

BASE DE DATOS

UN ENFOQUE PRÁCTICO



Libro de apoyo para los cursos de BD1 y BD2 de la carrera Analista Programador de la Universidad ORT Uruguay

El siguiente trabajo es el producto de más de 10 años de docencia en Base de Datos, no pretende sustituir la bibliografía recomendada por cada docente sino por el contrario, es un complemento donde se intenta buscar un hilo conductor basado en un problema único (el problema de las válvulas) enfocado en forma práctica y con resolución de la mayoría de los ejercicios planteados.

Contenido

Módulo 1

Concepto y origen de las BD y de los SGBD
Evolución de los SGBD
Objetivos y funcionalidad de los SGBD

Arquitectura
Modelos de BD
Lenguajes y usuarios
Administración de BD

Módulo 2

Diseño conceptual
 Modelo Entidad Relación
Diseño Lógico
 Transformación del MER a modelo relacional

Módulo 3

El lenguaje de definición de datos (DDL)
Creación de Tablas
Restricciones de Integridad
Otras restricciones (CONSTRAINTS)

Módulo 4

El lenguaje de manipulación de datos (DML)
El lenguaje SQL

INSERT
UPDATE
DELETE
SELECT

Módulo 5

Lenguajes de programación de SGBD

T-SQL

PL/SQL

Módulo 6

Procedimientos almacenados

Funciones almacenadas

Disparadores (triggers)

Vistas

Módulo 1

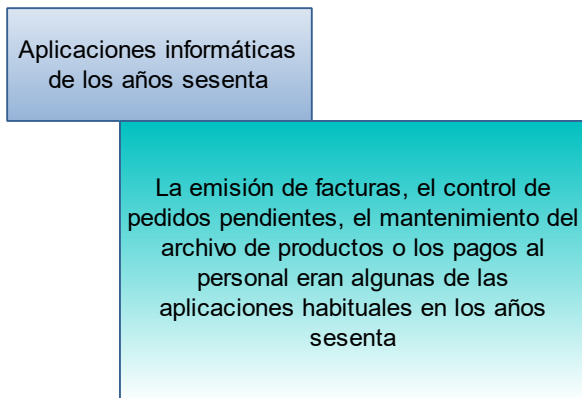
Concepto y origen de las BD y de los SGBD	7
Evolución de los SGBD	9
Objetivos y funcionalidades de los SGBD	10
Arquitectura de los SGBD	13
Modelos de SGBD	15
Lenguajes y usuarios	17
Administración de SGBD	19

Concepto y origen de las BD y de los SGBD

Las aplicaciones informáticas de los años sesenta acostumbraban a darse totalmente por lotes (batch) y estaban pensadas para una tarea muy específica relacionada con muy pocas entidades.

A medida que se integraban las aplicaciones, se tuvieron que interrelacionar sus ficheros y fue necesario eliminar la redundancia.

El nuevo conjunto de ficheros se debía diseñar de modo que estuviesen interrelacionados; al mismo tiempo, las informaciones redundantes (como por ejemplo, el nombre y la dirección de los clientes o el nombre y el precio de los productos), que figuraban en los ficheros de más de una de las aplicaciones, debían estar ahora en un solo lugar.



El software de gestión de ficheros era demasiado elemental para dar satisfacción a todas estas necesidades.

Por ejemplo, el tratamiento de las interrelaciones no estaba previsto, no era posible que varios usuarios actualizaran datos simultáneamente, etc.

De esta realidad surge el concepto de base de datos, una base de datos es un conjunto estructurado de datos que representa entidades y sus interrelaciones.

La representación será única e integrada, a pesar de que debe permitir utilizaciones varias y simultáneas

Evolución de los SGBD

Los SGBD de los años sesenta y setenta (IMS de IBM, IDS de Bull, DMS de Univac, etc.) eran sistemas totalmente centralizados, como corresponde a los sistemas operativos de aquellos años, y al hardware para el que estaban hechos (un gran ordenador para toda la empresa y una red de terminales sin inteligencia ni memoria)

La necesidad de tener una visión global de la empresa y de interrelacionar diferentes aplicaciones que utilizan BD diferentes, junto con la facilidad que dan las redes para la intercomunicación entre ordenadores, ha conducido a los SGBD actuales, que permiten que un programa pueda trabajar con diferentes BD como si se tratase de una sola. Es lo que se conoce como base de datos distribuida.

Tendencias actuales

C/S, SQL y 4GL...

... son siglas de moda desde el principio de los años noventa en el mundo de los sistemas de información.

Hoy día, los SGBD relacionales están en plena transformación para adaptarse a tres tecnologías de éxito reciente, fuertemente relacionadas:
la multimedia, la de orientación a objetos (OO) e Internet y la web.

Objetivos y funcionalidades de los SGBD

Los usuarios podrán hacer consultas de cualquier tipo y complejidad directamente al SGBD.

El SGBD tendrá que responder inmediatamente sin que estas consultas estén preestablecidas; es decir, sin que se tenga que escribir, compilar y ejecutar un programa específico para cada consulta.

Ficheros tradicionales

En los ficheros tradicionales, cada vez que se quería hacer una consulta se tenía que escribir un programa a medida

Los SGBD deben tener las siguientes funcionalidades a saber:

Flexibilidad e Independencia

Para conseguir esta independencia, tanto los usuarios que hacen consultas (o actualizaciones) directas como los profesionales informáticos que escriben programas que las llevan incorporadas, deben poder desconocer las características físicas de la BD con que trabajan. No necesitan saber nada sobre el soporte físico, ni estar al corriente de qué SO se utiliza, qué índices hay, la compresión o no compresión de datos, etc.

De este modo, se pueden hacer cambios de tecnología y cambios físicos para mejorar el rendimiento sin afectar a nadie. Este tipo de independencia recibe el nombre de independencia física de los datos.

Con la independencia física no tenemos suficiente, también queremos que los usuarios (los programadores de aplicaciones o los usuarios directos) no tengan que hacer cambios cuando se modifica la descripción lógica o el esquema de la BD (por ejemplo, cuando se añaden/suprimen entidades o interrelaciones, atributos, etc.)

Independencia lógica de los datos

Por ejemplo, el hecho de suprimir el atributo fecha de nacimiento de la entidad Alumno, y agregar otra entidad Aulas no debería afectar a ninguno de los programas existentes que no utilicen el atributo fecha de nacimiento

Control de Redundancia de Datos

El SGBD debe permitir que el diseñador defina datos redundantes, pero entonces tendría que ser el mismo SGBD el que hiciese automáticamente la actualización de los datos en todos los lugares donde estuviesen repetidos.

Integridad de los Datos

Cuando el SGBD detecte que un programa quiere hacer una operación que va contra las reglas establecidas al definir la BD, no se lo deberá permitir, y le tendrá que devolver un estado de error.

Al diseñar una BD para un SI concreto y escribir su esquema, no sólo definiremos los datos, sino también las reglas de integridad que queremos que el SGBD haga cumplir.

Arquitectura de los SGBD

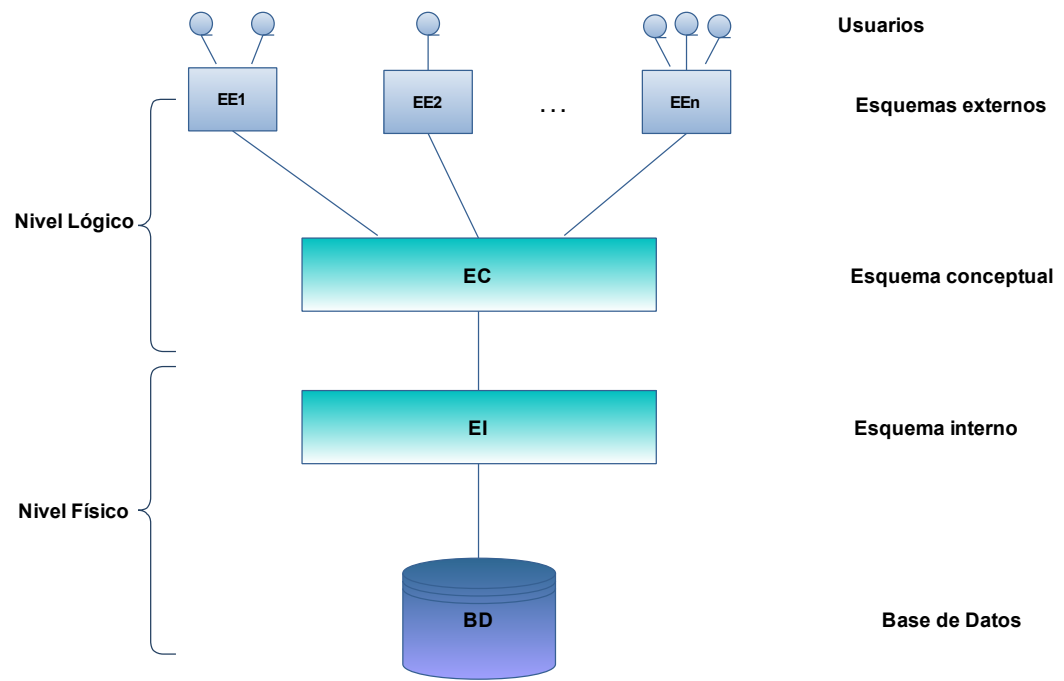
Para trabajar con nuestras BD, los SGBD necesitan conocer su estructura (qué entidades tipo habrá, qué atributos tendrán, etc.).

Los SGBD necesitan que les demos una descripción o definición de la BD. Esta descripción recibe el nombre de esquema de la BD, y los SGBD la tendrán continuamente a su alcance.

De acuerdo con la arquitectura ANSI/SPARC, debía haber tres niveles de esquemas (tres niveles de abstracción).

La idea básica de ANSI/SPARC consistía en descomponer el nivel lógico en dos: el nivel externo y el nivel conceptual.

Denominábamos nivel interno lo que aquí hemos denominado nivel físico.



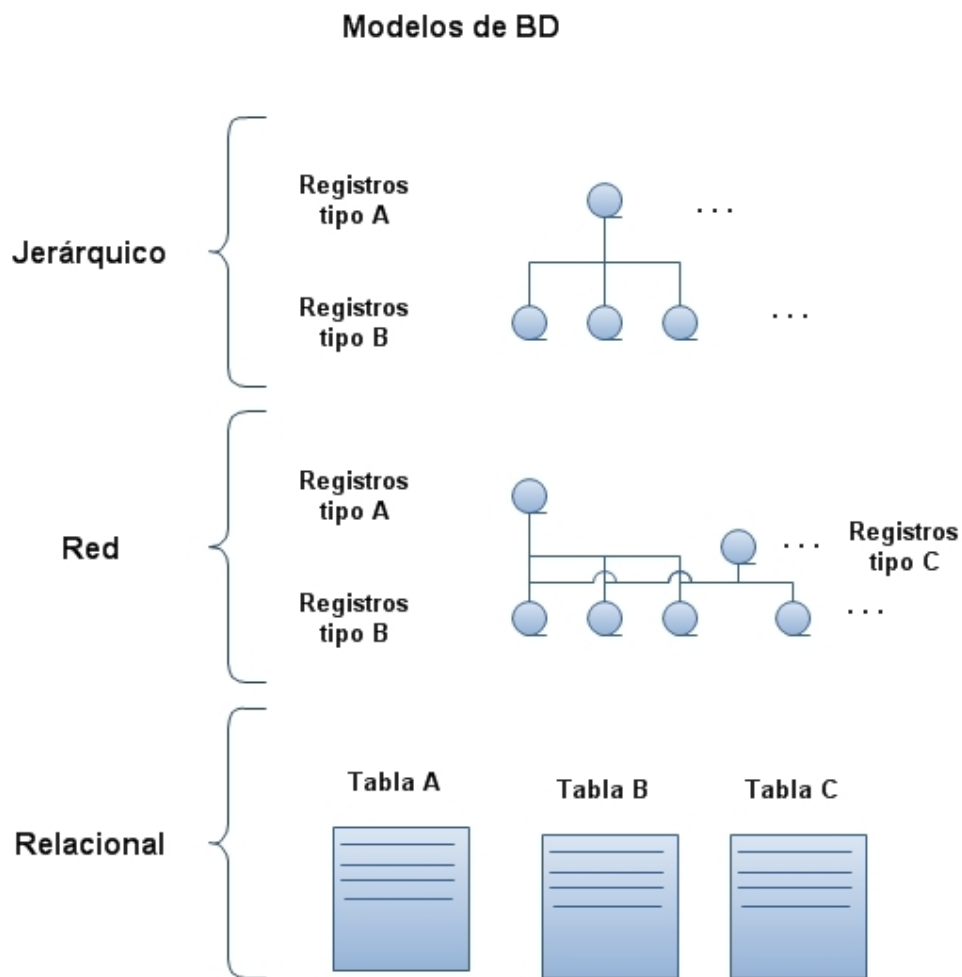
En el esquema conceptual se describirán las entidades tipo, sus atributos, las interrelaciones y las restricciones o reglas de integridad.

El esquema interno o físico contendrá la descripción de la organización física de la BD: caminos de acceso (índices, hashing, apuntadores, etc.), codificación de los datos, gestión del espacio, tamaño de la página, etc.

Modelos de Base de Datos

El conjunto de componentes o herramientas conceptuales que un SGBD proporciona para modelar recibe el nombre de modelo de BD.

Los cuatro modelos de BD más utilizados en los SI son el modelo relacional, el modelo jerárquico, el modelo en red y el modelo relacional con objetos.



Así como en los modelos pre relacionales (jerárquico y en red), las estructuras de datos constan de dos elementos básicos (los registros y las interrelaciones), en el modelo relacional constan de un solo elemento: la tabla, formada por filas y columnas. Las interrelaciones se deben modelar utilizando las tablas.

Otra diferencia importante entre los modelos pre relacionales y el modelo relacional es que el modelo relacional se limita al nivel lógico (no hace absolutamente ninguna consideración sobre las representaciones físicas). Es decir, nos da una independencia física de datos total.

Esto es así si hablamos del modelo teórico, pero los SGBD del mercado nos proporcionan una independencia limitada.

Lenguajes y usuarios

Para comunicarse con el SGBD, el usuario, ya sea un programa de aplicación o un usuario directo, se vale de un lenguaje.

Hay muchos lenguajes diferentes, según el tipo de usuarios para los que están pensados y el tipo de cosas que los usuarios deben poder expresar con ellos.

Hay lenguajes especializados en la escritura de esquemas; es decir, en la descripción de la BD. Se conocen genéricamente como DDL o “data definition language”

Otros lenguajes están especializados en la utilización de la BD (consultas y mantenimiento).

Se conocen como DML o “data management language”.

El lenguaje DDL

Son las instrucciones SQL que nos permiten definir la estructura de la información:

CREATE
DROP
ALTER

El lenguaje DML

Son las instrucciones SQL que nos permiten llevar a cabo tareas de consultas o manipulación de datos en un sistema de BD

INSERT
DELETE
UPDATE
SELECT

Administración de SGBD

Hay un tipo de usuario especial: el que realiza tareas de administración y control de la BD.

Una empresa o institución que tenga SI contruidos en torno a BD necesita que alguien lleve a cabo una serie de funciones centralizadas de gestión y administración, para asegurar que la explotación de la BD es la correcta.

Este conjunto de funciones se conoce con el nombre de administración de BD (DBA), y los usuarios que hacen este tipo especial de trabajo se denominan administradores de BD.

Módulo 2

Diseño conceptual	21
Entidades	22
Atributos	23
Claves	25
Interrelaciones	27
Cardinalidad	29
Autorrelaciones	32
Entidades débiles, categorización, agregación	33
Caso de estudio “El problema de las válvulas”	36

Diseño conceptual

El diseño de una base de datos consiste en definir la estructura de los datos que debe tener la base de datos de un sistema de información determinado.

En el caso relacional, esta estructura será un conjunto de esquemas de relación con sus atributos, dominios de atributos, claves primarias, claves foráneas, etc.

El nombre completo del modelo ER es entity-relationship, y proviene del hecho de que los principales elementos que incluye son las entidades y las interrelaciones (entities y relationships).

Traduciremos este nombre por 'entidad-interrelación'.

Entidades

Por entidad entendemos un objeto del mundo real que podemos distinguir del resto de objetos y del que nos interesan algunas propiedades.

Ejemplos de entidad

Algunos ejemplos de entidad son un empleado, un producto o un despacho. También son entidades otros elementos del mundo real de interés, menos tangibles pero igualmente diferenciables del resto de objetos; por ejemplo, una asignatura impartida en una universidad, un préstamo bancario, un pedido de un cliente, etc.

Las entidades son representadas por un rectángulo con su nombre en mayúsculas en el interior



Atributos

Las propiedades de los objetos que nos interesan se denominan atributos.

Ejemplos de atributo

Sobre una entidad empleado nos puede interesar, por ejemplo, tener registrados su ci, su dirección, su nombre, su apellido, su número de registro ante el BPS y su sueldo como atributos.

Los atributos se representan mediante su nombre en minúsculas unido con un guión al rectángulo de la entidad a la que pertenecen. Muchas veces, dado que hay muchos atributos para cada entidad, se listan todos aparte del diagrama para no complicarlo.

Notación diagramática



Todos los atributos tienen que ser univaluados.

Un atributo es univaluado si tiene un único valor para cada ocurrencia de una entidad.

Ejemplo de atributo univaluado

El atributo sueldo de la entidad empleado, por ejemplo, toma valores del dominio de los reales y únicamente toma un valor para cada empleado concreto; por lo tanto, ningún empleado puede tener más de un valor para el sueldo.

Claves

Una entidad debe ser distinguible del resto de objetos del mundo real.

Esto hace que para toda entidad sea posible encontrar un conjunto de atributos que permitan identificarla.

Este conjunto de atributos forma una clave de la entidad.

Ejemplo de clave

La entidad empleado tiene una clave que consta del atributo ci porque todos los empleados tienen números de cedula de identidad diferentes.

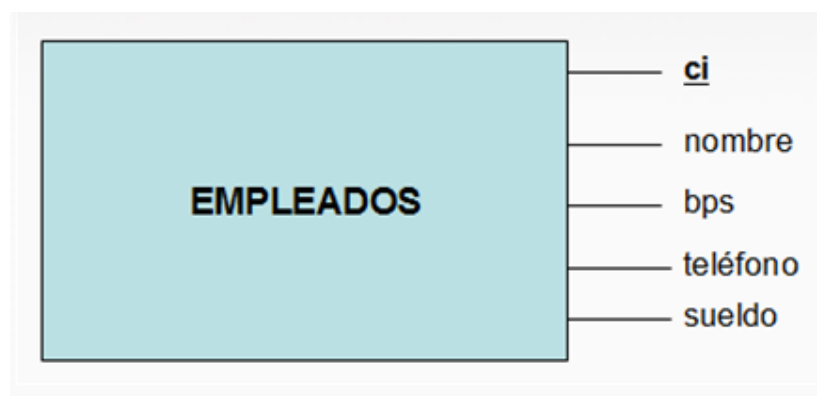
Una determinada entidad puede tener más de una clave; es decir, puede tener varias claves candidatas.

Ejemplo de clave candidata

La entidad empleado tiene dos claves candidatas, la que está formada por el atributo ci y la que está constituida por el atributo bps, teniendo en cuenta que el registro en el BPS también será diferente para cada uno de los empleados.

El diseñador elige una clave primaria entre todas las claves candidatas.

En la notación diagramática, la clave primaria se subraya para distinguirla del resto de las claves



Interrelaciones

Se define interrelación como una asociación entre entidades, las interrelaciones se representan en los diagramas del modelo ER mediante un rombo.

Junto al rombo se indica el nombre de la interrelación con letras mayúsculas.

Ejemplo de interrelación:

Consideremos una entidad empleado y una entidad despacho y supongamos que a los empleados se les asignan despachos donde trabajar.

Entonces hay una interrelación entre la entidad empleado y la entidad despacho.

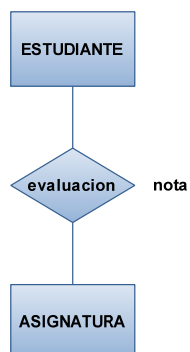
Esta interrelación, que podríamos denominar asignación, asocia a los empleados con las oficinas donde trabajan.



En ocasiones interesa reflejar algunas propiedades de las interrelaciones.

Por este motivo, las interrelaciones pueden tener también atributos.

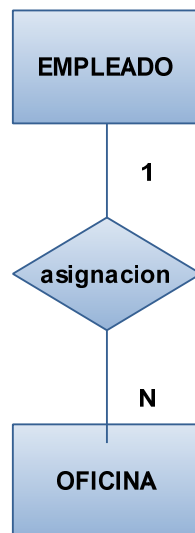
Los atributos de las interrelaciones, igual que los de las entidades, tienen un cierto dominio, deben tomar valores atómicos y deben ser univaluados.



Cardinalidad de las relaciones

La cardinalidad de una interrelación expresa el tipo de correspondencia que se establece entre las ocurrencias de entidades asociadas con la interrelación.

En el caso de las interrelaciones binarias, expresa el número de ocurrencias de una de las entidades con las que una ocurrencia de la otra entidad puede estar asociada según la interrelación.



Una interrelación binaria entre dos entidades puede tener tres tipos de cardinalidad:

- Cardinalidad uno a uno (1:1).

La cardinalidad 1:1 se denota poniendo un 1 a lado y lado de la interrelación.

- Cardinalidad uno a muchos (1:N).

La cardinalidad 1:N se denota poniendo un 1 en un lado de la interrelación y una N en el otro.

- Cardinalidad muchos a muchos: (N:N).

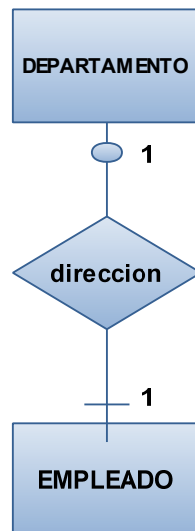
La cardinalidad N:N se denota poniendo una N en uno de los lados de la interrelación, y una N en el otro.

Dependencias de existencia

En algunos casos, una entidad individual sólo puede existir si hay como mínimo otra entidad individual asociada con ella mediante una interrelación binaria determinada.

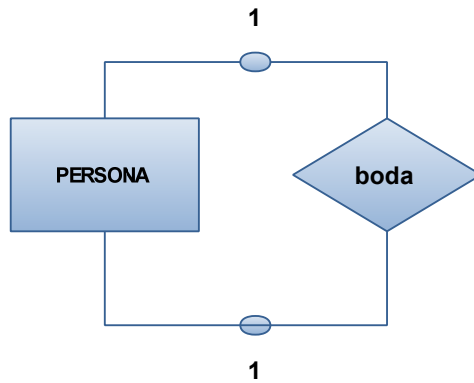
En estos casos, se dice que esta última entidad es una entidad obligatoria en la interrelación. Cuando esto no sucede, se dice que es una entidad opcional en la interrelación.

En el modelo ER, un círculo en la línea de conexión entre una entidad y una interrelación indica que la entidad es opcional en la interrelación. La obligatoriedad de una entidad a una interrelación se indica con una línea perpendicular. Si no se consigna ni un círculo ni una línea perpendicular, se considera que la dependencia de existencia es desconocida



Autorrelaciones

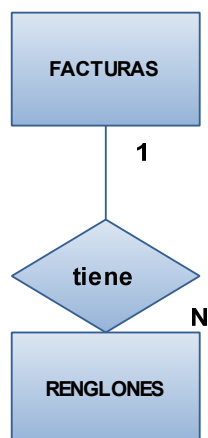
Una autorrelación o interrelación recursiva es una interrelación en la que alguna entidad está asociada más de una vez.



Entidades débiles

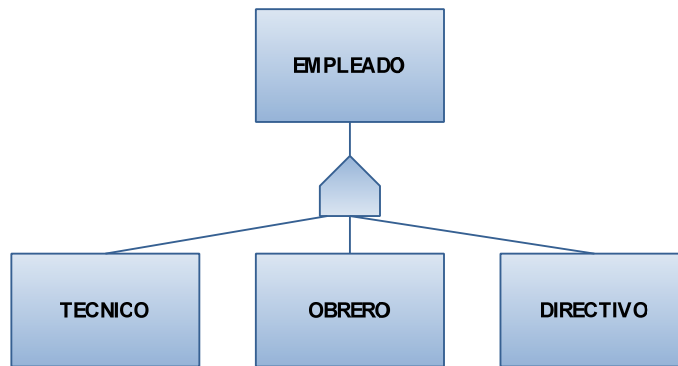
Una entidad débil es una entidad cuyos atributos no la identifican completamente, sino que sólo la identifican de forma parcial.

Esta entidad debe participar en una interrelación que ayuda a identificarla.



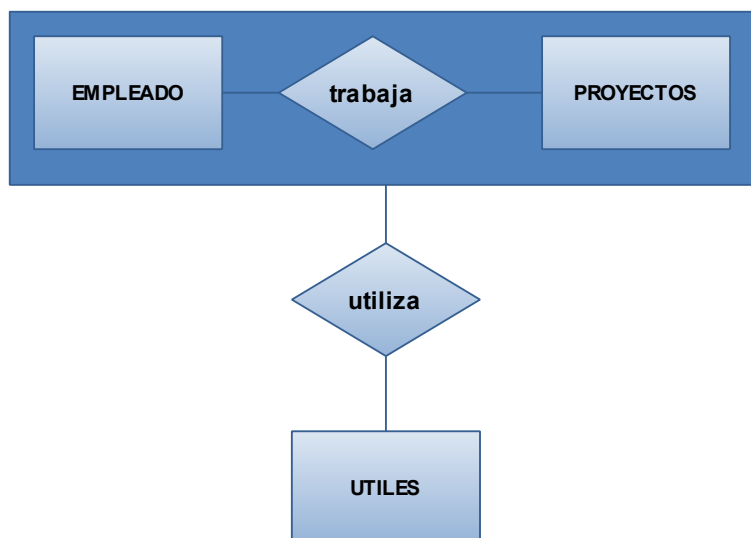
Categorización o Generalización/Especialización

La generalización/especialización permite reflejar el hecho de que hay una entidad general, que denominamos entidad superclase, que se puede especializar en entidades subclase:



Agregación

Cuando dada la realidad es necesario una relación entre una entidad y una relación de otras entidades se utiliza lo que se denomina agregación



Caso de Estudio

Una vez definidos todos los símbolos, vamos a trabajar sobre un caso real que nos acompañará a lo largo de todo el libro, el problema trata sobre una empresa que fabrica válvulas y las arrienda a la industria petroquímica.

La idea es ir descomponiendo el problema en partes pequeñas lo que nos ayudará a comprender todas las posibilidades del modelado de datos.

A continuación agregamos lo que dimos en llamar “**el problemas de las válvulas**”

Problema de las Válvulas

La compañía Valve Petroleum Corp tiene interés en desplegar tecnología para hacer un seguimiento a todas las válvulas que fabrica y que luego renta (alquila) a las empresas petroquímicas del país.

Las válvulas pueden ser del tipo aguja o de retención, las mismas transportan petróleo o químicos indistintamente y deben ser inspeccionadas y mantenidas periódicamente para asegurar su correcto funcionamiento, una falla en la válvula puede causar problemas significativos y la tecnología de seguimiento ayuda a reducir los riesgos.

A los tipos de válvula se los identifica con un código que puede tener 3 letras y 3 números, se conoce también su descripción, el diámetro expresado en mm y con que otro tipo de válvulas son compatibles, una válvula puede ser compatible con cualquier cantidad de otras válvulas. De las válvulas tipo aguja se debe registrar el diámetro que puede ser $\frac{1}{4}$, $\frac{1}{2}$ o $\frac{3}{4}$ y de las de retención la presión máxima que soporta.

Se lleva también un registro de todos los números de serie de cada una de las válvulas que salen de la fábrica, un número de serie pertenece a un único tipo de válvula, cada número de serie asociado a una válvula puede tener una única dimensión, las dimensiones están identificadas por un código, además se conoce una descripción de dicha dimensión.

Las petroquímicas que rentan las válvulas están identificadas con un RUT, se conoce también su nombre, dirección, teléfono y la fecha en que arrendó las válvulas, una petroquímica puede arrendar cualquier cantidad de válvulas y un tipo de válvula puede ser arrendado por cualquier cantidad de petroquímicas.

Además, las agencias gubernamentales requieren que las empresas proporcionen un registro de las inspecciones periódicas de las válvulas arrendadas, se sabe que una válvula arrendada puede tener muchas inspecciones a lo largo de su vida útil, de las inspecciones se sabe su número que la identifica, además se conoce la fecha en que se realizó y una breve descripción de dicha inspección.

Cada agencia gubernamental puede requerir que se hagan distintas inspecciones, a su vez cada inspección puede haber sido solicitada por más de una agencia gubernamental.

En el momento que la agencia gubernamental solicita una inspección, asigna un inspector líder para la misma, ese inspector líder está identificado por su CI, además se conoce su nombre y teléfono celular que debe ser único para cada inspector, cada inspector líder puede tener varias asignaciones de inspección.

Además, Valve Petroleum Corp una vez que tenga marcadas todas sus válvulas, quiere poder ofrecer a los clientes interesados un acceso a los datos en tiempo real sobre cada válvula, así como el seguimiento del trabajo de su personal, por eso le interesa registrar en que lugar está cada válvula arrendada, la empresa tiene identificados a los lugares con un código interno de hasta 5 dígitos, se sabe también la descripción de cada lugar.

A su vez, cada petroquímica tiene sucursales, por lo tanto es importante saber en que lugar de los que la empresa registra está cada sucursal de cada petroquímica.

Comenzaremos a analizar el “problemas de las válvulas” desde el punto de vista de algunas entidades que surgen rápidamente.

Por ejemplo las válvulas, sabemos que es una entidad que además posee ciertos atributos a saber:

Las válvulas están identificadas con un código, se conoce además una descripción y un diámetro

codValv dscValv diamValv



También podemos ver a las Petroquímicas como una entidad con un RUT que la identifica, además se conoce una descripción la dirección y el teléfono

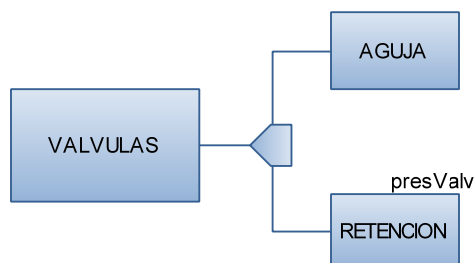
rutPet dscPet dirPet telPet



De la misma forma procedemos a “encontrar” todas las entidades que existen en el problema.

Volvemos otra vez a la entidad VALVULAS, según surge del análisis la entidad válvula está categorizada, ya que el problema dice que una válvula puede ser del tipo AGUJA o del tipo RETENCION, siguiendo esta lógica podemos determinar el modelo:

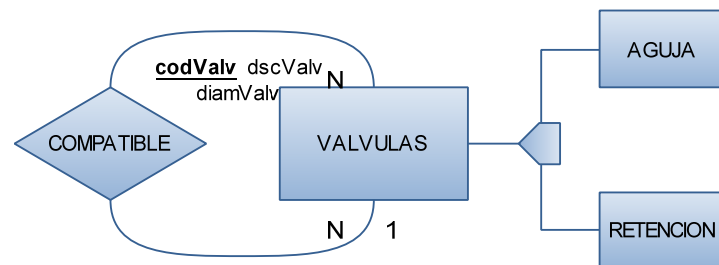
codValv dscValv diamValv



Note que la categoría RETENCION tiene un atributo presVal (presión de la válvula) que no poseen las válvulas del tipo AGUJA.

Un típico caso de autorrelación es el caso de las válvulas que son compatibles con otras válvulas.

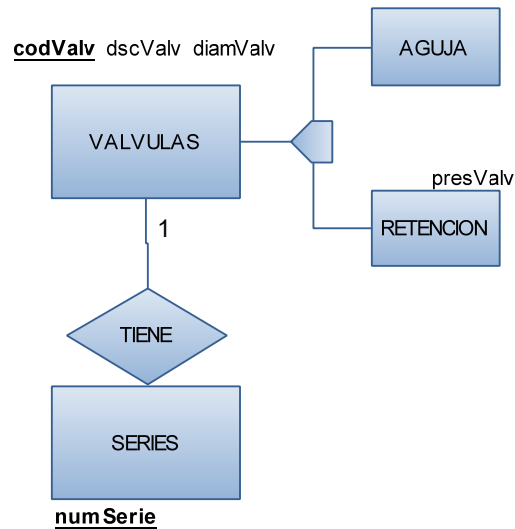
Al ser válvulas una entidad, la misma no se puede poner dos veces en el modelo, por lo tanto al ser una válvula compatible con varias otras válvulas el problema se resuelve aplicando una autorrelación



Veamos ahora el concepto de números de serie, se sabe que un tipo de válvula puede tener muchos números de serie asociados, pero sabemos también que si no existen las válvulas no existen tampoco los números de serie de cada una de ellas.

Por lo tanto surge del análisis que los números de serie de las válvulas dependen de estas últimas para existir, por lo tanto son débiles de la primera.

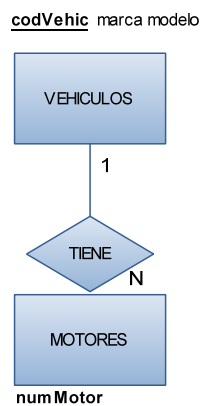
El modelo quedaría de la siguiente manera:



El concepto de números de serie se puede aplicar a otros casos, por ejemplo, suponga que existe una fábrica de vehículos, la misma posee una entidad vehículos, cada tipo de vehículo está identificado con un código, por ejemplo, los FIAT PALIO tienen el código FP01, los FIAT SIENNA el código FS01, los FIAT PANDA el FP01 y así para cada vehículo que es identificado por un código de la fábrica.

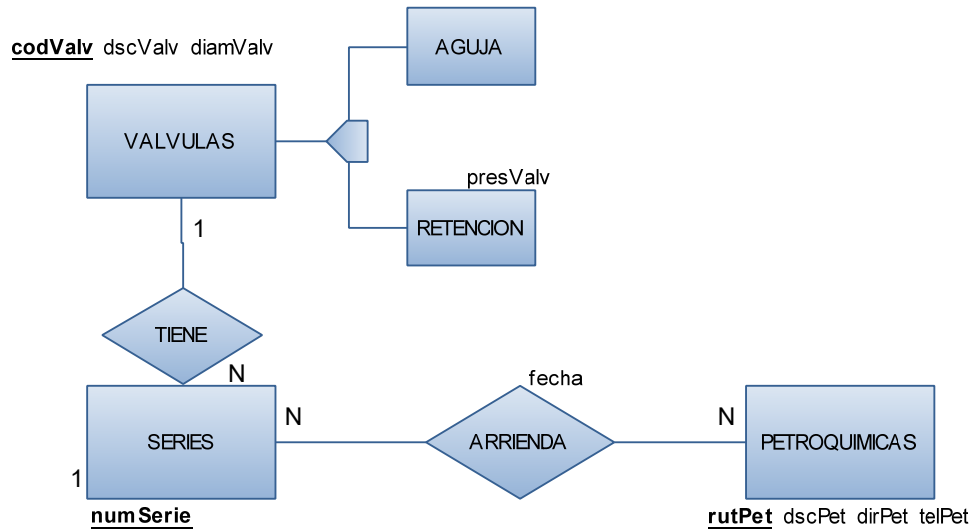
Ahora bien, de cada vehículo hay una innumerable cantidad de unidades, cada unidad está dada a su vez por un número de motor, entonces podemos decir que tenemos una entidad MOTORES y sabemos que existe una relación de 1 a N entre VEHICULOS y MOTORES, es decir, un vehículo puede tener varios números de motores, uno por cada unidad de ese vehículo que se fabrique, pero de por sí solo, los motores no se pueden identificar salvo que estén asociados a un vehículo.

Por lo tanto podemos resumir que MOTORES es débil de VEHICULOS



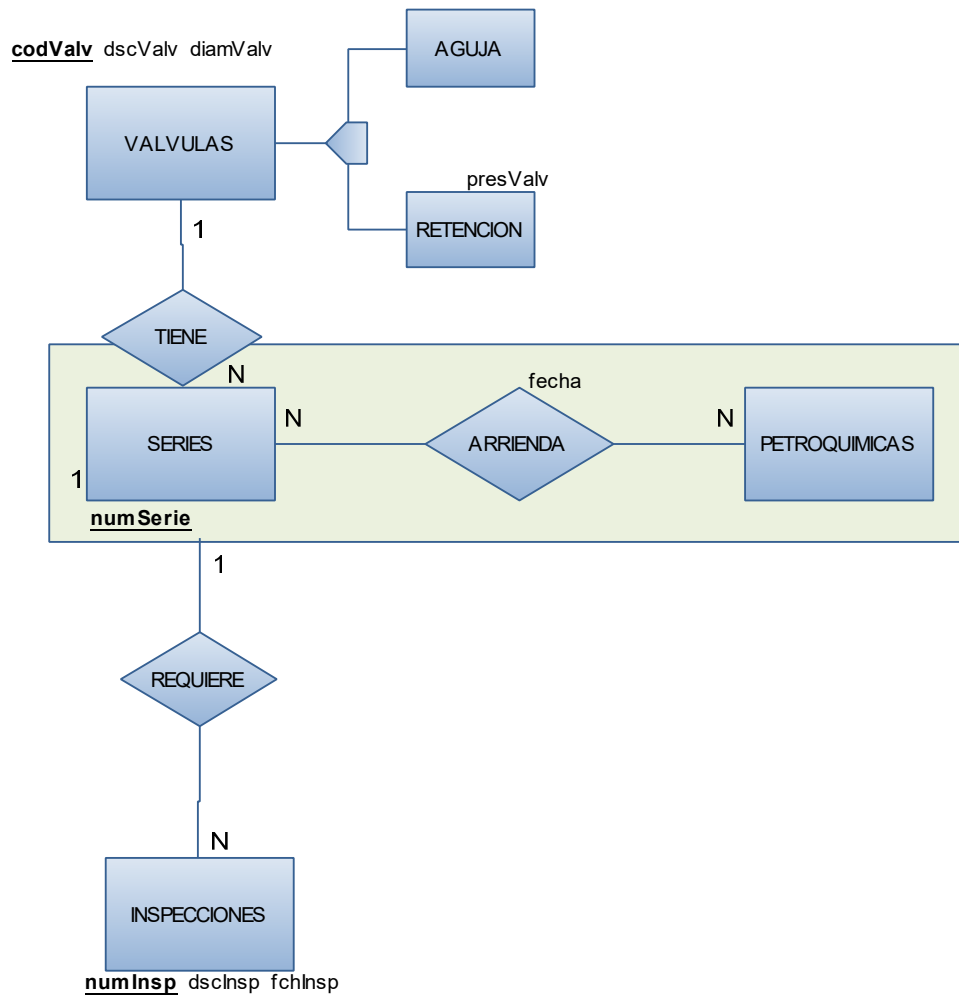
Regresando al problema de las válvulas, si seguimos adelante con el análisis podemos encontrarnos con varias relaciones.

Entre ellas una de las más importantes, sabemos que las Petroquímicas arriendan Válvulas, pero lo lógico de suponer es que arriendan varios números de serie de cada tipo de válvula, por lo tanto podemos determinar una relación ARRIENDA entre Petroquímicas y los números de serie de cada Válvula, se sabe que ese arrendamiento se da en una fecha determinada, por lo tanto asumimos que es un atributo de la relación



Si sabemos, siempre de acuerdo al análisis, que cada Arrendamiento requiere de Inspecciones, y se sabe que cada inspección es para un único Arrendamiento, podemos determinar que existe una relación entre la entidad Inspecciones y la relación Arrenda.

Como según las reglas de modelado, no podemos relacionar relaciones con relaciones, surge que la relación de Inspecciones es con la Agregación entre Petroquímicas y números de serie de las Válvulas

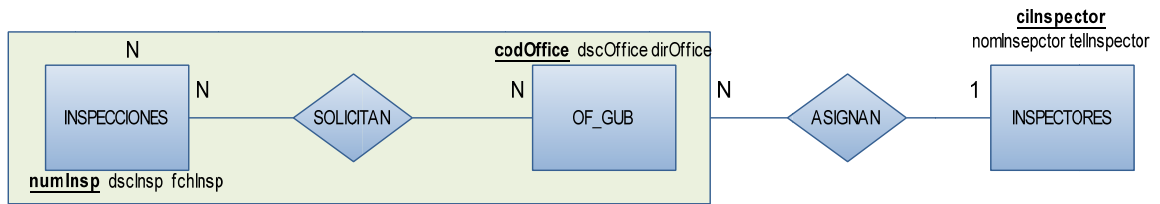


Note que el conector que une la entidad Inspecciones con Arrienda termina en el símbolo de la agregación.

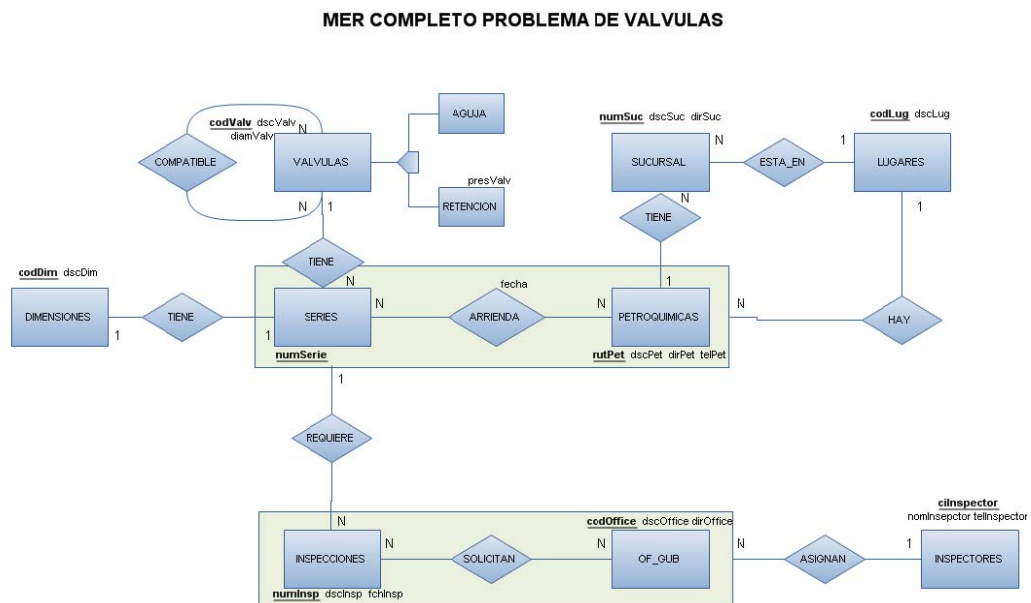
Por lo tanto podemos determinar que una agregación es como una gran entidad que engloba una relación de N a N en su interior.

Otro caso de Agregación encontramos entre las Inspecciones que son Solicitadas por las Oficinas Gubernamentales, sabemos que cada Solicitud tiene asignado un Inspector, y que un Inspector puede ser asignado a varias Solicitudes.

Si sabemos que Solicitud es la relación ente las Oficinas Gubernamentales y las Inspecciones, entonces su relación con Inspectores es a través de una Agregación



De esta manera y haciendo un estudio parte por parte del problema, podemos llegar a un Modelo Entidad Relación completo del problema de las válvulas tal como muestra la figura:



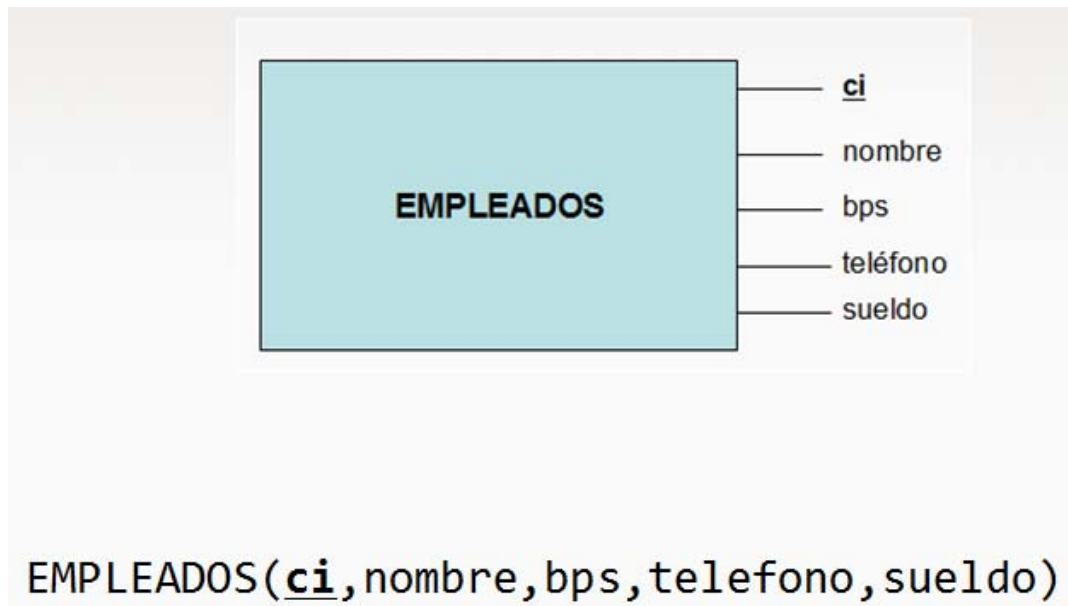
Nota: Solicite al docente una copia más grande del modelo.

Transformación del MER a modelo relacional

En este apartado trataremos el diseño lógico de una base de datos relacional. Partiremos del resultado de la etapa del diseño conceptual expresado mediante el modelo ER y veremos cómo se puede transformar en una estructura de datos del modelo relacional llevado a la tercera forma normal (3NF).

Cada entidad del modelo ER se transforma en una tabla del modelo relacional.

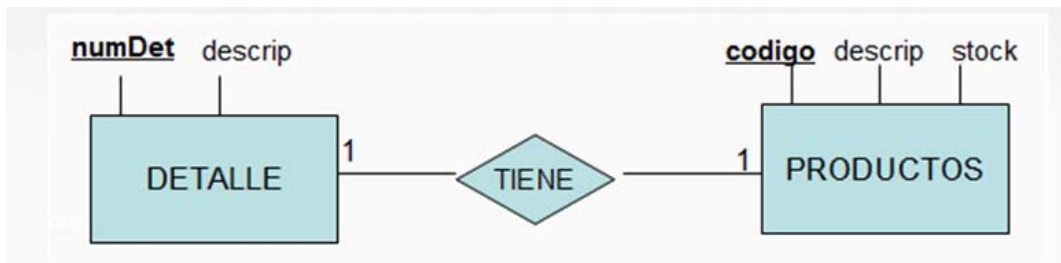
Los atributos de la entidad serán atributos de la tabla y, de forma análoga, la clave primaria de la entidad será la clave primaria de la tabla.



Relaciones de 1:1

Nuestro punto de partida es que las entidades que intervienen en la interrelación 1:1 ya se han transformado en tablas con sus correspondientes atributos.

Entonces sólo será necesario añadir a cualquiera de estas dos tablas una clave foránea que referencie a la otra relación



PRODUCTOS(codigo,descrip,stock)

DETALLE(numDet,descrip,codigo)

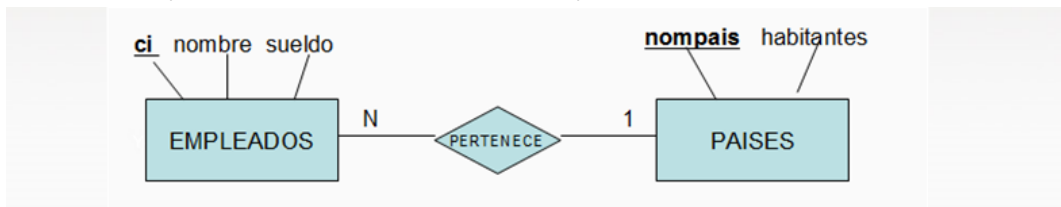
Your company name

Donde código en la tabla Detalle es la **Clave Foránea** de Productos

Relaciones de 1:N

Partimos del hecho de que las entidades que intervienen en la interrelación 1:N ya se han transformado en tablas con sus correspondientes atributos.

En este caso sólo es necesario añadir en la tabla correspondiente a la entidad del lado N, una clave foránea que referencie la otra tabla (su clave primaria)



PAISES(nompais,habitantes)

EMPLEADOS(ci,nombre, sueldo,nompais)

Your company name

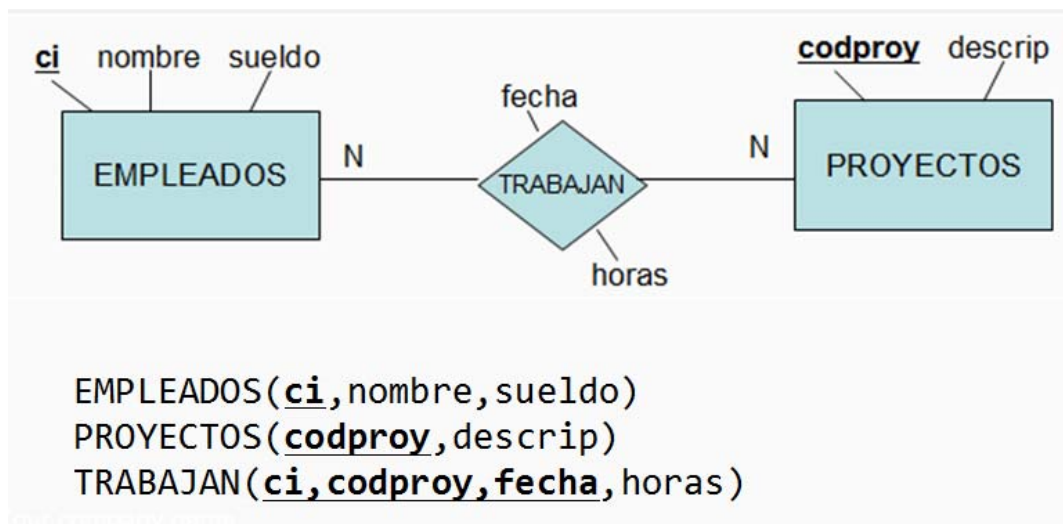
Donde nompais en la tabla Empleados es la Clave Foránea de Paises

Relaciones de N:N

Una interrelación N:N se transforma en una tabla.

Su clave primaria estará formada por los atributos de la clave primaria de las dos entidades interrelacionadas.

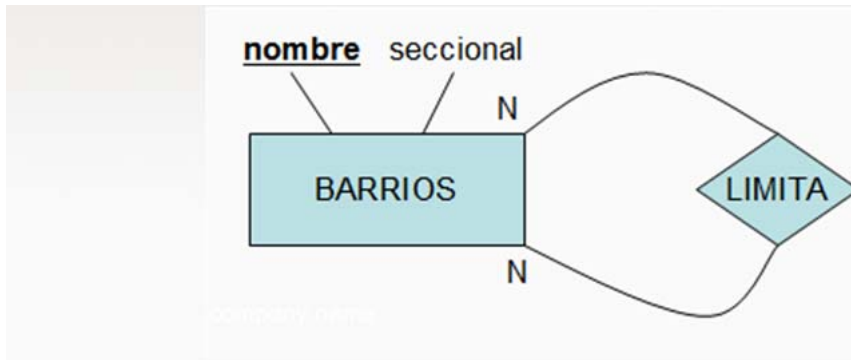
Los atributos de la interrelación serán atributos de la nueva relación.



Autorrelaciones o relaciones recursivas

Las transformaciones de las interrelaciones recursivas son similares a las que hemos visto para el resto de las interrelaciones.

De este modo, si una interrelación recursiva tiene conectividad 1:1 o 1:N, da lugar a una clave foránea, y si tiene conectividad N:N, origina una nueva tabla.



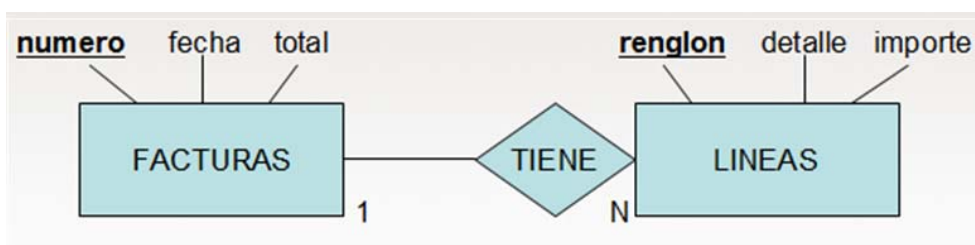
BARRIOS(nombre, seccional)

LIMITA(nombre1, nombre2)

Entidades Débiles

Las entidades débiles se traducen al modelo relacional igual que el resto de entidades, con una pequeña diferencia. Estas entidades siempre están en el lado N de una interrelación 1:N que completa su identificación.

Así pues, la clave foránea originada por esta interrelación 1:N debe formar parte de la clave primaria de la relación correspondiente a la entidad débil.



FACTURAS(numero, fecha, total)

LINEAS(renglon, numero, detalle, importe)

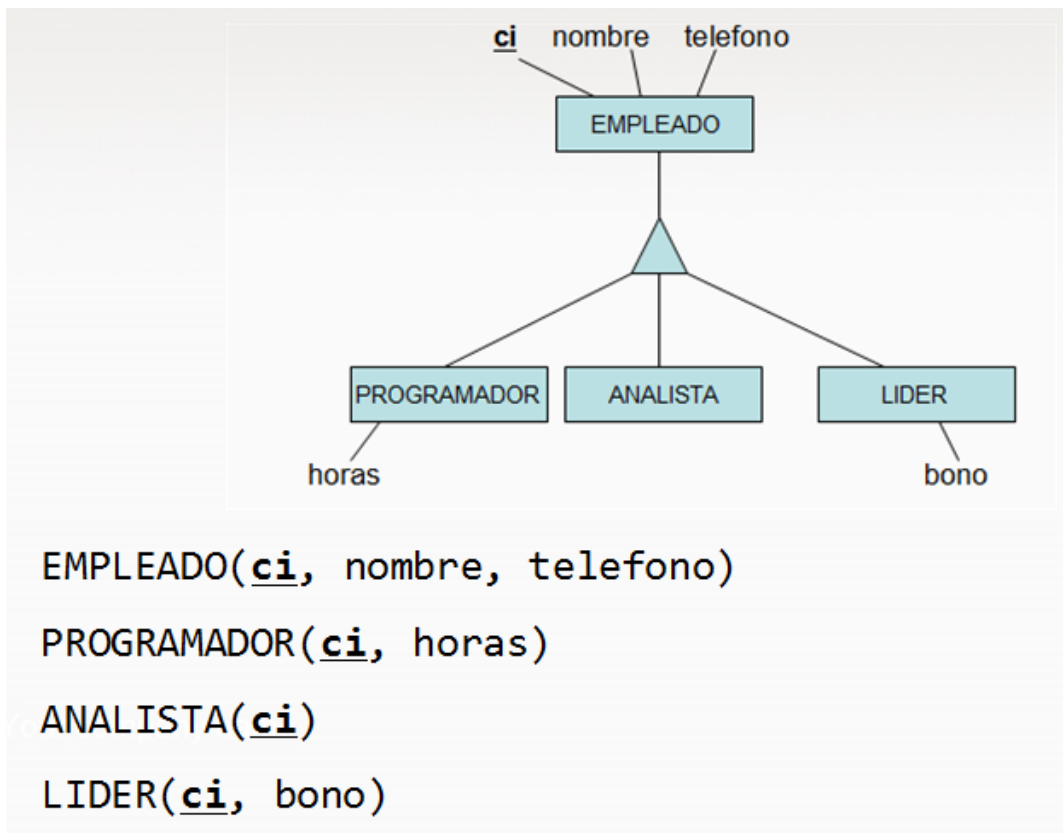
Categorizaciones

Cada una de las entidades superclase y subclase que forman parte de una generalización/especialización se transforma en una tabla:

La tabla de la entidad superclase tiene como clave primaria la clave de la entidad superclase y contiene todos los atributos comunes.

Las tablas de las entidades subclase tienen como clave primaria la clave de la entidad superclase y contienen los atributos específicos de la subclase.

NOTA: Dependiendo del problema planteado este modelo al ser transformado se puede simplificar agregando redundancia controlada como veremos más adelante.

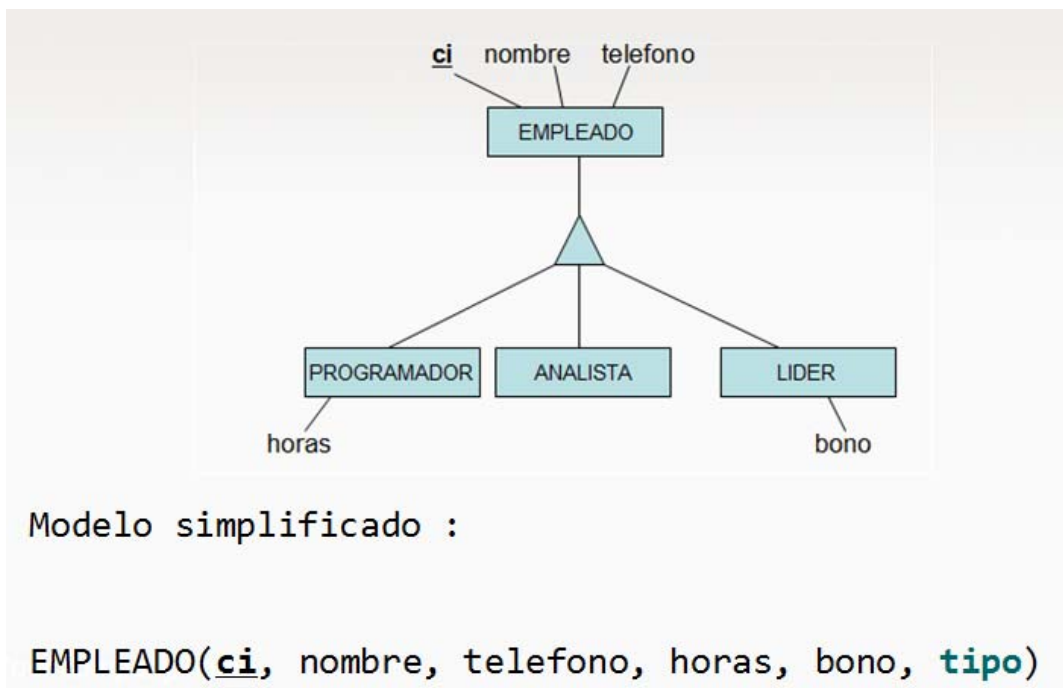


El hecho de simplificar el modelo en los casos de las categorizaciones nos lleva a tener que realizar algunos controles utilizando herramientas de definición de reglas del negocio (por ejemplo disparadores, que se verá más adelante)

Para simplificar el pasaje a tablas de este modelo categorizado, tenemos que crear una única tabla para la super-clase y agregarle todos los atributos de las sub-clases, también debemos crear un atributo, que podemos llamar TIPO, donde se definirá a que tipo de sub-clase nos estamos refiriendo, es decir, si hablamos de Programadores podemos inferir que es del tipo 'P', asimismo, para hablar de Analistas tenemos el tipo 'A' y de Lideres el tipo 'L'.

Cualquiera sea estos casos, sabemos que para los Empleados del tipo A, los campos horas y bono (pertenecientes a otras sub-clases) estarán en nulo, por lo tanto deberá ser controlado por alguna regla del negocio lo que permita mantener la integridad de los datos.

Modelo simplificado:



Visto los ejemplos anteriores, volvemos al “problema de las válvulas” para hacer el pasaje a tablas de dicho modelo.

codValv dscValv diamValv



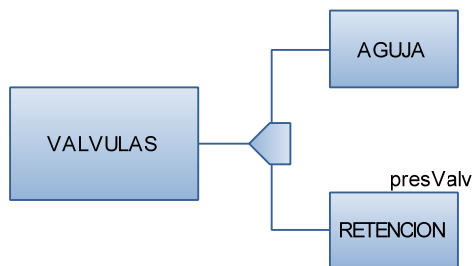
VALVULAS(**codValv**, dscValv, diamValv)

rutPet dscPet dirPet telPet



PETROQUIMICAS(**rutPet**, dscPet, dirPet, telPet)

codValv dscValv diamValv



VALVULAS(**codValv**, dscValv, diamValv)

AGUJA(**codValv**)

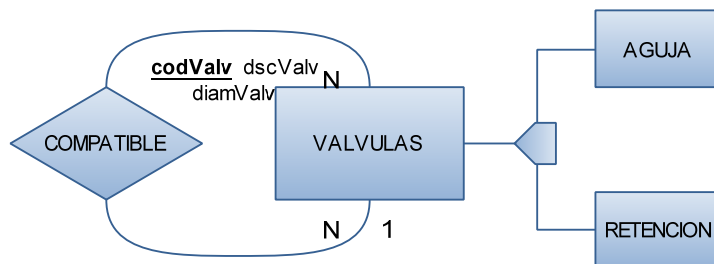
RETENCION(**codValv**, presValv)

Modelo simplificado:

VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

Prestar especial atención en el caso de las válvulas del tipo 'A' el campo presValv estará en nulo y se deberá controlar con alguna herramienta de definición de reglas de negocio.

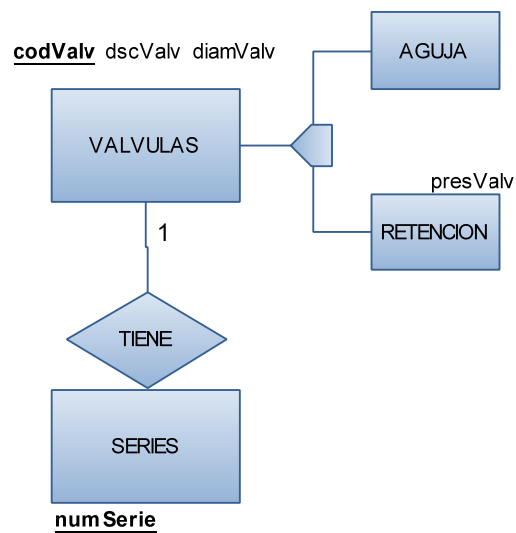
Autorrelación



VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

COMPATIBLE(codValv1, codValv2)

Entidad débil

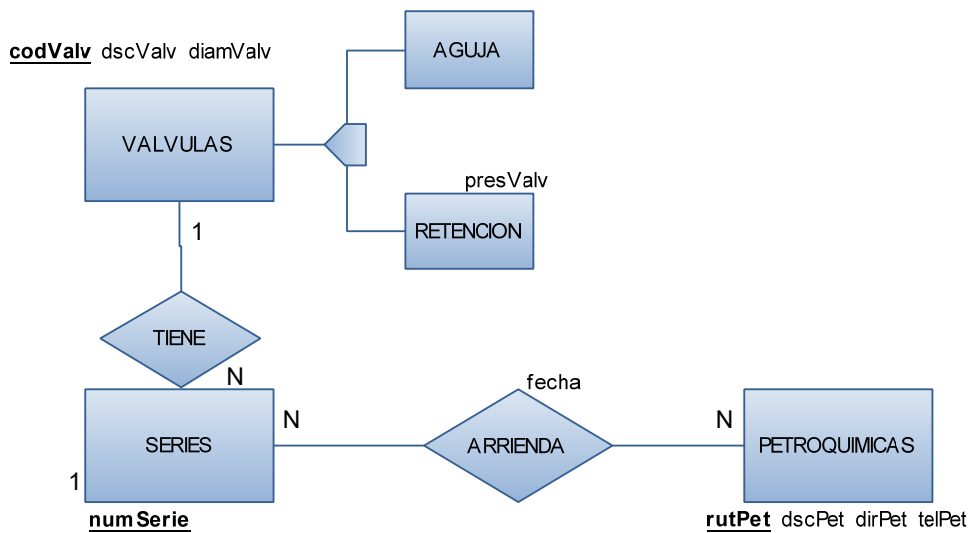


La normalización de la entidad débil determina que la clave de dicha entidad es su propia clave mas la clave de la entidad fuerte (en este caso VALVULAS)

VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

SERIES(numSerie, codValv)

Relaciones de N:N



VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

SERIES(numSerie, codValv)

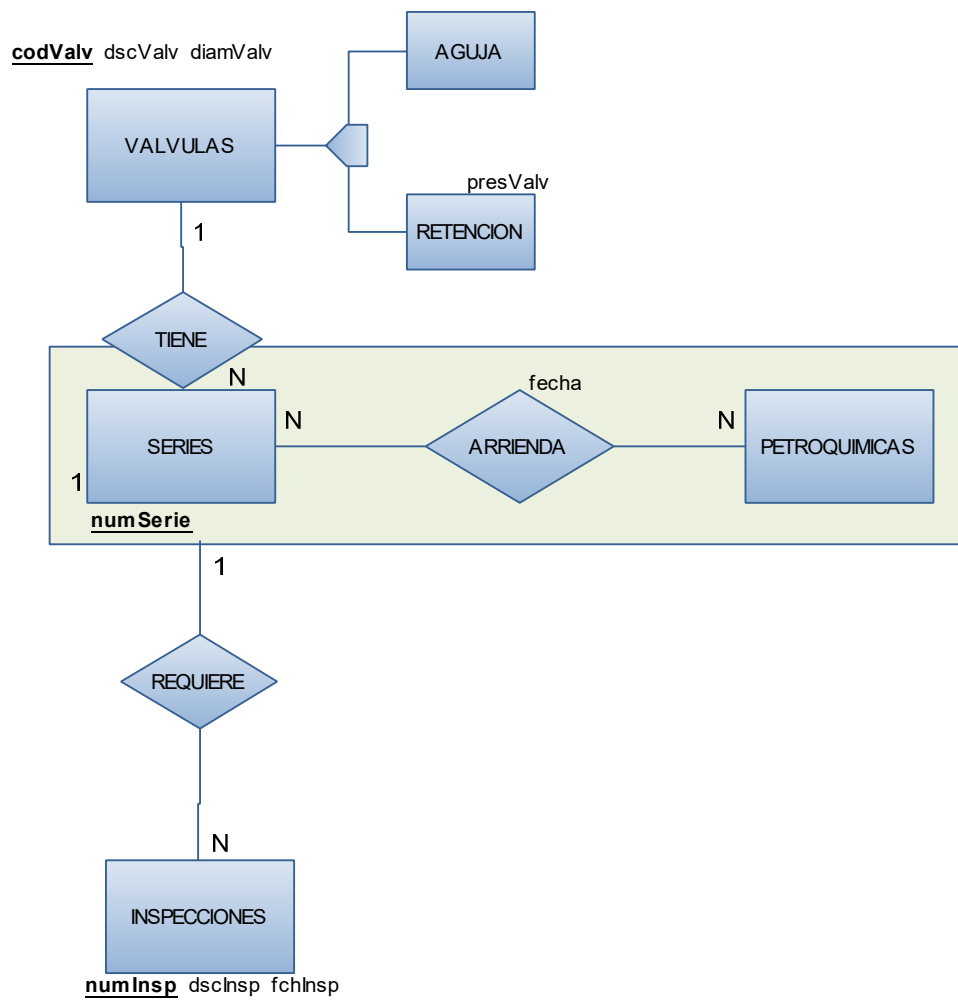
PETROQUIMICAS(rutPet, dscPet, dirPet, telPet)

ARRIENDA(numSerie, codValv, rutPet, fecha)

Note un tema muy importante, la clave de la tabla ARRIENDA incorpora también el atributo fecha, esto es así por definición de clave, se sabe que una clave permite identificar como único un registro dentro de una tabla, por lo tanto luego de aplicar el pasaje a tercera forma normal (3NF) es necesario analizar si se cumple esta premisa, de no cumplirse, se debe buscar otro atributo para agregar a la clave.

Agregación

Para este caso, sabemos que tenemos que manejar la relación encerrada en la agregación como si fuera una gran entidad, por lo tanto en el ejemplo, a la tabla Inspecciones se le agrega como calve foránea la clase de Arrienda, ya que Inspecciones tiene la cardinalidad N en la relación.



VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

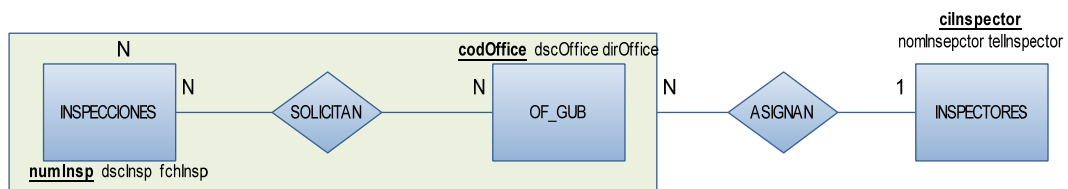
SERIES(numSerie, codValv)

PETROQUIMICAS(rutPet, dscPet, dirPet, telPet)

ARRIENDA(numSerie, codValv, rutPet, fecha)

INSPECCIONES(numInsp, dscInsp, fchInsp, *numSerie*, *codValv*, *rutPet*, *fecha*)

Otro caso de Agregación



INSPECCIONES(numInsp, dscInsp, fchInsp, *numSerie*, *codValv*, *rutPet*, *fecha*)

OF_GUB(codOffice, dscOffice, dirOffice)

INSPECTORES(ciInsp, nomInsp, telInsp)

SOLICITAN(numInsp, codOffice, *ciInsp*)

A continuación mostramos el pasaje a tablas en Tercera Forma Normal (3NF) del problema completo.

DIMENSIONES(codDim, dscDim)

VALVULAS(codValv, dscValv, diamValv, tipo, presValv)

SERIES(numSerie, codValv, *codDim*)

PETROQUIMICAS(rutPet, dscPet, dirPet, telPet)

INSPECCIONES(numInsp, dscInsp, fchInsp, *numSerie*, *codValv*, *rutPet*, *fecha*)

OF_GUB(codOffice, dscOffice, dirOffice)

INSPECTORES(ciInsp, nomInsp, telInsp)

SOLICITAN(numInsp, codOffice, *ciInsp*)

LUGARES(codLug, dscLug)

SUCURSAL(numSuc, rutPet, dscSuc, dirSuc, *codLug*)

ARRIENDA(numSerie, codValv, rutPet, fecha, *codLug*)

COMPATIBLE(codValv1, codValv2)

Módulo 3

El lenguaje de definición de datos (DDL)	68
Creación de Tablas	69
Restricciones de Integridad	73
Otras restricciones (CONSTRAINTS)	77

Lenguaje de Definición de Datos (DDL)

Para poder trabajar con bases de datos relacionales, lo primero que tenemos que hacer es definir las. Veremos los órdenes del estándar SQL92/2003 para crear y borrar una base de datos relacional y para insertar, borrar y modificar las diferentes tablas que la componen.

Para crear una base de datos en el SGBD se utiliza:

```
USE master; /* Base de datos maestra de SQLServer */  
GO  
CREATE DATABASE miBaseDeDatos;  
GO
```

Nota: ejemplo aplicable a SQLServer

Como ya hemos visto, la estructura de almacenamiento de los datos del modelo relacional son las tablas.

Para crear una tabla, es necesario utilizar la sentencia CREATE TABLE

```
CREATE TABLE nombre_tabla  
    (  
        definición_columna  
        [, definición_columna...]  
        [, restricciones_tabla]  
    );
```

Definición de columna:

```
nombre columna {tipo datos|dominio} [def defecto] [restric col]
```

El proceso que hay que seguir para crear una tabla es el siguiente:

1. Lo primero que tenemos que hacer es decidir qué nombre queremos poner a la tabla (nombre_tabla).
2. Después, iremos dando el nombre de cada uno de los atributos que formarán las columnas de la tabla (nombre_columna).
3. A cada una de las columnas le asignaremos un tipo de datos predefinido o bien un dominio definido por el usuario. También podremos dar definiciones por defecto y restricciones de columna.
4. Una vez definidas las columnas, sólo nos quedará dar las restricciones de tabla.

Tipos de dato más utilizados dentro del estándar SQL:

NUMERIC

CHARACTER

VARCHAR

DATE

Alcance:

El alcance determina que tamaño de cada tipo de datos, en el caso del tipo de dato DATE o DATETIME no es necesario definir dicho alcance

Por ejemplo:

NUMERIC(5) /* Número entero de largo 5 */

NUMERIC(12,2) /* Número de largo 12 con 2 decimales */

CHARACTER(10) /* Hasta 10 caracteres alfanuméricos reserva de espacio */

VARCHAR(30) /* Hasta 30 caracteres alfanuméricos sin reserva de espacio */

```
| CREATE TABLE empleados (cedula NUMERIC(8),
                             nombre VARCHAR(30),
                             fnac DATE,
                             sueldo NUMERIC(12,2),
                             - tipo CHAR(1));
```

Modificar estructuras:

Añadir una columna a una tabla:

```
ALTER TABLE T_PEDIDOS ADD TEXTOPEDIDO Varchar(35);
```

Cambiar el tamaño de una columna en una tabla:

```
ALTER TABLE T_PEDIDOS ALTER COLUMN TEXTOPEDIDO Varchar(135);
```

Hacer NOT NULL una columna en una tabla:

```
ALTER TABLE T_PEDIDOS ALTER COLUMN TEXTOPEDIDO NOT NULL;
```

Eliminar una columna a una tabla:

```
ALTER TABLE T_PEDIDOS DROP COLUMN TEXTOPEDIDO;
```

Valor por defecto de una columna:

```
ALTER TABLE T_PEDIDOS ALTER COLUMN TEXTOPEDIDO Varchar(135) DEFAULT 'ABC...';
```

Añade dos columnas:

```
ALTER TABLE T_PEDIDOS
    ADD (SO_PEDIDOS_ID INT, TEXTOPEDIDO Varchar(135));
```

Por ejemplo, supongamos que en el problema de las válvulas queremos agregar el precio en dólares de cada arrendamiento

```
ALTER TABLE arriendo ADD precio NUMERIC(12,2)
```

Agregamos el porcentaje de ganancia que cada modelo de válvula tiene:

```
ALTER TABLE valvulas ADD porcValv NUMERIC(4,2)
```

Agregamos la fecha en que finaliza un arrendamiento

```
ALTER TABLE arriendo ADD finFch DATE
```

Restricciones o Constraints

¿Qué es un Constraints?

El SGBD utiliza constraints para prevenir el registro de datos no válidos a las tablas.

Se pueden utilizar los constraints para lo siguiente:

- Implementar o imponer reglas en los datos de una tabla cuando una fila es insertada, modificada o borrada de la tabla, el constraint se debe cumplir para que la operación se realice.
- Previene la eliminación de una tabla si existen dependencias con otras

Una vez hemos dado un nombre, hemos definido una tabla y hemos impuesto ciertas restricciones para cada una de las columnas, podemos aplicar restricciones sobre toda la tabla, que siempre se deberán cumplir

Las restricciones más utilizadas son:

PRIMARY KEY

FOREIGN KEY

NOT NULL

UNIQUE

CHECK

Constraint	Descripcion
NOT NULL	Especifica que la columna no puede tener un valor nulo
PRIMARY KEY	Identifica de manera única cada fila de una tabla
FOREIGN KEY	Establece una relación entre una columna de la tabla y otra de la tabla referenciada
UNIQUE	La columna no puede tener valores repetidos. Es una clave alternativa.
CHECK	La columna debe cumplir la condición establecida

Ejemplo:

```
CREATE TABLE empleados (cedula NUMERIC(8),
                           nombre VARCHAR(30) NOT NULL,
                           fnac DATE,
                           sueldo NUMERIC(12,2),
                           tipo CHAR(1),
                           PRIMARY KEY(cedula),
                           CHECK (sueldo > 5000)
                           );
```

```
CREATE TABLE ciudades (nomCiudad CHAR(20),
                        habitantes NUMERIC(10),
                        fch_fundacion DATE,
                        PRIMARY KEY(nomCiudad));
```

```
CREATE TABLE empleados (cedula NUMERIC(8),
                           nombre VARCHAR(30) NOT NULL,
                           fnac DATE,
                           sueldo NUMERIC(12,2),
                           tipo CHAR(1),
                           nomCiudad CHAR(20),
                           PRIMARY KEY(cedula),
                           CHECK (sueldo > 5000),
                           FOREIGN KEY (nomCiudad) REFERENCES ciudades(nomCiudad)
                           );
```

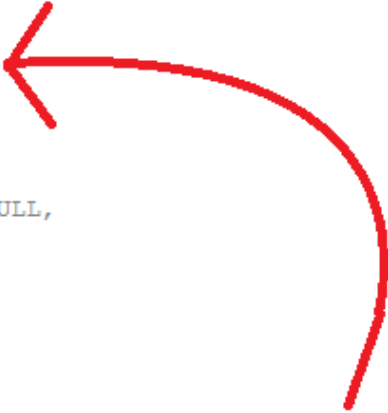


```

CREATE TABLE ciudades(nomCiudad CHAR(20),
                        habitantes NUMERIC(10),
                        fch_fundacion DATE,
                        PRIMARY KEY(nomCiudad));

CREATE TABLE empleados(cedula NUMERIC(8),
                        nombre VARCHAR(30) NOT NULL,
                        fnac DATE,
                        sueldo NUMERIC(12,2),
                        tipo CHAR(1),
                        nomCiudad CHAR(20),
                        PRIMARY KEY(cedula),
                        CHECK (sueldo > 5000),
                        FOREIGN KEY (nomCiudad) REFERENCES ciudades(nomCiudad)
                        );

```



NOT NULL

El constraint NOT NULL se asegura de que las columnas no contengan valores nulos.

Las columnas sin un constraint NOT NULL pueden contener valores nulos por defecto.

El constraint NOT NULL puede ser especificado solamente a nivel de columnas y no a nivel de tabla.

UNIQUE

Un constraint UNIQUE requiere que todos los valores en una columna o conjunto de columnas sean únicos, esto es, dos filas de una tabla no pueden tener valores duplicados en la columna o conjunto de columnas especificadas.

Un constraint UNIQUE puede ser definido a nivel de columna o tabla.

CREATE TABLE empleados

```

(cedula NUMERIC(8),
 nombre VARCHAR(30),
 email VARCHAR(25),
 sueldo NUMERIC(12,2),
 CONSTRAINT UK_email UNIQUE(email) )

```

En el ejemplo anterior se aplica un constraint UNIQUE a la columna EMAIL de la tabla empleados. El nombre del constraint es UK_email

PRIMARY KEY

Un constraint PRIMARY KEY crea una llave primaria para la tabla.

Solo una llave primaria puede ser creada por cada tabla. El constraint PRIMARY KEY es una columna o conjunto de columnas que identifica de forma única cada fila de una tabla.

Estos constraints obligan valores únicos para la columna o combinación de columnas y aseguran que estas columnas no puedan contener valores nulos.

Los constraints PRIMARY KEY pueden ser definidos a nivel de columna o a nivel de tabla.

Una llave primaria compuesta es creada usando la definición a nivel de tabla.

Una tabla puede tener solo un constraint PRIMARY KEY pero puede tener diversos constraints UNIQUE.

FOREIGN KEY

Un Foreign Key o llave foránea, es un constraint de integridad referencial que designa a una columna o combinación de columnas como una llave foránea estableciendo una relación entre una llave primaria o llave única en la misma tabla o en una tabla diferente.

CHECK

El constraint CHECK define una condición que para cada fila debe satisfacerse.

Una columna puede tener múltiples constraints CHECK las cuales pueden referenciarse en la definición de la columna.

No se tienen límites en el número de constraints CHECK cuando se define una columna.

Añadir CONSTRAINTS a una tabla existente

Se pueden añadir constraints a tablas existentes con el uso de la sentencia ALTER TABLE y la cláusula ADD.

```
ALTER TABLE empleados ADD CONSTRAINT CHECK (sueldo < 10000);
```

```
ALTER TABLE mitabla ADD CONSTRAINT UNIQUE (columna1,columna2);
```

Campos IDENTITY (SQLServer)

Un campo numérico puede tener un atributo extra "identity".

Genera valores secuenciales que por defecto se inician en 1 y se incrementan en 1.

Se utiliza generalmente en campos correspondientes a códigos de identificación para generar valores únicos para cada nuevo registro que se inserta.

Sólo puede haber un campo "identity" por tabla.

Cuando un campo tiene el atributo "identity" no se puede ingresar valor para él, se inserta automáticamente tomando el último valor como referencia.

Sintaxis:

`IDENTITY [(seed , increment)]`

```
CREATE TABLE Productos (prodCod INTEGER PRIMARY KEY
                        IDENTITY(50,1),
                        nombre VARCHAR(50),
                        peso DECIMAL(9,2),
                        descripcion VARCHAR(200))
```

Volvemos al “problema de las válvulas”, con el pasaje a tablas y normalización de la página 66 estamos en condiciones de definir las estructuras que van a soportar los datos del problema.

En principio vamos a definir solo las restricciones de clave primaria y clave foránea, luego vamos a utilizar ALTER para agregar el resto de las restricciones

```

CREATE DATABASE BDVALVULAS;

CREATE TABLE valvulas(codValv CHARACTER(6),
                      dscValv VARCHAR(30),
                      diamValv NUMERIC(10,2),
                      CONSTRAINT PK_Val PRIMARY KEY(codValv))

CREATE TABLE aguja(codValv CHARACTER(6),
                    CONSTRAINT PK_Agu PRIMARY KEY(codValv))

CREATE TABLE retencion(codValv CHARACTER(6),
                        presValv NUMERIC(10,2),
                        CONSTRAINT PK_Ret PRIMARY KEY(codValv))

CREATE TABLE lugares(codLug NUMERIC(6) IDENTITY(1,1),
                      dscLug VARCHAR(30),
                      CONSTRAINT PK_Lug PRIMARY KEY(codLug))

CREATE TABLE inspectores(ciInsp NUMERIC(8),
                          nomInsp VARCHAR(30),
                          telInsp VARCHAR(30),
                          CONSTRAINT PK_Inspector PRIMARY KEY(ciInsp))

CREATE TABLE dimensiones(codDim NUMERIC(6) IDENTITY(1,1),
                          dscDim VARCHAR(30),
                          CONSTRAINT PK_Dim PRIMARY KEY(codDim))

CREATE TABLE series(numSerie CHAR(20),
                     codValv CHARACTER(6),
                     codDim NUMERIC(6),
                     CONSTRAINT PK_Serie PRIMARY KEY(numSerie,codValv),
                     CONSTRAINT FK_Dim FOREIGN KEY(codDim) REFERENCES dimensiones(codDim),
                     CONSTRAINT FK_SerVal FOREIGN KEY(codValv) REFERENCES valvulas(codValv))

CREATE TABLE compatible(codValv CHARACTER(6),
                         compValv CHARACTER(6),
                         CONSTRAINT PK_Com PRIMARY KEY(codValv,compValv),
                         CONSTRAINT FK_Valv1 FOREIGN KEY(codValv) REFERENCES valvulas(codValv),
                         CONSTRAINT FK_Valv2 FOREIGN KEY(compValv) REFERENCES valvulas(codValv))

CREATE TABLE petroquimica(rutPet CHAR(13),
                          dscPet VARCHAR(30),
                          dirPet VARCHAR(30),
                          telPet VARCHAR(30),
                          CONSTRAINT PK_Pet PRIMARY KEY(rutPet))

]CREATE TABLE arrienda(numSerie CHAR(20),
                       codValv CHARACTER(6),
                       rutPet CHAR(13),
                       fchArr DATE,
                       codLug NUMERIC(6),
                       CONSTRAINT PK_Arr PRIMARY KEY(numSerie,codValv,rutPet,fchArr),
                       CONSTRAINT FK_ArrSer FOREIGN KEY(numSerie,codValv) REFERENCES series(numserie,codValv),
                       CONSTRAINT FK_ArrPet FOREIGN KEY(rutPet) REFERENCES petroquimica(rutPet),
                       CONSTRAINT FK_ArrLug FOREIGN KEY(codLug) REFERENCES lugares(codLug))

```

En base a la letra del “problema de las válvulas” podemos determinar que son necesarias algunas constraints, por ejemplo, se sabe que el teléfono celular de los inspectores debe ser único en la tabla, por lo tanto podemos agregar el siguiente comando DDL:

```
ALTER TABLE inspectores ADD CONSTRAINT UNIQUE (telInsp);
```

También podemos simplificar el modelo en el caso de las válvulas, para ello tenemos que borrar las tablas creadas como categorías, luego modificar la estructura de la tabla VALVULAS y agregar las constraints necesarias para controlar el tipo.

```
DROP TABLE aguja;
```

```
DROP TABLE retencion;
```

```
ALTER TABLE valvulas ADD presValv NUMERIC(10,2);
```

```
ALTER TABLE valvulas ADD tipoValv CHARACTER(1);
```

```
ALTER TABLE valvulas ADD
```

```
    CONSTRAINT CK_TipValv CHECK (tipoValv IN ('A','R'));
```

Definimos en SQL Server los comandos DDL completos para el problema de las válvulas

```
CREATE TABLE valvulas(codValv CHARACTER(6),
    dscValv VARCHAR(30),
    diamValv NUMERIC(10,2),
    tipoValv CHARACTER(1),
    presValv NUMERIC(10,2),
    CONSTRAINT PK_Val PRIMARY KEY(codValv),
    CONSTRAINT CK_TipValv CHECK(tipoValv IN ('A','R')))
```

```
CREATE TABLE lugares(codLug NUMERIC(6) IDENTITY(1,1),
    dscLug VARCHAR(30),
    CONSTRAINT PK_Lug PRIMARY KEY(codLug))
```

```
CREATE TABLE inspectores(cilInsp NUMERIC(8),
    nomInsp VARCHAR(30),
    tellInsp VARCHAR(30),
    CONSTRAINT PK_Inspector PRIMARY KEY(cilInsp))
```

```
CREATE TABLE dimensiones(codDim NUMERIC(6) IDENTITY(1,1),
    dscDim VARCHAR(30),
    CONSTRAINT PK_Dim PRIMARY KEY(codDim))
```

```
CREATE TABLE series(numSerie CHAR(20),
    codValv CHARACTER(6),
    codDim NUMERIC(6),
    CONSTRAINT PK_Serie PRIMARY KEY(numSerie,codValv),
    CONSTRAINT FK_SerieDim
        FOREIGN KEY(codDim) REFERENCES dimensiones(codDim),
    CONSTRAINT FK_SerieVal
        FOREIGN KEY(codValv) REFERENCES valvulas(codValv))
```

```
CREATE TABLE compatible(codValv CHARACTER(6),
    CompValv CHARACTER(6),
    CONSTRAINT PK_Com PRIMARY KEY(codValv,compValv),
    CONSTRAINT FK_ComValv1
        FOREIGN KEY(codValv) REFERENCES valvulas(codValv),
    CONSTRAINT FK_ComValv2
        FOREIGN KEY(compValv) REFERENCES valvulas(codValv))
```

```
CREATE TABLE petroquimica(rutPet CHARACTER(13),
    dscPet VARCHAR(30),
    dirPet VARCHAR(30),
    telPet VARCHAR(30),
    CONSTRAINT PK_Pet PRIMARY KEY(rutPet))
```

```
CREATE TABLE arrienda(numSerie CHAR(20),
    codValv CHARACTER(6),
    rutPet CHARACTER(13),
    fchArr DATE,
    codLug NUMERIC(6),
    CONSTRAINT PK_Arr PRIMARY KEY(numSerie,codValv,rutPet,fchArr),
```

```

CONSTRAINT FK_ArrSer
FOREIGN KEY(numSerie,codValv) REFERENCES series(numserie,codValv),
CONSTRAINT FK_ArrPet
FOREIGN KEY(rutPet) REFERENCES petroquimica(rutPet),
CONSTRAINT FK_ArrLug
FOREIGN KEY(codLug) REFERENCES lugares(codLug))

```

```

CREATE TABLE inspecciones(numInsp NUMERIC(6) IDENTITY(1,1),
    dscInsp VARCHAR(30) not null,
    fchInsp DATETIME,
    numSerie CHARACTER(20),
    codValv CHARACTER(6),
    rutPet CHARACTER(13),
    fecha DATETIME,
    CONSTRAINT PK_Insp PRIMARY KEY(numInsp),
    CONSTRAINT FK_InspSerie
    FOREIGN KEY(numSerie,codValv)
    REFERENCES SERIES(numSerie,codValv),
    CONSTRAINT FK_InspPetro
    FOREIGN KEY(rutPet) REFERENCES petroquimica (rutPet))

```

```

CREATE TABLE of_gub(codOffice NUMERIC(6) IDENTITY(1,1),
    dscOffice VARCHAR(30) not null,
    dirOffice VARCHAR(30) not null,
    CONSTRAINT PK_Ofic PRIMARY KEY(codOffice))

```

```

CREATE TABLE solicitan(numInsp NUMERIC(6),
    codOffice NUMERIC(6),
    cilnsp NUMERIC(8),
    CONSTRAINT PK_Solic PRIMARY KEY(numInsp,codOffice),
    CONSTRAINT FK_SolNum
    FOREIGN KEY (numInsp) REFERENCES inspecciones(numInsp),
    CONSTRAINT FK_SolOfi
    FOREIGN KEY (codOffice) REFERENCES of_gub(codOffice),
    CONSTRAINT FK_SolCi
    FOREIGN KEY (cilnsp) REFERENCES inspectores(cilnsp) )

```

```

CREATE TABLE sucursal(numSuc NUMERIC(3),
    rutPet CHARACTER(13),
    dscSuc VARCHAR(30) not null,
    dirSuc VARCHAR(30) not null,
    codLug NUMERIC(6),
    CONSTRAINT PK_Suc PRIMARY KEY(numsuc,rutPet),
    CONSTRAINT FK_SucPet
    FOREIGN KEY(rutPet) REFERENCES petroquimica (rutPet),
    CONSTRAINT FK_SucLug
    FOREIGN KEY(codLug) REFERENCES lugares(codLug))

```

```
ALTER TABLE arriendo ADD precio NUMERIC(12,2)
```

```
ALTER TABLE valvulas ADD porcValv NUMERIC(4,2)
```

```
ALTER TABLE arriendo ADD finFch DATE
```


Módulo 4

El lenguaje de manipulación de datos (DML) 86

El lenguaje SQL

INSERT	87
DELETE	88
UPDATE	89
SELECT	91

El lenguaje DML

Son las instrucciones SQL que nos permiten llevar a cabo tareas de consultas o manipulación de datos en un sistema de BD

INSERT
DELETE
UPDATE
SELECT

INSERT (Añade filas a una tabla)

Un formato posible es:

INSERT INTO nombre-tabla VALUES (serie de valores)

El orden en el que se asignen los valores en la cláusula VALUES tiene que coincidir con el orden en que se definieron las columnas en la creación del objeto tabla, dado que los valores se asignan por posicionamiento relativo.

Por ejemplo:

```
INSERT INTO T_PEDIDOS VALUES (125,2,'PEPE');
```

Otra forma de usar la sentencia INSERT :

```
INSERT INTO nombre-tabla (columna1, columna2.....) VALUES (valor1, valor2.....)
```

En este caso los valores se asignarán a cada una de las columnas mencionadas por posicionamiento relativo.

Es necesario que por lo menos se asignen valores a todas aquellas columnas que no admiten valores nulos en la tabla (NOT NULL).

Por ejemplo:

```
INSERT INTO T_PEDIDOS (CODPEDIDO,ESTADO) VALUES (125,2);
```

DELETE (Borra una o más filas de una tabla, dependiendo de la condición WHERE

Sintaxis:

```
DELETE  
FROM nombre-tabla  
[WHERE condición]
```

DELETE (Borra una o más filas de una tabla, dependiendo de la condición WHERE

¡ATENCIÓN!

Si no se pone condición de selección, borra todas las filas de la tabla.

```
DELETE  
FROM T_PEDIDOS;
```

(Borrar todos los datos de la tabla)

```
DELETE  
FROM T_PEDIDOS  
WHERE COD_PEDIDO=15;
```

(Borrar solo el pedido numero 15)

UPDATE (Modifica uno o más datos de la tabla, depende de la condición WHERE)

```
UPDATE nombre-tabla  
SET columna1 = valor1 [, columna2 = valor2 ...]  
[WHERE condición]
```

Actualiza los campos correspondientes junto con los valores que se le asignen, en el subconjunto de filas que cumplan la condición de selección.

Si no se pone condición de selección, la actualización se da en todas las filas de la tabla.

Si se desea actualizar a nulos, se asignará el valor NULL.

En este ejemplo cambiamos el nombre y estado de un pedido:

```
UPDATE T_PEDIDOS
SET NOMBRE='JUAN',ESTADO=1
WHERE CODPEDIDO=125;
```

En este ejemplo cambiamos el estado de todos los pedidos:

```
UPDATE T_PEDIDOS
SET ESTADO=1;
```

En este ejemplo ponemos a nulo el nombre de un pedido:

```
UPDATE T_PEDIDOS
SET NOMBRE=NULL
WHERE CODPEDIDO=125;
```

SELECT

La selección sobre una tabla consiste en elegir un subconjunto de filas que cumplan (o no) algunas condiciones determinadas.

Sintaxis :

```
SELECT */ columna1, columna2,....
FROM nombre-tabla
[WHERE condición]
```

SQL está basado en el modelo relacional

Sentencia SQL típica:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

A1,A2 ... An representan los atributos

ri1,r2..rm representan las relaciones

P representa las condiciones

El resultado de una consulta SQL es una tabla (relación)

El select corresponde a la proyección de álgebra. Es usado para listar los atributos que queremos como resultado de la consulta.

Por ejemplo, mostrar el nombre de las sucursales que dieron préstamos

```
select Suc_Nom  
from Prestamos
```

Un asterisco en el select retorna “todos los atributos”

```
select *  
from Prestamos
```

NOTA: SQL no permite los signos matemáticos como nombres de campos

Cláusula WHERE

Con la sentencia `SELECT FROM` podemos seleccionar columnas de una tabla, pero para seleccionar filas de una tabla es preciso añadirle la cláusula `WHERE`. El formato es:

```
SELECT nombre_columnas_a_seleccionar  
FROM tabla_a_consultar  
WHERE condiciones;
```

La cláusula `WHERE` nos permite obtener las filas que cumplen la condición especificada en la consulta.

Veamos un ejemplo en el que pedimos "los códigos de los empleados que trabajan en el proyecto número 4":

```
SELECT codigo_empl  
FROM empleados  
WHERE num_proyec = 4;
```

Operadores

Para definir las condiciones en la cláusula `WHERE`, podemos utilizar alguno de los operadores de los que dispone el SQL, que son los siguientes: |

Operadores de comparación	
=	Igual
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual
< >	Diferente

Operadores lógicos	
NOT	Para la negación de condiciones
AND	Para la conjunción de condiciones
OR	Para la disyunción de condiciones

Veamos algunos ejemplos

Si ejecutamos:

```
SELECT *  
FROM T_PEDIDOS;
```

Nos da la salida:

COD_PEDIDO	NOMBRE	ESTADO
1	JUAN	0
2	ANTONIO	1
3	PEPE	0

```
SELECT *  
FROM T_PEDIDOS  
WHERE estado=0;
```

COD_PEDIDO	NOMBRE	ESTADO
1	JUAN	0
3	PEPE	0

Si deseamos solamente proyectar código de pedido y nombre:

```
SELECT cod_pedido,nombre
FROM T_PEDIDOS
WHERE estado=0;
```

Nos da la salida:

COD_PEDIDO	NOMBRE
1	JUAN
3	PEPE

El select puede contener expresiones aritméticas, +, -, *, /, estas operan con los contenidos de los campos

La consulta:

```
select prest_num, suc_nom, monto*100
from prestamos
```

Retorna los datos de los préstamos pero con los montos multiplicados por 100

La cláusula where corresponde a la condición que deben cumplir los registros para ser seleccionados.

Para obtener el número de préstamo realizado en la sucursal Paysandú y cuyo monto supere los \$1200.

```
select prest_num
from prestamos
where suc_nom = 'Paysandú' and monto > 1200
```

SQL incluye la cláusula between para poder obtener valores entre un rango determinado
Mostrar los numeros de préstamo cuyo monto están entre \$90,000 y \$100,000

```
select prest_num
from prestamos
where monto between 90000 and 100000
```


La cláusula FROM corresponde al producto cartesiano de las operaciones de álgebra relacional. Muestra las tablas que serán utilizadas para cumplir con la consulta.

Mostrar el producto cartesiano TIENE x RESTAMOS

```
select *  
from tiene, prestamos
```

Mostrar nombre, número de préstamo y monto de todos los clientes que tienen préstamos en la sucursal Paysandu

```
select cli_nombre, tiene.prest_num, monto  
from tiene, prestamos  
where tiene.prest_num = prestamos.prest_num  
and suc_nom = 'Paysandu'
```

SQL permite renombrar relaciones y atributos utilizando la cláusula:

```
nombre_viejo as nombre_nuevo
```

Obtener el nombre, el número de préstamo y el monto de todos los clientes renombrando el campo número de préstamo a número

```
select cli_nombre, tiene.prest_num as numero, monto  
from tiene, prestamo  
where tiene.prest_num = prestamo.prest_num
```

Utilizando la cláusula **as** pueden renombrarse las tablas de la cláusula from

Obtener nombre de cliente, número de préstamo y monto para todos los clientes que tienen préstamos en la misma sucursal

```
select T.cli_nom, T.loan-number, P.monto  
from tiene as T, prestamos as P  
where T.prest_num = P.prest_num
```

Obtener los nombres de todas las sucursales que tenga activos mayores que cualquiera de la sucursal de Paysandu

```
select distinct T.suc_nom
from sucursal as T, sucursal as S
where T.activos > S.activos and

      S.suc_ciudad = 'Paysandu'
```

SQL incluye operadores de string para comparar caracteres.

porcentaje (%). El % coincide con cualquier posición en la cadena
subrayado (_). El _ coincide con el inicio de la cadena

Obtener los nombres de los clientes en cuya calle figure la palabra Avenida

```
select cli_nombre
from clientes
where cli_calle like '%Avenida%'
```

Cuya calle comienza con la palabra Avenida
like 'Avenida\%' escape '^'

SQL soporta varios operadores de string, por ejemplo:

Concatenar (use "+"), convertir mayúscula a minúscula y viceversa (LOWER y UPPER), extraer partes de una cadena, largo de una cadena, etc etc.

Ordenar resultados

Listar en orden alfabético los nombres de todos los clientes que tienen préstamos en sucursales de Paysandú

```
select distinct cli_nombre
from tiene, prestamos
where tiene.prest_num=prestamos.prest_num

and suc_nom = 'Paysandu'
order by cli_nombre
```

Se puede especificar el orden inverso
order by cli_nombre desc

Funciones de agregación

Estas funciones operan en el conjunto de los valores de una columna y retornan un único valor.

- avg: promedio
- min: valor minimo
- max: valor maximo
- sum: suma de valores
- count: cantidad de tuplas

Obtener el promedio de saldos de las cuentas de Paysandu

```
select avg (saldo)
from cuentas,sucursal
where cuentas.suc_nom=sucursal.suc_nom
and suc_ciudad='Paysandu'
```

Obtener la cantidad total de clientes

```
select count (*)
from clientes
```

Obtener la cantidad de depositantes del banco

```
select count (distinct cli_nombre)
from deposita
```

Agrupar resultados

Si quisiera sacar totales agrupados por algún criterio debemos agregar una nueva cláusula a la consulta, la cláusula GROUP BY

Obtener el numero de depósitos por sucursal

```
select suc_nom, count (distinct cli_nombre)
  from deposita, cuentas
 where deposita.cta_num = cuentas.cta_num
 group by suc_nom
```

Nota: Los atributos de la cláusula select que no sean funciones de agregación, deben aparecer todos en la cláusula group by

Filtrar resultados agrupados

Si quisiera filtrar totales agrupados por algún criterio debemos utilizar la cláusula HAVING

Obtener los nombres de las sucursales que tengan un promedio en sus cuentas mayor a \$ 1200

```
select suc_nom, avg (saldo)
  from cuentas
 group by suc_nom
 having avg (saldo) > 1200
```

Nota: El filtro having se aplica luego de formado los grupos, a diferencia de la cláusula where que se aplica antes de obtener los datos, por eso podemos decir que Having es al Group By lo que Where es al Select, por lo tanto solo puede existir un having si hay un group by

Algunos ejemplos sencillos:

Si tenemos la tabla VALVULAS

	codValv	dscValv	diamValv	tipoValv	presValv	porcValv
498	JOL297	VALVULA JOL MOD :297	20.00	R	31.00	18.00
499	JOL298	VALVULA JOL MOD :298	20.00	R	31.00	18.00
500	JOL299	VALVULA JOL MOD :299	20.00	R	31.00	18.00
501	KHG5...	VALVULA KHG MOD :5...	35.00	R	40.00	18.00
502	KHG5...	VALVULA KHG MOD :5...	35.00	R	40.00	18.00
503	KHG5...	VALVULA KHG MOD :5...	35.00	R	40.00	18.00
504	KHG5...	VALVULA KHG MOD :5...	35.00	R	40.00	18.00
505	KHG5...	VALVULA KHG MOD :5...	35.00	R	40.00	18.00

y ejecutamos

```
SELECT SUM(presValv) as suma  
FROM valvulas
```

Obtenemos

	suma
1	293

GROUP BY

Suponemos que tenemos la tabla EMPLEADOS

ciEmp	nomEmp	fchNac	sueldoEmp	nomCiud
11111111	Juan Perez	1968-03-15	23500.00	CARMELO
11111115	Julio Menendez	1969-04-19	25500.00	ARTIGAS
11111116	Pedro Figueredo	1969-05-01	26500.00	CARMELO
11111118	Jose Gandolfo	1966-07-15	25500.00	ARTIGAS
11113111	Rodrigo Perez	1966-01-12	28500.00	PANDO
11115111	Rosendo Mesa	1967-04-10	20500.00	PANDO
11116111	Rodolfo Lagni	1967-09-10	20500.00	ARTIGAS
11117111	Ignacio Curbelo	1968-08-27	20500.00	ARTIGAS
12111111	Sebastian Almagro	1972-06-22	23500.00	CARMELO
13111111	Pablo Alvarez	1973-03-15	22500.00	CARMELO
14111111	Lucas Viassi	1974-09-10	23500.00	CARMELO
15111111	Lucia Torres	1968-02-01	22500.00	CARMELO
61111111	Andrea Pereira	1965-02-25	20500.00	CARMELO
71111111	Leticia Correa	1963-12-25	20500.00	PANDO
81111111	Eugenio Acosta	1961-11-15	25500.00	PANDO

Ejecutamos la siguiente consulta:

```
SELECT nomCiud,SUM(sueldoEmp) as sueldoCiudad
FROM empleados
GROUP BY nomciud
```

Obtenemos los sueldos de los empleados por departamento:

nomCiud	sueldoCiudad
ARTIGAS	92000.00
CARMELO	162500.00
PANDO	95000.00

El operador GROUP BY reorganiza en el sentido lógico la tabla representada en el FROM formando particiones o GRUPOS de manera que dentro de un grupo dado todas las filas tengan el mismo valor en el campo GROUP BY

Cada expresión en la cláusula SELECT debe producir un solo valor para cada grupo

}

HAVING

El operador HAVING es a los grupos lo que WHERE es a las filas, si se especifica HAVING deberá haberse especificado previamente un GROUP BY

HAVING filtra resultados agrupados, si lo aplicamos al ejemplo anterior, queremos del total de sueldos por ciudad quedarnos solamente con los que son menores que 100.000

```
SELECT nomCiud, SUM(sueldoEmp) as sueldoCiudad
FROM empleados
GROUP BY nomciud
HAVING SUM(sueldoEmp) < 100000
```

nomCiud	sueldoCiudad
ARTIGAS	92000.00
PANDO	95000.00

Consultas de Reunión

Ejemplo

Obtener todas las combinaciones de información de proveedor y producto donde el proveedor y la parte en cuestión estén situados en la misma ciudad pero omitiendo los proveedores cuya situación sea 20

```
SELECT S.*, P.*
FROM S, P
WHERE S.ciudad = P.ciudad AND
      S.situacion <> 20
```

Obtener nombre de empleado y descripción de la obra para los trabajos realizados en junio de 2010

```
SELECT empleados.nomEmp, obras.dscObra
FROM empleados, obras, trabajan
WHERE empleados.codEmp = trabajan.codEmp AND
      obras.codObra = trabajan.codObra AND
trabajan.fecha >= '01/06/2010' AND
trabajan.fecha <= '30/06/2010'
```

Obtener nombre de las maquinas que se utilizaron en trabajos de las obras situadas en Montevideo

```
SELECT maquinas.dscMq
FROM maquinas, trabajan, obras
WHERE maquinas.codMq = trabajan.codMq AND
      obras.codObra = trabajan.codObra AND
      obras.ciudad = 'MONTEVIDEO';
```

Obtener nombre de empleado y descripción de obra para todos los empleados que trabajaron en alguna obra, si un empleado trabajo más de una vez en una misma obra solo mostrarlo una vez

```
SELECT DISTINCT(nomEmp), dscObra
FROM empleados, trabajan, obras
WHERE empleados.codEmp = trabajan.codEmp AND
      trabajan.codObra = obras.codObra;
```


Si vamos al “problema de las válvulas” podemos definir los siguientes ejercicios:

Simples y de reunión

1. Mostrar los datos de las válvulas cuyo diámetro esté ente 10 y 20 mm
2. Mostrar RUT y Descripción de todas las Petroquímicas
3. Mostrar todos los datos de los arrendamientos entre el 01/01/2010 y el 31/12/2013
4. Mostrar número de serie y descripción de todas las válvulas
5. Mostrar número de serie y descripción de las válvulas arrendadas en mayo de 2013
6. Mostrar número de serie, descripción de la válvula, descripción de la Petroquímica para todas las válvulas arrendadas.
7. Mostrar los datos de los Inspectores asignados a las solicitudes de inspección (no mostrar datos repetidos).
8. Mostar las descripciones de las válvulas que son compatibles entre si.
9. Mostrar la descripción de la petroquímica y la descripción de los lugares donde están sus sucursales.
10. Para cada válvula mostrar número de serie y la descripción de sus dimensiones

Con funciones de agregación

11. Mostrar la cantidad total de modelos de válvulas diferentes arrendadas en el año
12. Mostrar la fecha del último arrendamiento
13. Mostrar para cada válvula cuantos números de serie tiene
14. Mostrar la descripción de cada Petroquímica y cuantas sucursales tiene.
15. Mostrar para cada válvula cual fue su primer arrendamiento y cuantas veces fue arrendada por cada numero de serie.
16. Mostrar cuantas solicitudes de inspección tuvo cada Agencia Gubernamental.
17. Mostrar cuantos modelos diferentes de válvula fueron arrendados por cada Petroquímica
18. ¿ Cuantas inspecciones tuvo cada inspector hasta el momento ?
19. Modificar el ejercicio 13 para que solo muestre las válvulas que tuvieron menos de 10 número de serie.
20. Modificar el ejercicio 17 para que muestre solo las Petroquímicas que arrendaron menos de 2 modelos diferentes de válvula.

Soluciones:

1.

```
SELECT *  
FROM valvulas  
WHERE diamValv >= 10 and  
      diamValv <= 20
```

2.

```
SELECT rutPet,dscPet  
FROM petroquimica
```

3.

```
SELECT *  
FROM arrienda  
WHERE fchArr >= '01/01/2010' and  
      fchArr <= '31/12/2013'
```

4.

```
SELECT series.numSerie, valvulas.dscValv  
FROM valvulas, series  
WHERE valvulas.codValv=series.codValv
```

5.

```
SELECT series.numSerie, valvulas.dscValv  
FROM valvulas, series, arrienda  
WHERE valvulas.codValv=series.codValv and  
      series.numSerie=arrienda.numSerie and  
      series.codValv=arrienda.codValv and  
      arrienda.fchArr >= '01/05/2013' and  
      arrienda.fchArr <= '31/05/2013'
```

6.

```
SELECT series.numSerie, valvulas.dscValv, petroquimica.dscPet  
FROM valvulas, series, arrienda, petroquimica  
WHERE valvulas.codValv=series.codValv and  
      series.numSerie=arrienda.numSerie and  
      series.codValv=arrienda.codValv and  
      arrienda.rutPet=petroquimica.rutPet
```

7.

```
SELECT DISTINCT inspectores.*  
FROM inspectores, solicitan  
WHERE inspectores.ciInsp=solicitan.ciInsp
```

8.

```
SELECT A.dscValv,B.dscValv
FROM valvulas A,valvulas B,compatible
WHERE A.codValv=compatible.codValv and
      B.codValv=compatible.CompValv
```

9.

```
SELECT petroquimica.dscPet,lugares.dscLug
FROM lugares,sucursal,petroquimica
WHERE lugares.codLug=sucursal.codLug and
      sucursal.rutPet=petroquimica.rutPet
```

10.

```
SELECT series.numSerie,dimensiones.dscDim
FROM series,dimensiones
WHERE series.codDim=dimensiones.codDim
```

11.

```
SELECT COUNT(DISTINCT(arrienda.codValv)) as Total
FROM arrienda
WHERE YEAR(arrienda.fchArr)=YEAR(getdate())
```

12.

```
SELECT MAX(arrienda.fchArr) as Ultimo
FROM arrienda
```

13.

```
SELECT codValv,COUNT(numSerie) as Cantidad
FROM series
GROUP BY codValv
```

14.

```
SELECT petroquimica.dscPet,COUNT(sucursal.rutPet) as Cantidad
FROM petroquimica,sucursal
WHERE petroquimica.rutPet=sucursal.rutPet
GROUP BY petroquimica.dscPet
```

15.

```
SELECT codValv, MAX(fchArr) as Primero, COUNT(DISTINCT(numSerie)) as Veces
FROM arrienda
GROUP BY codValv
```

16.

```
SELECT of_gub.codOffice, of_gub.dscOffice,
COUNT(solicitan.numInsp) as Cantidad
FROM of_gub, solicitan
WHERE of_gub.codOffice=solicitan.codOffice
GROUP BY of_gub.codOffice, of_gub.dscOffice
```

17.

```
SELECT petroquimica.dscPet,
COUNT(DISTINCT(arrienda.codValv)) as Cantidad
FROM petroquimica, arrienda
WHERE petroquimica.rutPet=arrienda.rutPet
GROUP BY petroquimica.dscPet
```

18.

```
SELECT inspectores.nomInsp, COUNT(inspecciones.numInsp) as Cantidad
FROM inspectores, inspecciones, solicitan
WHERE inspectores.ciInsp=solicitan.ciInsp and
solicitan.numInsp=inspecciones.numInsp
GROUP BY inspectores.nomInsp
```

19.

```
SELECT codValv, COUNT(numSerie) as Cantidad
FROM series
GROUP BY codValv
HAVING COUNT(numSerie) < 10
```

20.

```
SELECT petroquimica.dscPet,
COUNT(DISTINCT(arrienda.codValv)) as Cantidad
FROM petroquimica, arrienda
WHERE petroquimica.rutPet=arrienda.rutPet
GROUP BY petroquimica.dscPet
HAVING COUNT(DISTINCT(arrienda.codValv)) < 2
```

Sub Consultas

Una subconsulta es una sentencia SELECT que aparece dentro de otra sentencia SELECT.

Normalmente se utilizan para filtrar una cláusula WHERE o HAVING con el conjunto de resultados de la subconsulta.

Una subconsulta tiene la misma sintaxis que una sentencia SELECT normal exceptuando que aparece encerrada entre paréntesis.

Por ejemplo podríamos consultar el último retiro de su cuenta de un cliente dado:

```
SELECT COD_CLIENTE,NOMBRE,FECHA
FROM CUENTAS
WHERE COD_CLIENTE=1111 AND
      FECHA = (SELECT MAX(FECHA)
              FROM CUENTAS
              WHERE COD_CLIENTE=1111)
```

La subconsulta se puede encontrar en la lista de selección, en el FROM, en la cláusula WHERE o en la cláusula HAVING de la consulta principal.

Tiene las siguientes restricciones:

- a. No puede contener la cláusula ORDER BY
- b. No puede ser la UNION de varias sentencias SELECT
- c. Si la subconsulta aparece en la lista de selección o está asociada a un operador “=” solo puede devolver un único registro.

Referencia Externa

A menudo, es necesario, dentro del cuerpo de una subconsulta, hacer referencia al valor de una columna de la fila actual en la consulta principal, ese nombre de columna se denomina referencia externa.

Una referencia externa es un campo que aparece en la subconsulta pero se refiere a una de las tablas designadas en la consulta principal.

Cuando se ejecuta una consulta que contiene una subconsulta con referencias externas, la subconsulta se ejecuta por cada fila de la consulta principal, por ejemplo:

Para cada empleado, muestra el código, el nombre y el menor sueldo cobrado de cada empleado

```
SELECT COD_EMPLEADO,NOMBRE,  
       (SELECT MIN(FECHA_SUELDO)  
        FROM SUELDOS  
        WHERE SuelDOS.COD_EMPLEADO = Empleados.COD_EMPLEADO)  
FROM EMPLEADOS;
```

Anidar subconsultas

Las subconsultas pueden anidarse de forma que una subconsulta aparezca en la cláusula WHERE (por ejemplo) de otra subconsulta que a su vez forma parte de otra consulta principal.

```
SELECT COD_EMPLEADO,NOMBRE  
FROM EMPLEADOS  
WHERE COD_EMPLEADO IN  
      (SELECT COD_EMPLEADO  
       FROM SUELDOS  
       WHERE ESTADO IN ( SELECT ESTADO  
                          FROM ESTADOS_SUELDOS  
                          WHERE EMITIDO = 'S'  
                          AND PAGADO = 'N'))
```

Utilización de subconsultas con UPDATE

Podemos utilizar subconsultas también en consultas de actualización conjuntamente con UPDATE.

```
UPDATE EMPLEADOS
SET SALARIO_BRUTO = (SELECT SUM(SALARIO_BRUTO)
                     FROM SUELDOS
                     WHERE SUELDOS.COD_EMPLEADO = EMPLEADOS.COD_EMPLEADO)
WHERE SALARIO_BRUTO IS NULL
```

La función EXISTS

EXISTS es una función SQL que devuelve verdadero cuando una subconsulta retorna al menos una fila.

```
SELECT COD_CLIENTE, NOMBRE
FROM CLIENTES
WHERE EXISTS ( SELECT *
               FROM MOROSOS
               WHERE COD_CLIENTE = CLIENTES.COD_CLIENTE
                 AND PAGADO = 'N');
```

La función EXISTS puede ser utilizada en cualquier sentencia SQL válida.

La función SOME

(5 < some	<table><tr><td>0</td></tr><tr><td>5</td></tr><tr><td>6</td></tr></table>	0	5	6) = true	5 < que alguna de las <u>tuplas</u> de la relación (6)
0						
5						
6						
(5 < some	<table><tr><td>0</td></tr><tr><td>5</td></tr></table>	0	5) = false	(5 no es < que alguna de las <u>tuplas</u> de la relación)	
0						
5						
(5 = some	<table><tr><td>0</td></tr><tr><td>5</td></tr></table>	0	5) = true	(5 es = a alguna de las <u>tuplas</u> de la relación)	
0						
5						

Por ejemplo:

Obtener el nombre de sucursales cuyos activos sean mayores a algún activo de la sucursal de Salto

```
SELECT nombre_sucursal
FROM sucursal
WHERE activo > SOME (SELECT activo
                     FROM sucursal
                     WHERE ciudad = 'Salto')
```

La función ALL

(5 < all	<table><tr><td>0</td></tr><tr><td>5</td></tr><tr><td>6</td></tr></table>	0	5	6) = false	5 no es < que todas las <u>tuplas</u> de la relación
0						
5						
6						
(5 < all	<table><tr><td>6</td></tr><tr><td>10</td></tr></table>	6	10) = true	5 es < que todas las <u>tuplas</u> de la relación	
6						
10						
(5 = all	<table><tr><td>4</td></tr><tr><td>5</td></tr></table>	4	5) = false	5 no es = que todas las <u>tuplas</u> de la relación	
4						
5						

Por ejemplo:

Obtener nombre de las sucursales que tienen activos mayores a todos los activos de la ciudad de Salto

```
SELECT nombre_sucursal
FROM sucursal
WHERE activo > ALL (SELECT activo
                   FROM sucursal
                   WHERE ciudad = 'Salto')
```


Subconsultas en el FROM

Como ya sabemos, el resultado de una consulta SQL es una tabla, por lo tanto puede utilizarse como cualquier otra tabla dentro de una cláusula FROM

Ejemplo:

Del total facturado por cliente, obtener el mayor importe, el menor importe y el promedio de totales.

```
SELECT MAX(miTabla.total), MIN(miTabla.total), AVG(miTabla.total)
FROM (SELECT cliente, SUM(importe) as total
      FROM facturas
      GROUP BY cliente) miTabla
```

Nota: Toda consulta que se utiliza dentro del FROM debe tener un nombre de tabla, para eso se utiliza un alias (en este caso **miTabla**)

Volviendo al “problema de las válvulas” podemos definir los siguientes ejercicios de subconsultas:

1. Para el arrendamiento más reciente, mostrar código y descripción de los modelos de válvulas de dicha fecha.
2. Mostrar los modelos de válvulas que nunca fueron arrendados en el año 2013
3. Mostrar nombre de los inspectores que nunca realizaron inspecciones.
4. Mostrar los datos de las petroquímicas que hicieron arrendamientos en la fecha más antigua.
5. Mostrar los datos de los lugares que tienen sucursales de petroquímicas.
6. Mostrar cuantos modelos diferentes de válvula fueron arrendados por cada Petroquímica, solo deben figurar aquellos modelos cuya cantidad sea superior al total general arrendado en el mes de febrero de 2014.
7. Para cada petroquímica, mostrar su nombre, la cantidad de modelos de válvula diferentes que arrendó, cual fue la fecha de su primer arrendamiento y cual fue la fecha de su último arrendamiento, si la petroquímica no tiene arrendamientos, también se deben mostrar sus datos.
8. Para las válvulas de tipo retención ('R') poner el valor de la presión de acuerdo al promedio de valores de presión de todas las válvulas del mismo tipo.
9. Borrar de la tabla dimensiones todos aquellos registros que no tienen ningún número de serie asociado.
10. Del total de arrendamientos que tuvo cada petroquímica, obtener cual fue el promedio, el máximo y el mínimo.

Soluciones:

1.

```
SELECT valvulas.codValv, valvulas.dscValv
FROM valvulas, arrienda
WHERE valvulas.codValv=arrienda.codValv and
      arrienda.fchArr = (SELECT MAX(fchArr)
                        FROM arrienda)
```

2.

```
SELECT valvulas.*
FROM valvulas
WHERE codValv NOT IN (SELECT codValv
                     FROM arrienda
                     WHERE fchArr >= '01/01/2013' and
                           fchArr <= '31/12/2013')
```

3.

```
SELECT nomInsp
FROM inspectores
WHERE ciInsp NOT IN (SELECT solicitacion.ciInsp
                   FROM solicitacion, inspecciones
                   WHERE solicitacion.numInsp=inspecciones.numInsp)
```

4.

```
SELECT petroquimica.*
FROM petroquimica, arrienda
WHERE petroquimica.rutPet=arrienda.rutPet and
      arrienda.fchArr = (SELECT MIN(fchArr)
                        FROM arrienda)
```

5.

```
SELECT lugares.*
FROM lugares
WHERE lugares.codLug IN (SELECT sucursal.codLug
                       FROM sucursal)
```

6.

```
SELECT petroquimica.dscPet,  
       COUNT(DISTINCT(arrienda.codValv)) as Cantidad  
FROM petroquimica,arrienda  
WHERE petroquimica.rutPet=arrienda.rutPet  
GROUP BY petroquimica.dscPet  
HAVING COUNT(DISTINCT(arrienda.codValv)) > (SELECT COUNT(distinct(codValv))  
                                             FROM arrienda  
                                             WHERE fchArr >= '01/02/2014' and  
                                             fchArr <= '28/02/2014')
```

7.

```
SELECT dscPet,(SELECT COUNT(DISTINCT(codValv))  
              FROM arrienda  
              WHERE rutPet=petroquimica.rutPet) as total,  
       (SELECT MIN(fchArr)  
        FROM arrienda  
        WHERE rutPet=petroquimica.rutPet) as primer,  
       (SELECT MAX(fchArr)  
        FROM arrienda  
        WHERE rutPet=petroquimica.rutPet) as ultimo  
FROM petroquimica
```

8.

```
UPDATE valvulas  
SET presValv=(SELECT AVG(presValv)  
              FROM valvulas  
              WHERE tipoValv='R')  
WHERE tipoValv='R'
```

9.

```
DELETE  
FROM dimensiones  
WHERE codDim NOT IN (SELECT codDim  
                     FROM series)
```

10.

```
SELECT AVG(miTabla.tot) as prom,  
       MAX(miTabla.tot) as maximo,  
       MIN(miTabla.tot) as minimo  
FROM (SELECT rutPet,COUNT(rutPet) as tot  
      FROM arrienda  
      GROUP BY rutPet) miTabla
```

SQL JOIN a partir del estándar SQL 92

A partir del estándar SQL-92 se pueden escribir las sentencias SQL de más de una tabla utilizando las cláusulas INNER JOIN y OUTER JOIN

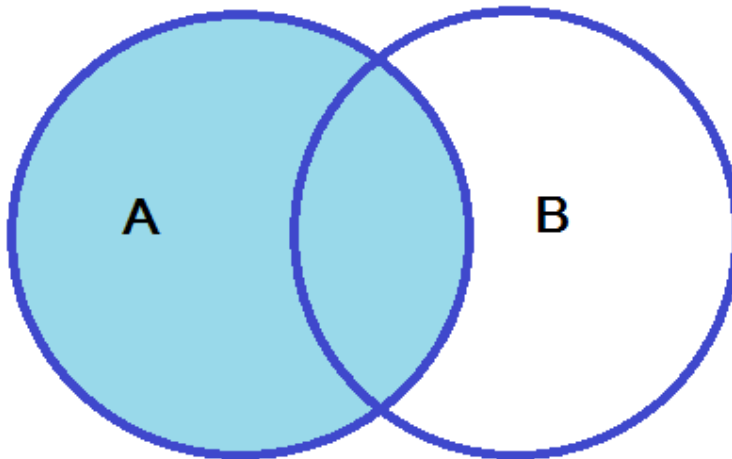
Por ejemplo la siguiente consulta:

```
SELECT series.numSerie, valvulas.dscValv  
FROM valvulas, series  
WHERE valvulas.codValv=series.codValv
```

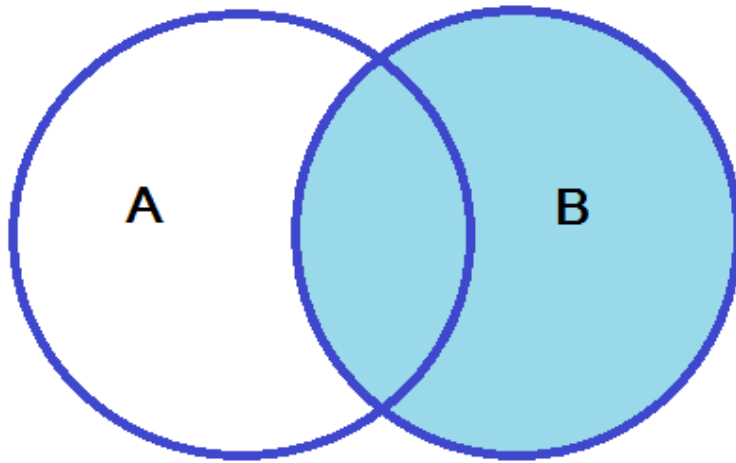
Puede escribirse de la siguiente manera:

```
SELECT series.numSerie, valvulas.dscValv  
FROM valvulas INNER JOIN series  
ON valvulas.codValv=series.codValv
```

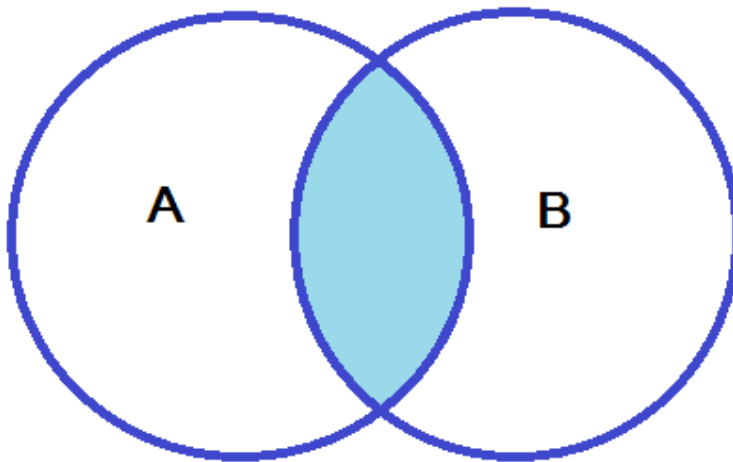
En los siguientes diagramas les dejamos más detalle sobre esta notación:



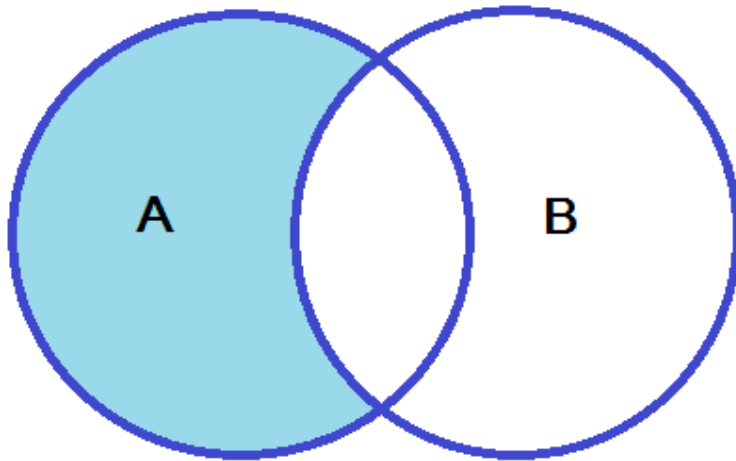
```
SELECT campos  
FROM A LEFT JOIN B  
ON A.clave = B.clave
```



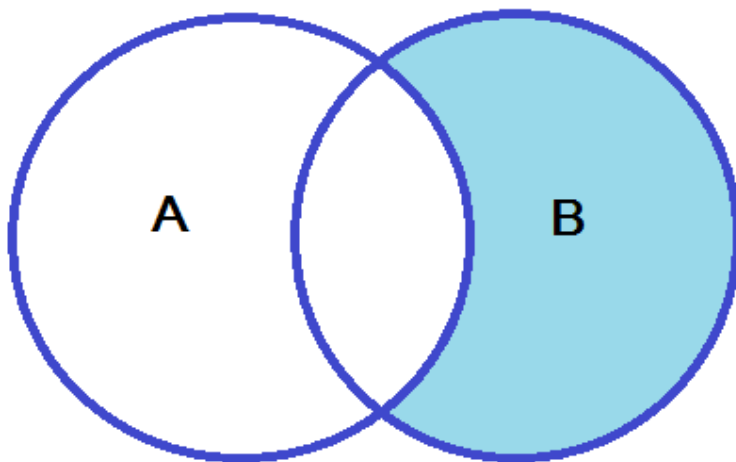
```
SELECT campos  
FROM A RIGHT JOIN B  
ON A.clave = B.clave
```



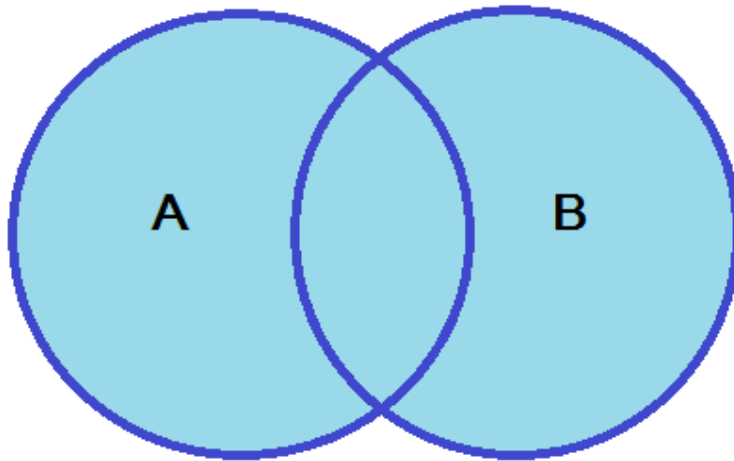
```
SELECT campos  
FROM A INNER JOIN B  
ON A.clave = B.clave
```



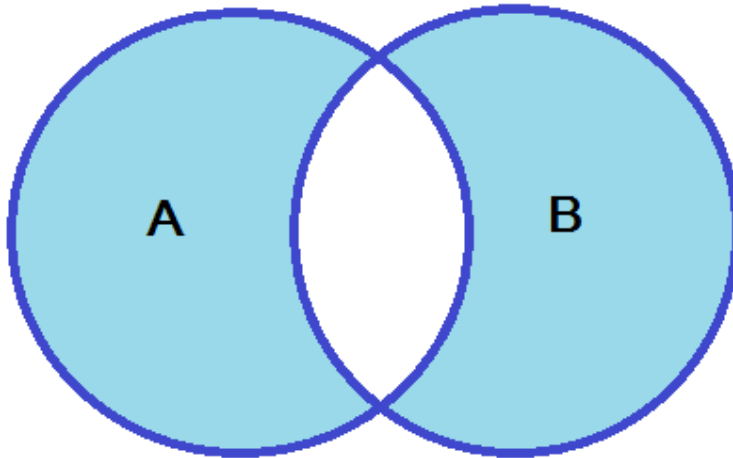
```
SELECT campos  
FROM A LEFT JOIN B  
ON A.clave = B.clave  
WHERE B.clave IS NULL
```



```
SELECT campos  
FROM A RIGHT JOIN B  
ON A.clave = B.clave  
WHERE A.clave IS NULL
```



```
SELECT campos  
FROM A FULL OUTER JOIN B  
ON A.clave = B.clave
```



```
SELECT campos  
FROM A FULL OUTER JOIN B  
ON A.clave = B.clave  
WHERE A.clave IS NULL or B.clave IS NULL
```

Módulo 5

Lenguajes de programación de SGBD

T-SQL

PL/SQL

SQL es un lenguaje de consulta para los sistemas de bases de datos relacionales, pero que no posee la potencia de los lenguajes de programación.

Transact SQL y PL/SQL son lenguajes de programación que proporcionan SQL Server y Oracle para ampliar SQL con los elementos característicos de los lenguajes de programación: variables, sentencias de control de flujo, bucles, etc.

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales.

Transact SQL y PL/SQL son los lenguajes de programación que proporcionan SQL Server y Oracle para extender el SQL estándar con otro tipo de instrucciones.

Microsoft T-SQL (Transact SQL)

Con Transact SQL vamos a poder programar las unidades de programa de la base de datos SQL Server, están son:

Procedimientos almacenados

Funciones

Triggers (disparadores)

Scripts

Tipos de datos en Transact SQL

Cuando definimos una tabla, variable o constante debemos asignar un tipo de dato que indica los posibles valores. El tipo de datos define el formato de almacenamiento, espacio que de disco-memoria que va a ocupar un campo o variable, restricciones y rango de valores validos.

Transact SQL proporciona una variedad predefinida de tipos de datos . Casi todos los tipos de datos manejados por Transact SQL son similares a los soportados por SQL.

Tipos de datos NUMERICOS

Bit. Una columna o variable de tipo bit puede almacenar el rango de valores de 1 a 0.

Tinyint. Una columna o variable de tipo tinyint puede almacenar el rango de valores de 0 a 255.

SmallInt. Una columna o variable de tipo smallint puede almacenar el rango de valores -32768 a 32767.

Int. Una columna o variable de tipo int puede almacenar el rango de valores -231 a 231-1 .

BigInt. Una columna o variable de tipo bigint puede almacenar el rango de valores -263 a 263-1 .

Decimal(p,s). Una columna de tipo decimal puede almacenar datos numéricos decimales sin redondear. Donde p es la precisión (número total del dígitos) y s la escala (número de valores decimales)

Float. Una columna de datos float puede almacenar el rango de valores -1,79x-10308 a 1,79x-10308, , si la definimos con el valor máximo de precisión. La precisión puede variar entre 1 y 53.

Real. Sinónimo de float(24). Puede almacenar el rango de valores -3,4x-1038 a 3,4x-1038

Money. Almacena valores numéricos monetarios de -263 a 263-1, con una precisión de hasta diez milésimas de la unidad monetaria.

SmallMoney. Almacena valores numéricos monetarios de -214.748,3647 a 214.748,3647, con una precisión de hasta diez milésimas de la unidad monetaria.

Todos los tipos de datos enteros pueden marcarse con la propiedad identity para hacerlos auto numéricos.

Tipos de datos de CHARACTER

Char(n). Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo char, siempre se utilizan los n caracteres indicados, incluso si la entrada de datos es inferior. Por ejemplo, si en un char(5), guardamos el valor 'A', se almacena 'A ', ocupando los cinco bytes.

Varchar(n). Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo varchar, unicamente se utilizan los caracteres necesarios, Por ejemplo, si en un varchar(255), guardamos el valor 'A', se almacena 'A', ocupando solo un byte bytes.

Nchar(n). Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferente idiomas.

Nvarchar(n). Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferente idiomas.

Tipos de datos FECHA

Datetime. Almacena fechas con una precisión de milisegundo. Debe usarse para fechas muy específicas.

SmallDatetime. Almacena fechas con una precisión de minuto, por lo que ocupa la mitad de espacio de que el tipo datetime, para tablas que puedan llegar a tener muchos datos es un factor a tener muy en cuenta.

TimeStamp. Se utiliza para marcar un registro con la fecha de inserción - actualización. El tipo timestamp se actualiza automáticamente cada vez que insertamos o modificamos los datos.

Tipos de datos BINARIOS

Binary. Se utiliza para almacenar datos binarios de longitud fija, con una longitud máxima de 8000 bytes.

Varbinary. Se utiliza para almacenar datos binarios de longitud variable, con una longitud máxima de 8000 bytes..Es muy similar a binary, salvo que varbinary utiliza menos espacio en disco.

Varbinary(max). Igual que varbinary, pero puede almacenar 231-1 bytes

Tipos de datos XML

XML. Una de las grandes mejoras que incorpora SQL Server es el soporte nativo para XML. Como podemos deducir, este tipo de datos se utiliza para almacenar XML.

```
DECLARE @myxml XML
```

```
set @myxml = (SELECT @@SERVERNAME NOMBRE FOR XML RAW, TYPE)
```

```
print cast(@myxml as varchar(max))
```

VARIABLES EN T-SQL

Una variable es un valor identificado por un nombre (identificador) sobre el que podemos realizar modificaciones.

En Transact SQL los identificadores de variables deben comenzar por el caracter @

Para declarar variables en Transact SQL debemos utilizar la palabra clave declare, seguido del identificador y tipo de datos de la variable.

```
declare @nombre varchar(50) -- declare declara una variable
```

```
set @nombre = 'www.google.com' -- El signo = es un operador
```

```
print @Nombre -- Imprime por pantalla el valor de @nombre.
```

Asignar variables en T-SQL

En Transact SQL podemos asignar valores a una variable de 2 formas:

A través de la instrucción SET

```
DECLARE @nombre VARCHAR(100)
-- La consulta debe devolver un único registro
SET @nombre = (SELECT nombre
                FROM CLIENTES
                WHERE ID = 1)
```

Utilizando una sentencia SELECT

```
DECLARE @nombre VARCHAR(100),

SELECT @nombre=nombre
FROM CLIENTES
WHERE ID = 1
```

Estructura condicional IF

```
IF (<expresion>)
BEGIN
    ...
END
ELSE IF (<expresion>)
BEGIN
    ...
END
ELSE
BEGIN
    ...
END
```

Estructura CASE

```
CASE <expresion>
  WHEN <valor_expresion> THEN <valor_devuelto>
  WHEN <valor_expresion> THEN <valor_devuelto>
  ELSE <valor_devuelto> -- Valor por defecto
END
```

Ejemplo

```
DECLARE @Descripcion varchar(100),
        @Tipo varchar(3)
SET @Tipo = 'ATI'
SET @Descripcion = (CASE
                      WHEN @Tipo = 'ATI' THEN 'Tecnologias de Informacion'
                      WHEN @Tipo = 'AP' THEN 'Analista Programador'
                      ELSE 'Otros'
                      END)
```

Bucle WHILE

El bucle WHILE se repite mientras expresión se evalúe como verdadero.
Es el único tipo de bucle del que dispone Transact SQL.

```
DECLARE @dato int
SET @dato = 0
WHILE (@dato < 100)
BEGIN
  SET @dato = @dato + 1
  PRINT 'Iteracion del bucle ' + cast(@dato AS varchar)
END
```

Control de errores con T-SQL

```
BEGIN TRY
DECLARE @divisor int ,@dividendo int,@resultado int
SET @dividendo = 100
SET @divisor = 0 -- Esta linea provoca un error de division por 0
SET @resultado = @dividendo/@divisor
PRINT 'No hay error'
END TRY
BEGIN CATCH
PRINT 'Se ha producido un error de dividir por 0'
END CATCH
```

Oracle PL/SQL

PL/SQL (Procedural Language/SQL) es una extensión de SQL, que agrega ciertas construcciones propias de lenguajes procedimentales, obteniéndose como resultado un lenguaje estructural más poderoso que SQL.

La unidad de programación utilizada por PL/SQL es el bloque.

Todos los programas de PL/SQL están conformados por bloques.

Un bloque tendrá siempre la siguiente estructura:

```
DECLARE
//Sección declarativa: variables, tipos, y subprogramas //de uso local
BEGIN
//Sección ejecutable: las instrucciones procedimentales, y de SQL //aparecen aquí. Es la única
sección obligatoria en el bloque.
EXCEPTION
//Sección de manejo de excepciones. Las rutinas de manejo de errores //aparecen aquí
END;
```

Las únicas instrucciones SQL permitidas en un bloque PL/SQL son INSERT, UPDATE, DELETE y SELECT, además de algunas instrucciones para manipulación de datos, e instrucciones para control de transacciones

Otras instrucciones de SQL como DROP, CREATE o ALTER no son permitidas

Se permite el uso de comentarios estilo `(/* ...*/)`.

PL/SQL no es “case sensitive” por lo que no hay distinción entre nombres con mayúsculas y minúsculas.

En la sección de declaraciones, se indican las variables que serán usadas dentro del bloque y sus tipos.

Por ejemplo:

```
DECLARE
miDescrip VARCHAR(20);
precio NUMBER(6,2);
```

Es posible inicializar las variables, mediante el operador :=

Además, mediante el uso del mismo operador es posible hacer asignaciones en el cuerpo del programa.

Por ejemplo:

```
DECLARE
precio NUMBER := 300;
BEGIN
    precio := precio + 150;
END;
```

Es posible usar sentencias condicionales y ciclos dentro de los bloques de PL/SQL.

```
IF (condicion)
THEN (lista de acciones)
ELSE (lista de acciones)
END IF;
```

Módulo 6

Procedimientos almacenados

Funciones almacenadas

Disparadores (Triggers)

Procedimientos Almacenados

Un procedimiento es un programa dentro de la base de datos que ejecuta una acción o conjunto de acciones específicas.

Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

En Transact SQL los procedimientos almacenados pueden devolver valores (numérico entero) o conjuntos de resultados.

```
CREATE PROCEDURE addCliente @nombre varchar(100),  
                             @apellido varchar(100),  
                             @fchNacimiento datetime  
AS
```

```
INSERT INTO CLIENTES  
(nombre, apellido, fchNacimiento)  
VALUES  
(@nombre, @apellido, @fchNacimiento)
```

Ventajas de usar SP

Compilación: La primera vez que se invoca un SP, el motor lo compila y a partir de ahí, se sigue usando la versión compilada del mismo, hasta que se modifique o se reinicie el servicio de SQL. Esto significa que se tendrá un mejor rendimiento que las consultas directas que usan cadenas con las instrucciones T-SQL, que se compilan cada vez que se invocan.

Automatización: si tenemos un conjunto de instrucciones T-SQL, las cuales queremos ejecutar de manera ordenada, un SP es la mejor manera de hacerlo.

Administración: cuando realizamos aplicaciones con un gran número de líneas de código, y queremos hacer cambios, solo implica modificar un SP y no toda la aplicación, lo que significa solo cambiamos los SP en el servidor y no tenemos que actualizar la aplicación en todos los equipos cliente.

Seguridad: A los usuarios de nuestra aplicación, solo les proporcionamos los permisos para ejecutar los procedimientos almacenados y no el acceso a todos los objetos de la base, si en nuestra aplicación encuentran una vulnerabilidad como SQL Injection no se podrá explotar ejecutando SQL directamente.

Programabilidad: Los SP admiten el uso de variables y estructuras de control como IF, Bucles, Case, etc. además del manejo de transacción y permite controlar excepciones.

Trafico de Red: Pueden reducir el trafico de la red, debido a que se trabaja sobre el motor (en el servidor), y si una operación incluye hacer un trabajo de lectura primero y en base a eso realizar algunas operaciones, esos datos que se obtienen no viajan por la red.

Parámetros de salida

Si queremos que los parámetros de un procedimiento almacenado sean de entrada-salida debemos especificarlo a través de la palabra clave OUTPUT , tanto en la definición del procedure como en la ejecución.

```
CREATE PROCEDURE spu_ObtenerSaldoCuenta
@numCuenta varchar(20),
@saldo decimal(10,2) output
AS
BEGIN
    SELECT @saldo = SALDO
    FROM CUENTAS
    WHERE NUMCUENTA = @numCuenta
END
```

Para ejecutarlo

```
DECLARE @saldo decimal(10,2)
EXEC spu_ObtenerSaldoCuenta '200700000001', @saldo output
PRINT @saldo
```

Otra característica muy interesante de los procedimientos almacenados en Transact-SQL es que pueden devolver uno o varios conjuntos de resultados.

```
CREATE PROCEDURE spu_MovimientosCuenta
@numCuenta varchar(20)
AS
BEGIN
    SELECT @numCuenta,
           SALDO_ANTERIOR,
           SALDO_POSTERIOR,
           IMPORTE,
           FXMOVIMIENTO
    FROM MOVIMIENTOS, CUENTAS
    WHERE CUENTAS.IDCUENTA=MOVIMIENTOS.IDCUENTA AND
          NUMCUENTA = @numCuenta
    ORDER BY FXMOVIMIENTO DESC
END
```

Funciones Almacenadas

SQL Server proporciona al usuario la posibilidad de definir sus propias funciones, conocidas como UDF (user defined functions).

Destacamos los dos tipos de funciones más utilizados:

Funciones escalares

Las funciones escalares devuelven un único valor de cualquier tipo de los datos tal como int, money, varchar, real, etc.

```
CREATE FUNCTION MultiplicaSaldo
(@NumCuenta VARCHAR(20),@Multiplicador DECIMAL(10,2))

RETURNS DECIMAL(10,2)

AS
BEGIN

    DECLARE @Saldo DECIMAL(10,2),
            @Return DECIMAL(10,2)
    SELECT @Saldo = SALDO
    FROM CUENTAS
    WHERE NUMCUENTA = @NumCuenta

    SET @Return = @Saldo * @Multiplicador

    RETURN @Return

END
```

Funciones en línea

Las funciones en línea son las funciones que devuelven un conjunto de resultados correspondientes a la ejecución de una sentencia SELECT.

```
CREATE FUNCTION MovimientosCuenta(@NumCuenta VARCHAR(20))
```

```
RETURNS TABLE
```

```
AS
```

```
RETURN
```

```
(  
  SELECT MOVIMIENTOS.*  
  FROM MOVIMIENTOS,CUENTAS  
  WHERE MOVIMIENTOS.IDCTA=CUENTAS.IDCTA AND  
         CUENTAS.IDCUENTA = @NumCuenta  
)
```

Ejercicios utilizando el problema de las “Válvulas”

1. Crear un procedimiento almacenado que reciba como parámetro todos los datos de una Petroquímica, si dicha Petroquímica existe que modifique sus datos, si no existe que la de de alta.
2. Realizar un procedimiento almacenado que baje un porcentaje dado de ganancia para todos aquellos modelos de válvulas que no se arriendan desde hace más de 2 años.
3. Crear un procedimiento almacenado que dado un RUT de una Petroquímica muestre todos sus arrendamientos que aún estén pendientes de finalizar.
4. Crear un procedimiento almacenado que reciba un rango de fechas y un código de lugar y muestre todos los arrendamientos entre ambas fechas para ese lugar, se deben mostrar las descripciones y no los códigos.
5. Crear una función almacenada que reciba un rango de fechas y retorne la cantidad de arrendamientos que se realizaron en dicho rango, cada arrendamiento se toma por un modelo de válvula y no por su número de serie.
6. Crear una función almacenada que dado un RUT de una Petroquímica, retorne el mayor precio de alquiler alguna vez cobrado a dicha Petroquímica.
7. Crear una función almacenada que retorne el nombre de la Petroquímica a la que se le cobró el menor precio de arrendamiento.
8. Crear una función almacenada que dado un inspector, retorne la fecha de su última inspección.

Soluciones:

1.

```
CREATE PROCEDURE addPetroquimica
@rutPet char(13),
@dscPet varchar(30),
@dirPet varchar(30),
@telPet varchar(30)

AS
BEGIN
SET NOCOUNT ON;
DECLARE @existe NUMERIC(6)

SELECT @existe=COUNT(*)
FROM petroquimica
WHERE rutPet=@rutPet

IF @existe=0
    INSERT INTO petroquimica VALUES(@rutPet,@dscPet,@dirPet,@telPet)
ELSE
    UPDATE petroquimica
    SET dscPet=@dscPet,dirPet=@dirPet,telPet=@telPet
    WHERE rutPet=@rutPet

END
```

2.

```
CREATE PROCEDURE modPorcentaje
@porcValv numeric(4,2)

AS
BEGIN

UPDATE valvulas
SET porcValv=porcValv - (porcValv * @porcValv)/100
WHERE codValv IN (SELECT codValv
    FROM arrienda
    WHERE year(fchArr) <= year(getdate())-2)

END
```


3.

```
CREATE PROCEDURE verPendientes
@rutPet char(13)

AS
BEGIN

    SET NOCOUNT ON;

    SELECT *
    FROM (select *,DATEDIFF(DAY, getdate(), finfch) as avencer
        from arrienda) miTabla
    WHERE avencer < 0 and rutPet=@rutPet
END
```

4.

```
CREATE PROCEDURE verArrienda
    @desde char(10),
    @hasta char(10),
    @codLug numeric(6)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    SELECT arrienda.codValv,dscValv,numSerie,
    arrienda.rutPet,dscPet,arrienda.codLug,
    dscLug,fcharr,precio,finfch
    FROM arrienda,valvulas,petroquimica,lugares
    WHERE arrienda.codLug=lugares.codLug and
    arrienda.codValv=valvulas.codValv and
    arrienda.rutPet=petroquimica.rutPet and
    fcharr >=@desde and fchArr <= @hasta and
    arrienda.codlug=@codLug

END
```

5.

```
CREATE FUNCTION darCantArrendamientos
(
    @desde char(10),@hasta char(10)
)
RETURNS NUMERIC(6)
AS
BEGIN

    DECLARE @ResultVal NUMERIC(6)

    SELECT @ResultVal=count(distinct(codValv))
    FROM arrienda
    WHERE fchArr >= @desde and
        fchArr <= @hasta

    RETURN @ResultVal

END
```

6.

```
CREATE FUNCTION darMaxPetroquimica
(
    @rutPet CHARACTER(13)
)
RETURNS NUMERIC(6)
AS
BEGIN

    DECLARE @ResultVal NUMERIC(6)

    SELECT @ResultVal=MAX(precio)
    FROM arrienda
    WHERE rutPet=@rutPet

    RETURN @ResultVal

END
```

7.

```
CREATE FUNCTION darMinPetroquimica()  
RETURNS VARCHAR(30)  
  
AS  
BEGIN  
  
    DECLARE @ResultVal VARCHAR(30)  
  
    SELECT TOP 1 @ResultVal=dscPet  
    FROM petroquimica,arrienda  
    WHERE petroquimica.rutPet=arrienda.rutPet AND  
          arrienda.precio = (SELECT MIN(precio)  
                             FROM arrienda)  
  
    RETURN @ResultVal  
  
END
```

8.

```
CREATE FUNCTION darMaxFecha  
(@ciInsp NUMERIC(8))  
RETURNS DATE  
  
AS  
BEGIN  
  
    DECLARE @ResultVal DATE  
  
    SELECT @ResultVal=MAX(fchInsp)  
    FROM (SELECT fchInsp  
          FROM solicitan,inspecciones  
          WHERE solicitan.numInsp=inspecciones.numInsp and  
                solicitan.ciInsp=@ciInsp) miTabla  
  
    RETURN @ResultVal  
  
END
```

Disparadores (Triggers)

Características de los disparadores

Está asociado a una única tabla base, es el concepto clave para implementar BD activas

Tiene 3 partes:

1. Un evento: indica la acción sobre la tabla base que causará se active el disparador (INSERT, UPDATE, DELETE)
2. Un tiempo de acción: indica cuando se activará el disparo BEFORE = Antes del evento AFTER = Después del evento
3. Una acción: se llevan a cabo si ocurre el evento, puede ser de dos tipos: Sentencia SQL Un bloque atómico de sentencias SQL

Permiten obtener notificaciones cuando ocurren ciertas condiciones.

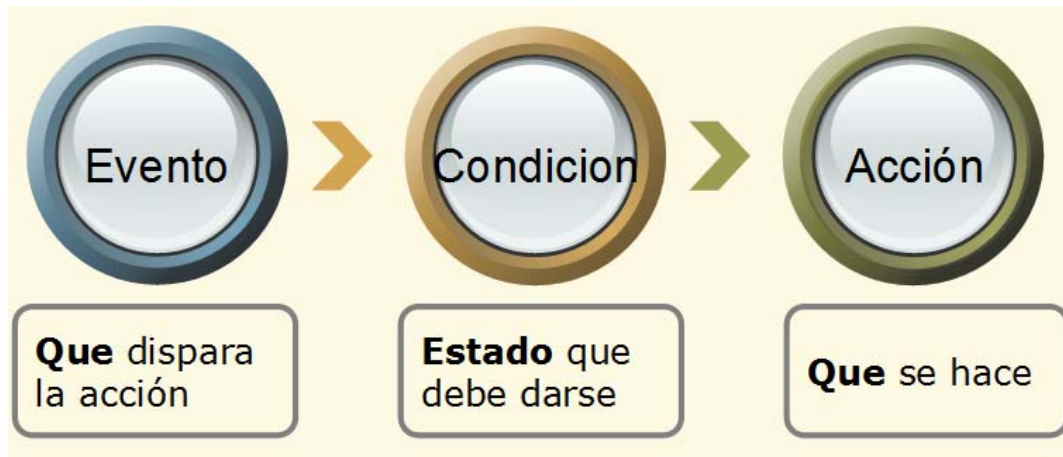
Permiten reforzar las restricciones de integridad.

Los disparadores son más potentes que las restricciones

Permiten la actualización automática de datos derivados evitando anomalías debidas a la redundancia

Ejemplo: Un disparador actualiza el saldo total de una cuenta bancaria cada vez que se inserta, elimina o modifica un movimiento de dicha cuenta

Cumple con las reglas ECA (Evento Condición Acción)



Evento

Que ocasiona la ocurrencia de un evento?

Una instrucción DML (antes o después): INSERT, DELETE, UPDATE

Una instrucción para gestión de transacciones: COMMIT, ROLLBACK

Una excepción: bloqueo, violación de autorización, etc.

El reloj: el 10 de abril a las 13:45hs

La aplicación: externo a la BD

Condición

Un predicado sobre la BD: consulta

Puede ser opcional: si no se incluye la condición es siempre cierta

Acción

Que se puede incluir en la reacción ?

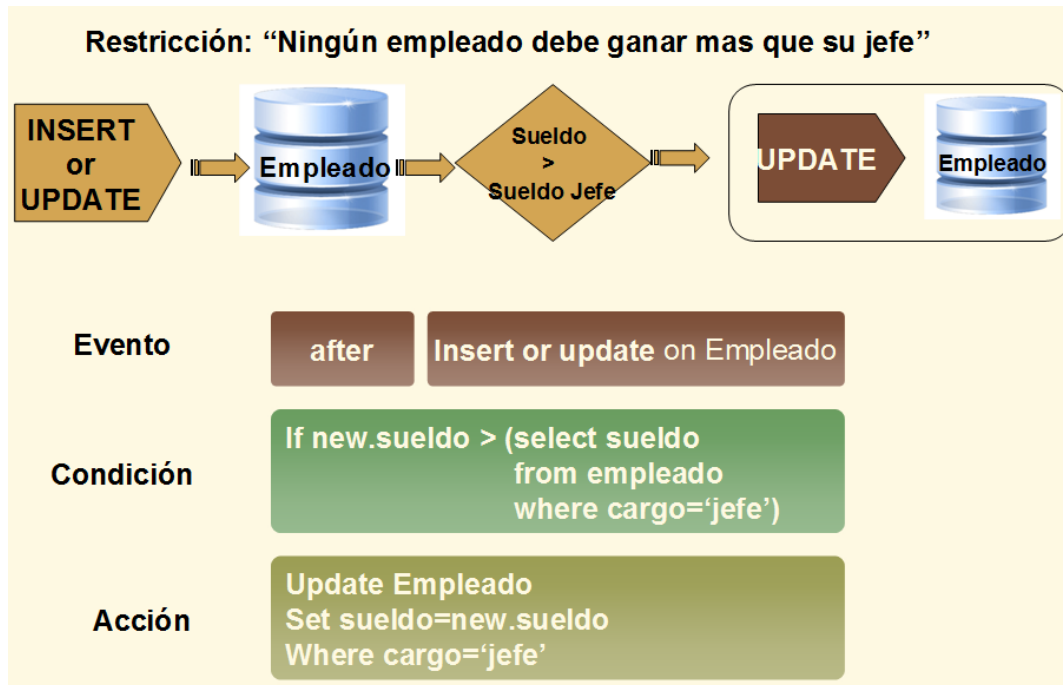
Operación en la BD: ordenes de SQL, INSERT, DELETE ...

Comandos de SQL extendido: PL/SQL, T-SQL

Llamadas externas: envío de mensajes

Abortar la transacción

Por ejemplo:



Sintaxis en T-SQL

```
CREATE TRIGGER <Trigger_Name>  
ON <Table_Name>  
AFTER INSERT,DELETE,UPDATE  
AS  
BEGIN  
  
END
```

Las tablas INSERTED y DELETED

Antes de ver un ejemplo es necesario conocer las tablas inserted y deleted.

Las instrucciones de triggers DML utilizan dos tablas especiales denominadas **inserted** y **deleted**.

SQL Server crea y administra automáticamente ambas tablas.

La estructura de las tablas inserted y deleted es la misma que tiene la tabla que ha desencadenado la ejecución del trigger.

La primera tabla (inserted) solo está disponible en las operaciones INSERT y UPDATE y en ella están los valores resultantes después de la inserción o actualización. Es decir, los datos insertados. Inserted estará vacía en una operación DELETE.

En la segunda (deleted), disponible en las operaciones UPDATE y DELETE, están los valores anteriores a la ejecución de la actualización o borrado. Es decir, los datos que serán borrados. Deleted estará vacía en una operación INSERT.

¿No existe una tabla UPDATED?

No, hacer una actualización es lo mismo que borrar (deleted) e insertar los nuevos (inserted).

La sentencia UPDATE es la única en la que inserted y deleted tienen datos simultáneamente.

No se pueden modificar directamente los datos de estas tablas.

El siguiente ejemplo, graba un histórico de saldos cada vez que se modifica un saldo de la tabla cuentas.

```
CREATE TRIGGER TR_CUENTAS
ON CUENTAS
AFTER UPDATE
AS
BEGIN
-- SET NOCOUNT ON impide que se generen mensajes de texto
-- con cada instrucción
SET NOCOUNT ON;
INSERT INTO HCO_SALDOS(IDCUENTA, SALDO, FXSALDO)
    SELECT IDCuenta, SALDO, getdate()
    FROM INSERTED
END
```

Una consideración a tener en cuenta es que el trigger se ejecutará aunque la instrucción DML (UPDATE, INSERT o DELETE) no haya afectado a ninguna fila. En este caso inserted y deleted devolverán un conjunto de datos vacío.

Otro ejemplo, modificamos el trigger para decirle a qué columna debe afectar

```
ALTER TRIGGER TR_CUENTAS
ON CUENTAS
AFTER UPDATE
AS
BEGIN

IF UPDATE(SALDO) -- Solo si se actualiza SALDO
BEGIN

INSERT INTO HCO_SALDOS (IDCUENTA, SALDO, FXSALDO)
    SELECT IDCuenta, SALDO, getdate()
    FROM INSERTED

END

END
```


Hasta el momento hemos aprendido que un trigger se crea sobre una tabla específica para un evento (inserción, eliminación o actualización).

También podemos especificar el momento de disparo del trigger.

El momento de disparo indica que las acciones (sentencias) del trigger se ejecuten luego de la acción (insert, delete o update) que dispara el trigger o en lugar de la acción.

La sintaxis para ello es:

```
create trigger NOMBREDISPARADOR  
on NOMBRETABLA  
MOMENTO DE DISPARO after o instead of  
ACCION insert, update o delete as  
SENTENCIAS
```

Entonces, el momento de disparo especifica cuando deben ejecutarse las acciones (sentencias) que realiza el trigger.

Puede ser "después" (after) o "en lugar de" (instead of) del evento que lo dispara.

Si no especificamos el momento de disparo en la creación del trigger, por defecto se establece como "after"

Hasta el momento, todos los disparadores que creamos han sido "after"

Los disparadores "instead of" se ejecutan en lugar de la acción desencadenante, es decir, cancelan la acción desencadenante (suceso que disparó el trigger) reemplazándola por otras acciones.

"instead of": sobrescribe la acción desencadenadora del trigger.

Se puede definir solamente un disparador de este tipo para cada acción (insert, delete o update) sobre una tabla o vista.

Sintaxis:

```
create trigger NOMBREDISPARADOR  
on NOMBRETABLA o VISTA  
instead of  
ACCION insert, update o delete as  
SENTENCIAS
```

Por ejemplo:

Forzar a que siempre se ingrese el nombre de la tabla Shippers en letras mayúsculas

```
CREATE TRIGGER trgInsert
```

```
    ON Shippers
    INSTEAD OF insert
AS
DECLARE
    @CompanyName nvarchar(40),
    @Phone nvarchar(24)
BEGIN
    SELECT @CompanyName=CompanyName,@Phone=Phone
    FROM inserted
    INSERT INTO Shippers VALUES(UPPER(@CompanyName),@Phone)
END
```

Triggers con afectación de inserciones múltiples

Muchas veces los triggers se ejecutan cuando existen actualizaciones, borrados o inserciones de múltiples registros, estos casos siempre deben ser contemplados.

Por ejemplo, dadas las siguientes tablas:

EMPLEADOS(ci, nombre, teléfono)

SUELDOS(recibo, fecha, importe, ci)

Quiero borrar un empleado, por lo tanto antes debo borrar los sueldos que tenga

Para el borrado de un único empleado

```
CREATE TRIGGER BorrarEmpleado
    ON Empleados
    INSTEAD OF delete
AS
DECLARE
    @ci numeric(8)
BEGIN
    SELECT @ci=ci
    FROM deleted

    DELETE
    FROM sueldos
    WHERE ci = @Ci

    DELETE
    FROM empleados
    WHERE ci = @Ci

END
```

La instrucción que dispara el trigger anterior sería:

```
DELETE
FROM empleados
WHERE ci=28574325
```

Pero que ocurre si la sentencia a utilizar es la siguiente:

```
DELETE
FROM empleados
```

Esto pediría borrar TODOS los empleados, por lo tanto el trigger daría un error al tratar de asignar la variable @ci

Por eso es necesario reescribir el trigger teniendo en cuenta esa posibilidad:

```
CREATE TRIGGER BorrarEmpleado
ON Empleados
INSTEAD OF delete
AS
BEGIN

DELETE
FROM susedos
WHERE ci IN (SELECT ci FROM deleted)

DELETE
FROM empleados
WHERE ci IN (SELECT ci FROM deleted)

END
```

Sintaxis en PL/SQL

Antes de pasar a los ejercicios de disparadores con T-SQL, vamos a ver algunos ejemplos de creación de triggers con PL/SQL de Oracle.

Sintaxis:

```
CREATE OR REPLACE TRIGGER nombre-del-trigger
  [FOLLOWS nombre-otro-trigger]
  [BEFORE/AFTER]
  [INSERT/DELETE/UPDATE/UPDATE OF columnas]
ON nombre-tabla
  [FOR EACH ROW/FOR EACH STATEMENT]
  [WHEN {condiciones}]
  {Bloque estándar de sentencias PL/SQL... BEGIN

EXCEPTION

END;
```

Diferencia entre UPDATE y UPDATE OF lista-columnas

En el primer caso el trigger se ejecuta ante la modificación de cualquier campo de la tabla

En el segundo caso, el trigger se ejecuta sólo cuando se modifica alguna de las columnas de la lista

Cláusula WHEN

Determina que el trigger PL/SQL se dispare sólo para los registros que cumplen la condición de la cláusula.

Esta cláusula sólo se puede usar en triggers a nivel de registro.

Ejemplo:

```
CREATE TRIGGER tr1_empleados
BEFORE INSERT OR UPDATE OF salario
ON empleados
FOR EACH ROW
WHEN (:new.salario > 5000);
BEGIN
    UPDATE empleados
    SET salario = 5000
    WHERE empleado_id = :new.empleado_id;
END;
```

En este ejemplo, si insertamos o actualizamos el registro de un empleado de manera que su salario sea superior a 5000 , el trigger PL/SQL actualizará dicho salario al valor de 5000 independientemente del salario que nosotros hayamos insertado o introducido al modificar el registro.

Note que en PL/SQL no existe el concepto de tablas inserted y deleted, para ello existen las variables :new y :old

Registros :old y :new

Un disparador con nivel de fila se ejecuta en cada fila en la que se produce el suceso.

Las variable :old y :new nos permiten acceder a los datos de la fila actual

Ejemplos :

Cargar una tabla de catalogo cada vez que se ingresa un empleado

```
CREATE OR REPLACE TRIGGER Disparador3
AFTER INSERT ON Empleados
FOR EACH ROW
BEGIN
INSERT INTO Bitacora VALUES
(Secuencia_Bitacora.NEXTVAL, :new.Empleado, SYSDATE);
END Disparador3;
```

Cargar en una bitácora que empleado se borra

```
CREATE OR REPLACE TRIGGER Disparador4
BEFORE DELETE ON Empleados
FOR EACH ROW
BEGIN
INSERT INTO Bitacora VALUES
(Secuencia_Bitacora.NEXTVAL, :old.Empleado, SYSDATE);
END Disparador3;
```

Cargar en una bitácora que empleado se se modificó, guardando el valor anterior y el valor actual

```
CREATE OR REPLACE TRIGGER Disparador5
BEFORE UPDATE ON Empleados
FOR EACH ROW
BEGIN
INSERT INTO Bitacora VALUES
(Secuencia_Bitacora.NEXTVAL,
:old.Empleado,
:new.Empleado,
SYSDATE);
END Disparador3;
```

Definir un control antes de ingresar un registro en la tabla cliente_cuenta que nunca permita ingresar una fecha de apertura mayor a la fecha del día, de ser así siempre cargar la fecha del día.

```
CREATE OR REPLACE TRIGGER trigg_fecha
BEFORE INSERT ON cliente_cuenta
BEGIN

    IF :new.fchApertura > SYSDATE THEN
        :new.fchApertura := SYSDATE
    END IF;

END;
```

Ejercicios utilizando el problema de las “Válvulas” (aplicando T-SQL)

Para resolver alguno de los ejercicios planteados crear una tabla Auditoria con la siguiente estructura:

```
CREATE TABLE auditoria(idAudit NUMERIC(6) IDENTITY(1,1),
    fchAudit DATETIME not null,
    tipoAudit VARCHAR(20) not null,
    dscAudit VARCHAR(200),
    CONSTRAINT PK_Audit PRIMARY KEY(idAudit),
    CONSTRAINT CK_TipAudit
    CHECK(tipoAudit IN('INGRESO','MODIFICACION','BORRADO')))
```

1. Antes de ingresar un nuevo registro a la tabla Dimensiones, forzar a que su descripción esté siempre en mayúsculas.
2. Crear un disparador que cada vez que se exista un movimiento sobre la tabla arrienda se cree un registro de auditoría indicando dicho cambio.
3. Crear un disparador que ante el ingreso de un nuevo número de serie de una válvula verifique si la válvula tiene otras válvulas compatibles, de ser así poner un aviso con la cantidad de válvulas compatibles que existen.
4. Crear un disparador que luego de ingresada una solicitud de inspección verifique que la inspección de dicha solicitud corresponda a válvulas con fecha de arrendamiento vigente, de no ser así generar un registro en la bitácora.
5. Crear un disparador que ante la actualización del precio de un arrendamiento, verifique que la petroquímica de dicho arrendamiento no tenga arrendamientos vencidos con precio 0, de ser así mostrar un aviso y generar un registro en la bitácora.
6. Crear un disparador que no permita registrar la presión a las válvulas que son del tipo Aguja
7. Crear un disparador que ante el ingreso de una válvula compatible verifique que no exista una compatibilidad inversa de dicha válvula

Soluciones:

1.

```
CREATE TRIGGER InsertDimension
ON dimensiones
INSTEAD OF insert
AS
DECLARE
@dscDim varchar(30)

BEGIN
    SELECT @dscDim=dscDim
    FROM inserted
    /* inserto el registro pero transformando a mayúscula la descripcion */
    INSERT INTO dimensiones VALUES(UPPER(@dscDim))
END
```

2.

```
CREATE TRIGGER InsertArrenda
ON arrenda
AFTER insert,delete,update
AS
DECLARE
@ins int,@del int,@tipo character(1)

BEGIN

/* identifico que operacion es INSERT/UPDATE/DELETE */
SELECT @ins=COUNT(*)
FROM inserted

SELECT @del=COUNT(*)
FROM deleted

SET @tipo = (CASE
    /* si las tablas inserted y deleted tienen datos es UPDATE */
    WHEN @ins+@del=2 THEN 'U'
        /* si la tabla inserted tiene datos y deleted no es INSERT */
    WHEN @ins = 1 AND @del =0 THEN 'I'
        /* si la tabla inserted no tiene datos y deleted si es DELETE */
    WHEN @ins = 0 AND @del =1 THEN 'D'
    ELSE 'X'
END)

IF @tipo='U'
BEGIN

INSERT INTO auditoria VALUES(getdate(),@tipo,'Se modifica un registro en ARRIENDA')
```

END

IF @tipo='D'
BEGIN

INSERT INTO auditoria VALUES(getdate(),@tipo,'Se borra un registro en ARRIENDA')

END

IF @tipo='I'
BEGIN

INSERT INTO auditoria VALUES(getdate(),@tipo,'Se ingresa un registro en ARRIENDA')

END

END

3.

CREATE TRIGGER InsertSerie
ON series
AFTER insert
AS
DECLARE

@codValv character(6),
@cant int

BEGIN

SELECT @codValv=codValv
FROM inserted

SELECT @cant=count(*)
FROM compatible
WHERE codvalv=@codValv or compvalv=@codValv

IF @cant > 0
PRINT 'La valvula ' + @codValv + ' tiene ' + str(@cant)
+ 'valvulas compatibles'

END

4.

```
CREATE TRIGGER InsertSolicitud
ON solicitan
AFTER insert
AS
DECLARE

@numInsp numeric(6),
@cant int

BEGIN

SELECT @numInsp=numInsp
FROM inserted

SELECT @cant=COUNT(*)
FROM arrienda
WHERE finFch < getdate() and
      codValv IN (SELECT codValv
                  FROM inspecciones
                  WHERE numInsp=@numInsp)

IF @cant > 0
    INSERT INTO auditoria
    VALUES(getdate(),'INGRESO','Hay valvulas con arrendamiento vencido')

END
```

5.

```
CREATE TRIGGER trgVencido
ON arrienda
AFTER INSERT
AS
BEGIN
DECLARE
@cant int,
@numSerie char(20),
@codValv char(6),
@rutPet char(13)

SELECT @numSerie=numSerie,@codValv=codValv,@rutPet=rutPet
FROM inserted

SELECT @cant=COUNT(*)
FROM arrienda
WHERE finfch < getdate() and
```

```

precio=0 and rutPet = @rutPet

IF @cant > 0
BEGIN
    PRINT 'La Petroquímica '+@rutPet+
        ' tiene arrendamientos vencidos con importe 0'
    INSERT INTO auditoria
    VALUES(getdate(),
        'INSERT', 'La Petroquímica '+@rutPet+
        ' tiene arrendamientos vencidos con importe 0')

END
END

```

6.

```

CREATE TRIGGER trgValvulas
ON valvulas
INSTEAD OF INSERT, UPDATE
AS
BEGIN
    DECLARE
        @tipoValv char(1),
        @PresValv numeric(10,2)

    SELECT @tipoValv=tipoValv, @presValv=presValv
    FROM inserted

    IF @tipoValv='A'
        SET @presValv=null

    INSERT INTO valvulas SELECT codValv, dscValv, diamValv, @tipoValv, @presValv, porcValv FROM
    inserted

END

```

7.

```
CREATE TRIGGER trgCompatible
ON compatible
INSTEAD OF INSERT
AS
BEGIN
DECLARE
    @codValv char(6),
    @compValv char(6),
    @cant int

    SELECT @codValv=codValv,@compValv=compValv
    FROM inserted

    SELECT @cant=COUNT(*)
    FROM compatible
    WHERE codValv=@compValv AND
           compValv=@codValv

    IF @cant > 0
        PRINT 'Ya existe una compatibilidad para la pareja '+@codValv+' - '+@compValv
    ELSE
        INSERT INTO compatible SELECT * FROM inserted

END
```

Vistas

Una vista es una consulta, que refleja el contenido de una o más tablas, desde la que se puede acceder a los datos como si fuera una tabla.

Dos son las principales razones por las que podemos crear vistas.

- a. Seguridad, nos pueden interesar que los usuarios tengan acceso a una parte de la información que hay en una tabla, pero no a toda la tabla.
- b. Comodidad, como hemos dicho el modelo relacional no es el más cómodo para visualizar los datos, lo que nos puede llevar a tener que escribir complejas sentencias SQL, tener una vista nos simplifica esta tarea.

Las vistas no tienen una copia física de los datos, son consultas a los datos que hay en las tablas, por lo que si actualizamos los datos de una vista, estamos actualizando realmente la tabla, y si actualizamos la tabla estos cambios serán visibles desde la vista.

Nota: No siempre podremos actualizar los datos de una vista, dependerá de la complejidad de la misma (dependerá de si el conjunto de resultados tiene acceso a la clave principal de la tabla o no), y del gestor de base de datos. No todos los gestores de bases de datos permiten actualizar vistas.

Sintaxis:

```
CREATE VIEW nombreVista  
AS  
Sentencias SQL;
```

Ejemplo:

Crear una vista que muestre las descripciones de las válvulas que son compatibles entre si

```
CREATE VIEW tCompatible
AS
SELECT A.dscValv as codigo,B.dscValv as compat
FROM valvulas A,valvulas B,compatible
WHERE compatible.codValv=A.codValv AND
       compatible.CompValv=B.codValv
```