

# SQL ESTÁNDAR Básico



# SQL ESTÁNDAR Básico

**Autor:**  
Carlos Tomás Reyes García



## UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

### **Rector**

Juan Ramón de la Fuente

### **Secretario General**

Enrique del Val Blanco

### **Director General de Servicios de Cómputo Académico**

Alejandro Pisanty Baruch

### **Directora de Cómputo para la Docencia**

Carmen Bravo Chaveste

## **Guías y Textos de Cómputo. SQL ESTÁNDAR Básico**

### **Coordinación de la publicación**

María Guadalupe Izquierdo Dyrzo

### **Autor**

Carlos Tomás Reyes García

### **Revisión técnica**

Gilberto López Rodríguez

### **Corrección de estilo**

Martha Patricia García Morales

Lucero Ledón Martínez

### **Diseño editorial**

Gabriela Lilí Morales Naranjo

### **Diseño de portada**

Rodolfo Cano Ramírez

### **Editor**

DR © Universidad Nacional Autónoma de México

Ciudad Universitaria, Coyoacán, CP. 04510, México, DF.



## **DIRECCIÓN GENERAL DE SERVICIOS DE CÓMPUTO ACADÉMICO**

Circuito exterior s/n, Ciudad Universitaria, Coyoacán, CP. 04510, México DF.

Prohibida la reproducción total o parcial por cualquier medio  
sin la autorización escrita del titular de los derechos patrimoniales

1ª. edición 2007

ISBN 978-970-32-5098-1

Impreso y hecho en México

## Contenido

---

|                                                                              |    |
|------------------------------------------------------------------------------|----|
| 1. Introducción .....                                                        | 1  |
| 1.1 Antecedentes .....                                                       | 1  |
| 1.2 Importancia .....                                                        | 3  |
| 1.3 Componentes (DDL, DML, DCL) .....                                        | 4  |
| 2. Definición de datos .....                                                 | 5  |
| 2.1 Tipos de datos del sistema y tipos de datos creados por el usuario ..... | 6  |
| 2.1.1 Tipos de datos numéricos exactos .....                                 | 6  |
| 2.1.2 Tipos de datos numéricos aproximados .....                             | 7  |
| 2.1.3 Tipos de datos carácter .....                                          | 8  |
| 2.1.4 Booleans (Booleanos) .....                                             | 8  |
| 2.1.5 DateTime (Fecha - Hora) .....                                          | 8  |
| 2.1.6 Tipos de datos creados por el usuario .....                            | 9  |
| 2.2 Tablas .....                                                             | 9  |
| 2.3 Creación de tablas .....                                                 | 10 |
| 2.4 Creación de reglas para datos por omisión .....                          | 16 |
| 2.4.1 NULL / NOT NULL .....                                                  | 16 |
| 2.4.2 DEFAULT .....                                                          | 17 |
| 2.4.3 CHECK.....                                                             | 18 |
| 2.4.6 AUTO_INCREMENT .....                                                   | 19 |
| 2.5 Reglas de integridad referencial.....                                    | 20 |
| 2.6 Reglas múltiples .....                                                   | 23 |
| 2.7 Modificación de tablas .....                                             | 23 |
| 2.8 Eliminación de tablas.....                                               | 28 |
| 2.9 Creación de índices.....                                                 | 28 |

|                                                                  |    |
|------------------------------------------------------------------|----|
| 3. Manipulación de datos.....                                    | 30 |
| 3.1 Inserción de información.....                                | 30 |
| 3.2 Eliminación de registros .....                               | 35 |
| 3.3 Actualización de información .....                           | 36 |
| 4. Selección de datos.....                                       | 38 |
| 4.1 Selección de información .....                               | 38 |
| 4.2 Especificación de criterios de selección de información..... | 41 |
| 4.3 Operadores SQL.....                                          | 45 |
| 4.3.1 Operadores para el manejo de datos tipo numérico.....      | 46 |
| 4.3.2 Operadores para el manejo de datos tipo carácter.....      | 48 |
| 4.3.3 Operador LIKE y comodines .....                            | 50 |
| 4.3.4 Operadores para el manejo de datos tipo Fecha .....        | 53 |
| 4.3.5 Operador BETWEEN.....                                      | 56 |
| 4.3.6 Operador IN.....                                           | 57 |
| 4.4 Valores nulos .....                                          | 59 |
| 4.5 Ordenamiento de la información.....                          | 60 |
| 4.6 Alias y operadores matemáticos .....                         | 65 |
| 4.7 Subconsultas.....                                            | 68 |
| 4.7.1 Cláusula IN .....                                          | 69 |
| 4.7.2 Cláusula ANY .....                                         | 72 |
| 4.7.3 Cláusula ALL.....                                          | 72 |
| 4.7.4 Cláusula EXISTS .....                                      | 73 |
| 5. Funciones de utilidad .....                                   | 74 |
| 5.1 Funciones para datos tipo carácter .....                     | 74 |
| 5.1.1 SUBSTRING .....                                            | 75 |
| 5.1.2 UPPER .....                                                | 75 |
| 5.1.3 LOWER .....                                                | 75 |
| 5.1.4 TRIM .....                                                 | 76 |
| 5.2 Funciones matemáticas.....                                   | 76 |
| 5.2.1 SUM.....                                                   | 77 |
| 5.2.2 COUNT .....                                                | 77 |
| 5.2.3 AVERAGE .....                                              | 77 |
| 5.3.4 MIN .....                                                  | 77 |
| 5.3.5 MAX.....                                                   | 78 |
| 5.3.6 POWER .....                                                | 78 |
| 5.3.7 SQRT .....                                                 | 78 |
| 5.3.8 ABS.....                                                   | 78 |

|                                                             |     |
|-------------------------------------------------------------|-----|
| 5.3.9 MOD .....                                             | 79  |
| 5.4 Funciones para datos tipo Fecha .....                   | 79  |
| 6. Tipos de JOINS.....                                      | 80  |
| 6.1 Inner Join .....                                        | 81  |
| 6.2 Outer Joins (Right Join – Left Join). ....              | 86  |
| 6.3 Self Join .....                                         | 91  |
| 6.4 Agrupación de información y funciones de agregado ..... | 92  |
| 7. Vistas .....                                             | 100 |
| Anexo A. Instalación de MySQL en Windows XP .....           | 104 |
| Cliente de MySQL .....                                      | 114 |
| Anexo B. Tipos de tablas de MySQL .....                     | 115 |
| Bibliografía .....                                          | 117 |



# CAPÍTULO 1

## Introducción

---

El Lenguaje Estructurado de Consultas (SQL) es el estándar creado para la ejecución de peticiones en una Base de Datos Relacional. Es un lenguaje compacto pero poderoso, utilizado en los Sistemas Manejadores de Bases de Datos Relacionales (SMBR) para la administración de los datos.

El SQL es en sí un lenguaje estructurado claramente definido, que posee una sintaxis para la elaboración de peticiones entendibles desde el usuario hacia un sistema de base de datos relacional.

Así pues, para la comprensión del Lenguaje Estructurado de Consultas es necesario tener bien fundamentada la teoría del Modelo Relacional de Bases de Datos propuesto por Edgar Codd en 1970 en el artículo *“A Relational Model of Data for Large Shared Data Banks”*.

### 1.1 Antecedentes<sup>1</sup>

Cuando Edgar Codd propone el modelo de bases de datos relacional en 1970, publica un conjunto de reglas a las cuales deben sujetarse dicho modelo, es posible destacar de forma muy particular la regla número 5, que se refiere a: *“Un solo lenguaje debe ser utilizado para la comunicación entre el Sistema Manejador de Base de Datos y el*

---

<sup>1</sup> VAN DER LANS, RICK F. Introduction to SQL: Mastering the Relational Data Base Language. Parte 1: “Introducción” Capítulo 2: “Introduction To SQL” Addison Wesley Professional. Estados Unidos de Norte América. 2006.

usuario.”. El modelo relacional de bases de datos tiene como fundamento la Lógica matemática y Teoría de conjuntos, lo que obligó a generar un lenguaje formal para establecer dicha comunicación, de esta manera, Edgar Codd define dos lenguajes matemáticos principalmente para la extracción de información de una base de datos relacional, éstos son el álgebra y cálculo relacional, los cuales constan de un conjunto de operadores lógicos que se aplican sobre las relaciones para así obtener una relación-resultado con la información requerida.

Los trabajos del Dr. Edgar Codd fueron llevados a cabo en los laboratorios de *IBM Research* en San José California (EUA); IBM mostró gran interés en las publicaciones del Dr. Codd y en su modelo relacional, hecho que los lleva a generar un Sistema Manejador de Base de Datos Relacional (RDBMS por sus siglas en inglés), y un lenguaje para la administración de peticiones entre el RDBMS y el usuario; este lenguaje fue llamado SEQUEL (Structured English Query Language) y más tarde implementado por el RDBMS experimental de IBM, el “System R” desarrollado en 1977. Uno de los problemas a los cuales se enfrentó el System R. de IBM, era que no existían lenguajes para la comunicación entre el RDBMS y el usuario, por tal motivo se crea SEQUEL, desarrollado por R. F. Boyce y D. D. Chamberlin.

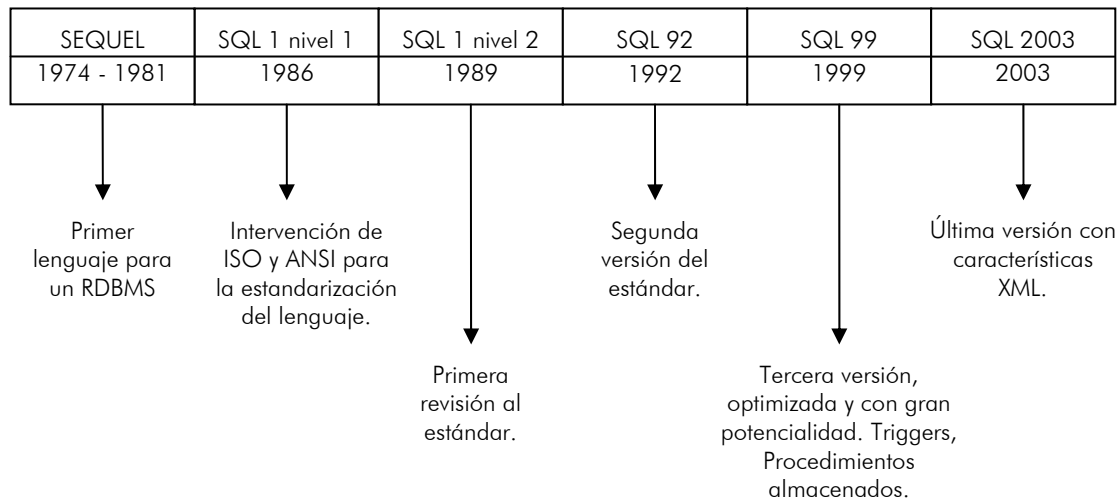
El System R. fue desarrollado en tres fases con tal de lograr una implementación funcional del SEQUEL, dichas fases abarcan de 1974 a 1981, donde madura el SEQUEL, que para ese entonces ya era llamado SQL debido a que SEQUEL representaba una marca comercial en los Estados Unidos.

Durante el periodo comprendido entre 1974 y 1981 fueron diversos los RDBMS que aparecieron en el mercado, cada uno por supuesto, implementaba el modelo relacional; sin embargo, el lenguaje utilizado para la comunicación con el usuario comenzaba a tener variantes de acuerdo con el fabricante. Para eliminar las diferencias entre el lenguaje utilizado por los diversos fabricantes respecto a la comunicación entre el RDBMS y el usuario, se decide estandarizar el lenguaje. El objetivo principal era homologar conceptos y madurar a los RDBMS con el objeto de eliminar las barreras entre un manejador y otro, y así lograr migración de datos entre manejadores relacionales.

En 1983, la Organización Internacional para la Estandarización, ISO por sus siglas en inglés, en conjunto con el Instituto Nacional Americano para la Estandarización, ANSI por sus siglas en inglés, comenzaron con los trabajos para la estandarización del SQL. Luego de tres años de trabajo en 1986, fue lanzada la primera versión de SQL, llamada ANSI SQL nivel 1, es una versión limitada de lo que conocemos hoy en día, en tanto de esa primera versión de SQL se han hecho diversas revisiones que han incrementado la potencialidad del lenguaje. La primera revisión del estándar fue realizada en 1986, más tarde en 1989 lo que da como resultado el ANSI SQL nivel 2, en 1992 aparece la versión 2 de SQL llamada ANSI SQL 92 que en realidad seguía siendo una revisión más extensa del SQL original.

En 1999, aparece el ANSI SQL 99 en donde ya hay la implementación de *triggers*, procedimientos almacenados, consultas recursivas y entre los mejores alicientes se encontraba la implementación de herencia, similar a los lenguajes de programación orientada a objetos. Esta versión de SQL es sin duda una versión con gran madurez, potencialidad y funcionalidad para el trabajo en RDBMS.

La versión más reciente de SQL es el ANSI SQL 2003, versión incrementada de ANSI SQL 99, ya que en esta nueva versión se implementan características del lenguaje XML, eXtensible Markup Language por sus siglas en inglés.



Evolución del Structured Query Language.

## 1.2 Importancia

La importancia de contar con un lenguaje estandarizado para el trabajo dentro de un Sistema Manejador de Base de Datos Relacional recae en que un solo lenguaje es utilizado para la comunicación entre el RDBMS y el usuario, como bien ya lo decía Codd en la regla 5 del Modelo Relacional de Bases de Datos. SQL al ser un lenguaje estándar, tiene la misma potencialidad en cualquier RDBMS independientemente del fabricante.

La mayor importancia del Lenguaje Estructurado de Consultas es que permite cumplir con los objetivos que persigue un sistema de base de datos, en tanto, los objetivos principales de un sistema de base de datos es disminuir los siguientes aspectos:

- *Redundancia.* Información que se encuentra repetida de forma innecesaria en la base de datos.
- *Inconsistencia en los datos.* Falta de concordancia entre datos que se supone son iguales.
- *Problemas de integridad.* Verificar cumplimiento de reglas de negocio y formato de ingreso de datos.
- *Aislamiento de los datos y anomalías del acceso concurrente.* Evitar inconsistencias por actualizaciones de usuarios que acceden al mismo tiempo a la base de datos.
- *Problemas de seguridad.* La información que se guarda en una base de datos no debe ser vista con la misma profundidad por todos los usuarios de la misma. Existen niveles de usuarios y restricciones para consultar la información.

### 1.3 Componentes (DDL, DML, DCL).

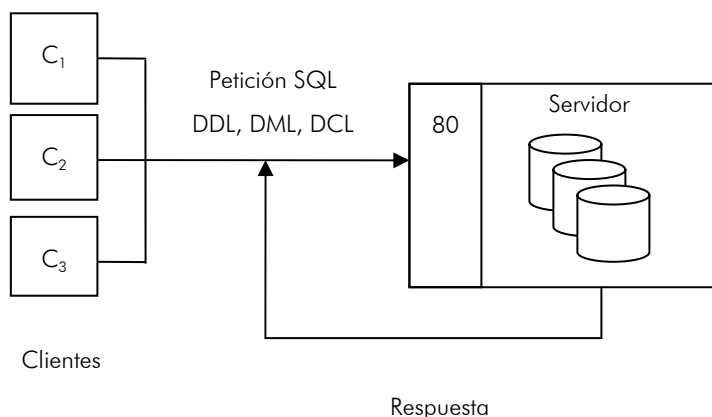
El Lenguaje Estructurado de Consultas SQL se encuentra dividido a su vez, en tres lenguajes de acuerdo con la funcionalidad que representan, dichos lenguajes son:

- *DDL*. Data Definition Language o Lenguaje para la Definición de Datos.
- *DML*. Data Manipulation Language o Lenguaje para la Manipulación de Datos.
- *DCL*. Data Control Language o Lenguaje para el Control de Datos.

El Lenguaje para la Definición de los Datos (DDL) consiste en un conjunto de instrucciones SQL encaminadas a la creación, alteración y destrucción de las estructuras en donde se almacenarán los datos en una base de datos relacional.

El Lenguaje para la Manipulación de Datos (DML) consiste en un conjunto de instrucciones SQL encaminadas a la inserción, actualización y eliminación de los datos en una base de datos relacional.

El Lenguaje para el Control de Datos (DCL) consiste en un conjunto de instrucciones SQL encaminadas a otorgar y revocar permisos a los usuarios sobre los datos y/o las estructuras donde éstos son almacenados.



Interacción Usuario – SQL – Base de Datos.

Conforme se haga el avance en capítulos siguientes, conoceremos a detalle cuáles son los comandos involucrados en cada uno de los sublenguajes SQL. Así mismo, es pertinente señalar que si bien SQL es un estándar de ANSI, pueden existir algunas diferencias, aunque mínimas, respecto de la implementación que hace cada manejador de base de datos relacional u objeto relacional, es decir, pueden existir pequeñas variaciones respecto a la sintaxis SQL. A su vez los RDBM implementan un conjunto de comandos para ejecutar tareas específicas dentro del manejador de base de datos, dichos comandos son independientes del estándar ANSI SQL y reflejan características particulares en los RDBMS.

En los capítulos siguientes, se estudiará la sintaxis del lenguaje SQL; la implementación de los ejercicios que aparecerán en este libro se harán, tomando como base el Manejador de Base de Datos MySQL en su versión liberada 5.0.37, y por lo antes ya señalado, existe la posibilidad de que la implementación en otro RDBMS varíe de acuerdo con algunas de las instrucciones, ya que cada RDBMS puede implementar versiones distintas del estándar SQL, así como la posibilidad de incorporar funciones propias.

# CAPÍTULO 2

## Definición de datos

---

El Lenguaje para la Definición de los Datos (DDL) es la parte de SQL que se utiliza para crear, alterar o destruir los elementos base de una base de datos relacional. Dentro de los elementos base se incluyen: tablas, vistas, esquemas o bases de datos, catálogos, entre otros<sup>2</sup>.

La tabla siguiente muestra comandos parte del Lenguaje para la Definición de Datos:

| Comando         | Utilidad                                                                     |
|-----------------|------------------------------------------------------------------------------|
| CREATE DATABASE | Permite la creación de una base de datos.                                    |
| CREATE TABLE    | Permite la creación de una tabla.                                            |
| CREATE VIEW     | Permite la creación de una vista.                                            |
| ALTER DATABASE  | Utilizado para cambiar o alterar una base de datos.                          |
| ALTER TABLE     | Sirve para modificar la estructura de una tabla o una columna en particular. |
| ALTER VIEW      | Permite alterar la consulta contenida en una vista.                          |
| DROP DATABASE   | Elimina una base de datos.                                                   |
| DROP TABLE      | Elimina una tabla.                                                           |
| DROP VIEW       | Elimina una vista.                                                           |

Este capítulo, por tanto, se avocará a determinar las instrucciones SQL necesarias para la creación, alteración y destrucción de los elementos base de una base de datos relacional, se interactuará con el manejador de base de datos MySQL y se utilizará un caso de estudio base para la aplicación práctica de SQL.

---

<sup>2</sup> TAYLOR, ALLEN G. SQL for Dummies. Indianapolis, Wiley Publishing, Inc. EUA 2006. pp 26 – 45.

En el modelo relacional de base de datos propuesto por Edgar Codd, la estructura persistente es la relación compuesta por un conjunto de tuplas y atributos que cuentan con un encabezado y un cuerpo; el encabezado de una relación es un conjunto de pares de {nombre de atributo, tipo de atributo}, mientras que el cuerpo es el conjunto de pares formado por {nombre de atributo, valor}. La implementación que utiliza una RDBMS para cumplir con el concepto de relación del modelo relacional es la tabla (estructura persistente), la cual está conformada por campos y registros, éstos a su vez, cuentan con un encabezado y un cuerpo: el encabezado es definido por un conjunto de instrucciones DDL mientras que el cuerpo de una tabla es definido a través de instrucciones DML.

Antes de comenzar con el Lenguaje de Definición de Datos (DDL), revisaremos el concepto de *Tipo de Datos*, para llevar a cabo la creación del encabezado de una tabla.

## **2.1 Tipos de datos del sistema y tipos de datos creados por el usuario**

Cuando se crean las tablas de una base de datos, es necesario realizar la especificación del tipo de dato para cada columna con que contará dicha tabla, es decir, se debe especificar qué tipo de información va a contener cada columna de la tabla. De acuerdo con la versión de ANSI SQL que se implemente en el RDBMS por utilizar, pueden existir diversos tipos de datos; para la especificación ANSI SQL 2003, existen seis tipos generales de datos predefinidos:

- Numéricos.
- Carácter.
- Boleanos.
- Fecha/Hora.
- Intervalos.
- XML.

Debido a que son tipos de datos generales, se subdividen generando tipos de datos específicos, y la implementación de éstos depende del RDBMS por utilizar y de la especificación ANSI SQL.

Así pues, definiremos de forma genérica los tipos de datos más utilizados en los RDBMS para la definición de columnas de una tabla en una base de datos relacional.

### **2.1.1 Tipos de datos numéricos exactos**

Los tipos de datos numéricos exactos como su nombre lo indica, permiten expresar un valor numérico cuyo valor es exacto, es decir, para el conjunto de números de los cuales se conoce el formato con el cual se dará ingreso o salida a éstos.

- **Integer (Entero)**

El tipo de dato Integer no posee una parte fraccional tal como su nombre lo indica, es utilizado para valores numéricos enteros cuyo rango de valores válidos depende de la especificación SQL que implemente el RDBMS.

- **SmallInt (Entero corto)**

A diferencia del tipo de dato Integer, SmallInt cuenta con un dominio menor, es decir, el conjunto de valores que son aceptados por este tipo de datos es menor respecto a los aceptados por un entero puro. Por ejemplo, un dato de tipo Integer puede ser de 32-bits de longitud, mientras que, un tipo de dato SmallInt, puede poseer una longitud de 16-bits. En algún RDBMS puede darse el caso de que sean de igual longitud.

- **BigInt**

Similar al tipo de dato Integer, sólo que con un dominio mayor, es decir, el conjunto de valores que son aceptados por este tipo de datos es mayor respecto a los aceptados por un entero puro. Por ejemplo, un dato de tipo Integer puede ser de 32-bits de longitud, mientras que un tipo de dato BigInt puede poseer una longitud de 64-bits.

- **Numeric (Numérico)**

El tipo de dato Numeric posee un componente fraccional adicional a la parte entera que maneja. Este tipo de dato permite establecer la precisión y escala del dato numérico. La precisión recordémoslo, es el número máximo de dígitos con que contará un dato. La escala, refiere al número de dígitos de la parte fraccional.

- **Decimal (Decimal)**

El tipo de dato Decimal posee, igual forma que el tipo de dato Numeric, una parte fraccional y una parte entera, donde es permisible especificar la precisión y escala. La diferencia radica en que el tipo de dato Decimal de acuerdo con la implementación, puede permitir especificar una precisión y escala mayores.

### **2.1.2 Tipos de datos numéricos aproximados**

A veces se desea almacenar datos numéricos cuyo formato no ha de estar especificado o que no se conoce cuál será, ya que dichos datos pueden ser producto de un cálculo matemático; en este caso, se utilizan los siguientes tipos de datos.

- **Real (Reales)**

Este tipo de dato tiene como principal característica que la escala es determinada dinámicamente al introducir el dato, es decir, el punto se encontrará flotante de acuerdo con la especificación realizada por el usuario en la inserción del dato. También es conveniente identificar el hardware por utilizar, ya que la precisión puede verse afectada al utilizar hardware de 32 o 64-bits.

- **Double (Doble)**

El tipo de dato Double ofrece una mayor precisión y escala de acuerdo con el tipo de dato *real*, una vez más, hay que mencionar que todos estos cambios dependen de la implementación SQL por utilizar.

- **Float (Flotante)**

El tipo de dato Flotante es el que tiene la mayor flexibilidad de los tipos de datos numéricos aproximados, ya que permite especificar cuál será la precisión numérica con

que contará. Es el tipo de dato más común utilizado en los procesos de migración de datos entre diversos RDBMS o versiones de los mismos RDBMS.

### **2.1.3 Tipos de datos carácter**

En una base de datos, siempre se desea registrar información que no viene precisamente expresada en números, por lo cual es pertinente realizar descripciones, observaciones, etc. Para ello, existen los tipos de dato Carácter, los cuales nos van a permitir introducir texto en una base de datos.

- **Char (Carácter)**

El tipo de dato Carácter obliga a especificar cuál será la longitud de la cadena por utilizar, es decir, se introducirán hasta tantos caracteres sea posible de acuerdo con la especificación realizada en la definición de la columna. Hay que ser muy cuidadoso cuando se emplea este tipo de dato, ya que si se especifica un dato de tipo Carácter de longitud 10, entonces siempre se almacenarán los bytes correspondientes a 10 caracteres, aun cuando se haya introducido sólo 1.

- **VarChar (Caracter variable)**

El tipo de dato Varchar obliga a especificar cuál será la longitud de la cadena por utilizar, al igual que con un dato de tipo Char, pero con la diferencia de que el número de bytes que son almacenados físicamente depende de la cantidad de caracteres introducidos, es decir, en caso de que la longitud sea mayor, ésta se ajusta al tamaño de la cadena especificada.

- **Text (Texto)**

Dependiendo de la implementación que realice cada RDBMS, el tipo de datos Text permite almacenar grandes cantidades de texto, la longitud es variante de acuerdo con el texto ingresado.

### **2.1.4 Booleans (Boleanos)**

Este tipo de datos permite almacenar valores expresado en 0 y 1, es decir, valores de verdadero y falso.

### **2.1.5 DateTime (Fecha - Hora)**

Para el manejo de datos históricos son utilizados los tipos de datos fecha y hora, con ellos la funcionalidad histórica de una base de datos se encuentra habilitada. Son muy frecuentes y su correcta utilización y formato dependen del RDBMS.

- **Date (Fecha)**

El tipo de dato Fecha permite almacenar los valores para día – mes – año de un dato histórico.

- **Time (Hora)**

Un tipo de dato Hora tiene la capacidad de almacenar los valores para las horas – minutos – segundos.

- **TimeStamp (FechaHora)**

El tipo de dato Timestamp permite almacenar los valores tanto para fecha como para hora, es un campo que implica ambos valores con rangos determinados por el RDBMS.

### 2.1.6 Tipos de datos creados por el usuario

Un tipo de dato creado por el usuario contiene la especificación de un tipo de dato del sistema con un nombre particular, para facilitar al usuario la definición de estructuras persistentes en la base de datos. Por ejemplo, suponga que desea crear un tipo de dato monetario personalizado, entonces utilizará la instrucción SQL `CREATE DISTINCT TYPE`, para especificar un tipo de dato de usuario, seguido de un nombre y un tipo de dato estándar, quedando como instrucción final:

```
CREATE DISTINCT TYPE pesosMexicanos AS DECIMAL(10,2);
```

Dicha instrucción SQL crea un tipo de dato llamado `pesosMexicanos`, el cual podrá ser utilizado en la definición de una tabla por ejemplo.

## 2.2 Tablas

Una tabla es la estructura persistente dentro de un RDBMS, ya que en ella se almacenarán los datos provenientes del exterior, está compuesta por renglones y columnas como se muestra a continuación.

| EDAD | NOMBRE | FECHA    |
|------|--------|----------|
| 18   | JUAN   | 10/03/02 |
| 16   | ANA    | 17/06/00 |

Tabla de alumnos.

Las tablas son la implementación que se realiza de una relación utilizada en el modelo relacional, por tanto cuenta con las características de encabezado y cuerpo. En particular, el encabezado de esta tabla sería:

Encabezado= {(‘EDAD’, integer), (‘NOMBRE’, varchar(15), (‘Fecha’, date)}

y el cuerpo:

Cuerpo= {[ (‘EDAD’, 18), (‘NOMBRE’, ‘JUAN’, (‘Fecha’, ‘10/03/02’)), [ (‘EDAD’, 16), (‘NOMBRE’, ‘ANA’, (‘Fecha’, ‘17/06/00’)) ] }

Dentro del proceso de construcción de una base de datos, el primer paso es la construcción de las estructuras persistentes, es decir, la creación de las tablas de la base de datos y de las relaciones que existen entre ellas; para realizar estas tareas, el usuario se auxilia de instrucciones DDL del lenguaje SQL.

Antes de comenzar con la implementación de una base de datos a través de lenguaje SQL, es necesario conocer el modelo por implementar; de esta manera, como entrada a este proceso, requeriríamos de un diagrama Entidad – Relación o Diagrama de Entidades, el cual permite identificar las tablas existentes en el modelo y su definición; también sería conveniente contar con el Diccionario de Datos, producto del análisis de la base de datos, en éste se realizan especificaciones adicionales sobre el modelo de la base de datos y reglas de negocio por implementar.

## 2.3 Creación de tablas

Previo a la creación de tablas, es necesario realizar la creación de una base de datos. Por lo cual ejemplificaremos cómo se realiza este procedimiento en el Manejador de Base de Datos MySQL.

Una vez que el servidor de base de datos MySQL está trabajando<sup>3</sup>, nos conectaremos a él a través de un cliente, en este caso lo haremos a través del *MySQL Command Line Client*., así ingresamos al RDBMS, éste nos da la bienvenida y el prompt visto en la línea de comandos es el siguiente:

```
mysql>
```

La línea de comandos está a la espera de que introduzcamos nuestras instrucciones SQL. La primera tarea que realizaremos es la creación de una base de datos para trabajar, por lo cual utilizaremos el comando `CREATE DATABASE`.

MySQL al igual que otros manejadores de bases de datos permiten ver la estructura de los comandos SQL que implementa, es decir, nos permite ver cuál es la sintaxis del comando; para ver esta sintaxis en el manejador MySQL, utilizaremos `\? CREATE DATABASE`<sup>4</sup>

La diagonal inversa y el signo de interrogación permiten desplegar información sobre un comando en particular, al ejecutar dicho comando la salida es la siguiente:

```
mysql> \? CREATE DATABASE
Name: 'CREATE DATABASE'
Description:
Syntax:
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification [, create_specification] ...]

create_specification:
    [DEFAULT] CHARACTER SET charset_name
    | [DEFAULT] COLLATE collation_name
```

La interpretación de la sintaxis del comando se hace de la siguiente forma: utilizar la palabra reservada `CREATE` entre corchetes se representa opciones, es decir, que se puede

---

<sup>3</sup> La instalación del RDBMS de MySQL se explica de forma breve en el Anexo A. Instalación de MySQL en Windows XP.

<sup>4</sup> Una variante de `\?` es la utilización de `help`, por ejemplo `mysql>help CREATE DATABASE`

utilizar el comando `CREATE DATABASE` o `CREATE SCHEMA`, seguido del nombre de la base de datos; entre corchete cuadrado ( `[]` ) aparecerán opciones adicionales del comando.

Así, diversas implementaciones del comando serían las siguientes:

```
mysql>CREATE DATABASE bd_prueba1;
mysql>CREATE SCHEMA bd_prueba2;
mysql>CREATE DATABASE IF NOT EXISTS bd_prueba3;
mysql>CREATE DATABASE bd_prueba4 CHARACTER SET utf8 COLLATE utf8_spanish_ci;
mysql> CREATE DATABASE equipo_de_computo;
```

A través de los cuatro ejemplos anteriores es como pudimos crear cuatro bases de datos distintas. La opción `IF NOT EXIST` evalúa la existencia de una base de datos con el mismo nombre antes de crear la base de datos que especificamos; así mismo, las opciones `CHARACTER SET` y `COLLATE` permiten especificar el juego de caracteres que utilizará la base de datos, cuando estas dos opciones no son especificadas MySQL asigna por omisión el juego de caracteres `latin1`, igual es importante señalar que el punto y coma (;) al final de cada sentencia es obligatoria, ya que ayuda al RDBMS a identificar el fin de una instrucción SQL.

Así pues, la estructura general del comando `CREATE DATABASE` es:

```
CREATE DATABASE nombre_base_de_datos.
```

Cuando se ejecuta una instrucción SQL, el servidor ofrecerá una respuesta indicando que se ha ejecutado satisfactoriamente el comando: `Query OK, 1 row affected (0.01 sec)` o bien que existe un error en la instrucción: `ERROR 1007 (HY000): Can't create database...`, de existir, revise que haya transcrito bien el código fuente SQL, revise con detenimiento la escritura de las palabras, recuerde que en principio es difícil familiarizarse tanto con la estructura como con la sintaxis de un nuevo lenguaje. También puede ir a la documentación del RDBMS para informarse sobre la naturaleza del error.

Un comando propio de MySQL utilizado para listar las bases de datos existentes es `SHOW DATABASES`; el cuál mostrará en pantalla las bases de datos que existen. Así pues, si ha ejecutado las instrucciones `CREATE DATABASE` anteriores, el comando `SHOW DATABASES` debe mostrarle lo siguiente:

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| bd_prueba1 |
| bd_prueba2 |
| bd_prueba3 |
| bd_prueba4 |
| equipo_de_computo |
| mysql |
+-----+
7 rows in set (0.01 sec)
```

Una vez listadas las bases de datos, se indica el número de registros existentes y el tiempo que tardó la consulta de esta información.

Como se puede observar dentro de la lista de bases de datos, aparece la base de datos mysql, la cual se crea por defecto al realizar la instalación de MySQL; dicha base de datos proporciona elementos para la administración de las bases de datos creadas por el usuario.

Por defecto cuando ingresamos a MySQL, no estamos haciendo uso de ninguna base de datos, por lo que debemos indicar al manejador que deseamos trabajar con una de ellas en particular, para ello, utilizamos un comando propio de MySQL: el comando `USE`; que permitirá utilizar una base de datos, así pues, ejecutamos el siguiente comando para utilizar la base de datos `equipo_de_computo`:

```
mysql> USE equipo_de_computo;
Database changed
```

Como confirmación recibimos un mensaje que indica que hemos cambiado de base de datos.

Ahora que se ha creado una base de datos, es posible la creación de una tabla<sup>5</sup>, de esta manera, utilizaremos el comando SQL `CREATE TABLE`; para conocer la sintaxis formal del comando implementada en MySQL, ejecutamos el siguiente comando:

```
mysql> \? CREATE TABLE
Name: 'CREATE TABLE'
Description:
Syntax:
CREATE TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_option ...]
```

Dado que la salida de información es amplia, se hará una explicación por cada bloque de información sustancial proporcionada por el comando. En esta primera sección, podemos observar que el comando `CREATE TABLE` también puede implementar la opción `IF NOT EXISTS`, la cual verifica que no exista un objeto con el mismo nombre antes de ser creado el nuevo, luego se debe especificar el nombre de la tabla, las opciones de definición para la creación y las opciones que tendrá la tabla.

Supongamos que deseamos construir una tabla que permita almacenar los diferentes vinos que en una licorería se ofrecen, entonces el comando `CREATE TABLE` para la creación de la tabla, hasta el momento quedaría de la siguiente forma: `CREATE TABLE vinos ( );`

El siguiente bloque de información sustancial presenta las opciones de definición para la creación, las cuales son:

```
create_definition:
    column_definition
    | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
```

---

<sup>5</sup> MySQL implementa diversos tipos de tabla los cuales son explicados en el Anexo B Tipos de Tabla de MySQL. El trabajo realizado en esta publicación se encuentra basado en tablas tipo `INNODB`, que son las de creación por defecto cuando se instala el RDBMS bajo el sistema operativo Windows; en el momento que se trabaje bajo un sistema operativo Linux, entonces el tipo de tabla por defecto es `MyISAM`, por tanto, en este caso se deberá especificar el tipo de tabla por crear como `INNODB` o realizar una tarea administrativa que permita asignar la creación de tablas por defecto a `INNODB`.

```

| {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
| [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY]
  [index_name] [index_type] (index_col_name,...)
| {FULLTEXT|SPATIAL} [INDEX|KEY] [index_name] (index_col_name,...)
| [CONSTRAINT [symbol]] FOREIGN KEY
  [index_name] (index_col_name,...) [reference_definition]
| CHECK (expr)

```

Esta parte nos indica que para la creación de una tabla es necesario definir cuáles serán las columnas que integran a la tabla, y posteriormente, algunas de las opciones enunciadas que sirven para dar integridad a la información entrante. La definición de una columna se hace de la siguiente forma:

column\_definition:

```

col_name data_type [NOT NULL | NULL] [DEFAULT default_value]
  [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
  [COMMENT 'string'] [reference_definition]

```

Colocar el nombre de la columna en primera instancia luego, el tipo de dato, la palabra NULL o la palabra NOT NULL, un valor por defecto, etc. Los tipos de datos implementados por MySQL en su versión 5.03 son:

data\_type:

```

BIT[(length)]
| TINYINT[(length)] [UNSIGNED] [ZEROFILL]
| SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
| MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
| INT[(length)] [UNSIGNED] [ZEROFILL]
| INTEGER[(length)] [UNSIGNED] [ZEROFILL]
| BIGINT[(length)] [UNSIGNED] [ZEROFILL]
| REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
| DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
| FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
| DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
| NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
| DATE
| TIME
| TIMESTAMP
| DATETIME
| YEAR
| CHAR(length)
  [CHARACTER SET charset_name] [COLLATE collation_name]
| VARCHAR(length)
  [CHARACTER SET charset_name] [COLLATE collation_name]
| BINARY(length)
| VARBINARY(length)
| TINYBLOB
| BLOB
| MEDIUMBLOB
| LONGBLOB
| TINYTEXT [BINARY]
  [CHARACTER SET charset_name] [COLLATE collation_name]
| TEXT [BINARY]
  [CHARACTER SET charset_name] [COLLATE collation_name]
| MEDIUMTEXT [BINARY]
  [CHARACTER SET charset_name] [COLLATE collation_name]
| LONGTEXT [BINARY]
  [CHARACTER SET charset_name] [COLLATE collation_name]
| ENUM(value1,value2,value3,...)
  [CHARACTER SET charset_name] [COLLATE collation_name]

```

```
| SET(value1,value2,value3,...)
| [CHARACTER SET charset_name] [COLLATE collation_name]
| spatial_type
```

Para mayor referencia sobre estos tipos de datos es necesario revisar la documentación de MySQL generalmente disponible en [www.mysql.com](http://www.mysql.com)

¿Cómo se aplica esta nueva información al comando CREATE TABLE?, muy sencillo, regresemos al ejemplo de la creación de una tabla para manejar un catálogo de vinos, teníamos la siguiente sintaxis: `CREATE TABLE vinos ( );`

La definición de una columna se realizaría de la siguiente forma:

```
CREATE TABLE vinos
(
Nombre_columa tipo_dato [NULL | NOT NULL] CONSTRAINT ... ,
Nombre_columa2 tipo_dato [NULL | NOT NULL] CONSTRAINT ... ,
);
```

Un **CONSTRAINT** es una regla que se debe cumplir para cuando se desee ingresar un dato en dicha columna. Para continuar con la instrucción SQL DDL, en la creación de una tabla incrementamos el código fuente de la siguiente manera:

```
CREATE TABLE vinos
(
idVino integer NOT NULL PRIMARY KEY,
txtNombreVino varchar(25) NOT NULL
);
```

La tabla Vinos especifica dos columnas, una de ellas permitirá identificar de forma única a un vino, es decir, será utilizada para ingresar la clave del vino; la segunda columna, permitirá almacenar la descripción del nombre del vino. Observe que luego de la palabra reservada **NOT NULL** aparece **PRIMARY KEY**, esta es una restricción de llave primaria, es decir, los valores que se almacenen en la columna `idVino` deberán ser únicos para cada caso. Es pertinente hacer notorio que la definición de columnas está separada por comas (,) sin embargo, la última definición de columna no deberá incluir la coma final, ya que antes del paréntesis no debe existir ésta, revise con detenimiento la instrucción anterior.

Como acabamos de ver, la estructura básica de un comando **CREATE TABLE** incluye: la palabra reservada **CREATE TABLE**, el nombre de la tabla que deseamos crear y entre paréntesis se especifica la definición de las columnas que conformarán a la tabla en cuestión. La definición de columnas de una tabla puede hacerse a través de diversas combinaciones entre tipos de datos, restricciones y otros, por lo que es importante que aprenda a interpretar la sintaxis proporcionada por el RDBMS, para así, usted pueda implementar la combinación que mejor se adecue a sus necesidades.

```
CREATE TABLE nombre_tabla
(
Definición de columna
);
```

Así pues, ejecute el siguiente comando para crear la tabla de vinos:

```
mysql> CREATE TABLE vinos
(
  idVino integer NOT NULL PRIMARY KEY,
  txtNombreVino varchar(25) NOT NULL
);
```

Nota: recuerde que “mysql>” es el prompt que usted visualiza cuando está conectado al servidor de base de datos MySQL, las instrucciones SQL que usted debe teclear comienzan después de este prompt.

Para comprobar que se ha creado la tabla que indicamos anteriormente, MySQL implementa dos comandos, el primero es **SHOW TABLES**, el cuál muestra las tablas existentes dentro de la base de datos que está actualmente en uso, y el comando **DESCRIBE nombre\_tabla**, que mostrará la estructura de la tabla indicada.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_equipo_de_computo |
+-----+
| vinos                        |
+-----+
1 row in set (0.00 sec)
```

```
mysql> DESCRIBE vinos;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| idVino     | int(11)       | NO   | PRI |          |       |
| txtNombreVino | varchar(25)  | NO   |     |          |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.17 sec)
```

Dentro de la especificación de columnas existen diferentes palabras reservadas que van a permitir implementar reglas de diversos tipos, ya sean de integridad, de valores por omisión, etc. Por ello, la siguiente tabla da una explicación sobre las diversas palabras reservadas implementadas en ANSI SQL 2003 para la creación de restricciones (reglas).

| Palabra reservada | Regla que implementa                                                                                                                                                    |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NULL / NOT NULL   | Permite o no que se acepten valores nulos en un campo. Un valor nulo es aquel que es desconocido, está vacío el campo, es diferente de un espacio, salto de línea, etc. |
| CONSTRAINT        | Es una palabra reservada que define que a continuación se especificará una restricción de algún tipo.                                                                   |
| DEFAULT           | Permite indicar el valor por omisión que tomará el campo cuando no se especifique éste al momento de introducir los datos en la tabla.                                  |
| CHECK             | Evalúa que los datos ingresados para un campo se encuentren dentro de un subconjunto de datos predeterminado.                                                           |
| AUTO_INCREMENT    | Utilizado para hacer autoincrementable el valor de un campo conforme se hagan inserciones en la tabla.                                                                  |

| Palabra reservada          | Regla que implementa                                                                                                                               |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| PRIMARY KEY                | Implementa una restricción de Llave Primaria, en donde el valor correspondiente a cada atributo debe ser único en la tabla.                        |
| FOREIGN KEY ... REFERENCES | Implementa una restricción de llave Foránea, que sirve de vínculo entre las tablas, regla de padres a hijos.                                       |
| UNIQUE [KEY   INDEX]       | Implementa una restricción para un campo en donde para cada nuevo valor se compruebe que no exista en la tabla, es decir, aceptará valores únicos. |

La sintaxis utilizada en uno u otro manejador puede variar según la implementación que hagan de ANSI SQL.

La creación de una tabla, en su definición de columnas, puede incluir una o más de estas opciones, a continuación veremos algunos ejemplos sencillos de cómo crear una instrucción SQL DDL para la creación de tablas con distintas opciones.

## 2.4 Creación de reglas para datos por omisión

Tomando como base la estructura del comando CREATE TABLE y las definiciones de columna propias del comando, implementaremos reglas para datos por omisión.

En el capítulo siguiente, aprenderá el Lenguaje para la Manipulación de los Datos (DML), donde se ilustra cómo ingresar datos a las tablas creadas a través del comando INSERT, por el momento, únicamente haremos referencia de éste comando para ilustrar el impacto que tiene la definición de las columnas en la creación de una tabla.

### 2.4.1 NULL / NOT NULL

Para ejemplificar el concepto de valor Nulo, crearemos una tabla con dos campos de tipo Entero: en el primer caso debe aceptar valores nulos y en segundo no aceptar valores nulos.

```
CREATE TABLE ejemploNulos
(
  intAceptaNulos integer NULL,
  intNoNulos integer NOT NULL
);
```

Si hacemos una descripción de la tabla, podemos observar que en efecto uno de los dos campos aceptará valores nulos.

```
mysql> DESCRIBE ejemploNulos;
+-----+-----+-----+-----+-----+-----+
| Field          | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| intAceptaNulos | int(11) | YES  |     | NULL    |       |
| intNoNulos     | int(11) | NO   |     |         |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.42 sec)
```

Para ingresar datos a la tabla antes creada, ejecutaremos las siguientes instrucciones de tipo INSERT.

```
INSERT INTO ejemploNulos VALUES(NULL,1);
INSERT INTO ejemploNulos VALUES(NULL,2);
INSERT INTO ejemploNulos VALUES(1,3);
INSERT INTO ejemploNulos VALUES(2,4);
```

En las primeras dos instrucciones, cuando ingresamos valores para la columna `intAceptaNulos`, asignamos la palabra `NULL`, que se encuentra sin un entrecomillado, lo cual la hace una palabra reservada, cuando insertamos de esta forma un dato, le decimos que dicho dato es Nulo, es decir no existe.

El efecto producido es que al insertar un nuevo registro siempre se deberá especificar valor para los campos indicados como `NOT NULL`, mientras que los campos indicados como `NULL` pueden carecer de valor. Si realizamos inserciones a la tabla anterior, obtendríamos un resultado como el siguiente.

| intAceptaNulos | intNoNulos |
|----------------|------------|
| NULL           | 1          |
| NULL           | 2          |
| 1              | 3          |
| 2              | 4          |

La primera columna tiene dos registros con `NULL`, no quiere decir que se haya insertado esa frase, sino que tal registro en ese campo carece de valor; esto fue posible por que se declaró la columna como `NULL`, mientras que la segunda columna siempre tiene un dato, ya que no es posible dejarla como `NULL`.

Si intentamos realizar una inserción de un campo `Null` en la columna `intNoNulos`, obtendremos el siguiente error:

```
mysql> INSERT INTO ejemploNulos VALUES(3,null);
ERROR 1048 (23000): Column 'intNoNulos' cannot be null
```

### 2.4.2 DEFAULT

Aplicar un `Default` en una definición de columna refiere a que al momento de inserción de un dato, éste puede darse por defecto, es muy recurrido utilizar los `Default` cuando hay un conjunto de valores pequeños que puede tomar un campo y, sin embargo, casi siempre se insertan datos con un valor en particular. Suponga que es una base de datos para el control de inventarios de una librería, donde existen libros en español e inglés, de los cuales el 90% de dichos libros que posee son en español; con la idea de facilitar el trabajo se crea una tabla con el `DEFAULT` español para la tabla *libro*, de esta manera se reduce la la carga de trabajo a los capturistas de información.

Ejemplo:

```
CREATE TABLE libro
(
  IdLibro integer NOT NULL PRIMARY KEY,
  txtNombreLibro varchar(50) NOT NULL,
  txtidioma varchar(20) NOT NULL DEFAULT 'Español'
);
```

Ahora, realizaremos algunas inserciones para ejemplificar el funcionamiento de la definición de columna hecha con una regla de datos por omisión.

```
INSERT INTO libro VALUES(1, 'Bases de Datos Relacionales', 'Español');
INSERT INTO libro VALUES(2, 'SQL Practical Guide Tool', 'Inglés');
INSERT INTO libro VALUES(3, 'ANSI SQL 2003, Guía de Referencia',DEFAULT);
```

Observe que en las inserciones de registros anteriores, para el caso 1 y 2, se especifica el idioma del libro, no así, para el caso 3 en donde el título del libro se especifica como DEFAULT, entonces, el RDBMS MySQL, revisa la definición de la tabla y asigna el valor por defecto para la columna idioma, que es 'Español'.

Cuando consultamos la información contenida en la tabla, encontramos lo siguiente:

| IdLibro | txtNombreLibro                    | txtidioma |
|---------|-----------------------------------|-----------|
| 1       | Bases de Datos Relacionales       | Español   |
| 2       | SQL Practical Guide Tool          | Inglés    |
| 3       | ANSI SQL 2003, Guía de Referencia | Español   |

Así, el uso de una definición de columna con un valor por omisión como es el DEFAULT, permitirá facilitar la tarea de inserción de datos.

### 2.4.3 CHECK

En la creación de tablas, particularmente en la definición de columnas, es posible aplicar una regla de tipo CHECK, para verificar que los datos por ingresar están contenidos dentro de un conjunto de valores válidos. Haciendo un pequeño cambio al ejemplo anterior, suponga que los libros que se almacenarán en la tabla *libro* antes creada, sólo pueden ser del idioma español o inglés, entonces cuando se ingresen los datos, se requerirá verificar que el dato por ingresar se encuentre en el conjunto de valores válidos definidos.

Ahora, crearemos una segunda tabla *libro2* para ilustrar una definición de columna con una regla CHECK.<sup>6</sup>

```
CREATE TABLE libro2
(
  IdLibro integer NOT NULL PRIMARY KEY,
  txtNombreLibro varchar(50) NOT NULL,
  txtidioma varchar(20) NOT NULL CHECK(txtidioma='Español' OR
  txtidioma='Inglés')
);
```

La especificación de un CHECK se hace con la palabra reservada CHECK y entre paréntesis la expresión a evaluar, en el caso de la tabla *libro2*, la expresión indica que el idioma debe ser 'Español' o 'Inglés'.

Cuando tratamos de ingresar un valor que se encuentra fuera del conjunto de valores válidos, entonces el RDBMS regresará un mensaje de error, por ejemplo:

```
INSERT INTO libro2 VALUES(5, 'Bases de Datos Relacionales', 'Alemán');
```

---

<sup>6</sup> La restricción de tipo CHECK en el RDBMS MySQL, no se encuentra correctamente implementada, se ejemplificará utilizando PostgreSQL.

```
ERROR: new row for relation "libro2" violates check constraint
"libro2_txtidioma_check"
```

El mensaje de error anterior indica que no se ha podido ingresar el Nuevo registro por que no cumple con la condición de que el valor para la columna idioma sea español o inglés.

MySQL para la implementación de reglas CHECK hace uso de un tipo de dato específico (ENUM), la implementación en una definición de columna es la siguiente:

```
CREATE TABLE libro3
(
  IdLibro integer NOT NULL PRIMARY KEY,
  txtNombreLibro varchar(50) NOT NULL,
  txtidioma ENUM('Español', 'Inglés')
);
```

Al tratar de ingresar un valor no especificado para la columna txtidioma a través de un comando INSERT, regresa el error siguiente:

```
mysql> INSERT INTO libro3 VALUES(1, 'Bases de Datos Relacionales',
'Alemán');
ERROR 1265 (01000): Data truncated for column 'txtidioma' at row 1
```

#### 2.4.6 AUTO\_INCREMENT

Cuando deseamos que los valores para un campo en un registro de una tabla se incrementen automáticamente, utilizamos en la definición de columna la regla de AUTO\_INCREMENT.

Por ejemplo, suponga una tabla que será el catálogo de productos ofertados por una empresa dedicada a la venta de computadoras, entonces se desea asignar un número de producto que será consecutivo, para ello, se podrá forzar al manejador de base de datos asigne automáticamente el valor siguiente para el campo especificado. La sintaxis para la definición de la tabla es la siguiente:

```
CREATE TABLE productos
(
  intNumProducto int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  txtDescripcion varchar(50) NOT NULL
);
```

Así, al hacer inserciones en la tabla, el número de producto se generará automáticamente a menos que se especifique lo contrario; cabe resaltar que sólo puede haber una columna con definición AUTO\_INCREMENT por cada tabla y ésta no se podrá combinar con un DEFAULT.

Para hacer las inserciones, ejecutamos los comandos siguientes:

```
INSERT INTO productos VALUES(DEFAULT, 'Compac Presario 1500');
INSERT INTO productos VALUES(DEFAULT, 'Hp Inspiron');
INSERT INTO productos VALUES(DEFAULT, 'DELL Dimension 4500');
INSERT INTO productos VALUES(600, 'SONY VAIO');
```

Así en los primeros tres casos se toma el valor auto incrementable, lo cual se hace al asignar un DEFAULT, a su vez, se puede especificar un valor fuera del incremento normal.

Al consultar la información de la tabla obtenemos este resultado:

```
+-----+-----+
| intNumProducto | txtDescripcion      |
+-----+-----+
```

|     |                      |
|-----|----------------------|
| 1   | Compac Presario 1500 |
| 2   | Hp Inspiron          |
| 3   | DELL Dimension 4500  |
| 600 | SONY VAIO            |

## 2.5 Reglas de integridad referencial

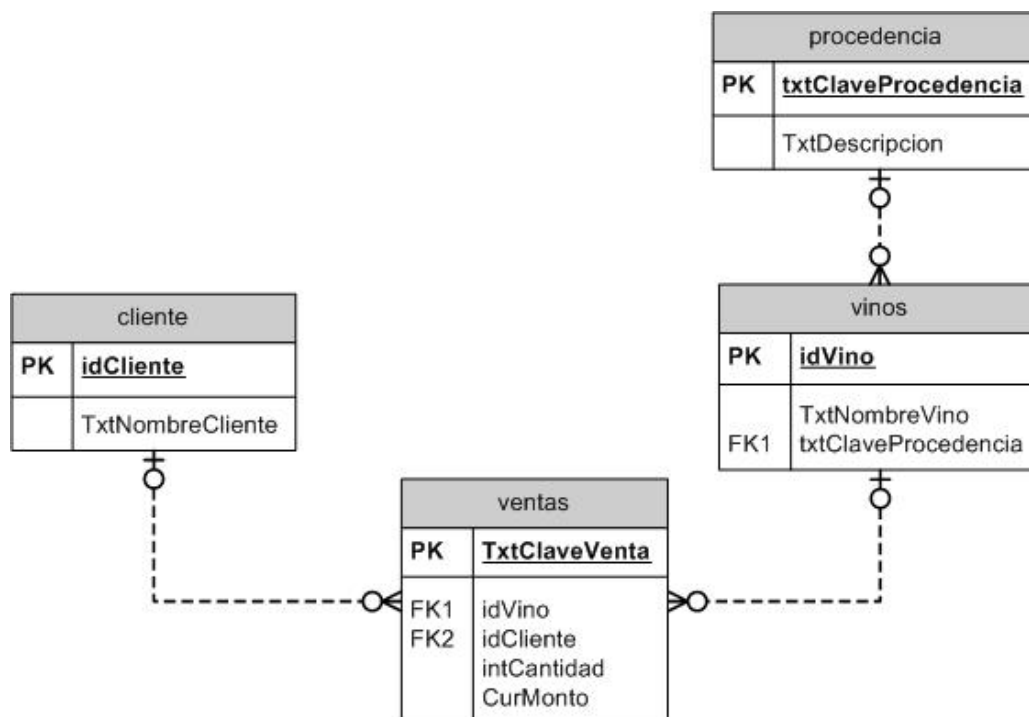
Las reglas de integridad referencial dentro del Modelo de Base de Datos Relacional, refieren a la construcción lógica de llaves, aplicadas sobre atributos de una relación para asegurar que la información contenida en la base de datos sea consistente. Las llaves que hacen cumplir la integridad referencial a través de la regla de padres a hijos son las llaves Primarias y Foráneas.

Una llave Primaria se aplica sobre un atributo que va a identificar de manera única una tupla en una relación, ejemplos de llaves primarias son: RFC, número de seguridad social, matrícula escolar, etc.

Una llave Foránea se aplica sobre un atributo de una relación A que hará referencia a un atributo en una relación B, reflejando un vínculo de asociación entre éstas.

Realizando el mapeo dentro de un RDBMS dichas llaves lógicas se implementan a través de **CONSTRAINTS** o restricciones de llave Primaria y Foránea.

Ahora sabemos que las llaves se aplican sobre atributos o mejor dicho sobre las columnas de una tabla, hablando en términos de un Manejador de Base de Datos Relacional. Para ejemplificar la forma en que se implementan estas reglas, primero debemos interpretar el siguiente diagrama Entidad - Relación.



Vinatería Diagrama Entidad – Relación.

El anterior diagrama Entidad – Relación es planteado como solución ante la problemática de una vinatería para hacer un seguimiento de las ventas que realiza. La tabla *cliente*, servirá para administrar los clientes a los que se venden los productos, dichos productos son vinos, que a su vez pueden ser de distinta procedencia, Francia o España principalmente; por último, la tabla *ventas* refleja la carga transaccional que tendrá la base de datos.

Comenzaremos la implementación del modelo con la construcción de las tablas que almacenarán la información, las tablas que se crearán en primera instancia serán las Entidades fuertes<sup>7</sup>: Cliente y Procedencia.

```
CREATE TABLE cliente
(
  idCliente int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  TxtNombreCliente varchar(70) NOT NULL
);
```

La tabla cliente ya hace una implementación de Llave Primaria a través de la palabra reservada **PRIMARY KEY** en la definición de columna, esto indica que cada valor almacenado en el campo *idCliente* deberá ser único, a su vez se indica que el campo tendrá un valor auto incrementable. La definición de llave primaria (PK) de la tabla cliente, es una definición a nivel de columna, ya que se especifico dentro de la definición de columna.

```
CREATE TABLE procedencia
(
  txtClaveProcedencia char(3) NOT NULL,
  TxtDescripcion varchar(50) NOT NULL,
  CONSTRAINT PRIMARY KEY(txtClaveProcedencia)
);
```

De igual forma para el campo *txtClaveProcedencia* se indica una restricción de llave primaria, pero esta vez dicha restricción se hace a nivel de tabla, y por ello, se utiliza una sintaxis diferente, primero la palabra reservada **CONSTRAINT** la cual especifica que se implementará una regla, posteriormente la palabra reservada **PRIMARY KEY** que es el tipo de restricción que se hace y posteriormente y entre paréntesis, se especifican los atributos que serán declarados como llave primaria. La sintaxis en general es la siguiente:

```
CONSTRAINT PRIMARY KEY (atributo1, atributo2, atributo3, ..., atributoN)
```

Una llave primaria puede estar formada aproximadamente hasta por 132 atributos dependiendo la versión ANSI SQL implementada por el RDBMS por utilizar. Cuando una pk es formada por dos o más atributos se dice que es una llave primaria compuesta.

En MySQL podemos utilizar el comando **DESCRIBE** para ver la estructura de una tabla, por ejemplo:

```
mysql> DESCRIBE cliente;
```

---

<sup>7</sup> Una entidad fuerte es aquella que no requiere de otra para existir, por ejemplo Cliente, puede existir un cliente independientemente de cualquier otra entidad en el diseño de la base de datos, a diferencia de esto, las entidades débiles son aquellas que requieren de otra entidad para existir, por ejemplo, la tabla de ventas, ya que para que ésta exista, debe haber previamente un cliente y un producto.

```
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| idCliente      | int(11)       | NO   | PRI | NULL    | auto_increment |
| TxtNombreCliente | varchar(70)   | NO   |     |         |                |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

La respuesta que ofrece el manejador para el caso de la tabla *cliente* nos permite observar que la columna *idCliente* tiene una especificación de llave primaria.

Continuando con la creación de las tablas, ahora revisaremos la construcción de la tabla *vinos*, que como se muestra en el diagrama tiene una relación con la tabla *procedencia*, dicha relación o vínculo se implementará a través de una llave primaria en la tabla *procedencia* vinculada con una llave foránea en la tabla de vinos.

```
CREATE TABLE vinos
(
  idVino int NOT NULL PRIMARY KEY,
  TxtNombreVino varchar(50) NOT NULL,
  TxtClaveProcedencia char(3) NOT NULL REFERENCES
  procedencia(txtClaveProcedencia)
);
```

Cuando se hace una especificación de llave foránea (fk) dentro de una tabla, las columnas que se vincularán deberán ser del mismo tipo de dato, por ello en el caso de la tabla *vinos* la columna *TxtClaveProcedencia*, es del mismo tipo de dato *char(3)*, que en la tabla *procedencia*.

La implementación anterior de llave foránea se hace a nivel de columna<sup>8</sup> utilizando la palabra reservada *REFERENCES*, el nombre de la tabla a la que se vinculará y entre paréntesis, la columna perteneciente a la tabla indicada anteriormente, que servirá de vínculo.

Al igual que para la especificación de llaves primarias, una llave foránea se puede implementar a nivel de tabla, y para ejemplificarlo, crearemos la tabla restante: la tabla de ventas.

```
CREATE TABLE ventas
(
  TxtClaveVentas varchar(14 )NOT NULL PRIMARY KEY,
  idVino int NOT NULL,
  idCliente int NOT NULL,
  CONSTRAINT fk_ventas1 FOREIGN KEY (idVino) REFERENCES vinos(IdVino),
  CONSTRAINT fk_ventas2 FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);
```

La implementación de una llave foránea a nivel de tabla se hace colocando la palabra reservada *CONSTRAINT* seguida del nombre de la restricción especificada por el usuario, la palabra reservada *FOREIGN KEY*, entre paréntesis la columna de la tabla que estamos creando, la cual servirá de vínculo, así como la palabra reservada *REFERENCES*,

---

<sup>8</sup> MySQL en su versión 5 no hace una correcta implementación de llave foránea a nivel de columna, ya que, no se cumple satisfactoriamente la restricción de padres a hijos. Cuando trabaje con MySQL es preferible hacer una definición de llave foránea a nivel de tabla.

que corresponde a la tabla donde se encuentra la llave primaria y entre paréntesis la columna a la cual se vinculará. De forma genérica, la sintaxis es la siguiente:

```
CONSTRAINT nombre_constraint FOREIGN KEY (columna) REFERENCES
tabla1(columna)
```

En algunos RDBMS, siempre que se especifica una restricción sea del tipo que sea, seguido de la palabra **CONSTRAINT** debe ir un nombre, ya que esa regla se almacena en las tablas que se manejan como Diccionario de Datos donde es obligatorio indicar un nombre.

Para verificar que se ha implementado correctamente la especificación de la llave foránea en una tabla, podemos utilizar el comando **DESCRIBE** que reflejara, en la salida, con el valor **MUL** en la columna **key** que hay una llave foránea especificada.

```
mysql> DESCRIBE ventas;
```

| Field          | Type        | Null | Key | Default | Extra |
|----------------|-------------|------|-----|---------|-------|
| TxtClaveVentas | varchar(14) | NO   | PRI |         |       |
| idVino         | int(11)     | NO   | MUL |         |       |
| idCliente      | int(11)     | NO   | MUL |         |       |

3 rows in set (0.00 sec)

Con esto terminamos el mapeo del Diagrama Entidad – Relación a instrucciones SQL DDL, obteniendo como resultado cuatro tablas existentes en el manejador de base de datos, cada una con reglas distintas; en capítulos siguientes, haremos referencia a estas tablas (clientes, procedencia, vinos y ventas) que serán utilizadas para ver instrucciones SQL DML.

## 2.6 Reglas múltiples

En la definición de columnas para la creación de tablas, es posible asignar reglas múltiples, debido a que un caso del mundo real no limita a una sola restricción, puede existir una multiplicidad.

En los ejemplos anteriores se han mezclado reglas, sin hacerlo notar explícitamente, por ejemplo en el caso de la tabla **producto** utilizada para el manejo de productos de una empresa dedicada a la venta de equipos de cómputo, en la definición de la columna **intNumProducto**, donde se especifican tres reglas: de denegación de nulos, de auto incremento y de llave primaria.

```
CREATE TABLE productos
(
  intNumProducto int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  txtDescripcion varchar(50) NOT NULL
);
```

Por tanto, se pueden implementar reglas múltiples en la definición de columnas de una tabla, siempre y cuando dichas reglas no resulten mutuamente excluyentes como utilizar **AUTO\_INCREMENT** y **DEFAULT**.

## 2.7 Modificación de tablas

Las necesidades de las organizaciones, siempre van en aumento y resulta común que una vez que se implantado un sistema de base de datos en una organización y que se ha

trabajado con él por algún periodo de tiempo razonable, se requiera almacenar nueva información que no estaba contemplada en el diseño original de base de datos, es decir, hay una tendencia a incrementar o evolucionar los sistemas que utilizan las organizaciones con el objeto de adecuarlos a situaciones actuales; es por ello que en algún momento se requiera agregar un campo a una tabla, agregar otra tabla y vincularla a las ya existentes o modificar alguna otra regla.

La instrucción SQL `ALTER` va a permitir alterar la estructura de un objeto de la base de datos, ya se una tabla, una vista, un índice u otro, con el objeto de adecuarlo a las necesidades actuales.

La sentencia `ALTER TABLE` será la encargada de hacer alteraciones a la estructura de una tabla, permitiendo ampliar o reducir las características con que se cuentan.

La sintaxis genérica del comando `ALTER TABLE` bajo el RDBMS de MySQL es la siguiente:

```
ALTER [IGNORE] TABLE tbl_name
    alter_specification [, alter_specification] ...

alter_specification:
    table_option ...
| ADD [COLUMN] column_definition [FIRST | AFTER col_name ]
| ADD [COLUMN] (column_definition,...)
| ADD {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
| ADD [CONSTRAINT [symbol]]
    PRIMARY KEY [index_type] (index_col_name,...)
| ADD [CONSTRAINT [symbol]]
    UNIQUE [INDEX|KEY] [index_name] [index_type] (index_col_name,...)
| ADD [FULLTEXT|SPATIAL] [INDEX|KEY] [index_name] (index_col_name,...)
| ADD [CONSTRAINT [symbol]]
    FOREIGN KEY [index_name] (index_col_name,...)
    [reference_definition]
| ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
| CHANGE [COLUMN] old_col_name column_definition
    [FIRST|AFTER col_name]
| MODIFY [COLUMN] column_definition [FIRST | AFTER col_name]
| DROP [COLUMN] col_name
| DROP PRIMARY KEY
| DROP {INDEX|KEY} index_name
| DROP FOREIGN KEY fk_symbol
| DISABLE KEYS
| ENABLE KEYS
| RENAME [TO] new_tbl_name
| ORDER BY col_name [, col_name] ...
```

Dentro de las posibilidades que ofrece el comando `ALTER TABLE` tenemos:

- Agregar
  - Columnas, índices, llaves primarias o foráneas.
- Modificar la definición de columna para una columna existente.
- Eliminar
  - Columnas, llaves o índices.

A continuación, veremos ejemplos para cada una de las posibilidades planteadas, tomaremos como base las tablas creadas para el modelo de base de datos de la vinatería.

De acuerdo con el modelo de base de datos planteado a la vinatería, se ha sugerido se agregue el campo `fechaDeVenta`, para la tabla de ventas, ya que es necesaria la generación de reportes estadísticos de ventas de acuerdo con periodos de tiempo. Para cumplir con el objetivo de agregar dicha columna a la tabla de ventas, agregaremos la columna utilizando el comando `ALTER TABLE`, para que quede de la siguiente manera la sentencia.

```
ALTER [IGNORE] TABLE nombre_tabla
  | ADD [COLUMN] (definicion_de_columna,...)
```

Ejemplo:

```
ALTER TABLE ventas
ADD dateFechaVenta timestamp;
```

Al realizar una descripción de la tabla `ventas`, obtenemos el siguiente resultado:

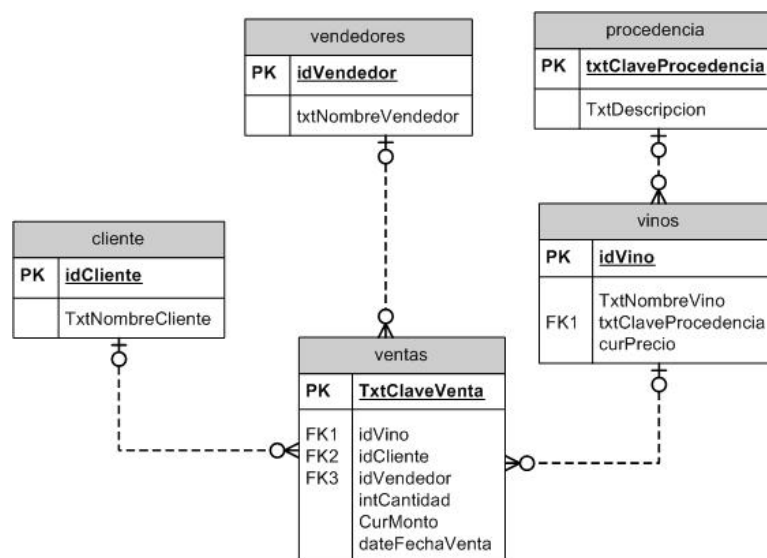
| Field          | Type        | Null | Key | Default           | Extra |
|----------------|-------------|------|-----|-------------------|-------|
| TxtClaveVentas | varchar(14) | NO   | PRI |                   |       |
| idVino         | int(11)     | NO   | MUL |                   |       |
| idCliente      | int(11)     | NO   | MUL |                   |       |
| dateFechaVenta | timestamp   | YES  |     | CURRENT_TIMESTAMP |       |

Ahora observamos que existe una nueva columna, respecto de la definición original de la tabla `ventas`. El comando `ALTER TABLE` tiene una gran sencillez y a su vez una gran potencialidad, ya que existe una amplia gama de posibilidades en su uso.

También agregaremos la columna `precio` a la tabla `vinos`:

```
ALTER TABLE vinos ADD curPrecio decimal (10,2) NOT NULL;
```

A su vez, se ha determinado que el modelo de base de datos de la vinatería contemple a los vendedores, ya que a éstos se les paga una comisión mensual de acuerdo con el número de ventas que hayan realizado. El impacto de esta nueva regla de negocio se puede ver reflejado en la versión 2 del modelo de base de datos para la vinatería.



## Vinatería\_v2 Diagrama Entidad – Relación.

El cambio respecto del modelo anterior implica una nueva tabla para la administración de los vendedores, y se han agregado dos nuevas columnas a la tabla ventas, que son: `dateFechaVenta` y `idVendedor` con restricción de llave foránea.

Para implementar esta nueva solución, debemos en primera instancia crear la tabla vendedores:

```
CREATE TABLE vendedores
(
  idVendedor integer NOT NULL PRIMARY KEY,
  txtNombreVendedor varchar(50) NOT NULL
);
```

Ahora agregamos la columna `idVendedor` para la tabla ventas.

```
ALTER TABLE ventas ADD idVendedor integer NOT NULL;
```

Para agregar el vínculo entre las tablas de vendedor y ventas, utilizaremos una sentencia `ALTER TABLE`, agregando una restricción de llave foránea.

```
ALTER TABLE nombre_tabla
ADD CONSTRAINT regla;
```

La sentencia anterior refleja la sintaxis general del comando `ALTER TABLE` para agregar una nueva regla; cuando dicha regla sea el caso de una restricción de llave primaria o foránea, se especificará tal como se hizo con la restricción de llave primaria o foránea a nivel de tabla, quedando este ejemplo de la siguiente manera:

```
ALTER TABLE ventas ADD CONSTRAINT fk3_ventas FOREIGN KEY(idVendedor)
REFERENCES vendedor(idVendedor);
```

Al realizar una descripción de la tabla `ventas`, obtenemos el siguiente resultado:

| Field          | Type        | Null | Key | Default           | Extra |
|----------------|-------------|------|-----|-------------------|-------|
| TxtClaveVentas | varchar(14) | NO   | PRI |                   |       |
| idVino         | int(11)     | NO   | MUL |                   |       |
| idCliente      | int(11)     | NO   | MUL |                   |       |
| dateFechaVenta | timestamp   | YES  |     | CURRENT_TIMESTAMP |       |
| idVendedor     | int(11)     | NO   | MUL |                   |       |

El comando `ALTER TABLE` también permite modificar la definición de una columna, a través de la mezcla con el comando `ALTER COLUMN`. La sintaxis general de esta opción es la siguiente:

```
ALTER TABLE nombre_tabla
ALTER COLUMN nombre_columna [definición_columna] SET [DEFAULT | CHECK | ...];
```

En el sistema de base de datos para la vinatería se requiere asignar como valor por defecto 'ESP' para la clave de procedencia de un vino, ya que el 75% de los vinos que maneja la vinatería son españoles, este valor por defecto se deberá asignar en la tabla `vinos`.

La implementación de esta regla de negocio en el RDBMS, la haremos a través del comando `ALTER TABLE` de la siguiente forma:

```
ALTER TABLE vinos ALTER COLUMN txtClaveProcedencia SET DEFAULT 'ESP';
```

Realizar modificaciones sobre una columna, no sólo implica agregar restricciones u otras reglas, se puede también hacer una redefinición de columna, para ello, utilizaremos dos variantes del comando ALTER TABLE: CHANGE y MODIFY.

CHANGE permite cambiar el nombre de una columna y su definición, opcionalmente también permite indicar la posición de la columna, es decir, si deseamos que aparezca antes o después de otra columna.

Para cambiar la definición de una columna de la tabla de vinos veamos el siguiente ejemplo:

| Field               | Type        | Null | Key | Default | Extra |
|---------------------|-------------|------|-----|---------|-------|
| idVino              | int(11)     | NO   | PRI |         |       |
| TxtNombreVino       | varchar(50) | NO   |     |         |       |
| TxtClaveProcedencia | char(3)     | NO   |     |         |       |

Si deseáramos cambiar el tipo de dato de la columna de Nombre de Vino, renombrarla y cambiarla de posición, ejecutamos el comando siguiente:

```
ALTER TABLE vinos CHANGE TxtNombreVino TxtNombre varchar(25) NOT NULL AFTER TxtClaveProcedencia;
```

Como resultado, a continuación vemos en la nueva descripción de la tabla de vinos, que el nombre de la columna para el nombre del vino cambió, así como la longitud del tipo de dato que utilizaba, cambiando de 50 a 25 caracteres; la posición de la columna de nombre se movió al final de las columnas.

| Field               | Type        | Null | Key | Default | Extra |
|---------------------|-------------|------|-----|---------|-------|
| idVino              | int(11)     | NO   | PRI |         |       |
| TxtClaveProcedencia | char(3)     | NO   |     |         |       |
| TxtNombre           | varchar(25) | NO   |     |         |       |

MODIFY se utiliza en el mismo sentido, sirve para cambiar la definición de una columna y su posición dentro de la tabla, pero no es posible cambiar el nombre. Por ejemplo:

```
ALTER TABLE vinos MODIFY TxtNombre varchar(50) NOT NULL AFTER idVino;
```

El resultado obtenido se aprecia en la descripción de la tabla *vinos*, donde se cambió la definición de la columna *nombre* y su posición en la tabla.

| Field               | Type        | Null | Key | Default | Extra |
|---------------------|-------------|------|-----|---------|-------|
| idVino              | int(11)     | NO   | PRI |         |       |
| TxtNombre           | varchar(50) | NO   |     |         |       |
| TxtClaveProcedencia | char(3)     | NO   |     |         |       |

La diferencia entre CHANGE y MODIFY es que CHANGE permite cambiar el nombre de la columna en cuestión, ya que en ambos casos el objetivo es cambiar la definición de una columna.

Para finalizar, cambiaremos el nombre de la columna `TxtNombre` a `TxtNombreVino` con la ayuda del comando `CHANGE`.

```
ALTER TABLE vinos CHANGE TxtNombre TxtNombreVino varchar(25) NOT NULL;
```

La eliminación de columnas es una tarea relativamente sencilla; el comando que logra este efecto por supuesto es `ALTER TABLE` con la variante `DROP`, la sintaxis genérica por utilizar es la siguiente:

```
ALTER TABLE nombre_tabla DROP nombre_columna;
```

En ejemplo de la utilización de este comando es el siguiente:

```
ALTER TABLE vinos DROP precio;
```

Con ello se eliminaría la columna `precio` de la tabla `vinos`, sin embargo, no ejecutaremos la instrucción en el RDMS ya que será útil dicha columna en ejemplos posteriores.

Haga las modificaciones pertinentes en la base de datos utilizando los comando `CREATE` y `ALTER` para que la base de datos se ajuste al modelo de la vinatería en su versión 2.

## **2.8 Eliminación de tablas**

La eliminación de tablas es una tarea poco frecuente en un RDBMS, no obstante, dicha posibilidad tiene implementación a través del comando `DROP`, el cual elimina objetos de la base de datos entre los que tenemos tablas, vistas, procedimientos almacenados, etc. La sintaxis genérica de este comando es:

```
DROP [TABLE | VIEW | DATABASE | ... ] nombre_objeto;
```

Para ejemplificar esta instrucción SQL, crearemos una tabla de ejemplo de la siguiente manera:

```
CREATE TABLE ejemplo  
(  
  numero1 integer  
);
```

Para la eliminación de la tabla, el comando `DROP` se implementaría de la siguiente forma:

```
DROP TABLE ejemplo;
```

El comando `DROP` debe ser utilizado con precaución al igual que las instrucciones para la modificación de datos, ya que el RDBMS no pedirá confirmación a menos que se configure esta opción.

## **2.9 Creación de índices**

Conceptualmente, un índice es un mecanismo de ordenación en una tabla, ya sea para una o más columnas, con el objeto de mejorar el rendimiento de una base de datos en la realización de consultas cuando se tienen grandes volúmenes de datos.

La creación de índices es posible a través del comando SQL `CREATE INDEX` cuya sintaxis general es la siguiente:

```
CREATE [UNIQUE] INDEX nombre_indice ON nombre_tabla(col1 [, col2 ..]);
```

Continuando con el sistema de base de datos para la vinatería, se ha pedido la generación de reportes de ventas por clientes, en donde es crítico el tiempo de respuesta, así pues, la generación de un índice para el campo de nombre de la tabla *cliente* será fundamental para optimizar la velocidad de las consultas.

La instrucción SQL por implementar para cumplir con este objetivo, es la siguiente:

```
CREATE INDEX index_cliente_nombre ON cliente(txtNombreCliente);
```

Un tipo de índice especial es el índice **UNIQUE**, el cual restringe un campo para que no se ingrese dos veces un mismo valor, similar a la utilización de una restricción de llave primaria.

# CAPÍTULO 3

## Manipulación de datos

---

El Lenguaje para la Manipulación de los Datos (DML) es la parte de SQL que se utiliza para la inserción, eliminación y modificación de información dentro de una base de datos relacional. Dado que la información está organizada en campos, registros y tablas, es posible manipular la información en alguno de esos tres niveles.

La tabla siguiente muestra algunos comandos del Lenguaje para la Definición de Datos:

| Comando | Utilidad                                                              |
|---------|-----------------------------------------------------------------------|
| INSERT  | Permite la inserción de un registro a una tabla de la base de datos.  |
| DELETE  | Permite la eliminación de registros de una tabla de la base de datos. |
| UPDATE  | Permite la actualización de información en la base de datos.          |

Este capítulo, por tanto, se avocará a determinar las instrucciones SQL necesarias para la inserción, eliminación y modificación de información dentro de una base de datos relacional, se interaccionará con el manejador de base de datos MySQL y se utilizará un caso de estudio base para la aplicación práctica de SQL.

### 3.1 Inserción de información

Una vez que se ha constituido la estructura de una base de datos y se han implementado las tablas en el manejador de base de datos, el paso siguiente es el almacenamiento de la información, para ello SQL provee a los RDBMS un comando que hará esta tarea; el comando **INSERT**, entonces, será el encargado de insertar nuevos registros en las tablas de la base de datos.

La estructura del comando **INSERT** es la siguiente:

```
INSERT INTO nombre_tabla(campo1, campo2, ..., campoN)
VALUES(valor1, valor2, ..., valorN);
```

En donde el valor1 corresponde al campo1, el valor2 corresponde al campo2 y así sucesivamente.

Tomando como base el capítulo anterior donde se crearon las estructuras persistentes para la base de datos de la vinatería, ahora veremos algunos ejemplos del comando `INSERT`, para ingresar información a la base de datos.

Recuerde que las tablas de la base de datos *vinatería* son las siguientes:

| Tables_in_vinateria |
|---------------------|
| cliente             |
| procedencia         |
| vendedores          |
| ventas              |
| vinos               |

Comenzaremos trabajando con la tabla de procedencia que como recordará, servirá de catálogo de los lugares de procedencia de los vinos que vende la vinatería.

La estructura de la tabla procedencia es la siguiente:

| Field               | Type        | Null | Key | Default | Extra |
|---------------------|-------------|------|-----|---------|-------|
| txtClaveProcedencia | char(3)     | NO   | PRI |         |       |
| TxtDescripcion      | varchar(50) | NO   |     |         |       |

Insertar un nuevo registro a la tabla de procedencia, lo haremos con la instrucción SQL (DML) `INSERT`, con la estructura antes mencionada, con lo cual queda un ejemplo de la siguiente manera:

```
INSERT INTO procedencia (txtClaveProcedencia, txtDescripcion)
VALUES ('ESP', 'España');
```

La instrucción anterior permite agregar un nuevo registro a la tabla procedencia asignando el valor ESP al campo de clave de procedencia, además de definir España en el campo de descripción. Es importante resaltar que cuando se manejen valores para tipos de dato carácter, carácter variable, texto o fecha, éstos deberán especificarse entre comillas simples (') o dobles ("), de lo contrario el RDBMS contestará con un error; observe bien el comando anterior donde se especifican los valores 'ESP' y 'España'.

Para verificar que la información ya se encuentra contenida en la tabla, nos adelantaremos un poco y ejecutaremos la instrucción más poderosa del SQL, la instrucción `SELECT`, la cual nos va a permitir recuperar información de una o más tablas, por ahora, ejecutaremos dicha instrucción en su forma más sencilla:

```
SELECT * FROM nombre_tabla
```

La instrucción anterior permite seleccionar todos los campos (\*) de la tabla que indiquemos; para verificar que se han insertado los datos en la tabla de procedencia, ejecutaremos la siguiente instrucción:

```
SELECT * FROM procedencia;
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |

A partir de ahora, cuando queramos ver la información de una tabla, utilizaremos la instrucción **SELECT**, tal como se acaba de describir anteriormente.

Una variante del comando **INSERT** es cuando se omite el nombre de los campos por insertar; esto es posible ya que el RDBMS conoce los campos que contiene la tabla especificada, la única restricción es ingresar los valores en el orden en que fueron definidos los campos en la construcción de la tabla, observe el siguiente ejemplo:

```
INSERT INTO procedencia VALUES('FRA','Francia');
```

La inserción de dicho registro se hace sin ningún problema, ya que los valores coinciden con las columnas que fueron definidas en la construcción de la tabla; cuando se desee utilizar un orden diferente para ingresar los valores, deberán especificarse también los nombres de las columnas, como en la primera variante de **INSERT**. Por lo tanto, la siguiente instrucción regresaría un mensaje de error:

```
INSERT INTO procedencia VALUES('Francia', 'FRA');
```

Como parte de los ejercicios propuestos en este libro, le sugerimos realice las instrucciones SQL necesarias para ingresar la siguiente información en la tabla *procedencia*:

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |

A través de la instrucción SQL DML **INSERT**, es posible asignar los valores por defecto que se especificaron en la definición de la estructura de las tablas creadas; así mismo, se comprueba el cumplimiento de las reglas de integridad especificadas, ya sean restricciones de llave primaria, foránea o cualquier otra definida. Cuando la instrucción SQL **INSERT** no cumpla con las condiciones especificadas, el RDBMS regresará un error.

Cuando se desee insertar un valor por defecto o un nulo se hará de la siguiente manera:

```
INSERT INTO nombre_tabla VALUES(DEFAULT, NULL, 'cadena', 12);
```

En la lista de valores de la instrucción anterior, observe que para ingresar un valor por defecto se deberá utilizar la palabra reservada **DEFAULT** (sin entrecomillado), y cuando se desee dejar un valor Nulo, utilizaremos la palabra reservada **NULL**.

A continuación ingresaremos un nuevo registro en la tabla *vinos* para ejemplificar el valor por defecto que tiene la columna clave de procedencia. La estructura de la tabla *vinos* es la siguiente:

| Field               | Type          | Null | Key | Default | Extra |
|---------------------|---------------|------|-----|---------|-------|
| idVino              | int(11)       | NO   | PRI |         |       |
| TxtNombreVino       | varchar(50)   | NO   |     |         |       |
| TxtClaveProcedencia | char(3)       | NO   |     | ESP     |       |
| curPrecio           | decimal(10,2) | NO   |     |         |       |

```
INSERT INTO vinos (idVino, txtNombreVino, TxtClaveProcedencia, curPrecio)
VALUES(1, 'Viña Lanciano Reserva 1998', DEFAULT, 483);
```

El resultado invariablemente es un nuevo registro en la tabla *vinos*, observe que el valor para el campo Clave de Procedencia resultó ser ESP, aun cuando éste no fue especificado en la instrucción **INSERT**, esto es debido a que se especificó un valor por defecto en la definición de la tabla y el comando **INSERT** únicamente invocó el valor por defecto con la palabra reservada **DEFAULT**.

Otros ejemplos son:

```
INSERT INTO vinos (idVino, txtNombreVino, TxtClaveProcedencia, curPrecio)
VALUES(2, 'Valtravieso Crianza 2002', 'ESP', 325);
INSERT INTO vinos (idVino, txtNombreVino, TxtClaveProcedencia, curPrecio)
VALUES(3, 'Clos Vougeot 2000', 'FRA', 1246);
```

La definición de la tabla *vinos* incluye una restricción de llave foránea en el campo *TxtClaveProcedencia*, lo que hace que para poder insertar un nuevo vino, la clave especificada en el campo *TxtClaveProcedencia*, debe forzosamente existir en la tabla *procedencia*, ya que de ello depende el cumplimiento de la integridad referencial.

*procedencia*

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |

*vinos*

| idVino | TxtNombreVino              | TxtClaveProcedencia | curPrecio |
|--------|----------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998 | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002   | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000          | FRA                 | 1246.00   |

Suponga que desea insertar un nuevo vino, cuya clave de procedencia sea ARG, el RDBMS entonces se verá obligado a buscar esa clave de procedencia en la tabla *procedencia* con el objeto de garantizar la integridad referencial en la base de datos, como no existe hasta el momento la clave ARG en la tabla *procedencia*, entonces, el RDBMS regresa un mensaje de error indicando que hay una restricción de llave primaria y foránea en la tabla *vinos*, y que, por lo tanto, es imposible agregar el nuevo registro.

Ejecute las instrucciones SQL necesarias para obtener el siguiente resultado en cada una de las tablas:

*Procedencia*

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |

## Vinos

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 8      | Señorio de Garcigrande, Verdejo | ESP                 | 72.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |

## Clientes

| idCliente | TxtNombreCliente                     |
|-----------|--------------------------------------|
| 401       | Vinos y Licores del Oeste            |
| 402       | El rincón de la Luna Restaurant      |
| 403       | Comercializadora Emporio de Vinicola |
| 404       | San Carlos Restaurant                |
| 405       | Productos vinicolas Esfinge          |

## Vendedores

| idVendedor | txtNombreVendedor       |
|------------|-------------------------|
| 301        | Fernando Gómez Galicia  |
| 302        | Jennifer Aguilar Cortés |
| 303        | Dario González Henkel   |
| 304        | Carlos Martínez Salinas |
| 305        | Arabia Leñero Orozco    |

## Ventas

| TxtClaveVentas | idVino | idCliente | idVendedor | intCantidad | curMonto | dateFechaVenta      |
|----------------|--------|-----------|------------|-------------|----------|---------------------|
| FACT0360       | 7      | 401       | 301        | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 402       | 305        | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 405       | 304        | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 401       | 302        | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0364       | 10     | 404       | 303        | 4           | 288.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 402       | 302        | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 403       | 302        | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 401       | 303        | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 403       | 303        | 9           | 5310.00  | NULL                |
| FACT0369       | 1      | 402       | 301        | 1           | 483.00   | NULL                |
| FACT0370       | 6      | 401       | 302        | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 401       | 305        | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 403       | 304        | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 403       | 302        | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 404       | 301        | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 402       | 301        | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 405       | 302        | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 402       | 304        | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 402       | 302        | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 402       | 303        | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 403       | 305        | 6           | 2994.00  | 2007-06-09 00:00:00 |

En los temas siguientes, será necesario tener la estructura de las tablas anteriores, con los mismos registros incluso, ya que, servirán de base para ejemplificar la utilización de otros comandos SQL como UPDATE, DELETE y SELECT.

### 3.2 Eliminación de registros

La eliminación de registros no es una actividad común, ya que una base de datos almacena información histórica, principalmente; entonces eliminar un registro significa perder historia y a futuro conocimiento ya que cuando se cuenta con grandes volúmenes de datos, es posible realizar diversos análisis que permitan la retroalimentación a la organización.

Cuando requiera eliminar registros de alguna tabla, la instrucción SQL DML por utilizar será DELETE, cuya estructura básica es la siguiente:

```
DELETE FROM nombre_tabla WHERE condición
```

Para eliminar un registro, debemos colocar las palabras reservadas DELETE FROM, seguidas del nombre de la tabla de donde se eliminarán los registros y, posteriormente, se deben restringir los registros que se quieren eliminar a través de la palabra reservada WHERE y una condicionante.

El uso de una instrucción DELETE debe hacerse mesuradamente ya que si no se especifica cuál es la condición de los registros por eliminar a través de la palabra reservada WHERE, entonces, se eliminarán todos los registros de la tabla.

Suponga que tiene una tabla llama productos con el siguiente contenido:

| intNumProducto | txtDescripcion       |
|----------------|----------------------|
| 600            | SONY VAIO            |
| 601            | Compac Presario 1500 |
| 602            | Hp Inspiron          |
| 603            | DELL Dimension 4500  |

Para comprobar lo antes mencionado, ejecutaremos el comando DELETE sin especificar una restricción, con lo cual la instrucción SQL queda de la siguiente manera:

```
DELETE FROM productos;
```

El resultado obtenido es la eliminación de todos los registros de la tabla, con lo cual queda vacía; si no se restringen los registros que se quieren eliminar, por defecto, se eliminan todos los contenidos en la tabla, he ahí la importancia de utilizar un condicionante a través de la palabra reservada WHERE.

La restricción de registros por eliminar debe pensarse muy bien, de tal forma, que se encuentre el criterio en donde sólo coincidan los registros que se desean eliminar, primero que nada, hay que contemplar la llave primaria, en tanto ésta identifica de manera única cada registro en una tabla, y entonces también, sirve como mecanismo para asegurarnos de eliminar un registro en particular, de igual forma sucede con una llave foránea.

Para conformar una restricción, haremos uso de la palabra reservada **WHERE** seguida de nombres de campo que cumplan con una condición, por ejemplo:

```
DELETE FROM tabla WHERE condición;  
DELETE FROM productos WHERE intNumProducto=603;
```

La restricción hecha en el comando anterior, indica que se eliminarán de la tabla *productos* los registros donde el valor de la columna *intNumProducto* sea igual a 603; dado que sólo un registro cumple con dicha condición, entonces únicamente se elimina éste.

| intNumProducto | txtDescripcion       |
|----------------|----------------------|
| 600            | SONY VAIO            |
| 601            | Compac Presario 1500 |
| 602            | Hp Inspiron          |

Existen variaciones en la forma de especificar condicionantes con la palabra reservada **WHERE**; sin embargo, este tema se aborda más adelante cuando se hable del comando **SELECT**, ahí se hará una revisión más profunda respecto de la especificación de criterios para formular condicionantes, por ende, sería idóneo que una vez revisado dicho tema, se relea el contenido de éste para lograr una mejor comprensión sobre cómo se debe restringir la información por eliminar o actualizar, que es objeto de estudio en el tema siguiente.

### 3.3 Actualización de información

La actualización de información es una tarea recurrente en bases de datos, ya que el objetivo principal de dicha tarea es mantener la consistencia en los datos, así como mantener la integridad de los mismos. La actualización de la información es requerida principalmente, por que se han cometido errores en la captura de los datos, o bien, por que con el paso del tiempo, algún dato almacenado requiere ser actualizado con el objeto de conocer el valor actual del mismo.

Cuando requiera actualizar información de alguna tabla, la instrucción SQL DML por utilizar será **UPDATE**, cuya estructura básica es la siguiente:

```
UPDATE nombre_tabla SET campo1=nuevo_valor1, campo2=nuevo_valor2 WHERE  
condición
```

Para actualizar el valor de un campo de un registro en la base de datos, se utilizará la palabra reservada **UPDATE** seguida del nombre de la tabla y posteriormente se indicará el nuevo valor del campo con ayuda de la palabra reservada **SET**, como último paso, se condicionarán los registros en donde se actualizarán los valores para los campos antes indicados.

Por ejemplo, considerando que se cuenta con la siguiente tabla, se desea actualizar el producto 601, ya que se ha capturado de manera errónea su nombre, en tanto el nombre correcto de éste es: "Compaq Presario 2700".

| intNumProducto | txtDescripcion       |
|----------------|----------------------|
| 600            | SONY VAIO            |
| 601            | Compac Presario 1500 |
| 602            | Hp Inspiron          |

Para conseguir actualizar el nombre del producto 601, se utilizará la instrucción SQL DML UPDATE, con lo cual dicha instrucción queda de la siguiente manera:

```
UPDATE productos SET txtDescripcion='Compaq Presario 2007'
WHERE intNumProducto=601;
```

La instrucción SQL anterior actualiza el campo `txtDescripcion` de la tabla `productos` asignando el valor de "Compaq Presario 2007" cuando la clave del producto es 601.

El resultado, después de realizar una consulta, es el siguiente:

| intNumProducto | txtDescripcion       |
|----------------|----------------------|
| 600            | SONY VAIO            |
| 601            | Compaq Presario 2007 |
| 602            | Hp Inspiron          |

De igual forma que con la instrucción DELETE, el uso de una instrucción UPDATE debe hacerse mesuradamente, ya que si no se especifica cuál es la condición de los registros por actualizar a través de la palabra reservada WHERE, entonces, el RDBMS actualizará todos los registros de la tabla. Por ejemplo, el resultado de ejecutar la instrucción sin el condicionante WHERE, sería el siguiente:

| intNumProducto | txtDescripcion       |
|----------------|----------------------|
| 600            | Compaq Presario 2007 |
| 601            | Compaq Presario 2007 |
| 602            | Compaq Presario 2007 |

No restringir los registros que se quieren actualizar o eliminar puede generar grandes cantidades de inconsistencia, integridad y/o redundancia en la base de datos; por ello, la importancia de la restricción de registros por actualizar o eliminar cuando se ejecuten las instrucciones SQL pertinentes.

La restricción de registros por utilizar para la eliminación o actualización de información se verá más a fondo en temas posteriores ya que como se trata de un tema de gran importancia, se contempla un tema en particular para su exploración y entendimiento; por ahora, sólo basta conocer que para realizar una operación SQL DML de actualización o eliminación de datos, se debe restringir el universo de datos que se quieren manipular con el objeto de asegurar la integridad y consistencia de la información.

# CAPÍTULO 4

## Selección de datos

---

Sí bien una base de datos cumple con el objetivo primordial de asegurar la persistencia de la información, también se deben generar mecanismos para la manipulación y consulta de los datos almacenados. La instrucción SQL **SELECT** es la encargada de permitir la recuperación de información almacenada en una base de datos. El **SELECT**, es el operador más poderoso en un RDBMS, ya que será el encargado de presentar al usuario, la información almacenada en la base de datos.

### 4.1 Selección de información

El **SELECT** tiene su fundamento en el álgebra y cálculo relacional descritos por el Dr. Edgar Codd, cuando hace la publicación del modelo relacional de base de datos.

La estructura del comando **SELECT** es muy amplia y variada, por lo que se desglosará la misma en subtemas, con lo cual se facilita su comprensión y alcance. Así pues, la estructura básica de una instrucción **SELECT** es la siguiente:

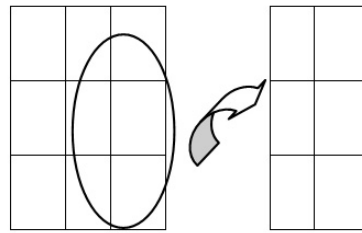
```
SELECT * FROM nombre_tabla;  
SELECT tabla.columna, tabla.columna2, ..., tabla.columnaN FROM tabla;
```

El \* (asterisco) es un operador lógico que funge como comodín, que señala la manera en que se mostrarán todos los campos de la(s) tabla(s) indicada(s).

Con el objeto de presentar ejemplos, retomaremos el caso de estudio de la vinatería, en dónde ya se han creado las estructuras persistentes, así como insertado datos en las mismas.

Las dos más grandes funcionalidades de una instrucción **SELECT** son: la proyección y la restricción, ambas fundamentadas en el álgebra relacional cuya aplicación en el lenguaje SQL se hace de la siguiente manera:

Una proyección, obtiene un subconjunto de atributos en una relación, hablando en términos de un RDBMS, una proyección permitirá mostrar un subconjunto de columnas de una tabla.



Proyección de una tabla.

Para ejemplificar una proyección tomaremos como base la tabla vinos, de la base de datos de la vinatería. La cuál contiene la siguiente información:

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |

Una proyección para la tabla *vinos* se hace a través de la siguiente instrucción SQL:

```
SELECT vinos.idVino, vinos.txtNombreVino, curPrecio FROM vinos;9
```

Lo que hace dicha instrucción es proyectar ciertas columnas de la tabla en cuestión, dejando otras fuera de la información por presentar, así es como obtenemos el siguiente resultado:

| idVino | txtNombreVino                   | curPrecio |
|--------|---------------------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | 483.00    |
| 2      | Valtravieso Crianza 2002        | 325.00    |
| 3      | Clos Vougeot 2000               | 1246.00   |
| 4      | Valtravieso Crianza 2002        | 325.00    |
| 5      | Valtravieso Reserva 1999        | 590.00    |
| 6      | Chablis Cuvée LC 1999           | 79.00     |
| 7      | Nadal 1510 1998                 | 499.00    |
| 8      | Señorío de Garcigrande, Verdejo | 72.00     |
| 9      | Bourgogne Vezelay 2001          | 155.00    |
| 10     | Finca Boadella 2000             | 72.00     |
| 11     | Saint Véran La Solitude 2001    | 220.00    |
| 12     | Pouilly Fuissé 2001             | 305.00    |

<sup>9</sup> Es posible omitir el nombre de la tabla cuando se hace referencia a una columna, esto siempre y cuando no se esté trabajando con un campo de otra tabla que se llame igual.

Si deseáramos conocer sólo los vinos que se tienen en catálogo, entonces utilizaríamos una proyección de la siguiente manera:

```
SELECT idVino, txtNombreVino FROM vinos;
```

| idVino | txtNombreVino                   |
|--------|---------------------------------|
| 1      | Viña Lanciano Reserva 1998      |
| 2      | Valtravieso Crianza 2002        |
| 3      | Clos Vougeot 2000               |
| 4      | Valtravieso Crianza 2002        |
| 5      | Valtravieso Reserva 1999        |
| 6      | Chablis Cuvée LC 1999           |
| 7      | Nadal 1510 1998                 |
| 8      | Señorío de Garcigrande, Verdejo |
| 9      | Bourgogne Vezelay 2001          |
| 10     | Finca Boadella 2000             |
| 11     | Saint Véran La Solitude 2001    |
| 12     | Pouilly Fuissé 2001             |

El resultado de dicha consulta es la proyección de las columnas id y nombre del vino, provenientes de la tabla *vinos*.

La proyección es sin duda el tipo de consulta más simple de generar con SQL, ya que únicamente deben listarse las columnas que deseamos sean mostradas. A su vez, cuando se quieren proyectar todas las columnas de una tabla, es posible utilizar un comodín, el asterisco (\*) provee la funcionalidad de indicar que se proyectarán todos los atributos de una tabla, tal como se muestra en el siguiente ejemplo:

```
SELECT * FROM cliente;
```

Dicha instrucción proyectará todas las columnas de la tabla *cliente*, así lo muestra la salida de información proporcionada por el RDBMS.

| idCliente | TxtNombreCliente                     |
|-----------|--------------------------------------|
| 401       | Vinos y Licores del Oeste            |
| 402       | El rincón de la Luna Restaurant      |
| 403       | Comercializadora Emporio de Vinicola |
| 404       | San Carlos Restaurant                |
| 405       | Productos vinícolas Esfinge          |

Nótese que en la ejecución de todas las instrucciones SQL **SELECT** anteriores, aparece siempre en el resultado todos los registros contenidos en las tablas, ello debido a que, la proyección únicamente delimita las columnas que se mostrarán al momento de presentar la información.

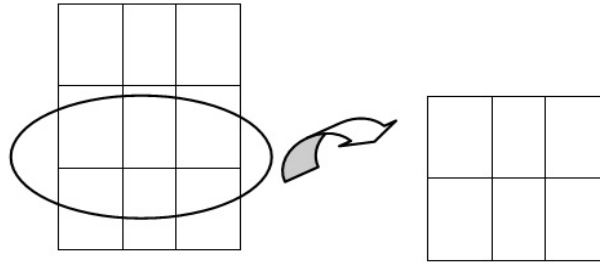
La siguiente gran funcionalidad que permite el operador **SELECT** es la restricción, la cual obtiene un subconjunto de registros de una relación, dicha funcionalidad se implementa colocando la palabra reservada **WHERE** dentro de la instrucción **SELECT**.

La restricción de información, no sólo es utilizada al momento de realizar consultas, sino que, también es de suma importancia al momento de realizar una actualización o eliminación de información de la base de datos, tal como se trató en capítulos anteriores;

ahora se abordará el tema de especificación de criterios de selección de manera más profunda.

## 4.2 Especificación de criterios de selección de información

Como bien hemos mencionado, una de las grandes funcionalidades que permite una instrucción **SELECT**, es la restricción de información, es decir, la obtención de un subconjunto de registros de una tabla.



Restricción de información.

La restricción de información se hace utilizando la palabra reservada **WHERE** dentro de la estructura de una instrucción **SELECT**, **DELETE**, **UPDATE** u otra, generalmente el **WHERE** se coloca al final de cualquiera de las instrucciones antes mencionadas.

La especificación de criterios se hace a través de condicionantes, lo cual hace se evalúe una tabla, registro a registro, para que el RDBMS decida si mostrará dicho registro en la presentación final de la información, o bien, si dicho registro se actualizará o eliminará.

La estructura de restricción de información en lenguaje SQL, se da de la siguiente forma:

**WHERE** tabla.columna=valor

Condición

La forma de estructurar una condición pareciera un poco rara en principio, sin embargo, desmenuzaremos sus elementos uno a uno para darle un mejor sentido y significado.

Seguido de la palabra reservada **WHERE**, se deberá indicar la condición de los registros que se desean manipular; dicha condición se compone de un nombre de columna, de un operador (**=**, **like**, **in**, **between**, etc.) y de un valor, el cual será el criterio que deba cumplir o no cumplir el registro por evaluar.

Para ejemplificar la forma en que se estructura una restricción, tomaremos como base la tabla **vinos** de la base de datos **vinatería**, que tiene la siguiente estructura e información:

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |

Supongamos que deseamos hacer una operación con los vinos cuya procedencia es Francia (FRA), para ello, debemos forzosamente hacer una restricción de registros, la condición es que sean vinos franceses, así la especificación de criterios, es la siguiente:

```
WHERE vinos.TxtClaveProcedencia='FRA'
```

La condición entonces, es que el campo de procedencia debe tener un valor igual a 'FRA'. La especificación de criterios anterior no es suficiente, debe indicarse previamente la operación que se quiere realizar, como sería una selección de información, así el ejemplo completo quedaría de la siguiente manera:

```
SELECT * FROM vinos WHERE vinos.TxtClaveProcedencia='FRA';
```

Esta instrucción proyectará todas las columnas de la tabla *vinos* y mostrará los registros que en el campo de procedencia tengan como valor FRA.

El resultado obtenido es la restricción de los registros de la tabla *vinos*, presentando únicamente los que cumplen con la condición de que el vino sea de procedencia francesa.

| idVino | TxtNombreVino                | TxtClaveProcedencia | curPrecio |
|--------|------------------------------|---------------------|-----------|
| 3      | Clos Vougeot 2000            | FRA                 | 1246.00   |
| 6      | Chablis Cuvée LC 1999        | FRA                 | 79.00     |
| 9      | Bourgogne Vezelay 2001       | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001 | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001          | FRA                 | 305.00    |

La misma restricción puede utilizarse en otros ámbitos como para la eliminación de registros de una tabla, tal como se muestra con la siguiente instrucción SQL.

```
DELETE FROM vinos WHERE vinos.TxtClaveProcedencia='FRA';
```

O bien para la actualización de datos:

```
UPDATE vinos SET curPrecio=100 WHERE vinos.TxtClaveProcedencia='FRA';
```

En las dos instrucciones anteriores, se eliminan o actualizan los registros que cumplen con la condición de ser vinos de procedencia francesa.

A su vez, también es posible restringir la información por utilizar mediante dos o más condiciones, para ello, es necesario el apoyo de los dos operadores lógicos más fundamentales, el **AND** y el **OR**. Suponga que quiere conocer cuáles son los vinos de procedencia francesa con precio mayor a los 300 pesos, hasta ahora, sólo había podido

restringir por medio de una condición, pero cuando se desea que la información por manipular cumpla con dos condiciones, entonces, utilizaremos la siguiente estructura:

WHERE tabla.columna=valor [AND | OR] tabla.columna=valor [AND | OR] ... <sup>10</sup>



La restricción de la información quedaría de la siguiente manera:

WHERE txtClaveProcedencia='FRA' AND CurPrecio>=300;<sup>11</sup>

Y poniendo en práctica esta restricción de información en una instrucción SELECT, el efecto es el siguiente:

```
SELECT * FROM vinos
WHERE txtClaveProcedencia='FRA' AND CurPrecio>=300;
```

El operador AND provee la potencialidad necesaria para evaluar dos condiciones, tanto que el vino por mostrar sea francés y que además, su precio sea mayor a 300 pesos.

| idVino | TxtNombreVino       | TxtClaveProcedencia | curPrecio |
|--------|---------------------|---------------------|-----------|
| 3      | Clos Vougeot 2000   | FRA                 | 1246.00   |
| 12     | Pouilly Fuissé 2001 | FRA                 | 305.00    |

Ahora revisemos qué pasaría si cambiásemos el operador lógico y en vez de usar un AND usáramos un OR, muy simple se evalúa que el vino sea francés o que su precio sea mayor a 300 pesos; al decir, o que su precio sea mayor a 300 pesos, estamos diciendo que no importa que se cumpla la primera condición, siempre y cuando se cumpla alguna de las dos, se mostrará el registro.

```
SELECT * FROM vinos
WHERE txtClaveProcedencia='FRA' OR CurPrecio>=300;
```

| idVino | TxtNombreVino                | TxtClaveProcedencia | curPrecio |
|--------|------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998   | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002     | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000            | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002     | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999     | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999        | FRA                 | 79.00     |
| 7      | Nadal 1510 1998              | ESP                 | 499.00    |
| 9      | Bourgogne Vezelay 2001       | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001 | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001          | FRA                 | 305.00    |

<sup>10</sup> Cuando compare dos valores, éstos deben ser del mismo tipo de dato, ya que hacer una comparación sobre diferentes tipos de datos puede llegar a generar errores al momento de evaluar la expresión o condicionante.

<sup>11</sup> Recuerde que cuando se trabaja con datos tipo Carácter, éstos deben colocarse entre comillas simples ('), no así para los valores de datos tipo Numérico.

Cuando se evalúa más de una condición, se deberán utilizar los operadores lógicos AND y OR, los cuales pueden llegar a ser excluyentes, ya que el resultado de evaluar una condición es un valor booleano y para evitar un efecto de exclusión o inclusión tome en consideración la tabla siguiente.

| Condición1 | OPERADOR | Condición2 | Resultado |
|------------|----------|------------|-----------|
| verdadero  | AND      | verdadero  | verdadero |
| verdadero  | AND      | falso      | falso     |
| falso      | AND      | verdadero  | falso     |
| falso      | AND      | falso      | falso     |
| verdadero  | OR       | verdadero  | verdadero |
| verdadero  | OR       | falso      | verdadero |
| falso      | OR       | verdadero  | verdadero |
| falso      | OR       | falso      | falso     |

Es correcto mencionar que se pueden agregar tantas condicionantes como se deseen, siempre y cuando se haga uso de los operadores lógicos AND y OR para su correcta evaluación.

Otros ejemplos de consultas de restricción son los siguientes:

Mostrar las ventas en donde se vendieron cinco productos o más:

```
SELECT TxtClaveVentas, intCantidad FROM ventas
WHERE intCantidad>=5
```

```
+-----+-----+
| TxtClaveVentas | intCantidad |
+-----+-----+
| FACT0365       | 6          |
| FACT0366       | 7          |
| FACT0367       | 8          |
| FACT0368       | 9          |
| FACT0372       | 5          |
| FACT0373       | 6          |
| FACT0377       | 8          |
| FACT0380       | 6          |
+-----+-----+
```

En el caso de la consulta anterior, se hace una mezcla interesante entre la proyección de atributos y restricción de registros, esta tarea de proyectar y restringir es muy común al momento de realizar consultas a una base de datos, ya que no siempre se desea mostrar toda la información o no es posible acceder a toda ella.

Cuando se hace una restricción en una consulta, no es obligatorio que se muestre el campo que contiene el valor por evaluar, por ejemplo, se desea conocer cuáles fueron las ventas en dónde se vendieron más de cinco productos y si el monto total de la venta fue mayor o igual a los 3,000 pesos.

```
SELECT TxtClaveVentas, intCantidad FROM ventas
WHERE intCantidad>=5 AND curMonto>=3000;
```

```
+-----+-----+
| TxtClaveVentas | intCantidad |
+-----+-----+
| FACT0366       | 7          |
| FACT0368       | 9          |
| FACT0377       | 8          |
+-----+-----+
```

En la consulta anterior se muestra la clave de la venta y la cantidad de productos vendidos, sin embargo se sabe que en esos registros el monto es mayor a 3,000 pesos, ello debido a que se especifica que se cumpla con dicha condición.

Si proyectamos también la columna de monto, podemos darnos cuenta que el resultado anterior es confiable.

```
SELECT TxtClaveVentas, intCantidad, curMonto FROM ventas
WHERE intCantidad>=5 AND curMonto>=3000;
```

| TxtClaveVentas | intCantidad | curMonto |
|----------------|-------------|----------|
| FACT0366       | 7           | 8722.00  |
| FACT0368       | 9           | 5310.00  |
| FACT0377       | 8           | 3992.00  |

Por lo tanto, podemos concluir que no es necesaria la proyección de una columna que será utilizada para la restricción de información. Por ejemplo, si deseáramos conocer las ventas realizadas por el vendedor 301 ejecutaríamos la siguiente consulta:

```
SELECT TxtClaveVentas, intCantidad, curMonto FROM ventas
WHERE idVendedor=301;
```

| TxtClaveVentas | intCantidad | curMonto |
|----------------|-------------|----------|
| FACT0360       | 2           | 998.00   |
| FACT0369       | 1           | 483.00   |
| FACT0374       | 3           | 3738.00  |
| FACT0375       | 4           | 1300.00  |

Ahora bien, una parte fundamental en la creación de condicionantes, es la flexibilidad de criterios por evaluar en donde no sólo se puede utilizar un criterio de igualdad, sino que es posible escoger de una gran gama de operadores lógicos, matemáticos o comodines.

### 4.3 Operadores SQL

Hacer una restricción de información, no siempre involucra crear condiciones de igualdad o donde algo sea mayor o menor que otra cosa; también en la consulta de información, se requiere evaluar texto, fechas y valores numéricos, por lo que el criterio de selección de información debe apoyarse en un operador que satisfaga las necesidades para la creación de una expresión por evaluar.

Los dos primeros operadores SQL de tipo lógico revisados en este capítulo fueron AND y OR, los cuales permiten evaluar un conjunto de expresiones o condicionantes; pero también, existen otro tipo de operadores que servirán para formar otro tipo de expresiones.

| Operador | Significado       |
|----------|-------------------|
| =        | Igualdad          |
| <>       | Diferencia        |
| >        | Mayor que         |
| >=       | Mayor o igual que |
| <        | Menor que         |
| <=       | Menor o igual que |

| Operador | Significado                 |
|----------|-----------------------------|
| LIKE     | Como                        |
| BETWEEN  | Entre                       |
| IN       | En (elementos de una lista) |
| IS NULL  | Es nulo                     |

#### **4.3.1 Operadores para el manejo de datos tipo numérico.**

El operador de igualdad (=) permite seleccionar las filas de una tabla donde una de las columnas contiene un valor igual al especificado.

```
SELECT TxtClaveVentas, intCantidad FROM ventas  
WHERE intCantidad=5;
```

```
+-----+-----+  
| TxtClaveVentas | intCantidad |  
+-----+-----+  
| FACT0372      |          5 |  
+-----+-----+
```

Para el ejemplo anterior, se pide al RDBMS mostrar de la tabla de ventas todas las filas en donde coincida que la cantidad de productos sea igual a cinco.

El operador de diferencia excluye todas aquellas filas que coincidan con la especificación hecha por el usuario, tal es el caso del siguiente ejemplo donde se pide se muestren todas las filas de la tabla ventas, y la cantidad de productos sea diferente a cinco.

```
SELECT TxtClaveVentas, intCantidad FROM ventas  
WHERE intCantidad<>5;
```

```
+-----+-----+  
| TxtClaveVentas | intCantidad |  
+-----+-----+  
| FACT0360      |          2 |  
| FACT0361      |          1 |  
| FACT0362      |          3 |  
| FACT0363      |          1 |  
| FACT0364      |          4 |  
| FACT0365      |          6 |  
| FACT0366      |          7 |  
| FACT0367      |          8 |  
| FACT0368      |          9 |  
| FACT0369      |          1 |  
| FACT0370      |          2 |  
| FACT0371      |          4 |  
| FACT0373      |          6 |  
| FACT0374      |          3 |  
| FACT0375      |          4 |  
| FACT0376      |          1 |  
| FACT0377      |          8 |  
| FACT0378      |          4 |  
| FACT0379      |          2 |  
| FACT0380      |          6 |  
+-----+-----+
```

El operador mayor que (>) evalúa que el término a la derecha sea mayor que el término de la izquierda, por ejemplo, si quisiéramos saber cuáles son las facturas con más de cinco productos registrados, entonces, ejecutaríamos la siguiente instrucción SQL SELECT.

```
SELECT TxtClaveVentas, intCantidad FROM ventas  
WHERE intCantidad>5;
```

| TxtClaveVentas | intCantidad |
|----------------|-------------|
| FACT0365       | 6           |
| FACT0366       | 7           |
| FACT0367       | 8           |
| FACT0368       | 9           |
| FACT0373       | 6           |
| FACT0377       | 8           |
| FACT0380       | 6           |

A su vez, el operador mayor o igual que ( $\geq$ ) evalúa que el término de la derecha sea mayor o igual que el término a la izquierda de la expresión, cuando se cumple esta condición, entonces se muestra la fila de la tabla indicada.

```
SELECT TxtClaveVentas, intCantidad FROM ventas
WHERE intCantidad>=5;
```

| TxtClaveVentas | intCantidad |
|----------------|-------------|
| FACT0365       | 6           |
| FACT0366       | 7           |
| FACT0367       | 8           |
| FACT0368       | 9           |
| FACT0372       | 5           |
| FACT0373       | 6           |
| FACT0377       | 8           |
| FACT0380       | 6           |

Si observa con detenimiento los resultados de las dos instrucciones anteriores en donde se utilizan los operadores  $>$  y  $\geq$ , se podrá dar cuenta que son similares los resultados; sin embargo al utilizar el operador  $\geq$ , se obtiene una fila más, ya que la Factura 0372 tiene cinco artículos y ello hace verdadera la condición  $\geq 5$ .

La misma situación pero a la inversa, ocurre con los operadores menor que y menor o igual que ( $<$ ,  $\leq$ ), observe las siguientes instrucciones SQL.

```
SELECT TxtClaveVentas, intCantidad FROM ventas
WHERE intCantidad<5;
```

| TxtClaveVentas | intCantidad |
|----------------|-------------|
| FACT0360       | 2           |
| FACT0361       | 1           |
| FACT0362       | 3           |
| FACT0363       | 1           |
| FACT0364       | 4           |
| FACT0369       | 1           |
| FACT0370       | 2           |
| FACT0371       | 4           |
| FACT0374       | 3           |
| FACT0375       | 4           |
| FACT0376       | 1           |
| FACT0378       | 4           |
| FACT0379       | 2           |

```
SELECT TxtClaveVentas, intCantidad FROM ventas
WHERE intCantidad<=5;
```

| TxtClaveVentas | intCantidad |
|----------------|-------------|
| FACT0360       | 2           |
| FACT0361       | 1           |
| FACT0362       | 3           |
| FACT0363       | 1           |
| FACT0364       | 4           |
| FACT0369       | 1           |
| FACT0370       | 2           |
| FACT0371       | 4           |
| FACT0372       | 5           |
| FACT0374       | 3           |
| FACT0375       | 4           |
| FACT0376       | 1           |
| FACT0378       | 4           |
| FACT0379       | 2           |

Los operadores =, >, >=, <, <= y <> tienen como finalidad principal evaluar términos numéricos, a los cuales brindan su mayor potencialidad; sin embargo, es posible utilizarlos dentro de expresiones cuyos términos son de tipo Carácter o Fecha, observe los siguientes ejemplos.

#### 4.3.2 Operadores para el manejo de datos tipo caracter

Comenzaremos con el operador de igualdad para el caso de su uso con tipos de dato Carácter, donde es de suma importancia hacer notar que al hacer uso de este operador se activa una sensibilidad a mayúsculas o minúsculas al momento de evaluar la expresión. Tomando como base la tabla de procedencia, realizaremos los ejemplos pertinentes:

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |

Suponga desea seleccionar las filas en que la procedencia sea España, para ello, utilizaría el operador de igualdad, en la expresión parte del WHERE de la instrucción SQL, por lo cual ejecuta la siguiente instrucción.

```
SELECT * FROM Procedencia
WHERE TxtDescripcion='España';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |

Como podemos observar, el resultado realmente se ha cumplido, no obstante, podemos preguntarnos qué ocurre cuando se escribe la palabra España, con todas las letras en altas; claro, como lo dijimos anteriormente, el RDBMS no regresa ninguna fila ya que España no es igual a ESPAÑA, la explicación es simple, cada carácter tiene un equivalente en código ASCII, de esta manera si se compara la 'a' con la 'A' se obtienen valores diferentes, por tal razón el RDBMS determina una diferencia entre ambas cadenas de texto.

```
SELECT * FROM Procedencia
WHERE TxtDescripcion='ESPAÑA';
Empty set (0.00 sec)
```

La configuración del RDBMS puede cambiarse en cualquier momento para que la sensibilidad entre mayúsculas y minúsculas sea admitida o anulada, en tal caso, la siguiente instrucción SQL SELECT regresa una fila:

```
SELECT * FROM Procedencia
WHERE TxtDescripcion='FRANCIA';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| FRA                 | Francia        |

Así mismo los operadores >, >=, <, <= también pueden ser utilizados con valores para datos tipo Carácter, ello, tomando como base la premisa de que todo carácter tiene una equivalencia numérica en código ASCII, por ejemplo si se evalúa la expresión A>Z, el resultado es falso ya que una A tiene menor valor que una Z, si se evalúa A<Z el resultado es verdadero, incluso existe la posibilidad de evaluar A>a en donde se obtiene un valor verdadero. Dentro de una instrucción SQL los ejemplos pertinentes son los siguientes:

```
SELECT * FROM Procedencia
WHERE TxtDescripcion > 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ITA                 | Italia         |

En este caso, Italia posee un valor mayor que Francia; los valores España y Australia no cumplen con la condición ya que son valores menores, vea el siguiente ejemplo:

```
SELECT * FROM Procedencia
WHERE TxtDescripcion < 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| AUS                 | Australia      |

Ahora el resultado consta de dos filas, aquí España y Australia son menores que Francia, si se tomase como operador >=, o <=, entonces Francia sería un valor más de coincidencia.

```
SELECT * FROM Procedencia
WHERE TxtDescripcion >= 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| FRA                 | Francia        |
| ITA                 | Italia         |

```
SELECT * FROM Procedencia
WHERE TxtDescripcion <= 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |

Tenga siempre presente que cuando se trabaja con valores para tipos de dato Carácter o Fecha, siempre se deben entrecomillar: 'Francia', 'España', '21/01/1985'.

### 4.3.3 Operador LIKE y comodines

Cuando se desea hacer restricciones a través de la palabra reservada **WHERE** dentro de una instrucción SQL y además se trabaje con datos de tipo Carácter, resulta más útil el operador **LIKE**, el cual va a permitir una mejor evaluación de este tipo de datos, ya que aunado a él, se pueden utilizar comodines.

El operador **LIKE** usado para datos tipo Cadena va a reflejar en parte, la funcionalidad del operador de igualdad (=), y decimos en parte, debido a que la implementación de una expresión regular permite ampliar su visión.

El operador **LIKE** puede utilizarse para restringir los lugares de procedencia en donde éste sea Francia, por ejemplo, en tal caso la instrucción SQL por utilizar es la siguiente:

```
SELECT * FROM Procedencia
WHERE TxtDescripcion LIKE 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| FRA                 | Francia        |

Como puede percatarse, el resultado es exactamente igual a cuando se utilizó el operador de igualdad, sin embargo, el operador **LIKE** tiene la posibilidad de ampliar el criterio de búsqueda.

Ejecute las instrucciones SQL necesarias para ampliar el conjunto de datos de la tabla procedencia para obtener el siguiente resultado:

```
SELECT * FROM procedencia;
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| ARG                 | Argentina      |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

Con este conjunto de datos ampliado de la tabla *procedencia*, es como haremos un mejor uso del operador **LIKE**, cuya potencialidad radica en el uso del comodín %.

El símbolo de porcentaje (%) sirve como comodín, ya que así puede incrementarse una cadena de texto, suponga que desea consultar los lugares de procedencia que

comiencen con la letra A, para ello, utilizaremos el operador LIKE y el símbolo %, tal como se muestra en el siguiente ejemplo:

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE 'A%';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| AUS                 | Australia      |
| ARG                 | Argentina      |

En este caso, el símbolo de % permite indicarle al RDBMS que la condición es que no importa el contenido completo de la cadena, basta con que ésta comience con la letra A, el porcentaje entonces sirve de comodín al indicar que seguido de la A puede ir cualquier otro conjunto de caracteres o ninguno.

```
SELECT * FROM procedencia WHERE TxtDescripcion LIKE 'Argentina%';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ARG                 | Argentina      |

A su vez, el comodín % puede utilizarse al principio de la cadena para indicar que no importa cuál sea el principio de la cadena, siempre y cuando el final sea como se indique, ejemplo:

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE '%ia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| FIN                 | Finlandia      |

La consulta anterior regresa como resultado, todas las filas en donde el lugar de procedencia termina en 'ia', siendo el operador de porcentaje (%) el responsable de esta acción, ya que significa uno o más caracteres.

También es posible el uso de símbolos de porcentaje al principio y al final de una cadena para especificar que no importa el principio ni el final, sino que una parte del contenido se encuentre dentro del texto por buscar, como, buscar en una cadena de texto, la letra 'L'.

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE '%l%';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| AUS                 | Australia      |
| ITA                 | Italia         |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

En la consulta se especifica que no importa el conjunto de caracteres al principio, ni al final, siempre y cuando exista la letra 'L' en la cadena. La letra 'L', incluso puede estar contenida al principio y/o al final, observe el siguiente ejemplo para la letra 'A'.

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE '%a%';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| ARG                 | Argentina      |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

Otro de los comodines que es posible utilizar con el operador LIKE es el guión bajo (\_), el cual sirve de comodín para un solo carácter.

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE 'Itali_';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ITA                 | Italia         |

El comodín \_, entonces puede tomar cualquier valor para un solo carácter.

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE 'Ital_';
Empty set (0.00 sec)
```

La expresión 'Ital\_' no regresa valor por que el comodín \_ sólo sirve para un carácter; para obtener un resultado, sería correcta la utilización del operador %.

Si en algún momento, desea hacer una consulta sobre una tabla, donde alguno de los campos contenga el carácter '%' o '\_', y desea hacer una consulta sobre esa columna, requerirá utilizar una secuencia de escape para los caracteres antes mencionados, así éstos no serán interpretados como un comodín sino como un carácter más. La secuencia de escape se representa con el símbolo #.

```
SELECT * FROM procedencia
WHERE TxtDescripcion LIKE 'Ital#%';
Empty set (0.00 sec)
```

En la consulta anterior se está buscando igualdad con el siguiente conjunto de caracteres 'Ital%', en tanto el '%' no se interpretó como un comodín, sino como un carácter más, ello debido a la secuencia de escape utilizada (#).

En la consulta anterior se está buscando igualdad con el siguiente conjunto de caracteres 'Ital%', esto debido a que el % no se interpreto como un comodín, sino que éste fue escapado por el símbolo #.

El asterisco (\*), también es un comodín sólo se utiliza para la realización de proyecciones, con la idea de abarcar todas las columnas.

El operador de diferencia (<>) tiene a su vez una equivalencia cuando se utiliza el operador LIKE; hacer uso del conjunto de operadores NOT LIKE va a permitir mostrar todas aquellas filas en donde no se cumpla con la condición.

Supongamos que se desea seleccionar los lugares de procedencia que en este caso no sea Francia, para lo cual haremos uso del operador NOT LIKE, de la siguiente forma:

```
SELECT * FROM procedencia
WHERE TxtDescripcion NOT LIKE 'Francia';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| AUS                 | Australia      |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| ARG                 | Argentina      |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

Por lo tanto, se regresan todas las filas donde el lugar de procedencia no es Francia.

A su vez, también es posible la incursión de comodines en la expresión, suponga que desea consultar todos los países de procedencia que no comiencen con la letra A, para ello se ejecutará la siguiente instrucción:

```
SELECT * FROM procedencia
WHERE TxtDescripcion NOT LIKE 'a%';
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

Observe que los resultados no incluyen ningún lugar de procedencia que comenzará con una A en su nombre, Argentina y Australia por tanto fueron quienes quedaron fuera.

NOT LIKE puede entenderse como la negación del comando LIKE, es decir, si ejecuta la instrucción LIKE en cualquiera de sus formas, evaluará sobre condiciones verdaderas, hacer uso de NOT LIKE evaluará sobre condiciones falsas.

LIKE que sea como....

NOT LIKE que no sea como...

#### 4.3.4 Operadores para el manejo de datos tipo Fecha

El manejo de fechas siempre es especialmente complicado en un principio, ya que el formato para introducir una fecha puede variar de acuerdo con el estándar establecido como predeterminado en el RDBMS; a su vez, existen formatos de fecha y hora cortos, y formatos largos, con distinta nomenclatura llámese utilizar día mes y año o mes día año. Es importante verificar que la nomenclatura de fecha y/o hora por utilizar concuerde con la establecida por el RDBMS que utilice.

Para el caso de la base de datos de la vinatería, el RDBMS de MySQL asignó el formato de fecha por defecto, el cual es el siguiente: 'aaaa/mm/dd', Año, Mes, Día.

Cuando se deseen hacer restricciones sobre la información que se quiere manejar y para el trabajo de fecha se utilizarán los operadores =, >, >=, <, <=, <>, en donde lo que se comparará son tipos de dato Fecha.

Para ejemplificar la restricción de información a partir de datos de tipo Fecha utilizaremos una proyección de la tabla de ventas de la base de datos de la vinatería.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0364       | 10     | 4           | 288.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 9           | 5310.00  | NULL                |
| FACT0369       | 1      | 1           | 483.00   | NULL                |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

Para comenzar con el operador de igualdad, seleccionaremos la información de las ventas del día 24 de diciembre de 2006, recordemos que el formato de fecha que reconoce MySQL es aaaa-mm-dd.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE dateFechaVenta='2006-12-24';
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |

El criterio que utilizamos para la consulta anterior fue la igualdad entre dos fechas, la almacenada en el campo de fecha de ventas y la fecha de la cual nos interesaba obtener información.

Es muy simple realizar condicionantes utilizando los operadores =, >, >=, <, <=, únicamente hay que considerar que los datos de ambos lados del operador sean del mismo tipo.

Si quisiéramos saber cuáles son las ventas realizadas después del 31 de diciembre de 2006, entonces ejecutamos la instrucción siguiente:

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE dateFechaVenta>'2006-12-31';
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

De igual forma que en casos anteriores, podemos utilizar los operadores =, >, >=, <, <=, <>, conociendo de antemano el resultado que se desea obtener, sin embargo, el trabajo de campos tipo Fecha tiene una potencialidad mayor, ya que éstos se utilizan para segmentar la información a partir de rangos de fecha, intervalos de tiempo en donde se tiene historia almacenada; algunos de los reportes estratégicos más comunes son por ejemplo reportes de ventas trimestrales, reportes de ventas anuales o por algún intervalo de tiempo determinado.

Para generar dichos reportes, es necesario utilizar dos expresiones condicionantes dentro de la restricción de información, ello con ayuda de los operadores AND y OR, por ejemplo si deseáramos rastrear todas las ventas del año 2007, ejecutaríamos la siguiente instrucción SQL.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE dateFechaVenta>='2007-01-01' AND dateFechaVenta<='2007-12-31';
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

Como se observa, se ha restringido la información correctamente; las condicionantes deben estructurarse de manera correcta para determinar el intervalo de tiempo que se desea restringir, ya que de lo contrario se pueden obtener resultados negativos, concordancias inexistentes, por ejemplo, si restringimos una fecha para que sea mayor al 1ero de abril de 2007 y a su vez que sea menor al 1ero de enero de 2007, obtendremos nada como resultado, ya que las condiciones son excluyentes, no existe ninguna coincidencia.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta
FROM ventas WHERE dateFechaVenta>='2007-03-01' AND dateFechaVenta<='2007-01-01';
Empty set (0.00 sec)
```

### 4.3.5 Operador BETWEEN

Existe dentro de la especificación ANSI SQL el operador **BETWEEN**, el cual va a condicionar a que un elemento se encuentre entre un rango de valores, es decir, se toma un valor mínimo y un valor máximo, y el elemento que se quiere evaluar debe existir dentro del dominio definido por el usuario.

Por ejemplo, suponga que se desea mostrar las ventas por las que se facturó entre los 3,000 y 9,000 pesos, para ello, existen dos posibilidades: utilizar los operadores  $\geq$  y  $\leq$  ó bien el operador **BETWEEN** que actuará de la manera en que actuarían los operadores anteriores, el ejemplo, es el siguiente.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE curMonto>=3000 and curMonto<=9000;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 9           | 5310.00  | NULL                |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |

En este primer caso se han utilizado los operadores  $\geq$ , **AND**,  $\leq$ . Se puede estructurar una consulta con el mismo sentido, pero ahora utilizando el operador **BETWEEN**, tal como se muestra a continuación.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE curMonto BETWEEN 3000 AND 9000;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 9           | 5310.00  | NULL                |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |

El uso del operador **BETWEEN** se utiliza para obtener los valores existentes entre un rango definido por el usuario; muy comúnmente el operador **BETWEEN** es utilizado para el manejo de fechas, por lo que es importante segmentar la información a través de intervalos de tiempo.

De esta manera, cuando decimos que queremos obtener información entre un periodo de tiempo, utilizaremos el operador **BETWEEN**.

Suponga que deseamos obtener información de las ventas realizadas entre el 1ero de enero de 2007 y el 30 de abril de 2007, es decir, información sobre el primer cuatrimestre del año, para ello, utilizaremos el operador **BETWEEN**, ya que como recuerda, una consulta muy similar se ha explicado utilizando los operadores  $\geq$ ,  $\leq$ .

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM ventas
WHERE dateFechaVenta BETWEEN '2007-01-01' AND '2007-04-30';
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |

Cuando utilizemos el operador **BETWEEN**, existe la posibilidad al igual que con el operador **LIKE**, de expresar los términos de la condición de forma negativa, es decir, utilizar el operador **BETWEEN** para restringir información que no se encuentre dentro de un intervalo determinado.

Por ejemplo, si quisiéramos obtener las ventas por las que no se facturó entre 1,500 y 6,000 pesos, entonces ejecutamos la siguiente consulta con **NOT BETWEEN**.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE curMonto NOT BETWEEN 1500 AND 6000;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0364       | 10     | 4           | 288.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0369       | 1      | 1           | 483.00   | NULL                |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |

#### 4.3.6 Operador IN

El siguiente operador que describiremos es el operador **IN**, el cual permite evaluar si un elemento se encuentra dentro de una lista de elementos definidos por el usuario.

La estructura del operador **IN**, es un poco distinta a otros operadores que hemos utilizado ya que la condicionante se expresa de una forma singular:

```
WHERE campo IN (elemento1, elemento2, ..., elementoN)
```

Seguido de la palabra reservada **IN** y entre paréntesis, se colocan los elementos con los cuales debe coincidir un campo; el operador **IN**, cumple con la funcionalidad del operador de igualdad, adjunto del operador **OR**, ya que el valor del campo por evaluar debe encontrarse en cualquiera de los valores de la lista que se encuentra entre paréntesis.

Es permisible recalcar el hecho de que los elementos de la lista pueden ser de cualquier tipo, siempre y cuando concuerde con el tipo de datos del campo que se quiere evaluar en la expresión, es decir, se utilizan con valores numéricos, carácter, fecha, etcétera.

Una consulta que refleja este comportamiento es aquella donde se vendió el vino 7, 8 ó 9.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE IdVino IN(7,8,9);
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

El operador **IN** simplifica el uso de múltiples condicionantes como se muestra a continuación:

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE IdVino=7 OR IdVino=8 OR IdVino=9;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

Tal como se puede observar, el operador **IN** simplifica el trabajo ya que el código se ve simplificado; por la misma cuestión de que el operador **IN** simplifica los operadores **=** y **OR**, es que se puede utilizar con datos de diverso tipo.

Recuerde que la lista de elementos que se desea evaluar en el operador **IN**, debe ir separada por comas, y cuando se utilicen datos tipo Carácter, éstos deberán ir entrecomillados, de igual forma como ya se ha mencionado.

```
SELECT * FROM procedencia
WHERE TxtDescripcion in ('España', 'Italia');
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| ITA                 | Italia         |

## 4.4 Valores nulos

El manejo de valores nulos dentro de una base de datos debe realizarse con cierta precaución, ya que la falta de valor para ciertos campos de una tabla puede generar errores al momento de procesar información. Los valores nulos contenidos en una tabla, bien pueden ser procesados, para ser actualizados ó ser omitidos en una consulta de información, ya que no tiene mucho sentido consultar un campo que no tiene valor.

Para la restricción de valores nulos dentro de los campos de una tabla, ANSI SQL especifica la utilización del operador `IS NULL`, el cual evalúa si un campo tiene en algún momento un valor nulo.

Para trabajar, retomaremos la proyección de la tabla de ventas con la que hemos venido trabajando.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0364       | 10     | 4           | 288.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 9           | 5310.00  | NULL                |
| FACT0369       | 1      | 1           | 483.00   | NULL                |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

Como se puede observar, el campo de fecha de venta implica dos valores nulos, con los cuales los podemos recuperar a través del operador `IS NULL`, y una vez que los separemos del resto de los demás, quizás valga la pena hacer alguna operación con ellos, tal como asignarles valor.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE dateFechaVenta IS NULL;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta |
|----------------|--------|-------------|----------|----------------|
| FACT0368       | 5      | 9           | 5310.00  | NULL           |
| FACT0369       | 1      | 1           | 483.00   | NULL           |

Por tanto, ya que identificamos los valores nulos, procederemos a actualizarlos en la base de datos, recuerde que para hacer actualizaciones o eliminación de información de manera efectiva, es muy importante seccionar o restringir la información por utilizar, y

que mejor forma de identificar cada uno de los renglones de la tabla de manera única que con el uso de la llave primaria.

```
UPDATE ventas SET dateFechaVenta='2007-02-15'
WHERE TxtClaveVentas ='FACT0368';
UPDATE ventas SET dateFechaVenta='2007-02-18'
WHERE TxtClaveVentas ='FACT0369';
```

Las dos instrucciones anteriores actualizan los datos para el campo Fecha de venta, pero sólo en lugares específicos, es decir, restringidos.

El operador IS NULL también puede utilizarse en conjunto con la cláusula NOT, es decir, podemos negar el significado del operador IS NULL para que busque dónde el valor de los campos no sea nulo.

```
SELECT TxtClaveVentas, IdVino, IntCantidad, CurMonto, dateFechaVenta FROM
ventas
WHERE dateFechaVenta IS NOT NULL;
```

| TxtClaveVentas | IdVino | IntCantidad | CurMonto | dateFechaVenta      |
|----------------|--------|-------------|----------|---------------------|
| FACT0360       | 7      | 2           | 998.00   | 2006-12-24 00:00:00 |
| FACT0361       | 7      | 1           | 499.00   | 2006-12-24 00:00:00 |
| FACT0362       | 7      | 3           | 1497.00  | 2006-12-31 00:00:00 |
| FACT0363       | 7      | 1           | 499.00   | 2006-12-31 00:00:00 |
| FACT0364       | 10     | 4           | 288.00   | 2006-12-31 00:00:00 |
| FACT0365       | 9      | 6           | 930.00   | 2007-01-06 00:00:00 |
| FACT0366       | 3      | 7           | 8722.00  | 2007-01-21 00:00:00 |
| FACT0367       | 8      | 8           | 576.00   | 2007-01-21 00:00:00 |
| FACT0368       | 5      | 9           | 5310.00  | 2007-02-15 00:00:00 |
| FACT0369       | 1      | 1           | 483.00   | 2007-02-18 00:00:00 |
| FACT0370       | 6      | 2           | 158.00   | 2007-03-05 00:00:00 |
| FACT0371       | 7      | 4           | 1996.00  | 2007-03-05 00:00:00 |
| FACT0372       | 2      | 5           | 1625.00  | 2007-03-05 00:00:00 |
| FACT0373       | 6      | 6           | 474.00   | 2007-03-16 00:00:00 |
| FACT0374       | 3      | 3           | 3738.00  | 2007-03-21 00:00:00 |
| FACT0375       | 2      | 4           | 1300.00  | 2007-04-16 00:00:00 |
| FACT0376       | 5      | 1           | 590.00   | 2007-04-24 00:00:00 |
| FACT0377       | 7      | 8           | 3992.00  | 2007-04-18 00:00:00 |
| FACT0378       | 4      | 4           | 1300.00  | 2007-05-30 00:00:00 |
| FACT0379       | 7      | 2           | 998.00   | 2007-05-30 00:00:00 |
| FACT0380       | 7      | 6           | 2994.00  | 2007-06-09 00:00:00 |

Respecto al caso negado del operador IS NULL, la palabra reservada NOT se coloca entre IS y NULL, para así evaluar si un campo es no nulo, es decir sí, IS NOT NULL.

## 4.5 Ordenamiento de la información

Partiendo de la premisa de que bajo el modelo relacional, las tuplas no están ordenadas, el estándar SQL implementa un mecanismo para ordenar las filas contenidas en una tabla de acuerdo con un criterio especificado por el usuario, dicha acción puede llevarse a cabo a través del predicado ORDER BY.

Por ejemplo, la tabla de procedencia, muestra los países de procedencia de vinos; pero haciendo una consulta como hasta ahora se había venido haciendo, los resultados aparecen sin un orden, más bien aparecen como se fueron ingresando los datos. Observe el contenido de la tabla:

```
SELECT * FROM procedencia;
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ESP                 | España         |
| FRA                 | Francia        |
| AUS                 | Australia      |
| ITA                 | Italia         |
| ESC                 | Escocia        |
| ARG                 | Argentina      |
| FIN                 | Finlandia      |
| ING                 | Inglaterra     |

Al realizar reportes para la toma de decisiones, generalmente se busca la mayor estandarización al momento de presentar la información; de presentarse los datos como hasta este momento se encuentran, resulta difícil la correcta lectura de los mismos, ya que no tienen un orden específico.

Para ordenar la información de manera ascendente, ya sea para valores numéricos, fecha, o carácter, SQL implementa la cláusula ORDER BY que permitirá ordenar la información.

```
SELECT * FROM procedencia
ORDER BY TxtDescripcion;
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ARG                 | Argentina      |
| AUS                 | Australia      |
| ESC                 | Escocia        |
| ESP                 | España         |
| FIN                 | Finlandia      |
| FRA                 | Francia        |
| ING                 | Inglaterra     |
| ITA                 | Italia         |

Haciendo uso de la cláusula ORDER BY, el RDMBS regresa la información ordenada alfabéticamente y en orden ascendente. Así mismo, también existe la posibilidad de mostrar la información en orden descendente, para ello, agregaremos la palabra reservada DESC, al final del conjunto de campos a través de los cuales se ordenará la información.

```
SELECT * FROM procedencia
ORDER BY TxtDescripcion DESC;
```

| txtClaveProcedencia | TxtDescripcion |
|---------------------|----------------|
| ITA                 | Italia         |
| ING                 | Inglaterra     |
| FRA                 | Francia        |
| FIN                 | Finlandia      |
| ESP                 | España         |
| ESC                 | Escocia        |
| AUS                 | Australia      |
| ARG                 | Argentina      |

Así pues, la estructura de la cláusula ORDER BY es la siguiente:

```
ORDER BY columna1, columna2, ..., columnaN [DESC]
```

Por lo anterior, podemos darnos cuenta que es posible ordenar la información por más de un criterio a la vez; para ejemplificar tal situación, tomaremos como base la tabla de vinos cuyo contenido es el siguiente:

```
SELECT * FROM vinos;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |

Suponga que deseamos mostrar la información ordenada de acuerdo con el precio del vino, mostrando los vinos de menor a mayor precio, entonces utilizaríamos la siguiente instrucción SQL:

```
SELECT * FROM vinos
ORDER BY curPrecio;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |

La cláusula **ORDER BY** permitió ordenar los registros de acuerdo con el criterio ordenado que fue el precio del vino. Pero qué ocurre si deseamos ver primero la información ordenada de acuerdo al lugar de procedencia, y en segunda instancia el precio del producto, para ello utilizaríamos la cláusula **ORDER BY** en conjunto con dos criterios, tal como se muestra a continuación:

```
SELECT * FROM vinos
ORDER BY TxtClaveProcedencia, curPrecio;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |

Tal como puede apreciarse en el resultado los vinos fueron ordenados en primera instancia por procedencia y en segundo plano, conforme al precio de venta.

Una variante más podríamos encontrarla al decir que queremos ordenar la información de acuerdo con el país de procedencia en orden descendente, pero los precios ordenados en forma descendente, para ello la cláusula **ORDER BY** permite indicar seguido del criterio de ordenamiento, la forma en que éste se hará, ya sea ascendente o descendentemente; cuando la ordenación sea de forma a ascendente se utiliza la palabra reservada **ASC** y cuando el ordenamiento se desee descendentemente, se utilizará la palabra reservada **DESC**; cabe mencionar que si no se indica la forma en que se ordenarán los registros, por defecto se toma el valor de **ASC**, es decir, ascendentemente.

```
SELECT * FROM vinos
ORDER BY TxtClaveProcedencia DESC, curPrecio ASC;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |

Vinos ordenados descendentemente de acuerdo con la procedencia, y ascendentemente de acuerdo con el precio.

```
SELECT * FROM vinos
ORDER BY TxtClaveProcedencia DESC, curPrecio DESC;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |

Vinos ordenados descendientemente de acuerdo con la procedencia y el precio.

```
SELECT * FROM vinos
ORDER BY TxtClaveProcedencia ASC, curPrecio ASC;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |

Vinos ordenados ascendentemente de acuerdo con la procedencia y el precio.

```
SELECT * FROM vinos
ORDER BY TxtClaveProcedencia ASC, curPrecio DESC;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |

Vinos ordenados ascendentemente de acuerdo con la procedencia, y descendientemente de acuerdo con el precio.

Para ordenar la información, se pueden incluir tantos criterios como columnas estén incluidas en las tablas de las cuales seleccionemos la información.

## 4.6 Alias y operadores matemáticos

En la incursión del SQL estándar básico, nos encontramos con los alias; un alias, es un mecanismo del estándar ANSI SQL, el cuál va a permitir renombrar nombres de campos o nombres de tablas. El objetivo principal de los alias, es la simplificación al momento de escribir código SQL, a su vez provee la funcionalidad de renombrar los nombres de columnas en la salida de información a través de la instrucción **SELECT**.

La palabra reservada del lenguaje SQL que va a permitir renombrar un campo o una tabla es **AS**; a continuación ejemplificaremos de forma sencilla la incursión de alias en una consulta de información.

Aunado a la incursión de alias, trabajaremos con los operadores matemáticos de suma, resta, multiplicación y división (+, -, \*, /).

La instrucción SQL **SELECT** no sólo permite recuperar información de las tablas de la base de datos, sino que también provee la funcionalidad de realizar operaciones matemáticas básicas, de utilizar funciones del sistema o funciones que implementa el RDBMS.

Tomando como base lo anterior, es posible ejecutar una instrucción **SELECT**, que permita ejecutar la siguiente operación matemática:  $25 * 63$ .

```
SELECT 25*63;
+-----+
| 25*63 |
+-----+
| 1575  |
+-----+
```

El resultado de la operación se expresa en forma de una tabla, ello debido a la propiedad de cierre de álgebra relacional<sup>12</sup>. Así mismo es posible no sólo realizar operaciones a través de la incorporación de operadores aritméticos, sino también utilizar funciones incorporadas al RDBMS.

```
SELECT version();
+-----+
| version() |
+-----+
| 5.0.37-community-nt |
+-----+
```

La función anterior regresa la versión del manejador MySQL con la cual nos encontramos trabajando; existen otro tipo de funciones que se abordarán en el capítulo número siete.

Ya que conocemos como es posible en principio utilizar los operadores aritméticos y funciones, procederemos a incorporar el concepto de alias.

Un alias como bien se ha mencionado, permitirá renombrar una columna o tabla al momento de seleccionar información, donde el objetivo principal es la simplificación de

<sup>12</sup> Propiedad de Cierre de Álgebra Relacional: Toda operación entre relaciones genera una nueva relación es decir  $r(A) + r(B) = r(C)$

código y mejorar el formato de salida. Por ejemplo, utilizaremos un alias para cambiar los nombres de las columnas de los ejemplos anteriores:

```
SELECT 25*63 AS Multiplica;
+-----+
| Multiplica |
+-----+
|      1575 |
+-----+
```

El concepto de alias se aplica a través de la palabra reservada AS seguida del sobrenombre por utilizar para la columna, un ejemplo más.

```
SELECT version() as MySQL;
+-----+
| MySQL |
+-----+
| 5.0.37-community-nt |
+-----+
```

Cuando trabaje con alias, es importante que el sobrenombre que utilice, preferentemente no contenga espacios o caracteres especiales en tanto puede causar confusión al RDBMS.

Ahora que se ha definido en términos simples el propósito que persiguen los alias, incrementaremos la visión de su utilidad, para ello, utilizaremos la siguiente proyección de la tabla de ventas.

```
SELECT TxtClaveVentas, intCantidad, CurMonto FROM ventas;
+-----+-----+-----+
| TxtClaveVentas | intCantidad | CurMonto |
+-----+-----+-----+
| FACT0360       | 2          | 998.00   |
| FACT0361       | 1          | 499.00   |
| FACT0362       | 3          | 1497.00  |
| FACT0363       | 1          | 499.00   |
| FACT0364       | 4          | 288.00   |
| FACT0365       | 6          | 930.00   |
| FACT0366       | 7          | 8722.00  |
| FACT0367       | 8          | 576.00   |
| FACT0368       | 9          | 5310.00  |
| FACT0369       | 1          | 483.00   |
| FACT0370       | 2          | 158.00   |
| FACT0371       | 4          | 1996.00  |
| FACT0372       | 5          | 1625.00  |
| FACT0373       | 6          | 474.00   |
| FACT0374       | 3          | 3738.00  |
| FACT0375       | 4          | 1300.00  |
| FACT0376       | 1          | 590.00   |
| FACT0377       | 8          | 3992.00  |
| FACT0378       | 4          | 1300.00  |
| FACT0379       | 2          | 998.00   |
| FACT0380       | 6          | 2994.00  |
+-----+-----+-----+
```

En la proyección que utilizaremos contamos con el número de factura, la cantidad de artículos comprados y el monto de la venta; ahora agregaremos una nueva columna, ésta será derivada de la información con que contamos actualmente, para ello supondremos que la columna de monto, no incluye el IVA y requerimos de una consulta que lo muestre, por tanto, utilizaremos una instrucción SELECT para generar la nueva columna derivada.

Para la obtención de la columna derivada, multiplicaremos el monto de la venta por 1.15.

```
SELECT TxtClaveVentas, intCantidad, CurMonto, CurMonto * 1.15 FROM ventas;
```

| TxtClaveVentas | intCantidad | CurMonto | CurMonto * 1.15 |
|----------------|-------------|----------|-----------------|
| FACT0360       | 2           | 998.00   | 1147.7000       |
| FACT0361       | 1           | 499.00   | 573.8500        |
| FACT0362       | 3           | 1497.00  | 1721.5500       |
| FACT0363       | 1           | 499.00   | 573.8500        |
| FACT0364       | 4           | 288.00   | 331.2000        |
| FACT0365       | 6           | 930.00   | 1069.5000       |
| FACT0366       | 7           | 8722.00  | 10030.3000      |
| FACT0367       | 8           | 576.00   | 662.4000        |
| FACT0368       | 9           | 5310.00  | 6106.5000       |
| FACT0369       | 1           | 483.00   | 555.4500        |
| FACT0370       | 2           | 158.00   | 181.7000        |
| FACT0371       | 4           | 1996.00  | 2295.4000       |
| FACT0372       | 5           | 1625.00  | 1868.7500       |
| FACT0373       | 6           | 474.00   | 545.1000        |
| FACT0374       | 3           | 3738.00  | 4298.7000       |
| FACT0375       | 4           | 1300.00  | 1495.0000       |
| FACT0376       | 1           | 590.00   | 678.5000        |
| FACT0377       | 8           | 3992.00  | 4590.8000       |
| FACT0378       | 4           | 1300.00  | 1495.0000       |
| FACT0379       | 2           | 998.00   | 1147.7000       |
| FACT0380       | 6           | 2994.00  | 3443.1000       |

El resultado que se muestra contiene una columna derivada, la cual no se encuentra almacenada en disco duro, ni se guarda o almacena en una tabla de la base de datos, más bien ésta se genera dinámicamente al momento de ejecutar la consulta; como se muestra, el nombre de la columna generada aparece tal como se realizó la operación, sin embargo, por motivos de formato es posible renombrarla asignándole un nombre más descriptivo, para realizar dicha tarea haremos uso de un alias como se muestra a continuación:

```
SELECT TxtClaveVentas, intCantidad, CurMonto, CurMonto * 1.15 AS 'Monto con Iva' FROM ventas;
```

| TxtClaveVentas | intCantidad | CurMonto | Monto con Iva |
|----------------|-------------|----------|---------------|
| FACT0360       | 2           | 998.00   | 1147.7000     |
| FACT0361       | 1           | 499.00   | 573.8500      |
| FACT0362       | 3           | 1497.00  | 1721.5500     |
| FACT0363       | 1           | 499.00   | 573.8500      |
| FACT0364       | 4           | 288.00   | 331.2000      |
| FACT0365       | 6           | 930.00   | 1069.5000     |
| FACT0366       | 7           | 8722.00  | 10030.3000    |
| FACT0367       | 8           | 576.00   | 662.4000      |
| FACT0368       | 9           | 5310.00  | 6106.5000     |
| FACT0369       | 1           | 483.00   | 555.4500      |
| FACT0370       | 2           | 158.00   | 181.7000      |
| FACT0371       | 4           | 1996.00  | 2295.4000     |
| FACT0372       | 5           | 1625.00  | 1868.7500     |
| FACT0373       | 6           | 474.00   | 545.1000      |
| FACT0374       | 3           | 3738.00  | 4298.7000     |
| FACT0375       | 4           | 1300.00  | 1495.0000     |
| FACT0376       | 1           | 590.00   | 678.5000      |
| FACT0377       | 8           | 3992.00  | 4590.8000     |
| FACT0378       | 4           | 1300.00  | 1495.0000     |
| FACT0379       | 2           | 998.00   | 1147.7000     |
| FACT0380       | 6           | 2994.00  | 3443.1000     |

Como se observa, el nombre de la columna derivada ha cambiado, y fue posible utilizar espacios en blanco dentro del nombre, debido a la incorporación de comillas simples para indicar el nuevo nombre de columna.

El uso de alias tiene un uso con mayor potencialidad al generar consultas para realizar agrupaciones, funciones de agregado, subconsultas y por supuesto en donde se haga uso de join's, por ahora, sólo basta con que comprenda la forma de uso y funcionalidad de este elemento, ya que la explotación más fuerte que hagamos de él se verá reflejada en los temas siguientes, que implicará retomar todo lo aprendido del operador SELECT y de las variantes hasta ahora revisadas.

## 4.7 Subconsultas

Hasta ahora la interacción que hemos tenido con el operador SELECT ha sido única y exclusivamente para la recuperación de información proveniente de una sola tabla; sin embargo, en la realidad esto es poco usual, en tanto, la información que se desea utilizar para la toma de decisiones, no siempre estará en una sola tabla, quizá sea necesario utilizar dos o más.

Lo aprendido hasta el momento, nos sirve para estructurar de forma correcta una proyección y restricción de información, ya que éstas son la base para las subconsultas y las consultas especializadas con join's.

Una subconsulta va a permitir recuperar información de una tabla tomando como base la información proveniente de una consulta de información previa; la forma en que éstas van a interaccionar, será a través de la cláusula **WHERE**, la cual tomará información proveniente de la consulta previa.

A grandes rasgos, podemos decir que la estructura de una subconsulta es la siguiente:

```
SELECT * FROM tabla
WHERE .... ( SELECT * FROM tabla2 )
```

Se utilizan las subconsultas debido a que las tablas transaccionales de una base de datos regularmente se encuentran compuestas de llaves foráneas que permiten la existencia de vínculos entre las tablas; las tablas transaccionales, entonces, almacenan las llaves foráneas de catálogos de la base de datos y permiten conocer a qué valor pertenece cada clave, algunas veces resulta difícil, por ello, primero se realiza una consulta sobre un catálogo y, posteriormente, se retoma información de ésta para hacer una búsqueda en la tabla transaccional.

La siguiente tabla muestra las cláusulas que van a permitir la realización de subconsultas en una base de datos relacional a través de SQL.

| Cláusula | Utilidad                                                                                       |
|----------|------------------------------------------------------------------------------------------------|
| IN       | Evalúan los elementos que son la salida de una subconsulta para la restricción de información. |
| ANY      |                                                                                                |
| ALL      |                                                                                                |
| EXISTS   |                                                                                                |

### 4.7.1 Cláusula IN

Para ejemplificar el uso de la cláusula **IN** en la realización de subconsultas, tomaremos como base la tabla de vendedores y una proyección de la tabla de ventas como se muestra a continuación.

```
SELECT * FROM vendedores;
```

| idVendedor | txtNombreVendedor       |
|------------|-------------------------|
| 301        | Fernando Gómez Galicia  |
| 302        | Jennifer Aguilar Cortés |
| 303        | Dario González Henkel   |
| 304        | Carlos Martínez Salinas |
| 305        | Arabia Leñero Orozco    |

```
SELECT TxtClaveVentas, IdVino, IdVendedor, intCantidad, CurMonto FROM  
ventas;
```

| TxtClaveVentas | IdVino | IdVendedor | intCantidad | CurMonto |
|----------------|--------|------------|-------------|----------|
| FACT0360       | 7      | 301        | 2           | 998.00   |
| FACT0361       | 7      | 305        | 1           | 499.00   |
| FACT0362       | 7      | 304        | 3           | 1497.00  |
| FACT0363       | 7      | 302        | 1           | 499.00   |
| FACT0364       | 10     | 303        | 4           | 288.00   |
| FACT0365       | 9      | 302        | 6           | 930.00   |
| FACT0366       | 3      | 302        | 7           | 8722.00  |
| FACT0367       | 8      | 303        | 8           | 576.00   |
| FACT0368       | 5      | 303        | 9           | 5310.00  |
| FACT0369       | 1      | 301        | 1           | 483.00   |
| FACT0370       | 6      | 302        | 2           | 158.00   |
| FACT0371       | 7      | 305        | 4           | 1996.00  |
| FACT0372       | 2      | 304        | 5           | 1625.00  |
| FACT0373       | 6      | 302        | 6           | 474.00   |
| FACT0374       | 3      | 301        | 3           | 3738.00  |
| FACT0375       | 2      | 301        | 4           | 1300.00  |
| FACT0376       | 5      | 302        | 1           | 590.00   |
| FACT0377       | 7      | 304        | 8           | 3992.00  |
| FACT0378       | 4      | 302        | 4           | 1300.00  |
| FACT0379       | 7      | 303        | 2           | 998.00   |
| FACT0380       | 7      | 305        | 6           | 2994.00  |

Como se puede apreciar, para la proyección de la tabla de ventas, se ha incluido la columna que contiene el Identificador de los vendedores de nuestra tienda de vinos, si quisiéramos conocer cuáles son las ventas realizadas por un vendedor o vendedores en particular, entonces deberíamos conocer su clave, cosa difícil ya que pudiese darse el caso de tener más de mil vendedores en catálogo, por ende, conocer la clave específica de uno de ellos, es complicado.

La utilización de consultas va a facilitar esta tarea, por ejemplo, suponga que deseamos conocer las ventas del vendedor Fernando Gómez, entonces debemos hacer una restricción sobre la tabla de ventas en donde sólo aparezcan las ventas de dicho vendedor.

```
SELECT TxtClaveVentas, IdVino, IdVendedor, intCantidad, CurMonto FROM ventas  
WHERE idVendedor IN  
(  
    SELECT idVendedor FROM vendedores  
    WHERE TxtNombreVendedor LIKE 'Fernando Gómez%'  
);
```

| TxtClaveVentas | IdVino | IdVendedor | intCantidad | CurMonto |
|----------------|--------|------------|-------------|----------|
| FACT0360       | 7      | 301        | 2           | 998.00   |
| FACT0369       | 1      | 301        | 1           | 483.00   |
| FACT0374       | 3      | 301        | 3           | 3738.00  |
| FACT0375       | 2      | 301        | 4           | 1300.00  |

El código SQL anterior puede parecer complejo a primera vista; sin embargo, si lo analizamos detenidamente, podremos darnos cuenta de que es muy simple, considerando lo que ya hemos realizado con anterioridad.

La consulta anterior se compone de seis líneas de código, la primera línea tiene como finalidad realizar una proyección de la tabla de ventas; en la segunda, se hace una restricción de la tabla de ventas en dónde se evalúa que el identificador de los vendedores se encuentre en la lista de elementos que proporciona como resultado la subconsulta encontrada en las líneas 4 y 5.

El operador **IN** como ya lo hemos mencionado, evalúa que el criterio de búsqueda coincida con los elementos proporcionados en la lista ubicada entre paréntesis, sin embargo, en el caso anterior no tenemos una lista de elementos, sino que se tiene otra consulta SQL, la cual regresa un conjunto de elementos para ser evaluados.

Si ejecutamos primeramente la consulta:

```
SELECT idVendedor FROM vendedores
WHERE TxtNombreVendedor LIKE 'Fernando Gómez%';
```

obtenemos como resultado

```
+-----+
| idVendedor |
+-----+
|      301  |
+-----+
```

Este resultado es el que se toma para evaluar dentro del operador **IN**, ya que la subconsulta o consulta anidada regresa un valor que es utilizado por la consulta principal para la restricción de información. La interpretación que se haría es la siguiente, tomando como base el resultado arrojado por la consulta anidada.

```
SELECT TxtClaveVentas, IdVino, IdVendedor, intCantidad, CurMonto FROM ventas
WHERE idVendedor IN (301);
```

Es importante recalcar que hay una relación estrecha entre la consulta y la subconsulta ya que si es observador se podrá dar cuenta de que en la consulta se restringe la información a partir del *id* del vendedor y dentro de la subconsulta se proyecta el *id* del vendedor.

```
SELECT TxtClaveVentas, IdVino, IdVendedor, intCantidad, CurMonto FROM ventas
WHERE idVendedor IN
(
    SELECT idVendedor FROM vendedores
    WHERE TxtNombreVendedor LIKE 'Fernando Gómez%'
);
```

La concordancia de los campos es necesaria ya que de ello depende el éxito de la consulta, esto debido a que la lista de elementos del operador IN es proporcionada por la salida de información de la consulta.

Ahora trabajaremos con la proyección de la tabla de ventas y con la tabla de vinos, la cual contiene la siguiente información:

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio |
|--------|---------------------------------|---------------------|-----------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    |

El objetivo de la siguiente consulta es identificar las ventas de los vinos con precio menor a los 300 pesos, como es notorio, la tabla de ventas no contiene información sobre el precio de venta de los productos, dicho dato, se encuentra almacenado en la tabla de vinos, al momento de referirnos al uso de dos tablas, inmediatamente podemos contemplar el uso de una subconsulta como se muestra a continuación.

En primer lugar, hay que realizar una búsqueda de los productos con precio menor a los 300 pesos.

```
SELECT idVino FROM vinos WHERE curPrecio <=300;
```

| idVino |
|--------|
| 6      |
| 8      |
| 9      |
| 10     |
| 11     |

Ahora que sabemos cuáles son esos vinos, la tabla de ventas puede utilizar el resultado de la consulta para especificarnos son las ventas de los vinos con precio menor o igual a los 300 pesos.

```
SELECT TxtClaveVentas, IdVino, IdVendedor, intCantidad, CurMonto FROM ventas
WHERE idVino IN
```

```
(
    SELECT idVino FROM vinos WHERE curPrecio <=300
);
```

| TxtClaveVentas | IdVino | IdVendedor | intCantidad | CurMonto |
|----------------|--------|------------|-------------|----------|
| FACT0364       | 10     | 303        | 4           | 288.00   |
| FACT0365       | 9      | 302        | 6           | 930.00   |
| FACT0367       | 8      | 303        | 8           | 576.00   |
| FACT0370       | 6      | 302        | 2           | 158.00   |
| FACT0373       | 6      | 302        | 6           | 474.00   |

Observe que en la consulta se restringe la información a través de la columna `idVino`, y que en la proyección de la subconsulta se utiliza la misma columna.

Como podemos darnos cuenta, el operador `IN` utilizó los elementos que recuperó la subconsulta para la restricción de la información.

#### 4.7.2 Cláusula **ANY**

La cláusula `ANY` evalúa que alguno de los elementos de la subconsulta cumplan con la expresión condicionante de la consulta principal.

Por ejemplo, si deseáramos conocer cualquiera de las ventas donde se vendieron los vinos con precio mayor a 500 pesos, entonces ejecutamos la consulta siguiente:

```
SELECT TxtClaveVentas, idVino FROM ventas
WHERE IdVino = ANY (SELECT idVino FROM vinos WHERE curPrecio>500);
```

| TxtClaveVentas | idVino |
|----------------|--------|
| FACT0366       | 3      |
| FACT0368       | 5      |
| FACT0374       | 3      |
| FACT0376       | 5      |

La subconsulta, por ende, obtiene los identificadores de vino cuyo precio del vino es mayor a los 500 pesos, el resultado de la subconsulta es:

```
SELECT idVino FROM vinos WHERE curPrecio>500;
```

| idVino |
|--------|
| 3      |
| 5      |

Por su parte, la cláusula `ANY` permite mostrar las ventas para cualquiera de los dos vinos que se obtuvieron como resultado en la consulta anterior.

#### 4.7.3 Cláusula **ALL**

La cláusula `ALL` permite evaluar que todos los resultados de la subconsulta sean evaluados en la expresión condicionante de información; si ejecutamos la consulta anterior donde utilizamos `ANY` y reemplazamos la cláusula por `ALL`, obtenemos lo siguiente:

```
mysql> SELECT TxtClaveVentas, idVino FROM ventas
WHERE IdVino = ALL (SELECT idVino FROM vinos WHERE curPrecio>500);
Empty set (0.02 sec)
```

El resultado es nada debido a que la cláusula `ALL` evalúa como un operador `AND`, es decir, que el identificador del vino sea igual a Todos (`ALL`) los identificadores de vino cuyo precio de vino es mayor a los 500 pesos, mientras que la cláusula `ANY` funciona como un operador `OR`, ya que los identificadores de vino deben ser como cualquiera de los entregados por la subconsulta.

#### 4.7.4 Cláusula EXISTS

La cláusula EXISTS tiene una funcionalidad similar a la cláusula IN, ya que se busca que los resultados de la consulta principal concuerden o existan en la subconsulta realizada y viceversa. Para ejecutar la cláusula EXISTS, se deberán utilizar dos tablas que tengan columnas en común, las cuales servirán para referenciar las tablas.

```
SELECT TxtClaveVentas, IdVino, CurMonto FROM ventas
WHERE EXISTS
(SELECT * FROM vinos WHERE curPrecio>500 AND ventas.IdVino=vinos.IdVino);
```

| TxtClaveVentas | IdVino | CurMonto |
|----------------|--------|----------|
| FACT0366       | 3      | 8722.00  |
| FACT0368       | 5      | 5310.00  |
| FACT0374       | 3      | 3738.00  |
| FACT0376       | 5      | 590.00   |

La subconsulta anterior muestra la clave de venta, el identificador del vino y el monto de la tabla de ventas donde exista la condición de que el vino implique más de 500 pesos; la forma como se restringe la información, depende de que exista una columna en cada tabla que sirva de referencia para restringir la información, así es como se obtienen sólo las ventas de los vinos 3 y 5, en tanto, ambos valen unitariamente más de 500 pesos.

# CAPÍTULO 5

## Funciones de utilidad

---

El SELECT de SQL provee la más grande potencialidad de los RDBMS, ya que ciertas ocasiones más que mostrar información almacenada de la base de datos, se muestran atributos derivados que son obtenidos a través de funciones predeterminadas por el estándar SQL.

Existen diversas funciones incorporadas a los RDBMS; el estándar SQL implementado por el manejador, determina las funciones que son posibles de ejecutar, no obstante, presentaremos un breve compendio de las funciones más utilizadas.

### 5.1 Funciones para datos tipo carácter

Las funciones para datos tipo Carácter reciben como parámetro una cadena de datos, esa cadena que es procesada de alguna forma, y con lo cual se obtiene como resultado una cadena de caracteres. SQL provee seis funciones básicas para el manejo de datos tipo Carácter.

| Función   | Utilidad                                                                                                                                                                                                                                                                                                                          |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SUBSTRING | Obtiene un subconjunto de caracteres de una cadena de datos.                                                                                                                                                                                                                                                                      |
| UPPER     | Transforma una cadena de caracteres a mayúsculas.                                                                                                                                                                                                                                                                                 |
| LOWER     | Transforma una cadena de caracteres a minúsculas.                                                                                                                                                                                                                                                                                 |
| TRIM      | Elimina los espacios contenidos al final o al principio de una cadena.                                                                                                                                                                                                                                                            |
| TRANSLATE | Ambas funciones tienen como finalidad cambiar el conjunto de caracteres que se utilizarán para la salida de información, por ejemplo, la información puede estar almacenada en latin1 y se desea una salida de información con un conjunto de caracteres griego, chino u otro. La implementación depende del manejador utilizado. |
| CONVERT   |                                                                                                                                                                                                                                                                                                                                   |

La potencialidad del SELECT de SQL permite seleccionar y presentar información aun cuando no esté contenida en la base de datos, la cual puede ser definida por el usuario, tal como se muestra a continuación:

```
SELECT 'Hola Mundo';
```

```
+-----+
| Hola Mundo |
+-----+
| Hola Mundo |
+-----+
```

Ahora utilizando alias:

```
SELECT 'Hola Mundo' AS Ejemplo1;
+-----+
| Ejemplo1 |
+-----+
| Hola Mundo |
+-----+
```

### 5.1.1 SUBSTRING

La función SUBSTRING permite obtener un subconjunto de caracteres de una cadena de datos. SUBSTRING (cadena\_entrada FROM inicio FOR 7)

La forma de implementar esta función es a través de la palabra reservada SUBSTRING y colocando entre paréntesis la cadena inicial y el rango de caracteres por obtener, por ejemplo:

```
SELECT SUBSTRING('Carlos Reyes García' FROM 8 FOR 5);
+-----+
| SUBSTRING('Carlos Reyes García' FROM 8 FOR 5) |
+-----+
| Reyes |
+-----+
```

La consulta anterior obtiene un subconjunto de caracteres de la cadena 'Carlos Reyes García', el conjunto de caracteres que se obtiene comienza en el carácter 7 y se toman 5 caracteres a partir de éste; es decir el subconjunto de caracteres va del carácter 7 al 12, la longitud es de 5.

### 5.1.2 UPPER

Esta función SQL transforma una cadena de caracteres a mayúsculas, tiene una gran utilidad cuando se desea estandarizar la salida de información con el objeto de comparar cadenas de caracteres.

```
SELECT UPPER('hola mundo') AS Mayusculas;
+-----+
| Mayusculas |
+-----+
| HOLA MUNDO |
+-----+
```

La función UPPER, recibe como parámetro una cadena de caracteres y devuelve la misma cadena de caracteres convertida a mayúsculas.

### 5.1.3 LOWER

Esta función SQL transforma una cadena de caracteres a minúsculas, tiene una gran utilidad cuando se desea estandarizar la salida de información para comparar cadenas de caracteres.

```
SELECT LOWER('HOLA MUNDO') AS Minusculas;
+-----+
| Minusculas |
+-----+
| hola mundo |
+-----+
```

La función **LOWER** recibe como parámetro una cadena de caracteres y devuelve la misma cadena de caracteres convertida a mayúsculas.

### **5.1.4 TRIM**

La función **TRIM**<sup>13</sup> devuelve una cadena de caracteres sin espacios finales o iniciales dependiendo la proporcionada como parámetro.

La estructura de la función consiste en colocar la palabra reservada **TRIM** y entre paréntesis: una de las siguiente tres opciones **LEADING**, **TRAILING** o **BOTH**, el carácter por recortar y la cadena que se transformará.

```
TRIM ([LEADING | TRAILING | BOTH] caracter FROM cadena)
SELECT TRIM(LEADING ' ' FROM '   HOLA MUNDO   ');
      ltrim
-----
HOLA MUNDO
SELECT TRIM(TRAILING ' ' FROM '   HOLA MUNDO   ');
      rtrim
-----
HOLA MUNDO
SELECT TRIM(BOTH ' ' FROM '   HOLA MUNDO   ');
      btrim
-----
HOLA MUNDO
```

En los ejemplos anteriores, se recortó el carácter de espacio de la cadena ' HOLA MUNDO ', donde en primer lugar, se recortaron los espacios iniciales; en el segundo caso, los espacios finales y, por último, los espacios finales e iniciales.

## **5.2 Funciones matemáticas**

Las funciones matemáticas permiten hacer cálculos con datos de tipo numérico, para obtener atributos derivados que no se requieran almacenar en la base de datos; entre las funciones más utilizadas encontramos:

| Función                 | Utilidad                                                                         |
|-------------------------|----------------------------------------------------------------------------------|
| <b>SUM(columna)</b>     | Suma o totaliza los valores contenidos dentro de la columna especificada.        |
| <b>COUNT(columna)</b>   | Cuenta el número de registros existentes a partir de una columna determinada.    |
| <b>AVERAGE(columna)</b> | Obtiene el promedio de los valores contenidos dentro de la columna especificada. |

---

<sup>13</sup> La función **TRIM** se comprobó en el RDBMS PostgreSQL v.8.2.3 ya que en MySQL 5 no se encuentra implementada.

| Función      | Utilidad                                                      |
|--------------|---------------------------------------------------------------|
| MIN(columna) | Obtiene el valor mínimo contenido en la columna especificada. |
| MAX(columna) | Obtiene el valor máximo contenido en la columna especificada. |
| POWER        | Eleva un número a la potencia indicada.                       |
| SQRT         | Obtiene la raíz cuadrada de un valor especificado.            |
| ABS          | Obtiene el valor absoluto de un número.                       |
| MOD          | Obtiene el residuo de una división.                           |

### 5.2.1 SUM

La función SUM tiene como mayor funcionalidad totalizar el valor de una columna. Por ejemplo de la tabla de ventas, tenemos la columna de Monto, si quisiéramos conocer el monto total, entonces debemos utilizar la función SUM, con la cual se sumarán todos los valores de la columna Monto.

```
SELECT SUM(CurMonto) as 'Monto Total' FROM ventas;
```

```
+-----+
| Monto Total |
+-----+
|    38967.00 |
+-----+
```

La función SUM recibe como parámetro un conjunto de valores contenidos en una columna.

### 5.2.2 COUNT

La función COUNT cuenta el número de valores contenidos en una columna. A diferencia de la función SUM, COUNT no hace más que un recuento del número de coincidencias existentes. Haciendo un COUNT sobre la misma columna de Monto de la tabla de ventas, obtenemos el siguiente resultado.

```
SELECT COUNT(curMonto) AS 'Total de Ventas' FROM ventas;
```

```
+-----+
| Total de Ventas |
+-----+
|                21 |
+-----+
```

El resultado nos indica el número de ventas existentes en la tabla de ventas.

### 5.2.3 AVERAGE

La función AVERAGE obtiene un promedio a partir de un conjunto de valores contenidos en una columna; retomando la salida de las consultas anteriores, podemos decir que en promedio, por cada venta, un cliente invierte  $38967.00 / 21$ ; una forma de obtener el resultado, es con la función AVERAGE, tal como se muestra a continuación:

```
SELECT AVG(curMonto) AS 'Promedio de Ventas' FROM ventas;
```

```
+-----+
| Promedio de Ventas |
+-----+
|    1855.571429 |
+-----+
```

### 5.3.4 MIN

La función MIN obtiene el valor mínimo contenido en un conjunto de datos, por ejemplo, si deseáramos conocer cuál fue el monto menor registrado para una venta ejecutamos:

```
SELECT MIN(curMonto) AS 'Mínimo Monto Invertido' FROM ventas;
+-----+
| Mínimo Monto Invertido |
+-----+
|           158.00 |
+-----+
```

### 5.3.5 MAX

La función MAX, obtiene el valor máximo contenido en un conjunto de datos, por ejemplo si deseáramos conocer cuál fue el monto mayor registrado para una venta ejecutamos:

```
SELECT MAX(curMonto) AS 'Máximo Monto Invertido' FROM ventas;
+-----+
| Máximo Monto Invertido |
+-----+
|           8722.00 |
+-----+
```

### 5.3.6 POWER

La función POWER permite elevar un número a una potencia determinada, por ejemplo, si deseáramos elevar el número 3 a la 5ª potencia, ejecutamos:

```
SELECT POWER(3,5);
+-----+
| POWER(3,5) |
+-----+
|          243 |
+-----+
```

La función POWER recibe dos argumentos, en primer término el número que se quiere elevar y, en segundo término, la potencia a la que se elevará el número indicado. El ejemplo anterior representa  $3^5$ .

### 5.3.7 SQRT

La función SQRT recibe como argumento un valor numérico del cual se desea obtener su raíz cuadrada.

```
SELECT SQRT(64);
+-----+
| SQRT(64) |
+-----+
|          8 |
+-----+
```

### 5.3.8 ABS

La función ABS regresa como resultado, el valor absoluto de un número proporcionado.

```
mysql> SELECT ABS(-3.1416);
+-----+
| ABS(-3.1416) |
+-----+
|          3.1416 |
+-----+
```

### 5.3.9 MOD

La función MOD obtiene el residuo a partir de la división entre dos números, el módulo resultante de dividir 26 entre 5 es 1, tal como se muestra:

```
mysql> SELECT 26 MOD 5;
```

```
+-----+
| 26 MOD 5 |
+-----+
|          1 |
+-----+
```

O bien:

```
SELECT MOD(26, 5);
```

```
+-----+
| MOD(26, 5) |
+-----+
|          1 |
+-----+
```

## 5.4 Funciones para datos tipo Fecha

El manejo de fechas siempre es un caso muy importante debido a que una base de datos almacena datos históricos, y cuando se desea consultar información en dicha base de datos, se restringe información a partir de intervalos de tiempo en donde ocurrieron hechos específicos. Por ello, es importante conocer el momento en que se almacenan los datos, es decir, conocer la fecha y hora del sistema.

Las funciones para datos tipo Fecha más comunes, son:

| Función           | Utilidad                                                                         |
|-------------------|----------------------------------------------------------------------------------|
| CURRENT_DATE      | Obtiene la fecha actual regresando un valor cuyo tipo de dato es fecha.          |
| CURRENT_TIME      | Obtiene la hora actual regresando un valor cuyo tipo de dato es hora.            |
| CURRENT_TIMESTAMP | Obtiene la fecha actual regresando un valor cuyo tipo de dato es fecha/hora.     |
| FORMAT            | Las funciones para formato de fecha varían de acuerdo con el RDBMS por utilizar. |

Ejemplos de las funciones anteriores son:

```
SELECT CURRENT_DATE();
```

```
+-----+
| CURRENT_DATE() |
+-----+
| 2007-06-17      |
+-----+
```

```
SELECT CURRENT_TIME();
```

```
+-----+
| CURRENT_TIME() |
+-----+
| 19:10:19        |
+-----+
```

```
SELECT CURRENT_TIMESTAMP();
```

```
+-----+
| CURRENT_TIMESTAMP() |
+-----+
| 2007-06-17 19:10:22 |
+-----+
```

# CAPÍTULO 6

## Tipos de JOINS

---

Si bien las subconsultas nos ayudan a recuperar información de varias tablas recuperando información de consultas previas que se hagan llegar a una consulta principal, el proceso resulta a veces complicado y no muy funcional, en tanto se deben estructurar las consultas previas.

La forma más eficiente de recuperar información de diversas tablas es a través del operador JOIN, el cual va a permitir unir dos o más tablas, siempre y cuando se mantenga una relación entre ellas, a partir de la integridad referencial que va de llaves primarias a foráneas.

El operador JOIN a diferencia de los antes estudiados, no tiene la funcionalidad de restringir información, sino que permite ampliar la visión de una tabla, mediante información proveniente de una o más tablas.

El operador JOIN tiene tres variantes, como se muestra en la siguiente tabla.

| Cláusula         | Utilidad                                                                                                                                         |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| INNER JOIN       | Permite juntar la información de dos o más tablas en dónde coincidan exactamente igual los campos de la llave primaria con los de llave foránea. |
| LEFT OUTER JOIN  | Muestra todos los campos coincidentes del lado derecho de la consulta y los no coincidentes del lado izquierdo de la consulta.                   |
| RIGHT OUTER JOIN | Muestra todos los campos coincidentes del lado izquierdo de la consulta y los no coincidentes del lado derecho de la consulta.                   |

Para la correcta comprensión del uso del operador JOIN, se ha subdividido la información en los siguientes temas.

## 6.1 Inner Join

El **INNER JOIN** permite juntar la información de dos o más tablas, mediante una referencia de integridad referencial. Por ejemplo, la tabla de ventas contiene información de las ventas de vinos realizadas a través del tiempo; las columnas que tiene esta tabla, en su mayoría son columnas con una restricción de llave foránea: *idVino*, *idCliente*, *idVendedor*; estas tres columnas hacen referencia a las tablas de vinos, clientes y vendedores respectivamente.

La funcionalidad del operador **JOIN** va encaminada a complementar la información de una tabla en donde existe una o más llaves foráneas, suponga que de la tabla de ventas requerimos solamente conocer el identificador del vino, el nombre de éste; como hasta ahora habíamos venido trabajando con las subconsultas, esto era imposible, ya que una subconsulta sólo regresa valores para ser evaluados por una consulta principal.

```
SELECT TxtClaveVentas, IdVino, intCantidad, curMonto FROM ventas;
```

| TxtClaveVentas | IdVino | intCantidad | curMonto |
|----------------|--------|-------------|----------|
| FACT0360       | 7      | 2           | 998.00   |
| FACT0361       | 7      | 1           | 499.00   |
| FACT0362       | 7      | 3           | 1497.00  |
| FACT0363       | 7      | 1           | 499.00   |
| FACT0364       | 10     | 4           | 288.00   |
| FACT0365       | 9      | 6           | 930.00   |
| FACT0366       | 3      | 7           | 8722.00  |
| FACT0367       | 8      | 8           | 576.00   |
| FACT0368       | 5      | 9           | 5310.00  |
| FACT0369       | 1      | 1           | 483.00   |
| FACT0370       | 6      | 2           | 158.00   |
| FACT0371       | 7      | 4           | 1996.00  |
| FACT0372       | 2      | 5           | 1625.00  |
| FACT0373       | 6      | 6           | 474.00   |
| FACT0374       | 3      | 3           | 3738.00  |
| FACT0375       | 2      | 4           | 1300.00  |
| FACT0376       | 5      | 1           | 590.00   |
| FACT0377       | 7      | 8           | 3992.00  |
| FACT0378       | 4      | 4           | 1300.00  |
| FACT0379       | 7      | 2           | 998.00   |
| FACT0380       | 7      | 6           | 2994.00  |

Observe la proyección anterior de la tabla de ventas que contiene el número de factura, el identificar del vino vendido, la cantidad y el monto total; suponga que deseamos generar un reporte en donde no sólo se muestre el identificador del vino, sino que a demás se conozca más información del vino vendido, registro a registro.

El operador **JOIN** proveerá la potencialidad suficiente al operador **SELECT** para cumplir con el objetivo antes descrito.

La estructura del operador **JOIN** en cualquiera de sus modalidades, es la siguiente:

```
SELECT tabla1.columna1, tabla2.columna2, tabla3.columna1, ...,
tablaN.columnaN
FROM tabla1 INNER JOIN tabla2 ON tabla1.columna1=tabla2.columna1
```

Dado que el operador **JOIN** permite obtener información de dos o más tablas, es necesario indicar en los nombres de columna que se desean proyectar cuál es su tabla de procedencia, posteriormente, se indican las tablas de donde se obtendrá la información: se coloca el nombre de la primera tabla, el operador **JOIN**, luego el nombre de la

segunda tabla y se indica a través del operador **ON**, el campo bajo el cual están referenciadas estas tablas.

Por ejemplo, tomando la proyección anterior de la tabla de ventas, suponga que deseamos conocer el nombre del vino que se vendió en cada caso, para ello, ejecutamos la siguiente instrucción **SELECT**.

```
SELECT ventas.TxtClaveVentas, ventas.idVino, vinos.TxtNombreVino,
ventas.IntCantidad, ventas.CurMonto
FROM ventas INNER JOIN vinos ON ventas.IdVino=vinos.IdVino;
```

| TxtClaveVentas | idVino | TxtNombreVino                   | IntCantidad | CurMonto |
|----------------|--------|---------------------------------|-------------|----------|
| FACT0360       | 7      | Nadal 1510 1998                 | 2           | 998.00   |
| FACT0361       | 7      | Nadal 1510 1998                 | 1           | 499.00   |
| FACT0362       | 7      | Nadal 1510 1998                 | 3           | 1497.00  |
| FACT0363       | 7      | Nadal 1510 1998                 | 1           | 499.00   |
| FACT0364       | 10     | Finca Boadella 2000             | 4           | 288.00   |
| FACT0365       | 9      | Bourgogne Vezelay 2001          | 6           | 930.00   |
| FACT0366       | 3      | Clos Vougeot 2000               | 7           | 8722.00  |
| FACT0367       | 8      | Señorío de Garcigrande, Verdejo | 8           | 576.00   |
| FACT0368       | 5      | Valtravieso Reserva 1999        | 9           | 5310.00  |
| FACT0369       | 1      | Viña Lanciano Reserva 1998      | 1           | 483.00   |
| FACT0370       | 6      | Chablis Cuvée LC 1999           | 2           | 158.00   |
| FACT0371       | 7      | Nadal 1510 1998                 | 4           | 1996.00  |
| FACT0372       | 2      | Valtravieso Crianza 2002        | 5           | 1625.00  |
| FACT0373       | 6      | Chablis Cuvée LC 1999           | 6           | 474.00   |
| FACT0374       | 3      | Clos Vougeot 2000               | 3           | 3738.00  |
| FACT0375       | 2      | Valtravieso Crianza 2002        | 4           | 1300.00  |
| FACT0376       | 5      | Valtravieso Reserva 1999        | 1           | 590.00   |
| FACT0377       | 7      | Nadal 1510 1998                 | 8           | 3992.00  |
| FACT0378       | 4      | Valtravieso Crianza 2002        | 4           | 1300.00  |
| FACT0379       | 7      | Nadal 1510 1998                 | 2           | 998.00   |
| FACT0380       | 7      | Nadal 1510 1998                 | 6           | 2994.00  |

El resultado anterior es posible gracias a la incursión del operador **JOIN** en la consulta, que permitió complementar la información de la tabla *ventas* con la información de la tabla de vinos; el complemento de la información es posible gracias a la existencia de una llave foránea en la tabla de ventas en la columna de *idVino*, la cual hace referencia a la tabla de vinos.

En primer término, se utiliza la palabra reservada **SELECT**, posteriormente se listan los nombres de las columnas que deseamos aparezcan en el resultado final, dichas columnas pueden provenir de cualquiera de las tablas por juntar; después se utiliza la palabra reservada **FROM** para indicar la procedencia de los datos, en primer lugar queremos información de la tabla de ventas, la cual es complementada por la tabla de vinos, dicho complemento se logra utilizando la palabra reservada **INNER JOIN**, que a su vez complementa la información de las tablas tomando como referencia el atributo indicado seguido de la palabra reservada **ON**.

Una forma manual de explicar este fenómeno, es que cuando se toma la tabla de ventas, en el primer registro, se indica la venta del vino número 7; el RDBMS entonces va a la tabla de vinos y consulta por el vino 7, obtiene la información del nombre del vino y lo junta a la tabla de ventas. Recuerde que las tablas originales no se modifican, siempre que haga una consulta, el resultado es una nueva tabla alojada en memoria, aquí se cumple con la propiedad de cierre de álgebra relacional.

Siempre que se realiza una consulta es posible acceder a todos los campos de las tablas seleccionadas, no se confunda, las columnas proyectadas son las que se muestran en la salida de información, pero es posible utilizar otras columnas diferentes para restringir la información.

Ahora, continuando con la implementación del operador JOIN en la modalidad INNER JOIN, realizaremos el mismo ejemplo haciendo uso de alias para renombrar nombres de tablas, a su vez restringiremos información a través de una columna que no será presentada en la salida de información, para ello, tomaremos como base el ejemplo anterior.

De esta manera requerimos de una consulta que muestre la siguiente información: número de factura, nombre del vino vendido y cantidad de artículos vendidos, en donde el vendedor sea el número 301.

```
SELECT ventas.TxtClaveVentas, vinos.TxtNombreVino, ventas.IntCantidad
FROM ventas INNER JOIN vinos ON ventas.IdVino=vinos.IdVino
WHERE ventas.idVendedor=301;
```

| TxtClaveVentas | TxtNombreVino              | IntCantidad |
|----------------|----------------------------|-------------|
| FACT0360       | Nadal 1510 1998            | 2           |
| FACT0369       | Viña Lanciano Reserva 1998 | 1           |
| FACT0374       | Clos Vougeot 2000          | 3           |
| FACT0375       | Valtravieso Crianza 2002   | 4           |

En primer lugar proyectamos las columnas que queremos mostrar, luego las tablas de donde obtendremos la información, esto es posible gracias al operador JOIN y a que indicamos a través del operador ON las columnas en que hacen coincidencia estas dos tablas; por último, restringimos la información para presentar sólo las ventas del vendedor 301.

Como es posible apreciar, el código SQL se incrementa de manera considerable al momento de anteponer el nombre de la tabla en el nombre de la columna, lo cual es posible acortarlo un poco a través de la utilización de un alias, ya que si utilizamos un alias podemos cambiar el nombre de la tabla 'vinos' asignándole 'vi' como nombre, y en lugar de anteponer el nombre ventas en los nombres de columna, podemos anteponer el nuevo nombre de la tabla ventas, es decir, 've'. Observe el nuevo código SQL implementando alias.

```
SELECT ve.TxtClaveVentas, vi.TxtNombreVino, ve.IntCantidad
FROM ventas AS ve INNER JOIN vinos AS vi ON ve.IdVino=vi.IdVino
WHERE ve.idVendedor=301;
```

| TxtClaveVentas | TxtNombreVino              | IntCantidad |
|----------------|----------------------------|-------------|
| FACT0360       | Nadal 1510 1998            | 2           |
| FACT0369       | Viña Lanciano Reserva 1998 | 1           |
| FACT0374       | Clos Vougeot 2000          | 3           |
| FACT0375       | Valtravieso Crianza 2002   | 4           |

El resultado obviamente es el mismo, el uso de alias únicamente permite la simplificación de código, note que en vez de utilizar los nombres completos de las tablas de ventas y vinos, utilizamos un renombre de ellas: ve y vi.

Los elementos que retomamos para el uso del operador `JOIN` son las proyecciones, restricciones, alias, operadores. No es complejo el funcionamiento del operador `JOIN`, una vez que se tiene una base sólida respecto de los elementos antes mencionados.

Ahora bien, vamos a complementar la información de la consulta anterior, agregando el nombre del cliente que realizó dichas compras. Juntaremos a las tablas de ventas y vinos, la tabla de clientes para obtener el nombre del cliente que compró el producto; juntar estas tres tablas será posible gracias a que la tabla de ventas hace referencia a la tabla de clientes a través de la columna de `idCliente`.

Seleccionaremos de las tablas de ventas, vinos y clientes, las columnas de clave de ventas, nombre de cliente y nombre de vino adquirido, en donde el vendedor tiene la clave 301.

```
SELECT ve.TxtClaveVentas, cli.txtNombreCliente, vi.TxtNombreVino
FROM ventas AS ve
INNER JOIN vinos AS vi ON ve.IdVino=vi.IdVino
INNER JOIN cliente as cli on cli.idCliente=ve.idCliente
WHERE ve.idVendedor=301;
```

| TxtClaveVentas | txtNombreCliente                | TxtNombreVino              |
|----------------|---------------------------------|----------------------------|
| FACT0360       | Vinos y Licores del Oeste       | Nadal 1510 1998            |
| FACT0369       | El rincón de la Luna Restaurant | Viña Lanciano Reserva 1998 |
| FACT0374       | San Carlos Restaurant           | Clos Vougeot 2000          |
| FACT0375       | El rincón de la Luna Restaurant | Valtravieso Crianza 2002   |

Para la consulta anterior, se utilizan dos operadores `JOIN` debido a que el número de operadores `JOIN` por utilizar en una consulta, será igual al número de tablas menos 1, fíjese que para unir la tabla de vinos y ventas utilizamos sólo un operador `JOIN`, mientras que para unir las tablas de ventas, vinos y clientes, utilizamos dos operadores `JOIN`.

La forma en que se interpreta la instrucción SQL anterior es que en primer lugar se proyectan diversos atributos provenientes de diversas tablas, posteriormente se juntan las tabla de ventas y vinos; el RDBMS interpreta que una vez que se han juntado esas dos tablas, debe juntar la tabla de clientes, es decir, el resultado es incremental, en tanto las tablas se van juntando una a una formando un complemento entre todas.

Si en vez de proyectar un número de columnas determinado por el usuario, se utiliza el comodín `*`, se indicará al RDBMS que se mostrarán todas las columnas de todas las tablas seleccionadas, mire un ejemplo de ello, al juntar las tablas de procedencia y vinos.

```
SELECT * FROM vinos AS v INNER JOIN Procedencia as P on
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| idVino | TxtNombreVino                   | TxtClaveProcedencia | curPrecio | txtClaveProcedencia | TxtDescripcion |
|--------|---------------------------------|---------------------|-----------|---------------------|----------------|
| 1      | Viña Lanciano Reserva 1998      | ESP                 | 483.00    | ESP                 | España         |
| 2      | Valtravieso Crianza 2002        | ESP                 | 325.00    | ESP                 | España         |
| 3      | Clos Vougeot 2000               | FRA                 | 1246.00   | FRA                 | Francia        |
| 4      | Valtravieso Crianza 2002        | ESP                 | 325.00    | ESP                 | España         |
| 5      | Valtravieso Reserva 1999        | ESP                 | 590.00    | ESP                 | España         |
| 6      | Chablis Cuvée LC 1999           | FRA                 | 79.00     | FRA                 | Francia        |
| 7      | Nadal 1510 1998                 | ESP                 | 499.00    | ESP                 | España         |
| 8      | Señorío de Garcigrande, Verdejo | ESP                 | 72.00     | ESP                 | España         |
| 9      | Bourgogne Vezelay 2001          | FRA                 | 155.00    | FRA                 | Francia        |
| 10     | Finca Boadella 2000             | ESP                 | 72.00     | ESP                 | España         |
| 11     | Saint Véran La Solitude 2001    | FRA                 | 220.00    | FRA                 | Francia        |
| 12     | Pouilly Fuissé 2001             | FRA                 | 305.00    | FRA                 | Francia        |

Como puede notar se juntaron todas las columnas contenidas en la tabla de procedencia y de la tabla de vinos, observe que se repite la columna TxtClaveProcedencia; esto debido a que es parte de ambas tablas, por ello, cuando se restringe información se debe especificar la procedencia de la columna por proyectar, ya que pudiera existir la posibilidad de proyectar una columna ambigua debido a que está contenida en ambas tablas.

Con la nueva información que acabamos de obtener, podemos concluir que es necesario indicar la procedencia de las columnas siempre y cuando éstas puedan resultar ambiguas por estar contenidas en más de una tabla de las que se hace referencia. Ejemplo:

```
SELECT TxtNombreVino, TxtClaveProcedencia, TxtDescripcion FROM vinos AS v
INNER JOIN Procedencia as P on P.TxtClaveProcedencia=v.TxtClaveProcedencia;
ERROR 1052 (23000): Column 'TxtClaveProcedencia' in field list is ambiguous
```

Ejemplo:

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion FROM vinos AS v
INNER JOIN Procedencia as P on P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Finca Boadella 2000             | ESP                 | España         |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |

Suponga que deseamos juntar las tablas de ventas, vinos, cliente y procedencia con el objeto de generar una consulta que muestre el número de factura, el nombre del cliente, el nombre del vino y su procedencia, para las ventas realizadas por el vendedor 304.

Dado que son cuatro tablas las que vamos a utilizar, por ende utilizaremos tres operadores JOIN.

```
SELECT ve.TxtClaveVentas, cli.txtNombreCliente, vi.TxtNombreVino,
p.TxtDescripcion
FROM ventas AS ve
INNER JOIN vinos AS vi ON ve.IdVino=vi.IdVino
INNER JOIN cliente as cli on cli.idCliente=ve.idCliente
INNER JOIN procedencia as p ON p.TxtClaveProcedencia=vi.TxtClaveProcedencia
WHERE ve.idVendedor=301;
```

| TxtClaveVentas | txtNombreCliente                | TxtNombreVino              | TxtDescripcion |
|----------------|---------------------------------|----------------------------|----------------|
| FACT0360       | Vinos y Licores del Oeste       | Nadal 1510 1998            | España         |
| FACT0369       | El rincón de la Luna Restaurant | Viña Lanciano Reserva 1998 | España         |
| FACT0374       | San Carlos Restaurant           | Clos Vougeot 2000          | Francia        |
| FACT0375       | El rincón de la Luna Restaurant | Valtravieso Crianza 2002   | España         |

En la consulta anterior, se proyectan las columnas de clave de venta, nombre de cliente, nombre de vino y procedencia posteriormente se van juntando las tablas; y se toma como base la tabla de ventas, a ella a su vez, se junta la tabla de cliente, dado que en ellas existe un atributo común llamado idCliente, una vez que se han unido esas dos tablas, se

junta la tabla de vinos, en tanto comparten un atributo en común, el id de vino: una vez que se han juntado estas tres tablas, se junta la tabla de procedencia.

Hacer consultas con el operador **JOIN** resulta de gran utilidad para generar reportes completos para obtener información proveniente de dos o más tablas, mostrando el detalle de la tabla que contiene las columnas de la llave foránea.

Siempre que trabaje con el operador **JOIN** es casi un hecho que requiera hacer proyecciones, restricciones, uso de operadores y de alias, así como de funciones de agregado y grupos, temas que se verán más adelante.

Ahora que ha comprendido el funcionamiento del operador **JOIN** en su variante **INNER JOIN**, le resultará muy sencillo comprender el funcionamiento de las otras dos variantes: el **LEFT OUTER JOIN** y el **RIGHT OUTER JOIN**.

## 6.2 Outer Joins (Right Join – Left Join).

Como se mencionó en el tema anterior, el operador **JOIN** va a permitir juntar información de dos o más tablas con el objeto de complementar información entre ellas; la primera variante del operador **JOIN** es el **INNER JOIN**, el cual va a juntar dos o más tablas para regresar como resultado, aquellos casos que implican una coincidencia, recordemos la siguiente consulta en donde se unen las tablas de vinos y procedencia.

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion
FROM vinos AS v INNER JOIN Procedencia as P on
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Finca Boadella 2000             | ESP                 | España         |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |

En el resultado mostrado se proporciona el nombre del vino y su lugar de procedencia, nótese que los lugares de procedencia son únicamente España y Francia ya que no hay registrados otros vinos donde el lugar de origen sea Italia o Argentina, los cuales están registrados como lugar de procedencia en la tabla de procedencia.

En el caso de que quisiera conocer toda la información de la tabla de procedencia juntándola con la información de la tabla de vinos, no sólo en donde coincidan los datos, se utilizará un operador **OUTER JOIN**, en alguna de las dos variables **LEFT** o **RIGHT**.

Al tipo de consulta que nos referimos es la siguiente, observe los resultados y posteriormente detallaremos la estructura de la consulta utilizada.

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion
FROM vinos AS v RIGHT OUTER JOIN Procedencia as P on
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Finca Boadella 2000             | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |
| NULL                            | AUS                 | Australia      |
| NULL                            | ITA                 | Italia         |
| NULL                            | ESC                 | Escocia        |
| NULL                            | ARG                 | Argentina      |
| NULL                            | FIN                 | Finlandia      |
| NULL                            | ING                 | Inglaterra     |

Como habíamos mencionado para el caso del operador INNER JOIN, éste complementa información de las tablas siempre y cuando exista una coincidencia en los valores de los campos; en el primer caso, utilizando el operador INNER JOIN, se mostraron todos los vinos y se complementó su información con la tabla de procedencia, para complementar la información se buscó que la clave de procedencia existiera en la tabla de procedencia, dado que existió FRA y ESP, entonces mostró el campo de descripción.

Para el segundo caso, utilizando el operador OUTER JOIN en su variante RIGHT OUTER JOIN, se mostró la información que coincidía entre las tablas de vinos y procedencia, las concordancias fueron a través de las claves ESP y FRA; sin embargo a demás de las coincidencias, el operador RIGHT OUTER JOIN, muestra forzosamente los valores que no obtuvieron una coincidencia de la tabla de procedencia con la tabla de vinos; es decir, se mostraron todos los lugares de procedencia independientemente de que tuvieran o no una coincidencia en la tabla de vinos:

|      |     |            |
|------|-----|------------|
| NULL | AUS | Australia  |
| NULL | ITA | Italia     |
| NULL | ESC | Escocia    |
| NULL | ARG | Argentina  |
| NULL | FIN | Finlandia  |
| NULL | ING | Inglaterra |

Como se puede observar cuando se listan los lugares de procedencia que no tuvieron concordancia en la tabla de vinos, entonces, se deja nulo el campo ya que no hay vinos dados de alta para esos lugares de procedencia. Para la consulta anterior, se muestran todos los lugares de procedencia independientemente de que tenga un vino dado de alta en la tabla de vinos, por ello, el nombre del vino tiene un valor nulo.

Se utilizó el operador RIGHT OUTER JOIN ya que de las dos tablas de las cuales se recupera información, la tabla de procedencia se encuentra a la derecha del operador

JOIN y la tabla de vinos a la izquierda del operador JOIN, observe de nuevo esa parte de la consulta:

```
FROM vinos AS v RIGHT OUTER JOIN Procedencia as P on
```

¿Qué ocurre si a la misma consulta aplicamos un LEFT OUTER JOIN?

Muy simple, se van a mostrar todos los vinos con su procedencia y además, se van a listar los vinos que no tienen indicada una procedencia.

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion
FROM vinos AS v LEFT OUTER JOIN Procedencia as P on
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Finca Boadella 2000             | ESP                 | España         |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |

El resultado es igual al generado por el operador INNER JOIN ya que no existen vinos a los que no se haya indicado su lugar de procedencia; todos los vinos cuentan con una clave de procedencia, en el supuesto caso de que un vino no tuviera indicada la procedencia, aparecería un resultado como el que se muestra a continuación.

|                     |      |      |
|---------------------|------|------|
| Pouilly Fuissé 2001 | NULL | NULL |
|---------------------|------|------|

Para la consulta anterior, se mostrarían todos los vinos aunque no tuvieran lugar de procedencia.

Es importante evaluar el lugar dónde se colocan los nombres de las tablas, si a la derecha o a la izquierda del operador JOIN, ya que esto será fundamental cuando se quiera aplicar un LEFT o RIGHT OUTER JOIN.

Vamos a utilizar un LEFT OUTER JOIN cambiando el orden de las tablas para mostrar todos los lugares de procedencia independientemente que tengan un vino dado de alta en la tabla de vinos; dicha consulta ya se ha resuelto con el operador RIGHT OUTER JOIN, sin embargo, el objetivo es demostrar la importancia de determinar el orden en que aparecen las tablas y el operador OUTER JOIN por utilizar.

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion
FROM procedencia as P LEFT OUTER JOIN vinos AS v ON
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Finca Boadella 2000             | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |
| NULL                            | AUS                 | Australia      |
| NULL                            | ITA                 | Italia         |
| NULL                            | ESC                 | Escocia        |
| NULL                            | ARG                 | Argentina      |
| NULL                            | FIN                 | Finlandia      |
| NULL                            | ING                 | Inglaterra     |

```
SELECT TxtNombreVino, P.TxtClaveProcedencia, TxtDescripcion
FROM vinos AS v RIGHT OUTER JOIN Procedencia as P on
P.TxtClaveProcedencia=v.TxtClaveProcedencia;
```

| TxtNombreVino                   | TxtClaveProcedencia | TxtDescripcion |
|---------------------------------|---------------------|----------------|
| Viña Lanciano Reserva 1998      | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Crianza 2002        | ESP                 | España         |
| Valtravieso Reserva 1999        | ESP                 | España         |
| Nadal 1510 1998                 | ESP                 | España         |
| Señorío de Garcigrande, Verdejo | ESP                 | España         |
| Finca Boadella 2000             | ESP                 | España         |
| Clos Vougeot 2000               | FRA                 | Francia        |
| Chablis Cuvée LC 1999           | FRA                 | Francia        |
| Bourgogne Vezelay 2001          | FRA                 | Francia        |
| Saint Véran La Solitude 2001    | FRA                 | Francia        |
| Pouilly Fuissé 2001             | FRA                 | Francia        |
| NULL                            | AUS                 | Australia      |
| NULL                            | ITA                 | Italia         |
| NULL                            | ESC                 | Escocia        |
| NULL                            | ARG                 | Argentina      |
| NULL                            | FIN                 | Finlandia      |
| NULL                            | ING                 | Inglaterra     |

El resultado es igual para ambos casos, ya que el orden de la tablas afecta la utilización de un LEFT o un RIGHT OUTER JOIN, por ello, es de suma importancia determinar qué operador OUTER JOIN se utilizará, esto tomando como base la posición que ocupan las tablas, ya sea a la derecha o a la izquierda del operador OUTER JOIN.

Con el objeto de aterrizar este tema, realizaremos una consulta que muestre los nombres de todos los vendedores independientemente de que hayan realizado o no una venta<sup>14</sup>:

```
SELECT TxtClaveVentas, TxtNombreVendedor FROM ventas RIGHT OUTER JOIN
vendedores on vendedores.idVendedor=ventas.IdVendedor;
```

<sup>14</sup> Para obtener el mismo resultado deberá agregar al vendedor Jorge Luis Reyes García cuya clave es la 306.

| TxtClaveVentas | TxtNombreVendedor       |
|----------------|-------------------------|
| FACT0360       | Fernando Gómez Galicia  |
| FACT0369       | Fernando Gómez Galicia  |
| FACT0374       | Fernando Gómez Galicia  |
| FACT0375       | Fernando Gómez Galicia  |
| FACT0363       | Jennifer Aguilar Cortés |
| FACT0365       | Jennifer Aguilar Cortés |
| FACT0366       | Jennifer Aguilar Cortés |
| FACT0370       | Jennifer Aguilar Cortés |
| FACT0373       | Jennifer Aguilar Cortés |
| FACT0376       | Jennifer Aguilar Cortés |
| FACT0378       | Jennifer Aguilar Cortés |
| FACT0364       | Dario González Henkel   |
| FACT0367       | Dario González Henkel   |
| FACT0368       | Dario González Henkel   |
| FACT0379       | Dario González Henkel   |
| FACT0362       | Carlos Martínez Salinas |
| FACT0372       | Carlos Martínez Salinas |
| FACT0377       | Carlos Martínez Salinas |
| FACT0361       | Arabia Leñero Orozco    |
| FACT0371       | Arabia Leñero Orozco    |
| FACT0380       | Arabia Leñero Orozco    |
| NULL           | Jorge Luis Reyes García |

Observe que como resultado último, se muestra al vendedor Jorge Luis Reyes García, quien no ha realizado ninguna venta porque en el campo de factura aparece un valor nulo.

Por último, recalcaremos una vez más la diferencia entre el OUTER JOIN y el INNER JOIN, ejecutaremos la misma consulta anterior, sólo que ahora utilizando un INNER JOIN.

```
SELECT TxtClaveVentas, TxtNombreVendedor FROM ventas INNER JOIN vendedores
on vendedores.idVendedor=ventas.IdVendedor;
```

| TxtClaveVentas | TxtNombreVendedor       |
|----------------|-------------------------|
| FACT0360       | Fernando Gómez Galicia  |
| FACT0369       | Fernando Gómez Galicia  |
| FACT0374       | Fernando Gómez Galicia  |
| FACT0375       | Fernando Gómez Galicia  |
| FACT0363       | Jennifer Aguilar Cortés |
| FACT0365       | Jennifer Aguilar Cortés |
| FACT0366       | Jennifer Aguilar Cortés |
| FACT0370       | Jennifer Aguilar Cortés |
| FACT0373       | Jennifer Aguilar Cortés |
| FACT0376       | Jennifer Aguilar Cortés |
| FACT0378       | Jennifer Aguilar Cortés |
| FACT0364       | Dario González Henkel   |
| FACT0367       | Dario González Henkel   |
| FACT0368       | Dario González Henkel   |
| FACT0379       | Dario González Henkel   |
| FACT0362       | Carlos Martínez Salinas |
| FACT0372       | Carlos Martínez Salinas |
| FACT0377       | Carlos Martínez Salinas |
| FACT0361       | Arabia Leñero Orozco    |
| FACT0371       | Arabia Leñero Orozco    |
| FACT0380       | Arabia Leñero Orozco    |

Como se observa, en el INNER JOIN sólo aparecen los vendedores que realizaron ventas, es decir, únicamente existieron coincidencias en ambas tablas, mientras que un OUTER JOIN muestra no sólo los registros que implican una coincidencia sino que muestra también aquellos registros de la tabla de la derecha o izquierda sin coincidencias.

### 6.3 Self Join

El Self Join es una variante muy particular del operador **JOIN**, ya que su objetivo principal es obtener información adicional utilizando una sola tabla; es usado un Self Join cuando por motivos de diseño de la base de datos se desea consultar información que está autorelacionada consigo misma.

Suponga una tabla de empleados cuyo objetivo principal es almacenar la información de los empleados de una compañía. Así pues y de manera breve suponga la siguiente estructura de la tabla.

| empleados |                                     |
|-----------|-------------------------------------|
| <b>PK</b> | <u><b>idEmpleado</b></u>            |
|           | TxtNombreEmpleado<br>idJefeEmpleado |

Tabla de empleados.

La tabla anterior contiene tres campos: un campo con llave primaria para identificar a los empleados, un campo para almacenar su nombre y un tercer campo que es el identificador de empleado de su jefe.

Ahora bien, almacenando registros en la tabla y haciendo una consulta simple, la información de la tabla es la siguiente:

```
SELECT * FROM empleados;
```

| idEmpleado | TxtNombreEmpleado            | idEmpleadoJefe |
|------------|------------------------------|----------------|
| 1          | Carlos Francisco Méndez Cruz | NULL           |
| 2          | Margarita Santibañez Mejía   | 1              |

Por el momento, sólo contamos con dos empleados: Carlos y Margarita, y la información presentada en la tabla nos dice que el jefe de Margarita es Carlos Méndez. Si se requiriera ver inmediatamente el nombre del jefe de Margarita sin consultar qué empleado tiene el id 1, entonces es cuando se explota la funcionalidad del Self Join.

Una consulta Self Join se realiza utilizando el operador **INNER JOIN** acompañado de un alias de tabla. El objetivo de la consulta es hacer un **JOIN** entre una tabla y consigo misma, para así presentar la información de manera clara.

El ejemplo de una consulta Self Join es el siguiente:

```
SELECT templeado1.idEmpleado, templeado1.TxtNombreEmpleado,
templeado2.TxtNombreEmpleado AS Jefe
FROM empleados AS templeado1
INNER JOIN empleados AS templeado2 on
templeado1.idEmpleadoJefe=templeado2.idEmpleado;
```

| idEmpleado | TxtNombreEmpleado          | Jefe                         |
|------------|----------------------------|------------------------------|
| 2          | Margarita Santibañez Mejía | Carlos Francisco Méndez Cruz |

La consulta anterior fue satisfactoria ya que la misma tabla de empleados se complementó para obtener el resultado correcto, así pues, decimos que un Self Join obtiene información de una tabla que es relacionada consigo misma a través de un operador `JOIN`.

Para realizar la consulta suponemos que vamos a utilizar la tabla de empleados y a su vez una copia de la misma, por ello, asignamos los alias `empleados1` y `empleados2`, es como si se tuvieran dos tablas de empleados. En la primera tabla de la consulta se proyecta información del `id` del empleado de la tabla 1 así como su nombre, la tercera columna es el nombre del empleado de la tabla 2 y se renombra a `Jefe`.

Cuando se hace el Join se dice que se desea obtener información de la tabla de empleados, se renombra a `empleados1` y se junta esta tabla con la tabla de empleados que es renombrada como `empleados2`, lo que da un efecto de utilizar dos tablas; la relación que se establece entre las tablas se da a partir de las columnas `idEmpleado` y `idEmpleadoJefe`, es como si `idEmpleadoJefe` tuviera una llave foránea que hace referencia a la tabla `empleados1` en la columna `idEmpleado`.

`empleados1`

| <code>idEmpleado</code> | <code>TxtNombreEmpleado</code> | <code>idEmpleadoJefe</code> |
|-------------------------|--------------------------------|-----------------------------|
|                         |                                |                             |
|                         |                                |                             |

`empleados2`

| <code>idEmpleado</code> | <code>TxtNombreEmpleado</code> | <code>idEmpleadoJefe</code> |
|-------------------------|--------------------------------|-----------------------------|
|                         |                                |                             |
|                         |                                |                             |

Como se puede notar se hace una copia virtual de la tabla de empleados, éstas se renombran y se establece una relación entre ellas a través de dos columnas diferentes. Así es posible expresar una relación de una tabla consigo misma, haciendo además, uso del operador `JOIN`.

## 6.4 Agrupación de información y funciones de agregado

Ahora que se ha mostrado la funcionalidad del operador `JOIN`, el cual permite recuperar información de dos o más tablas, ampliaremos su potencialidad al agrupar la información y ejecutar funciones de agregado sobre ella.

El agrupamiento de información se realiza principalmente debido a que se desea presentar información concentrada, dejándose de lado el enfoque del detalle de las cosas para presentar una visión más abstracta de los datos.

Cuando se atraviesa por un proceso para la toma de decisiones, se desea conocer información consolidada en grupos que permitan de manera rápida y fácil obtener una visión amplia del comportamiento del modelo de negocio de una organización.

Por ejemplo, suponga que en vez de ver un listado de 150 ventas de productos, únicamente se muestre en un reporte gerencial, el número de compras que hicieron los clientes, o mejor aún, la cantidad de dinero que invirtieron los clientes en nuestros productos al final de un mes.

Cuando se agrupa la información, el detalle de la misma queda un poco de lado, con el objeto de presentar una visión abstracta de los datos.

Una vez que se ha hecho una consulta utilizando **JOIN** es como podemos ampliar ésta para implementar el agrupamiento de información. Para formar grupos en las consultas, se utilizará la cláusula SQL **GROUP BY** que permitirá indicar a través de qué columnas de la consulta se agrupará la información.

Como primer ejemplo obtendremos a través de una consulta, el número de compras que históricamente han realizado los clientes. Comenzaremos haciendo una consulta con **JOIN** como hasta ahora sabemos hacerlo, en dicha consulta mostraremos el número de factura, el nombre del cliente y la cantidad de artículos que adquirió en su compra.

```
SELECT TxtClaveVentas, TxtNombreCliente, intCantidad
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente;
```

| TxtClaveVentas | TxtNombreCliente                     | intCantidad |
|----------------|--------------------------------------|-------------|
| FACT0360       | Vinos y Licores del Oeste            | 2           |
| FACT0363       | Vinos y Licores del Oeste            | 1           |
| FACT0367       | Vinos y Licores del Oeste            | 8           |
| FACT0370       | Vinos y Licores del Oeste            | 2           |
| FACT0371       | Vinos y Licores del Oeste            | 4           |
| FACT0361       | El rincón de la Luna Restaurant      | 1           |
| FACT0365       | El rincón de la Luna Restaurant      | 6           |
| FACT0369       | El rincón de la Luna Restaurant      | 1           |
| FACT0375       | El rincón de la Luna Restaurant      | 4           |
| FACT0377       | El rincón de la Luna Restaurant      | 8           |
| FACT0378       | El rincón de la Luna Restaurant      | 4           |
| FACT0379       | El rincón de la Luna Restaurant      | 2           |
| FACT0366       | Comercializadora Emporio de Vinicola | 7           |
| FACT0368       | Comercializadora Emporio de Vinicola | 9           |
| FACT0372       | Comercializadora Emporio de Vinicola | 5           |
| FACT0373       | Comercializadora Emporio de Vinicola | 6           |
| FACT0380       | Comercializadora Emporio de Vinicola | 6           |
| FACT0364       | San Carlos Restaurant                | 4           |
| FACT0374       | San Carlos Restaurant                | 3           |
| FACT0362       | Productos vinicolas Esfinge          | 3           |
| FACT0376       | Productos vinicolas Esfinge          | 1           |

La consulta muestra información de la tabla de ventas complementada con la información de la tabla de clientes, en dicha consulta se muestra la clave de la venta, el nombre del cliente y la cantidad de vinos que compró.

Si ahora deseáramos resumir la información mostrando únicamente el nombre del cliente y la cantidad de productos que en toda su historia ha comprado, entonces, debemos implementar un grupo y una función de agregado.

Retomando la consulta anterior, haremos algunas modificaciones e implementaremos un agrupamiento de información.

La consulta original es:

```
SELECT TxtClaveVentas, TxtNombreCliente, intCantidad
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente;
```

Una consulta que implementa un agrupamiento de información, se hace a través de la cláusula **GROUP BY**, la cual se agrega después de las restricciones de información, dado que en este caso no existen restricciones, pues la pondremos al final simplemente.

```
SELECT TxtClaveVentas, TxtNombreCliente, intCantidad
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente;
```

Seguido de la palabra reservada **GROUP BY** debemos incluir el nombre de la columnas que formarán los grupos, en nuestro caso requerimos agrupar la información a través de los clientes, por ello, luego de **GROUP BY** colocamos el nombre de la columna que almacena los nombres de los clientes.

Hasta este momento, la consulta no tiene funcionalidad, esto se debe a que por regla cuando se agrupe información, las columnas por proyectar deberán aparecer en su totalidad en la parte donde se indica el grupo, a menos que se trate de una función de agregado.

Esto significa que no podemos proyectar las columnas **TxtClaveVentas** e **intCantidad**, en tanto no especificamos que forman parte del grupo, la única columna que forma parte del grupo es **TxtNombreCliente**; si ejecutamos así la consulta, el RDBMS va a regresar un mensaje de error, por lo tanto debemos quitar las columnas que no forman parte del **GROUP BY**.

```
SELECT TxtNombreCliente
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente;
```

Si ejecutáramos así la instrucción, obtendríamos una tabla con los nombres de todos los clientes que han realizado una compra, más sin embargo, aún no conocemos el número de productos históricamente adquiridos.

```
+-----+
| TxtNombreCliente |
+-----+
| Comercializadora Emporio de Vinicola |
| El rincón de la Luna Restaurant      |
| Productos vinicolas Esfinge         |
| San Carlos Restaurant                |
| Vinos y Licores del Oeste           |
+-----+
```

En este primer bosquejo, hemos logrado compactar la información de la tabla de ventas agrupando los nombres de todos los clientes que han comprado, para conocer información extra, debemos utilizar una función de agregado.

Una función de agregado permite complementar información presentada cuando se agrupan datos, es decir, una función de agregado resumirá el detalle de información que se desea ocultar. En nuestro caso el detalle de información por ocultar, es la cantidad de vinos que adquirió en cada venta; por el contrario, se desea resumir la información de manera que se obtenga una totalización de productos que ha adquirido el cliente a través del tiempo.

Las funciones de agregado más comunes son:

| Función de agregado | Utilidad                                                                         |
|---------------------|----------------------------------------------------------------------------------|
| SUM(columna)        | Suma o totaliza los valores contenidos dentro de la columna especificada.        |
| COUNT(columna)      | Cuenta el número de registros existentes a partir de una columna determinada.    |
| AVERAGE(columna)    | Obtiene el promedio de los valores contenidos dentro de la columna especificada. |
| MIN(columna)        | Obtiene el valor mínimo contenido en la columna especificada.                    |
| MAX(columna)        | Obtiene el valor máximo contenido en la columna especificada.                    |

Retomando nuestro caso de ejemplo y ahora que conocemos sobre más sobre las funciones de agregado, entonces complementaremos nuestra consulta.

```
SELECT TxtNombreCliente
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente;
```

Ya que contamos con los clientes agrupados, lo siguiente es utilizar una función de agregado que totalice las compras, el número de vinos adquiridos a lo largo del tiempo; la función que implementaremos es SUM, la cual sumará los valores contenidos en la columna intCantidad.

```
SELECT TxtNombreCliente, SUM(intCantidad) as Vinos_Aquiridos
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente;
```

| TxtNombreCliente                     | Vinos_Aquiridos |
|--------------------------------------|-----------------|
| Comercializadora Emporio de Vinicola | 33              |
| El rincón de la Luna Restaurant      | 26              |
| Productos vinicolas Esfinge          | 4               |
| San Carlos Restaurant                | 7               |
| Vinos y Licores del Oeste            | 17              |

Ahora hay que destacar que la columna *Vinos\_Aquiridos* no se incluyó dentro de las columnas del GROUP BY, ya que esta es una función de agregado, la cual sí se puede proyectar sin incluir en la cláusula GROUP BY.

Las cifras no mienten, a lo largo de la historia quien más vinos ha comprado a la empresa es la Comercializadora Emporio Vinícola. Ahora quizás se requiera complementar la información presentando la información ordenada de acuerdo con el número de vinos adquiridos, ello para facilitar la lectura de los datos.

```
SELECT TxtNombreCliente, SUM(intCantidad) as Vinos_Aquiridos
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente
ORDER BY Vinos_Aquiridos DESC;
```

| TxtNombreCliente                     | Vinos_Aquiridos |
|--------------------------------------|-----------------|
| Comercializadora Emporio de Vinicola | 33              |
| El rincón de la Luna Restaurant      | 26              |
| Vinos y Licores del Oeste            | 17              |
| San Carlos Restaurant                | 7               |
| Productos vinicolas Esfinge          | 4               |

Es interesante ver que la Comercializadora Emporio Vinícola es la que más vinos ha adquirido en la historia de nuestra compañía, pero quizás no sea quien más dinero haya invertido, ya que los precios de los vinos son variados; para conocer cuánto dinero han invertido los clientes en la compra de vinos, totalizaremos la columna de monto, la cual contiene el monto por el que se facturó en cada compra a los clientes.

Totalizando la columna de monto, sabremos quién es nuestro mejor cliente.

```
SELECT TxtNombreCliente, SUM(intCantidad) as Vinos_Adquiridos, SUM(CurMonto)
AS Inversion
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente
ORDER BY Inversion DESC;
```

| TxtNombreCliente                     | Vinos_Adquiridos | Inversion |
|--------------------------------------|------------------|-----------|
| Comercializadora Emporio de Vinicola | 33               | 19125.00  |
| El rincón de la Luna Restaurant      | 26               | 9502.00   |
| Vinos y Licores del Oeste            | 17               | 4227.00   |
| San Carlos Restaurant                | 7                | 4026.00   |
| Productos vinicolas Esfinge          | 4                | 2087.00   |

Ahora que conocemos la inversión que los clientes han hecho en la compra de vinos, agregamos valor a la información presentada, la compactamos para la toma de decisiones. Aunado a los datos que tenemos sería interesante conocer el número de compras que los clientes han hecho, en tanto el cliente Comercializadora Emporio Vinícola pudo haber adquirido esa cantidad de vinos en una sola exhibición, más sin embargo, no volvió a comprar nunca o bien podemos darnos cuenta de que es un cliente frecuente que hace compras seguidas.

Ahora lo que deseamos es contar el número de factura que se tienen registradas a nombre de un mismo cliente, la función de agregado por utiliza es `COUNT`.

```
SELECT TxtNombreCliente, SUM(intCantidad) as Vinos_Adquiridos, SUM(CurMonto)
AS Inversion, COUNT(TxtClaveVentas) as Num_Compras
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente
ORDER BY Inversion DESC;
```

| TxtNombreCliente                     | Vinos_Adquiridos | Inversion | Num_Compras |
|--------------------------------------|------------------|-----------|-------------|
| Comercializadora Emporio de Vinicola | 33               | 19125.00  | 5           |
| El rincón de la Luna Restaurant      | 26               | 9502.00   | 7           |
| Vinos y Licores del Oeste            | 17               | 4227.00   | 5           |
| San Carlos Restaurant                | 7                | 4026.00   | 2           |
| Productos vinicolas Esfinge          | 4                | 2087.00   | 2           |

Es curioso notar que quien más compras ha hecho a la compañía es el Restaurante El rincón de la luna con siete compras, sin embargo, ha adquirido menos vinos que la Comercializadora Emporio Vinícola en cinco compras.

La diferencia entre las funciones de agregado `SUM` y `COUNT` es que `SUM` se utiliza para sumar los valores contenidos en la columna indicada, y `COUNT` cuenta cada una de los registros a partir de la columna indicada. Por ejemplo:

Suponga que existe una columna llamada `datos`, la cual contiene los siguientes valores:

Datos={100,400, 400}

El resultado al hacer un SUM(Datos) sería 900, ya que se suman los valores 100, 400, 400; mientras que el resultado de hacer un COUNT(Datos) sería tres, ya que son tres los valores existentes para esa columna.

La función de agregado AVERAGE obtiene un promedio a partir de todos los valores de la columna especificada. Si deseáramos conocer cuál fue en promedio la cantidad invertida de un cliente en cada compra, entonces utilizamos AVERAGE o AVG en algunos RDBMS de la siguiente manera.

```
SELECT TxtNombreCliente, SUM(CurMonto) AS Inversion, COUNT(TxtClaveVentas)
as Num_Compras , AVG(CurMonto) as PromedioInversion
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente
ORDER BY Inversion DESC;
```

| TxtNombreCliente                     | Inversion | Num_Compras | PromedioInversion |
|--------------------------------------|-----------|-------------|-------------------|
| Comercializadora Emporio de Vinicola | 19125.00  | 5           | 3825.000000       |
| El rincón de la Luna Restaurant      | 9502.00   | 7           | 1357.428571       |
| Vinos y Licores del Oeste            | 4227.00   | 5           | 845.400000        |
| San Carlos Restaurant                | 4026.00   | 2           | 2013.000000       |
| Productos vinicolas Esfinge          | 2087.00   | 2           | 1043.500000       |

Las funciones de agregado MIN y MAX se enfocan a obtener el valor mínimo y máximo de un conjunto de datos, por ejemplo, suponiendo nuevamente una columna de datos con la siguiente información:

Datos={5, 900, 75}

Utilizar una función de agregado MIN(Datos) arrojaría como resultado 5 mientras que MAX(Datos) obtendría como resultado 900.

Por ejemplo, se quisiéramos saber cuál fue la cantidad mínima y máxima invertida por un cliente agregaríamos las siguientes funciones de agregado.

```
SELECT TxtNombreCliente, SUM(CurMonto) AS Inversion, Min(CurMonto),
Max(CurMonto)
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente
ORDER BY Inversion DESC;
```

| TxtNombreCliente                     | Inversion | Min(CurMonto) | Max(CurMonto) |
|--------------------------------------|-----------|---------------|---------------|
| Comercializadora Emporio de Vinicola | 19125.00  | 474.00        | 8722.00       |
| El rincón de la Luna Restaurant      | 9502.00   | 483.00        | 3992.00       |
| Vinos y Licores del Oeste            | 4227.00   | 158.00        | 1996.00       |
| San Carlos Restaurant                | 4026.00   | 288.00        | 3738.00       |
| Productos vinicolas Esfinge          | 2087.00   | 590.00        | 1497.00       |

Con esta consulta, es como ahora sabemos que el cliente Comercializadora Emporio tuvo una inversión mínima de \$474.00 y una inversión máxima de \$8,722 pesos.

Al estar realizando un agrupamiento de información con el objeto de presentar bloques de información más sólidos, es posible que se requiera agrupar información no sólo de una columna sino de más. Por ejemplo, el agrupamiento de clientes tiene la peculiaridad

de que no se conoce la fecha de la venta, ya que se está trabajando con toda la información histórica.

Si deseáramos conocer la información agrupada por cliente pero en dónde a demás se agrupara esta información por año, es decir, mostrará año con año, cuál ha sido la inversión de un cliente; entonces se debe agregar un nuevo criterio de agrupamiento de información, lo cual se hace agregando la columna `dateFechaVenta` al `GROUP BY`.

```
SELECT TxtNombreCliente, SUM(CurMonto) AS Inversion,
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente, dateFechaVenta
ORDER BY Inversion DESC;
```

Realizar lo anterior, no es suficiente ya que como recordará, hay una regla que indica que todas las columnas proyectadas deben aparecer en el `GROUP BY` y viceversa, en este caso, falta agregar la columna `dateFechaVenta` en la proyección de información, tal como se muestra a continuación:

```
SELECT dateFechaVenta, TxtNombreCliente, SUM(CurMonto) AS Inversion
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente, dateFechaVenta
ORDER BY Inversion DESC;
```

| dateFechaVenta      | TxtNombreCliente                     | Inversion |
|---------------------|--------------------------------------|-----------|
| 2007-01-21 00:00:00 | Comercializadora Emporio de Vinicola | 8722.00   |
| 2007-02-15 00:00:00 | Comercializadora Emporio de Vinicola | 5310.00   |
| 2007-04-18 00:00:00 | El rincón de la Luna Restaurant      | 3992.00   |
| 2007-03-21 00:00:00 | San Carlos Restaurant                | 3738.00   |
| 2007-06-09 00:00:00 | Comercializadora Emporio de Vinicola | 2994.00   |
| 2007-05-30 00:00:00 | El rincón de la Luna Restaurant      | 2298.00   |
| 2007-03-05 00:00:00 | Vinos y Licores del Oeste            | 2154.00   |
| 2007-03-05 00:00:00 | Comercializadora Emporio de Vinicola | 1625.00   |
| 2006-12-31 00:00:00 | Productos vinicolas Esfinge          | 1497.00   |
| 2007-04-16 00:00:00 | El rincón de la Luna Restaurant      | 1300.00   |
| 2006-12-24 00:00:00 | Vinos y Licores del Oeste            | 998.00    |
| 2007-01-06 00:00:00 | El rincón de la Luna Restaurant      | 930.00    |
| 2007-04-24 00:00:00 | Productos vinicolas Esfinge          | 590.00    |
| 2007-01-21 00:00:00 | Vinos y Licores del Oeste            | 576.00    |
| 2006-12-31 00:00:00 | Vinos y Licores del Oeste            | 499.00    |
| 2006-12-24 00:00:00 | El rincón de la Luna Restaurant      | 499.00    |
| 2007-02-18 00:00:00 | El rincón de la Luna Restaurant      | 483.00    |
| 2007-03-16 00:00:00 | Comercializadora Emporio de Vinicola | 474.00    |
| 2006-12-31 00:00:00 | San Carlos Restaurant                | 288.00    |

Realizar la consulta de este modo, no cumple con nuestro objetivo ya que los datos se agrupan por día y nosotros queremos agrupar por año y por cliente, para conseguirlo, nos valdremos de una función para datos tipo Fecha, la función `YEAR` va a obtener la parte del año de un valor fecha. Por ejemplo:

```
SELECT YEAR(NOW());
+-----+
| YEAR(NOW()) |
+-----+
|          2007 |
+-----+
```

En esta consulta se obtiene el año `YEAR ( )` de la fecha actual `NOW( )`.

Ahora que conocemos esto, modificaremos la consulta para agrupar la información por año y por cliente, tal como se muestra a continuación:

```
SELECT YEAR(dateFechaVenta) as Anio, TxtNombreCliente, SUM(CurMonto) AS
Inversion
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente, Anio
ORDER BY Anio ASC, Inversion DESC;
```

| Anio | TxtNombreCliente                     | Inversion |
|------|--------------------------------------|-----------|
| 2006 | Productos vinicolas Esfinge          | 1497.00   |
| 2006 | Vinos y Licores del Oeste            | 1497.00   |
| 2006 | El rincón de la Luna Restaurant      | 499.00    |
| 2006 | San Carlos Restaurant                | 288.00    |
| 2007 | Comercializadora Emporio de Vinicola | 19125.00  |
| 2007 | El rincón de la Luna Restaurant      | 9003.00   |
| 2007 | San Carlos Restaurant                | 3738.00   |
| 2007 | Vinos y Licores del Oeste            | 2730.00   |
| 2007 | Productos vinicolas Esfinge          | 590.00    |

Ahora con la información agrupada por fecha y por cliente podemos ver que en el año 2006 quienes más invirtieron en la compañía fueron la compañía Productos Vinícolas Esfinge y Vinos y Licores del Oeste; sin embargo para el año 2007, quien más ha invertido en la compañía es la Comercializadora Emporio, un cliente con menor antigüedad, cabe mencionar.

Agrupar la información, va a permitir mostrar la información en bloques más sólidos, dejando de lado el detalle para concentrarse en la totalización de información, todo ello, con el objeto principal de servir de soporte en el proceso para la toma de decisiones.

# CAPÍTULO 7

## Vistas

---

Ahora que ya sabemos realizar consultas de información, ya sea de una o más tablas, es como podemos introducir el concepto de vista.

Una vista por definición es un *query* encapsulado en un nombre de objeto; con ello nos referimos a que es posible almacenar una consulta en la base de datos, asignándole un nombre en particular, al guardar la consulta en la base de datos, ésta es percibida por los usuarios como un objeto de la base de datos, la cual tendrá algunas de las características que poseen los demás objetos.

Las vistas son de gran utilidad cuando se generan un mismo tipo de reportes periódicamente, por ejemplo, un reporte mensual que muestre la inversión de los clientes en la compañía. Esta tarea supone la ejecución de una consulta cada mes que se requiera el listado, lo cual resulta una tarea repetitiva a lo largo del tiempo; con el objeto de realizar una vez el trabajo, se crea la consulta de manera general y se guarda con un nombre, para que esté disponible a través del tiempo.

Así pues, la consulta que creamos es la siguiente:

```
SELECT YEAR(dateFechaVenta) as Anio, MONTH(dateFechaVenta) as Mes,  
TxtNombreCliente, SUM(CurMonto) AS Inversion  
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente  
GROUP BY TxtNombreCliente, Anio, Mes  
ORDER BY Anio ASC, Mes ASC, Inversion DESC;
```

| Anio | Mes | TxtNombreCliente                     | Inversion |
|------|-----|--------------------------------------|-----------|
| 2006 | 12  | Productos vinicolas Esfinge          | 1497.00   |
| 2006 | 12  | Vinos y Licores del Oeste            | 1497.00   |
| 2006 | 12  | El rincón de la Luna Restaurant      | 499.00    |
| 2006 | 12  | San Carlos Restaurant                | 288.00    |
| 2007 | 1   | Comercializadora Emporio de Vinicola | 8722.00   |
| 2007 | 1   | El rincón de la Luna Restaurant      | 930.00    |
| 2007 | 1   | Vinos y Licores del Oeste            | 576.00    |
| 2007 | 2   | Comercializadora Emporio de Vinicola | 5310.00   |
| 2007 | 2   | El rincón de la Luna Restaurant      | 483.00    |
| 2007 | 3   | San Carlos Restaurant                | 3738.00   |
| 2007 | 3   | Vinos y Licores del Oeste            | 2154.00   |
| 2007 | 3   | Comercializadora Emporio de Vinicola | 2099.00   |
| 2007 | 4   | El rincón de la Luna Restaurant      | 5292.00   |
| 2007 | 4   | Productos vinicolas Esfinge          | 590.00    |
| 2007 | 5   | El rincón de la Luna Restaurant      | 2298.00   |
| 2007 | 6   | Comercializadora Emporio de Vinicola | 2994.00   |

La consulta muestra la inversión de los clientes, en donde se agrupan los datos por año, por mes y por cliente, así es como obtenemos una consulta de propósito general, la cual nos servirá a lo largo del tiempo; ahora bien, dicha consulta debemos encapsularla en un nombre de objeto, es decir, generamos una vista con la consulta anterior.

Para generar una vista, ejecutamos la instrucción SQL `CREATE VIEW`, la cual tiene la siguiente sintaxis:

```
CREATE VIEW nombreVista AS
(
[Sentencia SQL SELECT]
);
```

Para generar la vista que nos ayude a obtener los reportes mensuales, entonces ejecutamos la siguiente instrucción:

```
CREATE VIEW inversionMensual AS
(
SELECT YEAR(dateFechaVenta) as Anio, MONTH(dateFechaVenta) as Mes,
TxtNombreCliente, SUM(CurMonto) AS Inversion
FROM ventas INNER JOIN cliente on cliente.idCliente=ventas.idCliente
GROUP BY TxtNombreCliente, Anio, Mes
ORDER BY Anio ASC, Mes ASC, Inversion DESC
);
```

Al decir que la consulta se encapsula en un nombre de objeto, es porque dentro del RDBMS ésta puede utilizarse al igual que una tabla sólo en la consulta de datos; así mismo, se pueden asignar privilegios a los usuarios para que utilicen de forma especial la vista que hemos generado.

Ahora que hemos encapsulado nuestra consulta en el nombre de objeto `inversionMensual`, podemos hacer una consulta general sobre este objeto:

```
SELECT * FROM inversionMensual;
```

| Anio | Mes | TxtNombreCliente                     | Inversion |
|------|-----|--------------------------------------|-----------|
| 2006 | 12  | Productos vinicolas Esfinge          | 1497.00   |
| 2006 | 12  | Vinos y Licores del Oeste            | 1497.00   |
| 2006 | 12  | El rincón de la Luna Restaurant      | 499.00    |
| 2006 | 12  | San Carlos Restaurant                | 288.00    |
| 2007 | 1   | Comercializadora Emporio de Vinicola | 8722.00   |
| 2007 | 1   | El rincón de la Luna Restaurant      | 930.00    |
| 2007 | 1   | Vinos y Licores del Oeste            | 576.00    |
| 2007 | 2   | Comercializadora Emporio de Vinicola | 5310.00   |
| 2007 | 2   | El rincón de la Luna Restaurant      | 483.00    |
| 2007 | 3   | San Carlos Restaurant                | 3738.00   |
| 2007 | 3   | Vinos y Licores del Oeste            | 2154.00   |
| 2007 | 3   | Comercializadora Emporio de Vinicola | 2099.00   |
| 2007 | 4   | El rincón de la Luna Restaurant      | 5292.00   |
| 2007 | 4   | Productos vinicolas Esfinge          | 590.00    |
| 2007 | 5   | El rincón de la Luna Restaurant      | 2298.00   |
| 2007 | 6   | Comercializadora Emporio de Vinicola | 2994.00   |

Note que cuando ejecutamos la consulta sobre la vista `inversionMensual`, es como si hubiésemos ejecutado la consulta genérica antes desarrollada, el resultado que se obtiene es el mismo.

Ahora queda más claro cuando se dice que una vista es un *query* encapsulado en un nombre de objeto.

Ya que se creó la consulta genérica, podemos restringir la información para obtener los reportes mensuales que deseábamos, eso se logra de la siguiente manera:

Reporte de enero de 2007.

```
SELECT * FROM inversionMensual WHERE Anio='2007' AND Mes='01';
```

| Anio | Mes | TxtNombreCliente                     | Inversion |
|------|-----|--------------------------------------|-----------|
| 2007 | 1   | Comercializadora Emporio de Vinicola | 8722.00   |
| 2007 | 1   | El rincón de la Luna Restaurant      | 930.00    |
| 2007 | 1   | Vinos y Licores del Oeste            | 576.00    |

Reporte de febrero de 2007.

```
SELECT * FROM inversionMensual WHERE Anio='2007' AND Mes='02';
```

| Anio | Mes | TxtNombreCliente                     | Inversion |
|------|-----|--------------------------------------|-----------|
| 2007 | 2   | Comercializadora Emporio de Vinicola | 5310.00   |
| 2007 | 2   | El rincón de la Luna Restaurant      | 483.00    |

Reporte de marzo de 2007.

```
SELECT * FROM inversionMensual WHERE Anio='2007' AND Mes='03';
```

| Anio | Mes | TxtNombreCliente                     | Inversion |
|------|-----|--------------------------------------|-----------|
| 2007 | 3   | San Carlos Restaurant                | 3738.00   |
| 2007 | 3   | Vinos y Licores del Oeste            | 2154.00   |
| 2007 | 3   | Comercializadora Emporio de Vinicola | 2099.00   |

Reporte de abril de 2007.

```
SELECT * FROM inversionMensual WHERE Anio='2007' AND Mes='04';
```

```
+-----+-----+-----+-----+
| Anio | Mes | TxtNombreCliente | Inversion |
+-----+-----+-----+-----+
| 2007 | 4 | El rincón de la Luna Restaurant | 5292.00 |
| 2007 | 4 | Productos vinícolas Esfinge | 590.00 |
+-----+-----+-----+-----+
```

Como podemos notar, generar vistas es siempre útil cuando se desean realizar reportes con cierta periodicidad; de esta manera, una consulta genérica es almacenada con un nombre de objeto en la base de datos y posteriormente, se restringe la información presentada por la vista para hacer la consulta específica a una situación.

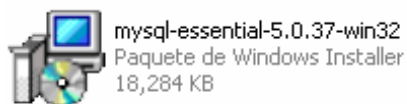
# ANEXOS

## Anexo A. Instalación de MySQL en Windows XP

---

¿Cómo Instalar MySQL 5.0 en un sistema operativo Windows?

1. Dar doble clic sobre el archivo ejecutable



2. Aparecerá la pantalla de bienvenida, damos clic en el botón Siguiente.



Fig. 1. Instalación de MySQL para Windows, Pantalla de Bienvenida.

3. A continuación indicaremos el tipo de instalación que queremos hacer, seleccionamos la opción Completa y damos clic en el botón Siguiente.

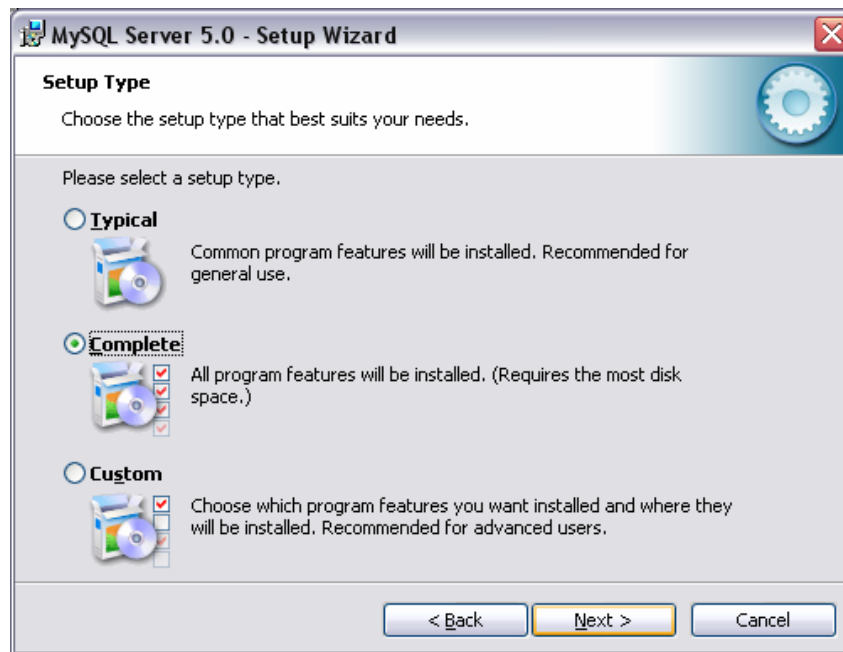


Fig. 2. Tipo de instalación.

4. Posteriormente se muestra los valores de la instalación, damos clic en la opción Instalar.



Fig. 3. Valores de instalación.

5. Esperamos a que la instalación termine.



Fig. 4. Progreso de la instalación.

6. Aparecerá una nueva ventana durante la instalación, la cual solicitará nos registremos en MySQL.com, así como crear una cuenta o simplemente pasamos por alto ese paso, por último, damos clic en el botón Siguiente.



Fig. 5. Registro en MySQL.com

- Al término de la instalación, seleccionaremos la casilla Configurar el servidor MySQL y damos clic en el botón Finalizar.



Fig. 6. Instalación completa.

- Para la configuración del servidor, aparecerá la siguiente pantalla de bienvenida, damos clic en la opción Siguiente.

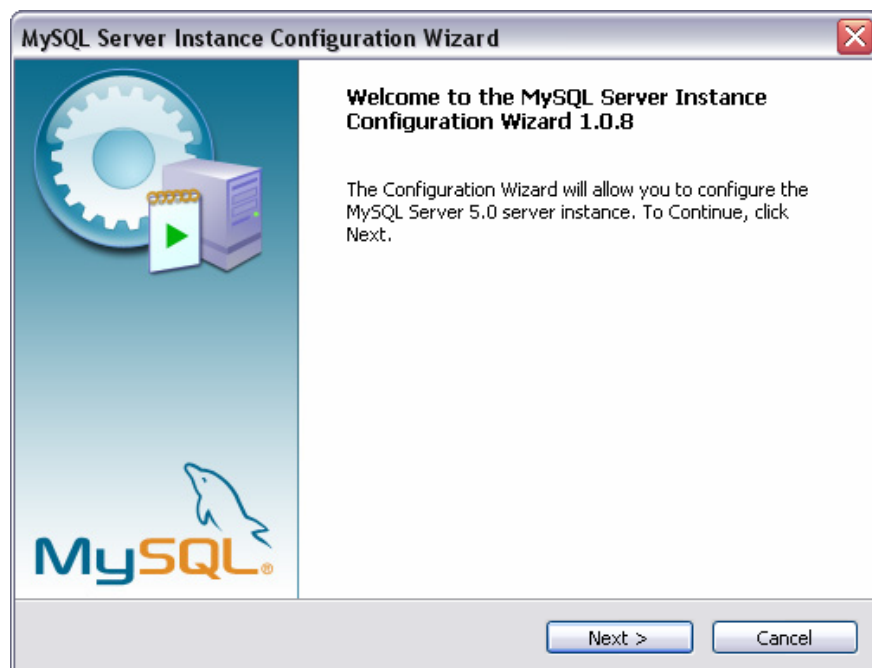


Fig. 7. Pantalla de bienvenida para la configuración del SGBD MySQL.

9. La siguiente pantalla nos da la opción de elegir entre una configuración detallada o una configuración estándar, seleccionamos la opción Configuración detallada y damos clic en el botón Siguiente.

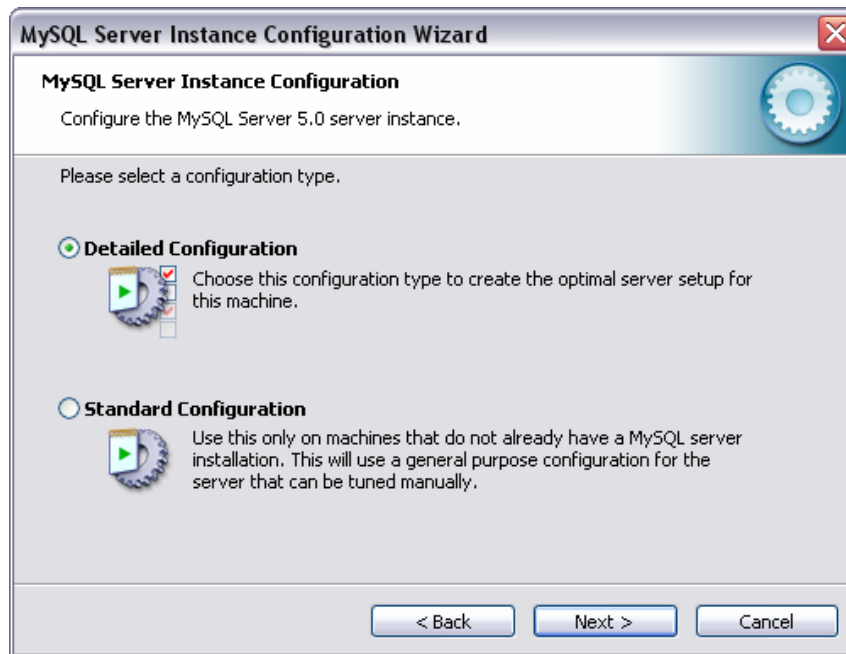


Fig. 8. Selección del tipo de configuración.

10. A continuación elegimos la instancia bajo la cual mantendremos nuestro servidor MySQL, elegiremos la opción Equipo de un desarrollador, y damos clic en la opción Siguiente.



Fig. 9. Selección del tipo de equipo donde se aloja el SMBD MySQL.

11. Ahora indicaremos el uso que daremos a nuestra base de datos, seleccionaremos la opción Base de datos multifuncionales, y damos clic en el botón Siguiente.

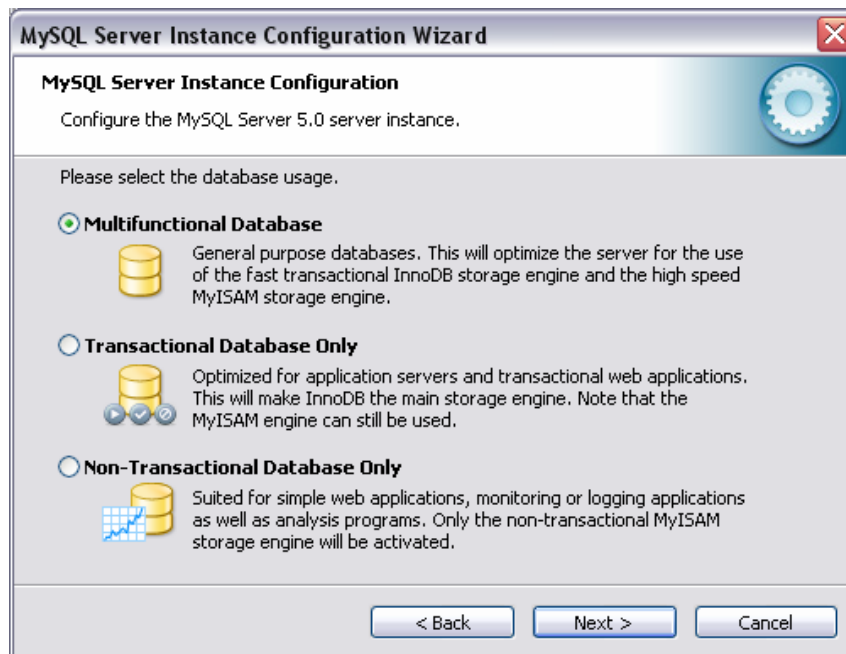


Fig. 10. Selección del uso que daremos a la base de datos.

12. Continuando con la instalación, seleccionaremos el lugar donde se almacenarán los archivos para el manejo de las tablas del sistema, y damos clic en el botón Siguiente.

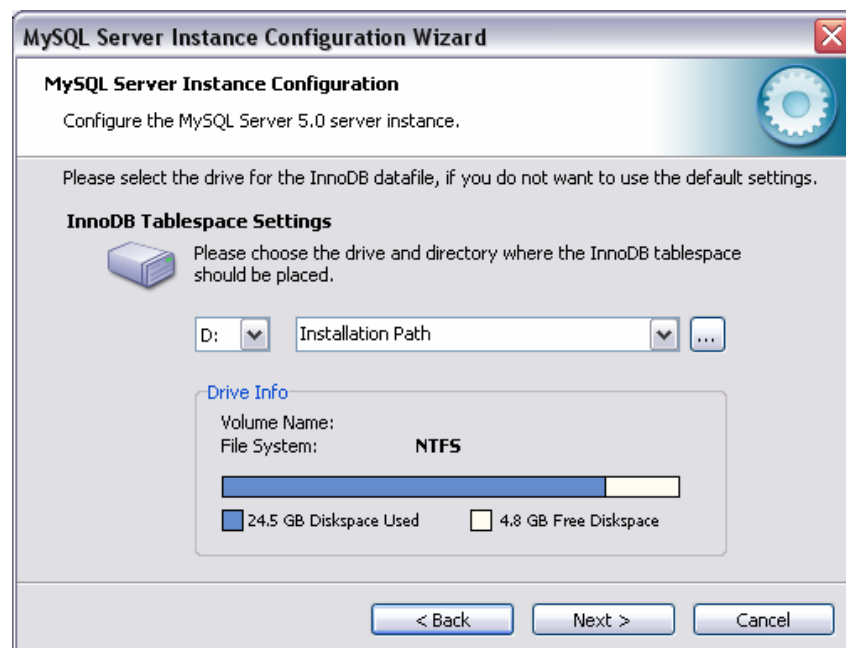


Fig. 11. Configuración del directorio que almacenará los archivos InnoDB.

13. Indicamos el número de conexiones concurrentes que esperamos para nuestra base de datos.

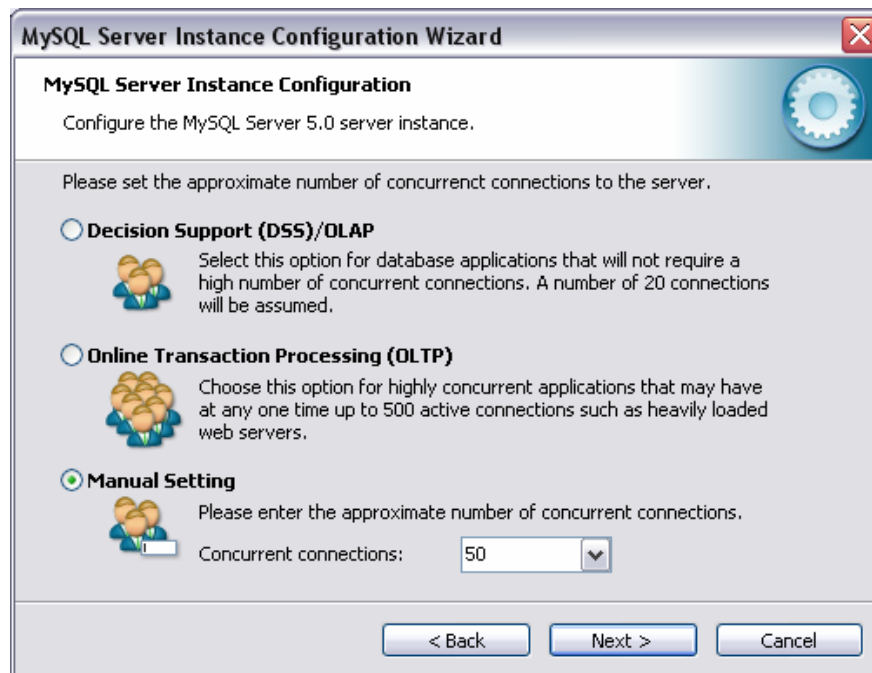


Fig. 12. Conexiones concurrentes esperadas.

14. Elegimos si estarán abiertas las conexiones TCP/IP.

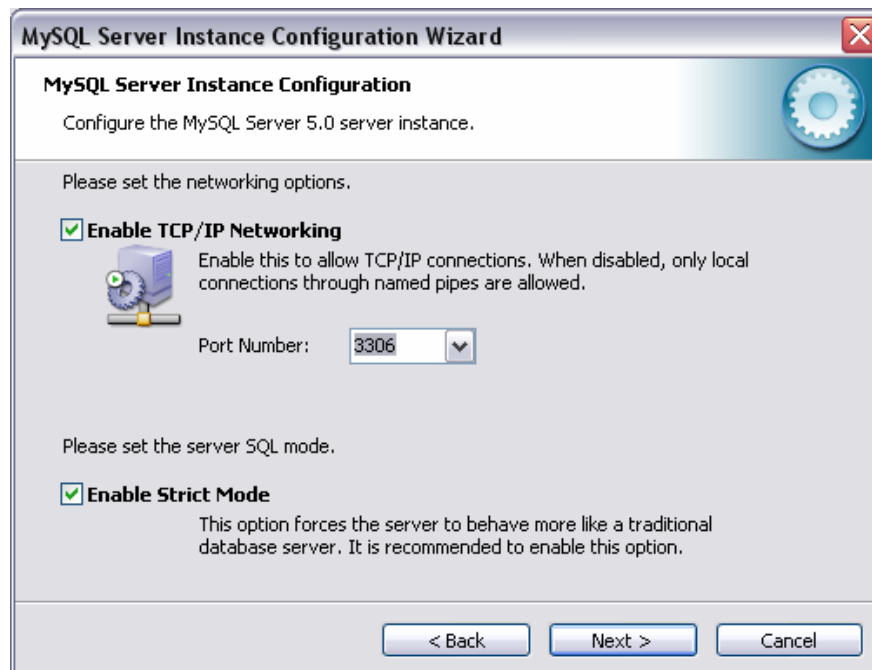


Fig. 13. Opciones de red.

15. Ahora debemos seleccionar la configuración de idioma que manejará MySQL al almacenar los datos, Elegimos la opción Latin1.

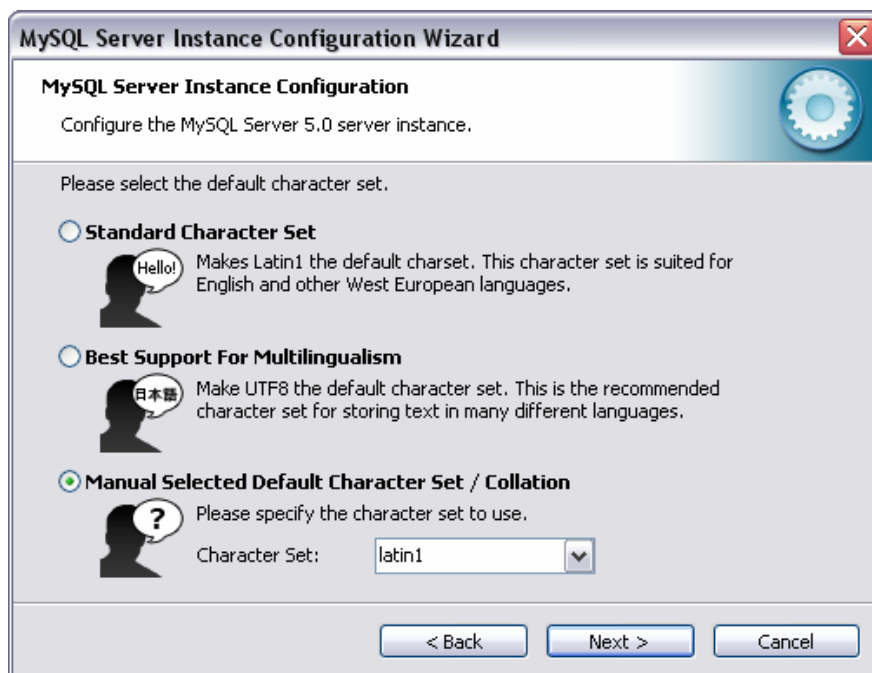


Fig. 14. Selección del "char set".

16. Indicamos las características del "servicio" que se levantará en Windows.

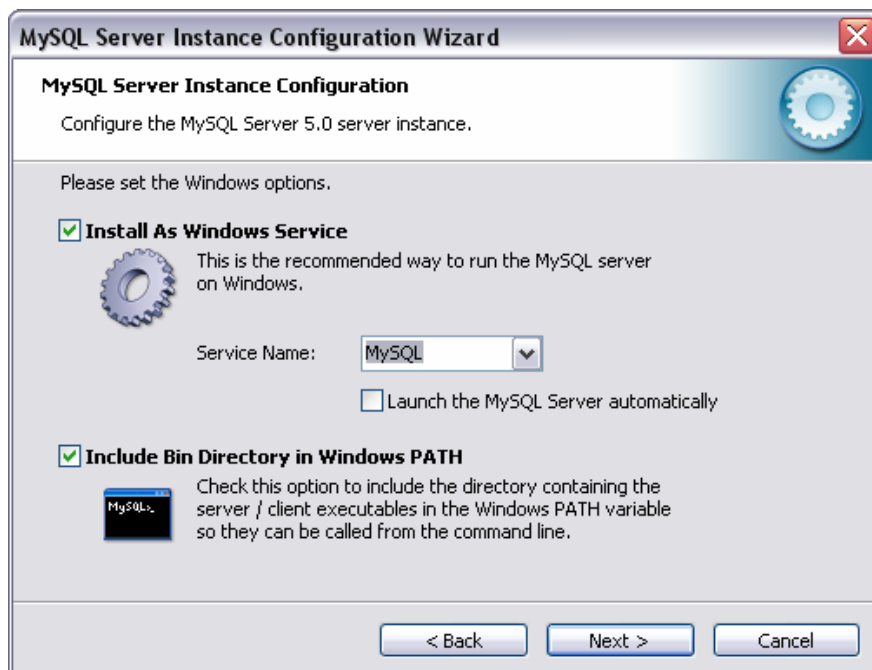


Fig. 15. Opciones de servicio para Windows.

17. Indicamos la contraseña para la opción root.



Fig. 16. Opciones de seguridad.

18. Ahora simplemente damos clic sobre el botón Ejecutar para que se configure nuestro SMBD.

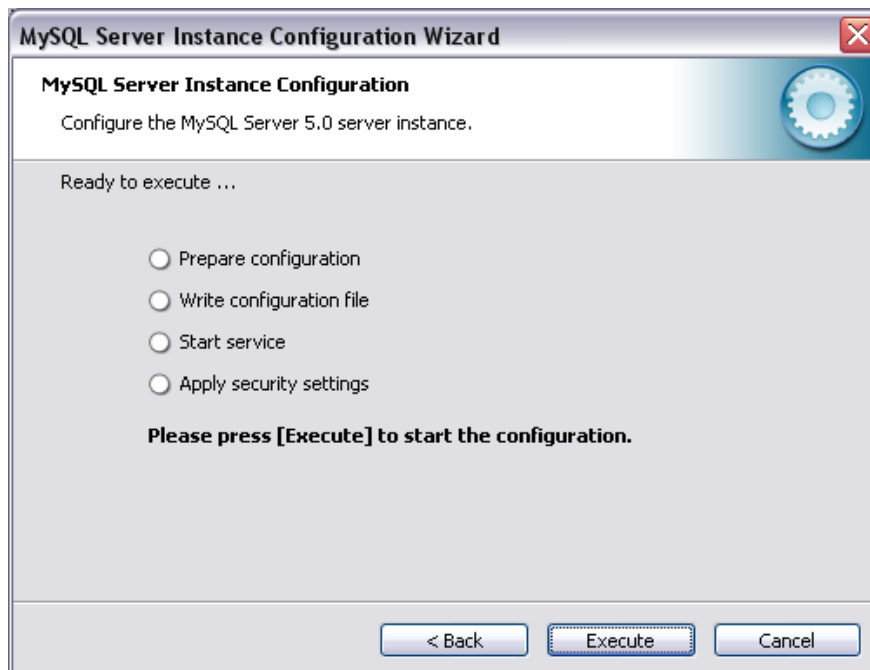


Fig. 17. Listo para la ejecución de la configuración.

19. Esperamos a que concluya la configuración.

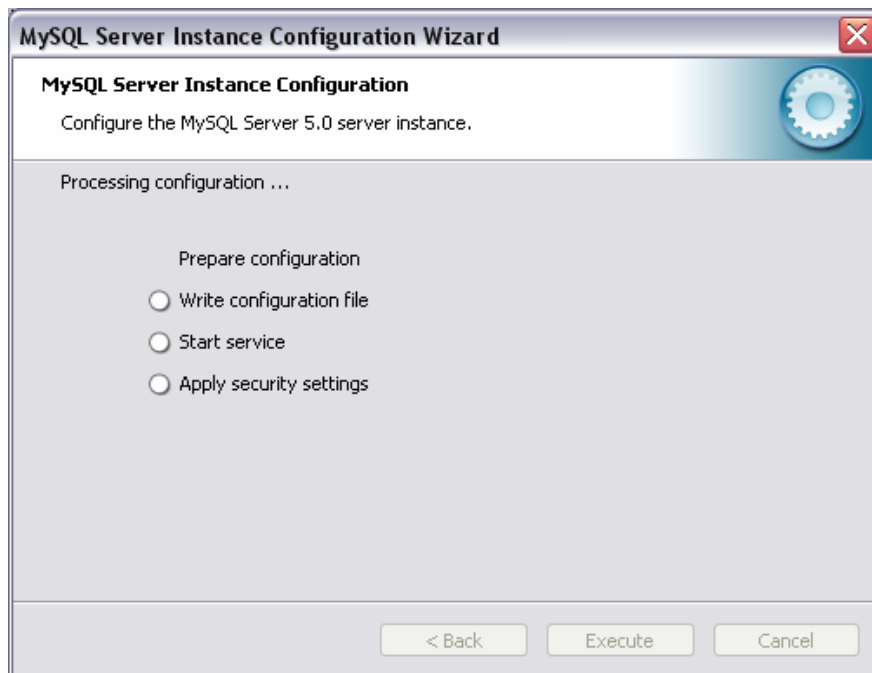


Fig 18. Configurándose el sistema.

20. Damos clic en la opción Finalizar una vez se ha indicado una configuración satisfactoria.

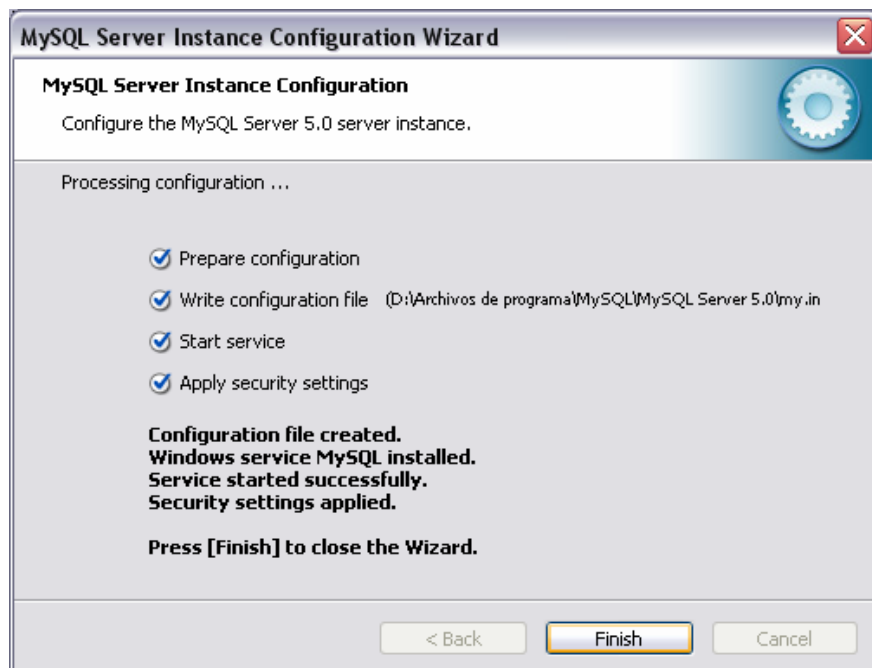


Fig. 19. Fin de la configuración.

## ***Cliente de MySQL***

Una vez que se ha instalado el cliente de MySQL, y tal como se muestra en la figura anterior, el servicio del manejador de base de datos MySQL se inicia; para el acceso a la aplicación cliente deberemos abrir una consola de Windows o símbolo del sistema.

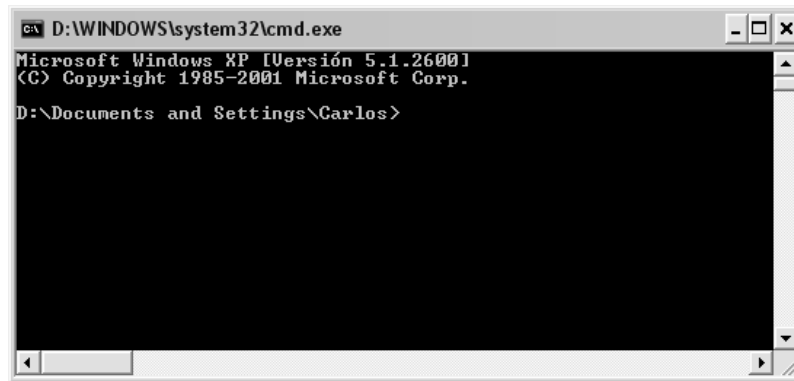


Fig. 20. Símbolo del sistema de Windows.

La siguiente instrucción permitirá ingresar a la aplicación cliente del RDBMS para trabajar con una base de datos, ejecutaremos el comando `mysql`, el cual cuenta con una gama interesante de propiedades, por el momento sólo utilizaremos dos de ellas: la opción `-u` para indicar un nombre de usuario y la opción `-p` para indicar una contraseña en caso de existir.

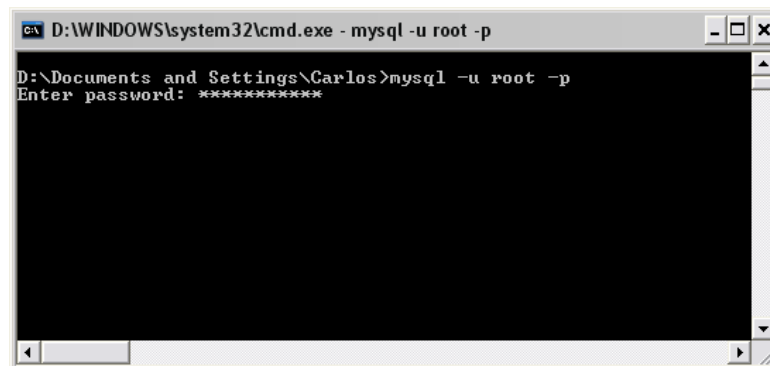


Fig. 21. Acceso al RDBMS.

Una vez que ingresamos, se muestra la siguiente información:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

Después de que se muestra el prompt (`mysql>`) de MySQL, es cuando podemos interactuar con el RDBMS a través del lenguaje SQL.

## Anexo B. Tipos de tablas de MySQL

MySQL soporta varios motores de almacenamiento que tratan con distintos tipos de tabla. Los motores de almacenamiento de MySQL incluyen algunos que tratan con tablas transaccionales y otros que no lo hacen:

- **MyISAM** trata tablas no transaccionales. Proporciona almacenamiento y recuperación de datos rápida, así como posibilidad de búsquedas fulltext. **MyISAM** se soporta en todas las configuraciones MySQL, y es el motor de almacenamiento por defecto a no ser que tenga una configuración distinta a la que viene por defecto con MySQL.
- El motor de almacenamiento **MEMORY** proporciona tablas en memoria. El motor de almacenamiento **MERGE** permite una colección de tablas **MyISAM** idénticas que deben ser tratadas como una simple tabla. Como **MyISAM**, los motores de almacenamiento **MEMORY** y **MERGE** tratan tablas no transaccionales y ambos se incluyen en MySQL por defecto.

**Nota:** el motor de almacenamiento **MEMORY** anteriormente se conocía como **HEAP**.

- Los motores de almacenamiento **InnoDB** y **BDB** proporcionan tablas transaccionales. **BDB** se incluye en la distribución binaria MySQL-Max en aquellos sistemas operativos que la soportan. **InnoDB** también se incluye por defecto en todas las distribuciones binarias de MySQL 5.0. En distribuciones fuente, puede activar o desactivar estos motores de almacenamiento configurando MySQL a su gusto.
- El motor de almacenamiento **EXAMPLE** es un motor de almacenamiento "tonto" que no hace nada. Puede crear tablas con este motor, pero no puede almacenar datos ni recuperarlos. El objetivo es que sirva como ejemplo en el código MySQL para ilustrar cómo escribir un motor de almacenamiento. Como tal, su interés primario es para desarrolladores.
- **NDB Cluster** es el motor de almacenamiento usado por MySQL Cluster para implementar tablas que se particionan en varias máquinas. Está disponible en distribuciones binarias MySQL-Max 5.0. Este motor de almacenamiento se encuentra disponible para Linux, Solaris, y Mac OS X.
- El motor de almacenamiento **ARCHIVE** se usa para guardar grandes cantidades de datos sin índices con una huella muy pequeña.
- El motor de almacenamiento **CSV** guarda datos en ficheros de texto usando formato de valores separados por comas.
- El motor de almacenamiento **FEDERATED** se añadió en MySQL 5.0.3. Este motor guarda datos en una base de datos remota. En esta versión sólo funciona con MySQL a través de la API MySQL C Client. En futuras versiones, será capaz de conectar con otras fuentes de datos usando otros drivers o métodos de conexión clientes.

Cuando crea una nueva tabla, puede decirle a MySQL qué tipo de tabla crear añadiendo la opción de tabla **ENGINE** o **TYPE** al comando **CREATE TABLE**:

```
CREATE TABLE t (i INT) ENGINE = INNODB;
CREATE TABLE t (i INT) TYPE = MEMORY;
```

Aunque se soporta `TYPE` en MySQL 5.0, `ENGINE` es el término preferido.

Si omite la opción `ENGINE` o `TYPE`, se usa el motor de almacenamiento por defecto, que es `MyISAM` en Linux. Puede cambiarlo usando las opciones de arranque `--default-storage-engine` o `--default-table-type`, o cambiando la variable de sistema `storage_engine` o `table_type`.

```
SHOW ENGINES;  
SET storage_engine=INNODB;  
SET storage_engine=MYISAM;
```

Cuando se instala MySQL en Windows usando el MySQL Configuration Wizard, `InnoDB` es el motor de almacenamiento por defecto en lugar de `MyISAM`.

Para convertir una tabla de un tipo a otro, use un comando `ALTER TABLE` que indique el nuevo tipo:

```
ALTER TABLE t ENGINE = MYISAM;  
ALTER TABLE t TYPE = BDB;
```

## Bibliografía

---

- Van Der Lans, Rick F.  
**Introduction to SQL: Mastering the Relational Data Base Language.**  
Estados Unidos de Norte América: Addison Wesley Professional. 2006.
- Taylor, Allen G.  
**SQL for Dummies.**  
Indianapolis, EUA. Wiley Publishing, Inc. 2006.
- Donahoo, Michel J. y Speegle, Gregory D.  
**SQL: Practical Guide for Developers.**  
San Francisco, UEA. Ed. Morgan Kauffman Ed., Elseiver Inc. 2005.
- MySQL AB.  
**MySQL Reference Manual. 2007. [en línea]**  
<<http://dev.mysql.com/doc/refman/5.0/en/index.html>>
- The PostgreSQL Global Development Group,  
**PostgreSQL 8.0 Documentation. 2005. [en línea]**  
<<http://www.postgresql.org/docs/8.0/static/index.html>>
- Microsoft Corporation.  
**Microsoft SQL Server 2000 References Guide. 2000. [Documento digital]**  
<Documento de ayuda incluido en la instalación del software>

# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

**DR. JUAN RAMÓN DE LA FUENTE**  
Rector

**LIC. ENRIQUE DEL VAL BLANCO**  
Secretario General

**MTRO. DANIEL BARRERA PÉREZ**  
Secretario Administrativo

**DRA. ROSAURA RUIZ GUTIÉRREZ**  
Secretaria de Desarrollo Institucional

**MTRO. JOSÉ ANTONIO VELA CAPDEVILA**  
Secretario de Servicios a la Comunidad

**MTRO. JORGE ISLAS LÓPEZ**  
Abogado General

## DIRECCIÓN GENERAL DE SERVICIOS DE CÓMPUTO ACADÉMICO

**DR. ALEJANDRO PISANTY BARUCH**  
Director General

**MAT. CARMEN BRAVO CHAVESTE**  
Directora de Cómputo para la Docencia

**ACT. FABIÁN ROMO ZAMUDIO**  
Subdirector de Tecnología para la Educación

**MTRO. JESÚS DÍAZ BARRIGA ARCEO**  
Subdirector de Planeación Académica

**MTRA. HORTENSIA CANO GRANADOS**  
Coordinadora de Producción Académica

**MTRA. ALEJANDRINA SAN JUAN REYES**  
Coordinadora del Centro de Extensión en  
Cómputo y Telecomunicaciones Nuevo León

**ING. SERGIO ALVA ARGUINZONIZ**  
Coordinador del Centro Mascarones

**LIC. JOSÉ LUIS MEDINA FLORES**  
Coordinador del Centro Coapa de Extensión  
en Cómputo y Telecomunicaciones

**LIC. PATRICIA ROMERO ELÍAS**  
Coordinadora del Centro Educativo  
Multidisciplinario Polanco

**MTRA. ALMA IBARRA OBANDO**  
Coordinadora del Centro San Agustín

**LIC. REBECA SÁNCHEZ ZALDIVAR**  
Responsable de los cursos de Cómputo  
del Palacio de la Autonomía, Fundación UNAM

**ING. PABLO DE LA O CRUZ**  
Coordinador del Centro Tlatelolco

**LIC. JOSÉ ANTONIO SÁNCHEZ YLLANEZ**  
Subdirector de Comunicación

**DG TOÑA ZIMMERMAN SONABEND**  
Jefa del Departamento de Diseño

*Guías y Textos de Cómputo*  
*SQL ESTÁNDAR Básico*  
*Octubre, 2007*