
DROP USER 'usuario_prueba'@'localhost';
```

También es posible crear usuarios desde la interfaz gráfica de MySQL Workbench, para más información puedes consultar el instructivo **Gestión de usuarios y permisos con MySQL Workbench**.

## Conexiones seguras con MySQL

**MySQL** incluye soporte para conexiones seguras (cifradas) entre los clientes MySQL y el servidor, utilizando el protocolo SSL (Secure Sockets Layer). La configuración de MySQL tiene la misión de ser tan rápida como sea posible, así que no se usan las conexiones cifradas por defecto. Hacerlo, haría que el protocolo cliente/servidor fuese mucho más lento. Cifrar datos es una operación que requiere un uso intensivo de CPU, y por lo tanto obliga a la máquina a realizar trabajo adicional que retrasa otras tareas de MySQL. Esto significa que cualquiera con acceso a la red podría ver el tráfico y mirar los datos que están siendo enviados o recibidos. Incluso podría cambiar los datos mientras están aún en tránsito entre el cliente y el servidor. Para aplicaciones que requieran la seguridad que proveen las conexiones cifradas, el trabajo de computación extra está justificado.

Cuando necesita mover información sobre una red de una manera segura, una conexión sin cifrar es inaceptable. El cifrado es la manera de hacer que cualquier dato sea ilegible. MySQL

permite que el cifrado sea activado para conexiones individuales. Puede escoger entre una conexión normal sin cifrar, o una segura cifrada mediante SSL dependiendo de los requerimientos de las aplicaciones individuales.

## Modelos de Persistencia

Al desarrollar una aplicación bajo el Modelo de Objetos todas nuestras entidades residen en memoria principal, donde sus relaciones se expresan como punteros que indican la posición de memoria donde encontrar el objeto vinculado.

El problema surge debido a que la memoria principal es limitada y es necesario poder almacenar los objetos en un medio que permita recuperarlos en cualquier momento futuro independientemente de la ejecución del programa (concepto que llamaremos **persistencia**).

Para solucionar este problema es que hacemos uso de las bases de datos relacionales, donde la información se guarda en forma de filas y columnas en tablas. Pero al comparar el modelo de objetos y el relacional surgen ciertas similitudes y diferencias:

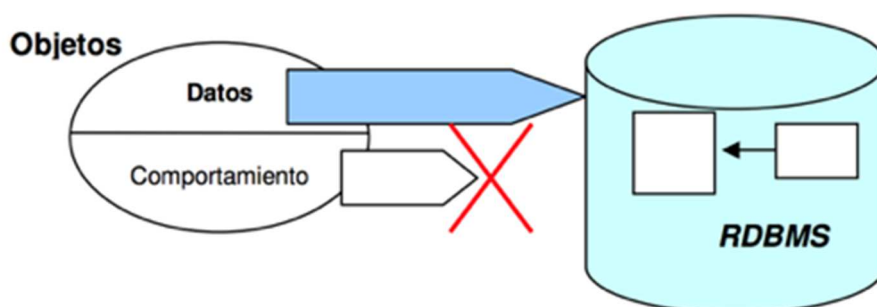


Fig. 29: Imposibilidad de mapear el comportamiento en ambos modelos

- Originalmente podemos establecer una equivalencia entre el concepto de Tabla/Clase, y Fila/Instancias de un objeto de esa clase. Pero más adelante empezaremos a tener ciertas dificultades para que esta asignación sea tan lineal.
- Los objetos tienen comportamiento mientras que las tablas sólo permiten habilitar ciertos controles de integridad (constraints) o pequeñas validaciones antes o después de una actualización (triggers). Como vimos anteriormente, los procedimientos almacenados no están asociados a una tabla lo que no se corresponde con la asignación de comportamiento de los objetos.
- Los objetos **encapsulan** información para favorecer la abstracción del observador. Una tabla no tiene esa habilidad; como vimos anteriormente usando las consultas SELECT podemos obtener uno, varios o todos los valores de las columnas.
- En el modelo de objetos es posible generar **Interfaces** que permiten establecer un contrato entre dos partes: quien publica un determinado servicio y quien lo usa. En el álgebra relacional la interfaz no se convierte en ninguna entidad.
- La **Herencia** es una relación estática que se da entre clases que favorece agrupar comportamiento y atributos en común. Cuando se instancia un objeto recibimos la definición propia de la clase y de todas las superclases de las cuales hereda. En el modelo lógico de Entidad/Relación existen supertipos y subtipos, pero en la implementación física las tablas no tienen el concepto de herencia; es por eso que surge la necesidad de realizar algunas adaptaciones que se describirán en esta sección.

- Al no existir el comportamiento en las tablas y no estar presente el concepto de interfaz no es posible aplicar el concepto de **Polimorfismo** en el álgebra relacional.

Las incompatibilidades entre ambos modelos se denominan **Impedance Mismatch**, lo que se traduce al español como "problema de impedancia", término que no es muy aclarador.

En realidad, el término proviene del campo de la electrónica, y sirve para explicar el fenómeno por el cual tenemos, por ejemplo, un parlante conectado a un amplificador, pero entre la entrada y salida de estos hay diferencias en una cierta propiedad (la impedancia), provocando que el audio no se escuche como se debe. Por extensión, se usa el término en otros campos, siempre hablando de conectar dos elementos que no se adaptan bien.



## ORM

Para acceder de forma efectiva a la base de datos relacional desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se llama ORM (object-relational mapping) o "mapeo de objetos a bases de datos", y está formada por objetos que permiten acceder a los datos y que contienen en sí mismos el código necesario para hacerlo.

La principal ventaja que aporta el ORM es la reutilización, permitiendo llamar a los métodos de un objeto de datos desde varias partes de la aplicación e incluso desde diferentes aplicaciones. Otra consideración importante que hay que tener en cuenta cuando se crean elementos de acceso a los datos: las empresas que crean las bases de datos utilizan variantes diferentes del lenguaje SQL. Si se cambia a otro sistema gestor de bases de datos, es necesario reescribir parte de las consultas SQL que se definieron para el sistema anterior. Si se crean las consultas mediante una sintaxis independiente de la base de datos y un componente externo se encarga de traducirlas al lenguaje SQL concreto de la base de datos, se puede cambiar fácilmente de una base de datos a otra. Este es precisamente el objetivo de las capas de abstracción de bases de datos. Esta capa obliga a utilizar una sintaxis específica para las consultas y a cambio realiza el trabajo de optimizar y adaptar el lenguaje SQL a la base de datos concreta que se está utilizando.

La principal ventaja de la capa de abstracción es la portabilidad, porque hace posible que la aplicación cambie a otra base de datos, incluso en mitad del desarrollo de un proyecto. Si se debe desarrollar rápidamente un prototipo de una aplicación y el cliente no ha decidido todavía la base de datos que mejor se ajusta a sus necesidades, se puede construir la aplicación utilizando una base de datos sencilla orientada a prototipos y cuando el cliente haya tomado la decisión, cambiar fácilmente a un motor más completo como MySQL. Solamente es necesario modificar una línea en un archivo de configuración y todo funciona correctamente.

Muchos paquetes han sido desarrollados para reducir el tedioso proceso de desarrollo de sistemas de mapeo relacional de objetos proveyendo bibliotecas de clases que son capaces de realizar mapeos automáticamente. Dada una lista de tablas en la base de datos, y objetos en el programa, ellos pueden automáticamente mapear solicitudes de un sentido a otro. La consulta

a un objeto persona por sus números telefónicos resultará en la creación y envío de la consulta apropiada a la base de datos, y los resultados son traducidos directamente en objetos de números telefónicos dentro del programa.

Desde el punto de vista de un programador, el sistema debe lucir como un almacén de objetos persistentes. Uno puede crear objetos y trabajar normalmente con ellos; los cambios que sufran terminarán siendo reflejados en la base de datos.

Sin embargo, en la práctica no es tan simple. Todos los sistemas ORM tienden a hacerse visibles en varias formas, reduciendo en cierto grado la capacidad de ignorar la base de datos. Peor aún, la capa de traducción puede ser lenta e ineficiente (comparada en términos de las sentencias SQL que escribe), provocando que el programa sea más lento y utilice más memoria que el código "escrito a mano".

## Hibernate ORM

**Hibernate** es una herramienta de ORM para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones. Busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: que se usa en la memoria de la computadora (orientación a objetos) y el utilizado en las bases de datos (modelo relacional).



Para lograr esto, el desarrollador puede detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Esta información le permite a la aplicación manipular los datos en la base de datos operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL; además genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Para descargar y configurar Hibernate en nuestros proyectos Java consulta el instructivo **Obtener Hibernate ORM utilizando Maven**.

### El archivo de configuración

Para poder indicar a Hibernate cuales son los datos de la conexión a nuestra base de datos, tales como el usuario y la contraseña y demás parámetros de configuración necesitamos el archivo XML hibernate.cfg.xml.

XML proviene de eXtensible Markup Language ("Lenguaje de Marcas Extensible"). Se trata de un metalenguaje (un lenguaje que se utiliza para decir algo acerca de otro) extensible de

etiquetas que fue desarrollado por el Word Wide Web Consortium (W3C), una sociedad mercantil internacional que elabora recomendaciones para la World Wide Web. Las bases de datos, los documentos de texto, las hojas de cálculo y las páginas web son algunos de los campos de aplicación del XML. El metalenguaje aparece como un estándar que estructura el intercambio de información entre las diferentes plataformas.

A continuación, se muestra un ejemplo básico del archivo de configuración, que deberá ubicarse en la carpeta **src/main/resources** relativa a la raíz del proyecto:

#### **hibernate.cfg.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/poo-cine
</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
  </session-factory>
</hibernate-configuration>
```

Podemos ver en el ejemplo anterior que existe una sección **hibernate-configuration** que posee una subsección llamada **session-factory** que es donde colocaremos todas las **propiedades** de la configuración, tales como la ruta de configuración a nuestra base de datos y las credenciales de autenticación.

La propiedad **hibernate.dialect** especifica que versión del lenguaje SQL se usará para confeccionar las consultas a la base de datos; en este caso en particular le indicamos que optimice las consultas para MySQL.

La propiedad **hibernate.connection.driver\_class** especifica que clase será la encargada de proveer el acceso a la base de datos, es decir quién será el encargado de enviar las consultas SQL al motor de base de datos y entregar la respuesta a nuestra aplicación; en este caso en particular utilizamos el controlador de JDBC de MySQL.

La propiedad **hibernate.connection.url** indica la ruta de conexión a la base de datos, es decir cuál será el protocolo de conexión, la IP del host en donde se encuentra instalado el servidor de bases de datos, el nombre de la base de datos a conectarnos, entre otras opciones de conexión. Su sintaxis es la siguiente:

```
jdbc:mysql://host:puerto/database
jdbc:mysql://host:puerto/database?user=usuario&password=clave
```

Las propiedades restantes **username** y **password** hacen referencia a los datos de la cuenta de usuario a usar para la conexión como vimos en la sección **Seguridad**.

Otras propiedades útiles para el archivo de configuración son **show\_sql** y **format\_sql** que pueden recibir valores *true* o *false* y permiten mostrar en tiempo de ejecución las consultas SQL que Hibernate genera para obtener los datos de la base de datos.

Una vez creado el archivo de configuración es necesario obtener una instancia de la clase *SessionFactory*, que es la que nos proveerá acceso a la *Session* actual de Hibernate, clase que es encargada de procesar todos nuestros pedidos de mapeo. Para obtener una instancia de esta clase debemos ejecutar el siguiente bloque de código:

```
final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
    .configure() // obtiene los valores de hibernate.cfg.xml
    .build();

try {
    SessionFactory sessionFactory = new MetadataSources(registry)
        .buildMetadata().buildSessionFactory();
}
catch (Exception e) {
    // La variable registry será destruida al crear el SessionFactory,
    // pero como han surgido problemas en el proceso de creación lo hacemos
    // manualmente
    StandardServiceRegistryBuilder.destroy(registry );
}
```

Este proceso sólo debe realizarse una vez para toda la aplicación al inicio de la ejecución, luego todos nuestros elementos de la capa de acceso a datos DAO (Data Access Object) harán uso de la *SessionFactory* anterior para pedir servicios de mapeo a Hibernate.

Luego en cualquier método de nuestra capa de acceso a datos podemos obtener una instancia de sesión para comunicarnos con Hibernate de la siguiente forma:

```
Session session = sessionFactory.openSession();
session.beginTransaction();
// nuestro código de operaciones con Hibernate
session.getTransaction().commit();
session.close();
```

Una *org.hibernate.Session* está diseñada para representar una única unidad de trabajo (una porción de código atómica, como vimos anteriormente en **ACID**). ¿Qué hace *sessionFactory.getCurrentSession()*? Primero, la puede llamar tantas veces como desee y en donde quiera, una vez consiga su *org.hibernate.SessionFactory*. El método *getCurrentSession()* siempre retorna la unidad de trabajo "actual".

Es importante notar el llamado al método *close()* luego de realizar las operaciones con Hibernate para poder liberar los recursos reservados al momento de la ejecución. Además, se puede ver que las operaciones de Hibernate se encuentran entre las sentencias de comienzo y

confirmación de una transacción; esto es importante ya que de no ejecutar nuestras sentencias dentro de una transacción de base de datos, las mismas no serán llevadas a cabo.

Una `org.hibernate.Session` se inicia cuando se realiza la primera llamada a `getCurrentSession()` para el hilo actual, es decir para la sesión actual. Luego Hibernate la vincula al hilo actual. Cuando termina la transacción, ya sea por medio de guardar o deshacer los cambios, Hibernate desvincula automáticamente la `org.hibernate.Session` del hilo y la cierra por usted. Si llama a `getCurrentSession()` de nuevo, obtiene una `org.hibernate.Session` nueva y obtiene una nueva `org.hibernate.Session` unidad de trabajo.

En relación con la unidad del campo de trabajo, ¿Se debería utilizar `org.hibernate.Session` de Hibernate para ejecutar una o varias operaciones de la base de datos? El ámbito de una `org.hibernate.Session` de Hibernate es flexible pero nunca debe diseñar su aplicación para que utilice una nueva `org.hibernate.Session` de Hibernate para cada operación de la base de datos. Aunque lo utilizamos en los ejemplos, se considera una mala práctica la sesión-por-operación.

**Nota:** Es posible obviar las llamadas a inicio y confirmación de transacciones para todas las operaciones de Hibernate asignando el valor *true* al parámetro de configuración `hibernate.connection.autocommit`. En caso de que necesitemos crear una transacción que involucre más de una operación en la base de datos podemos hacerlo con las llamadas a los métodos anteriores de forma explícita. Por ejemplo:

```
Session sess = factory.openSession();

Transaction tx = null;

try {
    // iniciamos la transaccion
    tx = sess.beginTransaction();

    // operaciones de acceso a datos
    // ...

    // si todo va OK, confirmamos
    tx.commit();
}
catch (RuntimeException e) {
    // ha ocurrido un error, volvemos atrás los cambios realizados
    if (tx != null) tx.rollback();

    // podemos mostrar un mensaje de error también
    throw e;
}
```



## El archivo de mapeo XML

Para poder indicar a Hibernate como vincular nuestras clases de entidad con tablas de la base de datos usamos también archivos XML denominados HBM, Hibernate Mapping File. Es una buena práctica crear un archivo de mapeo para cada clase de entidad que vayamos a utilizar, y su sintaxis es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="NombreDeLaClase" table="tabla_en_base_de_datos">
    [PROPIEDADES]
  </class>
</hibernate-mapping>
```

En la sección **Class** deberemos identificar todas las propiedades a mapear para nuestra clase de entidad, ya sean atributos miembro de la clase como relaciones o colecciones.

Además, para poder indicar a Hibernate que este archivo debe ser considerado a la hora de iniciar la SessionFactory, debemos agregar su referencia en el archivo de configuración hibernate.cfg.xml de la siguiente forma:

```
<mapping resource="Archivo.hbm.xml"/>
```

Veamos un ejemplo, sea la clase de entidad Género que representa una posible clasificación para las Películas del complejo de Cines la siguiente:

```
package poo.cine;

public class Genero {

    private Integer id;
    private String nombre;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
```

```

        this.nombre = nombre;
    }

    @Override
    public String toString() {
        return this.nombre;
    }
}

```

Es importante destacar que se agregó el atributo id (identificador) del tipo de datos Integer, ésta será la **clave primaria** para identificar a los géneros en la base de datos y crear las relaciones que sean necesarias como **claves foráneas**. El valor de este campo id es un número único correlativo de tipo entero. Algunas consideraciones sobre las clases de entidad:

- Las clases de entidad deben respetar el estándar de JavaBean para el nombrado de los métodos get y set, así como la visibilidad privada para todos los atributos miembro.
- El constructor sin argumentos (ya sea declarado de forma explícita o por defecto), que también es parte de la convención JavaBean, es un requisito para todas las clases de entidad. Hibernate necesita poder crear los objetos en tu nombre, usando una estrategia llamada Reflexión Java. Este constructor puede ser privado, sin embargo, es recomendable que su visibilidad sea pública o al menos protegida a nivel del paquete.

Para la clase anterior tenemos la siguiente consulta SQL de creación de la tabla correspondiente:

```

CREATE TABLE `generos` (
  `id` int(10) unsigned NOT NULL,
  `nombre` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
);

```

En este punto es importante destacar que hemos decidido asignar al tipo de datos de Java String (cadena de texto sin máximo de caracteres) el tipo de datos de MySQL VARCHAR que representa una cadena de caracteres con un valor máximo de longitud, en este caso 45.

Ahora bien, para el archivo XML de mapeo de Hibernate de la clase anterior tenemos lo siguiente:

#### Genero.hbm.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="poo.cine.Genero" table="generos">
    <id name="id">
      <generator class="increment"></generator>
    </id>
    <property name="nombre"></property>
  </class>
</hibernate-mapping>

```

Módulo Base de Datos

Versión 1.1 – Liberada el 08/02/2017

```
</class>
</hibernate-mapping>
```

Es importante notar que existe una sección `<id>` que especifica cuál es la columna que corresponde al identificador de la entidad, es decir su **clave primaria** que indicamos con el atributo *name*. Además, en su interior podemos ver una sección denominada *generador* que le indica a Hibernate que estrategia debe utilizar para asignar nuevos identificadores para las entidades que sean creadas. En este caso usamos una estrategia incremental, que obtiene el próximo valor numérico que corresponda.

Luego podemos ver la primera propiedad indicada con `<property>` en donde indicamos que existe un atributo miembro de la clase que además tiene el mismo nombre de columna en la base de datos: *nombre*. En caso de que la columna tenga un nombre diferente debemos indicarlo con el atributo **column** de la siguiente forma:

```
<property name="precio" column="precio_en_pesos" type="float" />
```

En el ejemplo anterior también hemos indicado el tipo de datos que se utilizará para asignar el valor del atributo miembro en la clase de entidad correspondiente. Esto puede ser de utilidad en algunos casos en donde Hibernate no pueda inferir el tipo de datos de forma automática, como es el caso de los números guardados con `java.math.BigDecimal`.

Ahora necesitamos indicar en el archivo de configuración de Hibernate que nuestro nuevo XML de mapeo debe ser considerado al momento de ejecución, por lo que agregamos la siguiente línea al final del archivo `hibernate.cfg.xml` antes de `</session-factory></hibernate-configuration>`:

```
<mapping resource="Genero.hbm.xml"/>
```

Entonces, al tener ya configurado nuestro mapeo para la clase de entidad, podemos guardar un nuevo Género de la siguiente forma:

```
Genero acción = new Genero();
acción.setNombre("Accion");

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(accion);
session.getTransaction().commit();
session.close();
```

Con el ejemplo de código anterior habremos persistido la entidad instancia de la clase Género en nuestra base de datos MySQL, podemos consultar el resultado de la inserción realizando un `SELECT` a la tabla correspondiente:

```
SELECT * FROM generos;
```

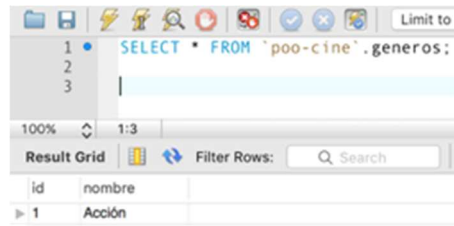


Fig. 30: Selección de todos los géneros

A su vez también es posible usar el método `saveOrUpdate()` que recibe el mismo parámetro que el caso anterior pero permite guardar los cambios realizados sobre un objeto que ya existía en nuestra base de datos, al contrario del método antes mencionado que se limita a la inserción de nuevas entidades.

De la misma forma, para obtener la instancia del Género con un ID específico podemos ejecutar el siguiente código:

```
Session session = sessionFactory.openSession();
session.beginTransaction();
Genero genero = session.get(Genero.class, 1);
System.out.println("Género obtenido: " + genero.toString());
session.getTransaction().commit();
session.close();
```

El resultado de la ejecución es el siguiente:

Género obtenido: Accion

De esta forma, para cualquier clase ya mapeada por Hibernate podemos realizar una obtención de un objeto de entidad usando el método `get()` que recibe por parámetros la clase correspondiente y la clave primaria, en el caso anterior el valor de ID: 1.

Para obtener todas las entidades podemos usar el método `list()` de **JPA Criteria**, que es una forma declarativa de realizar consultas utilizando una notación de métodos encadenados de **Java**, veremos más sobre **Criteria** en esta sección.

```
Session session = sessionFactory.openSession();
session.beginTransaction();

CriteriaQuery<Genero> query =
    session.getCriteriaBuilder().createQuery(Genero.class);
query.select(query.from(Genero.class));

List<Genero> generos = session.createQuery(query).list();
System.out.println("Generos: " + generos.size());

session.getTransaction().commit();
session.close();
```

Módulo Base de Datos

Versión 1.1 – Liberada el 08/02/2017

En el ejemplo anterior, luego de abrir una nueva sesión de Hibernate para poder interactuar con el ORM obtenemos una instancia de CriteriaQuery a la cual delegamos su creación a la sesión. Este objeto query representa una unidad de construcción programática de una consulta al ORM, la cual podremos especificar agregando restricciones y otras opciones. Es importante destacar que esta query en particular se encuentra estrechamente vinculada a la clase Genero, ya que su resultado será un conjunto de objetos instancia de esta clase, los cuales obtenemos luego mediante el llamado al método list(). Se puede notar que la sintaxis de construcción de la consulta query tiene similitudes con el lenguaje SQL, para más información y un listado completo de las restricciones disponibles, se puede consultar la documentación de JPA Criteria para Hibernate.

## Mapeo de relaciones

### Colecciones

Para poder mapear las colecciones de elementos como en la relación entre las clases Programacion y Funciones, donde una Programacion puede tener muchas Funciones pero una Función pertenece solo a una Programacion, podríamos tener sentencias DDL simplificadas de la siguiente forma:

```
CREATE TABLE `programaciones` (
  `id` INT UNSIGNED NOT NULL COMMENT '',
  `fecha_inicio` DATE NOT NULL COMMENT '',
  `fecha_fin` DATE NOT NULL COMMENT '',
  `fecha_hora_creada` DATETIME NOT NULL COMMENT '',

  PRIMARY KEY (`id`) COMMENT '');

CREATE TABLE `funciones` (
  `id` INT UNSIGNED NOT NULL COMMENT '',
  `id_programacion` INT UNSIGNED NOT NULL COMMENT '',
  `dia_semana` TINYINT(3) UNSIGNED NOT NULL COMMENT '',
  `id_pelicula` INT UNSIGNED NOT NULL COMMENT '',

  PRIMARY KEY (`id`) COMMENT '',
  INDEX `fk_funciones_programacion_idx` (`id_programacion` ASC) COMMENT '',
  CONSTRAINT `fk_funciones_programacion`
    FOREIGN KEY (`id_programacion`)
      REFERENCES `poo-cine`.`programaciones` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```

Y en el archivo de mapeo XML de la clase Programación, deberíamos agregar la siguiente etiqueta para completar el atributo miembro List<Funcion> funciones de la definición de la clase:

```
<hibernate-mapping>
  <class name="poo.cine.Programacion" table="programaciones">
    <id name="id">
      <generator class="increment"></generator>
    </id>
```

```

    <!-- otras propiedades -->

    <list name="funciones">
        <key column="id_programacion" not-null="true"/>
        <one-to-many class="poo.cine.Funcion"/>
    </list>

</class>
</hibernate-mapping>

```

Es importante destacar el atributo not-null puesto en true para la tabla de Funciones, ya que en ningún momento puede existir una Función que no pertenezca a una Programación. Esta regla de integridad también es importante a la hora de persistir las entidades en la base de datos.

### Muchos a Uno

Para relaciones de muchos a uno, como en el caso de Película con Genero, donde una Película tiene un Genero, el cual puede ser compartido por múltiples instancias de diferentes Películas, tendremos la siguiente tabla simplificada en MySQL:

```

CREATE TABLE `peliculas` (
  `id` INT UNSIGNED NOT NULL COMMENT '',
  `nombre` VARCHAR(45) NOT NULL COMMENT '',
  `id_genero` INT UNSIGNED NOT NULL COMMENT '',

  PRIMARY KEY (`id`) COMMENT '',

  INDEX `fk_peliculas_genero_idx` (`id_genero` ASC) COMMENT '',
  CONSTRAINT `fk_peliculas_genero`
    FOREIGN KEY (`id_genero`)
    REFERENCES `poo-cine`.`generos` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

```

Es importante destacar que esta relación de uno a muchos se encuentra representada en el esquema de la tabla como una **clave foránea**.

Ahora para la clase Película tendríamos el siguiente código y su archivo de mapeo:

### Pelicula.java

```

public class Pelicula {

    private Integer id;
    private String nombre;
    private Genero genero;

```

```

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Genero getGenero() {
        return genero;
    }

    public void setGenero(Genero genero) {
        this.genero = genero;
    }

    @Override
    public String toString() {
        return this.nombre;
    }
}

```

### **Película.hbm.xml**

```

<hibernate-mapping>
  <class name="poo.cine.Pelicula" table="peliculas">
    <id name="id">
      <generator class="increment"></generator>
    </id>
    <property name="nombre"></property>

    <many-to-one name="genero" class="poo.cine.Genero">
      <column name="id_genero"></column>
    </many-to-one>
  </class>
</hibernate-mapping>

```

Para indicar la relación usamos la etiqueta XML de Hibernate <many-to-one> con la siguiente sintaxis:

```
<many-to-one name="nombreAtributo" class="NombreClase">
  <column name="columna_fk"></column>
</many-to-one>
```

Así podemos insertar Películas con un Género de la siguiente forma:

```
Session session = sessionFactory.openSession();

session.beginTransaction();

Genero accion = session.get(Genero.class, 1);

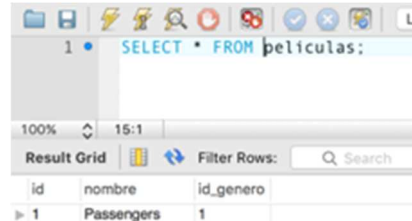
Película peli = new Película();
peli.setNombre("Passengers");
peli.setGenero(accion);

session.save(peli);

session.getTransaction().commit();
session.close();
```

Y el resultado podemos verlo en nuestra tabla haciendo un SELECT de la siguiente forma:

```
SELECT * FROM películas;
```



id	nombre	id_genero
1	Passengers	1

Fig. 31: Selección de todas las películas

Y realizando un JOIN con la tabla de géneros podemos ver a qué género corresponde cada película:

```
SELECT películas.id, películas.nombre, generos.nombre AS 'genero'
FROM películas
JOIN generos ON películas.id_genero = generos.id;
```



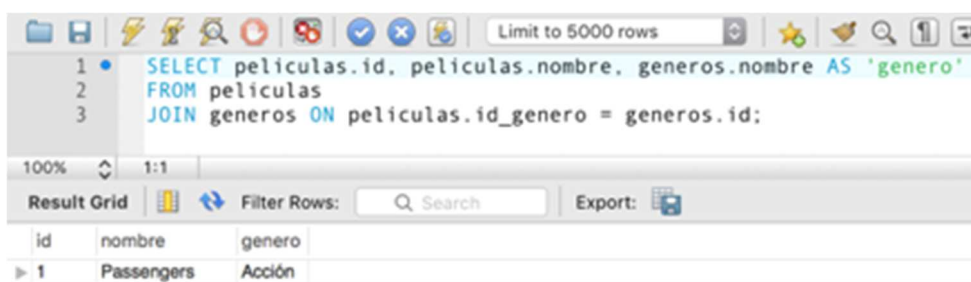


Fig. 32: Resultado de la ejecución del JOIN entre las dos tablas

Uno a uno

Para relaciones de uno a uno, como en el caso de Empleado con Usuario en la cual un Empleado puede o no tener un Usuario, pero un Usuario sólo corresponde a un Empleado podemos tener la siguiente definición de esquema simplificado:

```
CREATE TABLE `poo-cine`.`empleados` (
  `id` INT UNSIGNED NOT NULL COMMENT '',
  `nombre` VARCHAR(45) NULL COMMENT '',
  `apellido` VARCHAR(45) NULL COMMENT '',
  PRIMARY KEY (`id`) COMMENT '');

CREATE TABLE `poo-cine`.`usuarios` (
  `id` INT UNSIGNED NOT NULL COMMENT '',
  `email` VARCHAR(45) NOT NULL COMMENT '',
  `password` VARCHAR(45) NOT NULL COMMENT '',
  `id_empleado` INT UNSIGNED NOT NULL COMMENT '',
  PRIMARY KEY (`id`) COMMENT '',
  INDEX `fk_usuarios_empleado_idx` (`id_empleado` ASC) COMMENT '',
  CONSTRAINT `fk_usuarios_empleado`
    FOREIGN KEY (`id_empleado`)
    REFERENCES `poo-cine`.`empleados` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```

Y dentro de nuestra clase **Usuario** tendremos la siguiente definición de la relación:

```
private Empleado empleado;
```

Con sus métodos de *get* y *set* correspondientes. Ahora para indicar que esta relación es de uno a uno, en el archivo de mapeo XML de la clase **Usuario** deberemos especificarla de la siguiente forma:

```
<many-to-one name="empleado"
  class="poo.cine.Empleado"
  column="id_empleado"
```

```
unique="true"
not-null="true" />
```

Es importante destacar que su notación es muy similar a la que usamos en la sección anterior para relaciones de muchos a uno salvo que en este caso el valor de **id\_empleador** debe ser único.

### Muchos a Muchos

Como vimos anteriormente, un ejemplo de relación de muchos a muchos es el de la Película con Personaje, donde una Película tiene diferentes Personajes que a su vez pueden aparecer en diferentes Películas. Un ejemplo de ellos es Harry Potter, el cual aparece en las 8 Películas de la saga original. Para implementar estas relaciones necesitamos una *tabla intermedia*, cuyas sentencias de creación en conjunto con la tabla de personajes es la siguiente:

```
CREATE TABLE `personajes` (
  `id` int(10) unsigned NOT NULL,
  `nombreEnPelicula` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
);
```

```
CREATE TABLE `peliculas_personajes` (
  `id_pelicula` int(10) unsigned NOT NULL,
  `id_personaje` int(10) unsigned NOT NULL,
  PRIMARY KEY (`id_pelicula`,`id_personaje`),
  KEY `fk_peliculas_personajes_personaje_idx` (`id_personaje`),
  CONSTRAINT `fk_peliculas_personajes_pelicula` FOREIGN KEY (`id_pelicula`)
REFERENCES `peliculas` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT `fk_peliculas_personajes_personaje` FOREIGN KEY (`id_personaje`)
REFERENCES `personajes` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION
);
```

Para poder indicar a Hibernate que nuestras entidades de la clase Película pueden tener una colección de Personajes vinculados a través de esta tabla intermedia, debemos indicarlo de la siguiente forma en el archivo HBM:

### En Pelicula.hbm.xml

```
<bag name="personajes" table="películas_personajes">
  <key column="id_pelicula" />
  <many-to-many class="Personaje" column="id_personaje" />
</bag>
```

En el caso de que necesitemos indicar que se trata de una **relación bidireccional**, donde el Personaje pueda conocer en qué Películas participa, deberemos hacer el análogo en su archivo de mapeo HBM indicando que se trata de la relación inversa. Esto es importante ya que de no

hacerlo Hibernate no sabrá que Entidad es la dueña de la relación con el riesgo de duplicar los objetos en memoria:

#### En Personaje.hbm.xml

```
<bag name="películas" table="películas_personajes" inverse="true">
  <key column="id_personaje" />
  <many-to-many class="Película" column="id_pelicula" />
</bag>
```

#### Operaciones sobre las Entidades

Como mencionamos anteriormente, Hibernate nos permite obtener Sesiones para encapsular unidades de trabajo como operaciones sobre nuestra base de datos. Es responsabilidad del objeto sessionFactory otorgarnos una instancia de Session con la que podemos trabajar, pero queda de nuestro lado implementar correctamente las transacciones que finalmente impactarán los cambios sobre la base de datos. A continuación, repasaremos algunas de las operaciones más comunes que podemos llevar a cabo usando la *session*.

#### Persistir una nueva Entidad

El análogo a la operación de INSERT en la base de datos en el método save() que nos permite guardar un nuevo objeto de entidad en la base de datos.

```
Genero acción = new Genero();
```

```
acción.setNombre("Accion");
```

```
Session session = sessionFactory.openSession();
```

```
session.beginTransaction();
```

```
session.save(accion);
```

```
session.getTransaction().commit();
```

```
session.close();
```

Es importante recordar que debemos especificar en nuestro archivo de mapeo cuál será la estrategia de generación de valores <generator> para nuestra **clave primaria**, en este caso, la columna numérica ID. En el desarrollo de este módulo usaremos la estrategia *increment*, que se encarga de obtener el número siguiente al máximo que existe para el campo en la base de datos.

Existen otras estrategias de generación, entre las que podemos mencionar:

<b>native</b>	Hibernate usará alguno de los siguientes métodos dependiendo de la base de datos. De esta forma si cambiamos de base de datos se seguirá usando la mejor forma de generar la clave primaria
<b>identity</b>	Hibernate usará el valor de la columna de tipo autoincremento. Es decir, que al insertar la fila, la base de datos le asignará el valor. La columna de base de datos debe ser de tipo autonumérico
<b>sequence</b>	Se utiliza una secuencia como las que existen en Oracle o PostgreSQL , no es compatible con MySQL. La columna de base de datos debe ser de tipo numérico
<b>increment</b>	Se lanza una consulta SELECT MAX() contra la columna de la base de datos y se obtiene el valor de la última clave primaria, incrementando el nº en 1. La columna de base de datos debe ser de tipo numérico

Módulo Base de Datos

Versión 1.1 – Liberada el 08/02/2017

<b>uuid.hex</b>	Hibernate genera un identificador único como un String. Se usa para generar claves primarias únicas entre distintas bases de datos. La columna de base de datos debe ser de tipo alfanumérico.
<b>guid</b>	Hibernate genera un identificador único como un String pero usando las funciones que provee SQL Server y MySQL. Se usa para generar claves primarias únicas entre distintas bases de datos. La columna de base de datos debe ser de tipo alfanumérico.
<b>foreign</b>	Se usará el valor de otro objeto como la clave primaria. Un uso de ello es en relaciones <i>uno a uno</i> donde el segundo objeto debe tener la misma clave primaria que el primer objeto a guardar.

Obtener una entidad por su clave primaria

Para obtener una Entidad por su clave primaria usamos el método `get()` de la sesión de la siguiente forma:

```
Session session = sessionFactory.openSession();
session.beginTransaction();
Genero genero = session.get(Genero.class, 1);
session.getTransaction().commit();
session.close();
```

Al especificar como parámetro del método `get()` la clase sobre la cual vamos a obtener la entidad, Hibernate puede confeccionar la consulta SQL correspondiente donde buscará por el valor de la clave primaria, en este caso el ID 1.

Actualizar una Entidad

Para poder guardar los cambios sobre una Entidad que ya hemos obtenido del almacenamiento persistente, debemos utilizar el método `update()` o `saveOrUpdate()`. El último antes realizará una comprobación de que la entidad no exista antes en la base de datos, en tal caso funcionará de igual forma que el `save()` insertando un nuevo registro.

```
Session session = sessionFactory.openSession();
session.beginTransaction();

Genero genero = session.get(Genero.class, 1);
genero.setNombre("Acción y Aventuras");
session.saveOrUpdate(genero);

session.getTransaction().commit();
session.close();
```

Eliminar una Entidad

Hibernate también nos permite eliminar entidades a través de la llamada al método `delete()` de la sesión actual. Es importante destacar que las operaciones de eliminación estarán sujetas

a las restricciones de integridad referencial que hayamos declarado a la hora de crear el schema de nuestra base de datos.

En caso de que la eliminación no sea posible por motivos de integridad referencial que no puedan ser resueltos en cascada, obtendremos la siguiente excepción en tiempo de ejecución: **org.hibernate.exception.ConstraintViolationException**.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
```

```
Genero genero = session.get(Genero.class, 1);
session.delete(genero);
```

```
session.getTransaction().commit();
session.close();
```

### Buscar Entidades

Para la obtención de varias entidades utilizamos la API de Criterias de Hibernate compatible con la especificación JPA, una parte de la especificación de Java. Por lo tanto, no existe realmente como framework, sino que es simplemente un documento de especificación en el cual se detallan los principios básicos de gestión de la capa de persistencia en el mundo de **Java EE**. Hibernate en cambio **es una implementación concreta** y se trata de un framework que gestiona la capa de persistencia a través de ficheros XML o anotaciones.

La relación que existe entre JPA e Hibernate es que este último implementa como parte de su código la especificación de JPA. Es decir podemos usar Hibernate para construir una capa de persistencia apoyándonos en las definiciones y reglas que la especificación de JPA, aunque no es obligatorio.

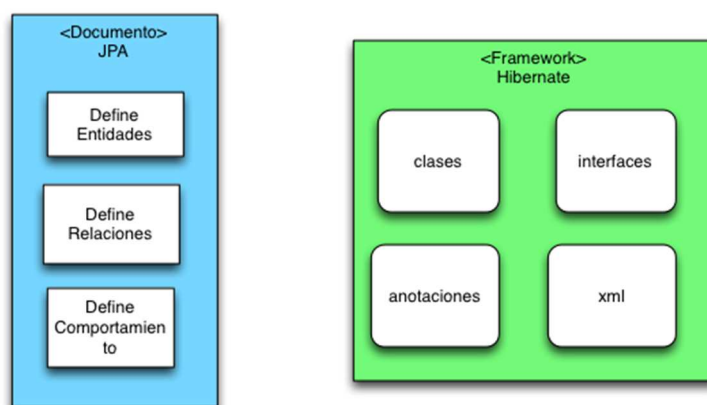


Fig. 33: Relación entre el framework Hibernate y la especificación JPA

En sí podemos considerar a una Criteria como un objeto auxiliar que nos permite construir dinámicamente mediante llamadas a métodos, una consulta Query a la base de datos que luego nuestro objeto session se encargará de ejecutar. La principal ventaja de usar Criteria es que se utiliza una sintaxis orientada a objetos para confeccionar las consultas, mediante llamadas a métodos y paso de parámetros hasta especificar la totalidad de las opciones y condiciones de filtro.

Módulo Base de Datos

Versión 1.1 – Liberada el 08/02/2017

Esto se traduce en la posibilidad de incluir estructuras de control como decisiones IF que nos permitan especificar condiciones bajo las cuales agregar una restricción a nuestra sentencia WHERE, limitar la cantidad de resultados a un valor obtenido de alguna variable, entre otras aplicaciones.

#### **Veamos un ejemplo:**

```
Session session = sessionFactory.openSession();
session.beginTransaction();

CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Película> query = builder.createQuery(Película.class);
query.select(query.from(Película.class));

List<Película> películas = session.createQuery(query).list();
System.out.println("Películas: " + películas.size());

session.getTransaction().commit();
session.close();
```

En primera instancia creamos un nuevo objeto de consulta **CriteriaQuery** cuyo resultado estará definido en términos de objetos de la clase **Película**. Esto es importante ya que le permite a Java realizar una comprobación en tiempo de compilación de tipos de datos. La responsabilidad de crear nuevas consultas y entregarnos sus instancias corresponde al **CriteriaBuilder**, que es a su vez provisto por el objeto session actualmente en ejecución.

Finalmente obtenemos nuestra colección de Películas llamando al método list() sobre la Query creada por la sesión en base a la CriteriaQuery que confeccionamos anteriormente. El método list() devuelve el conjunto de entidades Película que satisfacen las restricciones especificadas para la Criteria, al no haber especificado ninguna buscamos todas.

En caso de que quisieramos filtrar, por ejemplo por Género, podríamos hacerlo de la siguiente forma:

```
Genero genero = session.get(Genero.class, 2);

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Película> query = builder.createQuery(Película.class);
Root<Película> root = query.from(Película.class);
query.select(root);
query.where(builder.equal(root.get("genero"), genero));

List<Película> películas = session.createQuery(query).list();
```

Existen muchas otras restricciones que podemos agregar a nuestra **Criteria** para construir una consulta con diferentes alternativas de condicionales, aun utilizando condiciones de agrupamiento y orden.

Hibernate es un framework de ORM muy extenso y con un gran soporte por parte de la comunidad, para obtener más información sobre la implementación de diferentes relaciones, restricciones y consultas avanzadas usando el API de Criteria, entre otros temas puedes acudir a la documentación oficial (en inglés) disponible en <http://hibernate.org/orm/documentation/>.

## Fuentes de Información

- **Sommerville, Ian** - “INGENIERÍA DE SOFTWARE” 9na Edición (Editorial Addison-Wesley Año 2011).
- **Ramez, Elmasri, Shamkant b. Navathe** – “Fundamentos de Sistemas de Bases de Datos” 5ta Edición (Pearson Addison 2007)
- **Variables y Constantes:** <http://aurea.es/assets/2-tiposdatoslenguajec.pdf>
- **MySQL Manual:** <http://dev.mysql.com/doc/refman/5.7/en/>
- **Tutorial SQL:** <http://www.w3schools.com/sql/>
- [https://technet.microsoft.com/es-es/library/ms189575\(v=sql.105\).aspx](https://technet.microsoft.com/es-es/library/ms189575(v=sql.105).aspx)
- <http://www.desarrolloweb.com/articulos/2337.php>

## Índice de Figuras

Figura 1: Estructura de un Sistema que utiliza DBMS .....	11
Figura 2: Esquema de un Sistema de Base de Datos Distribuida .....	14
Fig. 3: El modelo de datos jerárquico .....	17
Fig. 4: El modelo de datos de red .....	17
Fig. 5: El modelo de datos relacional.....	18
Fig. 6: El modelo de datos orientado a objetos.....	19
Fig. 7: El modelo de datos orientado a documentos .....	19
Fig. 8: El modelo de datos multidimensional .....	20
Fig. 9: El modelo de datos multidimensional como un cubo .....	20
Fig. 10: El Teorema de CAP.....	23
Fig. 11: Estructura de un data warehouse .....	24
Fig. 12: Representación de una Entidad.....	27
Fig. 13: Representación de una Entidad con su atributo identificador.....	29
Fig. 14: Representación de Entidades y su forma de relacionarse, con su atributo referencial	30
Fig. 15: Vista de Clases del Modelo de Dominio del Complejo de Cines.....	31
Fig. 16: Vista parcial del Modelo de Entidad Relación para representar la relación entre Empleado y Usuario .....	32
Fig. 17: Vista parcial del Modelo de Entidad Relación para representar la relación entre la entidad Película y la entidad Género .....	32
Fig. 18: Vista parcial del Modelo de Entidad Relación para representar la relación entre la entidad Película y la entidad Género .....	33
Fig. 19: Vista parcial del Modelo de Entidad Relación para representar la relación entre las entidades Película y Personaje, con su entidad asociativa .....	34



Figura 20: Relación ESTUDIANTE.....	36
Figura 21: Conceptos vinculados a la Estructura Relacional .....	37
Figura 22: Ejemplo de Relación.....	38
Figura 23: Relación Cliente.....	40
Figura 24: Algunos Motores de Bases de Datos Comerciales .....	43
Fig. 25: Estados de una transacción y sus operaciones.....	46
Fig. 26: Un ejemplo de estructura de índice para la tabla 'empleados' .....	50
Fig.27: Representación de un índice de dos niveles .....	52
Fig.28: Vista parcial del Modelo de Entidad Relación del Complejo de Cines .....	59
Fig. 29: Imposibilidad de mapear el comportamiento en ambos modelos .....	75
Fig. 30: Selección de todos los géneros.....	84
Fig. 31: Selección de todas las películas.....	88
Fig. 32: Resultado de la ejecución del JOIN entre las dos tablas .....	89
Fig. 33: Relación entre el framework Hibernate y la especificación JPA .....	93