Unidad No. 3

Sistemas relacionales

Definición de datos

1.1 Definición de datos
1.2 Manipulación de datos
1.3 El catálogo del sistema
1.4 Vistas

3.1 Definición de datos

Esta sección está relacionada con el lenguaje de definición de datos (DDL) de SQL. Se considerara la porción relacional del DDL. Se consideraran únicamente los aspectos de interés para el usuario y no la manera en que se maneja el nivel interno del sistema, el cual es altamente dependiente del sistema específico. Desde el punto de vista del usuario, las principales sentencias de DDL son las siguientes:

CREATE TABLE	CREATE VIEW	CREATE INDEX
ALTER TABLE		
DROP TABLE	DROP VIEW	DROP INDEX

Tabla base

Una tabla base es un caso especial de un concepto más general “tabla”.

Una tabla en un sistema relacional consiste de un fila de encabezados de columna, junto con cero o más filas de valores de datos (diferentes números de filas de datos en diferentes tiempos). Para una tabla dada:
A) La fila de encabezado de la columna especifíca una o más columnas (dando a cada una de ellas, entre otras cosas, un tipo de dato).
B) Cada fila de datos contiene exactamente un valor escalar para cada columna especificada en la fila de encabezados de columna. Además, todos los valores en una columna dada son del mismo tipo de dato, nombrando el tipo de dato especificado en la fila de encabezado para esta columna.

En relación a la definición anterior, surgen dos puntos importantes:
1. No se menciona un ordenamiento para las filas. Estrictamente hablando, las filas de una tabla relaciona son consideradas desordenadas. Las filas de una relación constituyen un conjunto matemático, y los conjuntos en matemáticas no tienen ningún orden. Sin embargo, es posible imponer un orden sobre las filas cuando son recuperadas como respuesta a una consulta, pero este ordenamiento debe ser visto como algo conveniente para el usuario y nada más, no es intrínseco a la noción de tabla en sí.
2. En contraste con el punto 1, las columnas de la tabla son consideradas en orden, de izquierda a derecha. Por ejemplo, en la tabla de proveedores, el id del proveedor es la primera columna, el nombre del proveedor es la segunda columna y así sucesivamente. Sin embargo, en la práctica existen muy pocas ocasiones en las cuales este

ordenamiento de izquierda a derecha es importante, y con un poco de disciplina se puede evitar totalmente estos casos.

Nota:

Ahora bien, las filas y columnas tienen un orden físico en la versión almacenada de la tabla dentro del disco, lo que es más, este ordenamiento físico puede y tiene un gran efecto sobre el rendimiento del sistema. El punto es, sin embargo, que estos ordenamientos físicos son en todas las situaciones, transparentes al usuario.

Finalmente, se puede definir una tabla base como una tabla autónoma nombrada. Por "autónoma", se entiende que la tabla existe (a diferencia una vista, que no existe, pero que se deriva de una o más tablas base). Por "nombrada", se entiende que la tabla posee un nombre, otorgado vía la apropiada sentencia CREATE (a diferencia de una tabla que es construida meramente como el resultado de una consulta, la cual no tiene un nombre explícito y tiene una existencia efímera).

Create Table

El formato general de la sentencia es:

```
CREATE TABLE base-table (
  Column-definition [, column-definition] …
  [primary key definition]
  [foreign-key-definition [, foreign-key definition] … ] );
```

Donde Column-definition toma la forma:

Column data-type [NOT NULL]

La especifiación NOT NULL, así como la definición de llave primaria y llave foránea se explican más adelante. Los corchetes son utilizados como definiciones sintácticas que indican que el material encerrado entre ellos es opcional. Los puntos suspensivos (…) indican que la unidad sintáctica que le precede inmediatamente puede repetirse una o más veces. Las palabras escritas en mayúsculas deben escribirse exactamente como se muestran; las palabras escritas en minúsculas, deben ser reemplazadas por valores específicos elegidos por el usuario.

Tipos de datos

Algunos tipos de datos escalares:

Numéricos

- INTEGER Entero binario completo
- SMALLINT Medio entero binario
- DECIMAL (p,q) número decimal empacado (p dígitos y signo y asume q dígitos decimales a la derecha)
- FLOAT(p) número de punto flotante con p dígitos binaries de precisión.

Datos tipo string

- CHARACTER(n) String de longitud fija de exactamente n caracteres de 8 bits.

- VARCHAR(n) String de longitud variable de un máximo de n caracteres de 8 bits.
- GRAPHIC(n) String de longitud fija de exactamente n caracteres de 16 bits.
- VARGRAPHIC(n) String de longitud variable de un máximo de n caracteres de 16 bits.

Datos tipo fecha/tiempo

- DATE Fecha (yyyymmdd)
- TIME Tiempo (hhmmss
- TIMESTAMP Combinación de fecha y tiempo, con una exactitud de microsegundos.

Información perdida o incompleta

Este es un problema que se encuentra frecuentemente en el mundo real. Por ejemplo, los registros históricos algunas veces incluyen datos como fecha de nacimiento desconocida; las agendas de reunión frecuentemente muestran al expositor como "a ser anunciado". A raíz de esto es deseable tener alguna forma de manejar esta situación en los sistemas formales de bases de datos.

Los DBMS, típicamente representan esta información perdida o incompleta utilizando unas marcas especiales llamadas nulls. Si un registro dado tiene un null en un campo dado, significa que el valor de este campo es desconocido (o quizás no aplica). Es importante resaltar que null NO ES LO MISMO que un espacio en blanco o el valor CERO, de hecho, el null no es realmente un valor de dato en el sentido de este término, razón por la cual se refiere a null como una marca.

Alter table

Una tabla base existente puede ser modificada en cualquier momento agregando columnas a la derecha utilizando la sentencia ALTER TABLE:

ALTER TABLE <base table name> ADD <column-name> <data type>;

Esta instrucción agrega columnas a una tabla base, todos los registros exsitentes son (conceptualmente) expandidos para que posean un campo más; el valor del nuevo campo es NULL en todos los casos, la especificación NOT NULL no es permitida al agregar columnas. Incidentalmente, esta expansión de los registros existentes no es llevada a cabo físicamente al momento de ejecutar la instrucción; todo lo que pasa es que la descripción de los registros cambia.

Existen otros tipos de modificación de tablas, por ejemplo eliminar columnas o modificar el tipo de dato.

Drop table

Permite destruir una tabla base en cualquier momento:

DROP TABLE <base-table name>

La tabla base especificada es eliminada del sistema (más precisamente, la descripción de la tabla es removida del catálogo). Todos los índices y vistas definidos sobre la tabla base son automáticamente eliminados también.

Indexes

Como las tablas base, los índices son creados y eliminados utilizando sentencias de SQL de definición de datos (DDL). CREATE INDEX y DROP INDEX son las únicas sentencias en el lenguaje SQL que hacen referencia a los índices; otras sentencias – particularmente las sentencias de manipulación de datos tales como el SELECT – no incluyen ninguna referencia. La decisión de utilizar o no un índice en particular en respuesta a un requerimiento de datos en una consulta particular recae no sobre el usuario sino sobre el subcomponente optimizador.

Sintaxis para crear índices:

CREATE [UNIQUE] INDEX <index-name>
ON <base-table-name> (<column-name>[order][,<column-name> [order]] …)
[CLUSTER] ;

Cada especificación "order" puede tomar los valores ASC (ascendente) o DESC (descendente), si no se especifica, entonces se assume ASC por default. La secuencia de izquierda a derecha de las columnas indexadas en la sentencia corresponden al orden de uso de mayor a menor. Además, la especificación opcional CLUSTER significa que este es un índice en cluster; una tabla base dada solamente puede tener un índice en cluster. Ejemplo:

CREATE INDEX x ON t (p, q DESC, R) CLUSTER;

La opción UNIQUE en la sentencia CREATE INDEX especifica que no pueden existir 2 registros al mismo tiempo en la tabla base indexada que tengan el mismo valor en el campo o conjunto de campos indexados.

Los índices al igual que las tablas base pueden ser eliminados en cualquier momento. Debe tomar en cuenta que si intenta crear un índice único sobre una tabla cuyos datos violen esta regla, entonces la sentencia fallará.

Sentencia para borrar índices:
DROP INDEX <index-name>;

El índice es eliminado (la descripción es eliminada del catálogo). Si un plan de ejecución para una aplicación existente, depende del índice eliminado, entonces, este plan será automáticamente recreado la próxima vez que la aplicación se ejecute.

3.2 Manipulación de datos

SQL provee cuatro sentencias para manejar datos (DML) – SELECT, UPDATE, DELETE e INSERT –

Queries simples

Por ejemplo, la consulta "obtener los números de proveedor y su estado para los proveedores en París", se expresa en SQL de la siguiente forma:

```
SELECT SID, STATUS
  FROM proveedores
 WHERE CITY = 'PARIS';
```

Resultado:

SID	STATUS
S2	10
S3	30

El ejemplo ilustra la forma común de una sentencia SELECT de SQL – "SELECT especificar los campos FROM especificar las tablas base y vistas WHERE algunas condiciones que deben cumplirse". Lo primero que se debe notar es que el resultado de una consulta (Query) es otra tabla – una tabla que es derivada de cierta forma de las tablas base que existen en la base de datos. En otras palabras, el usuario en un sistema relacional siempre está operando en un framework tabular, una opción muy atractiva en tales sistemas.

También se pueden utilizar campos calificados cuando creamos una consulta:

```
SELECT prv.SID, prv.STATUS
  FROM proveedores prv
 WHERE prv.CITY = 'PARIS';
```

Un campo calificado consiste del nombre de la tabla (o alias de la tabla) y el nombre del campo, separados por una coma. Nunca es un error utilizar campos calificados, y muchas veces es necesario utilizarlos.

Estructura general para una sentencia SELECT simple:

```
SELECT [DISTINCT] lista de ítems
FROM lista de tablas
[WHERE condicion(es) ]
[GROUB BY lista de campos]
[HAVING condiciones]
[ORDER BY lista de campos];
```

La cláusula DISTINCT significa eliminar de la tabla resultado todas las filas duplicadas.

Recuperación de campos calculados

Para todas las piezas, obtener el número de pieza y el peso de la pieza en gramos (los pesos de las piezas en la tabla de PIEZAS está en libras).

```
SELECT pzs.id_pieza, 'Peso en gramos = ', pzs.weight * 460
FROM piezas pzs;
```

Resultado:

id_pieza		
P1	Peso en gramos =	5448
P2	Peso en gramos =	7718
…		

La cláusula SELECT (al igual que las cláusulas WHERE y HAVING) puede incluir expresiones escalares generales, utilizando operadores escalares tales como la suma y resta y funciones escalares tales como SUBSTR (substrings) como simples nombres de campo.

Recuperación simple (SELECT *)

Obtiene todos los campos de una tabla, ejemplo:

```
SELECT *
FROM proveedores;
```

El uso del "*" es muy conveniente para consultas iterativas, porque ahorra tiempo, sin embargo, es potencialmente peligroso en SQL inmerso (SQL dentro de un programa de aplicación), porque el significado del "*" puede cambiar si el programa es recompilado y algunos cambios fueran realizados en la definición de las tablas que componen la consulta.

Recuperación calificada

Obtener los códigos de proveedor para proveedores en París con estado mayor que 20.

```
SELECT prv.SID, prv.STATUS
  FROM proveedores prv
 WHERE prv.CITY = 'PARIS'
AND prv.STATUS > 20;
```

Resultado:	SID	STATUS
	S3	30

Las condiciones que siguen la cláusula WHERE pueden incluir los operadores =, <>, >, >=, < y <=; los operadores boléanos AND, OR y NOT; y paréntesis para indicar un orden específico de evaluación.

Recuperación con orden

Obtener los números de proveedor y el estado para proveedores en París, ordenados por estado descendentemente.

```
SELECT prv.SID, prv.STATUS
  FROM proveedores prv
 WHERE prv.CITY = 'PARIS'
ORDER BY prv.STATUS DESC;
```

Resultado:	SID	STATUS
	S3	30
	S2	10

También es posible identificar columnas en la cláusula ORDER BY utilizando los números de columna, utilizando el orden de izquierda a derecha de la tabla resultante, esta característica permite ordenar el resultado utilizando columnas que no tienen nombre.

JOIN QUERIES

La habilidad de unir dos o más tablas es una de las características más poderosas de los sistemas relacionales. La habilidad de las operaciones de unión, más que cualquier otra cosa, distingue los sistemas relacionales de los no relacionales. Un join es un Quero en el cual los datos son recuperados desde más de una tabla.

Ejemplo de equijoin simple: Obtener todas las combinaciones de proveedores y partes en las cuales tanto la parte como el proveedor están ubicados en la misma ciudad.

```
SELECT s.*, p.*
FROM proveedores s, partes p
WHERE s.ciudad = p.ciudad;
```

El resultado de este Query es un JOIN (unión) de las tablas “proveedores” y “partes” cuyos valores de ciudad son iguales. La condición s.ciudad = p.ciudad se conoce como “JOIN CONDITION”.

No existe un requerimiento para que el operador sea una igualdad en una JOIN CONDITION aunque lo normal es que lo sea. Si el operador es una igualdad, entonces el JOIN se conoce como EQUIJOIN. El equijoin por defiición debe producir un resultado que contenga dos columnas idénticas, sin embargo, si una de estas dos columnas es eliminada del Quero (de los datos seleccionados) entonces es llamado un NATURAL JOIN, el cual es la forma más utilizada y sencilla de JOIN, tanto así que el término JOIN muchas veces se utiliza específicamente como un NATURAL JOIN.

Ejemplo de un JOIN mayor que: Obtener todas las combinaciones de proveedores y partes tal que la ciudad del proveedor siga la ciudad de la parte en orden alfabético.

```
SELECT s.*, p.*
FROM proveedores s, partes p
WHERE s.ciudad > p.ciudad;
```

Ejemplo de un JOIN con una condición adicional: Obtener todas las combinaciones de proveedores y partes donde el proveedor y la parte estén en la misma ciudad, pero no incluir proveedores en estado 20.

```
SELECT s.*, p.*
FROM proveedores s, partes p
WHERE s.ciudad = p.ciudad
AND s.status != 20;
```

La clausula WHERE de un JOIN puede contener condiciones adicionales a las del JOIN en sí mismo.

Ejemplo de un JOIN que recupera campos específicos: Obtener todos los números de proveedor y números de parte tales que el proveedor y la parte estén en la misma ciudad.

```
SELECT s.S# id_proveedor, p.P# id_parte
FROM proveedores s, partes p
WHERE s.ciudad = p.ciudad;
```

Es posible darle un alias a los campos elegidos, en el ejemplo id_proveedor es el alias para el campo S# de la tabla proveedores e id_parte es el alias para el campo P# de la tabla parte.

Ejemplo: Unir tres tablas: Obtener todos los pares de nombres de ciudades tales que el proveedor localizado en la primera ciudad provea la parte almacenada en la segunda ciudad. Por ejemplo, el proveedor S1 provee partes P1ñ el proveedor S1 está localizado en Londres, y la parte P1 está almacenada en Londres, por lo que el par (Londres, Londres) debe estar en el resultado.

```
SELECT DISTINCT s.ciudad, p.ciudad
FROM proveedores s, partes p, partes_abastecidas sp
WHERE s.S# = sp.S#
AND p.P# = sp.P#;
```

No existe un límite en el número de tablas que pueden ser unidas. Tome en cuenta que el uso de DISTINCT es para eliminar las parejas duplicadas.

Ejemplo: Uniendo una tabla consigo misma: Obtener todos los pares de números de proveedores tales que los dos proveedores involucrados estén en la misma ciudad.

```
SELECT s1.S#, s2.S#
FROM proveedores s1, proveedores S2
WHERE s1.ciudad = s2.ciudad
AND s1.S# < s2.S#;
```

Note que el propósito de la condición s1.S# < s2.S# es para que (a) eliminar las parejas de números de proveedor de la forma (x,x); (b) garantizar que las parejas (x,y) y (y,x) no aparezcan juntas.

Funciones de agregación

SQL provee un número de funciones especiales de agregación para aumentar el poder básico de recuperación. Las funciones provistas son: COUNT, SUM, AVG, MAX y MIN. Aparte del caso especial de "COUNT(*)", cada una de estas funciones opera sobre un conjunto de valores en una columna de la misma tabla. Los valores resultantes se definen a continuación:

COUNT	--	Cantidad de valores en la columna
SUM	--	Suma de los valores en la columna
AVG	--	Promedia los valores en la columna
MAX	--	Valor mayor en la columna
MIN	--	Valor mínimo en la columna

Para las funciones SUM y AVG la columna debe contener valores numéricos. En general, el argumento de la función puede opcionalmente ser precedido por la palabra reservada DISTINCT, para indicar que los valores duplicados no deben ser tomados en cuenta en la aplicación de la función. Para las funciones MAX y MIN, sin embargo, la palabra reservada DISTINCT es irrelevante y no tienen efecto. Para COUNT, se puede especificar DISTINCT; siempre y cuando se especifique una columna y en este caso cuenta todas las filas sin incluir filas duplicadas en esta columna. (la cláusula COUNT(*) DISTINCT no es permitida).

Cualquier valor NULL en los argumentos de las columnas siempre son eliminados antes de aplicar las funciones. Si el resultado de estas funciones es un conjunto vacío, COUNT retorna el valor CERO, sin embargo, las otras funciones retornan NULL.

Ejemplo, utilizando funciones de agregación en la cláusula SELECT. Obtener el número total de proveedores:

```
SELECT COUNT(*)
FROM  proveedores;
```

Ejemplo: utilizando funciones de agregación en la cláusula SELECT con DISTINCT: Obtener el número total de proveedores que actualmente proveen partes:

```
SELECT COUNT(DISTINCT S#)
FROM partes_abastecidas;
```

Ejemplo: Funciones de agregación en la cláusula SELECT con condición: Obtener la cantidad de registros que pertenecen al proveedor P3.

```
SELECT COUNT(*)
FROM partes_abastecidas
WHERE P# = 'P3';
```

Ejemplo: Funciones de agregación en la cláusula SELECT con condición: Obtener la sumatoria total de partes abastecidas por el proveedor P3.

```
SELECT SUM(Qty)
FROM partes_abastecidas
WHERE P# = 'P3';
```

Ejemplo: Uso de GROUP BY: Obtener la sumatoria total de partes abastecidas por cada parte, es decir, por cada parte abastecida, obtener el número de partes y la cantidad total provista para esta parte.

```
SELECT P#, SUM(Qty)
FROM partes_abastecidas
GROUP BY P#;
```

El operador GROU BY arregla lógicamente la tabla representada por la cláusula FROM en particiones o grupos, de tal forma que dentro de cada grupo todos los registros tengan el mismo valor para los campos contenidos en el GROUP BY. La cláusula GROUP BY no implica un ORDER BY, para garantizar un resultado ordenado debe utilizar la cláusula ORDER BY.

Ejemplo: Uso de HAVING: Obtener los números de parte para todas las partes abastecidas por más de un proveedor.

```
SELECT P#
FROM partes_abastecidas
GROUP BY P#
HAVING COUNT(*)  > 1;
```

La cláusula HAVING es para los grupos lo que la cláusula WHERE es para las filas (registros), si la cláusula HAVING es especificada, la cláusula GROUP BY debe ser especificada también). En otras palabras, HAVING es utilizado para eliminar grupos tal como el WHERE es utilizado para eliminar filas (registros).

Características avanzadas

Ejemplo de consulta utilizando LIKE: Obtener todas las partes cuyos nombres empiezan con la letra C.

```
SELECT p.*
FROM proveedores p
WHERE p.nombre LIKE '%C';
```

En general la condición LIKE toma la forma "columna LIKE valor-string" donde la columna debe ser de tipo string. Para un registro, la condición es evaluada como verdadera si el valor dentro de la columna corresponde al formato especificado por el valor-string. Los caracteres dentro del valor-string se interprestan como sigue:

- El carácter "_" (underscore) se sustituye por cualquier carácter simple.
- El carácter "%" (porcentaje) se sustituye por cualquier secuencia de n caracteres (donde n puede ser cero).
- Otros caracteres se representan a sí mismos.

Ejemplo de recuperación de valores NULL. Una sintaxis especial es provista para evaluar la presencia o ausencia de marcas NULL.

```
SELECT S#
FROM proveedores s
WHERE estado IS NULL;
```

La forma general de esta forma de evaluación es: column-name IS [NOT] NULL.

Ejemplo de recuperación involucrando un subquery: Obtener los nombres de proveedores que abastecen la parte P3.

```
SELECT nombre
FROM proveedores s
WHERE S# IN (
        SELECT S#
        FROM partes_abastecidas sp
        WHERE p# = 'P3');
```

Un subquery es una sentencia SELECT – FROM –WHERE- GROUP BY- HAVING que está anidada dentro de otra expresión del mismo tipo. Los subqueries son típicamente utilizados para representar un conjunto de valores que son buscados vía una condición IN. El sistema, conceptualmente, evalúa primero los queries más anidados y luego los queries externos.

Ejemplo de subquery con múltiples niveles de anidamiento: Obtener los nombres de proveedores que proveen por lo menos una parte de color rojo.

```
SELECT nombre
FROM proveedores p
WHERE S# IN (
        SELECT S#
        FROM partes_abastecidas sp
        WHERE P# IN (
                SELECT P#
                FROM partes p
                WHERE color = 'ROJO'
        )
);
```

Los subqueries pueden ser anidados a cualquier nivel.

Ejemplo de subquery utilizando un operador de comparación distinto de IN.: obtener los números de proveedor que están localizados en la misma ciudad que el proveedor S1.

```
SELECT S#
FROM proveedores
WHERE ciudad = (
        SELECT ciudad
        FROM proveedores
        WHERE S# = 'S1');
```

Ejemplo de queries utilizando EXISTS: obtener los nombres de proveedor que abastezcan la parte P3.

```
SELECT nombre
FROM proveedores p
WHERE EXISTS (
        SELECT *
        FROM partes_abastecidas sp
        WHERE sp.S# = p.S#
        AND sp.P# = 'P3');
```

Ejemplo de queries involucrando la cláusula UNION: obtener los números de parte cuyo peso sea mayor a 16 libras o que son abastecidas por el proveedor S2.

```
SELECT P#
FROM partes p
WHERE peso > 16
UNION
SELECT P#
FROM partes_abastecidas
WHERE S# = 'S2';
```

La cláusula UNION es el operador tradicional utilizado en la unión de conjuntos. En otras palabras, la unión de los conjuntos A y B es el conjunto de todos los bojetos x tales que x es un miembro de A o x es un miembro de B (o de ambos). Todo elemento duplicado (filas duplicadas) son siempre eliminadas del resultado de una cláusula UNION.

Cláusula START WITH and CONNECT BY

Esta cláusula puede ser utilizada para consultar data que tiene relaciones jerárquicas (usualmente relaciones padre/hijo u objeto/parte). También son utilizadas cuando un plan de ejecución de una consulta es desplegado.

Las condiciones recursivas pueden hacer uso de la palabra clave PRIOR:

CONNECT BY PRIOR child = parent

Esta estructura establece la recursión. Todos los registros que son parte del siguiente nivel jerárquico cumplen la condición parent = child, en otras palabras child es un valor que se encuentra en el nivel actual de la jerarquía.

Ejemplo:

```
Table test_connect_by (
  Parent number,
  Child number  /* posee una llave única */
);

5 = 2+3
insert into test_connect_by values ( 5, 2);
insert into test_connect_by values ( 5, 3);
18 = 11+7
insert into test_connect_by values (18,11);
insert into test_connect_by values (18, 7);
17 = 9+8
insert into test_connect_by values (17, 9);
insert into test_connect_by values (17, 8);
```

26 = 13+1+12
insert into test_connect_by values (26,13);
insert into test_connect_by values (26, 1);
insert into test_connect_by values (26,12);
15=10+5
insert into test_connect_by values (15,10);
insert into test_connect_by values (15, 5);
38=15+17+6
insert into test_connect_by values (38,15);
insert into test_connect_by values (38,17);
insert into test_connect_by values (38, 6);
38, 26 y 18 no tienen padre (el padre es null)
insert into test_connect_by values (null, 38);
insert into test_connect_by values (null, 26);
insert into test_connect_by values (null, 18);

```
SELECT LPAD(' ',2*(level-1)) || TO_CHAR(child) s
  FROM test_connect_by
  START WITH parent is null
  CONNECT BY PRIOR child = parent;
```

Resultado:

```
38
  15
    10
    5
      2
      3
  17
    9
    8
  6
26
  13
  1
  12
18
  11
  7
```

Ejemplo: Podar ramas

```
TABLE prune_test (
  parent  number,
  child   number
);
```

A continuación se llena una estructura de árbol, cada hijo es el número del padre más un nuevo dígito a la derecha:

```
insert into prune_test values (null,   1);
insert into prune_test values (null,   6);
insert into prune_test values (null,   7);
```

```
insert into prune_test values (   1,  12);
insert into prune_test values (   1,  14);
insert into prune_test values (   1,  15);

insert into prune_test values (   6,  61);
insert into prune_test values (   6,  63);
insert into prune_test values (   6,  65);
insert into prune_test values (   6,  69);

insert into prune_test values (   7,  71);
insert into prune_test values (   7,  74);

insert into prune_test values (  12, 120);
insert into prune_test values (  12, 124);
insert into prune_test values (  12, 127);

insert into prune_test values (  65, 653);

insert into prune_test values (  71, 712);
insert into prune_test values (  71, 713);
insert into prune_test values (  71, 715);

insert into prune_test values (  74, 744);
insert into prune_test values (  74, 746);
insert into prune_test values (  74, 748);

insert into prune_test values ( 712,7122);
insert into prune_test values ( 712,7125);
insert into prune_test values ( 712,7127);

insert into prune_test values ( 748,7481);
insert into prune_test values ( 748,7483);
insert into prune_test values ( 748,7487);
```

Ahora, recuperaremos el árbol, pero podaremos todo debajo de las reamas 1 y 71. Esto no lo podemos hacer colocando una condición dentro de la cláusula WHERE de la sentencia de SQL, debido a que pertenece la cláusula CONNECT BY.

```
SELECT LPAD(' ', 2*level) || child
FROM  prune_test
START WITH  parent IS null
CONNECT BY PRIOR  prior child=parent
  AND parent not in (1, 71);
```

Resultado

```
1
  6
    61
    63
    65
      653
```

```
    69
  7
    71
    74
      744
      746
      748
        7481
        7483
        7487
```

Cómo saber si 2 items esten dentro de los ancestros de la relación de descendencia
Algunas veces queremos saber si dos ítems se encuentran dentro de los ancestros de la relación de descendencia, esto es si XYZ es padre, abuelo, tatarabuelo, … de ABC. El siguiente ejemplo nos puede ayudar a resolver esto.
TABLE parent_child(parent_ varchar2(20), child_ varchar2(20));

```
insert into parent_child values (null,  'a')

insert into parent_child values (  'a',  'af');
insert into parent_child values (  'a',  'ab');
insert into parent_child values (  'a',  'ax');

insert into parent_child values ( 'ab', 'abc');
insert into parent_child values ( 'ab', 'abd');
insert into parent_child values ( 'ab', 'abe');

insert into parent_child values ('abe','abes');
insert into parent_child values ('abe','abet');

insert into parent_child values ( null,   'b');

insert into parent_child values (  'b',  'bg');
insert into parent_child values (  'b',  'bh');
insert into parent_child values (  'b',  'bi');

insert into parent_child values ( 'bi', 'biq');
insert into parent_child values ( 'bi', 'biv');
insert into parent_child values ( 'bi', 'biw');
```

El siguiente query pregunta por un pariente y un supuesto hijo (nieto, tataranieto) y responde la pregunta si los encuentra en una relación de ancestros sucesores.

```
SELECT
  CASE WHEN COUNT(*) > 0 THEN
    '&&parent is an ancestor of &&child' ELSE
    '&&parent is no ancestor of &&child' END
    "And here's the answer"
FROM parent_child
WHERE child_ = '&&child'
START WITH parent_ = '&&parent'
CONNECT BY PRIOR child_ = parent_;
```

undefine child
undefine parent

En conclusion:
select ... start with initial-condition connect by nocycle recurse-condition

select ... connect by recurse-condition

Función SYS_CONNECT_BY_PATH
Se tiene la siguiente tabla que contiene una jerarquía simple:
SQL> DESC test_table;

Name	Null?	Type
A		NUMBER(38)
B		NUMBER(38)

Con los siguientes datos:

SQL> SELECT * FROM test_table;

A	B
1	-1
2	1
3	1
4	2
5	2
6	4
7	4
8	5
9	5
10	3
11	3

11 rows selected.

Se require una consulta que muestre cada nodo y todos sus antecesores (padres).
Ejemplo:

A	B
...	
9	9
9	5
9	2
9	1

...
Esto se muestra porque 9 se relaciona con 9, 9 se relaciona con 5 (de la tabla original), 9 indirectamente se relaciona con 2 (por 5), y así sucesivamente. Cómo se puede hacer esto en un query?

Se puede obtener la jerarquía entera de la siguiente forma:

```
SELECT a
  FROM test_table
  CONNECT BY PRIOR b=a
```

Este query obtiene la columna B en la salida deseada. Ahora si desearamos mostrar el nodo raíz de cada fila en esta jerarquía, se puede utilizar la función SYS_CONNECT_BY_PATH, y se podría obtener cada nodo raíz. Ejemplo:

```
SELECT a,
  SYS_CONNECT_BY_PATH (a,'.') scbp
  FROM test_table
  CONNECT BY PRIOR b=a
   ORDER BY 2
```

```
        A    SCBP
----------    ---------
...
        9    .9
        5    .9.5
        2    .9.5.2
        1    .9.5.2.1

...
```

Este Query empieza a tomar la forma de la salida deseada. El inicio de cada valor de la columna SCBP es la raíz de la jerarquía, y la columna A es el extremo final. Ahora lo completamos de la siguiente forma:

```
SELECT TO_NUMBER(
                  SUBSTR(scbp,1, INSTR(scbp,'.')-1)
                                ) b, a
   FROM (
     SELECT a,
       LTRIM(SYS_CONNECT_BY_PATH(a,'.'),'.') ||'.' scbp
    FROM test_table
   CONNECT BY PRIOR b=a
        )
   ORDER BY 2
```

Resultado:

```
        A            B
-----------    ----------
...
        9            9
        9            5
        9            2
        9            1

...
```

Otras características de los queries jerárquicos:

- CONNECT_BY_ROOT—Función que retorna la raíz de la jerar´quia en la fila actual.
- CONNECT_BY_ISLEAF—Es una bandera que indica que la fila actual tiene filas hijas
- CONNECT_BY_ISCYCLE—Es una bandera que indica si la fila actual es el inicio de un ciclo infinito en la jerarquía. Por ejemplo si A es padre de B, B es padre de C y C es padre de A, se tendría un ciclo infinito. Esta bandera se puede utilizar para ver qué fila o filas son el inicio de los ciclos infinitos en los datos.
- NOCYCLE—Permite que la cláusula CONNECT BY reconozca un ciclo infinito y se detenga sin error (en lugar de retornar un error de CICLO EN CONNECT BY)

La consulta anterior se podría reescribir de la siguiente forma:

```
SELECT CONNECT_BY_ROOT a cbr,
       a b
   FROM test_table
  CONNECT BY PRIOR b=a
  ORDER BY 1


      CBR          B
-----------    ----------
...
         9          9
         9          5
         9          2
         9          1

...
```

3.3 El catálogo del sistema

El catálogo es un sistema de base de datos que contiene información (algunas veces llamada “descriptores”) concerniente a objetos de interés para el sistema en sí mismo. Ejemplos de estos objetos son tablas base, vistas, índices, usuarios, planes de aplicación de queries, privilegios de acceso, etc. La información de descripción es esencial para que el sistema realice su trabajo correctamente. Por ejemplo, el componente optimizador de enlaces utiliza la información que el catálogo contiene acerca de índices (así como otra información) para elegir una estrategia específica de acceso. Otro ejemplo, el subsistema de autorización utiliza la información del catálogo acerca de usuarios y privilegios de acceso para conceder o denegar solicitudes del usuario.

Una de las mejores ventajas de los sistemas relacionales de bases de datos (DBMS), consiste en que el catálogo mismo consiste de relaciones (o tablas), como las tablas ordinarias de datos del usuario. Es de notar, que un sistema particular necesariamente contendrá una gran cantidad de información que es específica al sistema.

Este capítulo brinda una introducción a la estructura y contenido de un catálogo típico, y da una idea de cómo la información en este catálogo puede ayudar al usuario tanto como al sistema. Utilizaremos una forma simplificada del catálogo que contendrá las siguientes tablas base de catálogo:

- SysTables
  Esta tabla base del catálogo contiene una fila por cada tabla nombrada (tabla base o vista) en el sistema entero. Para cada tabla, se tendrá un nombre (NAME), el nombre del usuario que es dueño de la tabla (CREATOR), el número de columnas en la tabla (COLCOUNT), y muchos otros ítems de información.
- SysColumns
  Esta tabla base del catálogo contiene una fila por cada columna de cada tabla mencionada en la tabla base SYSTABLES. Por cada columna, se tendrá el nombre de la columna (NAME), el nombre de la tabla a la cual pertenece la columna (TBNAME), el tipo de dato de la columna (COLTYPE), y muchos otros campos.
- SysIndexes
  Esta tabla base del catálogo contiene una fila por cada índice en el sistema. Para cada uno de éstos índices se tendrá el nombre (NAME), el nombre de la tabla base indexada (TBNAME), el nombre del usuario que es dueño del índice (CREATOR), y así sucesivamente.

Consultando el catálogo

Debido a que el catálogo está formado por tablas, al igual que las tablas de los usuarios ordinarios, se puede consultar, esto es, utilizar sentencias SELECT de SQL como se hace con las tablas ordinarias.

Un usuario que no es familiar con la estructura de la base de datos, puede utilizar queries sobre las tablas del catálogo para descubrir la estructura de la base de datos. Por ejemplo, un usuario que desea consultar la base de datos de proveedores y partes, pero que no tiene el conocimiento detallado de las tablas que existen en esta base de datos, ni de las columnas exactas que contienen estas tablas, puede utilizar queries sobre el catálogo para obtener el conocimiento necesario antes de formular los queries sobre los datos.

Note que, en un sistema tradicional no relacional, los queries iniciales debían ser direccionados al diccionario del sistema en lugar de a la base de datos. La diferencia más importante entre los métodos relacionales y los métodos no relacionales es que en los sistemas de bases de datos relacionales, el catálogo y la base de datos son consultados utilizando la misma interfase, normalmente utilizando el propio lenguaje de consultas del sistema. En los sistemas no relacionales, por contraste, el diccionario y la base de datos siempre son distintos y deben ser utilizados a través de diferentes interfases.

Actualizando el catálogo

Ya se ha visto como el catálogo puede ser consultado utilizando la sentencia SELECT. Sin embargo, el catálogo no puede ser actualizado utilizando instrucciones UPDATE, DELETE e INSERT (y el sistema debe rechazar cualquier intento de hacer esto). La razón, radica en que al permitir tales operaciones podría ser potencialmente muy peligroso, por ejemplo, se podría destruir información muy importante en el catálogo de tal forma que el sistema no podría trabajar correctamente y afectaría el contenido de información de la base de datos del usuario.

Para actualizar el catálogo, existen las sentencias de definición de datos (DDL como CREATE TABLE, CREATE INDEX, etc.) que realizan estos INSERTS o UPDATES en el catálogo.

Nota: El catálogo también incluye entradas para las tablas del catálogo mismas. Sin embargo, estas entradas no son creadas explícitamente con operaciones CREATE TABLE, sino, son creadas automáticamente por el sistema mismo como parte del procedimiento de instalación del sistema (es decir, son incluidas como "hard code" dentro del sistema). En cada DBMS existe un usuario, como por ejemplo SYS, SYSIBM, SYSTEM, que es usado para designar al sistema en sí mismo; es decir, es el creador para las tablas del catálogo en sí.

La sentencia "COMMENT"

Como ya se vio antes, la sentencia UPDATE no puede atizarse directamente sobre el catálogo, sin embargo, la sentencia especial COMMENT, realiza un tipo de actualización directa sobre el catálogo. Las tablas del catálogo SYSTABLES y SYSCOLUMNS incluyen una columna llamada REMARKS, la cual puede ser utilizada (en cualquier registro particular) para contener un string que describe el objeto identificado por el resto de datos de esta columna. La sentencia COMMENT es utilizada para ingresar descripciones dentro de la columna REMARKS en estas tablas (SYSTABLES y SYSCOLUMNS). Los siguientes ejemplos ilustran el formato de la sentencia COMMENT:

COMMENT ON TABLE s IS 'Cada registro representa un proveedor';
COMMENT ON COLUMN p.city IS 'Ubicación de la bodega que almacena esta parte';

Los strings especificados son almacenados en los campos REMARKS para la tabla y campo especificados en las filas correspondientes para SYSTABLES y SYSCOLUMNS.

3.4 Vistas

Introducción

En el nivel externo de la arquitectura ANSI/SPARC, la base de datos es percibida como una "vista externa", definida por un esquema externo. Usuarios diferentes pueden tener diferentes vistas externas. Esto ya se había visto anteriormente, sin embargo, el términos "VISTA" está reservado en SQL (y en los sistemas relacionales en general) para instanciar el concepto de una tabla virtual, nombrada y derivada; el equivalente en SQL de una vista externa ANSI/SPARC es (normalmente) una colección de tablas, algunas de ellas vistas y otras tablas base en el sentido de SQL.

Ahora, el framework ANSI/SPARC es muy general y permite una variabilidad arbitraria entre el nivel externo y el conceptual. En principio, los tipos de datos soportados en ambos niveles pueden ser diferentes; por ejemplo, el nivel conceptual puede estar basado en relaciones, mientras que el usuario percibe una vista externa de la base de datos como una jerarquía. En la práctica, sin embargo, la mayoría de sistemas utilizan el mismo tipo de estructura como base para ambos niveles, y los sistemas de SQL no son la excepción a esta regla, una vista es todavía una tabla, como una tabla base. Y como el mismo tipo de objeto es soportado en ambos niveles, el mismo lenguaje de manipulación aplicará para ambos niveles también (SQL y DML).

Como se indicó anteriormente, una vista es una tabla virtual (una tabla que no existe físicamente, pero el usuario la percibe como si existiera). Por contraste, una tabla base es una tabla real, en el sentido que, para cada fila en la tabla base, existe una contraparte almacenada de esta fila en el almacenamiento físico. Las vistas no son soportadas por sus dueños (propietarios o creadores), están físicamente separadas y

distinguidas de los datos almacenados. En cambio, su definición en términos de otras tablas está almacenada en el catálogo (actualmente en una tabla base del catálogo llamada SYSVIEWS).

Ejemplo:

```
CREATE VIEW good_suppliers
        AS SELECT s#, status, city
        FROM s
        WHERE status > 15;
```

Cuando esta instrucción es ejecutada, el subquery que sigue la palabra reservada AS no se ejecuta; sino, simplemente se almacena en el catálogo. Para el usuario, sin embargo, es como si ahora tuviera una tabla en la base de datos llamada "GOOD_SUPPLIERS", con filas y columnas que cumplen la condición dada.

GOOD_SUPPLIERS es en efecto una "ventana" dentro de la tabla real S. Esta ventana es dinamica, es decir, los cambios realizados a S serán automática e instantáneamente vistos en la ventana; así mismo, los cambios realizados a GOOD_SUPPLIERS serán automática e instantáneamente apicados a la tabla real S, y por supuesto serán visibles en la ventana misma.

Ahora, dependiendo de los sofisticado del usuario, el usuario puede o no notar que GOOD_SUPPLIERS realmente es una vista; algunos usuarios pueden notar este hecho y pueden entender esto es la tabla real S "disfrazada", otros podrían creer que GOOD_SUPPLIERS es una tabla "real". El punto real es, los usuarios pueden operar sobre GOOD_SUPPLIERS como si fuera una tabla real (con algunas excepciones, que se discutirán después).

Las instrucciones SELECT, INSERT, UPDATE y DELETE se aplican a las vistas de la misma forma que a las tablas base.

Definición de vistas

La sintaxis general de la sentencia de SQL CREATE VIEW es la siguiente:

```
CREATE VIEW nombre_vista [ ( column_name, column_name, …) ]
        AS subquery
        [ WITH CHECK OPTION ] ;
```

Note la forma en que la sentencia CREATE VIEW combina la función del esquema externo (la "vista" y la lista opcional de columnas describen el objeto externo) y la función de mapeo externo/conceptual (la parte del subquery especifica el mapeo de este objeto hacia el nivel conceptual).

En principio, cualquier tabla derivada (cualquier tabla que puede ser generada vía una sentencia SELECT), puede teóricamente ser definida como una vista. En la práctica, sin embargo, esta aseveración no es 100% cierta en todos los DBMS.

Ejemplo:

```
CREATE VIEW redparts ( p#, pname, wt, city )
  AS SELECT p#, pname, weight, city
        FROM p
     WHERE UPPER(color) = 'RED';
```

El efecto de esta sentencia es crear una vista llamada REDPARTS, con cuatro columnas llamadas p#, pname, wt y city basadas en la tabla base "P". Si los nombres de columna no son explícitamente especificadas en el CREATE VIEW, entonces la vista hereda los nombres de columna del subquery que la origina. Los nombres de columnas

deben ser especificados en forma explícita (para todas las columnas de la vista) si (a) alguna columna de la vista es derivada de una función, una expresión , o un valor fijo, o (b) si dos o más columnas de la vista llegarían a tener el mismo nombre si se heredan los nombres del subquery que la forma.

Ejemplo:

```
CREATE VIEW (p#, totqty)
  AS SELECT p#, SUM (qty)
       FROM sp
       GROUP BY p# ;
```

En este ejemplo, no hay un nombre que se pueda heredar para la segunda columna, ya que la columna es derivada de una función; por lo tanto, los nombres de columna deben ser especificados explícitamente. Note que esta vista no es solo un simple subconjunto de filas y columnas de la tabla base (como en la vista REDPARTS), sino, más bien es un tipo de resumen estadístico de la tabla base.

Ejemplo:

```
CREATE VIEW city_pairs (scity, pcity)
  AS SELECT DISTINCT s.city, p.city
         FROM s, sp, p
       WHERE s.s# = sp.s#
          AND sp.p# = p.p#;
```

El objetivo de esta vista es que un para de nombres de ciudad (x,y) aparecerá en la vista si un proveedor ubicado en la ciudad “x” provee partes almacenadas en la ciudad “y”. Note que la definición de la vista involucra un join, de tal modo, que este ejemplo muestra una vista que tiene 2 tablas base como origen de su información. Note también que los nombres de las columnas deben ser especificados explícitamente, porque de otro modo las dos columna de la vista tendrían el mismo nombre.

Ejemplo:

```
CREATE VIEW london_redparts
  AS SELECT p#, wt
       FROM redparts
     WHERE UPPER(city)  = ‘LONDON’;
```

Debido a que la definición de una vista permite cualquier subquery válido, y debido a que un subquery puede obtener información de una vista de la misma forma que una tabla base, es perfectamente posible definir una vista en términos de otras vistas.

Ejemplo:

```
CREATE VIEW good_suppliers
  AS SELECT s#, status, city
       FROM s
     WHERE status > 15
  WHITH CHECK OPTION;
```

La cláusula “with check option” indica que las operaciones de UPDATE e INSERT que se realicen a través de la vista deben ser verificadas para asegurar que cada

registro que se actualice o inserte satisfaga la condición de la definición de la vista (status > 15).

Sentencia DROP VIEW

DROP VIEW view_name;

Ejemplo:

DROP VIEW redparts;

Operaciones DML sobre vistas

Como ya se analizó anteriormente, las instrucciones SELECT funcionan de la misma forma en tablas base que en vistas. Sin embargo, las instrucciones UPDATE, DELETE e INSERT, pueden variar, en algunos casos NO es permitido utilizarlas sobre algunas vistas, es decir, no todas las vistas son actualizables (refiriéndose tanto al INSERT y DELETE como al UPDATE).

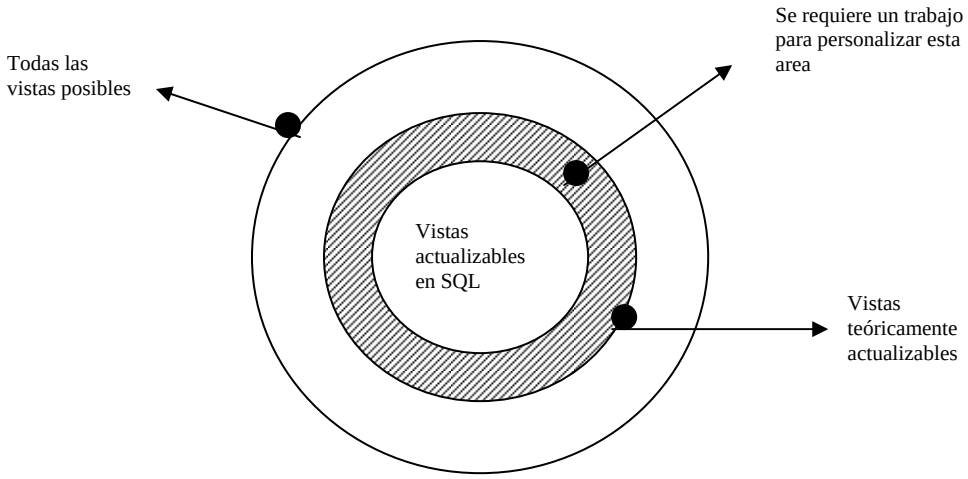
No se pueden realizar INSERTS en una vista cuando:

- La vista no incluye todos los campos NOT NULL. En este caso una instrucción INSERT sobre la vista, intentará poner NULL campos que por definición son NOT NULL, lo cual generará un error.
- Cuando la vista incluye funciones o expresiones para calcular las columnas.

No pueden realizarse DELETES ni UPDATE en una vista cuando:

- La vista no incluye la llave primaria de la tabla base, debido a que no podrá localizar los registros que necesita actualizar.
- Cuando la vista incluye funciones o expresiones para calcular las columnas.

Debido a lo anteriormente expuesto, se puede concluir que existen vistas actualizables y vistas no actualizables, debido a que no todas las vistas pueden actualizarse sin ayuda de un usuario humano que valide la operación que se realiza, es posible clasificar las vistas como se muestra en el diagrama siguiente:



Clasificación de vistas

Como el diagrama muestra, existen ciertas vistas que son teóricamente actualizables, pero que no son actualizables en algunos DBMS. El problema es, que aunque sabemos que la vista existe, no sabemos con exactitud lo que la vista es; esto es también un problema de investigación para determinar exactamente que es lo que tal vista representa. Por esta razón la mayoría de productos (DBMS) soportan actualización de vistas que son subconjuntos de filas y columnas de una o más tablas base.

Nota: El hecho de que no todas las vistas son actualizables es frecuentemente expresada de la siguiente forma: “No se puede actualizar un JOIN”. Esta afirmación no describe completamente la situación ni el problema: Existen algunas vistas que no son JOINS y que no son actualizables, y también existen vistas que son JOINS y que son teóricamente actualizables (aunque talvez la mayoría de productos no puedan actualizarla). Sin embarco, es cierto que los JOINS representan el “caso de interés”, en el sentido de que sería muy conveniente permitir actualizar ciertas vistas cuya definición envuelve un JOIN.

Independencia lógica de datos

Ya se ha visto lo que es una vista, pero para qué son las vistas? Una de las cosas para las cuales existen la vistas es para lo que se llama “Independencia lógica de datos”, es llamada así para distinguirla de la independencia física de datos. Un sistema cualquiera (DBMS) debe proveer independencia física de datos porque los usuarios y los programas de los usuarios no dependen de la estructura física de la base de datos almacenada. Un sistema se dice que provee independencia lógica de datos si los usuarios y los programas de los usuarios son también independientes de la estructura lógica de la base de datos. Existen 2 aspectos para este último tipo de independencia, el crecimiento y la reestructuración.

Crecimiento

Así como la base de datos crece para incorporar nuevos tipos de información, la definición de la base de datos debe crecer también. Existen 2 tipos de crecimiento que pueden ocurrir:

1. La expansión de una tabla base existente para incluir un nuevo campo (correspondiente a la adición de nueva información concerniente a un objeto ya existente) por ejemplo, agregar el campo "descuento" a la tabla base de proveedores.
2. La inclusión de una nueva tabla base (correspondiente a agregar un nuevo tipo de objeto, por ejemplo la información de la proyección en la base de datos de proveedores y partes).

Ninguno de estos tipos de cambio debe tener algún efecto sobre los usuarios existentes.

Reestructuración

Ocasionalmente, es necesario reestructurar la base de datos de tal forma que, el contenido de información contenido represente lo mismo, pero la colocación de la información dentro de la base de datos cambia (por ejemplo, el orden de almacenamiento de los campos es alterado de alguna forma). Antes de continuar, debe hacerse énfasis en que tales reestructuraciones no son deseables, pero en algunos casos son inevitables. Por ejemplo, es necesario partir verticalmente una tabla, de tal forma que los campos comúnmente requeridos sean almacenados en un dispositivo de almacenamiento más rápido y los campos requeridos con menos frecuencia son almacenados en un dispositivo de almacenamiento más lento.

Ventajas de las vistas

- Proveen cierta cantidad de independencia lógica de datos en la fase de reestructuración de la base de datos.
- Permiten que la misma información sea vista por diferentes usuarios en diferentes formas (al mismo tiempo). Esta consideración es muy importante cuando se tienen muchas categorías diferentes de usuarios interactuando con una base de datos integrada.
- La percepción del usuario es simplificada.
- Seguridad automática es provista para los datos ocultos. Los datos ocultos se refieren a los datos que no son vistos a través de una vista en particular.