



UNIVERSIDAD AUTONOMA DEL ESTADO DE

MEXICO

CU. TEXCOCO

Elaboración de apuntes de:
BASE DE DATOS y SQL Avanzado

PRESENTA:
Dra. Alma Delia Cuevas Rasgado

Texcoco, Estado de México.

Febrero de 2015

Resumen

Se presenta un compendio de los temas que forman la materia Fundamentos de Base de datos para las Licenciaturas: Ing. en Computación e Informática Administrativa, así como una parte de Base de Datos Avanzadas de la Ing. en Computación de la Licenciatura en Informática impartida en la Universidad Autónoma del Estado de México, Centro Universitario Texcoco..

Las unidades de aprendizaje forman parte del conjunto de unidades básicas en la formación profesional en cómputo. Cada uno de los objetos de estudio que se desarrollan en este documento regularmente lleva el orden planteado en los temarios y que ha sido desarrollado específicamente para que los alumnos de la universidad satisfagan eficientemente las demandas de conocimiento de nuestros Estados en vías de desarrollo como lo es el Estado de México.

Se considera importante que al estudiar estos apuntes, el alumno tenga una idea de lo que son las Base de datos, el lenguaje SQL y el uso de, al menos, un manejador de base de datos relacional comercial. Por razones de escasez de tiempo en la elaboración de este documento, se presentan casos prácticos usando solo el manejador de base de datos SQL Server de Microsoft, con la firme intención de que; en lo posterior, se presente una mejora con casos aplicados a varios manejadores con la finalidad de estudiar el comportamiento de las instrucciones en otras herramientas. Por lo tanto, se deduce que en ese sentido, este trabajo no pretende ser completo pero si apoyar en un importante grado al conocimiento de herramientas avanzadas, para que el alumno pueda aplicarlos a las nuevas tecnologías emergentes de base de datos.

Introducción

El documento presenta cinco unidades interesantes y cuatro anexos. La unidad uno inicia con el estudio del Álgebra Relacional como un importante soporte matemático de las bases de datos relacionales, cada uno de estos temas ordenados y desarrollados se demuestran de manera práctica con ejemplos aplicados a consultas hechas en este lenguaje matemático.

En la unidad dos, se presenta el lenguaje SQL con temas que frecuentemente se usarán a lo largo de este documento y que son partes fundamentales para la creación de consultas avanzadas de base de datos, al igual que en las unidades subsecuentes, cada tema se demuestra teóricamente mediante ejemplos aplicados a Microsoft SQL Server.

La creación e implementación de las vistas se estudian en la unidad tres, como una importante herramienta de creación de tablas virtuales útiles en la funcionalidad de los sistemas de información.

La unidad cuatro presenta el estudio de los disparadores o desencadenadores que le dan soporte al diseño eficiente de base de datos relacionales.

Los procedimientos almacenados se estudian en la unidad cinco, como importantes fragmentos de código que abarcan: consultas y búsquedas con instrucciones de transacción en la funcionalidad de las bases de datos.

El anexo A presenta un modelo de entidad-relación de las cuales se apoyan los temas de los capítulos del dos al cinco. El anexo B presenta las instrucciones SQL para la creación de cada una de las tablas del modelo. El anexo C presenta el contenido de las tablas del modelo, con la finalidad de contar con los datos necesarios para la aplicación de los ejercicios y el Anexo D abarca la explicación y ejercicios de algunas instrucciones básicas de SQL, con la finalidad de que el alumno lo consulte cuando sea necesario.

La bibliografía presenta las fuentes de información de donde surgen las ideas de los temas de este documento, en unos casos de manera directa y en otros casos han sido adaptados para conservar la coherencia de la redacción. En ella, se encuentran los libros completos y explícitos recomendados en el temario de la materia [Elmasri, R. y S. B. Navathe, 2002] y [Peter Rob. Carlos Coronel, 2004] y se han adicionado otros libros que aportan en mucho al conocimiento del alumno. Existen artículos que son fuente importante en el soporte de las bases de datos, tales como [Cood E. F., 1983] y [Codd, E. F., 1987] que se recomienda leer si se desea profundizar o especializar esta área de conocimiento y finalmente algunas referencias de Internet de donde se han extraído ideas y ejemplos importantes considerados en los temas de este trabajo, así mismo, se recomienda la lectura de estos manuales, tutoriales y cursos si existe el interés de especializarse en estos tópicos.

Amén de todo lo anterior, este compendio de temas de instrucciones de SQL Avanzado intenta cubrir también los requerimientos de aprendizaje de aquellas personas autodidactas que consideren de gran utilidad el profundizar el conocimiento del lenguaje SQL aplicado a las bases de datos relacionales.

Índice

1.	Álgebra relacional	6
1.1.	Introducción al Álgebra Relacional.....	6
1.2.	Las operaciones básicas del Álgebra relacional.....	8
1.3.	El conjunto de operaciones de la teoría matemática de conjuntos.....	8
1.3.1.	Unión	8
1.3.2.	Intersección	9
1.3.3.	Diferencia.....	10
1.3.4.	Producto	10
1.4.	Las operaciones creadas específicamente para bases de datos relacionales	12
1.4.1.	Selección o Restricción.....	12
1.4.2.	Proyección.....	14
1.4.3.	Reunión	15
1.4.4.	División.....	18
1.5.	Secuencia de operaciones y cambio de nombre de los atributos	19
1.6.	Conjunto completo de operaciones del álgebra relacional	21
1.7.	Operaciones de cerradura recursiva.....	21
1.8.	Funciones agregadas.....	22
1.9.	Otras operaciones adicionales	23
1.9.1.	Ampliación.....	23
1.9.2.	Resumen.....	23
1.9.3.	División generalizada	24
1.9.4.	Reunión externa.....	25
1.9.5.	Unión externa	27
1.10.	Operaciones quizá.....	28
1.11.	Asignación relacional	29
1.12.	Ejemplos de consultas en el álgebra relacional	29
2.	El lenguaje Sql.....	32
2.1.	Estructura general de las consultas en SQL	36
2.2.	Combinación de tablas.....	37
2.3.	Predicados IN, IS NULL, BETWEEN, AND, OR, NOT, LIKE	45
2.4.	Cláusulas ORDER BY, GROUP BY	49
2.5.	Funciones agregadas COUNT, SUM, MAX, MIN, AVG.....	55
2.6.	Subconsultas.....	59
2.7.	Operadores IN, EXIST, ANY, ALL aplicados a subconsultas de renglón múltiple	62
2.8.	Cláusula HAVING (CON).....	66
2.9.	Combinaciones externas: OUTER JOIN, UNION JOIN	68
2.10.	El valor NULL.....	71
3.	Vistas	73
3.1.	Definición	73
3.2.	La opción WITH CHECK OPTION	78
3.3.	Operaciones DML sobre las vistas	79
3.4.	Funcionamiento de las vistas	80
3.5.	Utilización de las vistas	81
3.6.	Actualización de vistas	82
4.	Disparadores	93

4.1.	Definición	93
4.2.	Estado de los desencadenadores.....	94
4.3.	Tipos de desencadenadores	94
4.3.1.	AFTER	94
4.3.2.	INSTEAD OF.....	95
4.3.3.	For each row, statement	101
4.3.4.	IF UPDATE.....	101
4.4.	Grupos de desencadenadores	104
4.4.1.	Desencadenador de inserción	104
4.4.2.	Desencadenador de actualización.....	106
4.4.3.	Desencadenador de eliminación.....	107
4.5.	Desencadenadores anidados.....	109
4.6.	Desencadenadores recursivos.....	110
4.7.	Implementación de desencadenadores.....	111
4.8.	Usos, ventajas y desventajas	113
5.	Procedimientos almacenados	116
5.1.	Definición	116
5.2.	Algoritmo de ejecución.....	117
5.3.	Usos	121
5.4.	Ventajas	127
5.5.	Desventajas	127
Anexo A	128
Anexo B	129
Anexo C	137
Anexo D	145
Bibliografía	158

1. Álgebra relacional

1.1. Introducción al Álgebra Relacional

El álgebra relacional consiste en un conjunto de operadores de alto nivel que trabajan sobre relaciones. Cada uno de estos operadores toma una o dos relaciones como entrada y produce una nueva relación como salida. Codd¹ definió un conjunto muy específico de ocho operadores de este tipo, en dos grupos de cuatro cada uno:

1. las operaciones tradicionales de conjuntos *unión*, *intersección*, *diferencia* y *producto cartesiano* (todas ellas con ligeras modificaciones debidas al hecho de tener relaciones como operandos y no conjuntos arbitrarios; después de todo, una relación es un tipo especial de conjunto)
2. las operaciones relacionales especiales *restricción o selección*, *proyección*, *reunión* y *división*.

Las cuatro primeras se toman de la teoría de conjunto de las matemáticas; las cuatro siguientes son operaciones propias del álgebra relacional y la última es la operación estándar *Asignación* proporciona un valor a un elemento.

Los 8 operadores originales se representan en forma simbólica en la figura 2.

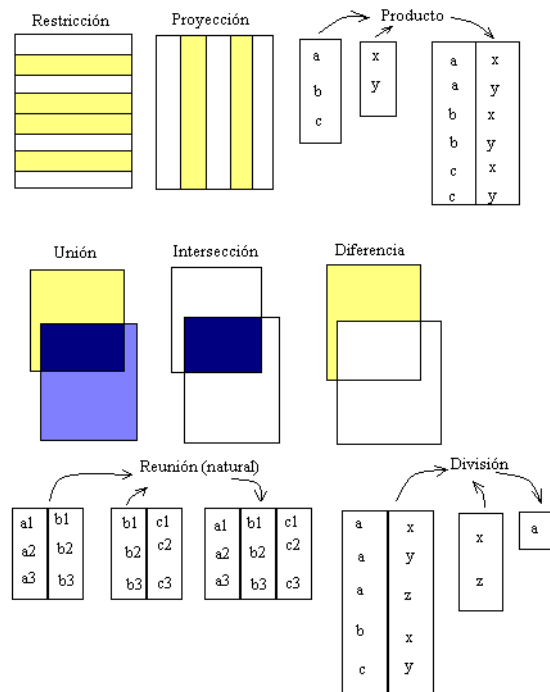


Figura 2 Representación gráfica de las operaciones originales de conjuntos

¹ Científico informático inglés (23 de agosto de 1923 - 18 de abril de 2003), conocido por sus aportes a la teoría de bases de datos relacionales. Más en: http://es.wikipedia.org/wiki/Edgar_Frank_Codd

¿Para qué sirve el álgebra relacional?

El objetivo principal del álgebra relacional no sólo es la obtención de datos. La intención fundamental del álgebra es *ayudar a escribir expresiones*. A su vez, la intención de esas expresiones es *coadyuvar a diversos propósitos*, entre los que está incluida desde luego *la obtención de datos*, pero de ninguna manera están limitados a esa única función. La siguiente lista indica algunas posibles aplicaciones de tales expresiones (en esencia, por supuesto, tales expresiones representan *relaciones* y esas relaciones a su vez definen el alcance de las operaciones de recuperación, actualización, etc. presentadas en la lista).

1. *definir el alcance de una recuperación*; es decir, definir los datos que se van a extraer como resultado de una recuperación.
2. *definir el alcance de una actualización*; es decir, definir los datos por insertar, modificar o eliminar como resultado de una operación de actualización
3. *definir datos virtuales*; es decir, definir los datos que se podrán ver en forma de relación virtual o vista
4. *definir datos de instantánea*; es decir, definir los datos que se han de mantener en forma de una relación tipo “instantánea”
5. *definir derechos de acceso*; es decir, definir los datos incluidos en algún tipo de autorización concedida.
6. *definir requerimientos de estabilidad*; es decir, definir los datos que abarcará alguna operación de control de concurrencia.
7. *definir restricciones de integridad*; es decir, definir alguna regla específica que debe satisfacer la BD, además de las reglas generales (o metarreglas) que son parte del modelo relacional y se aplican a todas las BD.

En general, la realidad es que las expresiones constituyen una *representación simbólica de alto nivel de la intención del usuario* (con respecto a una consulta determinada, por ejemplo) y precisamente porque son de alto nivel y simbólicas se pueden manipular de acuerdo con varias *reglas de transformación* simbólicas de alto nivel. Por ejemplo, la expresión:

$$((S \text{ JOIN } SP) \text{ WHERE } P\# = 'P2') [\text{SNOMBRE}]$$

(“nombres de los proveedores que suministran la parte 2”) puede transformarse en una expresión equivalente en cuanto a la lógica pero con toda probabilidad más eficiente:

$$(S \text{ JOIN } (SP \text{ WHERE } P\# = 'P2')) [\text{SNOMBRE}]$$

Por lo tanto, el álgebra constituye una base conveniente para la *optimización*; es decir, aunque el usuario exprese la consulta empleando la primera de las dos expresiones anteriores, el optimizador deberá convertirla en la segunda antes de ejecutarla (el desempeño de una consulta dada no deberá depender de la forma en la cual la exprese el usuario).

El álgebra se utiliza a menudo como *patrón de referencia* para medir la capacidad expresiva de un determinado lenguaje relacional (por ejemplo, SQL). En esencia, se dice que un lenguaje es *relacionalmente completo* si es tan expresivo como el álgebra, cuando menos; es decir, si sus expresiones permiten la definición de cualquier relación que pueda definirse mediante expresiones del álgebra.

1.2. Las operaciones básicas del Álgebra relacional

El Álgebra relacional consta de operaciones que le dan al usuario la libertad de realizar peticiones de recuperación básicos de datos. Como resultado de esta recuperación es una nueva relación que se forma a partir de una o más relaciones, mismas que se pueden volver a manipular para obtener otras relaciones y así sucesivamente.

Las operaciones del Álgebra relacional se clasifican en dos grupos:

1. el conjunto de operaciones de la teoría matemática de conjuntos y
2. las operaciones creadas específicamente para bases de datos relacionales

1.3. El conjunto de operaciones de la teoría matemática de conjuntos

Las operaciones que normalmente se aplican a los conjuntos son: *Unión*, *Intersección*, *Diferencia* y *Producto*. En este apartado se definirán cada uno de ellos.

1.3.1. Unión

Construye una relación formada por todas las tuplas que aparecen en cualquiera de las dos relaciones especificadas.

La unión de dos relaciones A y B compatibles respecto a la unión, A UNION B, es una relación cuya cabecera es idéntica a la de A o B y cuyo cuerpo está formado por todas las tuplas t pertenecientes ya sea a A o a B (o a las dos).

La operación de unión permite combinar datos de varias relaciones. Supongamos que una determinada empresa internacional posee una tabla de empleados para cada uno de los países en los que opera. Para conseguir un listado completo de todos los empleados de la empresa tenemos que realizar una unión de todas las tablas de empleados de todos los países.

No siempre es posible realizar consultas de unión entre varias tablas, para poder realizar esta operación es necesario e imprescindible que las tablas a unir tengan las mismas estructuras, que sus campos sean iguales.

Ejemplo:

Sean A y B las relaciones presentadas en la figura 3 (A contiene en términos intuitivos, los proveedores de Londres y B contiene los proveedores que suministran la parte P1). Entonces A UNION B consistirá en los proveedores que *o bien* están situados en Londres, o que suministran la parte P1 (o las dos cosas). Adviértase que el resultado tiene tres tuplas, no cuatro (se eliminan las tuplas repetidas).

A				B			
S#	SNOMBRE	SITUACIÓN	CIUDAD	S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres	S1	Salazar	20	Londres
S4	Corona	20	Londres	S2	Jaimés	10	París

A UNIÓN B

S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres
S4	Corona	20	Londres
S2	Jaimés	10	París

Figura 3 Ejemplo de Unión

1.3.2. Intersección

Construye una relación formada por aquellas que aparezcan en las dos relaciones especificadas.

La operación de intersección permite identificar filas que son comunes en dos relaciones. Supongamos que tenemos una tabla de empleados y otra tabla con los asistentes que han realizado un curso de inglés (los asistentes pueden ser empleados o gente de la calle). Queremos crear una figura virtual en la tabla denominada "Empleados que hablan Inglés", esta figura podemos crearla realizando una intersección de empleados y curso de inglés, los elementos que existan en ambas tablas serán aquellos empleados que han asistido al curso.

Ejemplo:

Sean A y B las relaciones presentadas en la figura 4 (A contiene en términos intuitivos, los proveedores de Londres y B contiene los proveedores que suministran la parte P1). Entonces A INTERSECT B consistirá en los proveedores que son comunes a ambas relaciones.

A				B			
S#	SNOMBRE	SITUACIÓN	CIUDAD	S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres	S1	Salazar	20	Londres
S4	Corona	20	Londres	S2	Jaimés	10	París

A INTERSECT B

S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres

Figura 4 Ejemplo de intersección

1.3.3. Diferencia.

Construye una relación formada por todas las tuplas de la primera relación que no aparezcan en la segunda de las dos relaciones especificadas.

La operación diferencia permite identificar filas que están en una relación y no en otra. Tomando como referencia el caso anterior, deberíamos aplicar una diferencia entre la tabla empleados y la tabla asistentes al curso para saber aquellos asistentes externos a la organización que han asistido al curso.

La diferencia entre dos relaciones compatibles respecto a la unión A y B, A MINUS B, es una relación cuya cabecera es idéntica a la de A o B y cuyo cuerpo está formado por todas las tuplas t pertenecientes a A pero no a B.

Ejemplo:

Sean A y B las relaciones presentadas en la figura 5 (A contiene en términos intuitivos, los proveedores de Londres y B contiene los proveedores que suministran la parte P1). Entonces A MINUS B consistirá en los proveedores que están en A y no en B. Mientras que B MINUS A consistirá en los proveedores que están en B y no en A.

A			
S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres
S4	Corona	20	Londres

B			
S#	SNOMBRE	SITUACIÓN	CIUDAD
S1	Salazar	20	Londres
S2	Jaimes	10	París

A MINUS B			
S#	SNOMBRE	SITUACIÓN	CIUDAD
S4	Corona	20	Londres

B MINUS A			
S#	SNOMBRE	SITUACIÓN	CIUDAD
S2	Jaimes	10	París

Figura 5 Ejemplo de diferencia

1.3.4. Producto

A partir de dos relaciones especificadas, construye una relación que contiene todas las combinaciones posibles de tuplas, una de cada una de las dos relaciones.

La operación producto consiste en la realización de un producto cartesiano entre dos tablas dando como resultado todas las posibles combinaciones entre los registros de la primera y los registros de la segunda. Esta operación se entiende mejor con el siguiente ejemplo:

TABLA A	
X	Y
10	22
11	25

TABLA B	
W	Z
33	54
37	98
42	100

TABLA A * TABLA B			
10	22	33	54
10	22	37	98
10	22	42	100
11	25	33	54
11	25	37	98
11	25	42	100

Figura 6 Ejemplo del producto

Producto cartesiano ampliado

En matemáticas, el producto cartesiano de dos conjuntos es el conjunto de todos los pares ordenados de elementos tales que el primer elemento de cada par pertenece al primer conjunto y el segundo elemento de cada par pertenece al segundo conjunto. Así, el producto cartesiano de dos relaciones sería un conjunto de pares ordenados de tuplas. Pero (una vez más) deseamos conservar la propiedad de cerradura; en otras palabras deseamos un resultado compuesto de tuplas, no de pares ordenados de tuplas (aquí también empleamos términos un poco informales). Por lo tanto, la versión del producto cartesiano en álgebra relacional es una forma ampliada de la operación, en la cual cada par ordenado de tuplas es reemplazado por la tupla resultante de la combinación de las dos tuplas en cuestión. Aquí “combinación” significa en esencia unión (en el sentido de la teoría de conjuntos, no del álgebra relacional); es decir, dadas las dos tuplas

$$(A1:a1, A2:a2, \dots, Am:am) \text{ y } (B1:b1, B2:b2, \dots, Bn:bn)$$

(se muestran los nombres de los atributos para hacerlas más explícitas), la combinación de las dos es la tupla única

$$(A1:a1, A2:a2, \dots, Am:am, B1:b1, B2:b2, \dots, Bn:bn)$$

Un problema que surge en conexión con el producto cartesiano es la necesidad de una cabecera bien formada para la relación resultante. Ahora bien, es evidente que la cabecera del resultado es en esencia sólo la combinación de las cabeceras de las dos relaciones de entrada. Por tanto, se presentará un problema si esas dos cabeceras tienen algún nombre de atributo en común. Así pues, si necesitamos formar el producto cartesiano de dos relaciones cuyas cabeceras tienen nombres de atributo común, debemos emplear antes el operador RENAME (renombrar) para modificar de manera apropiada los nombres de los atributos. Entonces diremos que dos relaciones son *compatibles respecto al producto* si y solo si sus cabeceras son disjuntas (es decir, no tienen nombres de atributo en común).

Por lo tanto, definimos el producto cartesiano de dos relaciones (compatibles respecto al producto) A y B, A TIMES B, como *una relación cuya cabecera es la combinación de las cabeceras de A y B y cuyo cuerpo está formado por el conjunto de todas las tuplas de t tales que t es la combinación de una tupla de a perteneciente a A y una tupla b perteneciente a B.*

Ejemplo: Sean A y B las relaciones presentadas en la figura 7 (A, en términos intuitivos, consiste en todos los números de proveedores vigentes y B en todos los números de parte vigentes). Entonces A TIMES B estará formada por todas las combinaciones de número de proveedor/número de partes vigentes.

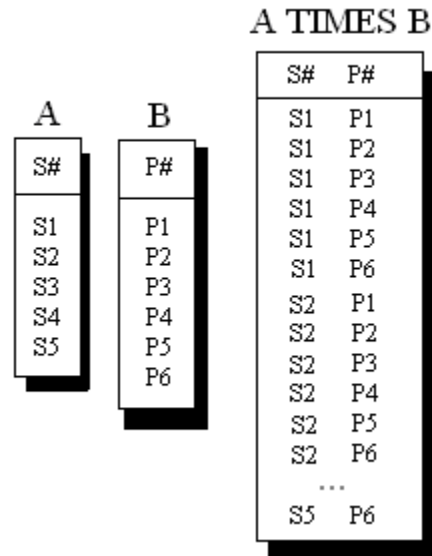


Figura 7 Ejemplo del producto cartesiano

No debemos terminar este análisis del producto cartesiano sin señalar que esta operación no tiene mucha importancia en la práctica. Es decir, no es importante en la práctica que un sistema relacional maneje esa operación² (aunque de hecho la mayor parte de ellos lo hacen). El producto cartesiano se incluye en el álgebra relacional sobre todo por razones conceptuales. En particular, el producto cartesiano es necesario como paso intermedio en la definición de la operación “reunión theta” (y esta última operación si es importante en la práctica).

1.4. Las operaciones creadas específicamente para bases de datos relacionales

Estas operaciones son: Seleccionar, Proyectar y Reunión. Las dos primeras son las más sencillas, mientras que la operación Reunión es mas compleja.

1.4.1. Selección o Restricción

También llamada Restricción, extrae las tuplas especificadas de una relación dada (o sea, restringe la relación sólo a las tuplas que satisfagan una condición especificada).

Se puede considerar la operación SELECCIONAR como un filtro que mantiene aquellas tuplas que satisfacen una condición de cualificación. Esta operación se denota por el símbolo σ . En general, esta operación se representa como:

$$\sigma_{\langle \text{condición de selección} \rangle}(\text{RELACIÓN O TABLA}).$$

Por ejemplo:

Para indicar que se debe seleccionar de la tabla EMPLEADO a aquellos cuyo departamento es el número 4, se representa como: $\sigma_{ND=4}(\text{EMPLEADO})$.

² la razón formal de la poca importancia del producto cartesiano es que *no hay más información en la salida que en la entrada*. En el caso de la figura 7, por ejemplo, el resultado no nos dice nada que no supieramos ya; sólo nos dice cuáles números de proveedor existen y cuáles números de parte existen

Sea theta la representación de cualquier operador de comparación escalar simple (por ejemplo =, <>, >, >=, etc). La restricción theta de la relación A según los atributos X y Y. A WHERE X theta Y es una relación con la misma cabecera que A y con un cuerpo formado por el conjunto de todas las tuplas t de A tales que la evaluación de la comparación X theta Y resulta verdadera en el caso de esa tupla t. (los atributos X y Y deben estar definidos sobre el mismo dominio y la operación theta debe ser aplicable a ese dominio. Además, por supuesto, la relación A no debe ser por fuerza una relación nombrada y puede representarse mediante una expresión arbitraria del álgebra relacional).

Se puede especificar un valor literal en vez del atributo X o del atributo Y (o de ambos, desde luego); de hecho, esto es lo más común en la práctica. Por ejemplo:

A WHERE X theta literal

Adviértase que el operador de restricción theta produce en realidad un subconjunto “horizontal” de una relación dada; es decir, el subconjunto de las tuplas de la relación dada para las cuales se satisface una comparación especificada. Se acostumbra abreviar “restricción theta” a sólo “restricción”.

La operación de restricción tal como se acaba de definir permite sólo una comparación simple en la cláusula WHERE; sin embargo es posible ampliar la definición que esté formada por una combinación booleana arbitraria de tales comparaciones simples, según se indica con las siguientes equivalencias:

1. A WHERE C1 AND C2 se define como equivalente a (A WHERE C1) INTERSECT (A WHERE C2)
2. A WHERE C1 OR C2 se define como equivalente a (A WHERE C1) UNION (A WHERE C2)
3. A WHERE NOT C se define como equivalente a A MINUS(A WHERE C)

La operación selección con restricciones consiste en recuperar un conjunto de registros de una tabla o de una relación indicando las restricciones que deben cumplir los registros recuperados, de tal forma que los registros devueltos por la selección han de satisfacer todas las condiciones que se hayan establecido. Esta operación es la que normalmente se conoce como consulta.

Se puede emplearla para saber que empleados son mayores de 45 años, o cuales viven en Madrid, incluso podemos averiguar los que son mayores de 45 años y residen en Madrid, los que son mayores de 45 años y no viven en Madrid, etc..

En este tipo de consulta se emplean los diferentes operadores de comparación (=,>, <, >=, <=, <>), los operadores lógicos (and, or, xor) o la negación lógica (not).

La expresión condicional en la cláusula WHERE de una restricción está formada por este tipo de combinaciones booleanas arbitrarias de comparaciones simples. Una expresión condicional como ésta se conoce como *condición de restricción*.

En la figura 8 se presentan algunos ejemplos de restricción.

S#	SNOMBRE	SITUACION	CIUDAD
S1	Salazar	20	Londres
S4	Corona	20	Londres

S WHERE CIUDAD = 'Londres'

P#	PNOMBRE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12	Londres
P5	Leva	Azul	12	Paris

P WHERE PESO < 14

S#	P#	CANT
S1	P1	300

SP WHERE S# = 'S1'
AND P# = 'P1'

Figura 8 Ejemplos de restricción

1.4.2. Proyección

Extrae los atributos especificados de una relación dada. Si se piensa en una relación como una tabla, la operación Selección restringe ciertas filas de una tabla y desecha las demás, mientras la Proyección selecciona ciertas columnas de la tabla desechando las restantes.

El Símbolo que denota esta operación es: π su representación general es:

$$\pi_{\langle \text{lista de atributos} \rangle} (\text{RELACIÓN O TABLA}).$$

El resultado de una operación Proyección contiene únicamente los atributos especificados en $\langle \text{lista de atributos} \rangle$ y en el mismo orden en que aparecen en la lista.

La proyección de la relación A según los atributos X, Y, ..., Z

A [X, Y, ..., Z] es una relación con (X, Y, ..., Z) como cabecera y cuyo cuerpo está formado por el conjunto de todas las tuplas (X:x, Y:y, ..., Z:z) tales que una tupla t aparece en A con el valor x en X, el valor y en Y, ... y el valor z en Z. Así, el operador de proyección produce un subconjunto “vertical” de una relación dada; o sea, el subconjunto obtenido mediante la selección de los atributos especificados y la eliminación de las tuplas repetidas dentro de los atributos seleccionados, la relación A tampoco necesita ser una relación renombrada y puede representarse mediante una expresión arbitraria.

Una proyección es un caso concreto de la operación selección, esta última devuelve todos los campos de aquellos registros que cumplen la condición establecida. Una proyección es una selección en la que seleccionamos aquellos campos que deseamos recuperar. Tomando como referencia el caso de la operación selección es posible que lo único que nos interese recuperar sea el número de la seguridad social, omitiendo así los campos teléfono, dirección, etc.. Este último caso, en el que seleccionamos los campos que deseamos, es una proyección.

La figura 9 presenta algunos ejemplos de proyección. Obsérvese en el inciso a (la proyección de proveedores según el atributo CIUDAD) que, aunque la relación S tiene cinco tuplas y por tanto cinco ciudades, sólo hay tres ciudades en el resultado; como ya se explicó, las tuplas repetidas se eliminan (como siempre). Desde luego, lo mismo puede decirse también de los otros ejemplos.

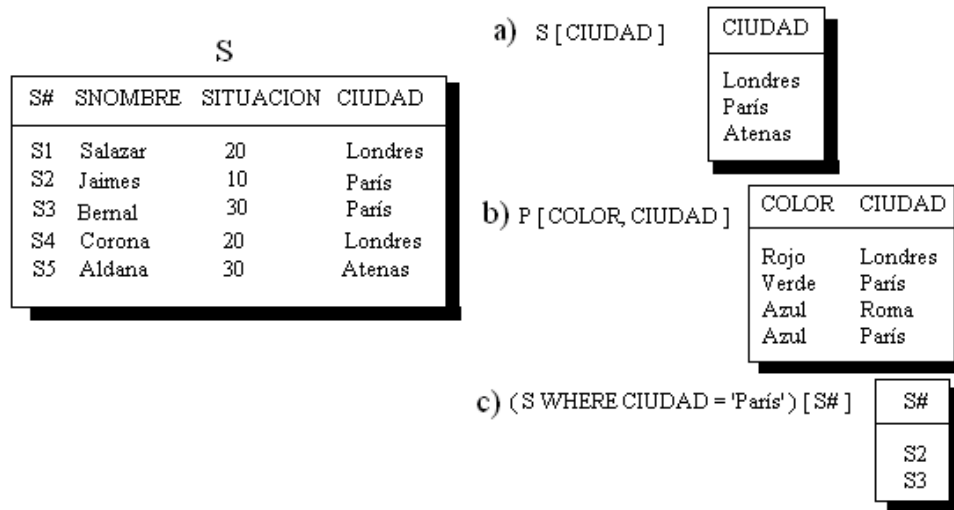


Figura 9. Ejemplo de Proyección

1.4.3. Reunión

Denotada en este documento por el símbolo \bowtie . Sirve para combinar tuplas relacionadas de dos relaciones en una sola tupla. Esta operación es muy importante en cualquier base de datos relacional que comprenda más de una relación, porque permite procesar los vínculos entre las relaciones. Con la finalidad de ilustrar la reunión. La forma de representarlo es:

$R1 \bowtie_{\langle \text{condición de reunión} \rangle} R2$

Donde: R1 y R2 son relaciones.

El resultado de la reunión es una relación R3 con el número de atributos de R1 y R2. R3 tiene una tupla por cada combinación de tuplas (una de R1 y una de R2) siempre que la combinación satisfaga la condición de reunión.

A partir de dos relaciones especificadas, construye una relación que contiene todas las posibles combinaciones de tuplas, una de cada una de las dos relaciones, tales que las dos tuplas participantes en una combinación dada satisfagan alguna condición especificada.

La reunión se utiliza para recuperar datos a través de varias tablas conectadas unas con otras mediante cláusulas JOIN, en cualquiera de sus tres variantes INNER, LEFT, RIGHT. La operación reunión se puede combinar con las operaciones selección y proyección.

Un ejemplo de reunión es conseguir los pedidos que nos han realizado los clientes nacionales cuyo importe supere 15.000 unidades de producto, generando un informe con el nombre del cliente y el código del pedido. En este caso se da por supuesto que la tabla clientes es diferente a la tabla pedidos y que hay que conectar ambas mediante, en este caso, un INNER JOIN.

Reunión natural

Denotada en este documento por el símbolo $*$.

La operación de reunión tiene varias formas distintas. Definitivamente la más importante, que se define como sigue:

Sean las cabeceras $(X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_n)$ y $(Y_1, Y_2, \dots, Y_n, Z_1, Z_2, \dots, Z_p)$ respectivamente; es decir, los atributos Y_1, Y_2, \dots, Y_n son (los únicos) comunes a las dos relaciones, los atributos X_1, X_2, \dots, X_m son los demás atributos de A, y los atributos Z_1, Z_2, \dots, Z_p son los demás atributos de B. Vamos a suponer también que los atributos correspondientes (es decir, los atributos con el mismo nombre) están definidos sobre el mismo dominio. Consideremos ahora (X_1, X_2, \dots, X_m) , (Y_1, Y_2, \dots, Y_n) y (Z_1, Z_2, \dots, Z_p) como tres atributos compuestos X, Y y Z, respectivamente. La reunión natural de A y B $A \text{ JOIN } B$ es una relación con la cabecera (X, Y, Z) y un cuerpo formado por el conjunto de todas las tuplas $(x:X, y:Y, z:Z)$ tales que una tupla a aparezca en A con el valor x en X y el valor y en Y, y una tupla b aparezca en B con el valor y en Y y el valor z en Z. Como siempre, las relaciones A y B pueden estar representadas por expresiones arbitrarias.

La reunión natural, es tanto asociativa como conmutativa.

$(A \text{ JOIN } B) \text{ JOIN } C$ y $A \text{ JOIN } (B \text{ JOIN } C)$ se pueden simplificar, sin provocar ambigüedad, además las dos expresiones equivalentes $A \text{ JOIN } B$ y $B \text{ JOIN } A$ son equivalentes.

Cabe señalar que, si A y B no tienen nombres de atributos en común, $A \text{ JOIN } B$ es equivalente a $A * B$ (es decir, la reunión natural degenera en el producto cartesiano, en este caso).

En la figura 10 se presenta un ejemplo de reunión natural (la reunión natural $S \text{ JOIN } P$, según el atributo común CIUDAD).

S				P				
S#	SNOMBRE	SITUACION	CIUDAD	P#	PNOMBRE	COLOR	PESO	CIUDAD
S1	Salazar	20	Londres	P1	Tuerca	Rojo	12	Londres
S2	Jaimes	10	París	P2	Perno	Verde	17	París
S3	Bernal	30	París	P3	Birlo	Azul	17	Roma
S4	Corona	20	Londres	P4	Birlo	Rojo	14	Londres
S5	Aldana	30	Atenas	P5	Leva	Azul	12	París
				P6	Engrane	Rojo	19	Londres

S JOIN P							
S#	SNOMBRE	SITUACION	CIUDAD	P#	PNOMBRE	COLOR	PESO
S1	Salazar	20	Londres	P1	Tuerca	Rojo	12
S1	Salazar	20	Londres	P4	Birlo	Rojo	14
S1	Salazar	20	Londres	P6	Engrane	Rojo	19
S2	Jaimes	10	París	P2	Perno	Verde	17
S2	Jaimes	10	París	P5	Leva	Azul	12
S3	Bernal	30	París	P2	Perno	Verde	17
S3	Bernal	30	París	P5	Leva	Azul	12
S4	Corona	20	Londres	P1	Tuerca	Rojo	12
S4	Corona	20	Londres	P4	Birlo	Rojo	14
S4	Corona	20	Londres	P6	Engrane	Rojo	19

Figura 10. Ejemplo de reunión natural

Reunión theta

Es adecuada para aquellas ocasiones (poco frecuentes en comparación, pero de ninguna manera desconocida) en las cuales necesitamos juntar dos relaciones con base en alguna condición diferente a la igualdad. Sean las relaciones A y B compatibles respecto al producto (o sea, no tienen nombres de atributos en común) y sea theta un operador según la definición dada en el análisis de la restricción. La reunión theta de la reunión A según el atributo X con la relación B según el atributo Y se define como el resultado de evaluar la expresión (A TIMES B) WHERE X theta Y

En otras palabras, es una relación con la misma cabecera que el producto cartesiano de A y B, y con el cuerpo formado por el conjunto de todas las tuplas t tales que t pertenece a ese producto cartesiano y la evaluación de la condición “X theta Y” resulta verdadera para esa tupla t. (Los atributos X y Y deberán estar definidos sobre el mismo dominio, y la operación theta debe ser aplicable a ese dominio).

Vamos a suponer, por ejemplo que deseamos calcular la “reunión mayor que de la relación S según CIUDAD con la relación P según CIUDAD”. Una expresión apropiada del álgebra relacional es:

```
((S RENAME CIUDAD AS SCIUDAD) * (P RENAME CIUDAD AS PCIUDAD))
WHERE SCIUDAD > PCIUDAD
```

La figura 11 presenta el resultado.

S				P				
S#	SNOMBRE	SITUACION	CIUDAD	P#	PNOMBRE	COLOR	PESO	CIUDAD
S1	Salazar	20	Londres	P1	Tuerca	Rojo	12	Londres
S2	Jaimes	10	París	P2	Perno	Verde	17	París
S3	Bernal	30	París	P3	Birlo	Azul	17	Roma
S4	Corona	20	Londres	P4	Birlo	Rojo	14	Londres
S5	Aldana	30	Atenas	P5	Leva	Azul	12	París
				P6	Engrane	Rojo	19	Londres

WHERE SCIUDAD > PCIUDAD								
S#	SNOMBRE	SITUACION	SCIUDAD	P#	PNOMBRE	COLOR	PESO	PCIUDAD
S2	Jaimes	10	París	P1	Tuerca	Rojo	12	Londres
S2	Jaimes	10	París	P4	Birlo	Rojo	14	Londres
S2	Jaimes	10	París	P6	Engrane	Rojo	19	Londres
S3	Bernal	30	París	P1	Tuerca	Rojo	12	Londres
S3	Bernal	30	París	P4	Birlo	Rojo	14	Londres
S3	Bernal	30	París	P6	Engrane	Rojo	19	Londres

Figura 11 Ejemplo de la reunión-theta

Sería suficiente renombrar sólo uno de los dos atributos CIUDAD; la única razón para cambiar el nombre de los dos es la simetría.

La reunión-theta no es una operación primitiva; siempre es equivalente a obtener el producto cartesiano ampliado de las dos relaciones (con modificaciones apropiadas de los

nombres de los atributos, si es necesario), y después realizar una restricción apropiada sobre el resultado.

Si theta es “igual a”, la reunión theta se llama “equirreunión”. Por la definición, el resultado de una equirreunión debe incluir dos atributos con la propiedad de que los valores de esos dos atributos son iguales entre sí en cada tupla de la reunión. Si se elimina uno de esos dos atributos (lo cual puede hacerse mediante una proyección), el resultado será ¡la reunión natural! Por lo tanto, la reunión natural tampoco es una operación primitiva; es una proyección de una restricción de un producto (una vez más, con las operaciones apropiadas para renombrar atributos).

1.4.4. División

Toma dos relaciones, una binaria y una unaria y construye una relación formada por todos los valores de un atributo de la relación binaria que concuerdan (en el otro atributo) con todos los valores en la relación unaria.

Sean las cabeceras de las relaciones A y B $(X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_n)$ y (Y_1, Y_2, \dots, Y_n) respectivamente, es decir, los atributos Y_1, Y_2, \dots, Y_n son comunes a las dos relaciones. Además, A tiene los atributos X_1, X_2, \dots, X_m y B no tienen otros atributos. (Las relaciones A y B representan al dividendo y al divisor, respectivamente). Vamos a suponer que los atributos correspondientes (es decir, los atributos con el mismo nombre) están definidos sobre el mismo dominio. Consideremos a (X_1, X_2, \dots, X_m) y (Y_1, Y_2, \dots, Y_n) como dos atributos compuestos X y Y. La división de A entre B $A \text{ DIVIDE BY } B$ es una relación con la cabecera (X) y un cuerpo formado por el conjunto de todas las tuplas $(x:X)$ tales que aparece una tupla $(x:X, y:Y)$ en A para todas las tuplas $(y:Y)$ presentes en B (en otras palabras, el resultado contiene todos los valores de X en A cuyos valores de Y correspondientes en A incluyen a todos los valores de Y en B, en términos informales). Como siempre, las relaciones A y B pueden ser expresiones.

La operación división es la contraria a la operación producto y quizás sea la más compleja de explicar, por tanto daré un ejemplo. Una determinada empresa posee una tabla de comerciales, otra tabla de productos y otra con las ventas de los comerciales. *Queremos averiguar que comerciales han vendido todo tipo de producto.*

Lo primero que hacemos es extraer en una tabla todos los códigos de todos los productos, en la Figura 12 a esta tabla la denominamos A.

TABLA A
CÓDIGO PRODUCTO
1035
2241
2249
2518

Figura 12 Tabla A

En una segunda tabla mostrada en la Figura 13 extraemos, de la tabla de ventas, el código del producto y el comercial que lo ha vendido, lo hacemos con una proyección y evitamos traer valores duplicados. El resultado podría ser el siguiente:

TABLA B	
CÓDIGO COMERCIAL	CÓDIGO PRODUCTO
10	2241
23	2518
23	1035
39	2518
37	2518
10	2249
23	2249
23	2241

Figura 13 Tabla B

Si dividimos la tabla B entre la tabla A obtendremos como resultado una tercera tabla presentada en la figura 14 que:

Los campos que contiene son aquellos de la tabla B que no existen en la tabla A. En este caso el campo Código Comercial es el único de la tabla B que no existe en la tabla A.

Un registro se encuentra en la tabla resultado si y sólo si está asociado en tabla B con cada fila de la tabla A

TABLA RESULTADO
CÓDIGO COMERCIAL
23

Figura 14 Tabla Resultado

¿Por qué el resultado es 23?. El comercial 23 es el único de la tabla B que tiene asociados todos los posibles códigos de producto de la tabla A.

Asociatividad.

Es fácil comprobar que la unión es asociativa; es decir, si A B y C son “proyecciones” arbitrarias, entonces las expresiones:

(A UNION B) UNION C y A UNION (B UNION C) son equivalentes.

Así, por comodidad, nos permitiremos escribir una secuencia de unión sin insertar paréntesis; por ejemplo, cualquiera de las dos expresiones anteriores puede simplificarse a: A UNION B UNION C sin provocar ambigüedad.

Algo análogo podría decirse de la intersección y el producto (pero no de la diferencia). Señalamos también que la unión, la intersección y el producto (pero no la diferencia) son conmutativas, es decir, las expresiones A UNION B y B UNION A son equivalentes también y lo mismo sucede con INTERSECT y *.

1.5. Secuencia de operaciones y cambio de nombre de los atributos

Una relación tiene dos partes importantes: una *cabecera* y un *cuerpo*; la *cabecera es el conjunto de nombres de atributos* y el *cuerpo consiste en los datos*, hablando en términos informales. Ahora bien, toda relación *nombrada* (es decir, toda relación –relación base, vista, etcétera- incluida de manera explícita en la definición de la BD) tendrá desde luego

una cabecera con todas las de la ley; pero, ¿Qué hay de las relaciones no nombradas (o sea, resultantes)?

Es importante que una relación resultante tenga un conjunto apropiado de nombres de atributos porque, desde luego, esa relación podría ser el resultado de una expresión anidada dentro de otra más grande y obviamente necesitaremos alguna forma de referirnos a los atributos del resultado de la expresión interior desde esa expresión exterior. Si desea un análisis más a fondo de este punto, recomendamos al lector consultar el artículo de Warden “The naming of columns” (bautizo de columnas) incluido en la referencia [Andrew Warden 1990].

Por lo tanto, nuestra versión del álgebra relacional se definirá de manera tal que garantice cabeceras apropiadas para *todas* las relaciones; es decir, cabeceras en las cuales cada atributo tenga un nombre propio no calificado y único dentro de la relación que lo contiene.

La operación de renombrar se identifica como RENAME o en otras fuentes como: R(<LISTA DE ATRIBUTOS>) y para asignarle un nombre inicial a una relación como: ←

Citaremos el nuevo operador, RENAME (renombrar), cuyo propósito es en esencia cambiar el nombre de los atributos dentro de una relación. En términos más precisos, el operador RENAME toma una relación especificada y –al menos en lo conceptual- crea una copia nueva de esa relación en la cual se ha dado un nombre diferente a uno de los atributos (la relación especificada podría ser, desde luego, el resultado de una expresión e incluir otras operaciones algebraicas) por ejemplo, podríamos escribir:

S RENAME CIUDAD AS SCIUDAD

El resultado de evaluar esta expresión es una relación –la cual por cierto no tiene nombre propio- con el mismo cuerpo que la relación S pero en la cual el atributo de ciudad se llama SCIUDAD en vez de CIUDAD. Los demás nombres de atributos se heredan sin modificaciones de sus contrapartes en la relación S.

Se pueden escribir las operaciones en una sola expresión del álgebra relacional anidándolas, o bien aplicar una operación cada vez y crear relaciones de resultados intermedios. En el segundo caso, se tendrán que renombrar las relaciones que contienen los resultados intermedios. Por ejemplo si se quiere obtener el nombre, el apellido y el salario de todos los empleados que trabajan en el departamento número 5, se deberá aplicar una operación SELECCIONAR y una operación PROYECTAR. Se puede escribir una sola expresión del álgebra relacional, de la siguiente forma:

$\Pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\sigma_{\text{ND}=5}(\text{EMPLEADO}))$

Como otra alternativa, se puede mostrar explícitamente la secuencia de operaciones, dando un nombre a cada una de las relaciones intermedias, como sigue:

EMPS_DEP5 ← $\sigma_{\text{ND}=5}(\text{EMPLEADO})$
RESULTADO ← $\pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\text{EMPS_DEP5})$

Frecuentemente es más sencillo descomponer una secuencia compleja de operaciones especificando relaciones de resultados intermedios que escribir una sola expresión del álgebra relacional. También se puede usar la técnica de renombrar los atributos de las relaciones intermedias y de la resultante. Útil en las operaciones complejas de UNION y REUNIÓN. Esta otra forma de renombrar los atributos de una relación basta con que se incluya una lista con los nuevos nombres de los atributos entre paréntesis, como en el siguiente ejemplo:

$TEMP \leftarrow \sigma_{ND=5}(EMPLEADO)$

$R(NOMBRE_PILA, PRIMER_APELL, SALARIO) \leftarrow \pi_{NOMBRE, APELLIDO, SALARIO}(TEMP)$

Sino se cambian los atributos de la relación resultante de una operación de selección serán los mismos que los de la relación original y estarán en el mismo orden. En el caso de la operación de Proyección sin renombrado, la relación resultante tendrá los mismos nombres de atributos que aparecen en la lista de proyección y en el mismo orden en el que aparecen en ella.

1.6. Conjunto completo de operaciones del álgebra relacional

El conjunto de operaciones $\{\sigma, \pi, \text{UNION}, \text{DIFERENCIA}, \text{PRODUCTO CARTESIANO}\}$ es un conjunto completo, es decir, cualquiera de las otras operaciones del álgebra relacional se puede expresar como una secuencia de operaciones de este conjunto. Por ejemplo, la operación INTERSECCIÓN se puede expresar empleando UNION y DIFERENCIA como sigue:

$R \text{ UNION } S = (R \text{ UNION } S) \text{ MINUS } ((R \text{ MINUS } S) \text{ UNION } (S \text{ MINUS } R))$

Otro ejemplo: una operación de REUNIÓN se puede especificar como un PRODUCTO CARTESIANO seguido de una operación SELECCIONAR.

De manera similar, una REUNIÓN NATURAL se puede especificar como un PRODUCTO CARTESIANO precedido de RENOMBRAR y seguido de operaciones SELECCIONAR y PROYECTAR. Así pues, las diversas operaciones de REUNIÓN tampoco son estrictamente necesarias para el poder expresivo del álgebra relacional: sin embargo, son muy importantes porque son cómodas y se emplean con mucha frecuencia en las aplicaciones de base de datos.

1.7. Operaciones de cerradura recursiva

Antes de explicar las operaciones de cerradura recursiva, se conocerá la propiedad de cerradura.

Propiedad de cerradura

El resultado de cada una de *las operaciones tradicionales de conjuntos y operaciones relacionales especiales* es por supuesto otra relación. Esta es la importantísima propiedad de **cerradura**. Dado que el resultado de cualquier operación es un objeto del mismo tipo que los operandos, todos son *relaciones*, el resultado de una operación puede convertirse en operando de otra. Así pues, es posible (por ejemplo) sacar la *proyección* de una *unión*, o una *reunión* de dos *restricciones*, o la *diferencia* de una *unión* y una *intersección*, etc. dicho de otro modo, es posible escribir expresiones relacionales anidadas; es decir, expresiones en las cuales los operandos mismos están representados mediante expresiones y no solo mediante nombres.

La *unión* incluida en el álgebra relacional no es la unión matemática completamente general; más bien, es una *forma limitada de unión*, en la cual se obliga a las dos relaciones de entrada a tener lo que podríamos llamar en términos informales “la misma forma”; si las dos relaciones tienen la misma forma en este sentido, podremos obtener su unión, y el resultado será también una relación con la misma forma; en otras palabras, se habrá conservado la propiedad de *cerradura*.

Un término más preciso para el concepto “la misma forma” es compatibilidad respecto a la *unión*, es decir, si sus cabeceras son idénticas, esto es que las dos tienen el mismo conjunto de nombres de atributos, a fuerza deben tener el mismo grado; y los atributos correspondientes (es decir, los atributos con el mismo nombre en las dos relaciones) se definen sobre el mismo dominio.

Operaciones de cierre recursivo

Las operaciones de cierre recursivo en general, no pueden especificarse en el álgebra relacional básica. Esta operación se aplica a un vínculo recursivo entre las tuplas del mismo tipo, como el vínculo recursivo entre un empleado y un supervisor. Este vínculo se describe mediante la clave externa NSS_SUPERV de la relación EMPLEADO indicado en la figura 15.



Figura 15 relación EMPLEADO con vínculo recursivo

En dicha figura relaciona cada tupla de EMPLEADO (en el papel de supervisado) con otra tupla de empleado (en el papel de supervisor). Un ejemplo de operación recursiva sería obtener todos los supervisados de un empleado e en todos niveles; es decir, todos los empleados e' supervisados directamente por e , todos los empleados e'' supervisados directamente por cada empleado e' , todos los empleados e''' supervisados directamente por cada empleado e'' y así sucesivamente. Aunque en el álgebra relacional resulta sencillo especificar todos los empleados supervisor por e en un nivel específico, no lo es especificar todos los supervisados en todos los niveles. Porque no se conoce el número máximo de niveles, ya que se necesitaría un mecanismo de ciclo.

1.8. Funciones agregadas

Dado que algunas consultas como “cuantos proveedores hay?” no se pueden expresar hasta ahora, el álgebra relacional ofrece una serie de funciones agregadas para ampliar su capacidad básica de recuperación de información; estas funciones son: COUNT, SUM, AVG, MAX, MIN, que trabajan sobre el total de valores en una columna de alguna tabla. - quizá una tabla derivada, es decir, una tabla construida como resultado de alguna consulta y produce un valor.

COUNT(*atributo*) El resultado es una tabla, con una fila y una columna (sin nombre) conteniendo un solo valor escalar (el valor de la suma de *atributo*).

SUM(CANT) presenta la suma de los valores de CANT.

AVG(CANT) presenta el promedio de la columna CANT.

MAX(CANT) presenta el máximo valor de la columna CANT.

MIN(CANT) presenta el mínimo valor de la columna CANT.

1.9. Otras operaciones adicionales

Varios autores han propuesto nuevos operadores de naturaleza algebraica como candidatos para añadirlos al conjunto original. Las descripciones de las primeras tres (ampliación, resumen, división generalizada) siguen los lineamientos generales de las descripciones de Warden [Andrew Warden 1990]; Warden a su vez se vio muy influido por el trabajo de Todd.

1.9.1. Ampliación

Hasta ahora el álgebra que se ha descrito no incluye la capacidad de cálculo. No obstante, es obvio que en la práctica tal capacidad es deseable. Por ejemplo, desearíamos poder obtener de la BD el valor de una expresión aritmética como PESO*454 o hacer referencia a un valor de este tipo en una cláusula WHERE. El propósito de EXTEND (ampliar) es brindar esa capacidad. En términos más precisos, EXTEND toma una relación especificada y (al menos en lo conceptual) *crea una nueva relación semejante a la original pero con un atributo más, cuyos valores se obtienen evaluando alguna expresión de cálculo (escalar) especificada*. Por ejemplo:

```
EXTEND P ADD ( PESO * 454 ) AS PESOGRS
```

La evaluación de esta expresión produce una relación cuya cabecera es idéntica a la de P, excepto que incluye además un atributo llamado PESOGRS. Cada tupla de esa relación es igual a la tupla correspondiente en P, excepto que incluye un valor de PESOGRS, calculado de la manera especificada. Ahora podemos emplear el atributo PESOGRS en proyecciones, restricciones, etc. por ejemplo:

```
( EXTEND P ADD ( PESO * 454 ) AS PESOGRS ) WHERE PESOGRS > 10 000
```

Podemos incorporar EXTEND en nuestra sintaxis, añadiendo un nuevo tipo de “expresión de una sola relación” cuya sintaxis es:

```
EXTEND término ADD cálculo-escalar AS atributo
```

También podemos abreviar “ampliaciones múltiples”, como sigue:

```
( EXTEND P ADD ‘Peso en gramos’ AS EXPLICACIÓN,  
  ( PESO * 454 ) AS PESOGRS )
```

1.9.2. Resumen

La operación EXTEND (ampliar) hace posible incorporar cálculos escalares al álgebra. SUMMARIZE (resumir) hace algo análogo con las operaciones de agregados (cuenta, suma, promedio, máximo, mínimo, y quizá otras). Por ejemplo, la expresión

```
SUMMARIZE SP GROUPBY ( P# ) ADD SUM ( CANT ) AS CANTTOTAL
```

Produce al evaluarse una relación con la cabecera (P#, CANTTOTAL), en la cual hay una tupla por cada valor distinto de P# en SP, dando ese valor de P# y la cantidad total correspondiente.

La nueva “expresión de una sola relación” añadida es:

SUMMARIZE término GROUPBY (listaconcomas-de-atributos)

ADD cálculo-de-agregados AS atributo

Si se omite la lista de atributos en la cláusula GROUPBY, el efecto es agrupar la relación especificada mediante “término” según *ningún atributo* y por tanto realizar el cálculo de agregados una sola vez para toda la relación. Por ejemplo:

SUMMARIZE SP GROUPBY () ADD SUM (CANT) AS GRANDTOTAL

Esta expresión produce una relación con un atributo y una tupla; el atributo se llama GRANDTOTAL, y el único valor escalar en la relación es el total de todos los valores CANT en la relación SP original.

También hay resúmenes múltiples:

(SUMMARIZE SP GROUPBY (P#) ADD SUM (CANT) AS CANTTOTAL,
AVG (CANT) AS CANTPROMEDIO)

1.9.3. División generalizada

El operador de división tal como se definió anteriormente se aplica sólo cuando las relaciones dividendo y divisor satisfacen la propiedad según la cual la cabecera del divisor es un subconjunto correcto de la cabecera del dividendo. El operador de **división generalizada**, en cambio, se aplica a *cualquier pareja de relaciones*. Definimos ese operador como sigue. Dadas las relaciones A(X, Y) y B(Y, Z), la expresión:

A DIVIDEBY B

Produce una relación con la cabecera (X, Z) y un cuerpo formado por todas las tuplas (x:X, z:Z) tales que aparece una tupla (x:X, y:Y) en A para todas las tuplas (y:Y, z:Z) que aparecen en B.

NOTA. Podemos conservar nuestra sintaxis original, porque de la definición se desprende que la división original es sólo un caso especial de la operación generalizada. En términos específicos, *si Z está vacío la operación se reduce a la división original de A entre B*; de manera similar, *si X está vacío la operación se reduce a la división original de B entre A*. Además, si Y está vacío la operación degenera en el producto cartesiano de A y B.

Vamos a suponer que tenemos dos relaciones SP(S#, P#) y PJ(P#, J#), donde SP indica cuáles proveedores suministran cuáles partes y PJ muestra cuáles partes se utilizan en cuáles proyectos. La expresión:

SP DIVIDEBY PJ

Producirá una relación con la cabecera (S#, J#) y formada por pares de números de proveedor y números de proyecto tales que el proveedor indicado suministra todas las partes empleadas en el proyecto indicado. De manera similar, la expresión:

PJ DIVIDEBY SP

Producirá una relación con la cabecera (J#, S#) y formada por pares de números de proyecto y números de proveedor tales que en el proyecto indicado se emplean todas las partes suministradas por el proveedor indicado.

1.9.4. Reunión externa

Es una forma ampliada de la operación ordinaria (o interna) de reunión, en cuyo resultado aparecen las tuplas de una relación que no tienen contraparte en la otra, con nulos en las posiciones de los demás atributos (en vez de hacerse caso omiso de ellos, como en la reunión ordinaria). No es una operación primitiva.

Dada la siguiente base de datos de proveedores y partes, mostrada en la figura 16.

S			
S#	SNOMBRE	SITUACION	CIUDAD
S1	Salazar	20	Londres
S2	Jaines	10	París
S3	Bernal	30	París
S4	Corona	20	Londres
S5	Aldana	30	Atenas

P				
P#	PNOMBRE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12	Londres
P2	Perno	Verde	17	París
P3	Birlo	Azul	17	Roma
P4	Birlo	Rojo	14	Londres
P5	Leva	Azul	12	París
P6	Engrane	Rojo	19	Londres

SP		
S#	P#	CANT
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P3	200
S4	P2	200
S4	P4	300
S4	P5	400

Figura 16 Relación de proveedores y partes

Por ejemplo, la siguiente operación en pseudoSQL podría servir para construir la reunión externa natural de proveedores y envíos según números de proveedor (es sólo “pseudoSQL” porque el verdadero SQL no permite NULL en una cláusula SELECT (seleccionar)).

```
SELECT S.*, SP.P#, SP.CANT FROM S, SP WHERE S.S# = SP.S#
UNION
SELECT S.*, NULL, NULL FROM S WHERE NOT EXISTS (SELECT * FROM SP
WHERE SP.S# = S.S#);
```

El resultado incluye las tuplas de los proveedores que no suministran partes, concatenadas con nulos en las posiciones de P# y CANT.

Aunque no es primitiva, la reunión externa se requiere con bastante frecuencia en la práctica y sería deseable que el sistema la manejara en forma directa, en vez de obligar al usuario a realizar largos circunloquios. Sin embargo, no se propone una sintaxis específica para la operación de reunión externa por las siguientes razones:

- En primer lugar, la cuestión de la reunión externa dista mucho de ser tan trivial. Un problema es que, aunque la reunión natural interna (ordinaria) es una proyección de la equirreunión interna, la reunión natural externa no es una proyección de la equirreunión externa. Una consecuencia de esto es la dificultad para añadir en forma elegante el manejo de la reunión externa a los lenguajes ya existentes –sobre todo SQL- porque esos lenguajes casi siempre están basados en la obtención de proyecciones. Varios productos de DBMS han intentado resolver este problema y han fracasado estrepitosamente.
- En segundo lugar, se presenta un problema severo de *interpretación* de los nulos que aparecen en el resultado de una reunión externa. Por ejemplo, ¿qué significan en el ejemplo anterior? Sin duda no significan “valor desconocido” ni “la propiedad no se aplica”.

Cabe señalar también que la reunión externa en ocasiones produce una relación con nulos en la posición de la clave primaria, lo cual hace imposible convertirla en una relación base.

A pesar de lo anterior, no cabe duda de que la reunión externa es importante en la práctica, y de hecho Codd considera a esa operación como parte del modelo relacional básico [C. J. Date 1986]. (Dicho en forma más precisa, incluye la reunión theta externa en el modelo pero no la reunión natural externa –posición un tanto extraña, en vista de que a) la reunión natural externa es mucho más útil en la práctica y b) la reunión natural externa no se puede derivar en forma directa de la reunión theta externa).

También es posible definir versiones “externas” de algunas otras operaciones del álgebra relacional, específicamente la unión, la intersección y la diferencia y también en este caso Codd incluye ahora por lo menos una de ellas, **la unión externa**, en el modelo relacional.

Tales operaciones permiten efectuar uniones (etcétera) entre dos relaciones aunque esas relaciones no sean compatibles respecto a la unión. En esencia, lo que hacen es *ampliar cada operando a fin de incluir los atributos exclusivos del otro (de modo que los operandos ya sean compatibles respecto a la unión), insertar nulos en cada tupla para todos estos atributos añadidos y realizar después la unión, intersección o diferencia normales, según sea el caso*. Sin embargo, no analizaremos estas operaciones en detalle, por las siguientes razones:

- **La intersección externa** siempre produce una relación vacía, excepto en el caso especial en que las relaciones originales sean compatibles respecto a la unión desde un principio, pues en ese caso degenera en la intersección normal. Por lo tanto, la intersección externa no parece ser muy útil.
- **La diferencia externa** siempre produce como resultado su primer operando, excepto en el caso especial en que las relaciones originales sean compatibles respecto a la unión desde un principio, ya que en ese caso degenera en la diferencia normal. Por tanto, la diferencia externa no parece ser muy útil.

- **La unión externa** tiene *graves* problemas de interpretación (mucho peores que el problema de interpretación de la reunión externa). En el artículo de Warnen “Into the Unknown” (“Hacia lo desconocido”) [Andrew Warden 1983] se analiza todo.
- **La reunión externa izquierda** supóngase que se desea una lista de todos los nombres de los empleados y también el nombre de los departamentos que dirigen, si es el caso de que dirijan un departamento; se puede aplicar una operación **reunión externa izquierda REI** para obtener el resultado como sigue:

```
TEMP←(EMPLEADO REINSS=NSS_JEFEDEPARTAMENTO)  
RESULTADO← $\pi$ NOMBRE, INIC, APELLIDO, NOMBRED(TEMP)
```

La operación REI conserva todas la tuplas de la relación de la izquierda, sino se encuentra una tupla coincidente en la relación DEPARTAMENTO, los atributos de éste del resultado se rellenan con valores nulos.

- **La reunión externa derecha** siguiendo el mismo ejemplo pero ahora con la reunión externa derecha **RED** se tiene:

```
TEMP←(EMPLEADO REDNSS=NSS_JEFEDEPARTAMENTO)  
RESULTADO← $\pi$ NOMBRE, INIC, APELLIDO, NOMBRED(TEMP)
```

Conserva en el resultado todas las tuplas de la relación de la derecha.

- **La reunión externa completa** conserva todas las tuplas de ambas relaciones, izquierda y derecha, cuando no se encuentran tuplas coincidentes, rellenándolas con valores nulos si es necesario.

1.9.5. Unión externa

Esta operación se creó para efectuar la unión de dos tuplas de dos relaciones que no son compatibles con la unión. Esta operación efectuará la unión de tuplas de dos relaciones que son parcialmente compatibles, lo que significa que sólo algunos de sus atributos son compatibles con la unión. Se espera una lista de atributos compatibles que incluya la clave de ambas relaciones. Las tuplas de las relaciones componentes con la misma clave, se representan sólo una vez en el resultado y tienen valores para todos los atributos del resultado. Los atributos de cada relación que no son compatibles con la unión se mantienen en el resultado y las tuplas que no tienen valores para dichos atributos se rellenan con valores nulos.

Por ejemplo se puede aplicar una unión externa a dos relaciones cuyos esquemas son ALUMNO (Nombre, NSS, Departamento, Asesor) y PROFESORADO(Nombre, NSS, Departamento, Rango). El esquema de la relación resultante es R (Nombre, NSS, Departamento, Asesor, Rango) y todas las tuplas de ambas relaciones se incluyen en el resultado.

Las tuplas de los alumnos tendrán nulos en el atributo rango y las tuplas de profesorado tendrán nulos en el atributo Asesor. Una tupla que exista en ambas relaciones tendrá valores para todos sus atributos.

1.10. Operaciones quizá

Solo mencionamos aquí estas operaciones para que nuestro análisis sea completo. En este punto nos limitaremos a señalar que Codd también incluye ahora ciertas “operaciones quizá” (específicamente, versiones quizá de las operaciones de restricción, reunión theta, reunión theta externa y división) en su definición del modelo relacional básico.

Por definición, si se aplica el operador QUIZÁ (MAYBE) a una expresión lógica p , el resultado será *verdadero* si al evaluar p se obtiene *desconocido*, y *falso* en cualquier otro caso. Así pues, QUIZÁ se define con la siguiente tabla de verdad:

QUIZÁ

v		f
d		v
f		f

nota: aquí nos estamos apartando un poco de la propuesta de Codd; él no introduce un operador lógico QUIZÁ en sí, sino que habla en términos de una opción QUIZÁ (MAYBE) en las consultas. También analiza versiones QUIZÁ de algunos de los operadores del álgebra relacional (por ejemplo, la “reunión theta quizá”, la cual junta las filas de los operandos cuando la condición de reunión resulta *desconocida* en vez de *verdadera*).

En su desarrollo, Codd considera después las implicaciones que las ideas anteriores tendrían en los operadores del álgebra relacional. Omitiremos aquí los detalles, limitándonos a señalar que siempre que las comparaciones escalares intervienen en esas operaciones, ya sea de manera explícita (como en la restricción) o implícita (como en la división), los casos importantes de esas comparaciones son los verdaderos, no los desconocidos ni (desde luego) los falsos. Así, por ejemplo, la consulta “obtener los proveedores de Londres” devolverá sólo las filas de la tabla S para los cuales el resultado de evaluar la condición: CIUDAD = ‘Londres’ sea *verdadero*, no *falso* ni *desconocido*.

Hasta ahora nos hemos limitado a examinar un solo tipo de nulo, a saber, “valor desconocido”. Sin embargo, pueden existir varios otros tipos, por ejemplo, “la propiedad no es aplicable”, “el valor no existe”, “el valor no está definido”, etcétera, etcétera. En la referencia [E. F. Codd 1987] Codd propone expandir el enfoque de la lógica para manejar un tipo adicional de nulo, a saber “la propiedad no es aplicable”. Así pues, pareciera que n tipos de nulo conducirían a una lógica de $(n + 2)$ valores. Considerando todos los problemas que surgen en el caso simple de $n = 1$, seguramente deberá cuestionarse la convivencia de ampliar el tratamiento a un $n > 1$ arbitrario.

En vista de la existencia de varios tipos de nulos, otro de los problemas que surgen es el de la *interpretación*. Por ejemplo, ¿qué significan los nulos generados por una recuperación de reunión externa? Los ejemplos sugieren significados diferentes en los distintos contextos. Y adviértase también que hay la misma probabilidad de errores en un sistema en el cual sí se manejan nulos, pero no del “tipo adecuado” (para el contexto en cuestión), que en un sistema en el cual no se manejan en absoluto los nulos. En la referencia [C. J. Date 1990] el lector puede encontrar un análisis más a fondo de este punto, así como en el artículo de Warden “Into the Unknown” (“Hacia lo desconocido”) incluido en la referencia [Andrew Warden 1983].

Por todas las razones identificadas y por muchas más, sentimos que hace falta mucha labor de investigación adicional sobre el tema de la información faltante. También pensamos que en tanto no se terminen de manera satisfactoria esas investigaciones, será preferible en la práctica una estrategia basada en valores por omisión definidos por el administrador de bases de datos.

1.11. Asignación relacional

Esta operación algebraica consiste en asignar un valor a uno o varios campos de una tabla.

El propósito de esa operación es poder “recordar” el valor de alguna expresión algebraica y así modificar el estado de la BD. Pero la asignación de relaciones es una operación un tanto burda, en cuanto a que sólo permite la sustitución total del valor completo de una relación. En la práctica, por supuesto, serán deseables algunas operaciones de actualización de mayor exactitud. En teoría, la operación de asignación podría servir como base para esas operaciones más precisas. Por ejemplo, teóricamente sería posible realizar inserciones y eliminaciones de la manera sugerida en los siguientes ejemplos:

```
S := S UNION { ( S#           : 'S6',
                SNOMBRE     : 'Beltrán',
                SITUACION    : 50,
                CIUDAD       : 'Madrid' ) } ;
SP := SP MINUS { ( S#           : 'S1',
                  P#           : 'P1',
                  CANT         : 300 ) } ;
```

La primera de estas asignaciones inserta la tupla del proveedor S6 en la relación S, la segunda elimina el envío del proveedor S1 y la parte P1 de la relación SP.

1.12. Ejemplos de consultas en el álgebra relacional

Ahora se presentan ejemplos adicionales para ilustrar el empleo de las operaciones del álgebra relacional. Todos ellos se refieren a la base de datos que la siguiente Figura 17.

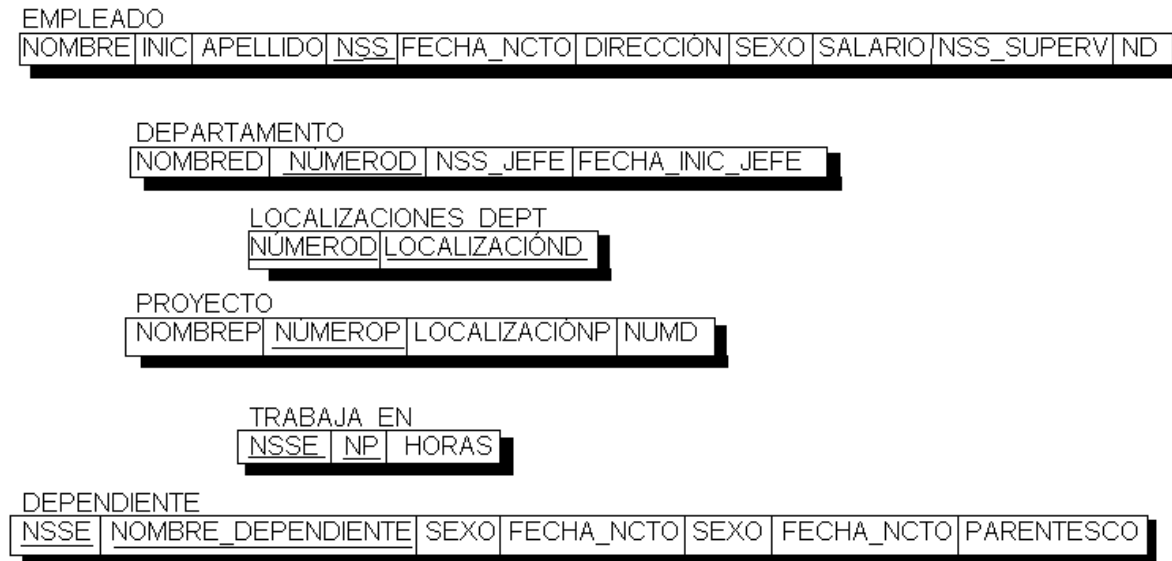


Figura 17. Diseño de la base de datos para representar las consultas en el álgebra relacional

En general, la misma consulta puede expresarse de muchas formas mediante diversas operaciones. Se expresarán las dos primeras consultas y se dejará al lector las otras formulaciones.

Consulta 1

Obtener el nombre y la dirección de todos los empleados que trabajan para el departamento 'Investigación'.

$$\begin{aligned}
 DEPTO_INVEST &\leftarrow \sigma_{NOMBRED='INVESTIGACIÓN'}(DEPARTAMENTO) \\
 EMPS_DEPTO_INVEST &\leftarrow (DEPTO_INVEST \bowtie_{NUMERO=ND} EMPLEADO) \\
 RESULTADO &\leftarrow \pi_{NOMBRE, APELLIDO, DIRECCIÓN}(EMPS_DEPTO_INVEST)
 \end{aligned}$$

Esta consulta se podía especificar de otras maneras; por ejemplo, se podría invertir el orden de las operaciones REUNIÓN y SELECCIONAR, o se podría sustituir la REUNIÓN por una REUNIÓN NATURAL.

Consulta 2

Para cada proyecto localizado en 'Santiago', obtenga una lista con el número de proyecto, el número del departamento que lo controla y el apellido, la dirección y la fecha de nacimiento del jefe de dicho departamento.

$$\begin{aligned}
 PROYS_SANTIAGO &\leftarrow \sigma_{LOCALIZACIONP='Santiago'}(PROYECTO) \\
 DEPTO_CONTR &\leftarrow (PROYS_SANTIAGO \bowtie_{NUMD=NUMEROD} DEPARTAMENTO) \\
 JEFE_DEPTO_PROY &\leftarrow (DEPTO_CONTR \bowtie_{NSS_JEFE=NSS} EMPLEADO) \\
 RESULTADO &\leftarrow \pi_{NUMEROP, NUMD, APELLIDO, DIRECCIÓN, FECHA_NCTO}(JEFE_DEPTO_PROY)
 \end{aligned}$$

Consulta 3

Buscar el nombre de los empleados que trabajan en todos los proyectos controlados por el departamento número 5.

Consulta 4

Preparar una lista con los números de los proyectos en que interviene un empleado cuyo apellido es 'Smith', ya sea como trabajador o como jefe del departamento que controla el proyecto.

Consulta 5

Preparar una lista con los nombres de todos los empleados que tienen dos o más personas dependientes de ellos.

En esta consulta se debe usar la operación función agregada CUENTA.

Consulta 6

Obtener los nombres de los empleados que no tienen otras personas dependientes de ellos.

Consulta 7

Obtener los nombres de los jefes que tienen por lo menos una persona dependiente de ellos.

2. El lenguaje Sql

Breve historia

La historia de SQL (que se pronuncia deletreando en inglés las letras que lo componen, es decir “ese-cu-ele” y no “siquel” como se oye a menudo) empieza en 1974 con la definición, por parte de Donald Chamberlin y de otras personas que trabajaban en los laboratorios de investigación de IBM, de un lenguaje para la especificación de las características de las bases de datos que adoptaban el modelo relacional. Este lenguaje se llamaba SEQUEL (Structured English Query Language) y se implementó en un prototipo llamado SEQUEL-XRM entre 1974 y 1975. Las experimentaciones con ese prototipo condujeron, entre 1976 y 1977, a una revisión del lenguaje (SEQUEL/2), que a partir de ese momento cambió de nombre por motivos legales, convirtiéndose en SQL. El prototipo (System R), basado en este lenguaje, se adoptó y utilizó internamente en IBM y lo adoptaron algunos de sus clientes elegidos. Gracias al éxito de este sistema, que no estaba todavía comercializado, también otras compañías empezaron a desarrollar sus productos relacionales basados en SQL. A partir de 1981, IBM comenzó a entregar sus productos relacionales y en 1983 empezó a vender DB2. En el curso de los años ochenta, numerosas compañías (por ejemplo Oracle y Sybase, sólo por citar algunos) comercializaron productos basados en SQL, que se convierte en el estándar industrial de hecho por lo que respecta a las bases de datos relacionales.

En 1986, el ANSI adoptó SQL (sustancialmente adoptó el dialecto SQL de IBM) como estándar para los lenguajes relacionales y en 1987 se transformó en estándar ISO. Esta versión del estándar va con el nombre de SQL/86. En los años siguientes, éste ha sufrido diversas revisiones que han conducido primero a la versión SQL/89 y, posteriormente, a la actual SQL/92.

El hecho de tener un estándar definido por un lenguaje para bases de datos relacionales abre potencialmente el camino a la intercomunicabilidad entre todos los productos que se basan en él. Desde el punto de vista práctico, por desgracia las cosas fueron de otro modo. Efectivamente, en general cada productor adopta e implementa en la propia base de datos sólo el corazón del lenguaje SQL (el así llamado Entry level o al máximo el Intermediate level), extendiéndolo de manera individual según la propia visión que cada cual tenga del mundo de las bases de datos.

Actualmente, está en marcha un proceso de revisión del lenguaje por parte de los comités ANSI e ISO, que debería terminar en la definición de lo que en este momento se conoce como SQL3. Las características principales de esta nueva encarnación de SQL deberían ser su transformación en un lenguaje stand-alone (mientras ahora se usa como lenguaje hospedado en otros lenguajes) y la introducción de nuevos tipos de datos más complejos que permitan, por ejemplo, el tratamiento de datos multimediales.

En una base de datos, no se necesita especificar la ruta de acceso a las tablas y no necesita saber como están almacenados físicamente los datos.

Para acceder la Base de datos, debe ejecutar instrucciones en un lenguaje estructurado de consultas Structured Query Language SQL, el cual es un estándar de la American National Standards Institute ANSI para operar sobre las Bases de Datos.

Componentes del SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

Este lenguaje contiene una gran cantidad de operadores que participan y combinan las tablas. Una Base de Datos puede ser modificada utilizando sentencias de SQL. Como se muestra en la figura 18.

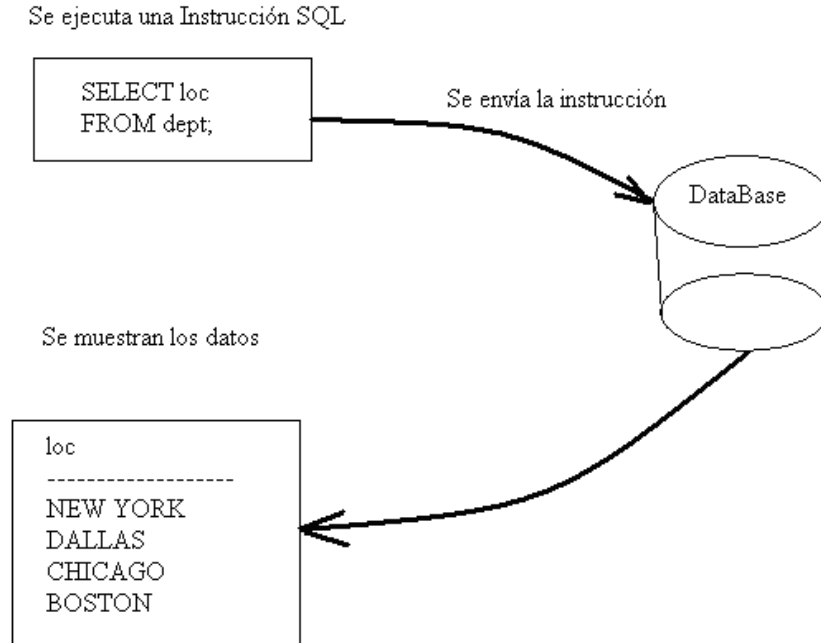


Figura 18 Comunicación con RDBMS usando SQL

SQL permite comunicarse con el servidor y se tiene las siguientes ventajas:

1. Eficiencia
2. Fácil de aprender y utilizar
3. Funcionalidad completa.

SQL también permite definir, recuperar y manipular datos en las tablas.

SELECT	Data retrieval (DML)	Recupera datos de una tabla
INSERT UPDATE DELETE	Data manipulation language (DML)	Agrega nuevos registros a las tablas Modifica los valores de los campos en un registro Elimina registros de una tabla
CREATE ALTER DROP	Data definition language (DDL)	Crea tablas Permite modificar la estructura de una tabla Elimina una tabla
COMMIT ROLLBACK	Transaction control	Permite grabar los cambios a la BD en disco Restaura los cambios hechos a la BD hasta el último COMMIT
GRANT REVOKE	Data control language (DCL)	Permite otorgar privilegios a los usuarios Elimina los privilegios otorgados a los usuarios

Instrucciones de SQL

Los comités de la ANSI y de la Internacional Standards Organization ISO, han designado a SQL como el lenguaje estándar para las Bases de Datos Relacionales, en 1992 este estándar se ha denominado ANSI SQL-92 o SQL2.

Comandos

Existen dos tipos de comandos SQL:

Los DDL que permiten crear y definir nuevas bases de datos, campos e índices.

Los DML que permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.

Comandos DDL:

Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices
DROP	Empleado para eliminar tablas e índices
ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Comandos DML:

Comando	Descripción
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados
DELETE	Utilizado para eliminar registros de una tabla de una base de datos

Cláusulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

Operadores lógicos

Operador	Uso
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor ó Igual que
>=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo
In	Utilizado para especificar registros de una base de datos

Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

Consultas básicas en SQL

SQL tiene una sentencia básica para recuperar información de una BD: la sentencia SELECT. Esta sentencia no tiene relación con la operación SELECT del álgebra relacional. Hay muchas opciones y matices para la sentencia SELECT de SQL, por lo que se presentarán sus características gradualmente. Antes de continuar debemos señalar una diferencia importante entre SQL y el modelo relacional formal: SQL permite que las tablas (relaciones) tengan dos o más tuplas idénticas en todos los valores de sus atributos. Por tanto, en general, una tabla de SQL no es un conjunto de tuplas, porque los conjuntos no permiten dos miembros idénticos; más bien, es un multiconjunto (a veces llamado bolsa o bag) de tuplas. Algunas relaciones SQL están obligadas a ser conjuntos porque se ha declarado una restricción de clave o porque se ha usado la opción DISTINCT en la sentencia SELECT.

2.1. Estructura general de las consultas en SQL

La forma básica de la sentencia SELECT, en ocasiones denominada correspondencia o bloque **select-from-where**, consta de tres cláusulas SELECT, FROM y WHERE y tiene la siguiente forma:

```
SELECT <lista de atributos o campos>  
FROM <tabla o lista de tablas>  
WHERE <condición>
```

Donde:

- <lista de atributos> es una lista de nombres de atributos cuyos valores van a ser recuperados por la consulta.
- <lista de tablas> es una lista de nombres de las relaciones necesarias para procesar la consulta.
- <condición> es una expresión condicional (booleana) que identifica las tuplas que van a ser recuperadas por la consulta.

Por ejemplo:

```
SELECT Nombre, Telefono FROM Clientes;
```

Esta consulta devuelve un recordset con el campo nombre y teléfono de la tabla clientes.

Devolver Literales

En determinadas ocasiones nos puede interesar incluir una columna con un texto fijo en una consulta de selección, por ejemplo, supongamos que tenemos una tabla de *Empleados* y deseamos recuperar las tarifas semanales de los electricistas, podríamos realizar la siguiente consulta:

```
SELECT  
Empleados.Nombre, 'Tarifa semanal: ', Empleados.TarifaHora * 40  
FROM Empleados  
WHERE Empleados.Cargo = 'Electricista'
```

Simplificando las consultas usando alias en las tablas

Calificar las columnas con los nombres de las tablas puede consumir tiempo, sobre todo cuando las tablas tienen nombres largos. Se puede utilizar **alias** para re-nombrar las tablas.

```
SELECT emp.empno, emp.ename, emp.deptno, dept.deptno, dept.loc  
FROM emp INNER JOIN dept ON emp.deptno = dept.deptno;
```

```
SELECT e.empno, e.ename, e.deptno, d.deptno, d.loc  
FROM emp e INNER JOIN dept d ON e.deptno = d.deptno;
```

En el ejemplo la tabla EMP toma el nombre de **e** y en la tabla DEPT el nombre de **d**.

2.2. Combinación de tablas

UNION CON JOIN

Se utiliza un JOIN para consultar datos en más de una tabla.

```
SELECT tabla1.columna, tabla2.columna  
FROM tabla1 INNER JOIN tabla2 ON tabla1.columna1 = tabla2.columna2
```

Escribe la condición-join en la cláusula FROM utilizando ON

Incluye el nombre de la tabla como prefijo del nombre de la columna, cuando el nombre de la columna aparezca en dos o más tablas.

Definición de JOIN

Cuando se requieren datos que están en más de una tabla, se requiere utilizar una condición-join. Los renglones de una tabla pueden ser “unidos” (joined) a los de otra tabla, solo si existen en ambas tablas un atributo que sea común entre ellas, normalmente, una relación de *llave primaria y llave foránea*.

Para desplegar datos de dos o más tablas que están relacionadas, escribe una simple condición-join es la cláusula FROM.

Sintaxis:

Table.column

Denota la tabla y columna donde los datos serán recuperados.

tabla1.columna1 = tabla2.columna2

Es la condición que “junta” relaciona (join) las tablas.

Cuando se escriba una columna que relacione dos tablas, es importante preceder cada columna con el nombre de su respectiva tabla, por claridad y mejorar el acceso a la BD.

Si el mismo nombre de la columna aparece en más de una tabla, el nombre de la columna **debe** ser precedido por el nombre de la tabla.

Para seleccionar *n* tablas, se necesita realizar el mínimo de *n-1 condiciones-join*.

Si se relaciona cuatro tablas se requieren hacer *tres condiciones-join*.

Algunos manejadores de BD no incluyen en su sintaxis las palabras INNER JOIN, JOIN, LEFT OUTER JOIN. Que pueden ser utilizadas directamente en la cláusula FROM junto con el indicador ON *condicion-join*. Por lo que el método tradicional para hacer un join es colocar las tablas afectadas en la cláusula FROM separadas por comas y la *condicion-join*

colocarla en la cláusula WHERE. Estas dos formas de crear joins generan exactamente el mismo resultado, solo que el método anterior (colocando la palabra join en la cláusula FROM) permite mayor comodidad y claridad para otras condiciones adicionales que afecten al resultado de la consulta.

Implementando un join tradicional

Esta es la sintaxis general para crear un join.

Se puede crear un join utilizando la condición-join en la cláusula WHERE.

```
SELECT      tabla1.columna, tabla2.columna
FROM        tabla1, tabla2
WHERE       tabla1.columna1 = tabla2.columna2;
```

Tipos de join

Existen dos tipos de Joins

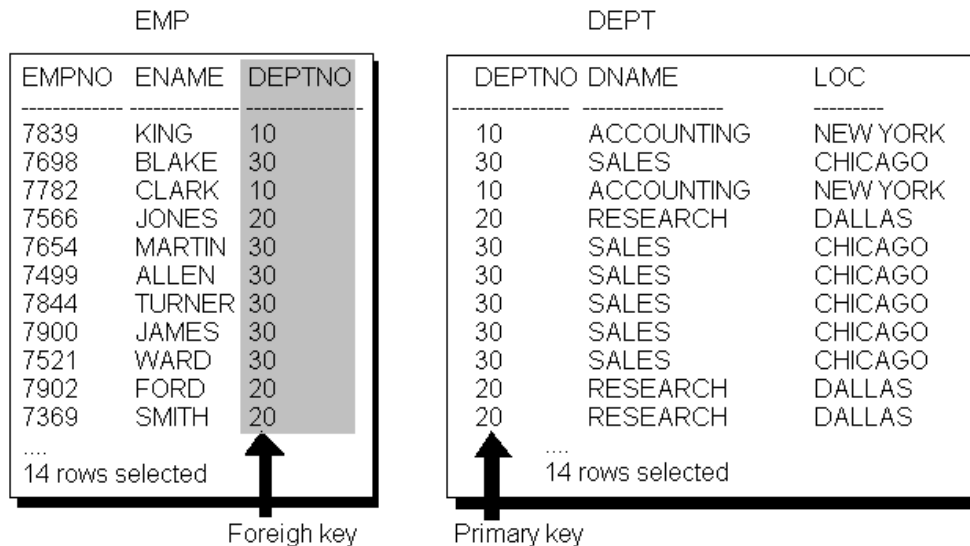
- Equijoins (inner join)
- Non-equijoins

Existen joins adicionales

- Outer Joins (left | right join)
- Self Joins

Equijoins

Para determinar el nombre de los departamentos a los que pertenecen cada empleado, se necesita comparar el valor en la columna DEPNO de la tabla EMPLEADO con los valores de DEPTNO en la tabla DEPARTAMENTO. La relación entre las tablas EMPLEADO y DEPARTAMENTO son conocidas como un equijoin, es decir, los valores de las columnas DEPTNO en ambas tablas deben coincidir.



Los equijoins son también llamados simplemente joins o inner joins.

```
SELECT emp.empno, emp.ename, emp.deptno, dept.deptno, dept.loc
```

FROM emp INNER JOIN dept ON emp.deptno = dept.deptno;

empno	ename	deptno	deptno	loc
7369	SMITH	20	20	DALLAS
7499	ALLEN	30	30	CHICAGO
7521	WARD	30	30	CHICAGO
7566	JONES	20	20	DALLAS

.....
14 rows selected.

En el ejemplo:

- La cláusula SELECT especifica las columnas a recuperar:
 - Número, nombre del empleado y número del departamento al que está asignado, tomados de la tabla EMP.
 - Número del departamento y localización que son columnas de la tabla DEPT
- La cláusula FROM especifica las tablas que deberán ser accedidas y el tipo de join que se efectuará entre ellas (INNER JOIN):
 - Tabla DEPT
 - Tabla EMP
 - En la cláusula ON se especifica la condición join: **emp.deptno = dept.deptno;**

Como la columna DEPTNO es común en ambas tablas debe ser calificado con el nombre de la tabla respectiva.

*SELECT emp.empno, emp.ename, emp.deptno, dept.deptno, dept.loc
FROM emp, dept
WHERE emp.deptno = dept.deptno;*

empno	ename	deptno	deptno	loc
7369	SMITH	20	20	DALLAS
7499	ALLEN	30	30	CHICAGO
7521	WARD	30	30	CHICAGO
7566	JONES	20	20	DALLAS

.....
14 rows selected.

En el ejemplo:

- La cláusula FROM especifica las tablas que deberán ser accedidas sin especificar el tipo de join que se efectuará entre ellas:
 - Tabla DEPT
 - Tabla EMP
- La cláusula WHERE especifica como las tablas realizarán el join en este caso es un equi-join: **emp.deptno = dept.deptno;**

Condiciones adicionales

Además de la condición-join, se puede indicar otros criterios de búsqueda en la cláusula WHERE por ejemplo, mostrar el número y nombre del empleado, número y localización del departamento para el empleado KING.

EMP			DEPT		
EMPNO	ENAME	DEPTNO	DEPTNO	DNAME	LOC
7839	KING	10	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	30	SALES	CHICAGO
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7566	JONES	20	20	RESEARCH	DALLAS
7654	MARTIN	30	30	SALES	CHICAGO
7499	ALLEN	30	30	SALES	CHICAGO
7844	TURNER	30	30	SALES	CHICAGO
7900	JAMES	30	30	SALES	CHICAGO
7521	WARD	30	30	SALES	CHICAGO
7902	FORD	20	20	RESEARCH	DALLAS
7369	SMITH	20	20	RESEARCH	DALLAS
....					
14 rows selected			14 rows selected		

De la siguiente manera:

```
SELECT empno, ename, emp.deptno, loc
FROM emp INNER JOIN dept ON emp.deptno = dept.deptno
WHERE ename = 'KING';
```

EMPNO	ENAME	DEPTNO	LOC
7839	KING	10	NEW YORK

O bien, utilizando la forma tradicional:

```
SELECT empno, ename, emp.deptno, loc
FROM emp, dept
WHERE emp.deptno = dept.deptno AND ename = 'KING';
```

Reuniendo mas de dos tablas

En ocasiones se necesitará hacer join con mas de dos tablas. Por ejemplo para mostrar el nombre, las órdenes y los ítems, el total de cada orden para el cliente TKB SPORT SHOP, se necesitará reunir las tablas CUSTOMER, ORD y ITEM.

CUSTOMER		ORD		
NAME	CUSTID	CUSTID	ORDID	ITEM
JOCKSPORTS	100	101	610	
TKB SPORT SHOP	101	102	611	
VOLLYRITE	102	104	612	
JUST TENNIS	103	106	601	
K+T SPORTS	105	102	602	
SHAPE UP	106	106		
WOMENS SPORTS	107	106		
.....			
9 rows selected		21 rows selected		

ORDID	ITEMID
610	3
611	1
612	1
601	1
602	1
.....	
64 rows selected	

La consulta quedaría de la siguiente manera:

```
SELECT c.name, o.ordid, i.itemid, i.itemtot, o.total
FROM customer c JOIN ord o ON c.custid = o.custid
      JOIN item I ON o.ordid = i.ordid
WHERE c.name = 'TKB SPORT SHOP';
```

El resultado sería:

NAME	ORDID	ITEMID	ITEMTOT	TOTAL
TKB SPORT SHOP	610	3	58	101.4
TKB SPORT SHOP	610	1	35	101.4
TKB SPORT SHOP	610	2	8.4	101.4

O en su forma tradicional:

```
SELECT c.name, o.ordid, i.itemid, i.itemtot, o.total
FROM customer c, ord o, item i
WHERE c.custid = o.custid
AND o.ordid = i.ordid
AND c.name = 'TKB SPORT SHOP';
```

El operador UNION

Combina los resultados de dos o mas consultas a un solo conjunto de resultados que consta de todas las filas que pertenecen a todas las consultas en la unión. Es diferente a usar JOIN que combinan columnas de 2 o mas tablas de diferentes tipos.

Las reglas básicas para combinar el resultado de 2 consultas con UNION es:

- se deben indicar en todas las consultas el orden de las columnas
- los tipos de datos deben ser compatibles

```
< consulta >
UNION [ ALL ]
< consulta >
[ UNION [ ALL ] < consulta >
[ ...n ] ]
```

Argumentos

< consulta >

Es una consulta que regresa los renglones a unir con los renglones de otra consulta y así sucesivamente. Las definiciones de las columnas que son parte de la operación UNION no tienen que ser idénticos pero si compatibles.

La tabla muestra las reglas de comparación con tipos de datos.

Tipos de datos en las columnas	Tipos de datos en el resultado
Ambas son char con longitud L1 y L2.	Tipo de datos resultante es char con longitud igual al mayor de L1 y L2.
Ambas son varchar con longitud L1 y L2.	Tipo de datos resultante es varchar con longitud igual al mayor de L1 y L2.
Ambos son tipos de datos numéricos (por ejemplo, smallint , int , float , Money , numeric).	El tipo de datos es igual a la máxima precisión de las 2 columnas. Por ejemplo, si una columna de la tabla A es de tipo int y una columna de la tabla B es float , entonces el tipo de datos de la tabla resultante es float porque float es más preciso que int .
Ambas columnas especifican NOT NULL.	El resultado especifica NOT NULL.

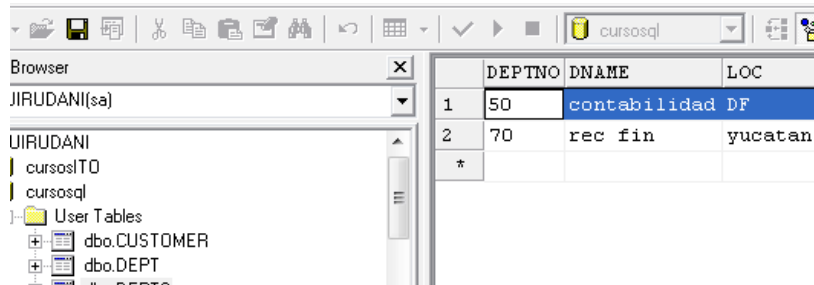
UNION: Combinación de múltiples conjuntos, regresando uno solo.

ALL: Añade todos los renglones al resultado, incluyendo duplicados. Si no se especifica, se eliminan los duplicados.

Ejemplo de una vista creada a partir de la unión de dos tablas, por lo pronto se describirá que las vistas se crean con la instrucción CREATE VIEW nombreDeLaVista, en la Unidad 3 se presentará una explicación más detallada de las vistas.

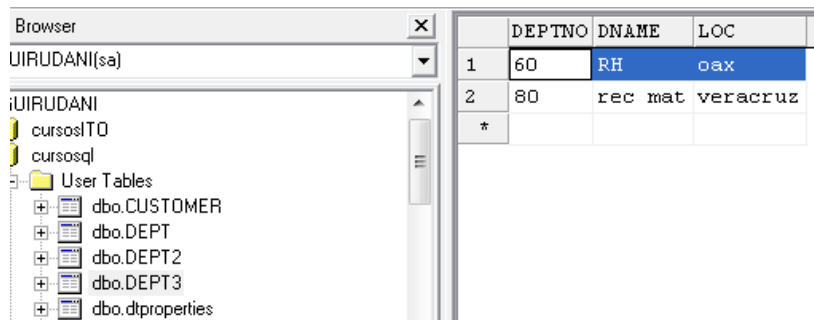
```
create view dept4
as select *
from dept2 union
select *
from dept3
```

La tabla dept2 contiene:



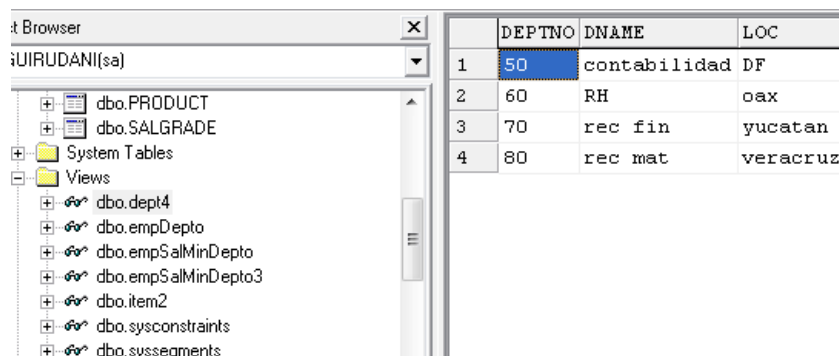
	DEPTNO	DNAME	LOC
1	50	contabilidad DF	
2	70	rec fin	yucatan
*			

La tabla dept3 contiene:



	DEPTNO	DNAME	LOC
1	60	RH	oax
2	80	rec mat	veracruz
*			

La vista resultante de la unión contiene:



	DEPTNO	DNAME	LOC
1	50	contabilidad DF	
2	60	RH	oax
3	70	rec fin	yucatan
4	80	rec mat	veracruz

El número de atributos de las tablas subyacentes deben coincidir, por ejemplo se tiene la siguiente vista:

```
create view dept5
as select deptno, dname
from dept2 union
select deptno
from dept3
```

El sistema devuelve el siguiente error:

```
create view dept5
as select deptno, dname
   from dept2 union
   select deptno |
   from dept3
```

Server: Msg 8157, Level 16, State 1, Procedure dept5, Line 2
All the queries in a query expression containing a UNION operator must have the same number of expressions in their select lists

La Intersección de dos tablas.

En SQL Server no se contempla el operador Intersect sino hasta la versión 2005. Esta operación de intersección se puede resolver de la siguiente manera:

```
create view interseccion
as select distinct deptno
from emp
where deptno in ( select distinct deptno from emp where ename like '%T%')
               and deptno in (select distinct deptno from emp where ename like '%r%')
```

Regresa la intersección de dos tablas (EMP), elimina los duplicados antes de la intersección.

La consulta anterior realiza la intersección de la Tabla EMP con el atributo deptno de dos consultas, en la primera se obtienen los departamentos de los empleados en cuyo nombre haya una T y aquellos empleados en cuyo nombre haya una R, el resultado es una tabla con un atributo y los departamentos 20 y 30.

```
USE Northwind
GO
DECLARE @MyProduct int
SET @MyProduct = 10
IF (@MyProduct <> 0)
    SELECT *
    FROM Products
    WHERE ProductID = @MyProduct
GO
```

La Operación EXCEPT

En SQL Server no se contempla el operador Except sino hasta la versión 2005. Esta operación de intersección se puede resolver de la siguiente manera:

```
create view diferencia
as
select distinct deptno
from emp
where deptno not in ( select distinct deptno from emp where ename like '%T%')
                   and deptno in (select distinct deptno from emp where ename like '%r%')
```

Regresa la diferencia entre dos consultas a la tablas (EMP), elimina los duplicados antes de la intersección.

La consulta anterior realiza la diferencia de la Tabla EMP con el atributo deptno de dos consultas, se toman los elementos de la segunda consulta que no estén en la primera.

2.3. Predicados IN, IS NULL, BETWEEN, AND, OR, NOT, LIKE

El operador IN

Para verificar si un valor específico se encuentra en una lista se utiliza el operador IN.

```
SELECT empno, ename, sal, mgr
FROM emp
WHERE mgr IN (7902, 7566, 7788);
```

empno	ename	sal	mgr
7902	FORD	3000	7566
7369	SMITH	800	7902
7788	SCOTT	3000	7566
7876	ADAMS	1100	7788

El ejemplo anterior muestra el número de empleado, el nombre, el salario y su respectivo manager para aquellos empleados cuyos jefes sean 7902, 7566 o 7788.

El operador IN puede ser usado con otros tipos de datos. El siguiente ejemplo regresa los empleados cuyos nombres estén en la lista.

```
SELECT empno, ename, mgr, deptno
FROM emp
WHERE ename IN ('FORD', 'ALLEN');
```

El operador LIKE

Se puede seleccionar renglones que coincidan con un patrón de caracteres utilizando el operador LIKE. Se pueden utilizar dos caracteres (comodines) para la realización de búsquedas con patrones.

```
SELECT ename
FROM emp
WHERE ename LIKE 's%';
```

La consulta regresa el nombre del empleado para aquellos cuyo nombre inicie con una "S". Nombres que inicien con una "s" no serán mostrados.

Símbolo	Descripción
%	Representa una secuencia de cero o más caracteres
_	Representa un solo carácter

Pueden utilizarse los símbolos y comodines % y _ en combinación para hacer mas exacta la búsqueda.

```
SELECT ename
FROM emp
WHERE ename LIKE '_A%';
```

ENAME
JAMES
WARD
MARTIN

En el ejemplo se obtienen aquellos empleados que en el nombre tengan como segunda letra una A.

La función IS NULL

Convierte un valor nulo a un valor indicado

Sintaxis:

ISNULL (expr1, expr2)

Donde: expr1: es el valor o expresión que contiene valores nulos

expr2: es el valor por el que se reemplazará NULL

Por ejemplo:

```
SELECT ename, sal, comm, (sal * 12) + ISNULL(comm,0)
FROM emp;
```

ENAME	SAL	COMM	
SMITH	800.00		9600.00
ALLEN	1600.00	300.00	19500.00
WARD	1250.00	500.00	15500.00
JONES	2975.00		35700.00
MARTIN	1250.00	1400.00	16400.00
...			
14 rows selected.			

En el ejemplo, se calcula el sueldo anual mas la comisión, los empleados que no ganan comisión se reemplaza el valor nulo por cero, por lo que el resultado se muestra en el ejemplo. En caso contrario no se utiliza la función ISNULL el resultado no es el mismo (véase la siguiente consulta).

```
SELECT ename, 12*sal+comm
FROM emp;
```

ename	
...	
KING	NULL

- Tipos de datos comunes que se utilizan para evitar un NULL son: date, carácter y numeric.
- Ejemplos:
 - ISNULL (comm, 0)

- ISNULL (hiredate, GETDATE())
- ISNULL (job, 'No job yet')

La función IS NULL obliga a las funciones a incluir los valores NULOS.

```
SELECT AVG(ISNULL(comm, 0))
FROM emp;
```

```
-----
157.14286
```

En el ejemplo, el promedio salarial se calcula en base a todos los renglones ya sea que tengan o no valores nulos en la columna COMM.

El predicado BETWEEN

Se puede mostrar renglones basados en rangos de valores utilizando el operador BETWEEN. Los rangos que especifique contienen límites inferior y superior.

```
SELECT ename, sal
FROM emp
WHERE sal BETWEEN 1000 AND 1500;
```

ename	sal	Límite Inferior	Límite Superior
MARTIN	1250		
TURNER	1500		
WARD	1250		
ADAMS	1100		
MILLER	1300		

El ejemplo anterior obtiene los empleados que ganen entre \$1000 y \$1500 inclusive.

El operador AND

AND requiere que ambas condiciones sean TRUE.

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
AND job = 'CLERK';
```

empno	ename	job	sal
7876	ADAMS	CLERK	1100
7934	MILLER	CLERK	1300

En el ejemplo ambas condiciones deben ser verdaderas para que un registro sea mostrado. De tal forma que, si un empleado tiene el puesto de CLERK y gana más de \$1100 será seleccionado.

El operador OR

OR requiere que al menos una condición sea TRUE.

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 2000
      OR job = 'CLERK';
```

empno	ename	job	sal
7039	KING	PRESIDENT	5000
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7505	JONES	MANAGER	2975
...			
(10 row(s) affected)			

En el ejemplo, ya sea que aquellos empleados que ganen de \$2000 en adelante o tengan el puesto CLERK serán seleccionados.

El operador NOT

```
SELECT ename, job
FROM emp
WHERE job NOT IN ('CLERK', 'MANAGER', 'ANALYST');
```

ename	job
KING	PRESIDENT
MARTIN	SALESMAN
ALLEN	SALESMAN
TURNER	SALESMAN
WARD	SALESMAN

En el ejemplo se muestran los nombres y puestos de los empleados que no se encuentran en la lista ('CLERK', 'MANAGER', 'ANALYST').

El operador NOT puede ser utilizado con otros operadores de SQL tales como BETWEEN, LIKE y NULL.


```
... WHERE NOT job IN ('CLERK', 'ANALYST')  
... WHERE sal NOT BETWEEN 1000 AND 1500  
... WHERE ename NOT LIKE '%A%'  
... WHERE comm IS NOT NULL
```

Eliminando los renglones duplicados

Para eliminar renglones duplicados, se incluye la palabra **DISTINCT** en la cláusula **SELECT** inmediatamente después de la palabra **SELECT**.

```
SELECT DISTINCT deptno  
FROM emp;
```

```
deptno  
-----  
10  
20  
30
```

En el ejemplo anterior, la tabla **EMP** tiene catorce registros pero solo existen tres departamentos diferentes.

Se puede utilizar **DISTINCT** antes de varias columnas. El calificador **DISTINCT** afecta a todas las columnas seleccionadas y el resultado es una combinación diferente de las columnas.

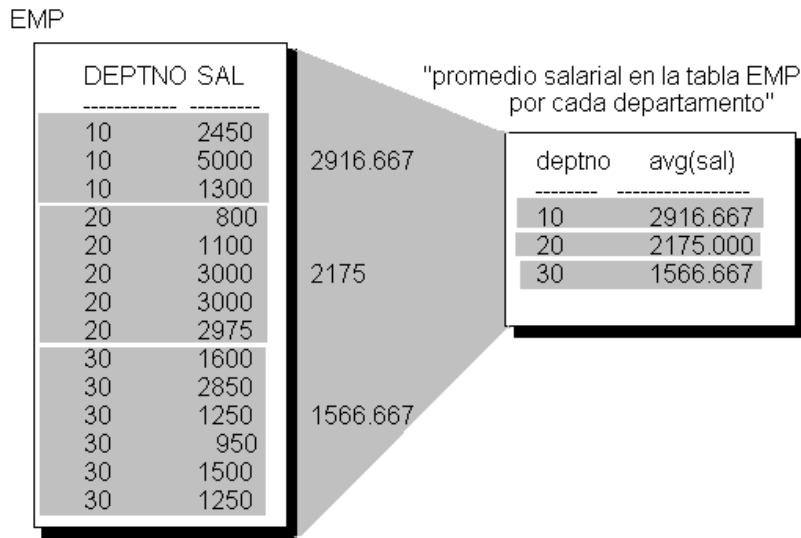
```
SELECT DISTINCT deptno, job  
FROM emp;
```

```
deptno  job  
-----  -  
10      CLERK  
10      MANAGER  
10      PRESIDENT  
20      ANALYST  
...  
(9 row(s) affected)
```

2.4. Cláusulas **ORDER BY**, **GROUP BY**

Hasta ahora, todas las funciones de grupo tratan a la tabla como un solo grupo de información. En ocasiones, se necesitará dividir la tabla en pequeños grupos de información. Esto se puede realizar utilizando la cláusula **GROUP BY**.

GROUP BY reorganiza en el sentido lógico la tabla representada por la cláusula **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**.



Se puede utilizar la cláusula **BROUP BY** para dividir en pequeños grupos de información una tabla. Entonces se puede utilizar las funciones de grupo para resumir la información de estos grupos.

```
SELECT columna,funcion_de_agrupacion (columna)
FROM tabla
[WHERE condicion]
[GROUP BY expresion group_by]
[ORDER BY column];
```

Expresión group_by especifica las columnas por las que se efectuará el agrupamiento de los renglones de la tabla.

- Utilizando WHERE se puede pre-excluir renglones antes de ser divididos en grupos
- No se puede utilizar alias de columnas en GROUP BY, debe ser el nombre del atributo definido en la tabla base
- Por defecto, los renglones son ordenados ascendentemente por las columnas especificadas en GROUP BY. Se puede alterar este orden utilizando ORDEN BY

Cuando se utilice la cláusula GROUP BY se debe asegurar de que las columnas en la lista de SELECT que no estén en funciones de grupo se encuentren en la cláusula GROUP BY. El ejemplo muestra el número de departamento y el promedio salarial por cada departamento.

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno;
```

deptno	
10	2916.6667
20	2175.0000
30	1566.6667

Las columnas que aparecen en GROUP BY no necesariamente deben aparecer en la lista de SELECT.

```
SELECT AVG(sal)
FROM emp
GROUP BY deptno;
```

2916.6667
2175.0000
1566.6667

En ciertas ocasiones se necesitará ver los resultados de grupos más pequeños dentro de los mismos grupos. El ejemplo muestra un reporte que despliega el salario total para cada puesto, dentro de cada departamento.

EMP

DEPTNO	JOB	SAL
10	MANAGER	2450
10	PRESIDENT	5000
10	CLERK	1300
20	CLERK	800
20	CLERK	1100
20	ANALYST	3000
20	ANALYST	3000
20	MANAGER	2975
30	SALESMAN	1600
30	SALESMAN	1250
30	SALESMAN	1500
30	SALESMAN	1250
30	MANAGER	2850
30	CLERK	850

"sumar los salarios de la tabla EMP por puesto, agrupados por departamento"

DEPTNO	JOB	SUM(SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2950
30	SALESMAN	5600

Se pueden regresar valores por grupos y subgrupos, listando más de una columna en GROUP BY.

A continuación se presenta un ejemplo:

```
SELECT deptno, job, SUM(sal) 'Sum'
FROM emp
GROUP BY deptno, job;
```

deptno	job	Sum
10	CLERK	13000
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
...		
9 rows selected.		

La instrucción SELECT del ejemplo se evalúa de la siguiente forma:

- La cláusula SELECT especifica las columnas a mostrar
- La cláusula FROM indica de donde se tomarán los datos
- La cláusula GROUP BY especifica como agrupar los renglones:
 - Primero, los renglones son agrupados por número de departamento
 - Segundo, dentro de cada grupo del mismo departamento, los renglones son reagrupados por puesto.

De esta forma, la suma se aplica al salario para cada puesto dentro de un mismo departamento.

La cláusula ORDER BY

El orden en que se muestran los renglones de una tabla no está definido, de hecho aparecen en el orden en el que los registros fueron almacenados por primera vez en la tabla. Se ordena la salida con la cláusula ORDER BY

- ASC orden ascendente por defecto
- DESC orden descendente

La cláusula ORDER BY siempre debe ser la ultima en una instrucción SELECT.

```
SELECT ename, job, deptno, hiredate
FROM emp
ORDER BY hiredate;
```

ename	job	deptno	hiredate
SMITH	CLERK	20	1980-12-17 00:00:00.000
ALLEN	SALESMAN	30	1981-02-20 00:00:00.000
...			
(14 rows(s) affected)			

Se puede especificar una expresión o un alias para ordenar.

```
SELECT expr
FROM table
[WHERE condition(s)]
[ORDER BY {column, expr} [ASC|DESC]];
```

Donde:

ORDER BY especifica el orden en el que serán mostrados los renglones.

column, expr el o los atributos por los que se va a ordenar

AS ordena en forma ascendente, este ordenamiento es por defecto

DESC ordena en forma descendente

El orden por defecto es ascendente:

- Los valores numéricos son mostrados del menor al mayor
- Las fechas son mostradas con el valor de la fecha más pasada, por ejemplo: 01-ENE-92 es primero que 01-ENE-95
- Las cadenas de caracteres son desplegadas en orden alfabético.
- Los valores nulos aparecen al principio cuando es ascendente y al final cuando es descendente el ordenamiento

```
SELECT ename, job, deptno, hiredate
FROM emp
ORDER BY hiredate DESC;
```

ename	job	deptno	hiredate
ADAMS	CLERK	20	1983-01-12 00:00:00.000
SCOTT	ANALYST	20	1982-12-09 00:00:00.000
....			
(14 rows(s) affected)			

Ordenando con alias

Se puede utilizar una columna renombrada con un alias en la cláusula ORDER BY.

```
SELECT empno, ename, sal*12, ann_sal
FROM emp
ORDER BY ann_sal;
```

empno	ename	ann_sal
7369	SMITH	9600
7900	JAMES	11400
7876	ADAMS	13200
7654	MARTIN	15000
7521	WARD	15000
7934	MILLER	15600
7844	TURNER	18000
...		
(14 row(s) affected)		

El ejemplo ordena los datos por el salario anual.

Se puede utilizar alias con espacios en blancos y este alias puede aparecer en una cláusula ORDER BY.

Por ejemplo:

```
SELECT sal * 12 AS "Sal. Anual"
FROM emp
ORDER BY "Sal. Anual";
```

Ordenando con múltiples columnas

Se puede ordenar los resultados de la consulta con una o mas columnas, las cuales se toman como criterios de ordenamiento, el límite es el número de columnas que tenga la tabla.

En una cláusula ORDER BY, se especifica las columnas separadas or comas, si se desea cambiar el orden por defecto se usa DESC después de cada columna que se desee cambiar el orden.

```
SELECT ename, deptno, sal
FROM emp
ORDER BY deptno, sal DESC;
```

ename	deptno	sal
KING	10	5000
CLARK	10	2450
MILLER	10	1300
FORD	20	3000
...		
(14 row(s) affected)		

En el ejemplo, el primer criterio de ordenamiento es por número de departamento (en forma ascendente, por defecto) y luego por salario en forma descendente.

Nota. Si se utiliza mas de un criterio de ordenamiento, se debe tomar en cuenta que el primer criterio debe contener valores repetidos para que la consulta tenga sentido. Como en el ejemplo siguiente:

```
SELECT *
FROM emp
ORDER BY empno, deptno, sal DESC;
```

No tiene sentido ya que el primer criterio utilizado, es un atributo que no contiene valores repetidos, para que los demás criterios puedan afectar el resultado.

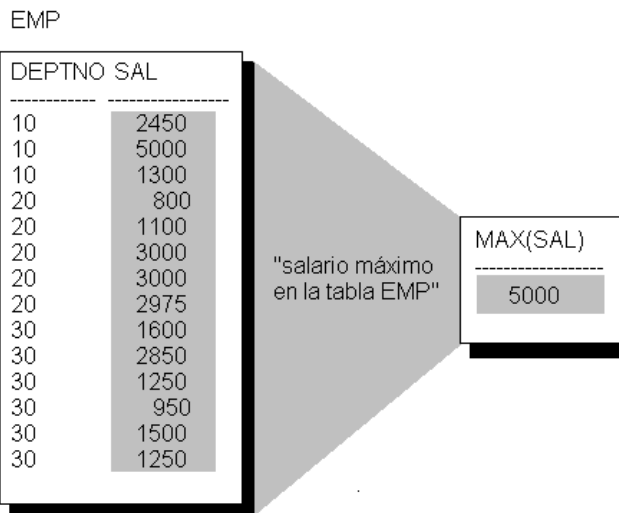
Se puede utilizar funciones de grupo en ORDER BY.

```
SELECT deptno, AVG(sal) 'AVG'
FROM emp
GROUP BY deptno
ORDER BY AVG(sal);
```

DEPTNO	AVG
30	1566.6667
20	2175.0000
10	2916.6667

2.5. Funciones agregadas COUNT, SUM, MAX, MIN, AVG

Las funciones para grupos operan con un conjunto de renglones para devolver un solo resultado.



A diferencia de las funciones de renglón simples, las funciones para grupos, operan con un conjunto de renglones para obtener un resultado por grupo. Este conjunto de renglones debe ser o bien toda la tabla o la tabla dividida en grupos.

Cada función acepta un argumento. La siguiente tabla identifica las opciones que se pueden utilizar:

Función	Descripción
AVG([DISTINCT ALL] <i>n</i>)	Valor promedio de <i>n</i> , ignora valores null
COUNT({*[DISTINCT ALL] <i>expr</i> })	Número de renglones, donde <i>expr</i> valore otro valor diferente de null. Cuenta todos los renglones seleccionados utilizando *, incluyendo renglones duplicados con valores nulos.
MAX([DISTINCT ALL] <i>expr</i>)	Valor máximo de <i>expr</i> , ignorando valores null
MIN([DISTINCT ALL] <i>expr</i>)	Valor mínimo de <i>expr</i> ., ignorando valores null
SUM([DISTINCT ALL] <i>n</i>)	Suma los valores de <i>expr</i> , ignorando valores null

Guía para el uso de funciones de grupos

DISTINCT realiza la función de considerar solo aquellos renglones no duplicados; **ALL** considera los renglones duplicados. Por default es **ALL** y no necesita ser especificado. Todas las funciones de grupo excepto **COUNT(*)** ignoran los valores null. Para sustituir los valores null, se utiliza la función **ISNULL**.

Se presenta en formato de una consulta que usa función de grupo:

```
SELECT columna, funcion_de_grupo(columna)
FROM tabla
[WHERE condicion]
[ORDER BY column];
```

Utilizando las funciones AVG, MAX, MIN y SUM

Se puede utilizar las funciones **AVG**, **MAX**, **MIN** y **SUM** con columnas que almacenan datos numéricos. El ejemplo siguiente muestra el promedio, el máximo, el mínimo y la suma de los salarios de los empleados que son vendedores.

```
SELECT AVG(sal) 'AVG', MAX(sal) 'MAX',
       MIN(sal) 'MIN', SUM(sal) 'SUM'
FROM emp
WHERE job LIKE 'SALES%';
```


El resultado de la consulta se presenta a continuación:

AVG	MAX	MIN	SUM
1400	1600	1250	5600

Se puede utilizar las funciones MAX y MIN con otros tipos de datos.

```
SELECT MIN(hiredate) "MIN DATE"
       MAX(hiredate) "MAX DATE"
FROM emp
```

MIN DATE	MAX DATE
1980-12-17 00:00:00.000	1983-01-12 00:00:00.000

El ejemplo muestra el más reciente y más antiguo empleado.

```
SELECT MIN(ename), MAX(ename)
FROM emp;
```

ADAMS WARD

Nota: las funciones AVG y SUM solo pueden ser utilizadas con datos numéricos.

Utilizando la función COUNT

La función COUNT tiene dos formatos:

- COUNT (*)
- COUNT (expr.)

COUNT (*) regresa el número de renglones en una tabla, incluyendo renglones que contengan valores null.

A diferencia de COUNT (expr), esta regresa el número de renglones no nulos en la columna identificada por expr.

En el ejemplo siguiente, la consulta regresa el número de empleados en el departamento 30.

```
SELECT COUNT(*)
FROM emp
WHERE deptno = 30;
```

6

COUNT (expr) regresa el número de renglones non-null.

En el ejemplo siguiente se muestra el número de empleados del departamento 30 que ganan una comisión. Nótese que para el resultado obtenido, el total de renglones es de 4 debido a que existen empleados que no ganan una comisión contienen un valor null en su columna **comm**.

```
SELECT COUNT(columna)  
FROM emp  
WHERE deptno = 30;
```

4

Otro ejemplo

Mostrar el número de departamentos en la tabla EMP

```
SELECT COUNT(deptno)  
FROM emp;
```

14

Mostrar el número de departamentos distintos

```
SELECT COUNT(DISTINCT deptno)  
FROM emp;
```

3

Funciones de grupo y valores NULL

Todas las funciones, excepto COUNT (*) ignoran los valores nulos en una columna. En el ejemplo, el promedio es calculado basado solo en los renglones que tienen valores válidos almacenados en la columna COMM.

El promedio es calculado como la suma de todas las comisiones dividido entre el número de empleados (4).

```
SELECT AVG(comm)  
FROM emp;
```

550

Utilizando la función ISNULL

La función ISNULL obliga a las funciones para que incluyan los valores nulos. En el siguiente ejemplo, el promedio salarial es calculado en base a todos los renglones, ya sea que tengan o no valores nulos en la columna COMM.

```
SELECT AVG (ISNULL(comm,0))  
FROM emp;
```

```
-----  
157.14286
```

2.6. Subconsultas

Una subconsulta es una instrucción SELECT que está dentro de otra instrucción SELECT. Se pueden construir instrucciones poderosas con tan solo utilizar subconsultas. Pueden ser muy útiles cuando se necesite seleccionar renglones de una tabla con una condición que depende de datos en la misma tabla.

```
SELECT    select_list  
FROM      table  
WHERE     expr. Operator  
          (SELECT    select_list  
            FROM      table)
```

La subconsulta interna (inner query) se ejecuta una vez antes de la consulta principal. El resultado de la subconsulta lo utiliza la consulta principal (outer query). Se puede colocar una subconsulta en las siguientes cláusulas:

- Cláusula WHERE
- Cláusula HAVING
- Cláusula FROM

En la sintaxis:

Operador incluye un operador de comparación tal como: >, =, IN.

Nota: los operadores de comparación se dividen en dos tipos:

- operadores de renglón simple (>, =, >=, <, <>, <=)
- operadores de renglón múltiple (IN, ALL, ANY).

Una subconsulta es frecuentemente llamada un SELECT anidado. La subconsulta interna generalmente se ejecuta primero y su salida se usa para completar la condición de la consulta principal.

Utilizando una subconsulta para resolver un problema

¿Quién gana un salario mayor al de Jones?

Supóngase que se quiere escribir una consulta para encontrar a los empleados que ganan más que el salario de **Jones**.

Para resolver este problema, se necesitan dos consultas: una para encontrar el salario de **Jones** y otra para encontrar quien gana más que este salario.

Se puede resolver este problema combinando dos consultas, colocando una consulta interna dentro de otra.

Una consulta interna o subconsulta regresa un valor que es utilizado por una consulta externa o consulta principal. Usar una subconsulta es equivalente a ejecutar dos consultas secuenciales y utilizar el resultado de la primera consulta como búsqueda en la segunda consulta.

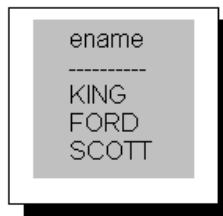
Guía para el uso de subconsultas

- encerrar las subconsultas en paréntesis
- colocar las subconsultas de lado derecho del operador de comparación
- no incluir una cláusula ORDER BY a la subconsulta. Sólo puede existir una cláusula ORDER BY por instrucción, si se utiliza esta debe ser la última cláusula del SELECT principal

Por ejemplo:

```
SELECT ename  
FROM emp  
WHERE sal > (SELECT sal  
             FROM emp  
             WHERE empno = 7566)
```

El resultado se presenta a continuación:



ename
KING
FORD
SCOTT

En el ejemplo, la consulta interna determina el salario del empleado 7566. La consulta externa toma el resultado de la consulta interna (regresa 2975) y lo utiliza para desplegar a todos los empleados que ganen más que esta cantidad.

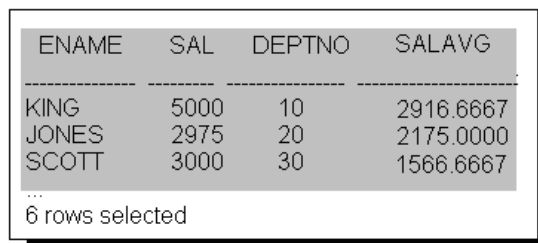
Utilizando una subconsulta en la cláusula FROM

Se puede usar una subconsulta en la cláusula FROM de la instrucción SELECT, la cual es muy similar a como se generan las vistas.

```
SELECT a.ename, a.sal, a.deptno, b.salavg  
FROM emp a, (SELECT deptno, AVG(sal) salavg  
            FROM emp  
            GROUP BY deptno) b
```

```
WHERE a.deptno = b.deptno  
AND a.sal > b.salavg;
```

El resultado sería:



ENAME	SAL	DEPTNO	SALAVG
KING	5000	10	2916.6667
JONES	2975	20	2175.0000
SCOTT	3000	30	1566.6667
...			

6 rows selected

El ejemplo muestra los nombres de empleado, salarios, números de departamentos y promedio salarial para todos los empleados que ganan más del salario promedio en su departamento.

La consulta que genera la tabla **b** tendrá los siguientes datos:

DEPTNO	SALAVG
10	2916.6667
20	2175.0000
30	1566.6667

Tipos de subconsultas

- **subconsultas de renglón-simple:** son consultas que regresan solo un valor en la instrucción SELECT de la consulta interna.
- **Subconsultas de múltiple renglón:** son consultas que regresan más de un renglón en la instrucción SELECT de la consulta interna.

Subconsultas de renglón-simple

Son consultas que regresan solo un valor en la instrucción SELECT de la consulta interna. Este tipo de subconsultas utiliza operadores de renglón simple.

Ejemplo: Se puede mostrar datos desde una consulta principal utilizando funciones de grupo en una subconsulta que regrese un solo renglón.

```
SELECT ename, job, sal
FROM emp
WHERE sal = (SELECT MIN(sal)
             FROM emp)
```

El resultado es el siguiente:

ENAME	JOB	SAL
SMITH	CLERK	800

El ejemplo obtiene el nombre del empleado, puesto y salario de todos los empleados cuyo salario sea igual al mínimo. La función de grupo MIN regresa un solo valor (800) a la consulta externa.

Una instrucción SELECT puede ser considerada como un bloque. El ejemplo muestra a todos los empleados cuyo puesto es el mismo que el del empleado 7369 y que ganan más que el salario del empleado 7876.

```
SELECT ename, job
FROM emp
WHERE job = (SELECT job
             FROM emp
             WHERE empno = 7369)
```

```
AND sal > ( SELECT sal
            FROM emp
            WHERE empno = 7876)
```

El resultado es el siguiente:

ENAME	JOB
MILLER	CLERK

El ejemplo consiste en tres consultas: *la consulta externa* y *dos consultas internas*. Las consultas internas son realizadas primero, produciendo los resultados: CLERK y 1100, respectivamente. La consulta externa procesa estos valores regresados por las consultas internas para completar su condición en la cláusula WHERE.

Ambas consultas internas que retornan un valor o renglón (CLERK y 1100) son conocidas como subconsultas de renglón simple.

Nota: las consultas internas y externas pueden obtener datos de diferentes tablas.

2.7. Operadores IN, EXIST, ANY, ALL aplicados a subconsultas de renglón múltiple

Las subconsultas que regresan más de un renglón se llaman *subconsultas de múltiple renglón* deben utilizar un operador de múltiple renglón en lugar de los operadores de renglón simple. Los operadores de renglón múltiple esperan uno o más valores.

El predicado IN

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

```
SELECT ename, sal, deptno
FROM emp
WHERE sal IN (SELECT MIN(sal)
              FROM emp
              GROUP BY deptno);
```

Ejemplo: encontrar a los empleados que ganen el mismo salario que el salario mínimo de los departamentos. La consulta interna se ejecuta primero, produciendo un resultado con tres renglones: 800, 950, 1300. La consulta principal procesa estos valores para completar su condición. De hecho, la consulta externa se convierte en:

```
SELECT ename, sal, deptno
FROM emp
WHERE sal IN (800,950,1300);
```

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

Usando el operador ALL en subconsultas de renglón múltiple

ALL

El predicado ALL se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Si se cambia ANY por ALL en el ejemplo anterior, la consulta devolverá únicamente aquellos productos cuyo precio unitario sea mayor que el de todos los productos vendidos con un descuento igual o mayor al 25 por ciento. Esto es mucho más restrictivo.

Forma parte de una consulta con predicado. Si no se incluye ninguno de los predicados se asume ALL. El Motor de base de datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL. No es conveniente abusar de este predicado ya que obligamos al motor de la base de datos a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados.

SELECT ALL FROM Empleados;

*SELECT * FROM Empleados;*

El operador ALL compara un valor con todos los valores regresados por una subconsulta.

```
SELECT empno, ename, job
FROM emp
WHERE sal > ALL (SELECT AVG(sal)
                FROM emp
                GROUP BY deptno)
```

empno	ename	job
7839	KING	PRESIDENT
7566	JONES	MANAGER
7902	FORD	ANALYST
7788	SCOTT	ANALYST

El ejemplo muestra a los empleados cuyo salario es mayor que el salario promedio de todos los departamentos. El salario promedio mayor de un departamento es \$2916.66, por lo tanto la consulta regresa aquellos empleados cuyo salario sea mayor que \$2916.66.

>ALL significa mayor que el maximo y <ALL significa menor que el mínimo

El operador NOT puede ser utilizado con los operadores IN, ANY y All.

Usando el operador ANY en subconsultas de renglón múltiple

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

El operador ANY compara un valor para cada valor regresado por la subconsulta.

```
SELECT empno, ename, job
FROM emp
WHERE sal >ANY (SELECT sal
                FROM emp
                WHERE job = 'CLERK')
                AND job <> 'CLERK';
```

El resultado sería:

empno	ename	job
7654	MARTIN	SALESMAN
7521	WARD	SALESMAN

El ejemplo muestra aquellos empleados cuyo salario es menor que cualquiera del puesto CLERK y que no tengan ese puesto. El salario máximo que ese puede ganar en un puesto CLERK es de \$1300. la instrucción SQL muestra a todos los empleados que no tienen el puesto de CLERK pero que ganan menos de \$1300.

<ANY significa menos que el máximo del grupo. >ANY significa mayor que el mínimo del grupo. =ANY es equivalente a IN.

Usando el operador EXISTS

EXIST

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

El operador EXIST, se puede utilizar con instrucciones SELECT anidadas. Este operador es frecuentemente utilizado para correlacionar subconsultas.

La consulta externa se ejecuta registro tras registro, tal y como lo hacen las consultas normales, pero por cada renglón de esta consulta, se evalúa el operador EXIST recorriendo la consulta interna renglón tras renglón hasta que algún renglón de la consulta interna cumpla la condición que se haya especificado. Si se encuentra un renglón en la consulta interna que cumpla la condición, el operador EXIST se evalúa a **verdadero**, sin tener que terminar de recorrer esta consulta interna. En caso contrario, si ningún renglón de esta consulta cumple la condición especificada, EXIST devuelve **falso** a la consulta eterna.

Si una subconsulta devuelve un valor:

- La búsqueda no continúa en la consulta interna
- La condición se evalúa como verdadera

Si una subconsulta no regresa un valor:

- La condición se evalúa a falsa
- La búsqueda no continúa en la consulta interna

De la misma forma NOT EXIST evalúa si no existe algún valor.

Ejemplo: encontrar los empleados que al menos tienen a una persona como subordinado.

```
SELECT empno, ename, job, deptno
FROM emp out
WHERE EXISTS (SELECT empno
              FROM emp inn
              WHERE inn.mgr = out.empno);
```

empno	ename	job	deptno
7566	JONES	MANAGER	30
7699	BLAKE	MANAGER	30
7782	CLARK	MANAGER	10
7788	SCOTT	ANALYST	20
7839	KING	PRESIDENT	10
7902	FORD	ANALYST	20

Resumiendo, el operador EXISTS asegura que la búsqueda en la consulta interna no continúe cuando al menos encuentra un renglón que cumpla la condición de que un MANAGER tenga al menos un empleado como subordinado.

De la misma forma NOT EXISTS evalúa si no existe algún valor.

Usando el operador NOT EXISTS

Encontrar los departamentos que no tienen empleados.

```
SELECT deptno, dname
FROM dept d
WHERE NOT EXISTS (SELECT '1'
                  FROM emp e
                  WHERE d.deptno = e.deptno);
```

deptno	dname
40	OPERATIONS

Nótese que el SELECT interno no necesita regresar un valor específico, se puede recuperar una literal. Utilizar esto para mejorar el desempeño de las consultas, es mas rápido seleccionar una constante que una columna.

```
SELECT deptno, dname
FROM dept
WHERE deptno NOT IN (SELECT deptno
                     FROM emp);
```

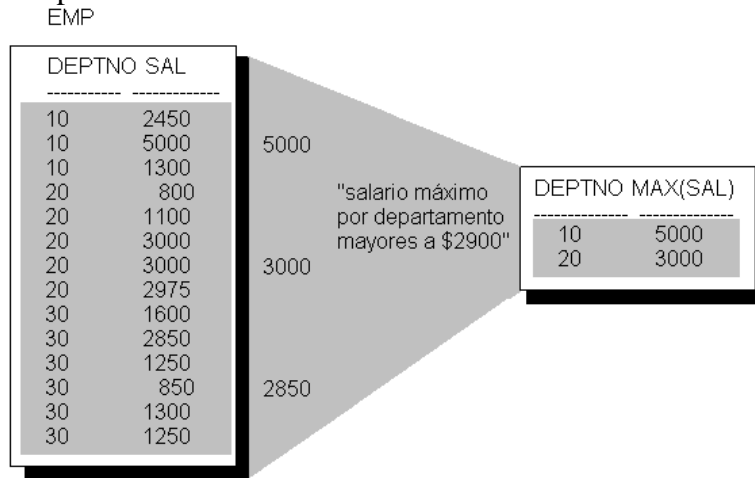
Solución alternativa

Como se mostró en el ejemplo anterior, un NOT IN puede ser utilizado como una alternativa al NOT EXIST. Sin embargo, NOT IN se evaluará a falso si cualquier miembro del conjunto es nulo. En este caso la consulta no regresará renglones.

2.8. Cláusula HAVING (CON)

De la misma forma que WHERE elimina renglones en un SELECT, se utiliza HAVING para condicionar resultados por grupo. Si se especifica HAVING deberá haberse especificado también GROUP BY.

Dada la siguiente representación:



Para encontrar el salario máximo de cada departamento y que muestre solo aquellos departamentos cuyo salario máximo sea mayor a \$2900, se necesita realizar lo siguiente:

- Encontrar el salario máximo por cada departamento agrupando por número de departamento.
- Restringir cada resultado de grupo, para que muestre solo aquellos que el salario máximo sea mayor a \$2900.

```
SELECT deptno, MAX(sal)
FROM emp
GROUP BY deptno
HAVING MAX(sal) > 2900;
```

Cuando se utiliza la cláusula HAVING para restringir grupos, se realiza lo siguiente:

- Se agrupan los renglones
- Se aplican las funciones de grupo a cada grupo
- Se muestran los grupos que cumplen la condición HAVING

```
SELECT columna, funcion_de_grupo
FROM tabla
[WHERE condicion]
[GROUP BY expresion_group_by]
[HAVING condicion_de_grupo]
[ORDER BY columna];
```

El ejemplo muestra los números de departamento y el salario máximo para aquellos departamentos con un salario máximo mayor a \$2900.

Se puede utilizar GROUP BY sin utilizar funciones de grupo.

```
SELECT deptno, MAX(sal) 'MAX SAL'
FROM emp
GROUP BY deptno
HAVING MAX(sal) > 2900
```

deptno	MAX SAL
10	5000
20	3000

El ejemplo siguiente muestra los números de departamento y el promedio salarial para aquellos departamentos cuyo salario máximo sea mayor a \$2900.

```
SELECT deptno, AVG(sal) 'AVG'
FROM emp
GROUP BY deptno
HAVING MAX(sal) > 2900;
```

DEPTNO	AVG
10	2916.6667
20	2175.0000

El siguiente ejemplo muestra el puesto y la suma total salarial por puesto, para aquellos puestos con una nómina total mayor a \$5000. En el ejemplo se elimina a los puestos SALESMAN, se ordena por suma total de cada puesto.

```
SELECT JOB, SUM(sal) PAYROLL
FROM emp
WHERE job not like 'SALES%'
GROUP BY job
HAVING SUM(sal) > 5000
ORDER BY SUM(sal);
```

JOB	PAYROLL
ANALYST	6000
MANAGER	8275

Cláusula HAVING con subconsultas

Se puede utilizar subconsultas no solo en la cláusula WHERE, sino también en la cláusula HAVING. SQL Server ejecuta la subconsulta y el resultado se devuelve a la consulta principal en la cláusula HAVING.

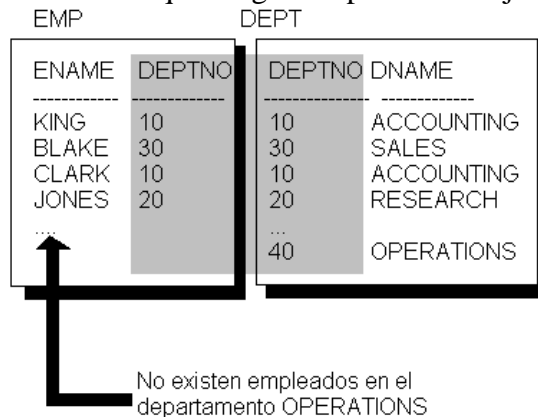
```
SELECT deptno, MIN(sal)
FROM emp
GROUP BY deptno
HAVING MIN(sal) > (SELECT MIN(sal)
FROM emp
WHERE deptno = 20)
```

El ejemplo muestra todos los departamentos que tengan un salario mínimo mayor que el del departamento 20.

2.9. Combinaciones externas: OUTER JOIN, UNION JOIN

Outer Joins (LEFT | RIGHT)

Si los renglones no satisfacen la condición JOIN, el renglón no aparecerá en el resultado de la consulta. Por ejemplo, en el equi-join de EMP y DEPT, el departamento 40 OPERATIONS no aparece debido a que ningún empleado trabaja en ese departamento.



La consulta sería la siguiente:

```
SELECT e.ename, e.deptno, d.dname
FROM emp e JOIN dept d
ON e.deptno = d.deptno;
```

El resultado:

ENAME	DEPTNO	DNAME
KING	10	ACCOUNTING
BLAKE	30	SALES
CLARK	10	ACCOUNTING
JONES	20	RESEARCH
...		
ALLEN	30	SALES
TURNER	30	SALES
JAMES	30	SALES
...		
14 rows selected		

Se puede utilizar un OUTER JOIN para ver también aquellos renglones que normalmente no cumplen la condición-join.

OUTER JOIN puede ser por la izquierda (LEFT) o por la derecha (RIGHT).

```
SELECT tabla1.columna, tabla2.columna
FROM tabla1 LEFT OUTER JOIN tabla2
ON tabla1.columna = tabla2.columna;
```

```
SELECT tabla1.columna, tabla2.columna
FROM tabla1 LEFT OUTER JOIN tabla2
ON tabla1.columna = tabla2.columna;
```

```
SELECT tabla1.columna, tabla2.columna
FROM tabla1 RIGHT OUTER JOIN tabla2
ON tabla1.columna = tabla2.columna;
```

Los renglones que no cumplen la condición pueden ser mostrados dependiendo de que lado se encuentran.

Se utiliza LEFT o RIGHT dependiendo de que lado se encuentran los renglones que desee aparezcan aun cuando no cumpla la condicion-join.

NOTA: este método no se aplica de la misma forma que los demás tipos JOIN para el método tradicional (de colocar la condición-join en la cláusula WHERE), por lo que cada manejador utiliza su propia sintaxis para implementar este tipo de joins, en ORACLE por ejemplo se utiliza el operador (+) de lado de la condición que evaluará el LEFT o RIGHT. Es preciso colocar la palabra LEFT o RIGHT de lado que se desee mostrar los renglones con la deficiencia de información.

Por ejemplo en la consulta siguiente:

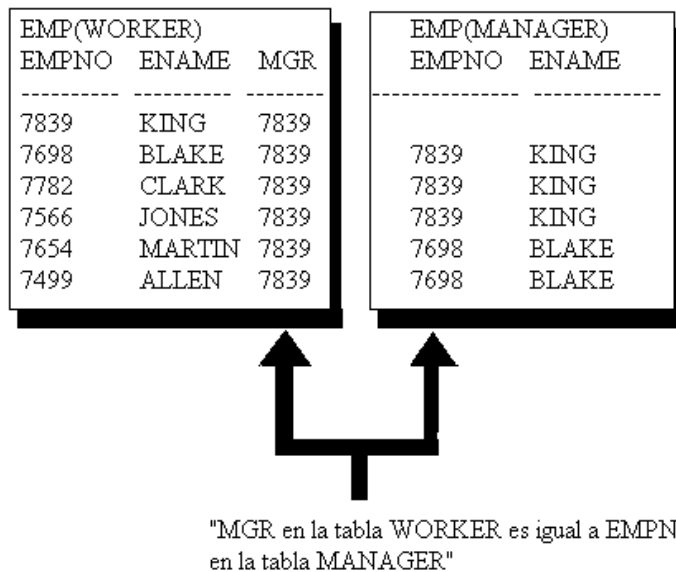
```
SELECT e.ename, d.deptno, d.dname
FROM emp e RIGHT OUTER JOIN dept d
      ON e.deptno = d.deptno
ORDER BY d.deptno;
```

Cuyo resultado es:

ename	deptno	dname
KING	10	ACCOUNTING
CLARK	10	ACCOUNTING
...	40	OPERATIONS
15 rows selected.		

SELF JOIN

En ocasiones se necesitará hacer un JOIN con la misma tabla. Por ejemplo, para encontrar el nombre del jefe de cada empleado, se necesita hacer un JOIN con EMP y con la misma tabla.



Por ejemplo, para encontrar el nombre del “jefe” de Blake, se necesita:

- Encontrar a Blake en la tabla EMP mediante la columna ENAME.
- Encontrar el número de jefe para Blake en la columna MGR. El número de jefe de Blake es 7839.
- Encontrar el nombre del jefe con EMPNO = 7839, mirando en la columna ENAME, King es el nombre que tiene el número 7839. de tal forma que, King es el jefe de Blake

La consulta resulta ser:

```
SELECT worker.ename + ' Works for' + manager.ename
FROM emp worker JOIN emp manager
```

ON worker.mgr = manager.empno;

El resultado es el siguiente:

BLAKE	Works for	KING
CLARK	Works for	KING
JONES	Works for	KING
MARTIN	Works for	BLAKE
...		
13 rows selected.		

Un SELF JOIN puede ser implementado con el método tradicional de la siguiente manera:

SELECT worker.ename + ' Works for ' + manager.ename

FROM emp worker, emp manager

WHERE worker.mgr = manager.empno;

2.10. El valor NULL

El valor nulo es la ausencia de valor en un campo o intersección de renglón columna, un valor nulo no es lo mismo que cero o espacios en blanco. El cero es un número y los espacios en blanco son caracteres.

Las columnas de cualquier tipo pueden contener valores nulos, a menos que dichas columnas hayan sido definidas como NOT NULL o llaves primarias (PRIMARY KEY) cuando se crea la columna.

SELECT ename, job, comm

FROM emp;

ename	job	comm

KING	PRESIDENT	NULL
ALLEN	SALESMAN	300.00
WARD	SALESMAN	500.00
JONES	MANAGER	NULL
MARTIN	SALESMAN	1400.00
...		
(14 row(s) affected)		

En la columna COMM de la tabla EMP, se puede notar que solo aquellos que son SALESMAN (vendedores) pueden tener una comisión. TURNER, que es un vendedor no gana comisión alguna, pero esta columna tiene cero y no un valor nulo.

Si el contenido de una columna es nulo dentro de una expresión aritmética, el resultado es NULL. Por ejemplo, si se intenta dividir entre cero se obtendrá un error. Sin embargo, si se divide un número entre un valor nulo, el resultado es un nulo.

*SELECT ename, 12*sal+comm*

FROM emp;

ename	

...	
KING	NULL

Todas las funciones excepto COUNT (*) ignoran los valores nulos en una columna.

```
SELECT AVG(comm)
FROM emp;
```

```
-----
550
```

En el ejemplo, el promedio es calculado en base a solo en los renglones que tienen valores válidos almacenados en la columna COMM.

El promedio es calculado como la suma de todas las comisiones dividido entre el número de empleados (4).

3. Vistas

Vistas. En SQL se refiere de manera específica a una tabla virtual derivada con nombre; el equivalente en SQL de una vista externa ANSI/SPARC es (por lo regular) un conjunto de varias tablas, algunas de ellas vistas en el sentido SQL y otras tablas base.

El “esquema externo” está formado por definiciones de esas vistas y tablas base.

3.1. Definición

En SQL una vista sigue siendo una tabla igual que las tablas base y se aplica el mismo lenguaje manipulativo o sea al DML de SQL, aunque una vista es una tabla virtual, es decir, una tabla que no existe en si, pero ante el usuario parece existir, en cambio una tabla base es una tabla real, en el dispositivo físico de almacenamiento.

Las vistas no se sustentan en sus propios datos almacenados, separados físicamente y distinguibles, en vez de ello se almacena su definición en el catálogo en términos de otras tablas, en el caso de Microsoft SQL Server en una tabla de catálogo llamada VIEWS.

Por ejemplo, se crea la vista *buenosEmpleados*:

```
create view buenosEmpleados as
select ename, sal, sal*0.30 salCom, comm
from emp
where comm > sal*0.30
```

La nueva vista se localiza en la carpeta Views de la Base de datos donde ha sido creada:

	ename	sal	salCom	comm
1	WARD	1250.00	375.0000	500.00
2	MARTIN	1250.00	375.0000	1400.00

Cuando se ejecuta CREATE VIEW no se ejecuta la subconsulta que sigue a la palabra AS (lo cual es de hecho la definición de la vista); solo se almacena en el catálogo. Para el usuario, es como si verdaderamente existiera en la base de datos una tabla llamada BUENOSEMPLEADOS con las filas y las columnas mostradas en la figura anterior.

De hecho, BUENOSEMPLEADOS es una “ventana” a través del cual se ve la tabla real EMP. Además esa “ventana” es dinámica. Las modificaciones hechas a EMP serán visibles automática e instantáneamente a través de esa “ventana” (siempre que esas modificaciones cumplan la condición de la vista), de manera similar las modificaciones hechas a BUENOSEMPLEADOS se aplicarán en forma automática e instantánea a la tabla real EMP y por supuesto se podrán ver a través de la “ventana”.

Ahora bien, dependiendo de lo avanzado de los conocimientos del usuario (y quizá también de la aplicación en cuestión), el usuario podrá o no darse cuenta de que BUENOSEMPLEADOS es en realidad una vista. En todo caso, tendrá poca importancia: lo fundamental es que los usuarios pueden trabajar con BUENOSEMPLEADOS como si fuera cualquier tabla real (con ciertas excepciones).

Por ejemplo se puede consultar la vista BUENOSEMPLEADOS como cualquier tabla:

```
SELECT *
FROM BUENOSEMPLEADOS
WHERE ename = 'WARD'
```

Resultado:

	ename	sal	salCom	comm
1	WARD	1250.00	375.0000	500.00

Esta proposición SELECT tiene todo el aspecto de una selección normal realizada sobre una tabla base normal. El sistema, maneja este tipo de operaciones convirtiéndolas en una operación equivalente realizada sobre la tabla o tablas subyacente (s).

La operación equivalente en la tabla subyacente es:

```
SELECT ename, sal, sal*0.30 salCom, comm
FROM emp
WHERE comm > sal * 0.30 and ename = 'WARD'
```

Resultado:

	ename	sal	salCom	comm
1	WARD	1250.00	375.0000	500.00

La conversión se hace combinando la proposición SELECT emitida por el usuario con la proposición SELECT guardada en el catálogo, el sistema sabe que “FROM BUENOSEMPLEADOS” significa en realidad “FROM EMP”; también sabe que cualquier selección de BUENOSEMPLEADOS debe ir calificada además por la condición “WHERE COMM > sal*0.30” y también sabe que “SELECT *” de BUENOSEMPLEADOS significa en realidad “SELECT ename, sal, sal*0.30 salCom, comm ” de EMP. Por tanto, es capaz de traducir la proposición SELECT original sobre la tabla virtual BUENOSPROVEEDORES a una proposición SELECT equivalente sobre la tabla real EMP; equivalente en el sentido de que el efecto de ejecutar esa proposición sobre la tabla real EMP es como si verdaderamente existiera una tabla base llamada BUENOSEMPLEADOS y la proposición SELECT original se ejecutara sobre ella.

Creación de vistas

```
CREATE VIEW vista [(COLUMNA [, COLUMNA]...)]
AS subconsulta
[WITH CHECK OPTION];
```

Véase como en la terminología ANSI/SPARC la proposición CREATE VIEW combina la función de esquema externo (la porción “vista” y la porción “columna” describen el objeto externo) con la función de correspondencia externa/conceptual (la porción “subconsulta” especifica la correspondencia de ese objeto con el nivel conceptual).

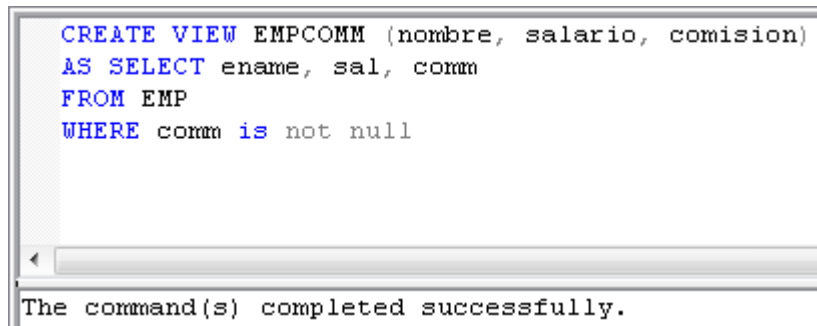
En principio, cualquier tabla derivable –es decir, cualquier tabla que puede obtenerse mediante una proposición SELECT- se puede en teoría definirse como una vista, pero en la práctica, esta aseveración no es cien por ciento verdadera en lo que respecta a DB2, ya que no permite incluir el operador UNION en una definición de vista; porque no permite la operación UNION en las subconsultas. El Microsoft SQL Server no se acepta el ORDER BY en las vistas, pero si las funciones agregadas, funciones anidadas, subconsultas, siempre que no se indiquen en el renglón final de la definición de la vista.

Hay algunos ejemplos de CREATE VIEW:

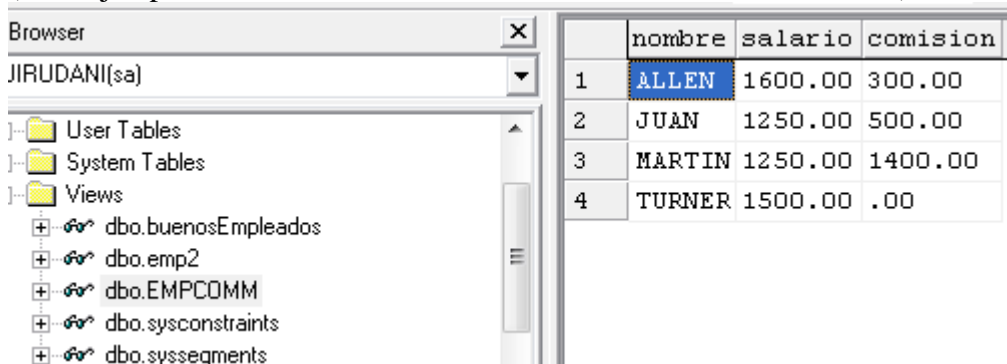
Vista de “subconjunto de filas”

```
CREATE VIEW EMPCOMM (nombre, salario, comision)
AS SELECT ename, sal, comm
FROM EMP
WHERE comm is not null
```

Resultado:



El efecto de esta proposición es crear una vista llamada EMPCOMM, con tres columnas llamadas NOMBRE, SALARIO, COMISIÓN, las cuales corresponden respectivamente a las tres columnas ENAME, SAL, COMM de la tabla subyacente EMP. Sino se especifican de manera explícita los nombres de las columnas en la proposición CREATE VIEW, entonces la vista heredará los nombres de columna del origen de la vista en la forma obvia (en el ejemplo, los nombres heredados serían ENAME, SAL, COMM).



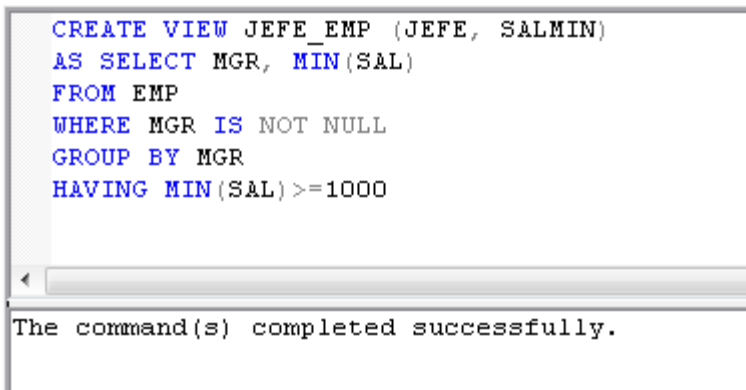
Los nombres de columna deberán especificarse de manera explícita (para todas las columnas de la vista) si:

- a) cualquier columna de la vista se deriva de una función, una expresión operacional o un literal (y, por tanto, carece de un nombre para heredar), o

- b) si al no hacerlo dos o más columnas de la vista recibirán el mismo nombre. En los dos siguientes ejemplos se ilustran ambos casos.

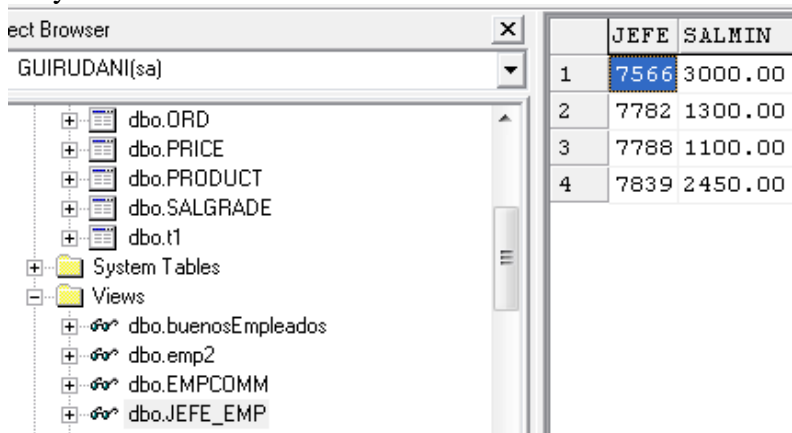
La columna de la vista se deriva de una función y consulta de resumen estadístico.

```
CREATE VIEW JEFE_EMP (JEFE, SALMIN)
AS SELECT MGR, MIN(SAL)
FROM EMP
WHERE MGR IS NOT NULL
GROUP BY MGR
HAVING MIN(SAL)>=1000
```



La vista contiene el número de jefe y el salario de su empleado que gane menos, excluyendo aquellos cuyo jefe sea desconocido y aquellos grupos cuyo salario mínimo sea menor a \$1000.

En este ejemplo, no existe un nombre que pueda heredar la segunda columna, pues ésta se deriva de una función; por tanto, es preciso especificar de manera explícita los nombres de las columnas. Adviértase que esta columna no es tan sólo un subconjunto simple de filas y columnas de la tabla base subyacente (a diferencia de las otras columnas) en vez de ello, podría considerarse como una especie de resumen estadístico o compresión de dicha tabla subyacente.



Si no se indicara los nombres de las columnas, presentaría el siguiente mensaje de error:

```
CREATE VIEW JEFE_EMP2
AS SELECT MGR, MIN(SAL)
FROM EMP
WHERE MGR IS NOT NULL
GROUP BY MGR
HAVING MIN(SAL) >= 1000
```

Server: Msg 4511, Level 16, State 1, Procedure JEFE_EMP2, Line 2
Create View or Function failed because no column name was specified for column 2.

Vista derivada de varias tablas

```
CREATE VIEW CUSTOMER_ORD_ITEM (name, ordId, itemId, itemTot, total)
AS SELECT c.name, o.ordid, i.itemid, i.itemtot, o.total
FROM customer c JOIN ord o ON c.custid = o.custid
JOIN item i ON o.ordid = i.ordid
WHERE c.name = 'TKB SPORT SHOP';
```

El significado de esta vista en particular es que aparecerá el nombre, las ordenes y los ítems, el total de cada orden para el cliente TKB SPORT SHOP. Nótese que la definición de esta vista implica una reunión de las tablas CUSTOMER, ORD e ITEM. De manera que este es un ejemplo de vista derivada de varias tablas subyacentes. Adviértase también que (una vez más) los nombres de columna deben especificarse de manera explícita, pues así se evitan errores en la creación de la vista por ausencia de nombres de columnas en la tabla base, o por duplicidad de nombres de columnas.

El contenido de la vista generada es:

	name	ordId	itemId	itemTot	total
1	TKB SPORT SHOP	610	1	35.00	101.40
2	TKB SPORT SHOP	610	2	8.40	101.40
3	TKB SPORT SHOP	610	3	58.00	101.40

Crear vistas en términos de otras vistas.

```
CREATE VIEW COI_MAYOR
AS SELECT NAME, ORDID, ITEMTOT
FROM CUSTOMER_ORD_ITEM
WHERE ITEMTOT >= 35.00
WITH CHECK OPTION
```

	NAME	ORDID	ITEMTOT
1	TKB SPORT SHOP	610	35.00
2	TKB SPORT SHOP	610	58.00

Como la definición de una vista puede ser cualquier subconsulta válida, y como una subconsulta puede extraer datos de vistas así como de tablas base, es del todo posible definir una vista en términos de otras vistas, como en el ejemplo, que toma CUSTOMER_ORD_ITEM para crear una vista COI_MAYOR que contiene los ITEMCOM >= 35.00.

3.2. La opción WITH CHECK OPTION

Esta opción significa: CON OPCIÓN DE VERIFICACIÓN e indica que las operaciones de modificación (UPDATE) e inserción (INSERT) realizadas con la vista deben verificarse para garantizar que toda fila modificada o insertada satisfaga la condición de definición de la vista (comm > sal*0.30) en el ejemplo.

Volvemos a la vista buenosEmpleados2

Se crea una vista sin “With check option”

```
create view buenosEmpleados2 as
select empno, ename, sal, sal*0.30 salCom, comm
from emp
where comm > sal*0.30
```

Se intenta actualizar un atributo de la vista

```
update buenosEmpleados2
set sal = 50000
where empno = 7521
```

```
update buenosEmpleados2
set sal = 50000
where empno = 7521
```

(1 row(s) affected)

Ahora se presenta los datos de la vista modificada:

```
update buenosEmpleados2
set sal = 50000
where empno = 7521
```

	empno	ename	sal	salCom	comm
1	7654	MARTIN	1250.00	375.0000	1400.00

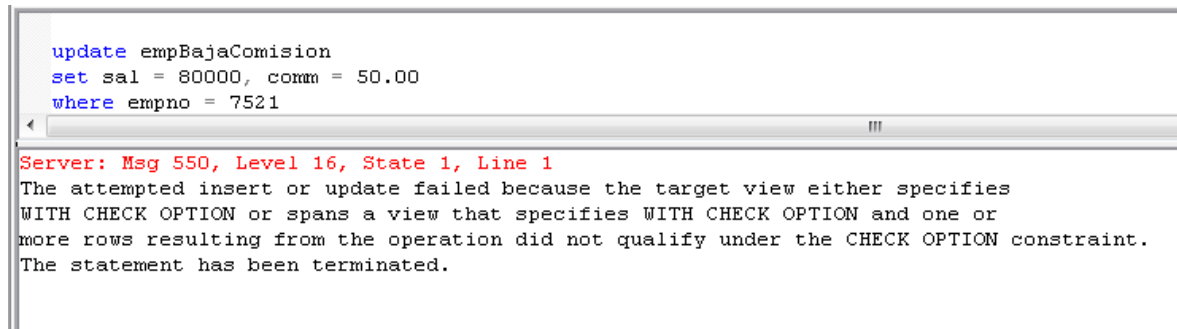
Se observa que el registro 7521 ya no cumple con la condición de la vista, por tanto ya no forma parte de ella.

Se crea una vista con With Check option

```
create view buenosEmpleados3 as
  select empno, ename, sal, sal*0.30 salCom, comm
  from emp
  where comm > sal*0.30
with check option
```

Se intenta actualizar un atributo de la vista

```
update empBajaComision
set sal = 80000, comm = 50.00
where empno = 7521
```



```
update empBajaComision
set sal = 80000, comm = 50.00
where empno = 7521
```

Server: Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint. The statement has been terminated.

Los datos de la vista se han mantenido:

	empno	ename	sal	salCom	comm
1	7521	WARD	5.00	1.5000	500.00
2	7654	MARTIN	1250.00	375.0000	1400.00

Las operaciones con UPDATE e INSERT de las vistas que incluyen “With Check option” se verificarán para comprobar que todas las filas modificadas o insertadas satisfagan la condición de definición de la vista. Sino se especifica la opción entonces las inserciones y modificaciones del tipo de las antes ilustradas se aceptarán, pero las filas recién insertadas o modificadas desaparecerán de inmediato de la vista.

3.3. Operaciones DML sobre las vistas

Solo algunas operaciones de DML se pueden aplicar sobre las vistas.

DROP VIEW

Para desechar una vista se usa:
DROP VIEW NomVista;

La vista especificada en NomVista se desecha (es decir, se elimina su definición del catálogo). He aquí un ejemplo:

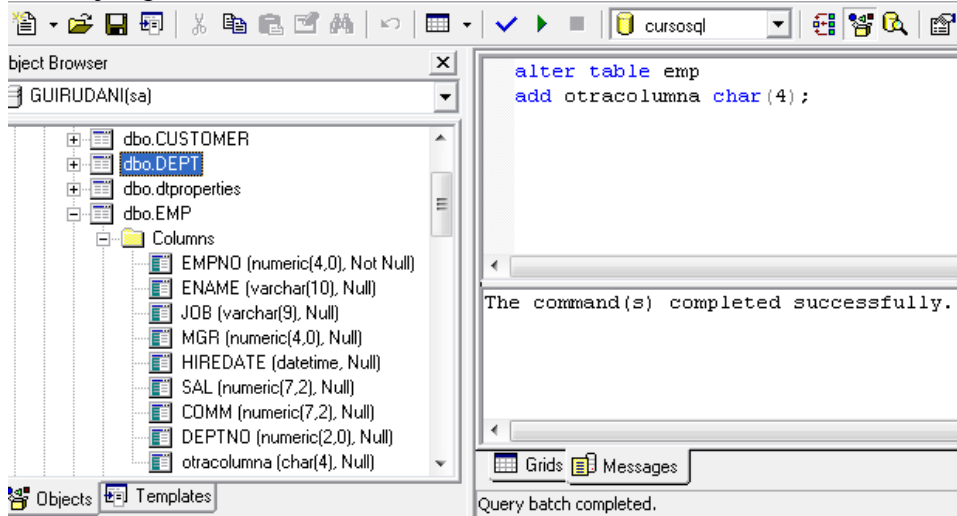
DROP VIEW buenosEmpleados;

Si se desecha una tabla (tabla base o vista), se deserrarán también en forma automática todas las vistas definidas en términos de esa tabla.

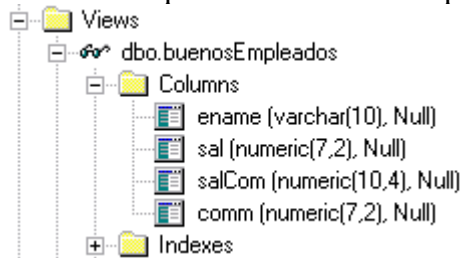
ALTER

No existe una proposición ALTER VIEW (alterar vista). La alteración de una tabla base (mediante ALTER TABLE) no afecta a las vistas ya existentes.

En el ejemplo se añade la columna “otracolumna” a la tabla EMP.



Se observa que la vista buenosEmpleados no presenta la columna nueva.



3.4. Funcionamiento de las vistas

El problema de implementar eficientemente consultas sobre una vista es complejo. Se han sugerido dos aproximaciones principales.

1. una estrategia llamada **modificación de consultas**, conlleva convertir la consulta sobre la vista en una consulta sobre las tablas de base subyacentes. La desventaja de esta aproximación es que no es eficiente para las vistas definidas via consultas complejas que exigen mucho tiempo de ejecución, especialmente si se aplican consultas múltiples a la vista dentro de un breve periodo de tiempo.
2. La otra estrategia, llamada **materialización de vistas**, conlleva crear físicamente una tabla de vista temporal cuando se consulta primeramente la vista y mantener esa tabla en el supuesto de que se realicen otras consultas sobre la vista. En este caso, se debe desarrollar una estrategia eficiente para actualizar automáticamente la tabla de la vista cuando se actualizan las tablas de base, para mantener la vista actualizada.

Para este propósito, se han desarrollado técnicas que utilizan el concepto de **actualización incremental**, donde se determina qué tuplas nuevas deben ser insertadas, borradas o modificadas en la tabla de la vista materializada cuando se aplica un cambio a una de las tablas de base de definición. Generalmente la vista se mantiene mientras sea consultada. Si la vista no se consulta en un cierto periodo de tiempo, el sistema puede eliminar automáticamente la tabla de vista física y recalcularle desde el principio cuando futuras consultas hagan referencia a la vista.

3.5. Utilización de las vistas

Las vistas sirven para lograr la llamada independencia lógica de los datos, término adoptado para distinguir ésta de la independencia física de los datos.

Se dice que un sistema proporciona **independencia física** de los datos porque los usuarios y sus programas no dependen de la estructura física de la BD almacenada.

Se dice que un sistema ofrece **independencia lógica** de los datos si los usuarios y sus programas son así mismo independientes de la estructura lógica de la BD; esto presenta dos aspectos, a saber, **crecimiento** y **reestructuración**.

1. **crecimiento**. Conforme crezca la BD para incorporar nuevos tipos de información, así también deberá crecer la definición de la BD.
 - a. La expansión de una tabla base ya existente para incluir un campo nuevo. Por ejemplo. La inclusión de un nuevo campo DESCUENTO en la tabla base CUSTOMER.
 - b. La inclusión de una nueva tabla base (adición de un nuevo tipo de objeto, por ejemplo la adición de información sobre SALES a la BD EMP y CUSTOMER);Ninguno de estos dos tipos de modificaciones deberá afectar en absoluto a los usuarios ya existentes.

2. **reestructuración**. Reestructurar la BD de manera tal que, aunque el contenido total de información siga siendo el mismo, se modifique la colocación de la información dentro de esa BD, es decir, se altere de alguna manera la asignación de los campos de las tablas, no son deseables pero inevitables. Por ejemplo, podría ser necesario dividir una tabla en “sentido vertical” a fin de poder almacenar campos muy solicitados es un dispositivo más rápido y otros menos solicitados en un dispositivo más lento.

Suponiendo, para efectos del ejemplo, que resulta necesario sustituir a tabla EMP por estas dos tablas base:

EMP_GRAL1(empno, ename, job, mgr)

EMP_GRAL2(empno, hiredate, sal, comm, deptno)

El aspecto crucial que ha de observarse en este ejemplo es que la antigua tabla EMP es la reunión de las dos nuevas tablas EMP_GRAL1 y EMP_GRAL2 donde la reunión se refiere a la reunión natural en base en la clave de empleado.

Por ejemplo en la tabla EMP teníamos:

(7369, SMITH, CLERK, 7902, 1980-12-17, 800.00, null, 20)

En EMP_GRAL1 tenemos ahora el registro:

(7369, SMITH, CLERK, 7902)

Y en EMP_GRAL2 tenemos ahora el registro:

(7369, 1980-12-17, 800.00, null, 20)

Si los reunimos obtendremos el registro:

(7369, SMITH, CLERK, 7902, 1980-12-17, 800.00, null, 20) como antes: Por lo tanto, creamos una vista que sea exactamente esa reunión y la llamamos EMP:

```
CREATE VIEW EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno)
AS SELECT E1.empno, E1.ename, E1.job, E1.mgr, E2.hiredate, E2.sal, E2.comm,
E2.deptno
FROM EMP_GRAL1 E1, EMP_GRAL2 E2
WHERE E1.empno = E2.empno
```

Cualquier programa que hiciera referencia antes a la tabla base EMP hará referencia ahora a la vista S. las operaciones SELECT seguirán funcionando tal y como lo hacían antes (aunque requerirán un análisis adicional durante el proceso de ligado y quizá eleven un tanto el tiempo de ejecución). En teoría las operaciones de actualización se podrían aplicar a la vista EMP.

Ventajas de las vistas.

1. ofrecen un cierto grado de independencia lógica de los datos en casos de reestructuración de la BD.
2. permiten a diferentes usuarios ver los mismos datos de distintas maneras (al mismo tiempo)
3. permite a los usuarios concentrarse de manera exclusiva en los datos de interés personal y hacer caso omiso al resto.
4. se cuenta con seguridad automática para datos ocultos, es decir, información no visible a través de una vista dada. Es evidente que tales datos están a salvo del acceso a través de esa vista específica. Así, obligar a los usuarios a utilizar la BD a través de vistas en un mecanismo sencillo pero efectivo para el control de autorizaciones.
5. se simplifica la percepción del usuario. El mecanismo de vistas permite a los usuarios concentrarse en los datos de interés particular y hacer caso omiso del resto.

3.6. Actualización de vistas

Las operaciones de actualización se manejan de manera similar. Por ejemplo, la operación:

```
UPDATE BUENOSEMPLEADOS
SET ename = 'juan'
WHERE ename = 'WARD';
```

Resultado:

```
UPDATE BUENOSEMPLEADOS
SET ENAME = 'JUAN'
WHERE ENAME = 'WARD';
```

(1 row(s) affected)

El contenido de BUENOSEMPLEADOS presenta:

	ename	sal	salCom	comm
1	JUAN	1250.00	375.0000	500.00
2	MARTIN	1250.00	375.0000	1400.00

El contenido de la tabla base EMP se muestra como:

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7369	SMITH	CLERK	7902	1980-12-17 00:00:00.000	800.00		20
2	7499	ALLEN	SALESMAN	7698	1981-02-20 00:00:00.000	1600.00	300.00	30
3	7521	JUAN	SALESMAN	7698	1981-02-22 00:00:00.000	1250.00	500.00	30
4	7566	JONES	MANAGER	7839	1981-04-02 00:00:00.000	2975.00		20
5	7654	MARTIN	SALESMAN	7698	1981-09-28 00:00:00.000	1250.00	1400.00	30
6	7698	BLAKE	MANAGER	7839	1981-05-01 00:00:00.000	2850.00		30
7	7782	CLARK	MANAGER	7839	1981-06-09 00:00:00.000	2450.00		10
8	7788	SCOTT	ANALYST	7566	1982-12-09 00:00:00.000	3000.00		20
9	7839	KING	PRESIDENT		1981-11-17 00:00:00.000	5000.00		10
10	7844	TURNER	SALESMAN	7698	1981-09-08 00:00:00.000	1500.00	.00	30
11	7876	ADAMS	CLERK	7788	1983-01-12 00:00:00.000	1100.00		20
12	7900	JAMES	CLERK	7698	1981-12-03 00:00:00.000	950.00		30

COMO SE CONVIERTEN LAS OPERACIONES REALIZADAS SOBRE VISTAS EN OPERACIONES EQUIVALENTES SOBRE LA TABLA BASE

La actualización en la vista BUENOS EMPLEADOS:

```
UPDATE BUENOSEMPLEADOS
SET ename = 'juan'
WHERE ename = 'WARD';
```

El ligador lo convirtió en la tabla EMP por:

```
UPDATE EMP
SET ename = 'juan'
WHERE comm > sal * 0.30 and ename = 'WARD';
```

CONVERSIÓN DE LAS OPERACIONES DE RECUPERACION DE DATOS (SELECT) DE LAS VISTAS A LAS TABLAS SUBYACENTES

Se tiene la siguiente consulta en BUENOSEMPLEADOS:

```
SELECT *
FROM BUENOSEMPLEADOS
WHERE ename = 'WARD'
```

La operación equivalente a la tabla EMP es:

```
SELECT ename, sal, sal*0.30 salCom, comm
FROM emp
WHERE comm > sal * 0.30 and ename = 'WARD'
```

La mayoría de las consultas funcionan a la perfección pero la situación es diferente en el caso de las operaciones de actualización. NO TODAS LAS VISTAS SE PUEDEN ACTUALIZAR.

No todas las vistas se pueden actualizar, actualizar quiere decir hacer operaciones de inserción, eliminación, además de las operaciones de modificación. Es decir se pueden realizar en sus datos las operaciones de MODIFICAR, INSERTAR y ELIMINAR.

Por ejemplo:

```
create view buenosEmpleados as
select ename, sal, sal*0.30 salCom, comm
from emp
where comm > sal*0.30
```

y

```
create view buenosEmpleados2 as
select empno, ename, sal, sal*0.30 salCom, comm
from emp
where comm > sal*0.30
```

De estas dos vistas, buenosEmpleados2 se puede actualizar (en teoría), en tanto que buenosEmpleados no se puede actualizar (también en teoría) las razones son:

- se puede insertar un registro nuevo en la vista, digamos el registro (9991, 'Pedro', 'Salesman', 7839, '1-may-81', 1299, 600, 30, null) en la tabla EMP.

INSERT INTO EMP

```
VALUES (9991, 'Pedro', 'Salesman', 7839, '1-may-81', 1299, 600, 30, NULL);
```

The screenshot shows a SQL Server Enterprise Manager window with a query result grid. The grid has columns for empno, ename, sal, salCom, and comm. The data is as follows:

	empno	ename	sal	salCom	comm
1	7521	JUAN	1250.00	375.0000	500.00
2	7654	MARTIN	1250.00	375.0000	1400.00
3	9991	Pedro	1299.00	389.7000	600.00

The left pane shows the database structure with views 'buenosEmpleados' and 'buenosEmpleados2' selected.

B) Se puede eliminar un registro ya existente de la vista, digamos el registro (9991) eliminando en realidad el registro correspondiente (9991, 'Pedro', 'Salesman', 7839, '1-may-81', 1299, 600, 30, NULL).

Se debe tener especial cuidado de considerar las restricciones de claves foráneas al eliminar algún renglón de lo contrario la operación será anulada.

```
delete from buenosEmpleados2
where empno = 7521;
```

Level 16, State 1, Line 1
 conflicted with COLUMN REFERENCE constraint 'CUSTOMER_REPID_FK'. The conflict occurred because an attempt was made to delete a row that is referenced by a foreign key. The operation has been terminated.

b) se puede modificar un campo ya existente en la vista, digamos cambiar el nombre de Juan por el de WARD haciendo en realidad la misma modificación sobre el campo correspondiente de la tabla subyacente EMP.

```
UPDATE buenosEmpleados2
SET ename = 'JOHNY'
WHERE empno = 7521;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
2	7499 ALLEN	SALESMAN	7698	1981-02-20 00:00:00.000	1600.00	300.00	30
3	7521 JOHNY	SALESMAN	7698	1981-02-22 00:00:00.000	1250.00	500.00	30
4	7566 JONES	MANAGER	7839	1981-04-02 00:00:00.000	2975.00		20
5	7654 MARTIN	SALESMAN	7698	1981-09-28 00:00:00.000	1250.00	1400.00	30
6	7698 BLAKE	MANAGER	7839	1981-05-01 00:00:00.000	2850.00		30
7	7702 CLARK	MANAGER	7839	1981-06-09 00:00:00.000	2450.00		10
8	7788 SCOTT	ANALYST	7566	1982-12-09 00:00:00.000	3000.00		20
9	7839 KING	PRESIDENT		1981-11-17 00:00:00.000	5000.00		10
10	7844 TURNER	SALESMAN	7698	1981-09-08 00:00:00.000	1500.00	.00	30

La diferencia importante entre las vistas buenosEmpleados y buenosEmpleados2 es que en la segunda vista se incluye la clave primaria que hace posible identificar el registro correspondiente que se va a afectar en la tabla subyacente.

Se pretende insertar un registro nuevo en la vista que no tiene el campo llave de la tabla base subyacente

Recuérdese la vista buenosEmpleados:

```
create view buenosEmpleados as
select ename, sal, sal*0.30 salCom, comm
from emp
where comm > sal*0.30
```

Insertar un registro nuevo en la vista, digamos el registro ('Ann', 1394, 600), que equivale a: (NULL, 'Ann', NULL, NULL, NULL, 1394, 600, NULL, NULL) en la tabla EMP.

```
INSERT INTO buenosEmpleados
VALUES ('Ann', 1394, 418, 600);
```

```

INSERT INTO buenosEmpleados
VALUES ('Ann', 1394, 418, 600);

```

Server: Msg 4406, Level 16, State 1, Line 1
Update or insert of view or function 'buenosEmpleados' failed because it contains a derived or constant field.

Se pretende eliminar un registro de una vista que no tiene en sus columnas un campo llave o clave primaria de la tabla subyacente

Si se trata de eliminar algún registro ya existente en la vista buenosEmpleados, por ejemplo: el registro ('JHON', 1250.00, 375.00, 500.00), el sistema intenta eliminar algún registro correspondiente de la tabla base subyacente; pero ¿Cuál? El sistema no tiene forma de saberlo, porque no se ha especificado el número de empleado (y no puede especificarse, porque el campo empno no es parte de la vista).

```

delete
from buenosEmpleados
where ename = 'JHON';

```

(0 row(s) affected)

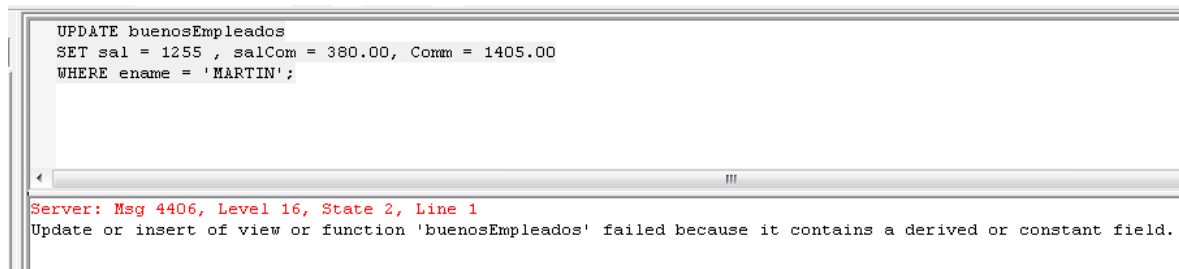
Aunque en la vista haya un campo ename con valor JOHN.

	ename	sal	salCom	comm
1	JOHN	1250.00	375.0000	500.00
2	MARTIN	1250.00	375.0000	1400.00

Se pretende modificar algún registro de una vista que no contiene en sus columnas la clave primaria de la tabla subyacente

Refiriéndose a la tabla buenosEmpleados, digamos que se quiere cambiar el registro ('MARTIN', 1250.00, 375.00, 1400.00) por ('MARTIN', 1255.00, 380.00, 1405.00), el sistema tendrá que tratar de modificar algún registro correspondiente en la tabla base subyacente; pero, de nuevo, ¿Cuál?

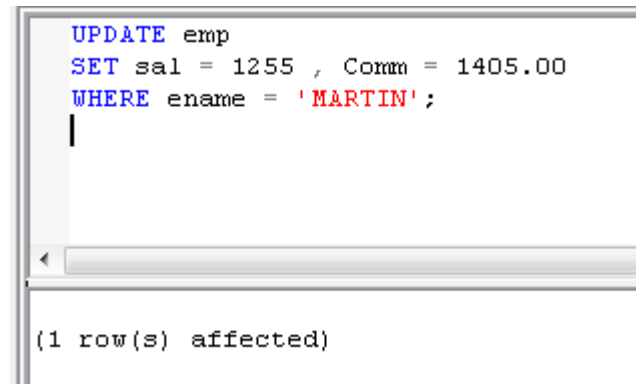
```
UPDATE buenosEmpleados
SET sal = 1255 , salCom = 380.00, Comm = 1405.00
WHERE ename = 'MARTIN';
```



```
UPDATE buenosEmpleados
SET sal = 1255 , salCom = 380.00, Comm = 1405.00
WHERE ename = 'MARTIN';
```

Server: Msg 4406, Level 16, State 2, Line 1
Update or insert of view or function 'buenosEmpleados' failed because it contains a derived or constant field.

Pero si funciona con EMP, quitando la columna salCom puesto que esta no está presente en la tabla base subyacente EMP:



```
UPDATE emp
SET sal = 1255 , Comm = 1405.00
WHERE ename = 'MARTIN';
```

(1 row(s) affected)

a) Las dos vistas anteriores: buenosEmpleados y BuenosEmpleados2 se pueden caracterizar como **vistas de subconjuntos de columnas** porque cada una está formada por un subconjunto de las columnas de una sola tabla subyacente. Una vista de **subconjunto de columnas** teóricamente se puede actualizar si conserva la clave primaria de la tabla subyacente.

b) Una **vista de subconjunto de filas**:

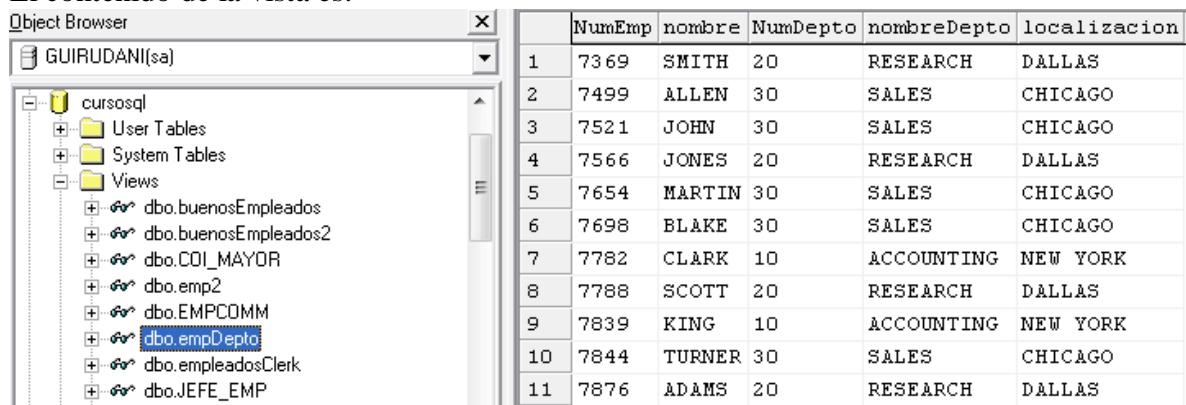
```
CREATE VIEW empleadosClerk (NumEmp, Nombre, Puesto, Salario)
AS SELECT empno, ename, job, sal
FROM EMP
WHERE job = 'CLERK'
```

Esta vista sí incluye la clave primaria de la tabla subyacente (tabla EMP) y se puede poner al día. (Por supuesto, una vista de **subconjunto de columnas** DEBE incluir la clave primaria de la tabla subyacente).

- c) Una **vista de equirreunión de varias tablas** por ejemplo: mostrar el nombre de los departamentos a los que pertenecen cada empleado.

```
CREATE VIEW empDepto (NumEmp, nombre, NumDepto, nombreDepto, localizacion)
AS SELECT emp.empno, emp.ename, emp.deptno, dept.dname, dept.loc
FROM EMP, DEPT
WHERE emp.deptno = dept.deptno
```

El contenido de la vista es:



	NumEmp	nombre	NumDepto	nombreDepto	localizacion
1	7369	SMITH	20	RESEARCH	DALLAS
2	7499	ALLEN	30	SALES	CHICAGO
3	7521	JOHN	30	SALES	CHICAGO
4	7566	JONES	20	RESEARCH	DALLAS
5	7654	MARTIN	30	SALES	CHICAGO
6	7698	BLAKE	30	SALES	CHICAGO
7	7782	CLARK	10	ACCOUNTING	NEW YORK
8	7788	SCOTT	20	RESEARCH	DALLAS
9	7839	KING	10	ACCOUNTING	NEW YORK
10	7844	TURNER	30	SALES	CHICAGO
11	7876	ADAMS	20	RESEARCH	DALLAS

Esta vista es la equirreunión de empleados y departamentos. En términos de la posibilidad de ponerla al día, la vista empDepto acepta modificaciones. Supongamos, por ejemplo, que intentamos cambiar la fila:

(7839, 'KING', 10, 'ACCOUNTING', 'NEW YORK')

Por:

(7839, 'KINNG', 10, 'ACCOUNTING', 'NEW YORK')

```
UPDATE empDepto
SET nombre = 'KINNG'
WHERE NumEmp = 7839;
```



```

FROM EMP, DEPT
WHERE emp.deptno = dept.deptno

UPDATE empDepto
SET nombre = 'KINNG'
WHERE NumEmp = 7839;
    
```

(1 row(s) affected)

Ahora si queremos cambiar:
 (7839, 'KINNG', 10, 'ACCOUNTING', 'NEW YORK')
 Por:
 (7839, 'KINNG', 10, 'ACCOUNTING', 'DALLAS')

```

UPDATE empDepto
SET nombreDepto = 'DALLAS'
WHERE NumEmp = 7839;
    
```

```

WHERE emp.deptno = dept.deptno

UPDATE empDepto
SET nombreDepto = 'DALLAS'
WHERE NumEmp = 7839;
    
```

(1 row(s) affected)

Nuestra vista muestra el siguiente resultado:

ser	NumEmp	nombre	NumDepto	nombreDepto	localizacion	
DAN(sa)	1	7369	SMITH	20	RESEARCH	DALLAS
	2	7499	ALLEN	30	SALES	CHICAGO
	3	7521	WARD	30	SALES	CHICAGO
	4	7566	JONES	20	RESEARCH	DALLAS
	5	7654	MARTIN	30	SALES	CHICAGO
	6	7698	BLAKE	30	SALES	CHICAGO
	7	7782	CLARK	10	DALLAS	NEW YORK
	8	7788	SCOTT	20	RESEARCH	DALLAS
	9	7839	KINNG	10	DALLAS	NEW YORK
	10	7844	TURNER	30	SALES	CHICAGO
	11	7876	ADAMS	20	RESEARCH	DALLAS

Por tanto la tabla DEPT ha sido modificada, y en la tabla EMP todos los empleados que están en el mismo departamento que KINNG (es decir, 10) se cambian al nuevo nombre de departamento DALLAS.

	DEPTNO	DNAME	LOC
1	10	DALLAS	NEW YORK
2	20	RESEARCH	DALLAS
3	30	SALES	CHICAGO
4	40	OPERATIONS	BOSTON
*			

d) Una **Vista de resumen estadístico**

```
CREATE VIEW item2 (orden, precio)
AS SELECT ordid, SUM(actualprice)
FROM ITEM
GROUP BY ORDID
```

La vista contiene:

	orden	precio
1	601	2.40
2	602	2.80
3	603	56.00
4	604	144.00
5	605	142.20
6	606	3.40
7	607	5.60
8	608	29.60
9	609	87.50
10	610	95.80
11	611	45.00

La vista contiene en sus columnas funciones de agregados, por lo tanto no es actualizable:

```
UPDATE item2
SET precio = 2.99
WHERE orden = 601;
```

```

UPDATE item2
SET precio = 2.99
WHERE orden = 601;

```

Server: Msg 4403, Level 16, State 1, Line 1
View or function 'item2' is not updatable because it contains aggregates.

En cambio, la actualización en la tabla base subyacente, es aceptada:

```

UPDATE item
SET actualprice = 2.99
WHERE ordid = 601;

```

(1 row(s) affected)

Por los ejemplos anteriores nos damos cuenta de que, por su naturaleza, algunas vistas se pueden poner al día, pero otras no (también por su misma naturaleza). Adviértase aquí la frase “por su naturaleza”. No es sólo cuestión de que algunos sistemas puedan manejar ciertas actualizaciones y otros no. Ningún sistema puede manejar en todos los casos y sin ayuda actualizaciones de vistas tales como item2 (con “sin ayuda” queremos decir “sin ayuda de algún usuario humano”).

Como consecuencia de esto, podemos clasificar las vistas en la forma indicada por el siguiente diagrama de Venn.



Como puede verse en el diagrama, sí existen ciertas vistas teóricamente susceptibles de actualización, pero que no se pueden poner al día en algunos sistemas. El problema es que aunque sabemos de la existencia de tales vistas, ignoramos cuáles son con exactitud; es todavía (en parte) un problema de investigación definir qué es precisamente lo que caracteriza a estas vistas. Por esta razón, la mayor parte de los productos actuales (en el mejor de los casos), permiten poner al día vistas que son subconjuntos de filas o bien subconjuntos de columnas (o una combinación de ellos) extraídos de una sola tabla base subyacente.

4. Disparadores

4.1. Definición

Los desencadenadores representan aplicaciones desarrolladas en lenguaje T-SQL y que se ejecutan, o mejor dicho, se "disparan" cuando sucede algún tipo de evento en una tabla. Los desencadenadores se llaman también disparadores o triggers.

Al crear un desencadenador, la información acerca del mismo se inserta en las tablas del sistema **sysobjects** y **syscomments**. Si se crea un desencadenador con el mismo nombre que uno existente, el nuevo reemplazará al original.

SQL Server no permite agregar desencadenadores definidos por el usuario a las tablas del sistema.

¿Por que entonces llamar disparadores a los disparadores? con ellos se permitió dar mas control al programador sobre los eventos que desencadenan un elemento activo, se le conoce en ingles como ECA rules o event-condition-action rule. Es por ello que los disparadores tienen una cláusula BEFORE, AFTER o INSTEAD (por cierto postgresql no tiene INSTEAD, sql Server no considera el Before) y bajo que evento (INSERT, UPDATE, DELETE) pero de esta forma el desencadenador se ejecutará para cada fila sometida al evento (cláusula FOR EACH ROW) pero el standard (que postgresql no cubre completamente) dice que puede ser también FOR EACH SENTENCE. Esto provoca que se ejecute el desencadenador una vez para toda la relación (o tabla) para la cual se define (cláusula ON).

La diferencia para los que lo han programado, por ejemplo en postgresql, queda clara entonces: cuando es FOR EACH ROW en la función postgresql que implementa el trigger se tiene un objeto NEW y uno OLD que se dispara o ejecuta una vez para cada fila afectada de la tabla o cada vez que se realizan operaciones en una tupla o fila completa de una tabla, en el trigger de STATEMENT tiene un objeto NEW y OLD que son la relación (o tabla) completa.

Esta claro entonces que es un poco más difícil implementar un trigger para statement que para fila (todavía postgresql no lo tiene).

Los desencadenadores son una herramienta útil para los creadores de bases de datos que deseen que se realicen determinadas acciones cuando se inserten, actualicen o eliminen datos en una tabla específica. Son un método especialmente útil para exigir reglas de la empresa y asegurar la integridad de los datos.

Un activador es un código de SQL de procedimientos que automáticamente es invocado por el RDBMS cuando ocurre un evento de manipulación de datos. Es útil recordar que:

- a) **Activa antes/después de un update:** un activador siempre se invoca antes o después de que una fila de datos se selecciona, inserta o actualiza.
- b) **Activa a una tabla:** Un activador siempre está asociado con una tabla de BD.
- c) **BD 0, 1, + activadores:** Cada BD puede o no tener uno más activadores.
- d) **Activa es parte de la trans:** Un activador se ejecuta como parte de la transacción que lo activó.

Para crear un desencadenador se requiere el siguiente formato:

```
CREATE TRIGGER nomTrg ON [nomTabla] [nomVista]
[FOR/INSTEAD OF/AFTER] [DELETE/INSERT/UPDATE]
AS
```

...

En suma, los activadores desempeñan un rol crítico al hacer que una BD sea verdaderamente útil.

4.2. Estado de los desencadenadores

Los desencadenadores pueden tener dos estados: **Deshabilitado o habilitado.**

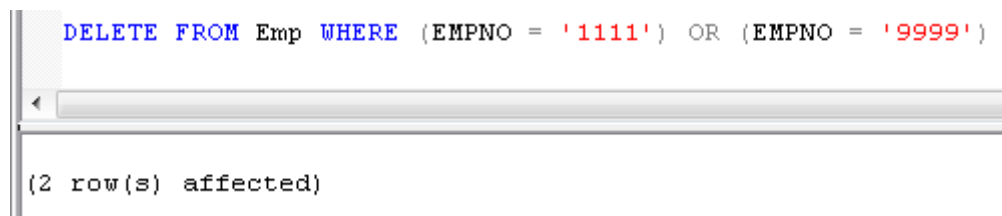
Si lo desea, puede deshabilitar o habilitar un desencadenador específico de una tabla o todos los desencadenadores que haya en ella. Cuando se deshabilita un desencadenador, su **definición** se mantiene, pero la ejecución de una instrucción INSERT, UPDATE o DELETE en la tabla no activa la ejecución de las acciones del desencadenador hasta que éste se vuelva a habilitar.

Los desencadenadores se pueden habilitar o deshabilitar en la instrucción ALTER TABLE.
ALTER TABLE *tabla*.

```
{ENABLE | DISABLE} TRIGGER
{ALL | nombreDesencadenador[,...n]}
```

Se deshabilita el desencadenador Emp_borrado de la tabla EMP:

```
ALTER TABLE EMP DISABLE TRIGGER Emp_borrado
```



```
DELETE FROM Emp WHERE (EMPNO = '1111') OR (EMPNO = '9999')
```

(2 row(s) affected)

4.3. Tipos de desencadenadores

Hay dos tipos de desencadenadores en Microsoft SQL Server 2000. Estos son: Los desencadenadores **INSTEAD OF** y los **AFTER**. Estos difieren en sus propósitos y en el momento en que son activados.

4.3.1. AFTER

Se ejecuta automáticamente después de que se haya completado la estructura que lo ha activado.

Por ejemplo:

```
CREATE TRIGGER trgInsert ON tabla
FOR INSERT
AS Print ('desencadenador AFTER [trgInsert] – ha desencadenado un trigger')
```

Por defecto se usa AFTER con solo indicar FOR, los AFTER no se pueden definir en las vistas.

Especifica que el desencadenador sólo se activa cuando todas las operaciones especificadas en la instrucción SQL desencadenadora se han ejecutado correctamente. Además, todas las acciones referenciales en cascada y las comprobaciones de restricciones deben ser correctas para que este desencadenador se ejecute.

AFTER es el valor predeterminado, si sólo se especifica la palabra clave FOR.

Los desencadenadores AFTER no se pueden definir en las vistas.

4.3.2. INSTEAD OF

Indica que se ejecuta el desencadenador en vez de la instrucción SQL desencadenadora, por lo que se suplantán las acciones de las instrucciones desencadenadoras. La mayor parte de los desencadenadores son reactivos; las restricciones y el desencadenador **INSTEAD OF** son proactivos.

Como máximo, se puede definir un desencadenador INSTEAD OF por cada instrucción INSERT, UPDATE o DELETE en cada tabla o vista. No obstante, en las vistas es posible definir otras vistas que tengan su propio desencadenador INSTEAD OF.

Los desencadenadores INSTEAD OF no se permiten en las vistas actualizables WITH CHECK OPTION. SQL Server emitirá un error si se agrega un desencadenador INSTEAD OF a una vista actualizable donde se ha especificado WITH CHECK OPTION. El usuario debe quitar esta opción mediante ALTER VIEW antes de definir el desencadenador INSTEAD OF.

```
{ [DELETE] [,] [INSERT] [,] [UPDATE] }
```

Son palabras clave que especifican qué instrucciones de modificación de datos activan el desencadenador cuando se intentan en esta tabla o vista. Se debe especificar al menos una opción. En la definición del desencadenador se permite cualquier combinación de éstas, en cualquier orden. Si especifica más de una opción, sepárelas con comas.

Para los desencadenadores INSTEAD OF, no se permite la opción DELETE en tablas que tengan una relación de integridad referencial que especifica una acción ON DELETE en cascada. Igualmente, no se permite la opción UPDATE en tablas que tengan una relación de integridad referencial que especifica una acción ON UPDATE en cascada.

Para ver como se crea y utiliza un activador, se examinará un problema de manejo de inventario simple. Por ejemplo, si la cantidad de un producto en existencia se actualiza cuando se vende el producto, el sistema tiene que verificar (automáticamente) si la cantidad en existencia se redujo por debajo de la cantidad mínima permisible.

Para demostrar este proceso primero se modifica la tabla PRODUCTO original. Para agregar una cantidad de productos en existencia P_ONHAND, una cantidad de pedido

mínima (P_MIN_ORDER) y una columna de reordenar P_REORDER. La columna P_REORDER será un campo booleano (Si/No) para indicar si el producto tiene que volverse a pedir o no. Los valores P_REORDER iniciales podrán en "No" para que sirvan como base para el desarrollo del activador inicial.

```
alter table PRODUCT
add P_ONHAND INT,
    P_MIN_ORDER INT,
    P_REORDER char(1),
CONSTRAINT CHECK_REORDER CHECK (P_REORDER = 'S' OR P_REORDER = 'N');
```

Se actualiza a N:

```
update PRODUCT
set P_REORDER = 'N';
```

El contenido de la tabla PRODUCTO es:

	PROID	DESCRIP	P_MIN_ORDER	P_REORDER	P_ONHAND
1	100860	ACE TENNIS RACKET I	25	N	500
2	100861	ACE TENNIS RACKET II	23	N	350
3	100870	ACE TENNIS BALLS-3 PACK	43	N	234
4	100871	ACE TENNIS BALLS-6 PACK	25	N	874
5	100890	ACE TENNIS NET	15	N	435
6	101860	SP TENNIS RACKET	12	N	543
7	101863	SP JUNIOR RACKET	18	N	453
8	102130	RH: "GUIDE TO TENNIS"	20	N	45
9	200376	SB ENERGY BAR-6 PACK	12	N	44
10	200380	SB VITA SNACK-6 PACK	25	N	55

Con la lista mostrada en la figura anterior se creará un activador para evaluar la cantidad de producto en existencia, P_ONHAND, si **la cantidad en existencia está por debajo de la cantidad mínima mostrada en P_MIN_ORDER la columna P_REORDER se pondrá en 'S'**.

Se creará un activador que sea ejecutado implícitamente AFTER de una UPDATE de la columna P_ONHAND. La acción del activador comparará la columna P_ONHAND con la columna P_MIN. Si el valor P_ONHAND es igual o menor que P_MIN, el activador debe actualizar la columna P_ORDER a 'S'.

```
CREATE TRIGGER trgNoHayProd ON PRODUCT
FOR UPDATE
AS
UPDATE PRODUCT
SET P_REORDER = 'S'
WHERE P_ONHAND <= P_MIN_ORDER;
```


Para probar el activador, se actualizará la cantidad en existencia del producto ACE TENNIS RACKET I que tiene el Id 100860 a 20.

```
UPDATE PRODUCT
SET P_ONHAND = 20
WHERE PRODID = 100860;
```

Después de la actualización, el activador es accionado automáticamente y coloca en la columna P_REORDER en 'S' para la actualización porque está por debajo del mínimo.

	PRODID	DESCRIP	P_MIN_ORDER	P_REORDER	P_ONHAND
1	100860	ACE TENNIS RACKET I	25	S	20
2	100861	ACE TENNIS RACKET II	23	N	350
3	100870	ACE TENNIS BALLS-3 PACK	43	N	234
4	100871	ACE TENNIS BALLS-6 PACK	25	N	874
5	100890	ACE TENNIS NET	15	N	435
6	101860	SP TENNIS RACKET	12	N	543
7	101863	SP JUNIOR RACKET	18	N	453
8	102130	RH: "GUIDE TO TENNIS"	20	N	45
9	200376	SB ENERGY BAR-6 PACK	12	N	44
10	200380	SB VITA SNACK-6 PACK	25	N	55

¿Qué pasa si se reduce la cantidad mínima del producto ACE TENNIS BALLS-3 PACK con Id 100870?

```
UPDATE PRODUCT
SET P_MIN_ORDER = 300
WHERE PRODID = 100870;
```

La figura siguiente muestra que cuando se actualiza la columna P_MIN_ORDER también se activa el desencadenador.

En los desencadenadores de Microsoft SQL Server no se indican directamente la actualización de una columna en particular sino en la actualización de la tabla PRODUCT y por tanto si cualquier columna de la tabla presenta la condición que obedece el desencadenador este se activará.

Los manejadores de BD ORACLE y DB2 requieren de la indicación de una columna particular de la tabla: AFTER UPDATE OF P_ONHAND ON PRODUCT que se activará solo si se actualiza la columna P_ONHAND y entonces surgen inconsistencias que solicitan un cambio en el activador de tal manera que se incluyan también el campo P_MIN_ORDER de la manera siguiente:

```
AFTER UPDATE OF P_ONHAND, P_MIN_ORDER ON PRODUCT.
```

¿Qué pasa si se incrementa el valor de P_ONHAND del producto 100870?

```
UPDATE PRODUCT
SET P_ONHAND = 600
WHERE PRODID = 100870;
```

	PRODID	DESCRIP	P_MIN_ORDER	P_REORDER	P_ONHAND
1	100860	ACE TENNIS RACKET I	25	S	20
2	100861	ACE TENNIS RACKET II	23	N	350
3	100870	ACE TENNIS BALLS-3 PACK 300		S	600
4	100871	ACE TENNIS BALLS-6 PACK	25	N	874
5	100890	ACE TENNIS NET	15	N	435
6	101860	SP TENNIS RACKET	12	N	543
7	101863	SP JUNIOR RACKET	18	N	453
8	102130	RH: "GUIDE TO TENNIS"	20	N	45
9	200376	SB ENERGY BAR-6 PACK	12	N	44
10	200380	SB VITA SNACK-6 PACK	25	N	55

¿Porque el activador no cambió la señal de volver a pedir en 'N'? La respuesta es que el activador no está considerando todos los casos posibles.

A continuación se examinarán los problemas con el activador.

- El activador se activa y realiza una sentencia de actualización de todas las filas en la tabla PRODUCT, aunque la sentencia de activación actualiza solo una. Esto puede afectar el desempeño de la BD.
- El activador sólo coloca el valor de P_REORDER en 'S'; nunca repone el valor en 'N' cuando claramente se requiere tal acción.
- El desencadenador se crea solamente en la BD actual; sin embargo, un desencadenador puede hacer referencia a objetos que están fuera de la BD actual.
- Solo se pueden aplicar a una tabla
- La misma acción del desencadenador puede definirse para más de una acción del usuario (por ejemplo, INSERT y UPDATE) en la misma instrucción CREATE TRIGGER
- En un desencadenador se puede especificar cualquier instrucción SET. La opción SET elegida permanece en efecto durante la ejecución del desencadenador y después, vuelve a su configuración anterior.

A continuación se verá cómo puede modificarse el activador para que maneje todos los escenarios de actualización

Solución 1:

```
CREATE TRIGGER trgNoHayProd ON PRODUCT
FOR UPDATE AS
Begin
UPDATE PRODUCT
SET P_REORDER = 'S'
WHERE P_ONHAND <= P_MIN_ORDER;

UPDATE PRODUCT
SET P_REORDER = 'N'
WHERE P_ONHAND > P_MIN_ORDER;
End
```

Solución 2:

Este disparador ejecuta una función:

```
create function compara (@ponhand int, @pminorder int ) returns char(1)
as
Begin
declare @res char(1);
if (@ponhand <= @pminorder)
    set @res = 'S'
else
    set @res = 'N'
return @res
End
```

```
CREATE TRIGGER trgNoHayProd ON PRODUCT
FOR UPDATE AS
Begin
    UPDATE PRODUCT
    SET P_REORDER = dbo.compara(P_ONHAND,P_MIN_ORDER);
End
```

Solución 3:

Usando Store procedures:

```
CREATE PROCEDURE verifyStock @PRODID numeric(6,0)
AS
DECLARE @P_ONHAND INT
DECLARE @P_MIN_ORDER INT
SELECT @P_ONHAND = P_ONHAND,
       @P_MIN_ORDER = P_MIN_ORDER
FROM PRODUCT
WHERE PRODID = @PRODID;

IF(@P_ONHAND <= @P_MIN_ORDER)
    UPDATE PRODUCT
    SET P_REORDER = 'S'
    WHERE PRODID = @PRODID;
ELSE
    UPDATE PRODUCT
    SET P_REORDER = 'N'
    WHERE PRODID = @PRODID;
```

Se realiza una actualización en P_ONHAND

```
UPDATE PRODUCT
SET P_ONHAND = 800
WHERE PRODID = 100870;
```

Se prueba el store procedure...

El procedimiento almacenado se prueba así: exec verifyStock 100870;

Los desencadenadores **INSTEAD OF** cancelan la acción desencadenante original y realizan su propia función en su lugar.

Un desencadenador **INSTEAD OF** se puede especificar en tablas y vistas.

Este desencadenador se ejecuta en lugar de la acción desencadenante original.

Los desencadenadores **INSTEAD OF** aumentan la variedad de tipos de actualizaciones que se pueden realizar en una vista. Cada tabla o vista está limitada a un desencadenador **INSTEAD OF** por cada acción desencadenante (**INSERT**, **UPDATE** o **DELETE**).

Ejemplo

En este ejemplo considerando la tabla **DEPT2** y **DEPT3**. Mediante un desencadenador **INSTEAD OF** colocado en **DEPT2** se redirigen las actualizaciones a la tabla **DEPT3**. Se produce actualización en la tabla **DEPT3** *en lugar de* la tabla **DEPT2**.

Se crea el desencadenador:

```
CREATE TRIGGER depto_actual
ON DEPT2
INSTEAD OF UPDATE AS
BEGIN
    UPDATE DEPT3
    SET DEPT3.DEPTNO = Inserted.DEPTNO,
        DEPT3.DNAME = Inserted.DNAME
    FROM DEPT3 JOIN Inserted
    ON DEPT3.DEPTNO = Inserted.DEPTNO
END
```

Probar el desencadenador mediante la actualización en **DEPT2**:

```
UPDATE DEPT2 SET DNAME = 'CONTA',
              LOC = 'MEXICO'
WHERE DEPTNO = '50'
```

Comprobar el estado de la tabla **DEPT2** y **DEPT3**:

```

SELECT DNAME, LOC FROM DEPT2
WHERE DEPTNO = '50'

SELECT DNAME, LOC FROM DEPT3
WHERE DEPTNO = '50'

```

	DNAME	LOC
1	contabilidad	DF

	DNAME	LOC
1	CONTA	DF

Los desencadenadores utilizan la caché de procedimientos para almacenar el plan de ejecución.

4.3.3. For each row, statement

Existen los desencadenadores para tuplos, (FOR EACH ROW) y los desencadenadores para sentencias (FOR STATEMENT).

Las restricciones se comprueban antes de la ejecución de la instrucción INSERT, UPDATE o DELETE. Si se infringen las restricciones, el desencadenador no se ejecuta.

Las tablas pueden tener varios desencadenadores para cualquier acción.

Microsoft SQL Server permite anidar varios desencadenadores en una misma tabla. Una tabla puede tener definidos múltiples desencadenadores. Cada uno de ellos puede definirse para una sola acción o para varias.

4.3.4. IF UPDATE

Cuando se especifica una acción FOR UPDATE, la cláusula IF UPDATE (*nombreColumna*) permite centrar la acción en una columna específica que se actualice.

IF UPDATE (*nombreColumna*)

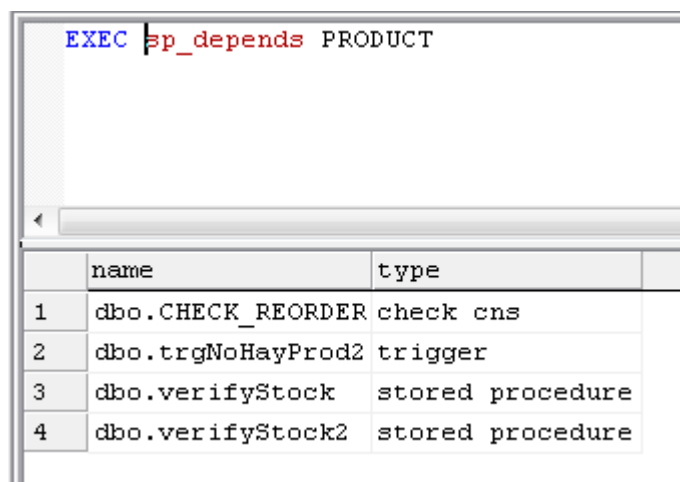
Imposibilidad de incluir determinadas instrucciones

SQL Server no permite utilizar las instrucciones siguientes en la **definición** de un desencadenador:

ALTER DATABASE

CREATE DATABASE
DISK INIT
DISK RESIZE
DROP DATABASE
LOAD DATABASE
LOAD LOG
RECONFIGURE
RESTORE DATABASE
RESTORE LOG

Para conocer qué tablas tienen desencadenadores, ejecute el procedimiento almacenado del sistema **sp_depends** <nombreTabla>.



```
EXEC sp_depends PRODUCT
```

	name	type
1	dbo.CHECK_REORDER	check cns
2	dbo.trgNoHayProd2	trigger
3	dbo.verifyStock	stored procedure
4	dbo.verifyStock2	stored procedure

Para ver la **definición** de un desencadenador, ejecute el procedimiento almacenado del sistema **sp_helptext** <nombreDesencadenador>.

```
EXEC sp_helptext trgNoHayProd2
```

	Text
1	CREATE TRIGGER trgNoHayProd2 ON PRODUCT
2	FOR UPDATE AS
3	Begin
4	UPDATE PRODUCT
5	SET P_REORDER = 'S'
6	WHERE P_ONHAND <= P_MIN_ORDER;
7	
8	UPDATE PRODUCT
9	SET P_REORDER = 'N'
10	WHERE P_ONHAND > P_MIN_ORDER;
11	End
12	

Para determinar los desencadenadores que hay en una tabla específica y sus acciones respectivas, ejecute el procedimiento almacenado del sistema **sp_helptrigger** <nombreTabla>.

```
EXEC sp_helptrigger product
```

	trigger_name	trigger_owner	isupdate	isdelete	isinsert	isafter	isinsteadof
1	trgNoHayProd2	dbo	1	0	0	1	0

Modificar un desencadenador

Alteración de un desencadenador Si debe cambiar la **definición** de un desencadenador existente, puede alterarlo sin necesidad de quitarlo.

Al cambiar la **definición** se reemplaza la **definición** existente del desencadenador por la nueva. También es posible alterar la acción del desencadenador.

Por ejemplo, si crea un desencadenador para INSERT y, posteriormente, cambia la acción por UPDATE, el desencadenador modificado se ejecutará siempre que se actualice la tabla.

```
ALTER TRIGGER Empl_Delete ON Employees
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 6
BEGIN
RAISERROR('You cannot delete more than six employees at a time.', 16, 1)
ROLLBACK TRANSACTION
END
```

Eliminación de un desencadenador -

Si desea eliminar un desencadenador, puede quitarlo. Los desencadenadores se eliminan automáticamente cuando se elimina la tabla a la que están asociados.

De forma predeterminada, el permiso para eliminar un desencadenador corresponde al propietario de la tabla y no se puede transferir. Sin embargo, los miembros de las funciones de administradores del sistema (sysadmin) y propietario de la base de datos (db_owner) pueden eliminar cualquier objeto si especifican el propietario en la instrucción **DROP TRIGGER**.

Sintaxis

DROP TRIGGER *nombreDesencadenador*

4.4. Grupos de desencadenadores

En función del tipo de evento, tenemos los siguientes grupos de desencadenadores:

- **Desencadenadores de inserción.** Estos desencadenadores se ejecutan cuando se añade un registro o varios.
- **Desencadenadores de actualización.** Se ejecutan cuando se ha actualizado uno o varios registros.
- **Desencadenadores de eliminación.** Se ejecutan cuando se ha eliminado uno o varios registros.

Con estos desencadenadores se asegura la lógica de negocio y se define la integridad del usuario. Antiguamente (versiones anteriores a SQL Server 2000), la integridad referencial en cascada tenía que implementarse mediante desencadenadores que permitiesen la actualización y eliminación en cascada.

4.4.1. Desencadenador de inserción

Un desencadenador INSERT se invoca cuando se intenta insertar una fila en una tabla que el desencadenador protege. Todas las inserciones se registran en una tabla especial llamada **inserted**.

Cuando se activa un desencadenador INSERT, las nuevas filas se agregan a la tabla del desencadenador y a la tabla **inserted**. Se trata de una tabla lógica que mantiene una copia de las filas insertadas. La tabla **inserted** contiene la actividad de inserción registrada proveniente de la instrucción INSERT. La tabla **inserted** permite hacer referencia a los datos registrados por la instrucción INSERT que ha iniciado el desencadenador. El

desencadenador puede examinar la tabla **inserted** para determinar qué acciones debe realizar o cómo ejecutarlas.

Las filas de la tabla **inserted** son siempre duplicados de una o varias filas de la tabla del desencadenador.

Se registra toda la actividad de modificación de datos (instrucciones INSERT, UPDATE y DELETE), pero la información del registro de transacciones es ilegible. Sin embargo, la tabla **inserted** permite hacer referencia a los cambios registrados provocados por la instrucción INSERT. Así, es posible comparar los cambios a los datos insertados para comprobarlos o realizar acciones adicionales. También se puede hacer referencia a los datos insertados sin necesidad de almacenarlos en variables.

Puede definir un desencadenador de modo que se ejecute siempre que una instrucción INSERT inserte datos en una tabla.

Ejemplo

El desencadenador de este ejemplo se creó para actualizar una columna (**UnitsInStock**) de la tabla **Products** siempre que se pida un producto (siempre que se inserte un registro en la tabla **Order Details**). El nuevo valor se establece al valor anterior menos la cantidad pedida.

```
CREATE TRIGGER OrdDet_Insert
ON [Order Details]
FOR INSERT
AS
UPDATE P SET
UnitsInStock = (P.UnitsInStock - I.Quantity)
FROM Products AS P INNER JOIN Inserted AS I
ON P.ProductID = I.ProductID
```

La tabla product contiene:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	5	0	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	86	0	0
13	Konbu	6	8	2 kg box	6.0000	24	0	5

La tabla Order Details contiene:

	OrderID	ProductID	UnitPrice	Quantity	Discount
1	10248	11	14.0000	12	0.
2	10248	42	9.8000	10	0.
3	10248	72	34.8000	5	0.
4	10249	14	18.6000	9	0.
5	10249	51	42.4000	40	0.
6	10250	41	7.7000	10	0.
7	10250	51	42.4000	35	0.15
8	10250	65	16.8000	15	0.15
9	10251	22	16.8000	6	5.e-002
10	10251	57	15.6000	15	5.e-002
11	10251	65	16.8000	20	0.
12	10252	20	64.8000	40	5.e-002
13	10252	33	2.0000	25	5.e-002
14	10252	60	27.2000	40	0.
15	10253	31	10.0000	20	0.

Se registra un artículo más a la orden 10248, el artículo es *Queso Manchego La Pastora* con la clave 12, esto provoca una inserción de renglón en la tabla Order Details de la siguiente manera:

insert into [Order Details] values (10248, 12, 14.0000, 10, 0.);

La inserción ha provocado el disparador, de manera que se ha disminuido el UnitsInStock de 22 a 12.

El atributo *UnitsInStocks* de la tabla *Product* ha disminuido en 10:

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
2	2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25
3	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25
4	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0
5	5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0
6	6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25
7	7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10
8	8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0
9	9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	39	0	0
10	10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0
11	11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30
12	12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	12	0	0
13	13	Konbu	6	8	2 kg box	6.0000	24	0	5

4.4.2. Desencadenador de actualización

Un desencadenador definido para una instrucción UPDATE se invoca siempre que se intenta actualizar datos de la tabla en la que está definido.

La instrucción UPDATE mueve la fila original a la tabla **deleted** e inserta la fila actualizada en la tabla **inserted**.

Se puede considerar que una instrucción UPDATE está formada por dos pasos:

- el paso DELETE que captura la *imagen anterior* de los datos y
- el paso INSERT que captura la *imagen posterior*.

El desencadenador puede examinar las tablas **deleted** e **inserted** así como la tabla actualizada, para determinar si se han actualizado múltiples filas y cómo debe ejecutar las acciones oportunas.

Para definir un desencadenador que supervise las actualizaciones de los datos de una columna específica puede utilizar la instrucción IF UPDATE. De este modo, el desencadenador puede aislar fácilmente la actividad de una columna específica. Cuando detecte una actualización en esa columna, realizará las acciones apropiadas, como mostrar un mensaje de error que indique que la columna no se puede actualizar o procesar un conjunto de instrucciones en función del nuevo valor de la columna.

Sintaxis

IF UPDATE (<nombreColumna>)

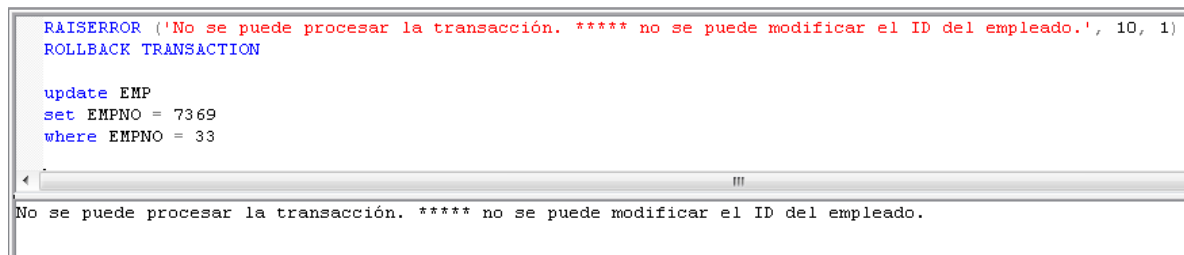
Ejemplo.

En este ejemplo se evita que un usuario modifique la columna **EmployeeID** de la tabla **Employees**.

```
CREATE TRIGGER Emp_Update ON Emp
FOR UPDATE
AS
IF UPDATE (Empno)
RAISERROR ('No se puede procesar la transacción. ***** no se puede modificar el ID del
empleado.', 10, 1)
ROLLBACK TRANSACTION
```

Se intenta actualizar:

```
update EMP
set EMPNO = 7369
where EMPNO = 33
```



```
RAISERROR ('No se puede procesar la transacción. ***** no se puede modificar el ID del empleado.', 10, 1)
ROLLBACK TRANSACTION

update EMP
set EMPNO = 7369
where EMPNO = 33
```

No se puede procesar la transacción. ***** no se puede modificar el ID del empleado.

4.4.3. Desencadenador de eliminación

Un desencadenador DELETE se invoca siempre que se intenta eliminar información de la tabla en la que está definido.

Cuando se activa un desencadenador DELETE, las filas eliminadas en la tabla afectada se agregan a una tabla especial llamada **deleted**. Se trata de una tabla lógica que mantiene una copia de las filas eliminadas. La tabla **deleted** permite hacer referencia a los datos registrados por la instrucción DELETE que ha iniciado la ejecución del desencadenador.

Al utilizar el desencadenador DELETE, tenga en cuenta los hechos siguientes: Cuando se agrega una fila a la tabla **deleted**, la fila deja de existir en la tabla de la base de datos, por lo que la tabla **deleted** y las tablas de la base de datos no tienen ninguna fila en común.

Ejemplo

El desencadenador de este ejemplo se creó para actualizar una columna **Discontinued** de la tabla **Products** cuando se elimine una categoría (cuando se elimine un registro de la tabla **Categories**). Todos los productos afectados se marcan con 1, lo que indica que ya no se suministran.

```
CREATE TRIGGER Category_Delete
ON Categories
FOR DELETE
AS
UPDATE P SET Discontinued = 1
FROM Products AS P INNER JOIN deleted AS d
ON P.CategoryID = d.CategoryID
```

Tabla producto antes del desencadenador:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	76	0	0
13	Konbu	6	8	2 kg box	6.0000	24	0	5

La tabla *Categories* antes de la ejecución del desencadenador:

CategoryID	CategoryName	Description	Picture
1	Beverages	(...)	(...)
2	Condiments	(...)	(...)
3	Confections	(...)	(...)
4	Dairy Products	(...)	(...)
5	Grains/Cereals	(...)	(...)
6	Meat/Poultry	(...)	(...)
7	Produce	(...)	(...)
8	Seafood	(...)	(...)
*			

Se ejecuta el desencadenador:

```
delete from Categories where CategoryID = 8
```

Se observa la tabla afectada: products

ID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discount
3	Anisado Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25	0
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25	0
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10	0
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0	0
10	Ikura	4	8	12 - 300 ml jars	31.0000	31	0	0	1
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30	0
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	76	0	0	0
13	Konbu	6	8	2 kg box	6.0000	34	0	5	1
14	Tofu	6	7	40 - 100 g pkgs.	23.2500	35	0	0	0

4.5. Desencadenadores anidados

Los desencadenadores pueden anidarse hasta 32 niveles. Si los desencadenadores anidados están habilitados, un desencadenador que modifique una tabla podrá activar un segundo desencadenador, que a su vez podrá activar un tercero, y así sucesivamente.

Cuando el anidamiento está habilitado, un desencadenador que cambie una tabla podrá activar un segundo desencadenador, que a su vez podrá activar un tercero y así sucesivamente.

El anidamiento se habilita durante la instalación, pero se puede deshabilitar y volver a habilitar con el procedimiento almacenado del sistema **sp_configure**. Los desencadenadores pueden anidarse hasta 32 niveles. Si un desencadenador de una cadena anidada provoca un bucle infinito, se superará el nivel de anidamiento. Por lo tanto, el desencadenador terminará y deshará la transacción.

Los desencadenadores anidados pueden utilizarse para realizar funciones como almacenar una copia de seguridad de las filas afectadas por un desencadenador anterior. Al utilizar desencadenadores anidados, tenga en cuenta los siguientes hechos: De forma predeterminada, la opción de configuración de desencadenadores anidados está activada.

Un desencadenador anidado no se activará dos veces en la misma transacción; un desencadenador no se llama a sí mismo en respuesta a una segunda actualización de la misma tabla en el desencadenador. Por ejemplo, si un desencadenador modifica una tabla que, a su vez, modifica la tabla original del desencadenador, éste no se vuelve a activar.

Los desencadenadores son transacciones, por lo que un error en cualquier nivel de un conjunto de desencadenadores anidados cancela toda la transacción y las modificaciones a los datos se deshacen. Por tanto, se recomienda incluir instrucciones PRINT al probar los desencadenadores para determinar dónde se producen errores.

La función @@NESTLEVEL permite ver el nivel actual de anidamiento.

Si el anidamiento está deshabilitado, un desencadenador que modifique otra tabla no invocará ninguno de los desencadenadores de esa tabla.

sp_configure 'nested triggers', 0

Las siguientes son algunas razones por las que podría decidir deshabilitar el anidamiento: Los desencadenadores anidados requieren un diseño complejo y bien planeado. Los cambios en cascada pueden modificar datos que no se deseaba cambiar.

4.6. **Desencadenadores recursivos**

Recursividad directa, que se da cuando un desencadenador se ejecuta y realiza una acción que lo activa de nuevo.

Recursividad indirecta, que se da cuando un desencadenador se activa y realiza una acción que activa un desencadenador de otra tabla.

Cuando la opción de desencadenadores recursivos está habilitada, un desencadenador que cambie datos en una tabla puede activar un segundo desencadenador que, a su vez puede, activar el primer desencadenador al modificar datos de la tabla original.

Cualquier desencadenador puede contener una instrucción UPDATE, INSERT o DELETE que afecte a la misma tabla o a otra distinta. Cuando la opción de desencadenadores recursivos está habilitada, un desencadenador que cambie datos de una tabla puede activarse de nuevo a sí mismo, en ejecución recursiva.

Esta opción se deshabilita de forma predeterminada al crear una base de datos, pero puede habilitarla con la instrucción ALTER DATABASE.

Activación recursiva de un desencadenador

Para habilitar los desencadenadores recursivos, utilice la instrucción siguiente:

Sintaxis

```
ALTER DATABASE Northwind  
SET RECURSIVE_TRIGGERS ON --opción de desencadenadores
```

sp_dboption *nombreBaseDatos*, 'recursive triggers', True ---desencadenadores recursivos

Si la opción de desencadenadores anidados está desactivada, la de desencadenadores recursivos también lo estará, sin importar la configuración de desencadenadores recursivos de la base de datos.

Utilice el procedimiento almacenado del sistema **sp_settriggerorder** para especificar un desencadenador que se active como primer desencadenador AFTER o como último desencadenador AFTER.

Cuando se han definido varios desencadenadores para un mismo suceso, su ejecución no sigue un orden determinado. Cada desencadenador debe ser autocontenido.

Recursividad directa, que se da cuando un desencadenador se ejecuta y realiza una acción que lo activa de nuevo.

Por ejemplo, una aplicación actualiza la tabla **T1**, lo que hace que se ejecute **Desen1**. **Desen1** actualiza de nuevo la tabla **T1**, con lo que **Desen1** se activa una vez más.

Recursividad indirecta, que se da cuando un desencadenador se activa y realiza un acción que activa un desencadenador de otra tabla, que a su vez causa una actualización de la tabla original. De este modo, el desencadenador original se activa de nuevo.

Por ejemplo, una aplicación actualiza la tabla **T2**, lo que hace que se ejecute **Desen2**. **Desen2** actualiza la tabla **T3**, con lo que **Desen3** se activa una vez más. A su vez, **Desen3** actualiza la tabla **T2**, de modo que **Desen2** se activa de nuevo.

Conveniencia del uso de los desencadenadores recursivos

Los desencadenadores recursivos son una característica compleja que se puede utilizar para resolver relaciones complejas, como las de autorreferencia (conocidas también como *cierres transitivos*). En estas situaciones especiales, puede ser conveniente habilitar los desencadenadores recursivos.

4.7. Implementación de desencadenadores

A continuación se presentan algunas implementaciones sobre desencadenadores, se recomienda al usuario realizar el desencadenador independientemente de la idea que se expone en este apartado, con la finalidad de obtener diferentes maneras de resolverlos.

Los productos con pedidos pendientes no se pueden eliminar

```
Create trigger NoBorrarProdConPedidos on product
for delete
as
IF (Select Count (*)
FROM item INNER JOIN deleted
ON item.ProdID = deleted.ProdID) > 0
print('no se pueden eliminar productos con pedidos pendientes')
ROLLBACK TRANSACTION
```

```
delete from product where prodId = 200376;
```

Los clientes no se pueden excederse de su límite de crédito

```
create trigger ClienNoExcLimCred on ORD
for insert
as
declare @total int
declare @limCred int
set @total = (select sum(o.total)
from inserted i inner join ord o
on i.custid = o.custid)
set @limCred = (select c.creditlimit
from customer c inner join inserted i
on c.custid = i.custid)
if (@total > @limCred)
print('el cliente ha excedido del limite de su credito')
```

rollback transaction

insert into ord (ordid, custid,total) values (622, 106, 100.00)

Disparador que impide que se borren departamentos que ya están asignados a empleados

```
create trigger NoBorrarDeptosConEmp on Dept
for delete
as if (select count(*)
      from emp e inner join deleted d
      on e.deptno = d.deptno)>0
print('no se puede borrar un departamento que tiene asignado empleados')
rollback transaction
```

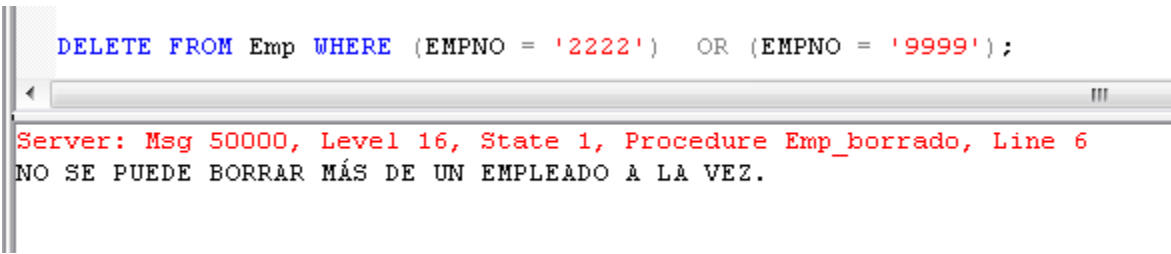
delete from dept where deptno = 30

Desencadenador que impide que los usuarios puedan eliminar varios empleados a la vez.

Aplicado a la tabla EMP, el desencadenador se activa cada vez que se elimina un registro o grupo de registros de la tabla. El desencadenador comprueba el número de registros que se están eliminando mediante la consulta de la tabla **Deleted**. Si se está eliminando más de un registro, el desencadenador devuelve un mensaje de error personalizado y deshace la transacción.

```
CREATE TRIGGER Emp_borrado ON EMP
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 1
BEGIN
    RAISERROR('NO SE PUEDE BORRAR MÁS DE UN EMPLEADO A LA VEZ.', 16, 1)
    ROLLBACK TRANSACTION
END
```

La instrucción DELETE siguiente activa el desencadenador y evita la transacción.



```
DELETE FROM Emp WHERE (EMPNO = '2222') OR (EMPNO = '9999');
Server: Msg 50000, Level 16, State 1, Procedure Emp_borrado, Line 6
NO SE PUEDE BORRAR MÁS DE UN EMPLEADO A LA VEZ.
```

La instrucción DELETE siguiente activa el desencadenador y permite la transacción.


```
DELETE FROM Emp WHERE (EMPNO = '2222');  
  
(1 row(s) affected)
```

4.8. Usos, ventajas y desventajas

Usos

Los desencadenadores son una herramienta útil para los creadores de bases de datos que deseen que se realicen determinadas acciones cuando se inserten, actualicen o eliminen datos en una tabla específica. Son un método especialmente útil para exigir reglas de empresa y asegurar la integridad de los datos.

Los desencadenadores se suelen crear para exigir integridad referencial o coherencia entre datos relacionados de forma lógica en diferentes tablas. Como los usuarios no pueden evitar los desencadenadores, éstos se pueden utilizar para exigir reglas de empresa complejas que mantengan la integridad de los datos.

Sólo el propietario de la tabla, los miembros de la función fija de servidor **sysadmin** y los miembros de las funciones fijas de base de datos **db_owner** y **db_ddladmin** pueden crear y eliminar desencadenadores de esa tabla. Estos permisos no pueden transferirse.

Los propietarios de tablas no pueden crear desencadenadores **AFTER** en vistas o en tablas temporales. Sin embargo, los desencadenadores pueden hacer referencia a vistas y tablas temporales.

Los propietarios de las tablas pueden crear desencadenadores **INSTEAD OF** en vistas y tablas, con lo que se amplía enormemente el tipo de actualizaciones que puede admitir una vista.

Regularmente los usuarios o programadores no esperan ver ningún conjunto de resultados cuando ejecutan una instrucción **UPDATE**, **INSERT** o **DELETE**. Los desencadenadores no devuelven conjuntos de resultados ni pasan parámetros

Los desencadenadores pueden tratar acciones que impliquen a múltiples filas. Una instrucción **INSERT**, **UPDATE** o **DELETE** que invoque a un desencadenador puede afectar a varias filas. En tal caso, puede elegir entre:

- Procesar todas las filas juntas, con lo que todas las filas afectadas deberán cumplir los criterios del desencadenador para que se produzca la acción.
- Permitir acciones condicionales. Por ejemplo, si desea eliminar tres clientes de la tabla **Customers**, puede definir un desencadenador que asegure que no queden pedidos activos ni facturas pendientes para cada cliente eliminado. Si uno de los tres clientes tiene una factura pendiente, no se eliminará, pero los demás que cumplan la condición sí.

Para determinar si hay varias filas afectadas, puede utilizar la función del sistema @@ROWCOUNT.

Ventajas

- a) **En restricciones de diseño:** pueden utilizarse activadores para hacer que se cumplan restricciones que pueden ser aplicadas a niveles de diseño y ejecución de DBMS.
- b) **Más funcionalidad:** Los activadores agregan funcionalidad con la automatización de acciones críticas y el suministro de advertencias y sugerencias apropiadas para la realización de acciones reparadoras. De hecho, uno de los usos más comunes de los activadores es mejorar la aplicación de la integridad referencial.
- c) **Más poder de procesamiento:** Los activadores agregan poder de procesamiento al RDBMS y al sistema de BD como un todo.
- d) La misma acción del desencadenador puede definirse para más de una acción del usuario (por ejemplo, INSERT y UPDATE) en la misma instrucción CREATE TRIGGER
- e) Los desencadenadores son adecuados para mantener la integridad de los datos en el nivel inferior, pero *no* para obtener resultados de consultas.
- f) La ventaja principal de los desencadenadores consiste en que pueden contener lógica compleja de proceso.
- g) Los desencadenadores pueden hacer cambios en cascada en tablas relacionadas de una base de datos, mantener datos *no normalizados* y comparar el estado de los datos antes y después de su modificación.
- h) **Los desencadenadores pueden exigir una integridad de datos más compleja que una restricción CHECK.** A diferencia de las restricciones CHECK, los desencadenadores pueden hacer referencia a columnas de otras tablas. Por ejemplo, podría colocar un desencadenador de inserción en la tabla **Order Details** que compruebe la columna **UnitsInStock** de ese artículo en la tabla **Products**. El desencadenador podría determinar que, cuando el valor **UnitsInStock** sea menor de 10, la cantidad máxima de pedido sea tres artículos. Este tipo de comprobación hace referencia a columnas de otras tablas. Con una restricción CHECK esto no se permite. Los desencadenadores son útiles para asegurar la realización de las acciones adecuadas cuando deban efectuarse eliminaciones o actualizaciones en cascada. Si hay restricciones en la tabla del desencadenador, se comprueban antes de la ejecución del mismo. Si se infringen las restricciones, el desencadenador no se ejecuta.
- i) **Definición de mensajes de error personalizados.** Los desencadenadores permiten invocar mensajes de error personalizados predefinidos o dinámicos cuando se den determinadas condiciones durante la ejecución del desencadenador.
Raiserror(cadena, num1, num2)
Num1: 10 errores informales, **11-16 definidos por el usuario** (pueden ser corregidos por el usuario), **17-25 errores del software o hardware** (corregidos por el admndor del sistema), los **17-19** te permiten seguir trabajando, pero no se ejecutarán ciertas estructuras, el admndor debe imprimir los errores de 17 a 25 para resolverlos.
Error 17: recursos insuficientes: falta disco duro

Error 18: error no fatal detectado: hay problemas con el software pero la estructura termina y se mantiene la conexión con el servidor. Cuando la consulta detecta un error interno durante la optimización de la consulta.

Error 19: error en un recurso: algunos mensajes indican límite interno no configurable. Y el proceso por lotes actual no termina. Ocurre raramente pero debe ser corregido por el administrador.

Error 20 a 25 Errores fatales que dicen que el proceso no está corriendo. La conexión del cliente se cierra.

Desventajas

- a) **Afecta el desempeño de la BD.** El activador se activa y realiza una sentencia de actualización de todas las filas de la tabla indicada, aunque la sentencia de activación actualiza solo una. Esto puede afectar el desempeño de la BD.
- b) **El desencadenador se crea solamente en la BD actual;** sin embargo, un desencadenador puede hacer referencia a objetos que están fuera de la BD actual.
- c) **Solo se pueden aplicar a una tabla.** Los desencadenadores se definen para una tabla específica, denominada tabla del desencadenador.
- d) **Invocación automática.** Cuando se intenta insertar, actualizar o eliminar datos de una tabla en la que se ha definido un desencadenador para esa acción específica, el desencadenador se ejecuta automáticamente. No es posible evitar su ejecución.
- e) **Imposibilidad de llamada directa.** A diferencia de los procedimientos almacenados de sistema normales, no es posible invocar directamente los desencadenadores, que tampoco pasan ni aceptan parámetros. Se pueden utilizar desencadenadores para hacer actualizaciones y eliminaciones en cascada en tablas relacionadas de una base de datos. Observe que muchas bases de datos creadas con versiones anteriores de Microsoft SQL Server pueden contener este tipo de desencadenadores. Por ejemplo, un desencadenador de eliminación en la tabla **Products** de la base de datos **Northwind** puede eliminar de otras tablas las filas que tengan el mismo valor que la fila **ProductID** eliminada. Para ello, el desencadenador utiliza la columna de clave externa **ProductID** como una forma de ubicar las filas de la tabla **Order Details**.

5. Procedimientos almacenados

5.1. Definición

Los procedimientos almacenados o guardados son un conjunto nombrado de sentencias SQL y de procedimientos. Este conjunto nombrado se guarda en la base de datos. Los procedimientos guardados son invocados por nombre.

Un procedimiento almacenado es un procedimiento de base de datos, similar a un procedimiento en otros lenguajes de programación, que está contenido en la misma base de datos. En SQL Server, se pueden crear procedimientos almacenados mediante Transact-SQL o mediante Common Language Runtime (CLR) y uno de los lenguajes de programación de Visual Studio 2005 como por ejemplo Visual Basic o C#. Por lo general, los procedimientos almacenados de SQL Server pueden realizar lo siguiente:

- Aceptar parámetros de entrada y devolver múltiples valores en forma de parámetros de salida al procedimiento de llamada o el lote.
- Contener instrucciones de programación que realicen operaciones en la base de datos, incluyendo llamadas a otros procedimientos.
- Devolver un valor de estado a un procedimiento de llamada o lote para indicar el éxito o el error (y la razón del error).

Los procedimientos guardados se invocan como una unidad. Por consiguiente, cuando desea representarse una transacción de activación múltiple, puede crearse un procedimiento que se guarda en la base de datos. El contenido completo del procedimiento guardado se transmite y ejecuta, con lo que se evita la transmisión y ejecución de sentencias SQL individuales por la red. Por consiguiente, el uso de procedimientos guardados reduce sustancialmente el tráfico por la red, con lo que se mejora el rendimiento.

Sintaxis

Para crear un procedimiento guardado se utiliza la siguiente sintaxis:

```
CREATE PROCEDURE nombre_Proc (argumento [OUTPUT] tipo de dato, etc)
AS BEGIN
  DECLARE nombre_var Tipo de dato [= predeterminado] [OUTPUT]
  PL/SQL o SQL enunciados;
END;
```

Como ayuda para interpretar las líneas del procedimiento guardado se observa que:

- DECLARE se utiliza para especificar las variables utilizadas en el procedimiento. Debe especificarse tanto el nombre de la variable como el tipo de datos.
- Argumento: especifica los parámetros que se trasladan al procedimiento guardado.
- OUTPUT indica si el parámetro es de SALIDA
- Tipo de dato es uno de los tipos de datos SQL de procedimientos utilizado en el RDBMS. Los tipos de datos normalmente concuerdan con los tipos de datos utilizados en la sentencia de creación de la tabla RDBMS.

Llamada de un procedimiento guardado

```
EXEC NomProcGuard parámetros
```

También puede invocarse un procedimiento del interior de un activador.

Se creará un procedimiento guardado para ajustar los valores de la columna P_REORDER.

```
Create procedure prod_reorder_set
As begin
  Update product
  Set p_reorder = 1
  Where p_onhand <= p_min_order;
  Update product
  Set p_reorder = 0
  Where p_onhand > p_min_order;
End;
```

Se ejecuta el procedimiento prod_reorder_set

```
EXEC prod_reorder_set
```

En SQL Server existen 5 tipos de procedimientos almacenados:

- **Procedimientos del sistema:** son los que se encuentran almacenados en la BD **master** y algunas en las BD del usuario, estos brindan información acerca de los datos y características del servidor. En el nombre usan como prefijo **sp_**.
- **Procedimientos locales,** son los procedimientos almacenados en una BD.
- **Procedimientos temporales,** son los procedimientos locales y sus nombres empiezan con los prefijos # o ##, dependiendo si se desea que sea un procedimiento global a todas las conexiones o local a la conexión que lo define.
- **Procedimientos remotos,** son los procedimientos almacenados en servidores distribuidos.
- **Procedimientos extendidos,** son aquellos que nos permiten aprovechar las funcionalidades de otras librerías externas a SQL Server. Estos procedimientos usan el prefijo **xp_** y se encuentran en la BD **master**.

5.2. Algoritmo de ejecución

Se presentan algunos ejemplos de creación de procedimientos almacenados.

Crear procedimientos almacenados

Para crear un procedimiento almacenado en SQL Server.

```
CREATE PROC[EDURE] NomProc [ @parametro tipoDeDatos [= predeterminado] [OUTPUT]] [, ...n]
[WITH {RECOMPILE | ENCRYPTION}]
AS
  Sentencias SQL [1..n]
```

Donde:

@parámetro: El usuario puede tener hasta un máximo de 1024 parámetros. El nombre del parámetro debe comenzar con un signo @. Los parámetros son locales al procedimiento.

Predeterminado: es un valor predeterminado para el parámetro.

OUTPUT: indica que se trata de un parámetro de salida. El valor de esta opción puede devolverse a EXEC[CUTE]. Utilice los parámetros OUTPUT para devolver información al procedimiento que llama.

{RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION}

RECOMPILE indica que SQL Server no almacena en la caché un plan para este procedimiento, con lo que el procedimiento se vuelve a compilar cada vez que se ejecuta. Utilice la opción RECOMPILE cuando emplee valores atípicos o temporales para no anular el plan de ejecución que está almacenado en la memoria caché.

ENCRYPTION indica que SQL Server codifica la entrada de la tabla **syscomment** que contiene el texto de la instrucción CREATE PROCEDURE.

Entre otras observaciones podemos mencionar:

- El tamaño máximo de un procedimiento es de 128 Mb.
- Un procedimiento sólo puede crearse en la BD actual.
- Se puede crear otros objetos de BD dentro de un procedimiento almacenado. Puede hacer referencia a un objeto creado en el mismo procedimiento almacenado, siempre que se cree antes de que se haga referencia
- Puede hacer referencia a tablas temporales dentro de un procedimiento almacenado.

Nota: Si ejecuta un procedimiento almacenado que llama a otro procedimiento almacenado, el procedimiento al que se llama podrá tener acceso a todos los objetos creados por el primer procedimiento, incluida las tablas temporales.

Las siguientes instrucciones no se pueden emplear dentro de un procedimiento:

- CREATE TRIGGER
- CREATE VIEW
- CREATE PROCEDURE

Crear el siguiente procedimiento:

Use Matriculas

Go

CREATE PROCEDURE ListaPromedios

As

SELECT NombreApe = (Nom + ' ' + Pat + ' ' + Mat), Inscritos.Sec,

Curso = Case

When Substring(Inscritos.Sec, 2, 1) = '1'

Then 'V. Basic'

When Substring(Inscritos.Sec, 2, 1) = '2'

Then 'V. Fox'

When Substring(Inscritos.Sec, 2, 1) = '3'

Then 'SQL Server'

End,

Promedio = (N1 + N2)/2

From Alumnos INNER JOIN Inscritos

ON Alumnos.codalu = Inscritos.codalu

INNER JOIN Calificaciones

ON Inscritos.codalu = Calificaciones.codalu

AND Inscritos.sec = Calificaciones.sec

GO

Para poder ejecutar el procedimiento:

EXEC ListaPromedios

Para ver la información de la implementación del procedimiento almacenado:

Sp_HelpText ListaPromedios

Para ver cuales son las columnas que producen la información presentada por el procedimiento.

Sp_Depends ListaPromedios

Modificar Procedimientos Almacenados

Si se desea modificar el procedimiento almacenado utilizar la siguiente sintaxis:

ALTER PROC[EDURE] NombreProc [{ @parametro tipoDeDatos } [=predeterminado][OUTPUT]] [...n]

[WITH {RECOMPILE | ENCRYPTACION}]

AS

Sentencias SQL [...n]

Para ver un ejemplo realice lo siguiente:

```
ALTER PROCEDURE ListaPromedios
WITH ENCRYPTION
As
    SELECT NombreApe = (Nom + ' ' + Pat + ' ' + Mat), Inscritos.Sec,
        Curso = Case
            When Substring(Inscritos.Sec, 2, 1) = '1'
                Then 'V. Basic'
            When Substring(Inscritos.Sec, 2, 1) = '2'
                Then 'V. Fox'
            When Substring(Inscritos.Sec, 2, 1) = '3'
                Then 'SQL Server'
            End,
        Promedio = (N1 + N2)/2
    From Alumnos INNER JOIN Inscritos
        ON Alumnos.codalu = Inscritos.codalu
        INNER JOIN Calificaciones
        ON Inscritos.codalu = Calificaciones.codalu
        AND Inscritos.sec = Calificaciones.sec
GO
```

Ejemplo:

Se implementa un procedimiento que muestre el promedio de cada alumno de acuerdo a la sección indicada en el argumento:

```
CREATE PROCEDURE PromPorSeccion (@seccion char(4) = NULL)
As
    IF @seccion IS NULL
        BEGIN
            RAISERROR('Debe indicar un código', 10,1)
            RETURN
        END
    IF NOT EXISTS (Select Sec From Inscritos Where Sec=@seccion)
        BEGIN
            RAISERROR ('La sección no tiene alumnos', 10, 1)
            RETURN
        END
    SELECT NomApe = (Nom + ' ' + Pat + ' ' + Mat), Inscritos.Sec,
        Curso = Case
            When SubString(Inscritos.Sec, 2, 1)='1'
                Then 'V.Basic'
            When SubString(Inscritos.Sec, 2, 1)='2'
                Then 'V.Fox'
            When SubString(Inscritos.Sec, 2, 1)='3'
                Then 'SQL Server'
```



```
End,  
Promedio = (N1+N2)/2  
From Alumnos INNER JOIN Inscritos  
On Alumnos.codalu = Inscritos.codalu  
INNER JOIN Calificaciones  
On Inscritos.codalu = Calificaciones.codalu  
AND Inscritos.sec = Calificaciones.sec  
WHERE Inscritos.Sec = @Seccion  
GO
```

Para ejecutar realizaremos lo siguiente:

```
EXEC PromPorSeccion  
GO  
EXEC PromPorSeccion '9090'  
GO  
EXEC PromPorSeccion '2315'  
GO  
Sp_HelpText PromPorSeccion  
GO  
Sp_Depends PromPorSeccion  
GO
```

Eliminar Procedimientos Almacenados

Para eliminar un procedimiento almacenado utilice el siguiente formato:

```
DROP PROCEDURE <Nombre del procedimiento>
```

5.3. Usos

Usar un procedimiento almacenado sin parámetros

El tipo más sencillo de procedimiento almacenado de SQL Server al que puede llamar es el que no contiene parámetros y devuelve un solo conjunto de resultados. El controlador JDBC de Microsoft SQL Server 2005 ofrece la clase **SQLServerStatement**, que puede usar para llamar a este tipo de procedimiento almacenado y procesar los datos que devuelve.

Si se usa el controlador JDBC para llamar a un procedimiento almacenado sin parámetros, se debe usar la secuencia de escape de SQL call. La sintaxis de la secuencia de escape call sin parámetros es la siguiente:

```
{call NomDelProc}
```

```
CREATE PROCEDURE GetContactFormalNames  
AS  
BEGIN  
SELECT TOP 10 Title + ' ' + FirstName + ' ' + LastName AS FormalName  
FROM Person.Contact  
END
```

Este procedimiento almacenado devuelve un solo conjunto de resultados que contiene una columna de datos, compuesta por una combinación del título y el nombre de los diez primeros contactos de la tabla Person.Contact.

En el siguiente ejemplo, se pasa a la función una conexión abierta a la base de datos de ejemplo AdventureWorks y se usa el método **executeQuery** para llamar al procedimiento almacenado GetContactFormalNames.

```
public static void executeSprocNoParams(Connection con) {
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("{call dbo.GetContactFormalNames}");

        while (rs.next()) {
            System.out.println(rs.getString("FormalName"));
        }
        rs.close();
        stmt.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}}
```

Usar un procedimiento almacenado con parámetros de entrada

Un SQL Server procedimiento almacenado al que se puede llamar es aquel que contiene uno o más parámetros IN, parámetros que se pueden usar para pasar datos al procedimiento almacenado. El controlador JDBC de Microsoft SQL Server 2005 ofrece la clase SQLServerPreparedStatement, que puede usar para llamar a este tipo de procedimiento almacenado y procesar los datos que devuelve.

Si se usa el controlador JDBC para llamar a un procedimiento almacenado con los parámetros IN, debe usar la secuencia de escape de SQL call junto con el método prepareCall de la clase SQLServerConnection. La sintaxis de la secuencia de escape call con los parámetros IN es la siguiente:

```
{call procedure-name([parameter],[parameter]...)}
```

Al crear la secuencia de escape call, especifique los parámetros IN mediante el carácter ? (signo de interrogación). Este carácter actúa como un marcador de posición para los valores de parámetros pasados al procedimiento almacenado. Para especificar un valor de un parámetro, puede usar uno de los métodos de establecimiento de la clase SQLServerPreparedStatement. El método de establecimiento que puede usar se determina mediante el tipo del parámetro IN.

Cuando pasa un valor al método establecedor, debe especificar no sólo el valor real que se usará en el parámetro, sino el lugar ordinal que ocupa el parámetro en el procedimiento almacenado. Por ejemplo, si su procedimiento almacenado contiene un solo parámetro IN, su valor ordinal será 1. Si el procedimiento almacenado contiene dos parámetros, el primer valor ordinal será 1 y el segundo 2.

Como ejemplo de cómo llamar a un procedimiento que contiene un parámetro IN, use el procedimiento almacenado `uspGetEmployeeManagers` de la base de datos de ejemplo `AdventureWorks` de `SQL Server 2005`. Este procedimiento almacenado acepta un solo parámetro de entrada llamado `EmployeeID` (Id. del empleado), que es un valor entero, y devuelve una lista recursiva de empleados y sus jefes según el `EmployeeID` especificado. El código Java para llamar a este procedimiento almacenado es el siguiente:

```
public static void executeSprocInParams(Connection con) {
    try {
        PreparedStatement pstmt = con.prepareStatement("{ call
        dbo.uspGetEmployeeManagers(?)}");
        pstmt.setInt(1, 50);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            System.out.println("EMPLOYEE:");
            System.out.println(rs.getString("LastName") + ", " + rs.getString("FirstName"));
            System.out.println("MANAGER:");
            System.out.println(rs.getString("ManagerLastName") + ", " +
            rs.getString("ManagerFirstName"));
            System.out.println();
        }
        rs.close();
        pstmt.close();
    }

    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Usar un procedimiento almacenado con parámetros de salida

Un procedimiento almacenado de `SQL Server` al que se puede llamar es el que devuelve uno o más parámetros OUT, que son los parámetros que el procedimiento almacenado usa para devolver los datos a la aplicación que realiza la llamada. El controlador JDBC de `Microsoft SQL Server 2005` ofrece la clase **SQLServerCallableStatement**, que puede usar para llamar a este tipo de procedimiento almacenado y procesar los datos que devuelve.

Cuando se llama a este tipo de procedimiento almacenado con el controlador JDBC, se debe usar la secuencia de escape de SQL `call` junto con el método **prepareCall** de la clase **SQLServerConnection**. La sintaxis para la secuencia de escape `call` con parámetros OUT es la siguiente:

```
{call procedure-name([parameter],[parameter]]...)}
```

Al crear la secuencia de escape `call`, especifique los parámetros OUT mediante el carácter `?` (signo de interrogación). Este carácter actúa como un marcador de posición para los valores de parámetros devueltos por el procedimiento almacenado. Para especificar un valor para un parámetro OUT, debe especificar el tipo de datos de cada parámetro mediante

el método **registerOutParameter** de la clase **SQLServerCallableStatement** antes de ejecutar el procedimiento almacenado.

Además, al pasar un valor al método **registerOutParameter** para un parámetro OUT, debe especificar no sólo el tipo de datos usado para el parámetro, sino también la posición ordinal del parámetro en el procedimiento almacenado. Por ejemplo, si el procedimiento almacenado contiene un solo parámetro OUT, su valor ordinal es 1 y, si el procedimiento almacenado contiene dos parámetros, el primer valor ordinal es 1 y el segundo 2.

Nota: El controlador JDBC no admite el uso de los tipos de datos CURSOR, SQLVARIANT, TABLE y TIMESTAMP SQL Server como parámetros OUT.

Cree, a modo de ejemplo, el siguiente procedimiento almacenado en la base de datos de ejemplo AdventureWorks de SQL Server 2005:

```
CREATE PROCEDURE GetImmediateManager
    @employeeID INT,
    @managerID INT OUTPUT
AS
BEGIN
    SELECT @managerID = ManagerID
    FROM HumanResources.Employee
    WHERE EmployeeID = @employeeID
END
```

Este procedimiento almacenado devuelve un solo parámetro OUT (managerID), que es un entero, en función del parámetro IN (employeeID) especificado, que también es un entero. El valor devuelto en el parámetro OUT es ManagerID en función de EmployeeID en la tabla HumanResources.Employee.

En el siguiente ejemplo, se pasa una conexión abierta a la base de datos de ejemplo AdventureWorks a la función y se usa el método **execute** para llamar al procedimiento almacenado GetImmediateManager:

```
public static void executeStoredProcedure(Connection con) {
    try {
        CallableStatement cstmt = con.prepareCall("{ call dbo.GetImmediateManager(?, ?)}");
        cstmt.setInt(1, 5);
        cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
        cstmt.execute();
        System.out.println("MANAGER ID: " + cstmt.getInt(2));
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Nota: En este ejemplo se usa el método **execute** de la clase **SQLServerCallableStatement** para ejecutar el procedimiento almacenado. Se usa dicho método porque el procedimiento almacenado no ha devuelto ningún conjunto de resultados. En caso contrario, se usa el método **executeQuery**.

Usar un procedimiento almacenado con un estado de devolución

Un procedimiento almacenado de SQL Server al que puede llamar es el que devuelve un parámetro de estado o resultado. Se usa normalmente para indicar el funcionamiento correcto o incorrecto del procedimiento almacenado. El controlador JDBC de Microsoft SQL Server 2005 ofrece la clase **SQLServerCallableStatement**, que puede usar para llamar a este tipo de procedimiento almacenado y procesar los datos que devuelve.

Al llamar a este tipo de procedimiento almacenado mediante el controlador JDBC, debe usar la secuencia de escape call de SQL junto con el método **prepareCall** de la clase **SQLServerConnection**. La sintaxis para la secuencia de escape call con un parámetro de estado de devolución es la siguiente:

```
{[?]=call procedure-name([parameter][,[parameter]]...)}
```

Al crear la secuencia de escape call, especifique el parámetro de estado de devolución mediante el carácter ? (signo de interrogación). Este carácter actúa como un marcador de posición para el valor del parámetro que devolverá el procedimiento almacenado. Para especificar un valor para un parámetro de estado de devolución, debe especificar el tipo de datos del parámetro mediante el método **registerOutParameter** de la clase **SQLServerCallableStatement** antes de ejecutar el procedimiento almacenado.

Nota: Al usar el controlador JDBC con una base de datos de SQL Server, el valor especificado para el parámetro de estado de devolución del método **registerOutParameter** siempre es un entero, que se puede especificar mediante el tipo de datos `java.sql.Types.INTEGER`.

Además, al pasar un valor al método **registerOutParameter** para un parámetro de estado de devolución, debe especificar no sólo los tipos de datos usados para el parámetro, sino también la posición ordinal del parámetro en la llamada al procedimiento almacenado. En el caso del parámetro de estado de devolución, su posición ordinal siempre es 1 debido a que es siempre el primer parámetro de la llamada al procedimiento almacenado.

Cree, a modo de ejemplo, el siguiente procedimiento almacenado en la base de datos de ejemplo AdventureWorks de SQL Server 2005:

```
CREATE PROCEDURE CheckContactCity @cityName CHAR(50)
AS
BEGIN
    IF ((SELECT COUNT(*)
        FROM Person.Address
        WHERE City = @cityName) > 1)
        RETURN 1
    ELSE
        RETURN 0
END
```

Este procedimiento almacenado devuelve un valor de estado de 1 ó 0 en función de si la ciudad especificada en el parámetro cityName se encuentra en la tabla Person.Address.

En el siguiente ejemplo, se pasa una conexión abierta a la base de datos de ejemplo AdventureWorks a la función y se usa el método **execute** para llamar al procedimiento almacenado CheckContactCity:

```
public static void executeStoredProcedure(Connection con) {
```

```

try {
    CallableStatement cstmt = con.prepareCall("{? = call dbo.CheckContactCity(?)}");
    cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
    cstmt.setString(2, "Atlanta");
    cstmt.execute();
    System.out.println("RETURN STATUS: " + cstmt.getInt(1));
}
cstmt.close();
catch (Exception e) {
    e.printStackTrace();
}
}

```

Usar un procedimiento almacenado con un recuento de actualizaciones

Para modificar los datos de una base de datos de SQL Server con un procedimiento almacenado, el controlador JDBC de Microsoft SQL Server 2005 proporciona la clase **SQLServerCallableStatement**. Con la clase `SQLServerCallableStatement` puede llamar a los procedimientos almacenados que modifican los datos contenidos en la base de datos y devuelven un recuento del número de filas afectadas, lo que se denomina recuento de actualizaciones.

Una vez configurado el procedimiento almacenado con la clase `SQLServerCallableStatement`, puede llamar al procedimiento almacenado con el método **execute** o **executeUpdate**. El método `executeUpdate` devuelve un valor entero que contiene el número de filas afectadas por el procedimiento almacenado, mientras que el método `execute` no lo hace. Si usa el método `execute` y desea obtener el recuento del número de filas afectadas, puede llamar al método **getUpdateCount** después de ejecutar el procedimiento almacenado.

Nota: Si desea que el controlador JDBC devuelva todos los recuentos de actualizaciones, incluidos los recuentos de actualizaciones devueltos por todos los desencadenadores activados, establezca la propiedad de cadena de conexión `lastUpdateCount` en "false". Para obtener más información sobre la propiedad `lastUpdateCount`.

A modo de ejemplo, cree la tabla y el procedimiento almacenado siguientes en la base de datos de ejemplo AdventureWorks de SQL Server 2005:

```

CREATE TABLE TestTable
    (Col1 int IDENTITY,
    Col2 varchar(50),
    Col3 int);

```

```

CREATE PROCEDURE UpdateTestTable
    @Col2 varchar(50),
    @Col3 int
AS
BEGIN
    UPDATE TestTable
    SET Col2 = @Col2, Col3 = @Col3
END;

```

En el siguiente ejemplo, se pasa una conexión abierta a la base de datos de ejemplo AdventureWorks a la función, se usa el método **execute** para llamar al procedimiento almacenado UpdateTestTable y, a continuación, se usa el método **getUpdateCount** para devolver un recuento de las filas afectadas por el procedimiento almacenado.

```
public static void executeUpdateStoredProcedure(Connection con) {
    try {
        CallableStatement cstmt = con.prepareCall("{ call dbo.UpdateTestTable(?, ?)}");
        cstmt.setString(1, "A");
        cstmt.setInt(2, 100);
        cstmt.execute();
        int count = cstmt.getUpdateCount();
        cstmt.close();

        System.out.println("ROWS AFFECTED: " + count);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

5.4. Ventajas

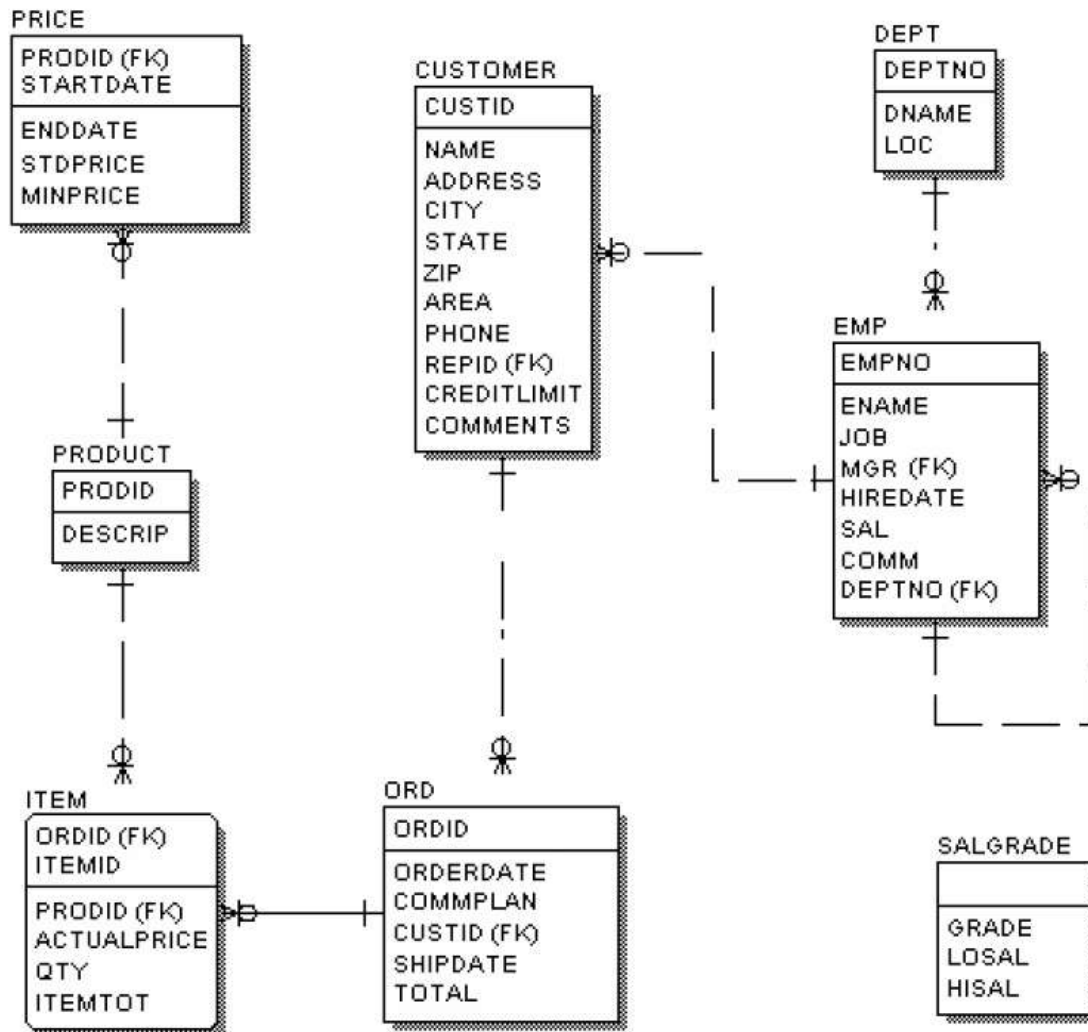
- Pueden reducir el tráfico de la red.
- Pueden utilizarse como mecanismo de seguridad, ya que se puede conceder permisos a los usuarios para ejecutar un procedimiento almacenado, incluso sino cuentan con permiso para ejecutar las instrucciones del procedimiento.
- Permiten una ejecución más rápida, ya que los procedimientos son analizados y optimizados en el momento de su creación y es posible utilizar una versión del procedimiento que se encuentra en la memoria después de que se ejecuta por primera vez.
- Mayor velocidad de respuesta ya que dicho código ya está compilado en la BD y se ejecuta mucho más rápido.

5.5. Desventajas

- Se corrompe la base de datos y se pierde toda la información ya que existe programación dentro de la base de datos pero se debe tener una buena estrategia de respaldos de la misma, si ya tenemos una buena BD en producción se debe generar los script necesarios que contengan los CREATE PROCEDURE, TABLE, VIEW, DATABASE, TRIGGER y todos los objetos de la BD. De tal manera que si le pasa algo a las BD su estructura está protegida.
- Al ejecutar sentencias SQL estableciendo una conexión entre la aplicación y la BD por la naturaleza de la aplicación Web se demora la ejecución de las sentencias (sobre todo si son pesadas).

Anexo A

Modelo Entidad-Relación



Anexo B

Script para la creación de la base de datos del modelo E-R

```
CREATE DATABASE sqlAvanzado
go
```

```
USE sqlAvanzado
go
```

```
CREATE TABLE DEPT (
  DEPTNO      NUMERIC(2) NOT NULL,
  DNAME       VARCHAR(14),
  LOC         VARCHAR(13),
  CONSTRAINT DEPT_PRIMARY_KEY PRIMARY KEY (DEPTNO)
);
```

```
CREATE TABLE EMP (
  EMPNO       NUMERIC(4) NOT NULL,
  ENAME       VARCHAR(10),
  JOB         VARCHAR(9),
  MGR         NUMERIC(4) CONSTRAINT EMP_MGR_FK REFERENCES EMP (EMPNO),
  HIREDATE    DATETIME,
  SAL         NUMERIC(7,2),
  COMM        NUMERIC(7,2),
  DEPTNO      NUMERIC(2),
  CONSTRAINT EMP_DEPTNO_FK FOREIGN KEY (DEPTNO) REFERENCES DEPT (DEPTNO),
  CONSTRAINT EMP_EMPNO_PK PRIMARY KEY (EMPNO)
);
```

```
CREATE TABLE SALGRADE (
  GRADE       NUMERIC,
  LOSAL       NUMERIC,
  HISAL       NUMERIC
);
```

```
CREATE TABLE CUSTOMER (
  CUSTID      NUMERIC (6) NOT NULL,
  NAME        VARCHAR (45),
  ADDRESS     VARCHAR (40),
  CITY        VARCHAR (30),
  STATE       VARCHAR (2),
  ZIP         VARCHAR (9),
  AREA        NUMERIC (3),
  PHONE       VARCHAR (9),
  REPID       NUMERIC (4),
  CREDITLIMIT NUMERIC (9,2),
  COMMENTS    TEXT,
  CONSTRAINT CUSTOMER_CUSTID_PK PRIMARY KEY (CUSTID),
  CONSTRAINT CUSTOMER_CUSTID_CK CHECK (CUSTID > 0),
  CONSTRAINT CUSTOMER_REPID_FK FOREIGN KEY (REPID) REFERENCES EMP(EMPNO)
);
```

```
CREATE TABLE ORD (
  ORDID      NUMERIC (4) NOT NULL,
  ORDERDATE  DATETIME,
  COMMPLAN   VARCHAR (1),
  CUSTID     NUMERIC (6),
  SHIPDATE   DATETIME,
  TOTAL      NUMERIC (8,2) CONSTRAINT ORD_TOTAL_CK CHECK (TOTAL >= 0),
  CONSTRAINT ORD_CUSTID_FK FOREIGN KEY (CUSTID) REFERENCES CUSTOMER (CUSTID),
  CONSTRAINT ORD_ORDID_PK PRIMARY KEY (ORDID)
);
```

```
CREATE TABLE PRODUCT (
  PRODID     NUMERIC (6) CONSTRAINT PRODUCT_PRODID_PK PRIMARY KEY,
  DESCRIP    VARCHAR (30)
);
```

```
CREATE TABLE PRICE (
  PRODID     NUMERIC (6) NOT NULL,
  STARTDATE  DATETIME NOT NULL,
  ENDDATE    DATETIME,
  STDPRICE   NUMERIC (8,2),
  MINPRICE   NUMERIC (8,2),
  CONSTRAINT PRICE_PRODID_STARTDATE_PK PRIMARY KEY (PRODID, STARTDATE),
  CONSTRAINT PRICE_PRODID_FK FOREIGN KEY (PRODID) REFERENCES PRODUCT (PRODID)
);
```

```
CREATE TABLE ITEM (
  ORDID      NUMERIC (4) NOT NULL,
  ITEMID     NUMERIC (4) NOT NULL,
  PRODID     NUMERIC (6),
  ACTUALPRICE NUMERIC (8,2),
  QTY        NUMERIC (8),
  ITEMTOT    NUMERIC (8,2),
  CONSTRAINT ITEM_ORDID_FK FOREIGN KEY (ORDID) REFERENCES ORD (ORDID),
  CONSTRAINT ITEM_ORDID_ITEMID_PK PRIMARY KEY (ORDID, ITEMID),
  CONSTRAINT ITEM_PRODID_FK FOREIGN KEY (PRODID) REFERENCES PRODUCT (PRODID)
);
```

```
INSERT INTO DEPT VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO DEPT VALUES (20,'RESEARCH','DALLAS');
INSERT INTO DEPT VALUES (30,'SALES','CHICAGO');
INSERT INTO DEPT VALUES (40,'OPERATIONS','BOSTON');
```

```
-----
INSERT INTO EMP VALUES (7839,'KING','PRESIDENT',NULL,'17-nov-81',5000,NULL,10);
INSERT INTO EMP VALUES (7698,'BLAKE','MANAGER',7839,'1-may-81',2850,NULL,30);
INSERT INTO EMP VALUES (7782,'CLARK','MANAGER',7839,'9-jun-81',2450,NULL,10);
INSERT INTO EMP VALUES (7566,'JONES','MANAGER',7839,'2-apr-81',2975,NULL,20);
INSERT INTO EMP VALUES (7654,'MARTIN','SALESMAN',7698,'28-sep-81',1250,1400,30);
INSERT INTO EMP VALUES (7499,'ALLEN','SALESMAN',7698,'20-feb-81',1600,300,30);
INSERT INTO EMP VALUES (7844,'TURNER','SALESMAN',7698,'8-sep-81',1500,0,30);
INSERT INTO EMP VALUES (7900,'JAMES','CLERK',7698,'3-dec-81',950,NULL,30);
INSERT INTO EMP VALUES (7521,'WARD','SALESMAN',7698,'22-feb-81',1250,500,30);
INSERT INTO EMP VALUES (7902,'FORD','ANALYST',7566,'3-dec-81',3000,NULL,20);
INSERT INTO EMP VALUES (7369,'SMITH','CLERK',7902,'17-dec-80',800,NULL,20);
```

```
INSERT INTO EMP VALUES (7788,'SCOTT','ANALYST',7566,'09-dec-82',3000,NULL,20);
INSERT INTO EMP VALUES (7876,'ADAMS','CLERK',7788,'12-jan-83',1100,NULL,20);
INSERT INTO EMP VALUES (7934,'MILLER','CLERK',7782,'23-jan-82',1300,NULL,10);
```

```
INSERT INTO SALGRADE VALUES (1,700,1200);
INSERT INTO SALGRADE VALUES (2,1201,1400);
INSERT INTO SALGRADE VALUES (3,1401,2000);
INSERT INTO SALGRADE VALUES (4,2001,3000);
INSERT INTO SALGRADE VALUES (5,3001,9999);
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('96711', 'CA', '7844', '598-6609',
    'JOCKSPORTS',
    '100', '5000', 'BELMONT', '415', '345 VIEWRIDGE',
    'Very friendly people to work with -- sales rep likes to be called Mike.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('94061', 'CA', '7521', '368-1223',
    'TKB SPORT SHOP',
    '101', '10000', 'REDWOOD CITY', '415', '490 BOLI RD.',
    'Rep called 5/8 about change in order - contact shipping.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('95133', 'CA', '7654', '644-3341',
    'VOLLYRITE',
    '102', '7000', 'BURLINGAME', '415', '9722 HAMILTON',
    'Company doing heavy promotion beginning 10/89. Prepare for large orders during
    winter.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('97544', 'CA', '7521', '677-9312',
    'JUST TENNIS',
    '103', '3000', 'BURLINGAME', '415', 'HILLVIEW MALL',
    'Contact rep about new line of tennis rackets.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('93301', 'CA', '7499', '996-2323',
    'EVERY MOUNTAIN',
    '104', '10000', 'CUPERTINO', '408', '574 SURRY RD.',
    'Customer with high market share (23%) due to aggressive advertising.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('91003', 'CA', '7844', '376-9966',
    'K + T SPORTS',
    '105', '5000', 'SANTA CLARA', '408', '3476 EL PASEO',
    'Tends to order large amounts of merchandise at once. Accounting is considering
    raising their credit limit. Usually pays on time.');
```

```
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('94301', 'CA', '7521', '364-9777',
    'SHAPE UP',
    '106', '6000', 'PALO ALTO', '415', '908 SEQUOIA',
    'Support intensive. Orders small amounts (< 800) of merchandise at a time.');
```

```

INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('93301', 'CA', '7499', '967-4398',
    'WOMENS SPORTS',
    '107', '10000', 'SUNNYVALE', '408', 'VALCO VILLAGE',
    'First sporting goods store geared exclusively towards women. Unusual promotion
    al style and very willing to take chances towards new products!');
INSERT INTO CUSTOMER (ZIP, STATE, REPID, PHONE, NAME, CUSTID, CREDITLIMIT,
    CITY, AREA, ADDRESS, COMMENTS)
VALUES ('55649', 'MN', '7844', '566-9123',
    'NORTH WOODS HEALTH AND FITNESS SUPPLY CENTER',
    '108', '8000', 'HIBBING', '612', '98 LONE PINE WAY', '');

INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('101.4', '08-jan-87', '610', '07-jan-87', '101', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('45', '11-jan-87', '611', '11-jan-87', '102', 'B');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('5860', '20-jan-87', '612', '15-jan-87', '104', 'C');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('2.4', '30-may-86', '601', '01-may-86', '106', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('56', '20-jun-86', '602', '05-jun-86', '102', 'B');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('698', '30-jun-86', '604', '15-jun-86', '106', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('8324', '30-jul-86', '605', '14-jul-86', '106', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('3.4', '30-jul-86', '606', '14-jul-86', '100', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('97.5', '15-aug-86', '609', '01-aug-86', '100', 'B');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('5.6', '18-jul-86', '607', '18-jul-86', '104', 'C');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('35.2', '25-jul-86', '608', '25-jul-86', '104', 'C');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('224', '05-jun-86', '603', '05-jun-86', '102', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('4450', '12-mar87', '620', '12-mar87', '100', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('6400', '01-feb-87', '613', '01-feb-87', '108', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('23940', '05-feb-87', '614', '01-feb-87', '102', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('764', '10-feb-87', '616', '03-feb-87', '103', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('1260', '04-feb-87', '619', '22-feb-87', '104', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('46370', '03-mar87', '617', '05-feb-87', '105', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('710', '06-feb-87', '615', '01-feb-87', '107', '');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('3510.5', '06-mar87', '618', '15-feb-87', '102', 'A');
INSERT INTO ORD (TOTAL, SHIPDATE,ORDID, ORDERDATE, CUSTID, COMMPPLAN)
VALUES ('730', '01-jan-87', '621', '15-mar87', '100', 'A');

```

```
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('100860', 'ACE TENNIS RACKET I');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('100861', 'ACE TENNIS RACKET II');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('100870', 'ACE TENNIS BALLS-3 PACK');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('100871', 'ACE TENNIS BALLS-6 PACK');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('100890', 'ACE TENNIS NET');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('101860', 'SP TENNIS RACKET');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('101863', 'SP JUNIOR RACKET');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('102130', 'RH: "GUIDE TO TENNIS"');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('200376', 'SB ENERGY BAR-6 PACK');
INSERT INTO PRODUCT (PRODID, DESCRIP)
VALUES ('200380', 'SB VITA SNACK-6 PACK');
```

```
INSERT INTO ITEM (QTY, PRODID, ORCID, ITEMTOT, ITEMID, ACTUALPRICE)
VALUES ('1', '100890', '610', '58', '3', '58');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '1', '100861', '611', '45', '1', '45');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '100', '100860', '612', '3000', '1', '30');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '1', '200376', '601', '2.4', '1', '2.4');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '20', '100870', '602', '56', '1', '2.8');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '3', '100890', '604', '174', '1', '58');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '2', '100861', '604', '84', '2', '42');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '10', '100860', '604', '440', '3', '44');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '4', '100860', '603', '224', '2', '56');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '1', '100860', '610', '35', '1', '35');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '3', '100870', '610', '8.4', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '200', '200376', '613', '440', '4', '2.2');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '444', '100860', '614', '15540', '1', '35');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '1000', '100870', '614', '2800', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '20', '100861', '612', '810', '2', '40.5');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('150', '101863', '612', '1500', '3', '10');
INSERT INTO ITEM ( QTY , PRODID , ORCID , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '100860', '620', '350', '1', '35');
```

```

INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1000', '200376', '620', '2400', '2', '2.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('500', '102130', '620', '1700', '3', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ( '100', '100871', '613', '560', '1', '5.6');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('200', '101860', '613', '4800', '2', '24');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('150', '200380', '613', '600', '3', '4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '102130', '619', '340', '3', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '100860', '617', '1750', '1', '35');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '100861', '617', '4500', '2', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1000', '100871', '614', '5600', '3', '5.6');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '100861', '616', '450', '1', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '100870', '616', '140', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('2', '100890', '616', '116', '3', '58');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '102130', '616', '34', '4', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '200376', '616', '24', '5', '2.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '200380', '619', '400', '1', '4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '200376', '619', '240', '2', '2.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('4', '100861', '615', '180', '1', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1', '100871', '607', '5.6', '1', '5.6');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '100870', '615', '280', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('500', '100870', '617', '1400', '3', '2.8');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('500', '100871', '617', '2800', '4', '5.6');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('500', '100890', '617', '29000', '5', '58');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '101860', '617', '2400', '6', '24');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('200', '101863', '617', '2500', '7', '12.5');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '102130', '617', '340', '8', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('200', '200376', '617', '480', '9', '2.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('300', '200380', '617', '1200', '10', '4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('5', '100870', '609', '12.5', '2', '2.5');

```

```

INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1', '100890', '609', '50', '3', '50');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('23', '100860', '618', '805', '1', '35');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '100861', '618', '2255.5', '2', '45.11');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '100870', '618', '450', '3', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '100861', '621', '450', '1', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '100870', '621', '280', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '100871', '615', '250', '3', '5');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1', '101860', '608', '24', '1', '24');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('2', '100871', '608', '11.2', '2', '5.6');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1', '100861', '609', '35', '1', '35');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('1', '102130', '606', '3.4', '1', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '100861', '605', '4500', '1', '45');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('500', '100870', '605', '1400', '2', '2.8');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('5', '100890', '605', '290', '3', '58');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '101860', '605', '1200', '4', '24');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '101863', '605', '900', '5', '9');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('10', '102130', '605', '34', '6', '3.4');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('100', '100871', '612', '550', '4', '5.5');
INSERT INTO ITEM ( QTY , PRODIG , ORDIG , ITEMTOT , ITEMID , ACTUALPRICE)
VALUES ('50', '100871', '619', '280', '4', '5.6');

```

```

INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('4.8', '01-jan-85', '100871', '3.2', '01-dec-85');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('58', '01-jan-85', '100890', '46.4', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('54', '01-jun-84', '100890', '40.5', '31-may-84');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('35', '01-jun-86', '100860', '28', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('32', '01-jan-86', '100860', '25.6', '31-may-86');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('30', '01-jan-85', '100860', '24', '31-dec-85');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('45', '01-jun-86', '100861', '36', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PRODIG, MINPRICE, ENDDATE)
VALUES ('42', '01-jan-86', '100861', '33.6', '31-may-86');

```

```
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('39', '01-jan-85', '100861', '31.2', '31-dec-85');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('2.8', '01-jan-86', '100870', '2.4', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('2.4', '01-jan-85', '100870', '1.9', '01-dec-85');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('5.6', '01-jan-86', '100871', '4.8', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('24', '15-feb-85', '101860', '18', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('12.5', '15-feb-85', '101863', '9.4', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('3.4', '18-aug-85', '102130', '2.8', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('2.4', '15-nov-86', '200376', '1.75', '');
INSERT INTO PRICE (STDPRICE, STARTDATE, PROID, MINPRICE, ENDDATE)
VALUES ('4', '15-nov-86', '200380', '3.2', '');

CREATE INDEX PRICE_INDEX ON PRICE(PROID, STARTDATE);
```


Anexo C

Contenido de las tablas

Tabla EMP

Name	Null?	Type

EMPNO	NOT NULL	NUMERIC (4)
ENAME		VARCHAR (10)
JOB		VARCHAR (9)
MGR		NUMERIC (4)
HIREDATE		DATETIME
SAL		NUMERIC (7, 2)
COMM		NUMERIC (7, 2)
DEPTNO	NOT NULL	NUMERIC (2)

```
SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO

7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Tabla DEPT

Name	Null?	Type
DEPTNO	NOT NULL	NUMERIC (2)
DNAME		VARCHAR (14)
LOC		VARCHAR (13)

```
SELECT * FROM dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Tabla SALGRADE

Name	Null?	Type
GRADE		NUMERIC
LOSAL		NUMERIC
HISAL		NUMERIC

```
SELECT * FROM salgrade;
```

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Tabla ORD

Name	Null?	Type
ORDID	NOT NULL	NUMERIC(4)
ORDERDATE		DATETIME
COMMPLAN		VARCHAR(1)
CUSTID	NOT NULL	NUMERIC(6)
SHIPDATE		DATETIME
TOTAL		NUMERIC(8,2)

```
SELECT * FROM ord;
```

ORDID	ORDERDATE	C	CUSTID	SHIPDATE	TOTAL
610	07-JAN-87	A	101	08-JAN-87	101.4
611	11-JAN-87	B	102	11-JAN-87	45
612	15-JAN-87	C	104	20-JAN-87	5860
601	01-MAY-86	A	106	30-MAY-86	2.4
602	05-JUN-86	B	102	20-JUN-86	56
604	15-JUN-86	A	106	30-JUN-86	698
605	14-JUL-86	A	106	30-JUL-86	8324
606	14-JUL-86	A	100	30-JUL-86	3.4
609	01-AUG-86	B	100	15-AUG-86	97.5
607	18-JUL-86	C	104	18-JUL-86	5.6
608	25-JUL-86	C	104	25-JUL-86	35.2
603	05-JUN-86		102	05-JUN-86	224
620	12-MAR-87		100	12-MAR-87	4450
613	01-FEB-87		108	01-FEB-87	6400
614	01-FEB-87		102	05-FEB-87	23940
616	03-FEB-87		103	10-FEB-87	764
619	22-FEB-87		104	04-FEB-87	1260
617	05-FEB-87		105	03-MAR-87	46370
615	01-FEB-87		107	06-FEB-87	710
618	15-FEB-87	A	102	06-MAR-87	3510.5
621	15-MAR-87	A	100	01-JAN-87	730

Tabla PRODUCT

Name	Null?	Type
PRODID	NOT NULL	NUMERIC (6)
DESCRIP		VARCHAR (30)

```
SELECT * FROM product;
```

```
PRODID DESCRIP
-----
100860 ACE TENNIS RACKET I
100861 ACE TENNIS RACKET II
100870 ACE TENNIS BALLS-3 PACK
100871 ACE TENNIS BALLS-6 PACK
100890 ACE TENNIS NET
101860 SP TENNIS RACKET
101863 SP JUNIOR RACKET
102130 RH: "GUIDE TO TENNIS"
200376 SB ENERGY BAR-6 PACK
200380 SB VITA SNACK-6 PACK
```

Tabla ITEM

Name	Null?	Type
ORDID	NOT NULL	NUMERIC (4)
ITEMID	NOT NULL	NUMERIC (4)
PRODID		NUMERIC (6)
ACTUALPRICE		NUMERIC (8, 2)
QTY		NUMERIC (8)
ITEMTOT		NUMERIC (8, 2)

```
SELECT * FROM item;
```

ORDID	ITEMID	PRODID	ACTUALPRICE	QTY	ITEMTOT
610	3	100890	58	1	58
611	1	100861	45	1	45
612	1	100860	30	100	3000
601	1	200376	2.4	1	2.4
602	1	100870	2.8	20	56
604	1	100890	58	3	174
604	2	100861	42	2	84
604	3	100860	44	10	440
603	2	100860	56	4	224
610	1	100860	35	1	35
610	2	100870	2.8	3	8.4
613	4	200376	2.2	200	440
614	1	100860	35	444	15540
614	2	100870	2.8	1000	2800
612	2	100861	40.5	20	810
612	3	101863	10	150	1500
620	1	100860	35	10	350
620	2	200376	2.4	1000	2400
620	3	102130	3.4	500	1700
613	1	100871	5.6	100	560
613	2	101860	24	200	4800
613	3	200380	4	150	600
619	3	102130	3.4	100	340
617	1	100860	35	50	1750
617	2	100861	45	100	4500
614	3	100871	5.6	1000	5600

Continúa en la siguiente página

Tabla ITEM (continuación)

ORDID	ITEMID	PRODID	ACTUALPRICE	QTY	ITEMTOT
616	1	100861	45	10	450
616	2	100870	2.8	50	140
616	3	100890	58	2	116
616	4	102130	3.4	10	34
616	5	200376	2.4	10	24
619	1	200380	4	100	400
619	2	200376	2.4	100	240
615	1	100861	45	4	180
607	1	100871	5.6	1	5.6
615	2	100870	2.8	100	280
617	3	100870	2.8	500	1400
617	4	100871	5.6	500	2800
617	5	100890	58	500	29000
617	6	101860	24	100	2400
617	7	101863	12.5	200	2500
617	8	102130	3.4	100	340
617	9	200376	2.4	200	480
617	10	200380	4	300	1200
609	2	100870	2.5	5	12.5
609	3	100890	50	1	50
618	1	100860	35	23	805
618	2	100861	45.11	50	2255.5
618	3	100870	45	10	450
621	1	100861	45	10	450
621	2	100870	2.8	100	280
615	3	100871	5	50	250
608	1	101860	24	1	24
608	2	100871	5.6	2	11.2
609	1	100861	35	1	35
606	1	102130	3.4	1	3.4
605	1	100861	45	100	4500
605	2	100870	2.8	500	1400
605	3	100890	58	5	290
605	4	101860	24	50	1200
605	5	101863	9	100	900
605	6	102130	3.4	10	34
612	4	100871	5.5	100	550
619	4	100871	5.6	50	280

Tabla CUSTOMER

Name	Null?	Type
CUSTID	NOT NULL	NUMERIC(6)
NAME		VARCHAR(45)
ADDRESS		VARCHAR(40)
CITY		VARCHAR(30)
STATE		VARCHAR(2)
ZIP		VARCHAR(9)
AREA		NUMERIC(3)
PHONE		VARCHAR(9)
REPID	NOT NULL	NUMERIC(4)
CREDITLIMIT		NUMERIC(9,2)
COMMENTS		TEXT

Tabla CUSTOMER (continuación)

```
SELECT * FROM customer;
```

CUSTID	NAME	ADDRESS
100	JOCKSPORTS	345 VIEWRIDGE
101	TKB SPORT SHOP	490 BOLI RD.
102	VOLLYRITE	9722 HAMILTON
103	JUST TENNIS	HILLVIEW MALL
104	EVERY MOUNTAIN	574 SURRY RD.
105	K + T SPORTS	3476 EL PASEO
106	SHAPE UP	908 SEQUOIA
107	WOMENS SPORTS	VALCO VILLAGE
108	NORTH WOODS HEALTH AND FITNESS SUPPLY CENTER	98 LONE PINE WAY

CITY	ST	ZIP	AREA	PHONE	REPID	CREDITLIMIT
BELMONT	CA	96711	415	598-6609	7844	5000
REDWOOD CITY	CA	94061	415	368-1223	7521	10000
BURLINGAME	CA	95133	415	644-3341	7654	7000
BURLINGAME	CA	97544	415	677-9312	7521	3000
CUPERTINO	CA	93301	408	996-2323	7499	10000
SANTA CLARA	CA	91003	408	376-9966	7844	5000
PALO ALTO	CA	94301	415	364-9777	7521	6000
SUNNYVALE	CA	93301	408	967-4398	7499	10000
HIBBING	MN	55649	612	566-9123	7844	8000

COMMENTS

Very friendly people to work with -- sales rep likes to be called Mike.
 Rep called 5/8 about change in order - contact shipping.
 Company doing heavy promotion beginning 10/89. Prepare for large orders during winter
 Contact rep about new line of tennis rackets.
 Customer with high market share (23%) due to aggressive advertising.
 Tends to order large amounts of merchandise at once. Accounting is considering raising their credit
 limit
 Support intensive. Orders small amounts (< 800) of merchandise at a time.
 First sporting goods store geared exclusively towards women. Unusual promotional style

Tabla PRICE

Name	Null?	Type
PRODID	NOT NULL	NUMERIC(6)
STARTDATE	NOT NULL	DATETIME
ENDDATE		DATETIME
STDPRICE		NUMERIC(8,2)
MINPRICE		NUMERIC(8,2)

```
SELECT * FROM price;
```

PRODID	STDPRICE	MINPRICE	STARTDATE	ENDDATE
100871	4.8	3.2	01-JAN-85	01-DEC-85
100890	58	46.4	01-JAN-85	
100890	54	40.5	01-JUN-84	31-MAY-84
100860	35	28	01-JUN-86	
100860	32	25.6	01-JAN-86	31-MAY-86
100860	30	24	01-JAN-85	31-DEC-85
100861	45	36	01-JUN-86	
100861	42	33.6	01-JAN-86	31-MAY-86
100861	39	31.2	01-JAN-85	31-DEC-85
100870	2.8	2.4	01-JAN-86	
100870	2.4	1.9	01-JAN-85	01-DEC-85
100871	5.6	4.8	01-JAN-86	
101860	24	18	15-FEB-85	
101863	12.5	9.4	15-FEB-85	
102130	3.4	2.8	18-AUG-85	
200376	2.4	1.75	15-NOV-86	
200380	4	3.2	15-NOV-86	

Anexo D

Algunas Instrucciones básicas de SQL

La instrucción CREATE TABLE

Se utiliza la instrucción CREATE TABLE para crear tablas y almacenar datos. Esta inserción pertenece al lenguaje de definición de datos (DDL). Las instrucciones DDL son un conjunto de instrucciones de SQL utilizadas para crear, modificar o eliminar tablas de la base de datos. Estas instrucciones tienen un efecto inmediato en la base de datos.

Para crear una tabla, se debe tener los privilegios para crear objetos. El administrador de la base de datos utiliza las instrucciones del lenguaje de control de datos (DCL), para otorgar privilegios a los usuarios.

Sintaxis:

CREATE TABLE tabla

(columna tipo_de_datos [DEFAULT expr]);

tabla es el nombre de la tabla.

DEFAULT expr Especifica un valor por defecto si el valor es omitido en la instrucción INSERT.

columna es el nombre de la columna

tipo_de_datos es el tipo y longitud de la columna

Objetos de la base de datos

Una base de datos en Microsoft SQL Server puede contener diferentes estructuras de datos. Cada estructura pertenecerá a la BD y pueden ser los siguientes tipos:

1. tabla: almacena datos
2. vista: subconjunto de datos de una o mas tablas
3. índice: para mejorar el rendimiento de las consultas

Objeto	Descripción
tabla	Unidad básica de almacenamiento compuesta de renglones
Vista	Representación lógica de un grupo de datos de una o más tablas
Índice	Para mejorar el desempeño de las consultas

La opción default

A una columna se le puede dar un valor por defecto utilizando la opción DEFAULT. Esta opción previene la inserción de valores NULL, si se inserta un renglón sin un valor indicado para esta columna. El valor por defecto puede ser una literal, una expresión o una función SQL; tales como GETDATE() o USERNAME(). La expresión por defecto debe coincidir con el tipo de dato de la columna.

Por ejemplo: hiredate DATETIME DEFAULT GETDATE()

Creando una tabla

```
CREATE TABLE dept2
    (deptno NUMERIC (2),
    dname VARCHAR(14),
    loc VARCHAR(13));
```

Tipos de datos

Tipo de datos	Descripción
VARCHAR(tamaño)	Carácter de longitud variable
CHAR(tamaño)	Carácter de longitud fija
NUMERIC(p,s)	Datos numéricos
DATETIME	Valores tipo fecha y hora
TEXT	Dato tipo carácter de longitud variable hasta 2 gigabytes
IMAGE	Dato binario con una longitud máxima de hasta 2,147,483,647 bytes

La instrucción ALTER TABLE

Utilizar la instrucción ALTER TABLE para:

- agregar nuevas columnas
- eliminar una columna existente
- definir un valor por defecto para una nueva columna

Después de que se haya creado las tablas, se puede necesitar realizar cambios en las estructuras, en el caso de que le hayan faltado columnas. Se puede hacer esto utilizando la instrucción ALTER TABLE. Se puede agregar columnas utilizando la instrucción ALTER TABLE con la cláusula ADD.

ALTER TABLE table

ADD columna tipo_de_datos [DEFAULT expr]
[, columna tipo_de_datos]... ;

Sintaxis:

tabla es el nombre de la tabla
DEFAULT expr especifica un valor por defecto si el valor es omitido en la instrucción INSERT

columna es el nombre de la columna
tipo_de_datos es el tipo y longitud de la columna

Se puede eliminar columnas existentes en la tabla utilizando la instrucción ALTER TABLE con la cláusula DROP.

ALTER TABLE tabla
DROP COLUMN columna;

ALTER TABLE DEPT2
DROP COLUMN dname

Borrando una tabla

La instrucción DROP TABLE elimina una tabla de la BD. Cuando se elimina una tabla, se pierden todos los datos y los índices asociados con ella.

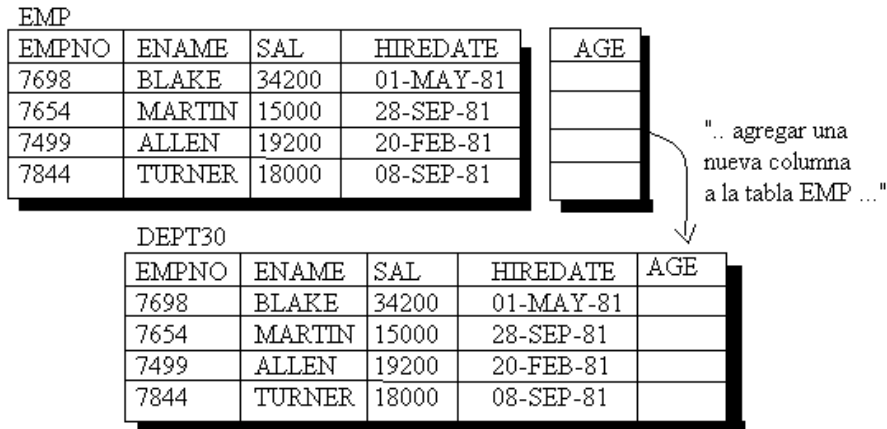
La instrucción DROP TABLE, una vez realizada, es irreversible. Microsoft SQL Server no pregunta si desea confirmar esta acción.

Agregando una columna

Utilizar la cláusula ADD para agregar columnas.

```
ALTER TABLE EMP
ADD AGE VARCHAR(9);
```

La nueva columna será la última de las columnas de la tabla.



El ejemplo agrega una columna llamada AGE a la tabla EMP, esta columna se coloca al final de la tabla.

```
ALTER TABLE EMP
ADD AGE VARCHAR(9);
```

La nueva columna será la última

EMPNO	ENAME	SAL	HIREDATE	AGE
7698	BLAKE	34200	01-MAY-81	
7654	MARTIN	15000	28-SEP-81	
7499	ALLEN	19200	20-FEB-81	
7844	TURNER	18000	08-SEP-81	
...				
6 rows selected.				

Nota: si la tabla ya contiene registros, esta nueva columna será inicializada con NULL en todo los registros existentes.

Reglas para la creación de CONSTRAINTS (restricciones)

Todas las restricciones son almacenadas en el diccionario de datos. Estas serán fáciles de referenciar si se les otorga un nombre adecuado. Para asignar un nombre a una restricción se debe utilizar las reglas estándar para el renombrado de objetos de la BD. Si no se asigna un nombre a la restricción, Microsoft SQL Server le asignará un nombre interno, lo cual puede confundir al momento de obtener errores.

Las restricciones pueden ser definidas al momento en que es creada la tabla o después de haberla creado.

Se pueden definir las restricciones a nivel de columnas o de tablas.

Incluyendo CONSTRAINTS (restricciones)

Los Constraints son restricciones a nivel de tablas.

Los constraints previenen borrados de una tabla si existen dependencias

Los siguientes tipos de constraint están disponibles en Microsoft SQL Server. Estas se usan para prevenir la inserción de datos inválidos en las tablas.

Constraints implícitos

- NOT NULL NN
- UNIQUE Key UK
- PRIMARY KEY PK
- FOREIGN KEY FK

Estos constrains se denominan implícitos, porque la regla de validación ya está dada por el manejador.

Constraints explícitos

- CHECK CK

Este es un constraint explícito debido a que, el usuario es quien debe especificar que tipo de validación se debe llevar a cabo para los valores que formarán el dominio de una columna para este constraint.

Definiendo Constraints

```
CREATE TABLE tabla
    (columna tipo_de_datos [DEFAULT expr]
    [constraint_columna],
    ...
    [constraint_tabla]);
```

```
CREATE TABLE emp (
    empno NUMERIC(4),
    ename VARCHAR(10) NOT NULL,
    comm MONEY,
    ...
    deptno NUMERIC(7,2) NOT NULL,
    CONSTRAINT emp_empno_pk
    PRIMARY KEY (EMPNO));
```

El ejemplo muestra la sintaxis para definir constraints mientras se crea una tabla.

Sintaxis:

tabla	es el nombre de la tabla
DEFAULT expr	especificar un valor por defecto si el valor es omitido en la instrucción INSERT
column	es el nombre de la columna
datatype	es el tipo y la longitud de la columna
column_constraint	es la restricción como parte de la definición de la columna
table_constraint	es la restricción como parte de la tabla

Constraint NOT NULL

El constraint NOT NULL asegura que no se permitan valores nulos en una columna. Las columnas que no tengan esa restricción pueden grabar valores nulos.

EMP					
EMPNO	ENAME	JOB	...	COMM	DEPTNO
7839	KING	PRESIDENT			10
7698	BLAKE	MANAGER			30
7782	CLARK	MANAGER			10
7566	JONES	MANAGER			20
...					

NOT NULL constraint (ningún renglón puede contener valores null en esta columna)

Ausencia del constraint NOT NULL (cualquier renglón puede contener null en esta columna)

NOT NULL constraint

El constraint NOT NULL solo puede ser especificado a nivel columna, no a nivel tabla.

```

CREATE TABLE emp (
  empno NUMERIC(4),
  ename VARCHAR(10) NOT NULL,
  job VARCHAR(9),
  mgr NUMERIC(4),
  hiredate DATETIME,
  sal NUMERIC(7,2),
  comm NUMERIC(7,2),
  deptno NUMERIC(2) NOT NULL);
    
```

El ejemplo aplica el constraint NOT NULL a las columnas ENAME y DEPTNO de la tabla EMP. Debido a que estos constraint no tienen nombre, Microsoft SQL Server les asigna uno.

Se puede especificar el nombre del constraint al momento de crearlo.

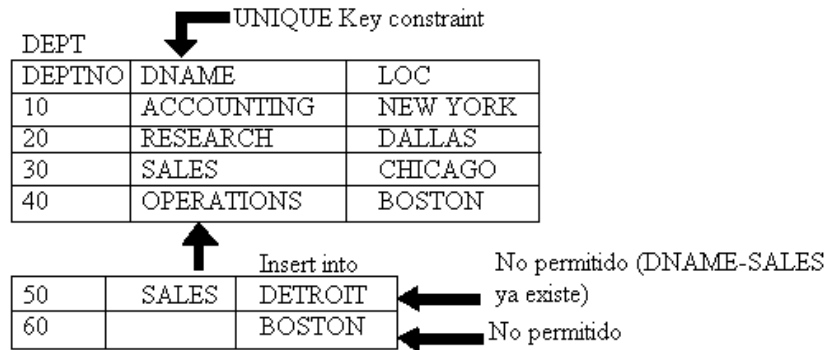
```

... DEPTNO NUMERIC(2),
CONSTRAINT emp_deptno_nn NOT NULL...
    
```

El constraint UNIQUE Key

Un constraint UNIQUE Key requiere que cada valor en la columna o conjunto de columnas sea único, es decir, dos renglones de la misma tabla no pueden tener el valor duplicado en la columna o columnas especificadas como UNIQUE.

El constraint UNIQUE permite la inserción de valores nulos a menos que defina un constraint NOT NULL para la misma columna; de hecho, cualquier renglón puede incluir valores NULL en las columnas debido a que un NULL es considerado igual a otro NULL. Un valor NULL en una columna (o en las columnas compuestas por UNIQUE) siempre satisface la restricción UNIQUE.



El constraint UNIQUE puede ser definido a nivel de columna o a nivel de tabla. Un UNIQUE compuesto por dos o más columnas debe ser creado utilizando la definición a nivel de tabla.

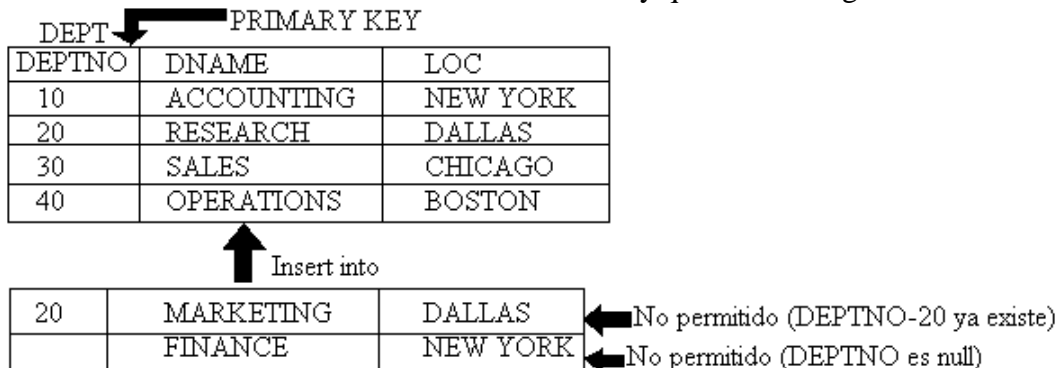
```
CREATE TABLE dept(
    deptno NUMERIC(2),
    dname VARCHAR(14),
    loc VARCHAR(13),
    CONSTRAINT dept_dname_uk UNIQUE(dname));
```

El ejemplo aplica un constraint UNIQUE a la columna de DNAME de la tabla DEPT. El nombre del constraint es dep_dname_uk.

Nótese Microsoft SQL Server crea un índice único al constraint UNIQUE.

El constraint PRIMARY KEY

El constraint PRIMARY KEY crea la llave primaria de la tabla. Sólo puede existir una llave primaria por cada tabla. El constraint PRIMARY KEY es una columna o conjunto de columnas que identifican en forma única a cada registro en la tabla. Este constraint fuerza a que los valores de la columna o columnas sean únicos y que no contengan valores nulos.



El constraint PRIMARY KEY puede ser definido a nivel columna o a nivel tabla. Sin embargo, una llave primaria compuesta solo debe ser creada a nivel de tabla.

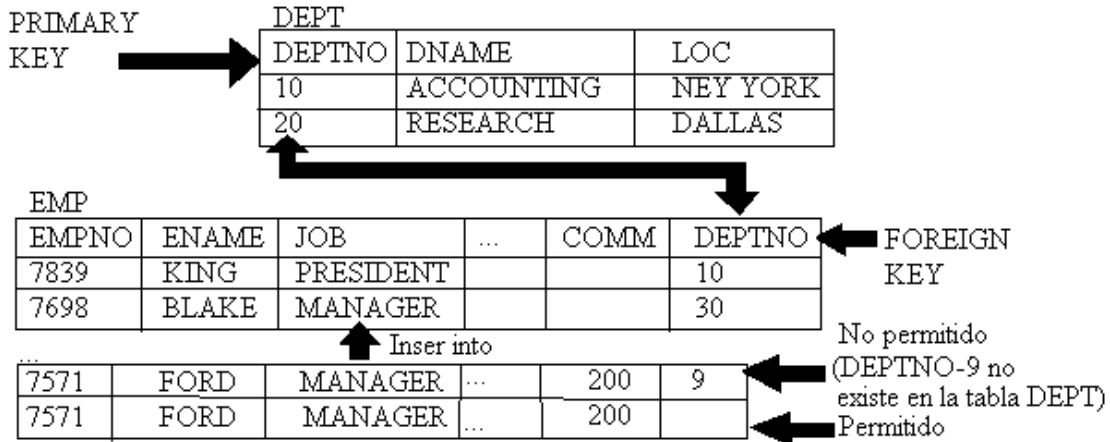
```
CREATE TABLE dept(
    deptno NUMERIC(2),
    dname VARCHAR(14),
    loc VARCHAR(13),
    CONSTRAINT dept_dname_uk UNIQUE(dname),
    CONSTRAINT dept_deptno_pk PRIMARY KEY (deptno));
```

El ejemplo crea una llave primaria en la columna DEPTNO para la tabla DEPT.

Nótese que si un constraint afecta a más de una columna (constraint compuesto) este solo puede ir a nivel de tabla.

El constraint FOREIGN KEY

El constraint FOREIGN KEY o regla de integridad referencial, designa una columna o combinación de columnas como una llave foránea y establece una relación entre una llave primaria en la misma tabla o en una tabla diferente.



En el ejemplo, DEPTNO ha sido definida como la llave foránea en la tabla EMP (tabla dependiente o tabla hija); hace referencia a la columna DEPTNO de la tabla DEPT (tabla padre o tabla referenciada).

El valor de una llave foránea debe coincidir con un valor existente en la tabla padre o ser nulo. Las llaves foráneas están basadas en valores y son apuntadores puramente lógicos, no físicos.

El constraint FOREIGN KEY puede ser definido a nivel de columna o a nivel de tabla. Sin embargo una llave foránea compuesta debe ser creada a nivel de tabla.

```
CREATE TABLE emp(
    empno    NUMERIC(4),
    ename    VARCHAR(10) NOT NULL,
    job      VARCHAR(9),
    mgr      NUMERIC(4),
    hiredate DATETIME,
    sal      NUMERIC(7,2),
    comm     NUMERIC(7,2),
    deptno   NUMERIC(7,2),
    CONSTRAINT emp_deptno_fk FOREIGN KEY(deptno)
    REFERENCES dept(deptno));
```

El ejemplo define una llave foránea en la columna DEPTNO de la tabla EMP. Se puede definir un constraint referencial a nivel de columna de la siguiente forma:

```
CREATE TABLE emp(  
    empno      NUMERIC(4),  
    ename      VARCHAR(10) NOT NULL,  
    job        VARCHAR(9),  
    mgr        NUMERIC(4),  
    hiredate   DATETIME,  
    sal        NUMERIC(7,2),  
    comm       NUMERIC(7,2),  
    deptno     NUMERIC(7,2) CONSTRAINT emp_deptno_fk  
              REFERENCES dept(deptno));
```

Una llave foránea se define en una tabla “hijo” y la tabla que contiene la columna referenciada es denominada tabla “padre”. La llave foránea es definida utilizando una combinación de las siguientes palabras reservadas:

- FOREIGN KEY es utilizada para definir la columna en la tabla “hijo” a nivel de tabla
- REFERENCES identifica a la tabla y columna en la tabla “padre”, se utiliza ya sea a nivel de columna o a nivel de tabla.

El constraint CHECK

El constraint CHECK define una condición que cada renglón debe cumplir. La condición puede ser definida al igual que las condiciones de las consultas.

Una simple columna puede tener múltiples constraint CHECK, no hay límites para el número de constraint CHECK, que se defina para una columna

```
..., deptno NUMERIC(2),  
    CONSTRAINT emp_deptno_ok  
    ...CHECK(DEPTNO BETWEEN 10 AND 99),...
```

Agregando constraints

Se puede agregar constraints a la tabla existente utilizando la excepción ALTER TABLE con la cláusula ADD.

```
ALTER TABLE tabla  
ADD [CONSTRAINT constraint_nombre] tipo (columna);
```

Sintaxis

```
tabla      es el nombre de la tabla  
columna    es el nombre de la columna afectada  
tipo       es el tipo de constraint  
constraint es el nombre del constraint
```

El nombre del constraint es opcional, sin embargo se recomienda asignar uno. Si no se define un nombre al constraint, el sistema generará uno por defecto para ese constraint, el cual al momento de obtener errores, no proporcionará información descriptiva del error.

Ejemplo: agregar un constraint FOREIGN KEY a la tabla empleados indicando que debe existir un manager válido para este empleado en la tabla.

```
ALTER TABLE emp  
ADD CONSTRAINT emp_mgr_fk  
    FOREIGN KEY(mgr) REFERENCES emp(empno);
```


El ejemplo crea un constraint FOREIGN KEY en la tabla EMP. Este constraint asegura que debe existir un manager valido en la tabla EMP para un empleado.

Eliminando un constraint

Para eliminar un constraint se debe identificar el nombre con el que fue asignado, entonces, utilizar la instrucción ALTER TABLE con la cláusula DROP.

Ejemplo: eliminar el constraint de manager de la tabla EMP.

```
ALTER TABLE emp
DROP CONSTRAINT emp_mgr_fk;
```

El lenguaje de manipulación de datos

El lenguaje de manipulación de datos DML es una parte de SQL. Cuando se desea agregar, actualizar o borrar datos en una BD se debe ejecutar instrucciones del DML. Un conjunto de instrucciones DML que forman una unidad lógica de trabajo se llama transacción.

Considérese una BD bancaria, cuando un cliente del banco transfiere dinero de su cuenta de ahorros a una cuenta de cheques, la transacción consistirá en tres operaciones separadas: **decrementar la cuenta de ahorros, incrementar la cuenta de cheques y registrar la transacción en la bitácora de transacciones.** Microsoft SQL Server debe garantizar que estas tres instrucciones SQL sean ejecutadas con éxito para mantener la consistencia de las cuentas.

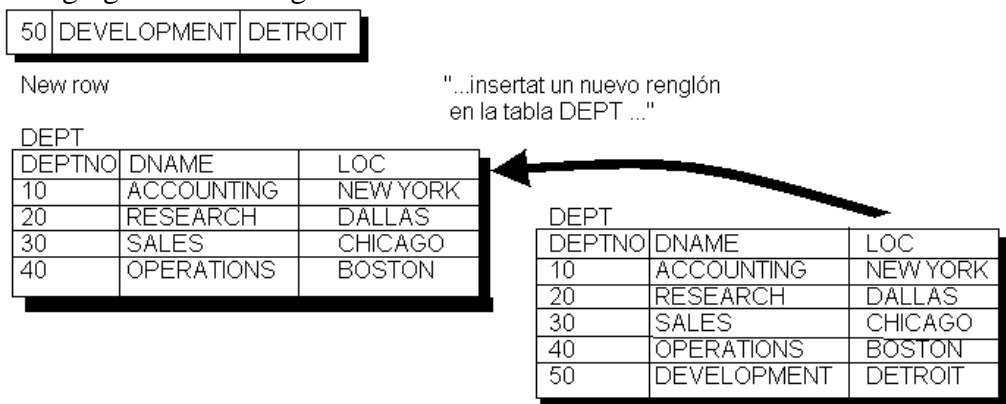
Una instrucción DML es ejecutada cuando:

- Agrega nuevos registros a la tabla
- Modifica los renglones existentes en la tabla
- Elimina renglones existentes de la tabla

Una **transacción** consiste en una colección de instrucciones DML que forman una unidad lógica de trabajo.

La instrucción INSERT

Se puede agregar nuevos renglones a la tabla utilizando la instrucción INSERT.



La sintaxis:

```
INSERT INTO tabla[(columna [, columna...])]
VALUES (valor [,valor...]);
```

Con esta sintaxis solo un renglón a la vez se puede insertar.

Sintaxis:

tabla es el nombre de la tabla

columna es el nombre de la columna

valor es el valor que le corresponderá a la columna

Debido a que se puede insertar un nuevo renglón que contiene valores para cada columna, la lista de columnas no es necesaria en la cláusula INSERT. Sin embargo, sino se utiliza la lista de columnas, los valores deben ser listados de acuerdo al orden por defecto que tienen las columnas en la tabla.

```
INSERT INTO dept (deptno, dname, loc)
VALUES (50, 'DEVELOPMENT', 'DETROIT');
```

Con INSERT es posible:

- Insertar un nuevo renglón conteniendo valores para cada columna
- Listar los valores en el orden por defecto de las columnas en la tabla.
- Listar opcionalmente las columnas en la cláusula INSERT.

Nota: encerrar los caracteres y fechas en comillas simples

Insertando registros con valores NULL

Existen dos métodos importantes:

- El método implícito: omitir a la columna de la lista de columnas.

```
INSERT INTO dept (deptno, dname)
VALUES (60, 'MIS');
```

- El método explícito: especificar la palabra NULL

```
INSERT INTO dept
VALUES (70, 'FINANCE', NULL);
```

Métodos por insertar valores nulos

Método	Descripción
Implícito	Omite la columna de la lista de columnas.
Explícito	Especifica la clave NULL en la lista de valores. Especifica la cadena de vacíos (' ') en la lista de valores; para cadenas de caracteres y datos solamente.

Asegurarse que las columnas permitan almacenar valores nulos verificando NULL? Al ver la estructura de la tabla.

Insertando valores especiales

La función GETDATE() registra la fecha y hora actuales.

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES (7196, 'GREEN', 'SALESMAN', 7782, GETDATE(), 2000, NULL, 10);
```

Se puede utilizar pseudocolumnas para intentar valores especiales en las tablas. El ejemplo graba al empleado GREEN en la tabla. Utiliza la función GETDATE() para obtener la fecha y hora actuales.

Confirmando la inserción en la tabla

```
SELECT empno, ename, job, hiredate, comm
FROM emp
WHERE empno = 7196;
```

EMPNO	ENAME	JOB	HIREDATE	COMM
7196	GREEN	SALESMAN	01-DEC-97	

Cambiando datos a la tabla

El ejemplo actualiza el número del departamento de Clark de 10 a 20.

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
...				

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		20
7566	JONES	MANAGER		20
...				

La instrucción UPDATE

La instrucción UPDATE permite:

- Modificar renglones existentes con la instrucción UPDATE

```
UPDATE tabla
SET column = valor [, column = valor]
[WHERE condición];
```

- Actualizar mas de un renglón a la vez, si se requiere

Sintaxis:

tabla es el nombre de la tabla.

columna es el nombre de la columna.

valor es el valor que le corresponderá a la columna.

condición identifica a los renglones que serán actualizados y se compone por nombres de columnas, expresiones, constantes, subconsultas y operadores de comparación.

Confirmar la operación de actualización ejecutando una consulta a la tabla para mostrar los registros actualizados.

Actualizando renglones en una tabla

La instrucción UPDATE modifica renglones específicos, si se especifica una cláusula WHERE.

```
UPDATE emp
SET deptno = 20
WHERE empno = 7782;
(1 row(s) affected)
```

El ejemplo transfiere el empleado 7782 (Clark) al departamento 20. Si se omite la cláusula WHERE, todos los renglones en la tabla serán modificados.

```
UPDATE employee
SET deptno = 20;
(14 row(s) affected)
```

Actualizando renglones basándose en otra tabla

Se puede utilizar subconsultas en una instrucción UPDATE para actualizar renglones en una tabla.

```
UPDATE employee
SET deptno = (SELECT deptno
              FROM emp
              WHERE empno = 7788)
WHERE job = (SELECT job
             FROM emp
             WHERE empno = 7788);
```

(2 row(s) affected)

El ejemplo actualiza la tabla EMPLOYEE basada en los valores de la EMP. Cambia el número de todos los empleados cuyo puesto sea el mismo que el del empleado 7788 al valor del departamento de este mismo empleado.

Eliminando renglones de una tabla

Se puede eliminar los renglones existentes utilizando la instrucción DELETE.

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT
60	MIS	

"... borrar un renglón de la tabla DEPT ..."

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
60	MIS	

Esta instrucción se utiliza de la siguiente manera:

```
DELETE [FROM] tabla
[WHERE condición];
```

La sintaxis es:

tabla es el nombre de la tabla.

condición identifica a los renglones que serán borrados y se compone por nombres de columnas, expresiones, constantes, subconsultas y operadores de comparación.

Se puede borrar renglones específicos utilizando la cláusula WHERE en la instrucción DELETE.

```
DELETE FROM    departament
WHERE          dname = 'DEVELOPMENT';
```

El ejemplo borra el departamento DEVELOPMENT de la tabla departament.

Otras ocasiones se querrán borrar todos los registros de una tabla si se omite la cláusula WHERE.

```
DELETE FROM    departament;
```

Borrando los renglones basados en otras tablas

Se puede utilizar subconsultas para borrar renglones de una tabla basado en los valores de otra tabla.

```
DELETE FROM employee
WHERE    deptno = (SELECT deptno
                  FROM dept
                  WHERE dname);
```

El ejemplo borra todos los empleados que están en el departamento 30. La subconsulta busca en la tabla DEPT para encontrar el número de departamento que pertenece a SALES. La subconsulta regresa el número de departamento a la consulta principal, el cual borra un registro de la tabla EMPLOYEE basado en este número de departamento.

Bibliografía

1. Andrew Warden. *Adventures in Relationland*. En C. J. Date, *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990). The Naming of Columns.
2. Codd E. F. *A Relational Model of Data for Large Shared Data Banks*. CACM 13, num. 6 (junio de 1970). Publicado en *Milestones of Research – Selected Papers 1958-1982* (CACM, ejemplar del 25 aniversario), CACM 26, num 1 (enero de 1983).
3. Codd, E. F. *More Commentary on Missing Information in Relational Databases (Applicable and Inapplicable Information)*. ACM SIGMOD Record 16 num. 1 (marzo de 1987).
4. Date C. J. *Support for the Conceptual Schema: The Relational and Network Approaches*. En C. J. Date, *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).
5. Date C. J. *Updating Views*. En C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).
6. Elmasri, R. y S. B. Navathe. *Sistemas de Bases de Datos. Conceptos fundamentales*. Tercera edición, Addison Wesley Iberoamericana, 2002, Wilmington, Delaware (USA).
7. Gama M. L. *Introducción a SQL Server* Instituto Tecnológico de Zacatepec. Año de edición del curso 2004.
8. Peter Rob. Carlos Coronel. *Sistemas de Bases de datos. Diseño, implementación y Administración*. Quinta Edición, Editorial Thompson, 2004.
9. <http://www.programatium.com/manuales/sql/modrel004.htm#007>
10. http://macine.epublish.cl/tesis/index-2_3_.html
11. http://www.sql-server-performance.com/nn_triggers.asp
12. <http://msdn2.microsoft.com/es-es/library/ms378371.aspx>
13. <http://www.emagister.com/sql-server-descripcion-del-entorno-creacion-bases-datos-cursos-1055411.htm>
14. <http://www.forosdelweb.com/showthread.php?t=422991>