SQL
EN LA PRÁCTICA

G. QUINTANA, M. MARQUÉS, J. I. ALIAGA Y M. J. ARAMBURU

Índice

PRÓLOGO

El objetivo de este texto es permitir al lector introducirse y profundizar de forma práctica y aplicada en el lenguaje SQL (*Structured Query Language*), un lenguaje utilizado en la mayor parte de los sistemas de gestión de bases de datos actuales, tanto en los sistemas destinados a las pequeñas empresas como los dedicados a las grandes corporaciones. De hecho, en pocas áreas de la informática un lenguaje predomina de forma tan clara y rotunda como el SQL en el campo de las bases de datos.

Este texto no está pensado para introducir conceptos teóricos de bases de datos. No obstante, se presentan los conceptos básicos necesarios sobre teoría de bases de datos para profundizar en este lenguaje.

El objetivo es introducir gradualmente al lector en el uso de este lenguaje mostrando sus diversos conceptos e ilustrándolos siempre con ejemplos aplicados, completos y sencillos. Todos los ejemplos giran en torno al sistema de facturación y control de *stocks* de una empresa, con lo cual se pretende atraer una mayor atención del lector a la par que se muestra un ejemplo con una utilidad práctica real en la empresa. La descripción de SQL será gradual, desde sentencias de dos líneas hasta las más avanzadas.

La presentación del lenguaje SQL tiene en este texto un enfoque muy práctico y aplicado. Se han incluido numerosos ejemplos y ejercicios al lado de cada concepto nuevo introducido para que de esta forma el lector pueda practicar y ver en la práctica cada concepto que se describe. De hecho, la mayor parte de los capítulos incluyen más de 20 ejercicios. Los ejercicios se presentan siempre con solución para que el lector pueda contrastar ésta con la suya o estudiarla sin más. Las soluciones se dan inmediatamente tras la propuesta para evitar al lector la labor de ir al final del texto tras cada ejercicio a buscar la solución en un apéndice final. Únicamente se han llevado a un apartado final las soluciones de los ejercicios de autoevaluación dado su carácter más evaluador y menos didáctico. Estos ejercicios, que ya no incluyen ayuda alguna, deben realizarse cuando el lector haya resuelto satisfactoriamente todos los anteriores.

Este texto está casi principal, aunque no exclusivamente, dedicado a la sentencia de recuperación de datos **select**, dado que es con mucho la más compleja. A su lado, las otras resultan simples reducciones o simplificaciones suyas. Esta sentencia presenta tres conceptos muy importantes sobre los que se sustenta la potencia del lenguaje SQL: la agrupación, la concatenación de tablas y las subconsultas. Estos conceptos se introducirán poco a poco con numerosos ejemplos.

El contenido de este libro puede agruparse en varias partes, cada una de las cuales se descompone en uno o más capítulos. A continuación se presentan brevemente cada una de las partes y cada una de los capítulos que las componen.

- La primera parte es, obviamente, una introducción general al mundo de las bases de datos y una introducción muy básica y rudimentaria del lenguaje SQL.
  - El capítulo 1 realiza una presentación general del mundo de las bases de datos, repasando su pasado, presente y futuro, y del lenguaje SQL. Finalmente presenta el ejemplo de base de datos que se va a emplear a lo largo de todo el texto.
  - El capítulo 2 inicia al lector en el lenguaje SQL y, concretamente, en la sentencia de recuperación de datos.
  - El capítulo 3 describe un conjunto de funciones y operadores que permiten realizar cálculos avanzados como por ejemplo las conversiones de tipos, extracciones de partes de una fecha, comparaciones avanzadas, etc.
- La segunda parte comienza ya a profundizar en el lenguaje SQL presentando las funciones de columnas y el concepto de agrupación.

- El capítulo 4 introduce las funciones de columna que suele incluir SQL. Estas son funciones que resumen toda una columna de una tabla en un solo valor, como por ejemplo, la suma, el máximo, la media, etc.
- El capítulo 5 presenta un concepto muy importante de SQL: la agrupación. Esta es una técnica muy útil que permite calcular un valor para cada grupo de filas de una tabla

- La tercera parte se mete ya a fondo en el lenguaje SQL presentando diversos conceptos muy importantes: la concatenación interna, la ordenación, las operaciones algebraicas, la concatenación externa y las subconsultas.
  - El capítulo 6 describe una forma de extraer información distribuida entre varias tablas: la concatenación interna de tablas. A las sentencias que emplean este tipo de operación se les denomina consultas multitabla dado que acceden a varias tablas.
  - El capítulo 7 presenta la ordenación del resultado de una consulta y las distintas operaciones algebraicas que ofrece este lenguaje.
  - El capítulo 8 presenta una variante de la concatenación interna de tablas: la concatenación externa de tablas, la cual permite procesar eficientemente tablas con valores nulos.
  - El capítulo 9 presenta un concepto muy importante de SQL: las subconsultas. Una subconsulta es una consulta metida dentro de otra consulta. Las subconsultas dotan de una gran potencia al lenguaje SQL.
- La cuarta parte describe dos aspectos muy importantes: la parte de definición de datos del lenguaje SQL y la actualización (inserción, borrado y modificación) de los datos de las tablas. En esta parte se presentan conceptos muy importantes como la creación de tablas, el borrado de tablas, la modificación del esquema de las tablas, la modificación del contenido de las tablas, etc.
  - El capítulo 10 realiza una introducción a la definición y modificación del esquema de las tablas y, además, a la actualización (inserción, borrado y modificación) del contenido de las tablas.
  - El capítulo 11 presenta los conceptos más avanzados de esta parte: vistas, índices, etc.
- La quinta parte se centra exclusivamente en presentar ejercicios y soluciones, es decir, su estudio no va a aportar nuevos conocimientos teóricos sobre el lenguaje SQL, pero sí va a permitir alcanzar unos mayores conocimientos aplicados.
  - El capítulo 12 contiene las soluciones a todos los ejercicios de autoevaluación de los capítulos anteriores. Estos ejercicios sirven para que el lector compruebe si ha alcanzado los objetivos previstos. Son ejercicios más complejos, sin ayuda ninguna, a los que el lector debe enfrentarse cuando haya resuelto sin problemas todos los ejercicios de cada capítulo.
  - El capítulo 13 presenta numerosos ejercicios de un nivel medio o alto que permiten practicar con todos los conceptos adquiridos en capítulos anteriores. Dada su mayor dificultad, el lector debe abordar estos problemas cuando haya resuelto satisfactoriamente todos los anteriores.

Finalmente, con estas líneas deseamos agradecer su ayuda a todas aquellas personas que han colaborado en las numerosas revisiones de este texto, entre las cuales hay que destacar a Carlos Serra Toro.

1 INTRODUCCIÓN

Este capítulo realiza una presentación general del lenguaje SQL, incluyendo la finalidad, motivación, orígenes y estándares. Asimismo, introduce los conceptos básicos necesarios sobre el modelo relacional. Finalmente describe la base de datos que se va a emplear a lo largo de todo el libro en los distintos ejemplos.

1.1 Origen y auge de las bases de datos

En los inicios de la informática, la información era procesada con aplicaciones construidas con lenguajes de programación tradicionales y era almacenada mediante un sistema de ficheros, habitualmente proporcionado por el sistema operativo. Más adelante, debido a las serias limitaciones y escasas posibilidades de los primeros sistemas de ficheros, éstos se fueron refinando y surgieron ciertas mejoras, proporcionando una mayor riqueza de características y operaciones (ficheros directos, ficheros secuencial-indexados, etc.).

Sin embargo, el avance más importante (y uno de los más importantes en el mundo de la informática) fue la aparición de las primeras bases de datos en los años 1960. Las ventajas que aportaron frente a los sistemas de ficheros han sido determinantes en su auge actual: mayor flexibilidad, mayor independencia del sistema operativo y del *hardware*, mayor tolerancia a fallos, mayor facilidad en el uso concurrente, el concepto de transacción, etc. Sin embargo, todas estas ventajas no son gratuitas pues requieren una elevada potencia de cómputo. Afortunadamente, los ordenadores no han cesado de incrementar su velocidad, lo cual ha permitido reducir drásticamente los costes del equipamiento informático necesario para hacer funcionar una base de datos.

Mención especial merece el campo de las bases de datos sobre ordenadores personales. A principios de los 1980 los primeros ordenadores personales, dada su escasa potencia, no disponían de bases de datos. Por ello, la información debía guardarse trabajando con el sistema de ficheros del sistema operativo. Posteriormente surgieron para estos ordenadores algunos paquetes de *software* que aportaban un sistema de ficheros secuencial-indexados, los cuales permitían al programador trabajar de forma más cómoda.

Conforme los ordenadores personales ganaron potencia, a finales de los 1980, comenzaron a surgir las primeras bases de datos (p.e. dBASE III+) para este tipo de ordenadores. Aunque en realidad éstas no eran tales pues carecían de numerosas e importantes características presentes en las bases de datos de los ordenadores grandes, estos sistemas se extendieron enormemente dado que permitían reducir drásticamente el tiempo necesario para desarrollar una aplicación. Por ejemplo, aplicaciones que con Pascal y un paquete de ficheros secuencial-indexados costarían varios meses en ser desarrolladas, podían terminarse en un solo mes con una de estas bases de datos.

Posteriormente, el incremento casi exponencial de la potencia de los ordenadores personales ha resultado en la difusión de una nueva generación de bases de datos más modernas, complejas y poderosas que sus predecesoras. En la actualidad, las nuevas bases de datos para ordenadores personales incluyen numerosas características y tienen poco que envidiar, excepto su potencia, a las bases de datos de los grandes sistemas. Por ejemplo, aplicaciones que con dBASE costarían un mes en ser desarrolladas, con una base de datos actual pueden terminarse en unas pocas horas.

Actualmente, aunque las grandes y medianas empresas disponen todas ellas de bases de datos desde hace muchos años, el interés en este campo no ha menguado, sino todo lo contrario. Por ejemplo, en 2002 el crecimiento de este campo fue del 30%. La informatización con bases de datos de las grandes y medianas empresas ha continuado y, lo que es más importante, su uso crece vertiginosamente en el mundo de la informática personal y de la pequeña empresa.

Las bases de datos se pueden agrupar en varios tipos o modelos teóricos: modelo jerárquico, en red y relacional. Las bases de datos jerárquicas surgieron en la década de 1960, poco después aparecieron las bases de datos en red y, un poco más tarde, las bases de datos relacionales.

El modelo relacional fue propuesto por primera vez por el Dr. Edgar Codd, de IBM, en 1970 y el primer sistema comercial apareció en 1976. Debido a las importantes ventajas que aporta en comparación con los dos modelos anteriores, los ha desplazado completamente del mercado. Hoy en día la gran mayoría de bases de datos comerciales pertenecen al modelo relacional.

En realidad, comienzan ya a extenderse bases de datos basadas en el modelo objeto-relacional, el cual es una mezcla del modelo relacional y del modelo de programación orientada a objetos, pero la vigencia del modelo relacional aún perdurará algunos años o lustros más.

1.2 Conceptos básicos de los sistemas relacionales

Una base de datos es un conjunto de datos relacionados entre sí. Un sistema de gestión de base de datos (SGBD) es un conjunto de programas que permiten almacenar y procesar la información contenida en una base de datos.

Una base de datos relacional es aquélla en la cual toda la información se almacena en tablas. Una tabla está formada por filas y columnas. Las bases de datos grandes pueden llegar a contener varias decenas de tablas, cada una con miles o millones de filas.

Cada tabla tiene un nombre único y un conjunto de filas y columnas. A continuación se presentan dos tablas de ejemplo. La primera contiene información relacionada con las facturas; la segunda, información relacionada con los clientes.

Tabla FACTURAS

CODFAC	CODCLI	FECHA	IVA	DTO
30	100	1-01-03	16	5
31	101	2-01-03		9
32	101		16	5
33	106	8-01-03	16	5

Tabla CLIENTES

CODCLI	NOMBRE	DIRECCIÓN	CODPUE
101	Alberto	Cuesta, 5	1000
102	Carlos	En proyecto, 3	1000
103	Pedro	Colón, 4	1001

Cada fila contiene información sobre una única entidad. Por ejemplo, cada fila de la tabla facturas contiene información sobre una factura.

Cada columna contiene información sobre una única propiedad o atributo de las entidades. Por ejemplo, la columna nombre de la tabla clientes contiene los nombres de los clientes.

Una celda es una intersección de una fila y de una columna. Cada celda puede contener bien un valor o bien ningún valor. Cuando no contiene ningún valor, se dice que tiene el valor nulo. El valor nulo puede tener dos orígenes: un valor desconocido o un valor no aplicable. Por ejemplo, la tabla de facturas tiene un valor nulo en la fecha de la factura con código 32.

Las filas de las tablas no están ordenadas ni tienen una posición fija. Por tanto, no se puede acceder a una determinada información de una tabla a través de su posición en la tabla (tercera fila, vigésima fila, etc.). Para acceder a una entidad hay que conocer alguna de sus propiedades o atributos.

La clave primaria es una columna o conjunto de columnas que identifican de forma única a cada una de las entidades (filas) que componen las tablas. En Access se la denomina clave principal a esta clave.

La clave ajena es una columna o conjunto de columnas cuyos valores coinciden con algunos valores de una clave primaria de una tabla.

Existen dos reglas que permiten mantener la integridad de la información de las bases de datos:

La *Regla de Integridad de Entidades* especifica que ninguna de las columnas que componen la clave primaria de una tabla puede contener valores nulos.

La *Regla de Integridad Referencial* especifica que las claves ajenas o bien tienen valores nulos o bien contienen valores tales que coinciden con algún valor de la clave primaria a la que referencian. Por ejemplo, las anteriores tablas no cumplen esta regla dado que la factura con código 33 referencia a un cliente que no existe (código de cliente 106).

Resulta muy conveniente que la información contenida en las bases de datos cumpla ambas reglas. En caso contrario, la información se encuentra en un estado inconsistente que a la larga sólo puede producir errores. Pensemos, por ejemplo, en facturas para clientes que no existen, en ventas de artículos que no existen, etc.

No todos los SGBD hacen cumplir estas reglas a los datos que contienen. Algunos sistemas permiten hacer cumplir ambas reglas de forma opcional. En este caso se recomienda encarecidamente indicar al sistema su obligado cumplimiento pues así el diseñador se libera de tener que comprobar la integridad de los valores tras cada actualización, borrado o inserción.

1.3 El lenguaje SQL

SQL (*Structured Query Language*) es un lenguaje de programación diseñado específicamente para el acceso a Sistemas de Gestión de Bases de Datos Relacionales (SGBDR). Como la mayor parte de los sistemas actuales son SGBDR y como el lenguaje SQL es el más ampliamente usado en éstos, se puede decir sin ningún género de dudas que este lenguaje es empleado mayoritariamente en los sistemas existentes hoy en día e indiscutiblemente no tiene rival alguno.

Este lenguaje es empleado en sistemas informáticos que van desde ordenadores personales muy básicos con apenas 64 MB de espacio en memoria central hasta los más potentes multiprocesadores y multicomputadores con decenas de procesadores superescalares de 64 bits.

Las principales ventajas que aportan SQL son dos:

- Su enorme difusión pues es empleado en la gran mayoría de los sistemas actuales.
- Su elevada potencia. Por ejemplo, operaciones que costarían semanas de duro esfuerzo en un lenguaje de programación tradicional pueden ser realizadas con SQL en tan sólo unos minutos.

El lenguaje SQL es un lenguaje de cuarta generación. Es decir, en este lenguaje se indica qué información se desea obtener o procesar, pero no cómo se debe hacerlo. Es labor interna del sistema elegir la forma más eficiente de llevar a cabo la operación ordenada por el usuario.

A la hora de describir el lenguaje SQL siempre hay que tomar una decisión difícil: decidir la variante de SQL que se va estudiar. Aunque los conceptos básicos son idénticos en todos los estándares y en todos los sistemas implementados, existen numerosas diferencias que dificultan la portabilidad. Desgraciadamente existen numerosos estándares de SQL (SQL-86, SQL-89, SQL-92, SQL-99, SQL-2003) y, lo que es peor, ninguno ha sido completamente aceptado. Además, las empresas fabricantes de SGBDR implementan el estándar que les da la gana y, lo que es peor, añaden y quitan características sin el menor rubor ni reparo.

Todo lo anterior hace bien difícil la elección de la variante de SQL en la que enfocar este texto. Así pues, para no centrar el libro en un estándar que nadie siga o centrarlo en un sistema comercial que deje de lado a otros sistemas y a los estándares, se va a procurar describir simultáneamente tanto un estándar como algunos de los sistemas comerciales más extendidos, procurando comentar las diferencias allí dónde las haya. Dado el carácter introductorio del texto este objetivo no va a resultar tan difícil pues las mayores divergencias surgen en los aspectos más avanzados. Las implementaciones comerciales del lenguaje SQL que se presentan son las de Microsoft Access, Oracle, PostgreSQL y MySQL al ser éstas de las más empleadas; la primera en el mundo de la pequeña empresa y la segunda entre las grandes compañías.

1.3.1 Partes de SQL

El lenguaje SQL consta de dos partes claramente diferenciadas:

- Lenguaje de Definición de Datos (en inglés *Data Definition Language* o DDL): Incluye aquellas sentencias que sirven para definir los datos o para modificar su definición, como por ejemplo la creación de tablas, índices, etc.
- Lenguaje de Manipulación de Datos (en inglés *Data Manipulation Language* o DML): Incluye aquellas sentencias que sirven para manipular o procesar los datos, como por ejemplo la inserción, borrado, modificación o actualización de datos en las tablas.

La primera parte, el DDL, se abordará en capítulos posteriores, mientras que ahora y en los capítulos inmediatos se van a estudiar las sentencias de manipulación de datos.

1.3.2 Sentencias del Lenguaje de Manipulación de Datos

SQL presenta cuatro sentencias de manipulación de datos:

- Sentencia **select**: Permite extraer información almacenada en la base de datos. Es una operación de sólo lectura.
- Sentencia **insert**: Permite insertar información en la base de datos.
- Sentencia **update**: Permite modificar información almacenada en la base de datos.
- Sentencia **delete**: Permite borrar información existente en la base de datos.

De estas cuatro sentencias, la más compleja y poderosa es sin duda la primera. De hecho, el funcionamiento y estructura de las tres últimas sentencias es un subconjunto de las posibilidades de la primera aplicadas a una tarea particular.

Por tanto, a continuación y en los temas siguientes se estudiará la sentencia **select**, dejando para el final las otras tres al ser su comportamiento mucho más sencillo y casi trivial en comparación con la primera.

1.3.3 Orígenes y estándares

El lenguaje SQL fue desarrollado por IBM dentro del proyecto System R a finales de 1970. Desde entonces ha ganando una gran aceptación y ha sido implementado por numerosos productos experimentales y, sobre todo, comerciales.

En 1986 y 1987 las organizaciones ANSI (*American National Standards Institute*) e ISO (*International Standards Organization*) publicaron el estándar SQL-86, oficialmente conocido como ANSI X3.135-1986 e ISO 9075:1987. Esta estandarización se concibió como un común denominador que debían poseer todas las implementaciones de SQL. Sin embargo, dada su limitación inicial, no incluyó muchas de las características entonces empleadas, que continuaron divergiendo. Este estándar ocupa alrededor de 100 páginas.

En 1989 las organizaciones ANSI e ISO mejoraron el estándar anterior añadiendo la integridad referencial y el soporte para otros lenguajes de programación. Este estándar fue oficialmente denominado ANSI X3.135-1989 e ISO/IEC 9075:1989 e informalmente conocido como SQL-89. Este estándar ocupa alrededor de 120 páginas.

Durante este tiempo X/Open publicó una especificación de SQL que no estaba ligada a los estándares anteriores, sino que reflejaba los productos existentes de sus participantes y accionistas. Finalmente, algunos años después la organización X/Open se alineó junto a los estandares ANSI e ISO.

En 1992 ANSI e ISO publicaron una nueva revisión del estándar SQL, conocido oficialmente como X3.135-1992 e ISO/IEC 9075:1992 e informalmente como SQL-92 o SQL2. Este estándar clasifica todas las características del lenguaje en varios niveles: *Entry SQL*, *Transitional SQL, Intermediate SQL* y *Full SQL*. X/Open aceptó el estándar de nivel más bajo, el *Entry SQL*, y diversas características del *Intermediate SQL*. Este estándar ocupa alrededor de 600 páginas.

El estándar SQL ha continuado evolucionando hasta hoy en día. Como una revisión y ampliación del estándar SQL-92 surgió el estándar SQL-99 o SQL3. Este ya no consta de niveles sino del núcleo (*Core SQL*) y de una parte no nuclear. Posee ciertos aspectos orientados a objetos. Este estándar ocupa alrededor de 2200 páginas.

El estándar SQL-2003 está ya más orientado hacia sistemas relacionales/orientados a objetos. Tanto las especificaciones de este estándar como las del anterior no están libremente disponibles, sino que se deben comprar a las organizaciones ANSI o ISO.

De todas formas, los distintos creadores de SGBD, tanto comerciales como no comerciales, que incluyen SQL no se han distinguido especialmente por su gran tradición en seguir los estándares, sino todo lo contrario. Más concretamente, dicen que siguen el estándar de turno, dicen que incluso aportan diversas características no incluidas en el estándar y, después, en letra más pequeña, dicen que algunas características del estándar no han sido incluidas. En resumen, han tomado del estándar de la fecha solamente lo que han querido y han añadido cuantas cosas han creído conveniente, incluso a veces sin respetar la sintaxis original propuesta en el estándar.

De hecho, a fecha de hoy algunos de los productos comerciales más vendidos y desarrollados por empresas consideradas muy serias aún no cumplen completamente el estándar SQL-92.

1.4 Base de datos de ejemplo

Siempre que se describe el manejo y programación de un sistema de gestión de base de datos, resulta muy conveniente la presentación de diversos ejemplos que muestren en la práctica los distintos conceptos teóricos.

Por ello, en este texto se ha elegido un ejemplo de base de datos, no muy complejo, pero sí lo suficiente para que se puedan estudiar en la práctica todos los conceptos más importantes. Este ejemplo es bastante real, aunque para su uso práctico en el mundo real podrían eliminarse algunas tablas, como la de pueblos y provincias. No obstante, se han dejado ambas dado que permiten practicar los distintos conceptos muy fácilmente con una escasa dificultad teórica.

La comprensión de este ejemplo es crucial para poder entender y asimilar mejor los capítulos siguientes. Por ello, se recomienda una lectura atenta de las distintas tablas que van a componer este ejemplo.

El ejemplo elegido va a ser el control de *stocks* y facturación de una determinada empresa. Su relativa sencillez, a la par que su posible uso profesional en la pequeña empresa, nos han llevado a elegirlo entre todos los posibles.

Seguidamente se presentan y describen las distintas tablas que lo componen. Para cada tabla se presenta su nombre y las columnas de que consta entre paréntesis. Las claves primarias aparecen subrayadas. Las claves ajenas están en cursiva.

- Tabla **provincias(codpro, nombre)**: Esta tabla almacena las provincias de España, cada una con su código de provincia (clave primaria) y su nombre.
- Tabla **pueblos(codpue, nombre, *codpro*)**: Almacena los pueblos de España o, por lo menos, aquéllos donde tenemos clientes. Para cada pueblo se dispone de su código de pueblo (clave primaria), su nombre y el código de la provincia a la que pertenece (clave ajena).
- Tabla **clientes(codcli, nombre, direccion, codpostal, *codpue*)**: Almacena información sobre los clientes de la empresa. Para cada cliente se dispone de su código de cliente (clave primaria), su nombre, su dirección, su código postal y el código de pueblo donde reside (clave ajena).
- Tabla **vendedores(codven, nombre, direccion, codpostal, *codpue, codjefe*)**: Almacena información sobre los vendedores de la empresa. Para cada vendedor se dispone de su código de vendedor (clave primaria), su nombre, su dirección, su código postal, el código de pueblo donde reside (clave ajena a la tabla pueblos) y el código de su jefe inmediato superior (clave ajena a la misma tabla de vendedores).
- Tabla **articulos(codart, descrip, precio, stock, stock_min)**: Almacena información sobre los artículos que ofrece la empresa y sus cantidades disponibles en el almacén (stocks). Para cada artículo se dispone de su código de artículo específico (clave primaria), su descripción, su precio actual, su stock y su stock mínimo, es decir, el valor umbral por debajo del cual se debe reponer.
- Tabla **facturas(codfac, fecha, *codcli*, *codven*, iva, dto)**: Almacena toda la información sobre las facturas, excepto sus líneas. Como en cada factura el número de líneas es variable, todas las líneas de todas las facturas se almacenan juntas en otra tabla. Para cada factura en esta tabla se guarda su código de factura (clave primaria), su fecha, el código del cliente que ha realizado la compra (clave ajena), el código del vendedor que ha realizado la venta (clave ajena), el iva aplicado y el descuento global de la factura.
- Tabla **lineas_fac(*codfac*, linea, cant, *codart*, precio, dto)**: Almacena información sobre las líneas de las facturas. Para cada línea se dispone del código de factura a la que pertenece (clave ajena), su número de línea, la cantidad de la línea, el código del artículo vendido (clave ajena), el precio al que se vende el artículo y el descuento que se debe aplicar en la línea. No hay que confundir este descuento, cuyo ámbito de aplicación es la línea, con el descuento global de la factura, el cual se halla obviamente en la tabla de facturas. La clave primaria de esta tabla va a ser la combinación del código de factura y del número de línea pues, por ejemplo, sólo existirá una única tercera línea de la factura 15.

Los nombres de las tablas y de sus columnas se han escrito sin acentuar para evitar problemas con algunos sistemas de gestión de bases de datos.

A continuación se muestra diversa información sobre las columnas de las tablas: si aceptan nulos y su tipo de datos (o dominio). Si en la segunda columna aparece el texto **not null**, entonces la columna no acepta nulos. La tercera columna muestra el tipo de datos: **VARCHAR2(x)** significa una tira de hasta **x** caracteres de longitud.

Tabla Provincias

Columna	¿Nulo?	Tipo de Datos
codpro	not null	VARCHAR2(2)

nombre	not null	VARCHAR2(30)

Como se puede ver, el código de la provincia es una tira de dos caracteres. Tanto el código como el nombre no aceptan nulos.

Tabla Pueblos

Columna	¿Nulo?	Tipo de Datos
codpue	not null	VARCHAR2(5)
nombre	not null	VARCHAR2(40)
codpro	not null	VARCHAR2(2)

Como se puede ver, el código de pueblo es una tira de 5 caracteres. Las tres columnas no aceptan nulos.

Tabla Clientes

Columna	¿Nulo?	Tipo de Datos
codcli	not null	NUMBER(5)
nombre	not null	VARCHAR2(50)
direccion	not null	VARCHAR2(50)
codpostal		VARCHAR2(5)
codpue	not null	VARCHAR2(5)

En la tabla de clientes el código de cliente es un número de hasta 5 dígitos. La única columna que acepta nulos es el código postal (éste puede ser desconocido).

Tabla Vendedores

Columna	¿Nulo?	Tipo de Datos
codven	not null	NUMBER(5)
nombre	not null	VARCHAR2(50)
direccion	not null	VARCHAR2(50)
codpostal		VARCHAR2(6)
codpue	not null	VARCHAR2(5)
codjefe	not null	NUMBER(5)

En la tabla de vendedores el código de cliente es también un número de hasta 5 dígitos. La única columna que acepta nulos es el código postal (éste puede ser desconocido).

Tabla Articulos

Columna	¿Nulo?	Tipo de Datos
codart	not null	VARCHAR2(8)
descrip	not null	VARCHAR2(40)
precio	not null	NUMBER(7,2)
stock		NUMBER(6)
stock_min		NUMBER(6)

En la tabla de artículos el código de artículo es una tira de hasta 8 caracteres. El precio es un número de hasta 7 dígitos, dos de los cuales son la parte fraccionaria (los céntimos de euros). Las columnas stock y stock_min son las únicas que aceptan nulos.

Tabla Facturas

Columna	¿Nulo?	Tipo de Datos
codfac	not null	NUMBER(6)
fecha	not null	DATE
codcli		NUMBER(5)
codven		NUMBER(5)
iva		NUMBER(2)
dto		NUMBER(2)

En la tabla de facturas el código de factura es un número de hasta 6 dígitos. La fecha es de tipo DATE. El resto de columnas aceptan valores nulos. El código de cliente o de vendedor puede ser nulo (valor desconocido). Si el descuento es nulo, se entiende que es cero.

Tabla Lineas_fac

Columna	¿Nulo?	Tipo de Datos
codfac	not null	NUMBER(6)
linea	not null	NUMBER(2)
cant		NUMBER(5)
codart	not null	VARCHAR2(8)
precio		NUMBER(7,2)
dto		NUMBER(2)

En la tabla de líneas de facturas la cantidad, el precio y el descuento pueden ser nulos. Si el descuento es nulo, se entiende que es cero.

El resto de este texto va a estar basado, pues, en este ejemplo. Por ello, se recomienda encarecidamente un esfuerzo especial en la comprensión de las distintas tablas anteriores y sus implicaciones, lo cual no debe resultar por otro lado muy costoso.

2 INICIACIÓN A SQL

El objetivo de este capítulo es iniciar brevemente al lector en la sentencia de recuperación de datos del lenguaje SQL.

2.1 Iniciación a la sentencia select

La sentencia **select** más sencilla consta de dos cláusulas: la cláusula **select** y la cláusula **from**. Habitualmente cada una se escribe en una línea distinta para mejorar la legibilidad, aunque nada impide escribirlas en una única línea.

Su formato es el siguiente:

```
select * | columna1, columna2, columna3,...
from   tabla ;
```

Toda sentencia debe terminar obligatoriamente con el carácter punto y coma (**;**).

La cláusula **select** permite especificar qué información se desea obtener. Se pueden poner tantas columnas de la tabla como se desee. Además, pueden ponerse expresiones de una o más columnas de la tabla. El carácter ***** indica que se muestren todas las columnas. En términos matemáticos, esta cláusula permite realizar una operación de proyección.

La cláusula **from** permite indicar de qué tabla se deben extraer los datos.

Cuando el SGBD ejecuta una sentencia con estas dos cláusulas, examina en primer lugar la cláusula **from** para saber qué tablas debe procesar y, a continuación, examina la cláusula **select** para determinar qué información se desea mostrar a partir de dichas tablas.

- **Ejercicio**: Mostrar el código y nombre de las provincias.
  Solución: A continuación se muestras tres posibles soluciones que obtienen exactamente el mismo resultado.

```
select * from provincias;

select codpro, nombre from provincias;

select codpro, nombre
from   provincias;
```

- **Ejercicio**: Mostrar el nombre y después el código de las provincias.
  Solución:

```
select nombre, codpro
from   provincias;
```

- **Ejercicio**: Mostrar el código de artículo y el doble del precio de cada artículo.
  Solución:

```
select codart, precio * 2
from   articulos;
```

- **Ejercicio**: Mostrar el código de factura, número de línea e importe de cada línea (sin considerar impuestos ni descuentos).
  Solución:

```
select codfac, linea, cant * precio
from   lineas_fac;
```

- **Ejercicio**: Mostrar el código de factura, número de línea e importe de cada línea.
  Solución:

```
select codfac, linea, cant * precio * ( 1 - dto / 100 )
from   lineas_fac;
```

2.2 Modificador distinct

La sentencia **select** admite opcionalmente el uso del modificador **distinct** en su primera cláusula, tal como se muestra a continuación:

```
select [ distinct ] * | columna1, columna2, columna3,...
from   tabla ;
```

Esta palabra es un modificador y no es una función por lo que no necesita paréntesis. Su función es obvia: eliminar filas o valores repetidos.

- **Ejercicio**: Mostrar los distintos tipos de ivas aplicados en las facturas.
  Solución:

```
select distinct iva
from   facturas ;
```

2.3 Restricción en las filas

En numerosas ocasiones resulta conveniente restringir o seleccionar sólo algunas filas que cumplen una determinada condición de entre todas las existentes en una tabla. Esta operación se lleva a cabo con la cláusula **where**, la cual es opcional y de aparecer, deber hacerlo siempre tras la cláusula **from**.

A esta palabra le debe seguir una expresión booleana o lógica que devuelva cierto o falso. Esta condición se evalúa para cada fila. Si para una fila da cierto, entonces la fila aparece en el resultado. En caso contrario (la condición devuelve falso o desconocido), la fila es ignorada y se descarta del resultado.

La expresión booleana puede incluir cualquier combinación correcta de otras expresiones lógicas conectadas con los operadores lógicos **and** y **or**.

Cuando el SGBD ejecuta una sentencia con las tres cláusulas **select**, **from** y **where**, examina en primer lugar la cláusula **from** para saber qué tablas debe procesar, a continuación realiza una criba dejando sólo aquellas filas que cumplan la condición o condiciones de la cláusula **where** y, finalmente, examina la cláusula **select** para determinar qué información se desea mostrar a partir de dichas tablas.

- **Ejercicio**: Mostrar el código y nombre de aquellas provincias cuyo código es menor que '20'.
  Solución:

```
select codpro, nombre
from   provincias
where  codpro < '20' ;
```

- **Ejercicio**: Mostrar los distintos tipos de descuentos aplicados por los vendedores cuyos códigos no superan el valor 50.
  Solución:

```
select distinct dto
from   facturas
where  codven <= 50 ;
```

- **Ejercicio**: Mostrar el código y descripción de aquellos artículos cuyo stock iguala o supera las 50 unidades.
  Solución:

```
select codart, descrip
from   articulos
where  stock >= 50 ;
```

- **Ejercicio**: Mostrar el código de factura y fecha de las facturas con iva 16 y del cliente 100.

Solución:

```
select codfac, fecha
from   facturas
where  iva = 16
and    codcli = 100 ;
```

- **Ejercicio**: Mostrar el código y fecha de las facturas del cliente 100 con iva 16 o con descuento 20.
  Solución:

```
select codfac, fecha
from   facturas
where  codcli = 100
and    ( iva = 16
or       dto = 20 ) ;
```

- **Ejercicio**: Mostrar el código de factura y el número de línea de aquellas líneas de facturas que superan los 100 euros sin considerar descuentos ni impuestos.
  Solución:

```
select codfac, linea
from   lineas_fac
where  cant * precio > 100.0 ;
```

2.4 Ejecución de sentencias

Cuando el SGBD ejecuta una sentencia, lo hace siempre de la misma forma, aplicando el siguiente método:
Examina la cláusula **from** para saber con qué tablas va a trabajar.
Si existe la cláusula **where**, realiza una criba dejando sólo aquellas filas que cumplan la condición o condiciones de esta cláusula.
A partir del contenido de la cláusula **select** determina qué información se desea mostrar de las filas previamente seleccionadas.
Si existe el modificado **distinct**, elimina las filas repetidas del resultado anterior.

2.5 Ejercicios

- **Ejercicio 2.1**: Mostrar el código de artículo y la cantidad de las líneas de la factura cuyo código es 105.
  Solución:

```
select codart, cant
from   lineas_fac
where  codfac = 105 ;
```

- **Ejercicio 2.2**: Mostrar el código de artículo y el precio de aquellos artículos cuyo precio supera los 2,05 euros y cuyo stock supera las 100 unidades.
  Solución:

```
select codart, precio
from   articulos
where  precio > 2.05
and    stock > 100 ;
```

- **Ejercicio 2.3**: Mostrar el código de artículo y la cantidad de aquellas líneas cuyo descuento es igual a 10 o cuyo precio supera los 5,05 euros.
  Solución:

```
select codart, cant
from   lineas_fac
where  dto = 10
or     precio > 5.05 ;
```

- **Ejercicio 2.4**: Tipos de descuentos aplicados en las facturas del cliente 222.
  Solución:

```
select distinct dto
from   facturas
where  codcli = 222 ;
```

- **Ejercicio 2.5**: Mostrar el código de artículo, la cantidad, el precio unitario y el importe una vez aplicado el descuento de cada una de las líneas de la factura 325.
  Solución:

```
select codart, cant, precio, cant * precio * ( 1 - dto/100 )
from   lineas_fac
where  codfac = 325 ;
```

2.6 Autoevaluación

- **Ejercicio 1**: Mostrar el código y nombre de aquellos vendedores cuyo jefe tiene el código 125.
- **Ejercicio 2**: Mostrar el código y descripción de aquellos artículos cuyo stock en el almacén supera los 100 euros.
- **Ejercicio 3**: Mostrar el código, sin que salgan repetidos, de los artículos vendidos en las facturas con código inferior a 100.

3 FUNCIONES Y OPERADORES ESCALARES

Este capítulo presenta las funciones y operadores escalares del lenguaje SQL. Son funciones que devuelven un valor modificado por cada valor inicial. Dado el elevado número de funciones y operadores que contienen tanto el estándar como las distintas implementaciones, en este capítulo se describen únicamente los más importantes. En Internet y en los manuales de referencia se pueden encontrar listas más exhaustivas.

En el caso de las funciones y operadores escalares la variación entre los distintos SQL es bastante alta. Afortunadamente, dado que estas funciones no son excesivamente complicadas, pasar de un SQL a otro no cuesta demasiado tiempo. Es más, en muchos casos se tratan de los mismos conceptos pero con nombres distintos. En el resto del capítulo se van a describir las funciones y operadores escalares más importantes tanto del estándar SQL-99 como de varias implementaciones comerciales de SQL.

3.1 Expresiones y Tipos

En la mayor parte de los casos SQL permite emplear una expresión en lugar de una columna. Las expresiones son una parte crucial dentro de una consulta. Las expresiones pueden construirse empleando los operadores matemáticos habituales y también las funciones escalares.

Toda expresión tiene un tipo determinado en su resultado. Es conveniente no mezclar los tipos para evitar problemas. Puede que algunos sistemas sean más laxos y permitan las mezclas obteniendo el resultado esperado, pero otros sistemas pueden ser más ortodoxos y el resultado puede ser muy distinto.

Un error muy habitual es la comparación de tiras de caracteres y números. Algunos sistemas realizan una conversión previa internamente y el resultado es el esperado. Sin embargo, en otros sistemas esto no es así y se puede producir un error o devolver un resultado no esperado.

Otro error frecuente es operar con valores enteros (sin decimales) cuando en realidad se desea trabajar con valores reales (con decimales). En algunos sistemas la operación 1/3 devuelve cero pues tanto el numerador como el denominador son enteros y, por tanto, se aplica la división entera. Si se desea trabajar con números reales puede realizarse con una operación de conversión explícita o bien añadiendo una parte fraccionaria. Si se desea realizar una división real, debería escribirse la instrucción 1.0/3.0.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select codpro, nombre
from   provincias
where  codpro > 20 ;
```

Solución: Esta sentencia puede funcionar correctamente en algunos sistemas, pero no es aconsejable escribirla de esta forma pues en la cláusula **where** se compara la columna **codpro** (que es una tira de caracteres con dos caracteres) y un número entero. Sería mucho más conveniente reescribirla de la siguiente forma:

```
select codpro, nombre
from   provincias
where  codpro > '20' ;
```

3.2 Operadores de comparación

El lenguaje SQL incluye los operadores habituales de comparación: =, <>, >, >=, <, <=, etc. No obstante, además de éstos, suele incluir diversos operadores avanzados que simplifican y reducen bastante las expresiones si se usan adecuadamente, como son los operadores **between**, **in** y **like**.

3.2.1 Operador between

Este operador, al igual que el siguiente, no introduce ninguna operación que no se pueda realizar utilizando los operadores de comparación tradicionales, pero permite reducir significativamente las expresiones.

Su funcionamiento es muy sencillo: **a between b and c** devuelve cierto si **a** se halla entre **b** y **c** ambos inclusive, y falso en caso contrario. Es decir, devuelve cierto si **a** es mayor o igual que **b** y **a** es menor o igual que **c**. La siguiente línea resume su funcionamiento:

a between b and c equivale a **(b <= a) and (a <= c)**.

El funcionamiento del operador **not between** es, obviamente, justo el contrario que el anterior. La siguiente línea resume su funcionamiento:

a not between b and c equivale a **(a < b) or (c < a)**.

Su funcionamiento es completamente idéntico en SQL-99, Access, Oracle, PostgreSQL y MySQL.

- **Ejercicio**: Escribir una expresión que devuelva el código de cliente y el nombre de aquellos clientes cuyos códigos se encuentran comprendidos entre 100 y 200, sin incluir éstos.
  Solución:

```
select codcli, nombre
from   clientes
where  codcli between 101 and 199 ;
```

- **Ejercicio**: Escribir una expresión que devuelva todas las columnas de los artículos cuyo stock no se halla entre el stock mínimo menos 500 unidades y el stock mínimo más 500 unidades.
  Ayuda: Uso del operador *not between* y del carácter * en la cláusula *select*.
  Solución:

```
select *
from   articulos
where  stock not between stock_min - 500 and stock_min + 500
;
```

3.2.2 Operador in

Su funcionamiento es muy sencillo: **a in (b, c, d,..., z)** devuelve cierto si **a** es igual a alguno de los valores incluidos en la lista entre paréntesis (**b, c, d,..., z**) y devuelve falso en caso contrario. La siguiente línea resume su funcionamiento:

a in (b, c, d) equivale a **(a = b) or (a = c) or (a = d)**.

El funcionamiento del operador **not in** es justo el contrario: **a not in (b, c, d,..., z)** devuelve cierto si **a** no es igual a ninguno (o sea, es distinto de todos) de los valores incluidos en la lista entre paréntesis (**b, c, d,..., z**) y devuelve falso en caso contrario. La siguiente línea resume su funcionamiento:

a not in (b, c, d) equivale a **(a <> b) and (a <> c) and (a <> z)**.

Su funcionamiento es completamente idéntico en los distintos SQL de SQL-99, Access, Oracle, PostGreSQL y MySQL.

♦ **Ejercicio**: Escribir una expresión que devuelva el código y nombre de los pueblos pertenecientes a la comunidad valenciana (Alicante tiene el código de provincia '03'; Castellón, el '12' y Valencia, el '46').
Solución:

```
select codpue, nombre
from   pueblos
where  codpro in ( '03', '12', '46' ) ;
```

3.2.3 Operador like

El operador **like** de SQL permite comparar cadenas de caracteres usando comodines. La principal diferencia en el uso de este operador en las distintas implementaciones de SQL radica en los comodines:

- Cualquier tira de caracteres de cualquier longitud se representa con el carácter comodín "%" en el estándar SQL-99, en Oracle y en PostgreSQL. En cambio, en Access se emplea el comodín "*".
- Cualquier carácter se representa con el carácter comodín "_" en el estándar SQL-99, en Oracle y en PostgreSQL. En cambio, en Access se emplea el comodín "?".

En resumen, la nomenclatura de comodines de Oracle es más estándar, pero la nomenclatura de Access es más popular e intuitiva.

Además, hay que recordar que el uso de mayúsculas y minúsculas difiere entre Access y Oracle: el primero no distingue entre mayúsculas y minúsculas, mientras que el segundo sí. Por tanto, en Access la expresión **like "*García*"** devolverá todos aquellos García, independientemente de cómo estén escritos (con cualquier combinación indistinta de mayúsculas y minúsculas).

El operador **not like** trabaja justo al revés que el operador **like**.

♦ **Ejercicio**: ¿Qué devuelven las siguientes sentencias?

```
nombre like 'a_b%'
nombre not like 'a_b%'
```

Solución: La primera devuelve cierto si el primer carácter de nombre es una 'a', el tercero es una 'b' y tras éste aparece cualquier tira de caracteres. La segunda sentencia devuelve justo lo contrario de la primera.

3.3 Operadores y funciones de elección

Seguidamente se muestran unos operadores denominados de elección pues permiten devolver un valor u otro en función de una expresión. Se presenta en primer lugar el operador **case**, perteneciente al estándar SQL-99 y que aparece también en Oracle 9i y en PostGreSQL. Sin embargo, no se halla en Oracle 8i ni en Access.

A continuación se presentan diversos operadores de Access y Oracle no estándares que permiten realizar funciones parecidas. Se incluyen estos operadores no estándares debido a que no todos los sistemas disponen del operador **case**.

3.3.1 Operador estándar case

El operador estándar de elección entre múltiples valores es el operador **case**. Tiene dos formas de funcionamiento algo distintas, pero muy intuitivas. A continuación se describen ambas.

La siguiente expresión devuelve el valor **ret1** si expresión se evalúa a **val1** y a continuación termina. Si no, devuelve el valor **ret2** si expresión se evalúa a **val2** y así sucesivamente. Si ninguna de las igualdades anteriores se ha cumplido, se devuelve **retn**. Si no existe cláusula **else** y ninguna de las igualdades anteriores se ha cumplido, entonces devuelve el valor **null**.

```
case expresión when val1 then ret1
               when val2 then ret2
                         …
                         else retn
end
```

La siguiente expresión devuelve el valor **ret1** si **expresion1** se evalúa a cierto y a continuación termina. Si no, devuelve el valor **ret2** si expresión se evalúa a cierto y así sucesivamente. Si ninguna de las expresiones anteriores se ha evaluado a cierto, se devuelve **retn**. Si no existe cláusula **else** y ninguna de las expresiones anteriores se ha evaluado a cierto, entonces devuelve el valor **null**.

```
case when expresión1 then ret1
     when expresión2 then ret2
                     …
                     else retn
end
```

♦ **Ejercicio**: Escribir una expresión que devuelva el código de factura y el texto 'normal' cuando el iva vale 16, el texto 'reducido' cuando el iva vale 7 y el texto 'otros' en cualquier otro caso para las facturas con descuento del 20 %.
Solución: Seguidamente se presentan dos soluciones equivalentes.

```
select codfac, case iva when 16 then 'normal'
                        when  7 then 'reducido'
                                else 'otros'
               end
from   facturas
where  dto = 20 ;
```

```
select codfac, case when iva = 16 then 'normal'
                    when iva =  7 then 'reducido'
                                  else 'otros'
               end
from   facturas
where  dto = 20 ;
```

3.3.2 Funciones no estándares de elección: decode, iif, switch y choose

En Oracle 8i no existe el operador **case**. No obstante, la función **decode** permite devolver un valor u otro según el resultado de una expresión. La llamada **decode(a, v1, ret1, v2, ret2,..., retn)** devuelve **ret1** si **a** es igual a **v1**; si no lo es y **a** es igual a **v2**, entonces devuelve **ret2** y así sucesivamente. Si el valor de **a** no iguala a ningún valor y el número de parámetros es impar, entonces devuelve el valor nulo. Si el valor de **a** no iguala a ningún valor y el número de parámetros es par, entonces devuelve el último parámetro.

En Access tampoco existe el operador **case**. En cambio, existen tres funciones que permiten elegir y devolver un valor de entre varios. Éstas son:

- Si Inmediato **Iif**. Su funcionamiento es muy sencillo e intuitivo. La llamada a la función **Iif(a, b, c)** devuelve **b** si **a** es cierto y **c** en caso contrario.
  Por ejemplo, si se desea mostrar la tira de caracteres "mayor" si el precio es superior a 100 y "normal" en cualquier otro caso, se debería hacer lo siguiente:
  - En Access:

```
iif( precio>100, "mayor", "normal" )
```

- En Oracle 8i:

```
decode( sign(precio-100), 1, 'mayor', 'normal' )
```

- **Switch**. Esta función suele ser menos útil que la función **Iif**. Debe tener un número par de parámetros. Si el primer parámetro es cierto, se devuelve el segundo y se termina; en caso contrario, si el tercer parámetro es cierto, se devuelve el cuarto y se termina; y así sucesivamente. Para devolver un valor final si no se cumple ninguna de las condiciones anteriores, se pone como penúltimo parámetro (última condición) el valor cierto, con lo cual esta condición se cumple siempre y se devolverá el último parámetro.
  Por ejemplo, si se desea mostrar la tira "normal" cuando el iva vale 16, "reducido" cuando el iva vale 7 y "otro" en cualquier otro caso se debería teclear lo siguiente:
  - En Access:

```
switch( iva=16, "normal", iva=7, "reducido", True, "otro" )
```

  - En Oracle 8i:

```
decode( iva, 16, 'normal', 7, 'reducido', 'otro' )
```

- **Choose**. Esta función también tiene una aplicación parecida. Si el primer parámetro se evalúa a uno, devuelve el segundo parámetro; si el primer parámetro se evalúa a dos, devuelve el tercero; y así sucesivamente.
  Por ejemplo, si se desea mostrar la tira "efectivo" cuando el modo de pago vale 1, "tarjeta" cuando modo de pago vale 2 y "cheque" cuando el modo de pago vale 3, se debería teclear lo siguiente:
  - En Access:

```
choose( modoPago, "efectivo", "cheque", "tarjeta" )
```

  - En Oracle 8i:

```
decode( modoPago, 1, 'efectivo', 2, 'cheque', 3, 'tarjeta'
)
```

3.4 Manejo de los valores nulos

Muchas veces en la realidad ocurren excepciones que producen falta de información. En el contexto de una base de datos, esta situación se representa dentro de la columna correspondiente por medio de un nulo. Un valor nulo (**null**) dentro de una celda significa que esa información se desconoce, es irrelevante o que no es aplicable a la fila en cuestión. A este respecto es muy importante resaltar que un valor nulo tiene un significado muy diferente de un valor cero, de una cadena en blanco, o de un valor booleano igual a **false**. Un nulo se corresponde con una ausencia de valor que no puede ser comparado ni operado directamente junto con ningún otro valor de ningún tipo de datos.

El procesamiento de los valores nulos es una parte muy empleada en casi todos los entornos debido a que las bases de datos reales contienen numerosos valores nulos en sus celdas que hay que procesar debidamente.

Las funciones más habituales para el procesamiento de valores nulos son dos: la detección del valor nulo y la conversión del valor nulo en un valor más tratable.

3.4.1 Tablas de verdad de los valores nulos

La especificación del estándar SQL no indica cómo se deben manejar los nulos, simplemente propone su utilización y deja el resto de cuestiones abiertas para que las resuelva cada fabricante de software a su manera.

Lo que sí proporciona el estándar SQL son las siguientes tablas de verdad para los operadores AND, OR y NOT. La ausencia de valor de los nulos hace que cualquier comparación de un valor con un nulo devuelva un valor desconocido. Por esta razón, se consideran tres posibles valores (V de verdadero, F de falso y D de desconocido), dando lugar a una lógica de tres valores para SQL. A continuación se muestran las tablas de verdad para los tres operadores lógicos:

AND	**V**	**D**	**F**
V	V	D	F
D	D	D	F
F	F	F	F

OR	**V**	**D**	**F**
V	V	V	V
D	V	D	D
F	V	D	F

NOT	
V	F
D	D
F	V

Como ya hemos dicho anteriormente, la ausencia de información asociada a los nulos imposibilita su comparación con el resto de valores de la base de datos. Por ejemplo, un nulo nunca va a ser igual a otro valor, y un nulo tampoco va a ser nunca diferente de otro valor. Ni siquiera se puede decir que dos nulos sean iguales o diferentes entre sí. Únicamente se puede decir que la verdad se desconoce debido a la ausencia de información.

Además los nulos presentan el serio inconveniente de que se propagan. Por ejemplo, si se desea calcular la media del stock de los artículos en la base de datos con la consulta **`select avg(stock) from articulos`**; en algunos sistemas de gestión de bases de datos la existencia de un artículo con un stock nulo provocaría una media igual a nulo. Esto se debe a que un nulo no se puede sumar a un entero, ni dividir por otro valor para producir un real. Como veremos más adelante, afortunadamente la mayoría de los sistemas eliminan los nulos antes de aplicar a un conjunto de valores una función de columna.

En el proceso de creación de tablas los diseñadores suelen tener la opción de especificar qué columnas pueden aceptar nulos y cuáles no. Por esta razón, antes de formular sus consultas, los programadores de bases de datos deben considerar que la probabilidad de que haya nulos y, cuando sea necesario, deben utilizar los operadores explicados para su detección y conversión.

3.4.2 Detección de valores nulos

La detección de valores nulos se suele realizar en la mayor parte de variantes de SQL (SQL-99, Oracle 9i, MySQL, PostGreSQL, etc.) con los operadores **is null** e **is not null**. En Access la detección de un valor nulo se realiza con la función **IsNull**.

Es importante destacar que los operadores de igualdad y de desigualdad no sirven para comparar con el valor nulo. Así pues, las expresiones **columna = null** y **columna <> null** siempre devolverán desconocido, valga lo que valga la parte izquierda.

Por ejemplo, para detectar si el descuento es nulo, se debería hacer lo siguiente:

- En el estándar y en la mayor parte de implementaciones:
  ```
  dto is null
  ```
- En Access:
  ```
  IsNull( dto )
  ```

Por ejemplo, para detectar si el descuento no es nulo, se debería hacer lo siguiente:

- En el estándar y en la mayor parte de implementaciones:
  ```
  dto is not null
  ```
- En Access:
  ```
  Not IsNull( dto )
  ```

♦ **Ejercicio**: Escribir una sentencia que muestre los códigos y fechas de aquellas facturas sin código de cliente o sin código de vendedor.
Solución:

```
select codfac, fecha
from   facturas
where  codcli is null
or     codven is null ;
```

♦ **Ejercicio**: Se desea escribir una sentencia que muestre el código de los artículos con un stock superior a 50 sabiendo que este campo es nulo en algunas ocasiones. Dadas las tres sentencias siguientes, ¿qué realiza cada una de ellas?

```
select codart
from   articulos
where  stock > 50;
```

```
select codart
from   articulos
where  not(stock <= 50);
```

```
select codart
from   articulos
where  ( stock > 50 )
or     (stock is null);
```

Solución: La primera sentencia devuelve aquellos artículos que efectivamente tienen un stock superior a 50 unidades, haciendo caso omiso de los artículos con un stock nulo o con un stock inferior. La segunda sentencia recupera los mismos artículos que la anterior (pues la negación de desconocido es desconocido). La tercera sentencia recupera los artículos cuyo stock supera las 50 unidades o es desconocido, es decir, devuelve aquellos artículos que podrían tener un stock superior a 50.

3.4.3 Conversión de valores nulos

La conversión de valores nulos se suele realizar en la mayor parte de variantes de SQL (SQL-99, Oracle 9i, MySQL, PostGreSQL, etc.) con la función **coalesce**. En cambio, en Oracle 8i se debe emplear la función **nvl** y en Access, la función **Nz**.

Por ejemplo, para devolver un cero si el descuento es nulo, se debería hacer lo siguiente:

- En el estándar y en la mayor parte de implementaciones:

```
coalesce( dto, 0 )
```

  En realidad, el formato de esta función admite *n* parámetros. Su funcionamiento en el caso generalizado es muy sencillo: devuelve el primer parámetro no nulo comenzando desde la izquierda. El comportamiento con dos parámetros es una simplificación del caso general.

- En Oracle 8i:

```
nvl( dto, 0 )
```

- En Access:

```
Nz( dto, 0 )
```

  En este caso el segundo parámetro es opcional: si no está, devuelve cero o la tira vacía, dependiendo de ciertos criterios internos de Access. No obstante, se recomienda encarecidamente el uso explícito del segundo parámetro debido a que estos criterios pueden fallar, devolviendo por ejemplo la tira vacía cuando se espera un cero.

♦ **Ejercicio**: Escribir una sentencia que muestre los códigos de aquellos artículos (sin mostrar repetidos) vendidos alguna vez sin descuento en sus líneas de facturas. Se considera que un artículo no tiene descuento cuando éste es cero o nulo.

Solución:

```
select distinct codart
from   lineas_fac
where  coalesce( dto, 0 ) = 0 ;
```

♦ **Ejercicio**: Escribir una sentencia que muestre código, la fecha y el descuento de las facturas sin iva (iva nulo o cero), visualizando un cero en aquellas facturas cuyo descuento sea nulo.
Ayuda: Uso de la función *nvl*.
Solución:

```
select codfac, fecha, nvl( dto, 0 )
from   facturas
where  nvl( iva, 0 ) = 0 ;
```

3.5 Procesamiento de fechas

El procesamiento de fechas es una labor muy importante y habitual en la mayor parte de las bases de datos reales. Por ello, tanto los estándares como casi todas las implementaciones de SQL incluyen numerosos operadores y funciones para el procesamiento y conversión de fechas y horas. Los operadores y funciones para procesar y convertir fechas son ciertamente distintos en los diferentes estándares e implementaciones de SQL. A continuación se describen los más habituales e importantes, pudiéndose encontrar en Internet y en los manuales de referencia más información sobre el resto.

3.5.1 Fecha actual

Para conseguir la fecha actual se suelen usar las siguientes funciones:

- En SQL-99:
  ```
  current_date
  ```
- En Oracle:
  ```
  sysdate
  ```
- En Access:
  ```
  Date()
  ```

3.5.2 Extracción de una parte de una fecha

Para extraer una parte de una fecha, como por ejemplo el año, se debería hacer lo siguiente:

- En SQL-99:
  ```
  extract( year from fecha )
  ```
  ```
  date_part( 'year', fecha )
  ```
- En Oracle:
  ```
  to_char( fecha, 'yyyy' )
  ```
- En Access:
  ```
  DatePart( "yyyy", fecha )
  ```

Por ejemplo, para seleccionar las facturas del año pasado, en Oracle se debería usar la cláusula siguiente:

```
where  to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1;
```

En cambio, en Access para realizar lo mismo se debería utilizar la cláusula siguiente:

```
where  DatePart( "yyyy", fecha ) =
       DatePart( "yyyy", Date() ) - 1;
```

Nótese que en Access la función **DatePart** devuelve un entero, mientras que en Oracle la función **to_char** devuelve obviamente una tira de caracteres, la cual se debe convertir posteriormente a un número con la función **to_number** si se desea realizar alguna operación matemática.

- **Ejercicio**: Escribir una expresión que devuelva el nombre del mes actual.
  Ayuda: Uso de la función *to_char* con el formato de fecha *month*, de la función *sysdate* y de la tabla *dual*.
  Solución:

```
select to_char( sysdate, 'month' )
from   dual ;
```

- **Ejercicio**: Escribir una expresión que devuelva el código, fecha, iva y descuento de las facturas del mes de diciembre del año pasado.
  Solución: Se emplea la sintaxis de Oracle.

```
select codfac, fecha, iva, dto
from   facturas
where  to_char( fecha, 'mm' ) = '12'
and    to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1;
```

3.5.3 Conversión de una fecha al formato deseado

Existen distintas funciones para convertir una fecha al formato deseado (o incluso para extraer una parte de una fecha) como una tira de caracteres. Véanse los siguientes ejemplos:

- En SQL-99:

```
to_char( fecha, 'dd-mm-yyyy' )
```

- En Oracle:

```
to_char( fecha, 'dd-mm-yyyy' )
```

- En Access:

```
Format( fecha, "dd-mm-yyyy" )
```

Como se puede ver, todas las funciones de procesamiento de fechas son ciertamente distintas, pero conceptualmente muy semejantes.

3.6 Procesamiento de cadenas de caracteres

El procesamiento de cadenas de caracteres es una parte muy empleada en la mayoría de las bases de datos. Por ello, tanto los estándares como casi todas las implementaciones de SQL incluyen numerosos operadores y funciones para procesar cadenas de caracteres. Los operadores y funciones para este tipo de tareas son bastante distintos en los diferentes estándares e implementaciones de SQL. A continuación se describen los más habituales e importantes, pudiéndose encontrar en Internet y en los manuales de referencia más información sobre el resto de operadores y funciones.

3.6.1 Delimitación de cadenas

La delimitación de una cadena se realiza con las comillas. Algunos sistemas, como Oracle, obligan a emplear las comillas simples. Otros, como Access, permiten usar tanto las comillas simples como las dobles.

3.6.2 Concatenación de cadenas

La concatenación de cadenas de caracteres se realiza en SQL-99 y en Oracle con el operador "||", mientras que en Access se realiza con el operador "**&**".

3.6.3 Longitud de una cadena

El cálculo de la longitud de una cadena de caracteres, una labor en ocasiones necesaria, se realiza de la siguiente forma:

- En SQL-99:
  ```
  char_length( tira )
  ```
- En Oracle:
  ```
  length( tira )
  ```
- En Access:
  ```
  Len( tira )
  ```

♦ **Ejercicio**: Escribir una expresión que devuelva la longitud del nombre de cada cliente cuyo código se encuentre comprendido entre 100 y 200, inclusive.
Solución: Se emplea la sintaxis de Oracle.

```
select length( nombre )
from   clientes
where  codcli between 100 and 200 ;
```

3.6.4 Extracción de una parte de una cadena

Las funciones para extracción de una parte de una cadena de caracteres varían ligeramente. A continuación se muestra en un ejemplo el uso de esta función en los distintos sistemas. El objetivo de este ejemplo es extraer 3 caracteres a partir de la posición 2, es decir, extraer el segundo, tercer y cuarto caracteres.

- En SQL-99:
  ```
  substring( tira from 2 for 3 )
  ```
- En Oracle:
  ```
  substr( tira, 2, 3 )
  ```
- En Access:
  ```
  Mid( tira, 2, 3 )
  ```
  En este sistema existen también las funciones **Left** y **Right** para extraer un determinado número de caracteres desde la izquierda o desde la derecha.

♦ **Ejercicio**: Escribir una expresión que devuelva el código y nombre de aquellos vendedores cuya última letra del nombre es una 'E'.
Solución: Se emplea la sintaxis de Oracle.

```
select codven, nombre
from   vendedores
where  substr( nombre, length( nombre ), 1 ) = 'E' ;
```

3.6.5 Conversiones a mayúsculas y minúsculas

El estándar de SQL y la mayor parte de implementaciones de este lenguaje contienen un par de funciones para convertir una tira de caracteres (todos sus caracteres, se entiende) a mayúsculas o minúsculas. A continuación se muestran los nombres de las funciones en el estándar SQL-99, seguido por la mayor parte de las implementaciones, y en el SQL de Access. En la primera línea se muestra la función para convertir a mayúsculas y en la segunda, la función para convertir a minúsculas.

- En SQL-99 y en la mayor parte de implementaciones de SQL:

```
upper( tira )
lower( tira )
```

- En Access:

```
ucase( tira )
lcase( tira )
```

3.7 Funciones matemáticas

Los distintos estándares e implementaciones de SQL suelen incluir una larga lista de funciones matemáticas cuyo funcionamiento resulta más bien obvio. En Internet y en los manuales de referencia se puede encontrar más información sobre este tipo de funciones.

3.7.1 Función round

La función **round** permite realizar operaciones de redondeo con la precisión deseada. Resulta muy útil en el trabajo con euros, pues habitualmente tras una operación aritmética se deben redondear los precios o importes dejando sólo dos decimales significativos. Tiene dos parámetros: el primero es el valor que se desea redondear; el segundo, el número de decimales que se deben dejar. El siguiente ejemplo rendondea al segundo decimal la operación indicada:

```
round( valor, 2 )
```

Su uso es prácticamente idéntico en todas las implementaciones de SQL (Oracle, Access, etc.). Hay que decir que esta operación de redondeo sólo está disponible a partir de Access 2000. En las versiones anteriores (p.e. Access 97) no se dispone de ella, aunque sí es posible implementarla como código VBA a partir de otras funciones existentes.

♦ **Ejercicio**: Mostrar el código de artículo, la cantidad, el precio unitario y el importe una vez aplicado el descuento de cada una de las líneas de las facturas 325. Redondear el importe para dejar sólo dos decimales. Recuérdese que el descuento nulo se considera como cero.
Solución:

```
select codart, cant,
       round( cant * precio *
              ( 1.0 - coalesce( dto, 0 ) / 100.0 ), 2 )
from   lineas_fac
where  codfac = 325 ;
```

3.7.2 Otras funciones

Como se ha comentado, el estándar y la mayor parte de implementaciones de SQL incluyen una larguísima lista de funciones matemáticas. Las más habituales son: **abs**, **mod**, **sign**, **floor**, etc. cuyo significado es obvio.

3.8 Alias de columnas

En una sentencia **select** el lenguaje SQL devuelve el nombre de la columna o el texto de la expresión como cabecera del resultado obtenido. Si la expresión es muy larga, a veces es truncada, con lo que la cabecera queda poco inteligible.

Los alias de columnas son un mecanismo muy sencillo que permite cambiar el nombre asignado a una columna o expresión en el resultado de una sentencia **select**. Si en la cláusula **select** una palabra sigue a un nombre de columna o expresión, en la cabecera del resultado aparecerá dicha palabra en lugar de la columna o expresión, mejorando de esta forma notablemente la legibilidad.

A veces los alias de columnas deben definirse separando la columna del alias con la palabra inglesa **as** (adverbio de comparación "como", en castellano).

A continuación se muestra una sentencia que utiliza un alias para el importe de las líneas de facturas.

```
select codfac, linea, cant * precio importe
from   lineas_fac ;
```

CODFAC	LINEA	IMPORTE
100	1	100,05
...	...	...

3.9 Pruebas de funciones y expresiones

La mayor parte de sistemas permiten probar el resultado de una expresión o el comportamiento de una expresión con datos sencillos antes de escribir una sentencia muy compleja.

- Algunos sistemas como MySQL, PostgreSQL y Access permiten emplear la cláusula **select** sin cláusula **from**. En dicho caso no se emplea ninguna tabla y, por tanto, no pueden usarse columnas en las expresiones, pero sí datos inmediatos. Por ejemplo:

```
select round( 300.199, 2 ) ;
select substring( 'mapas' tira from 2 for 2 ) ;
```

- Oracle no permite saltar la cláusula from, pero ofrece una minitabla con una fila y una columna que permite trabajar con datos inmediatos. Por ejemplo:

```
select round( 300.199, 2 ) from dual;
select substr( 'mapas', 2, 2 ) from dual ;
```

♦ **Ejercicio**: Escribir una consulta que muestre el nombre de los clientes cuyo código postal empieza por "02", "11" o "21".
Ayuda: Uso del operador *in* y de la función *substr*.
Solución:

```
select nombre
from   clientes
where  substr( codpostal, 1, 2 ) in ( '02', '11', '21' ) ;
```

3.10 Ejercicios

♦ **Ejercicio 3.1**: Código, fecha y descuento de las facturas cuyos códigos se encuentren entre 100 y 150. Visualizar el valor -100 en la columna descuento si éste es nulo.
Solución:

```
select codfac, fecha, coalesce( dto, -100 )
from   facturas
where  codfac between 100 and 150 ;
```

- **Ejercicio 3.2**: Código y fecha de aquellas facturas del año pasado cuyo iva es nulo.
  Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select codfac, fecha
from   facturas
where  iva is null
and    to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

- **Ejercicio 3.3**: Mostrar el código y la fecha de las facturas del último trimestre del año pasado para aquellos clientes cuyo código se encuentra entre 50 y 100.
  Ayuda: Uso del formato de fecha *q*.
  Solución:

```
select codfac, fecha
from   facturas
where  codcli between 50 and 100
and    to_char( fecha, 'q' ) = '4'
and    to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

- **Ejercicio 3.4**: Mostrar el código, descripción, precio original y precio de promoción de los artículos cuyo código comienza por la letra 'A'. El precio de promoción se calcula de la siguiente forma: Si el stock supera en más de un 100 % al stock mínimo, entonces se aplica un 20 %; en caso contrario no se aplica ningún descuento.
  Solución:

```
select codart, descrip, precio,
       precio * case when stock > 2 * stock_min then 0.8
                                                else 1
                end
from   articulos
where  codart like 'A%' ;
```

- **Ejercicio 3.5**: Códigos de los clientes, sin que salgan repetidos, a los que se les ha hecho alguna factura sin iva (iva cero o nulo).
  Solución:

```
select distinct codcli
from   facturas
where  coalesce( iva, 0 ) = 0 ;
```

- **Ejercicio 3.6**: Código y nombre de aquellos clientes cuyo nombre termina con la tira "ez".
  Solución: Se usa la sintaxis de Oracle.

```
select codcli, nombre
from   clientes
where  substr( nombre, length( nombre )-1, 2 ) = 'ez' ;
```

- **Ejercicio 3.7**: Códigos de artículos que no contienen caracteres alfabéticos.
  Solución: Un carácter alfabético es distinto en mayúsculas y en minúsculas. Por tanto, un carácter no alfabético será igual en mayúsculas que en minúsculas. La siguiente solución no funciona en Access pues éste sistema por defecto no distingue al comparar mayúsculas y minúsculas.

```
select codart, descrip
from   articulos
where  upper( codart ) = lower( codart );
```

- **Ejercicio 3.8**: Escribir una consulta que muestre el nombre y la dirección de aquellos clientes cuyo código postal empieza por "11". Además, debe aparecer la palabra "preferente" en aquellos clientes cuyo tercer dígito del código postal es mayor o igual que 5.
  Ayuda: Uso del operador *like* y de las funciones *decode*, *sign* y *to_number*. La solución consiste en extraer el tercer carácter, convertirlo a número, restarle una cantidad y, según el signo del resultado, mostrar un texto u otro.
  Solución:

```
select nombre, direccion,
      decode( sign( to_number( substr( codpostal, 3, 1 ) )-5
),
               -1, ' ','preferente' )
from  clientes
where codpostal like '11%' ;
```

- **Ejercicio 3.9**: Se desea promocionar los artículos de los que se posee un stock grande. Si el artículo es de más de 30 euros y el stock supera los 300 euros, se hará un descuento del 10%. Si el artículo es de 30 euros o menos y el stock supera los 150 euros, se hará un descuento del 15%. Mostrar un listado de los artículos que van a entrar en la promoción, con su código de artículo, precio actual y su precio en la promoción.
  Ayuda: Uso de las funciones *decode* y *sign*. El precio del artículo sirve para discernir entre un caso y otro.
  Solución:

```
select codart, precio,
       round( precio *
              decode( sign( precio - 30 ), 1, 0.9, 0.85 ), 2
)
from   articulos
where  ( precio > 30 and stock * precio > 300 )
or     ( precio <= 30 and stock * precio > 150 ) ;
```

3.11 Autoevaluación

- **Ejercicio 1**: Pueblos de la provincia de Castellón cuya primera y última letra coinciden.
- **Ejercicio 2**: Se desea hacer una promoción especial de los artículos más caros (aquéllos cuyo precio supera los 10 euros). Mostrar el código, descripción, precio original y precio de promoción de los artículos. El precio de promoción se calcula de la siguiente forma: Si el precio es menor de 20 euros, se aplica un 10 % de descuento en la promoción. Si es menor de 30 euros, se aplica un 20 %. Si es menor de 40 euros se aplica un 30 %. Si supera los 40 euros, se aplica un 40 %.
- **Ejercicio 3**: Código, fecha y código de cliente de las facturas de los diez primeros días del mes de febrero del año pasado.

4 FUNCIONES DE COLUMNA

Este capítulo presenta las funciones de columna del lenguaje SQL. Son funciones que resumen toda una columna en un único valor, de ahí su nombre.

En los ejemplos siguientes se emplearán obviamente las diversas tablas descritas en el primer capítulo. Para ilustrar algunos ejemplos de forma más concreta se considerará la tabla de artículos rellenada de la siguiente forma:

Tabla ARTICULOS

CODART	PRECIO	STOCK
A1	1	150
A2	2	
A3	2	30
A4		250
A5	3	10

4.1 Funciones Escalares frente a Funciones de Columna

En principio, una sentencia **select** (sin función de columna) devuelve una línea por cada fila de la tabla base. Por ejemplo:

```
select precio
from   articulos;
```

La sentencia anterior devuelve como resultado una línea con el precio por cada artículo existente en la tabla de artículos. En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado son los valores 1, 2, 2, nulo y 3.

En cambio, una sentencia **select** con al menos una función de columna devuelve una única línea para toda la tabla: Es decir, resume toda la tabla en un único valor. Por ejemplo:

```
select avg( precio )
from   articulos;
```

La sentencia anterior devuelve como resultado una única línea con un valor: la media del precio de los artículos. Es decir, resume todos los descuentos en un único valor. En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 2 euros (dado que se ignoran los valores nulos).

Hay que distinguir entre funciones normales (o funciones escalares) y funciones de columna. Una función normal (**substr**, **round**, etc.) devuelve un valor modificado por cada valor de la tabla original. Por ejemplo:

```
select sqrt( precio )
from   articulos;
```

La sentencia anterior devuelve como resultado tantas filas como artículos existen y en cada fila muestra la raíz cuadrada del precio del artículo. En cambio, una función de columna devuelve un único valor para toda la tabla, como puede ser el precio medio, el precio máximo, el precio mínimo, etc.

4.2 Funciones de Columna habituales

Las funciones de columna que suele incluir SQL son las siguientes:

```
avg( [distinct] valor )
count( [distinct] valor | * )
sum( [distinct] valor )
max( [distinct] valor )
min( [distinct] valor )
stddev( [distinct] valor )
variance( [distinct] valor )
```

La función de cada una de estas funciones es obvio. La primera calcula la media de los distintos valores. La segunda cuenta el número de filas o valores. La tercera suma los valores. La cuarta calcula el máximo. La quinta calcula el mínimo. La sexta calcula la desviación estándar. La última calcula la varianza.

La función **count** es la que admite una mayor variedad. Admite como argumento tanto un * como un valor. En el primer caso cuenta filas y en el segundo, valores no nulos.

En caso de presentar el modificador **distinct**, en primer lugar se eliminan los valores repetidos y, después, se aplica la función de grupo.

Algunos sistemas no incluyen todas las funciones de columna anteriormente enumeradas. En tales casos, por lo menos sí suelen incluir las cuatro primeras.

Dentro de una sentencia se puede emplear más de una función de columna.

Además, **valor** puede ser tanto una columna de la tabla como una expresión de una o más columnas.

En cuanto a los tipos de los datos retornados, la función **count** devuelve siempre un valor entero. Las funciones **max**, **min** y **sum** devuelven un valor del mismo tipo que su argumento. El resto de funciones (**avg**, **stddev** y **variance**) devuelven un valor real.

Estas funciones no pueden aparecer en la cláusula **where**, pues en dicha cláusula se procesa un único valor.

- **Ejercicio**: Escribir una sentencia que obtenga el precio más caro y el más barato de todos los artículos.
  Solución:
  ```
  select max( precio ), min( precio )
  from   articulos;
  ```
  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es de 3 euros para el precio máximo y de 1 euro para el precio mínimo.

- **Ejercicio**: Escribir una sentencia que obtenga el stock más alto y el más bajo, considerando el stock nulo como cero.
  Solución:
  ```
  select max( stock ), min( coalesce( stock, 0 ) )
  from   articulos ;
  ```
  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es de 250 unidades para el stock máximo y de 0 unidades para el stock mínimo.

4.3 Funciones de Columna con restricciones

Cuando existen restricciones (en el **where**), la función de columna se aplica a las filas resultantes de la criba.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?
  ```
  select avg( precio )
  from   articulos
  where  stock <= 30 ;
  ```

Solución: Calcula el precio medio de aquellos artículos cuyo stock no supera las 30 unidades.
Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es de 2,5 euros.

- **Ejercicio**: Escribir una sentencia que obtenga el stock más alto para aquellos artículos cuyo precio no supera los 2 euros.
  Solución:

```
select max( stock )
from   facturas
where  precio <= 2.00 ;
```

  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es de 150 unidades.

4.4 Funciones de Columna con valores nulos

Es muy importante conocer el comportamiento de las funciones de columna frente a los valores nulos. Éste es muy simple: Todas las funciones excepto **count(*)** ignoran los nulos.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select count( * )
from   articulos;
```

  Solución: Cuenta el número total de artículos.
  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 5.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select count( precio )
from   articulos;
```

  Solución: Devuelve el número de artículos con precio no nulo. Como la función de columna no es **count(*)**, los artículos con precio nulo son ignorados.
  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 4.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select count( distinct precio )
from   articulos;
```

  Solución: Devuelve el número de precios distintos de los artículos.
  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 3.

- **Ejercicio**: Escribe una sentencia que devuelva el número de artículos con precio.
  Solución:

```
select count( precio )
from   articulos;
```

  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 4.

- **Ejercicio**: Escribe una sentencia que calcule el stock medio de los artículos.
  Solución:

```
select avg( stock )
from   articulos;
```

  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 110.

- **Ejercicio**: Escribe una sentencia que calcule el stock medio de los artículos, pero considerando los nulos como ceros.
  Solución: En principio la función **avg** ignora los valores nulos. Para evitarlo hay que convertir dichos valores al valor cero antes de calcular la media.

```
select avg( coalesce( stock, 0 ) )
from   articulos;
```

  Otra posible solución consiste en calcular la suma de todos los precios (esta operación ignorará los valores nulos) y dividir por el total de artículos.

```
select sum( stock ) / count( * )
from   articulos;
```

  Resultado: En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado es 88.

4.5 Errores habituales

En este último apartado se van a comentar ciertos errores habituales entre los principiantes.

- **Caso 1**: ¿Cuál es la diferencia entre las siguientes dos sentencias?

```
select count( distinct precio )
from   articulos;

select distinct count( precio )
from   articulos;
```

  Solución: La primera sentencia calcula el número de precios distintos de los artículos. En cambio, la segunda calcula el número de artículos cuyos precios no son nulos y, una vez calculado este único valor, elimina los valores repetidos del resultado anterior obtenido. Como la función **count** devuelve un único valor, obviamente no puede haber repetidos y el modificador **distinct** no hace nada en este último caso. Así pues, la segunda sentencia calcula el número de artículos con precios no nulos. En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado de la primera sentencia es 3 y el de la segunda es 4.

- **Caso 2**: ¿Qué calcula la siguiente sentencia?

```
select count( distinct codart )
from   articulos;
```

  Solución: La anterior sentencia calcula el número de códigos de artículos distintos que existen en la tabla artículos. Como la columna **codart** es la clave primaria en la tabla de artículos, no pueden existir códigos de artículos repetidos, por lo que el operador **distinct** no puede eliminar ningún valor repetido y resulta más que inútil, contraproducente desde el punto de vista de las prestaciones. En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado de la sentencia es 5.

- **Caso 3**: ¿Cuál es la diferencia entre las siguientes dos sentencias?

```
select count( codart )
from   articulos;

select count( * )
from   articulos;
```

  Solución: La primera sentencia cuenta el número de códigos de artículos no nulos en la tabla de artículos. Ahora bien, ¿pueden existir códigos nulos? Efectivamente no si se cumple la regla de integridad de entidades, porque **codart** es la clave primaria. Así pues, lo que hace es contar el número de filas de la tabla de artículos. La segunda sentencia realiza la misma labor (número de filas de la tabla de artículos). La única diferencia entre ambas suele ser el tiempo de ejecución. La mayor parte de los

sistemas de gestión de bases de datos ejecutan mucho más eficientemente la segunda que la primera. En la tabla de artículos con el contenido mostrado al principio del capítulo, el resultado de ambas sentencias es 5.

- **Caso 4**: ¿Qué calcula la siguiente sentencia?

```
select precio
from   articulos
where  max( precio ) = 3 ;
```

Solución: La anterior sentencia no calcula nada pues es errónea. La restricción en el **where** se aplica siempre a cada fila: si la expresión da cierto, la fila se queda en el resultado; si la expresión da falso, la fila es ignorada. Ahora bien, la expresión existente (**max(precio)**) no puede aplicarse a cada fila pues es una operación que se aplica a toda una columna.
De este caso se puede extraer una regla general muy importante: Las funciones de columna no pueden aparecer nunca en la cláusula **where**.

4.6 Ejercicios

- **Ejercicio 4.1**: Escribir una sentencia que calcule el número de artículos cuyo stock es nulo.
Ayuda: Uso de la función de columna *count*.
Solución:

```
select count( * )
from   articulos
where  stock is null ;
```

- **Ejercicio 4.2**: Escribir una sentencia que calcule el importe de la facturación (suma del producto de la cantidad por el precio de las líneas de factura) de aquellos artículos cuyo código contiene la letra "A" (bien mayúscula o minúscula).
Ayuda: Uso de la función de columna *sum* sobre una expresión. No es necesario agrupar dado que se pide un solo resultado.
Solución:

```
select round( sum( cant * precio ), 2 )
from   lineas_fac
where  upper( codart ) like '%A%' ;
```

- **Ejercicio 4.3**: Escribir una sentencia que calcule el número de clientes sin código postal. No se debe emplear la cláusula **where**.
Ayuda: Calcular el número total de clientes y restar el número de clientes que sí tienen código postal.
Solución:

```
select count( * ) - count( codpostal )
from   clientes;
```

- **Ejercicio 4.4**: Escribir una sola sentencia que calcule el número de facturas con iva 16, el número de facturas con iva 7 y el número de facturas con otros ivas.
Ayuda: El número de facturas con iva 16 se puede calcular sumando el resultado de convertir los ivas 16 en unos y el resto en ceros usando la expresión **case**. El resto de información se puede calcular de forma análoga.
Solución:

```
select sum( case iva when 16 then 1 else 0 end ),
       sum( case iva when  7 then 1 else 0 end ),
       sum( case iva when 16 then 0 when 7 then 0
```

```
                                        else 1 end )
from   facturas;
```

♦ **Ejercicio 4.5**: Escribir una sentencia que calcule el porcentaje (%) de facturas que no tienen iva 16.
Ayuda: Este porcentaje se puede calcular restando del número total de facturas aquellas que sí tienen iva 16. También podría calcularse directamente las que no tienen iva 16, pero en dicho caso habría que ir con cuidado con el valor nulo. Se emplea 100.0 para que el resultado sea un número real.
Solución:

```
select 100.0 * ( count(*) -
                 sum( case iva when 16 then 1 else 0 end ) )
       / count(*)
from   facturas;
```

♦ **Ejercicio 4.6**: Escribir una sentencia que calcule el número medio mensual de facturas realizadas durante el año pasado por el vendedor con código 400.
Ayuda: El número medio mensual se calcula a partir del número anual de facturas dividido entre 12 meses. Si se emplea 12 como divisor, entonces el resultado será un entero (dado que el dividendo y el divisor son ambos enteros) y pueden perderse decimales. En cambio, si se usa 12.0 como divisor, el resultado será un número real con varios decimales (cuyo número podría limitarse con la función **round**).
Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select count( * ) / 12.0
from   facturas
where  codven = 400
and    to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

♦ **Ejercicio 4.7**: Escribir una sentencia que calcule el número de vendedores que han realizado al menos una factura durante el año pasado.
Ayuda: Es conveniente eliminar las filas repetidas.
Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select count( distinct codven )
from   facturas
where  to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

4.7 Autoevaluación

♦ **Ejercicio 1**: Escribir una sentencia que calcule el número máximo de líneas en una factura.

♦ **Ejercicio 2**: Escribir una sentencia que calcule el número de facturas sin descuento (cero o nulo); con descuento moderado (<= 10) y con descuento elevado (> 10).

♦ **Ejercicio 3**: Escribir una sentencia que calcule el número medio de unidades vendidas por factura.

5 AGRUPACIÓN

Este capítulo presenta la agrupación de filas del lenguaje SQL. Es una técnica que permite reunir las filas de una tabla en uno o varios grupos y extraer un valor por cada grupo. Es decir, la tabla se divide en varios grupos, cada uno de los cuales debe tener alguna característica común, y se realizan los cálculos deseados de resumen para cada grupo.

En los ejemplos siguientes se emplearán obviamente las diversas tablas descritas en el primer capítulo. Para ilustrar algunos ejemplos de forma más concreta se considerará las tablas rellenadas de la siguiente forma:

Tabla PUEBLOS

CODPUE	NOMBRE	CODPRO
101	NULES	12
102	ONDA	12
103	SARRION	44
104	LIRIA	46
105	GANDIA	46

Tabla FACTURAS

CODFAC	FECHA	CODCLI	IVA	DTO
1	14-1-05	100	16	5
2	14-2-05	100	16	0
3	14-1-06	100	7	5
4	14-1-07	101	16	0
5	15-4-08	102		0
6	15-4-08	102	7	5

5.1 Motivación. Introducción

Una sentencia **select** sin función de columna devuelve una línea por cada fila de la tabla. Por ejemplo, la siguiente sentencia devuelve tantos códigos de pueblos como filas tiene la tabla pueblos.

```
select codpue
from   pueblos;
```

Una sentencia **select** con al menos una función de columna devuelve una única línea para toda la tabla. Por ejemplo, la siguiente sentencia cuenta el número total de pueblos. En la tabla de pueblos con el contenido mostrado al principio del capítulo, el resultado es 5.

```
select count( * )
from   pueblos;
```

Ahora bien, si se quiere calcular el número de pueblos en la provincia de Castellón habría que escribir la siguiente sentencia:

```
select count( * )
from   pueblos
where  codpro = '12' ;
```

En la tabla de pueblos con el contenido mostrado al principio del capítulo, el resultado es 2.

¿Pero qué pasa si queremos saber el número de pueblos de cada provincia? Habría que escribir la anterior sentencia con cada código de provincia distinto. Además de costoso y tedioso (habría que escribir 52 consultas), este método es muy dado a errores pues es muy fácil teclear incorrectamente un código u olvidarse de algún código. Más aún, si en lugar de 52 grupos existieran 3000 esta solución sería completamente inviable.

Por tanto, si se quiere calcular el número de pueblos para cada provincia no queda más remedio que recurrir a la agrupación. Se agrupa la tabla de pueblos por provincias y para cada grupo (cada provincia) se cuenta el número de pueblos. La sentencia que calcula el resultado deseado es la siguiente:

```
select codpro, count( * )
from   pueblos
group  by codpro;
```

La agrupación se indica mediante la cláusula **group by**. Una vez agrupada una tabla, por cada grupo sólo se puede mostrar una línea en pantalla (o, lo que es lo mismo, una fila en el resultado final).

La anterior sentencia agruparía los pueblos del principio en grupos según su código de provincia, es decir, se crearían tres grupos como se puede ver en la figura siguiente:

CODPUE	NOMBRE	CODPRO
101	NULES	12
102	ONDA	12
103	SARRION	44
104	LIRIA	46
105	GANDIA	46

Una vez creados los grupos, para cada grupo se muestra lo indicado en la cláusula **select**, es decir, el código de provincia y el número de filas dentro del grupo. Por tanto, el resultado es el siguiente:

CODPRO	COUNT(*)
12	2
44	1
46	2

- **Ejercicio**: Mostrar el menor código de pueblo para cada provincia.
  Solución:

  ```
  select codpro, min( codpue )
  from   pueblos
  group  by codpro ;
  ```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado se muestra en la siguiente tabla:

CODPRO	MIN(CODPUE)
12	101
44	103
46	104

♦ **Ejercicio**: Para cada provincia calcular la longitud máxima de los nombres de sus pueblos.
Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select codpro, max( length( nombre ) )
from   pueblos
group  by codpro ;
```

Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado se muestra en la siguiente tabla:

CODPRO	MAX(LENGTH(NOMBRE))
12	5
44	7
46	6

5.2 Funciones de grupo

Las funciones de grupo son exactamente las mismas que las de columna, sólo que las primeras se aplican a cada grupo mientras que las segundas se aplican a toda la tabla. A veces ni siquiera se distingue entre unas y otras. En este texto sí se diferenciará entre ellas para saber cuándo se está agrupando una tabla y cuándo no se agrupa. Las funciones de grupo son las siguientes:

```
avg( [distinct] valor )
count( [distinct] valor | * )
sum( [distinct] valor )
max( [distinct] valor )
min( [distinct] valor )
stddev( [distinct] valor )
variance( [distinct] valor )
```

5.3 Tipos de agrupaciones

Un **grupo** es un conjunto de filas (una o más filas) con una misma característica común. Dicha característica puede ser una columna, un conjunto de columnas o incluso una expresión de una o más columnas. Véanse los ejemplos siguientes:

- Un ejemplo de agrupación por una columna es la agrupación de los clientes por su código de pueblo.

```
group by codpue
```

- Un ejemplo de agrupación por varias columnas es la agrupación de las facturas por cliente e iva aplicado. De dicha forma, cada grupo contendrá todas las facturas de un mismo cliente con un mismo iva. Los valores nulos se consideran como un valor más.

```
group by codcli, iva
```

- Un ejemplo de agrupación por una parte de una expresión de una o más columnas es la agrupación de las facturas por el año.

```
group by to_char( fecha, 'yyyy' )
```

La agrupación es mucho más costosa que las funciones de columna principalmente por dos motivos: El primero es que hay que calcular las funciones varias veces (tantas como grupos existan). El segundo es que para poder crear los grupos es necesario ordenar las filas o aplicar una función de dispersión a las filas.

- **Ejercicio**: Mostrar el número de pueblos para cada posible longitud del nombre (es decir, número de pueblos cuyo nombre contiene 4 caracteres, número de pueblos cuyo nombre contiene 5 caracteres, etc.)
  Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select length( nombre ), count( * )
from   pueblos
group  by length( nombre ) ;
```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado es: (4, 1; 5, 2; 6, 1; 7, 1).

- **Ejercicio**: Mostrar el número de facturas para cada año.
  Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select to_char( fecha, 'yyyy' ), count( * )
from   facturas
group  by to_char( fecha, 'yyyy' ) ;
```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado es: (05, 2; 06, 1; 07, 1; 08, 2).

5.4 Agrupaciones por múltiples factores

Como ya se comentado, una tabla puede agruparse en función de varias columnas o de una expresión de varias columnas. Ambos casos son muy parecidos a la agrupación por un único factor. Se construyen grupos según los factores de agrupación de tal forma que a un mismo grupo van a parar todas aquellas filas con los mismos valores en los factores de agrupación.

Vamos a verlo con un ejemplo. Se desea obtener el número de facturas para cada cliente y tipo de iva distinto. Con este enunciado, las filas de la tabla de facturas habría que agruparla por código de cliente e iva, quedando de la siguiente forma:

Tabla FACTURAS

CODFAC	FECHA	CODCLI	IVA	DTO
1	15-1-05	100	16	5
2	15-2-05	100	16	0
3	15-1-06	100	7	5
4	15-1-07	101	16	0
5	15-4-08	102		0
6	15-4-08	102	7	5

Como se puede ver, los valores nulos se consideran como un valor más, distinto de los demás. Contando el número de filas dentro de cada grupo, el resultado final sería el mostrado en la siguiente tabla:

CODCLI	IVA	COUNT(*)
100	7	1
100	16	2
101	16	1
102		1
102	7	1

La sentencia que realiza el anterior enunciado es la siguiente:

```
select codcli, iva, count( * )
from   facturas
group  by codcli, iva ;
```

- **Ejercicio**: Mostrar el número de facturas para cada cliente y año.
  Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

  ```
  select codcli, to_char( fecha, 'yyyy' ), count( * )
  from   facturas
  group  by codcli, to_char( fecha, 'yyyy' ) ;
  ```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado es: 100, 05, 2; 100, 06, 1; 101, 07, 1; 102, 08, 2.

5.5 Agrupaciones incorrectas

- **Ejercicio**: Escribir una sentencia que calcule la media del descuento de las facturas.
  Solución: En la tabla con el contenido mostrado al principio del capítulo, el resultado es 2,5.

  ```
  select avg( dto )
  from   facturas ;
  ```

- **Ejercicio**: Escribir una sentencia que calcule la media del descuento de las facturas para cada cliente.
  Solución: En la tabla con el contenido mostrado al principio del capítulo, el resultado se muestra en la siguiente tabla. Tras la tabla se muestra la sentencia en SQL.

  ```
  select codcli, avg( dto )
  from   facturas
  group  by codcli ;
  ```

  Resultado: En la tabla con el contenido mostrado al principio del capítulo, el resultado es el siguiente:

CODCLI	AVG(DTO)
100	3,33
101	0
102	2,5

Si no se ha entendido la anterior consulta, lo mejor es mostrar la tabla inicial agrupada por código de cliente y calcular para cada grupo la función de agrupación (**avg(dto)**).

CODFAC	CODCLI	IVA	DTO
1	100	16	5
2	100	16	0
3	100	7	5
4	101	16	0
5	102		0
6	102	7	5

Es importante destacar que para cada grupo, la consulta devuelve una sola fila. Así pues, ¿qué realiza la siguiente sentencia?

```
select codcli, avg( dto ), iva
from   facturas
group  by codcli ;
```

La solución es nada pues la sentencia es incorrecta dado que para cada cliente no existe un único iva sino varios (en algunas ocasiones pueden existir miles de resultados), los cuales no caben en una única línea. Este es un ejemplo de consulta incorrecta que devuelve muchos datos para cada grupo. En este caso para cada grupo únicamente puede devolverse la característica común del grupo (**codcli**) o una operación de agrupación (de resumen) del grupo, como por ejemplo la media del descuento, el número de filas, el iva máximo, etc.

5.6 Restricciones de fila y restricciones de grupo

Una consulta con agrupación admite dos tipos de restricciones: restricciones de filas y restricciones de grupo.

- Las restricciones de fila van en la cláusula **where**. Como ya hemos visto en capítulos anteriores, se aplican a cada fila. Si la expresión da cierto, la fila es procesada; si da falso o nulo, es ignorada.
- Las restricciones de grupo van en la cláusula **having**, la cual, de existir, va siempre tras la cláusula **group by**. Este tipo de restricciones tiene un funcionamiento análogo al de las restricciones de fila, pero aplicados al grupo. Una vez se ha realizado la agrupación, si esta restricción de grupo da cierto, el grupo se queda; en caso contrario, el grupo es descartado.

♦ **Ejercicio**: ¿Qué realiza la siguiente consulta?

```
select codcli, avg( dto )
from   facturas
where  codfac > 100
group  by codcli
having avg( dto ) > 4;
```

Solución: Calcula el descuento medio aplicado en las facturas cuyo código es mayor que 100 para cada cliente pero sólo si dicho descuento supera el 4 %.

Algunas restricciones deben ir obligatoriamente en el **where**, otras restricciones deben hacerlo en el **having** y unas pocas pueden ir en ambos sitios. ¿Cómo se distinguen? Muy fácil, viendo si la restricción debe aplicarse a cada fila o a cada grupo de la consulta. Si la restricción debe aplicarse a cada fila, entonces debe ir en el **where**. Si debe aplicarse a cada grupo, entonces debe ir en el **having**.

Ejercicios de restricciones adicionales a la consulta anterior:

- Procesar sólo facturas del año pasado: Restricción de fila.
- Procesar clientes con más de 10 facturas: Restricción de grupo.
- Procesar facturas cuyo código sea menor que 100: Restricción de fila.
- Procesar clientes cuyo iva máximo sea 16: Restricción de grupo.
- Procesar facturas con iva 16: Restricción de fila.

Unas pocas restricciones pueden ir tanto en el **where** como en el **having**. Un ejemplo de restricción que puede ir en ambos casos surge en la consulta para calcular el descuento medio para cada cliente cuyo código sea menor que 10. La restricción **codcli < 10** puede ir tanto en el **where** como en el **having**. Las siguientes sentencias realizan ambas el cálculo propuesto. La primera incluye la restricción en el **where**; la segunda, en el **having**.

```
select codcli, avg( dto )
from   facturas
where  codcli < 10
group  by codcli ;

select codcli, avg( dto )
from   facturas
```

```
group  by codcli
having codcli < 10 ;
```

- **Ejercicio**: ¿Dónde es mejor poner las restricciones que pueden ir en ambos sitios: en el **where** o en el **having**?
  Solución: Suele ser muy conveniente que la restricción vaya en el **where** pues el procesamiento de la consulta será habitualmente más rápido. ¿A qué se debe? A que si la restricción se aplica en el where el sistema rápidamente se quita de encima unas cuantas filas. En caso contrario, debe procesar todas las filas, crear todos los grupos y sólo finalmente descartar algunos grupos.

- **Ejercicio**: Para cada artículo, mostrar el descuento máximo (considerando el descuento nulo como cero) aplicado en sus facturas y el número de unidades vendidas. Considérense sólo las líneas de las primeras 100 facturas. Sólo se deben mostrar aquellos artículos cuyo número de unidades vendidas supera el centenar y cuyo código comienza por la letra 'A'.
  Solución:

```
select codart, max( coalesce( dto, 0 ) ), sum( cant )
from   lineas_fac
where  codfac between 1 and 100
and    codart like 'A%'
group  by codart
having sum( cant ) > 100 ;
```

5.7 Ejecución de una consulta con agrupación

La ejecución de una consulta con agrupación se realiza siempre de la siguiente forma:

1. Se determina la tabla origen de los datos a partir de cláusula **from**.
2. Se aplican las restricciones de fila, si las hay (cláusula **where**). De esta forma, se eliminan todas aquellas filas que no cumplen las restricciones solicitadas.
3. Se agrupan las filas restantes en grupos según la cláusula **group by**. Todos las filas de un mismo grupo deben tener los mismos valores en la expresión o expresiones que incluyen la cláusula **group by**. Este paso es de los computacionalmente más costosos.
4. Se aplican las restricciones de grupo, si las hay (cláusula **having**). De esta forma, se eliminan todos aquellos grupos que no cumplen las restricciones de grupo solicitadas.
5. Se devuelve una fila por cada grupo (cláusula **select**).

En el ejemplo siguiente se va mostrar los pasos que se siguen para ejecutar la consulta siguiente:

```
select codcli, avg( dto )
from   facturas
where  codfac > 100
group  by codcli
having avg( dto ) > 4;
```

Los pasos son los siguientes:

1. Se toma la tabla **facturas**.
2. Se dejan sólo las filas cuyo código es mayor que 100.
3. Se agrupan las filas restantes en grupos según el código del cliente. Es decir, se agrupan las facturas resultantes por código de cliente.
4. Se aplican las restricciones de grupo, es decir, se dejan aquellos clientes cuyo descuento medio supera el 4.
5. Para cada grupo se muestra el código del cliente y su descuento medio.

5.8 Combinación de Funciones de Grupo y Funciones de Columna

Es posible utilizar simultáneamente una función de columna y una función de agrupación. En este caso, la función de agrupación se aplica a cada grupo devolviendo un valor para cada grupo y la función de columna se aplica al resultado anterior para devolver un único resultado.

- **Ejercicio**: Escribir una sentencia que obtenga el máximo de los descuentos medios aplicados a los clientes en sus facturas.
  Solución: Hay que calcular para cada cliente el descuento medio con una función de agrupación y a partir de dichos descuentos obtener el máximo con una función de columna.

```
select max( avg( dto ) )
from   facturas
group  by codcli ;
```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado es 3,33.

- **Ejercicio**: Escribir una sentencia que obtenga el número máximo de pueblos de una provincia.
  Solución: Hay que calcular para cada provincia el número de pueblos con una función de agrupación y a partir de dichos datos obtener el máximo con una función de columna.

```
select max( count( * ) )
from   pueblos
group  by codpro ;
```

  Resultado: Con los datos de la tabla mostrada al principio del capítulo, el resultado es 2.

- **Ejercicio**: Obtener el número máximo de unidades vendidas de un artículo.
  Solución: A partir de la tabla de líneas de facturas hay que calcular el número de unidades vendidas para cada artículo con una función de agrupación y a partir de dichos valores obtener el máximo con una función de columna.

```
select max( sum( cant ) )
from   lineas_fac
group  by codart ;
```

5.9 Reglas Nemotécnicas

Para trabajar con la agrupación es conveniente siempre entender muy bien el problema y aplicar tres reglas nemotécnicas de fácil memorización:

1. **Regla de Oro**: Todo lo que aparece en el **select** y en el **having**, o son funciones de grupo o están en el **group by**.

```
select codcli, avg( dto ), iva
from   facturas
group  by codcli ;
```

   En la anterior sentencia **codcli** está en el **group by**, **avg(dto)** es una función de grupo, pero **iva** no está en el **group by** ni es función de grupo.
2. **Regla de Plata**: Las funciones de columna y grupo no pueden aparecer en el **where**.
3. **Regla de Bronce**: Normalmente no hace falta agrupar cuando se debe devolver un solo dato.
4. **Cuarta Regla**: Normalmente no hace falta agrupar si después (en las cláusulas **having** o **select**) no se utilizan funciones de grupo.

5.10 Ejercicios

- **Ejercicio 5.1**: Escribir una consulta que obtenga el máximo descuento aplicado cada año, considerando el descuento nulo como cero.
  Ayuda: Agrupación por una parte de la fecha.
  Solución:

```
select to_char( fecha, 'yyyy' ), max( coalesce( dto, 0 ) )
from   facturas
group  by to_char( fecha, 'yyyy' ) ;
```

- **Ejercicio 5.2**: Escribir una consulta que obtenga el máximo importe de la facturación de un artículo.
  Ayuda: Se debe combinar una función de grupo y una función de columna. La primera calculará la facturación de cada artículo y la segunda, el máximo de dichos valores.
  Solución:

```
select max( sum( cant * precio ) )
from   lineas_fac
group  by codart ;
```

- **Ejercicio 5.3**: Escribir una consulta que calcule el descuento mínimo (considerando el descuento nulo como un cero) realizado en las facturas para cada mes del año pasado.
  Ayuda: Se deben seleccionar las filas del año pasado y agruparlas por mes.
  Solución:

```
select to_char( fecha, 'mm' ), min( coalesce( dto, 0 ) )
from   facturas
where  to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by to_char( fecha, 'mm' ) ;
```

- **Ejercicio 5.4**: Número máximo de facturas realizadas por un vendedor el año pasado.
  Ayuda: Se debe combinar una función de grupo y una función de columna. La primera calculará el número de facturas para cada vendedor y la segunda, el máximo de dichos valores.
  Solución:

```
select max( count( * ) )
from   facturas
where  to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codven ;
```

- **Ejercicio 5.5**: Escribir una consulta que obtenga el código de los pueblos en los que tenemos dos o más clientes.
  Ayuda: Uso de las cláusulas *group by* y *having*. Agrupación de los clientes por pueblos.
  Solución:

```
select codpue
from   clientes
group  by codpue
having count( * ) >= 2 ;
```

- **Ejercicio 5.6**: Escribir una consulta que obtenga el número de facturas para cada uno de los tipos de ivas (normal (iva=16), reducido (iva=7), otros), pero sólo si hay más de 100 facturas.
  Ayuda: Se debe agrupar por una expresión que devuelva un valor para el iva 16, un segundo valor para el iva 7 y un tercer valor para cualquier otro valor del iva. Esta expresión se puede calcular con ayuda de la expresión **case**.

Solución:

```
select case iva when 16 then 1 when 7 then 2 else 3 end,
       count( * )
from   facturas
group  by case iva when 16 then 1 when 7 then 2 else 3 end
having count( * ) > 100 ;
```

- **Ejercicio 5.7**: Escribir una consulta que obtenga el código de aquellos artículos que siempre se han vendido al mismo precio.
  Ayuda: Un artículo se habrá vendido siempre al mismo precio si en las líneas de factura el precio máximo y el mínimo coinciden. Como se puede ver en este caso, se realiza un agrupamiento pero no se emplea ninguna función de grupo.
  Solución:

```
select l.codart
from   lineas_fac l
group  by l.codart
having max( l.precio ) = min( l.precio ) ;
```

- **Ejercicio 5.8**: Escribir una consulta que para cada cliente que ha hecho más de dos facturas durante el año pasado, con el 16% de IVA o sin descuento, muestre su código y el número de facturas realizadas. Se considera que una factura no tiene descuento si éste es cero o nulo.
  Ayuda: Uso de algunas restricciones en el *where* y otras en el *having*. Agrupación de las facturas por código de cliente.
  Solución:

```
select codcli, count( * )
from   facturas
where  to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    ( iva = 16
or       nvl( dto, 0 ) = 0 )
group  by codcli
having count(*) > 2 ;
```

- **Ejercicio 5.9**: Escribir una consulta que de los artículos cuyo código termina con la letra "X" más un dígito numérico, muestre el código y la cantidad total pedida en las líneas de factura.
  Ayuda: Uso de la función de agrupación *sum*. Agrupación de las líneas de facturas por artículos.
  Solución:

```
select codart, sum( cant )
from   lineas_fac
where  upper( substr( codart, length( codart )-1, 1 ) ) = 'X'
and    substr( codart, length( codart ), 1 ) between '0' and
'9'
group  by codart ;
```

5.11 Autoevaluación

- **Ejercicio 1**: Escribir una consulta que obtenga el importe de la factura más alta.
- **Ejercicio 2**: Escribir una consulta que calcule el número de clientes a los que ha realizado facturas cada uno de los vendedores de la empresa.
- **Ejercicio 3**: Escribir una consulta que obtenga el número más alto de clientes que viven en el mismo pueblo.

6 CONCATENACIÓN INTERNA DE TABLAS

Este capítulo describe una operación muy habitual, importante y útil en el procesamiento de información con el lenguaje SQL: el acceso y extracción de información relacionada que se encuentra repartida en varias tablas (más de una).

6.1 Concatenación Interna de dos tablas

La extracción de información que se halla en más de una tabla se realiza mediante el proceso de concatenación de tablas. En primer lugar se va a describir el proceso de concatenar dos tablas y, más adelante, se describirá el mismo proceso para más de dos tablas.

El acceso a información que se encuentra en dos tablas se suele realizar conceptualmente en dos pasos:

1. Generar el producto cartesiano de ambas tablas. Es decir, generar todas las posibles combinaciones de las filas de ambas tablas.
2. Seleccionar las filas que interesan de entre todas las generadas en el paso anterior.

Vamos a verlo con un ejemplo. Supongamos que se desea mostrar el nombre de cada pueblo junto al de la provincia a la que pertenece. Supongamos que tenemos las dos tablas siguientes.

Tabla PUEBLOS

CODPUE	NOMBRE	CODPRO
101	NULES	12
102	ONDA	12
103	SARRION	44

Tabla PROVINCIAS

CODPRO	NOMBRE
12	CASTELLÓN
44	TERUEL
46	VALENCIA

Para conseguir extraer la información deseada, como se ha comentado, hay que realizar dicho proceso en dos pasos. En el primer paso se genera el producto cartesiano, es decir, todas las posibles combinaciones de filas de ambas tablas. El producto cartesiano se genera sin más que escribir los nombres de ambas tablas en la cláusula **from**. Por ejemplo, la sentencia siguiente:

```
select *
from   pueblos, provincias ;
```

generará las siguientes filas:

CODPUE	NOMBRE	CODPRO	CODPRO	NOMBRE
101	NULES	12	12	CASTELLÓN
102	ONDA	12	12	CASTELLÓN
103	SARRION	44	12	CASTELLÓN
101	NULES	12	44	TERUEL

102	ONDA	12	44	TERUEL
103	SARRION	44	44	TERUEL
101	NULES	12	46	VALENCIA
102	ONDA	12	46	VALENCIA
104	LIRIA	46	46	VALENCIA

De todas las filas anteriores, hay algunas que resultan interesantes y otras que no. Por ejemplo, la tercera fila no resulta interesante pues Sarrión no pertenece a la provincia de Castellón. Las filas que interesan son las que están relacionadas. ¿Como seleccionarlas? Pues con una condición de relación en la cláusula **where**. De esta forma la sentencia siguiente:

```
select *
from   pueblos, provincias
where  pueblos.codpro = provincias.codpro ;
```

generará las filas deseadas. Estas son las filas que están relacionadas:

CODPUE	NOMBRE	CODPRO	CODPRO	NOMBRE
101	NULES	12	12	CASTELLÓN
102	ONDA	12	12	CASTELLÓN
103	SARRION	44	44	TERUEL

Nótese que en la cláusula **where** se le ha antepuesto a cada columna el nombre de la tabla. Ello se debe a que existen dos columnas en dos tablas distintas con el mismo nombre, lo cual puede dar lugar a ambigüedades. Por ello, para eliminar dichas ambigüedades, siempre que dos o más columnas tengan el mismo nombre se le debe anteponer a cada columna el nombre o un alias de la tabla.

Pero no todas las columnas que aparecen en el resultado anterior interesan. El objetivo inicial era mostrar el nombre del pueblo y el nombre de la provincia, por tanto las demás columnas sobran. ¿Cómo deshacernos de éstas? Pues muy fácil, como ya se ha comentado en capítulos anteriores, la cláusula **select** permite elegir las columnas deseadas. Así pues, la sentencia siguiente:

```
select pueblos.nombre, provincias.nombre
from   pueblos, provincias
where  pueblos.codpro = provincias.codpro ;
```

mostrará el siguiente resultado:

NOMBRE	NOMBRE
NULES	CASTELLÓN
ONDA	CASTELLÓN
SARRION	TERUEL

En el ejemplo anterior el resultado obtenido es casi el deseado. Un ligero retoque en la cabecera de la tabla utilizando etiquetas o alias de columnas permite mejorar la legibilidad. Seguidamente se muestra la nueva sentencia y el resultado:

```
select pueblos.nombre pueblo, provincias.nombre provincia
from   pueblos, provincias
where  pueblos.codpro = provincias.codpro ;
```

PUEBLO	PROVINCIA
NULES	CASTELLÓN

ONDA	CASTELLÓN
SARRION	TERUEL

6.2 Alias de tablas

Habitualmente en las sentencias los nombres de las tablas aparecen repetidamente en casi todas las cláusulas. Si, además, éstos son largos, la escritura de las sentencias pasa a convertirse en algo tedioso y, por tanto, dado a errores. Para evitarlo, SQL proporciona un mecanismo para construir fácilmente alias de tablas. Los alias se indican en la cláusula **from**. Si una palabra sigue a un nombre de tabla, dicha palabra pasa a ser un alias para la tabla anterior. Los alias pueden emplearse en cualquier sitio de la sentencia en lugar del nombre original de la tabla.

La siguiente sentencia realiza la misma labor que la anterior, pero usa alias. Nótese la reducción en su longitud.

```
select p.nombre pueblo, pr.nombre provincia
from   pueblos p, provincias pr
where  p.codpro = pr.codpro ;
```

Algunos sistemas recomiendan usar siempre los alias de tablas en todas las columnas pues permiten al intérprete de SQL localizar la tabla de origen de las columnas mucho antes. Pensemos en una consulta con varias tablas, cada una con decenas de columnas. La simple búsqueda de una columna para determinar en qué tablas aparece y si lo hace más de una vez (posibles ambigüedades) puede costar bastante tiempo.

En algunos SGBD los alias de tablas deben definirse separando la tabla del alias con la palabra inglesa **as** (adverbio de comparación "como", en castellano).

6.3 Sintaxis estándar

Existen ciertas variaciones en la sintaxis de la concatenación interna en los diferentes estándares e implementaciones de SQL. Habitualmente existen dos formas (sintaxis) distintas de realizar el mismo concepto de concatenación interna: la sintaxis tradicional y la sintaxis estándar.

La presentada hasta ahora ha sido la sintaxis tradicional. En ella la concatenación se realiza conceptualmente en dos fases y, por tanto, en dos lugares: el producto cartesiano (en la cláusula **from**) y la restricción de filas (en la cláusula **where**).

En este apartado se va a presentar la sintaxis estándar con el mismo ejemplo (o con ejemplos de dificultad similar) del apartado anterior.

La sintaxis estándar presenta cuatro tipos de operadores relacionados con la concatenación interna. A continuación se describen con detalle.

6.3.1 Operador A natural join B

Devuelve como resultado las filas de la tabla A concatenadas con las filas de la tabla B de tal forma que las columnas de A y B que se llaman igual tienen los mismos valores.

En algunos sistemas no hay que prefijar las columnas por las que se realiza la concatenación (las columnas con igual nombre) con el nombre de la tabla.

- **Ejercicio**: Escribir una sentencia que muestre el código de factura, la fecha y el nombre del cliente destinatario de la factura.

  Solución:

```
select f.codfac, f.fecha, c.nombre
from   facturas f natural join clientes c ;
```

♦ **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select *
from   pueblos p natural join provincias pr ;
```

Solución: Esta sentencia no devuelve el nombre de cada pueblo junto al de su provincia puesto que hay dos columnas en las tablas pueblos y provincias con el mismo nombre: la columna codpro y la columna nombre. Por tanto, devolverá aquellos pueblos concatenados con las provincias tales que su código de provincia y su nombre coincide. Es decir, devolverá aquellos pueblos cuyo nombre coincide con el de su provincia.

6.3.2 Operador A [inner] join B using (lista_columnas)

Como se puede ver, la palabra **inner** es opcional. Esta operación realiza la concatenación interna de las tablas A y B según las columnas indicadas explícitamente en la lista.

Si una columna aparece en la lista de columnas por las que se va a realizar la concatenación, en algunos sistemas no hay que prefijarla con el nombre de la tabla en toda la consulta.

♦ **Ejercicio**: Escribir una sentencia que muestre el código de factura, la fecha y el nombre del cliente destinatario de la factura.
Solución:

```
select f.codfac, f.fecha, c.nombre
from   facturas f join clientes c using ( codcli );
```

♦ **Ejercicio**: Escribir una sentencia que muestre el nombre de cada pueblo y el de su provincia.
Solución:

```
select p.nombre, pr.nombre
from   pueblos p join provincias pr using ( codpro ) ;
```

6.3.3 Operador A [inner] join B on expresión_booleana

Como se puede ver, la palabra **inner** es opcional. Esta operación realiza el producto cartesiano de las filas de A y de las filas de B, dejando sólo aquellas filas en las que la expresión booleana se cumple.

♦ **Ejercicio**: Escribir una sentencia que muestre el código de factura, la fecha y el nombre del cliente destinatario de la factura.
Solución:

```
select f.codfac, f.fecha, c.nombre
from   facturas f join clientes c on f.codcli = c.codcli ;
```

♦ **Ejercicio**: Escribir una sentencia que muestre el nombre de cada pueblo y el de su provincia.
Solución:

```
select p.nombre, pr.nombre
from   pueblos p join provincias pr on p.codpro = pr.codpro ;
```

6.3.4 Operador A cross join B

Realiza el producto cartesiano de las tablas A y B. Esta operación es muy raramente empleada, pero el estándar la suministra por si fuera necesaria.

- **Ejercicio**: Escribir una sentencia que obtenga el producto cartesiano de la tabla clientes y vendedores.
  Solución:

```
select *
from   clientes c cross join vendedores ;
```

6.3.5 Sintaxis tradicional frente a sintaxis estándar

La sintaxis estándar es aceptada por la mayor parte de implementaciones actuales: Oracle 9i, PostGreSQL, etc. Afortunadamente la mayor parte de los sistemas con sintaxis estándar también aceptan la sintaxis tradicional sin mayores problemas. De todas formas, algunos sistemas que aceptan la sintaxis estándar, como Access, sólo aceptan el modo **A inner join B on condición** y no el resto.

No obstante, en adelante se trabajará con la sintaxis estándar ya que es menos dada a errores. Un error muy habitual con la sintaxis tradicional es el olvido de la condición de restricción tras el producto cartesiano de dos tablas cuando se están concatenando muchas, pues la concatenación se realiza en dos fases y en dos partes distintas: el producto cartesiano se genera en la cláusula **from** y la restricción se realiza en la cláusula **where**. Con la sintaxis estándar dicho olvido resulta más difícil pues la operación de concatenación se realiza en un único lugar.

6.3.6 Método de trabajo con la sintaxis estándar

Dado el número de operadores de concatenación, es conveniente fijar algunas ideas sobre su funcionamiento y el método a seguir cuando se trabaja con ellos.

Cuando la concatenación es interna (la concatenación externa se verá más adelante), puede eliminarse sin problemas la palabra **inner**. En adelante, se escribirán así las consultas.

Pese a la variedad de operadores de la sintaxis estándar, el modo de trabajo es claro. Si hay que realizar un producto cartesiano se recurre al operador **A cross join B**. Si hay que realizar una concatenación interna, se recurre al resto.

El operador **A natural join B** no es muy aconsejable y se recomienda emplear los otros dos operadores por los siguientes motivos:

- Como ya se ha explicado, el operador **A natural join B** realiza la concatenación considerando *todas* las columnas cuyo nombre coincide. Desafortunadamente, es muy fácil olvidar que dos tablas tienen más de una columna con el mismo nombre, por lo que si se emplea este operador el resultado será muy distinto del esperado en caso de olvido.
- Como la gran ventaja que aportan las bases de datos frente a los sistemas de ficheros es su flexibilidad, durante la vida de la bases de datos es muy normal que se añadan algunas columnas para aumentar y mejorar la información almacenada. Por ello, se podría añadir inadvertidamente una nueva columna a una tabla cuyo nombre coincidiera en el de otra columna en otra tabla, por lo que una consulta con **natural join** de ambas tablas podría obtener un resultado muy distinto al original tras la inserción de la nueva columna. Además, este error no se detectaría inmediatamente sino sólo cuando se ejecutara la consulta, lo cual dificultaría su búsqueda y reparación.

Por tanto, se recomienda recurrir a los otros operadores pues en ellos hay que indicar explícitamente las columnas empleadas en la concatenación. Así pues, en caso de que las columnas por las que se desea realizar la operación de concatenación tengan el mismo nombre, se recurre al operador **A join B using**. En caso contrario, se usa el operador **A join B on**.

- **Ejercicio**: Escribir una sentencia que muestre el código de factura, la fecha y el nombre del vendedor.

Solución:

```
select f.codfac, f.fecha, v.nombre
from   facturas f join vendedores v using ( codven ) ;
```

- **Ejercicio**: Escribir una sentencia que muestre el código de factura, el número de línea, el código del artículo, la descripción del artículo y el número de unidades vendidas en dicha línea para todas las líneas de la factura cuyo código es 15.
  Solución:

```
select l.codfac, l.linea, codart, a.descrip, l.cant
from   lineas_fac l join articulos a using ( codart )
where  l.codfac = 15 ;
```

- **Ejercicio**: Escribir una sentencia que muestre el nombre del vendedor y el nombre del pueblo en el que reside para aquellos vendedores cuyo código se encuentra entre 100 y 200, inclusive.
  Solución:

```
select v.nombre, p.nombre
from   pueblos p join vendedores v using ( codpue )
where  v.codven between 100 and 200 ;
```

- **Ejercicio**: Escribir una sentencia que muestre el nombre de aquellos pueblos cuyo nombre coincide con algún nombre de provincia.
  Solución:

```
select nombre
from   pueblos p join provincias pr using ( nombre );
```

- **Ejercicio**: Escribir una sentencia que muestre el nombre de aquellas pueblos cuyo nombre coincide con el nombre de su provincia.
  Solución:

```
select nombre
from   pueblos p join provincias pr using ( codpro, nombre );
```

```
select nombre
from   pueblos p natural join provincias pr ;
```

- **Ejercicio**: Escribir una sentencia que muestre el nombre de aquellos clientes y vendedores cuyo nombre coincide.
  Solución:

```
select nombre
from   clientes c join vendedores v using ( nombre );
```

6.4 Concatenación Interna de tres o más tablas

La concatenación de tres o más tablas no introduce ninguna dificultad conceptual adicional. El método para concatenar tres tablas es bien sencillo: se concatenan dos tablas entre sí (con una concatenación como la descrita en apartados anteriores) y, después, se concatena el resultado con la tercera tabla.

Si se desean concatenar más de tres tablas, el método es análogo al anterior. Si se quiere extraer información de *n* tablas, harán falta *n-1* concatenaciones.

- **Ejercicio**: Escribir una sentencia que muestre el código y fecha de cada factura junto al nombre del cliente y el nombre del vendedor de la factura.
  Solución:

```
select f.codfac, f.fecha, c.nombre, v.nombre
from   facturas f join clientes c using ( codcli )
                  join vendedores v using ( codven ) ;
```

- **Ejercicio**: Escribir una sentencia que muestre el código y fecha de cada factura, junto al nombre y dirección completa del cliente.
  Solución:

```
select f.codfac, f.fecha, c.nombre, c.direccion, c.codpostal,
       p.nombre, pr.nombre
from   facturas f join clientes c using ( codcli )
                  join pueblos p using ( codpue )
                  join provincias pr using ( codpro ) ;
```

6.5 Concatenación de una tabla consigo misma

La concatenación de una tabla consigo mismo no es ningún problema en SQL. La única consideración especial es que el nombre de la tabla debe aparecer dos veces en la cláusula **from** pero con dos alias distintos.

- **Ejercicio**: Escribir una sentencia que muestre el nombre de cada vendedor y el de su inmediato jefe.
  Solución:

```
select v.nombre, j.nombre
from   vendedores v join vendedores j on v.codjefe =
j.codven;
```

- **Ejercicio**: Escribir una sentencia que muestre los códigos y la descripción de aquellos artículos con la misma descripción.
  Solución:

```
select a1.codart, a2.codart, descrip
from   articulos a1 join articulos a2 using ( descrip )
where  a1.codart < a2.codart ;
```

6.6 Concatenación y Agrupación

La combinación de la concatenación y la agrupación funciona como era de esperar. Hay que comentar que en ocasiones resulta obligatorio añadir algunos factores de agrupación que realmente no realizan ninguna agrupación adicional.

Considérese el siguiente ejemplo: Escribir una sentencia que muestre el código, nombre y número de pueblos para cada provincia. En principio la escritura de esta sentencia no requiere de ningún concepto nuevo: una concatenación de dos tablas y una mera agrupación. Véase la siguiente sentencia:

```
select codpro, pr.nombre, count( * )
from   pueblos p join provincias pr using ( codpro )
group  by codpro ;
```

En algunos sistemas la sentencia anterior produce un error de ejecución debido a que no se cumple la regla de oro: "Todo lo que está en el **select** y en el **having** o son funciones de grupo o están en el **group by**". En este caso, la columna **pr.nombre** no está en el **group by** ni es función de grupo. El error se produce porque el sistema ve que se ha agrupado sólo por **codpro** y se le está pidiendo que muestre todos los nombres de la provincia del grupo. Evidentemente sólo puede haber un nombre de provincia en cada grupo dado que hemos agrupado por código de provincia, pero el sistema no lo sabe y genera el error.

La forma de evitar este error es insertar una agrupación adicional que en realidad no agrupa más de lo que hace la primera, pero que "tranquiliza" al sistema. Por tanto, la sentencia queda de la siguiente forma:

```
select codpro, pr.nombre, count( * )
from   pueblos p join provincias pr using ( codpro )
group  by codpro, pr.nombre ;
```

Como se puede ver, se ha añadido una agrupación adicional por el nombre de la provincia. En realidad, esta agrupación no va a agrupar más las filas de lo que hace la primera, es decir, no va a generar nuevos grupos. Pero de esta forma se cumple la regla de oro y el sistema puede ejecutar la sentencia tranquilamente.

- **Ejercicio**: Escribir una sentencia que muestre el código y nombre de cada vendedor, junto al número de facturas que ha realizado.
  Solución:

```
select codven, v.nombre, count( * )
from   vendedores v join facturas f using ( codven )
group  by codven, v.nombre ;
```

6.7 Consideraciones sobre las prestaciones

Se ha dicho que la concatenación se realiza *conceptualmente* como una restricción del producto cartesiano, es decir, una selección de entre todas las posibles combinaciones de las filas. En la práctica, los SGBD aplican técnicas que permiten evitar tener que generar *realmente* el producto cartesiano con todas las posibles combinaciones, lo cual es un proceso demasiado costoso en tiempo y en espacio.

Las prestaciones varían enormemente de unos sistemas a otros. La misma sentencia puede ofrecer unas prestaciones radicalmente distintas cuando se trabaja sobre SGBD distintos, incluso sobre un mismo *hardware*.

Incluso la misma sentencia puede ofrecer resultados muy distintos si es reescrita de un modo ligeramente distinto.

En algunos sistemas el orden de las tablas en la cláusula **from** puede resultar muy importante. Por ejemplo, en versiones anteriores de Oracle se recomendaba que para obtener una mayor velocidad la última tabla de dicha cláusula fuera siempre la que tuviera menos filas. En otros sistemas el orden puede no ser tan importante o incluso puede que interese usar el orden completamente contrario. Cada sistema tiene sus preferencias.

En algunos sistemas las comparaciones a = b de la cláusula **where** o las que siguen a la palabra **on** son muy importantes. Aunque las comparaciones son conmutativas matemáticamente hablando (a = b es lo mismo que b = a), en algunos sistemas interesa que la expresión de la izquierda tenga un índice asociado (como por ejemplo la clave primaria) para que así la búsqueda se realice a una velocidad mucho mayor. De esta forma la expresión:

```
where c.codcli = f.codcli
```

se evaluaría mucho más rápidamente que la expresión:

```
where f.codcli = c.codcli
```

Aunque no existe ningún límite en el número de tablas que se pueden concatenar entre sí, cuanto mayor sea el número de tablas, menores serán las prestaciones. Algunos sistemas como por ejemplo SQL Server recomiendan que no se concatenen más de 4 tablas.

6.8 Ejercicios

- **Ejercicio 6.1**: Escribir una consulta que obtenga el código y nombre de cada cliente y el número de facturas que ha realizado durante el año pasado.
  Ayuda: Concatenación de dos tablas y agrupación por código de cliente. Es necesaria una agrupación adicional por nombre de cliente.
  Solución: La siguiente sentencia se ha escrito empleando las funciones de fecha de Oracle.

```
select codcli, c.nombre, count( * )
from   clientes c join facturas f using ( codcli )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c.nombre ;
```

- **Ejercicio 6.2**: Escribir una consulta que obtenga el código de factura, la fecha y el importe (sin considerar descuentos ni impuestos) de cada una de las facturas.
  Ayuda: Concatenación de dos tablas y agrupación por código de factura. Es necesaria una agrupación adicional por fecha.
  Solución:

```
select codfac, f.fecha, sum( l.cant * l.precio )
from   facturas f join lineas_fac l using ( codfac )
group  by codfac, f.fecha ;
```

- **Ejercicio 6.3**: Escribir una sentencia que calcule el código y nombre de cada vendedor y su facturación durante el año pasado.
  Ayuda: Concatenación de tres tablas.
  Solución:

```
select codven, v.nombre, sum( l.cant * l.precio )
from   vendedores v join facturas f using ( codven )
                    join lineas_fac l using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codven, v.nombre ;
```

- **Ejercicio 6.4**: Escribir una sentencia que calcule el número de unidades vendidas en cada provincia durante el año pasado.
  Ayuda: Concatenación de cinco tablas.
  Solución:

```
select codpro, pr.nombre, sum( l.cant )
from   provincias pr join pueblos p using ( codpro )
                     join clientes c using ( codpue )
                     join facturas f using ( codcli )
                     join lineas_fac l using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codpro, pr.nombre ;
```

- **Ejercicio 6.5**: Escribir una consulta que obtenga el código y nombre de aquellos clientes que han sido atendidos alguna vez por vendedores residentes en otras provincias.
  Ayuda: Concatenación de 5 tablas. Hay una restricción adicional para comprobar que la provincia del cliente y la provincia del vendedor son distintas.
  Solución:

```
select distinct codcli, c.nombre
from   clientes c join pueblos p1 on c.codpue = p1.codpue
                  join facturas f using ( codcli )
                  join vendedores v using ( codven )
                  join pueblos p2 on v.codpue = p2.codpue
where  p1.codpro <> p2.codpro ;
```

- **Ejercicio 6.6**: Escribir una consulta que obtenga el código y nombre de aquellos clientes de la provincia de Valencia que tienen alguna factura con 10 líneas o más.
  Ayuda: Concatenación de 5 tablas. Agrupación por código de factura y cliente para determinar si una factura tiene 10 o más líneas.
  Solución:

```
select distinct codcli, c.nombre
from   lineas_fac l join facturas f using ( codfac )
                    join clientes c using ( codcli )
                    join pueblos pu using ( codpue )
                    join provincias pr using ( codpro )
where  upper( pr.nombre ) = 'VALENCIA'
group  by codfac, codcli, c.nombre
having count( * ) > 9;
```

- **Ejercicio 6.7**: Escribir una consulta que obtenga el código y descripción de aquellos artículos que durante el año pasado se vendieron siempre en varios (más de uno) meses consecutivos. Por ejemplo, artículos vendidos en marzo, abril y mayo, pero no aquéllos vendidos en agosto y diciembre.

  Ayuda: Concatenación de 3 tablas. Agrupación por código de artículo. Para cada artículo hay que comprobar dos condiciones: ha sido vendido en más de un mes y ha sido vendido en meses consecutivos. Para comprobar que el artículo ha sido vendido en varios meses consecutivos hay que comprobar que el número mayor de mes menos el número menor de mes más uno es igual al número de meses en que ha sido vendido. Por ejemplo, si un artículo ha sido vendido en marzo, abril y mayo se cumple: 5 – 3 + 1 = 3. Por ejemplo, si un artículo ha sido vendido en agosto y diciembre, no se cumple: 12 – 8 +1 = 2.

  Solución: La siguiente consulta se ha escrito con la sintaxis de Oracle.

```
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
                   join facturas f using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codart, a.descrip
having count( distinct to_char( f.fecha, 'mm' ) ) > 1
and    max( to_number( to_char( f.fecha, 'mm' ) ) ) -
       min( to_number( to_char( f.fecha, 'mm' ) ) ) + 1 =
       count( distinct to_char( f.fecha, 'mm' ) ) ;
```

- **Ejercicio 6.8**: Escribir una consulta que muestre el código y nombre de aquellos clientes de la provincia de Castellón que han facturado más de 6000 euros.

  Solución:

```
select codcli, c.nombre
from   clientes c join facturas f   using ( codcli )
                  join lineas_fac l using ( codfac )
                  join pueblos p    using ( codpue )
where  p.codpro = '12'
group  by codcli, c.nombre
having sum( cant * precio ) > 6000.00 ;
```

- **Ejercicio 6.9**: Escribir una consulta que calcule la facturación máxima realizada por los clientes de la provincia de Castellón en un mes del año pasado.

  Solución:

```
select max( sum( l.cant * l.precio ) )
from   clientes c join facturas f   using ( codcli )
                  join lineas_fac l using ( codfac )
                  join pueblos p    using ( codpue )
where  p.codpro = '12'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, to_char( fecha, 'mm' ) ;
```

- **Ejercicio 6.10**: Escribir una consulta que obtenga el nombre de cada jefe y el número de vendedores que dependen de él (se considerará como jefe a aquel vendedor que es jefe de al menos otro vendedor).

Solución:

```
select j.codven, j.nombre, count( * )
from   vendedores j join vendedores v
         on ( v.codjefe = j.codven )
group  by j.codven, j.nombre ;
```

6.9 Autoevaluación

- **Ejercicio 1:** Para aquellos clientes de la Comunidad Valenciana cuyo nombre comienza por la misma letra que comienza el nombre del pueblo en el que residen, mostrar el nombre del cliente, el nombre del pueblo y el número de artículos distintos comprados durante el último trimestre del año pasado. En el listado final sólo deben aparecer aquellos clientes cuya facturación en el mismo periodo superó los 6000 euros, sin considerar impuestos ni descuentos.
- **Ejercicio 2:** Artículos cuya descripción consta de más de 15 letras o dígitos que han sido comprados por más de 5 clientes distintos de la provincia de Castellón durante los últimos diez días del año pasado. En el listado final se debe mostrar el artículo y su descripción.
- **Ejercicio 3:** Código y nombre de aquellos pueblos cuya primera letra del nombre es la misma que la primera letra del nombre de la provincia, en los que residen más de 3 clientes y en los que se han facturado más de 1000 unidades en total durante el tercer trimestre del año pasado.
- **Ejercicio 4:** Para aquellos vendedores cuyo primer o segundo apellido terminan con 'EZ' (se asume que ningún nombre de pila termina con dicho sufijo), mostrar el número de clientes de su misma provincia a los que ha realizado alguna venta durante los 10 últimos días del año pasado. Mostrar el código y nombre del vendedor, además del citado número de clientes.

7 ORDENACIÓN Y OPERACIONES ALGEBRAICAS

Este capítulo describe en primer lugar la ordenación del resultado obtenido por la consulta y, en segundo lugar, las tres operaciones algebraicas que suele proporcionar el lenguaje SQL: unión, intersección y diferencia.

7.1 Ordenación del resultado

La ordenación del resultado de una consulta suele ser un proceso muy costoso en la mayor parte de los casos (cuando no existen índices que aceleren el proceso). Por tanto, la ordenación sólamente debe realizarse si es estrictamente necesaria.

La ordenación del resultado de una consulta se indica mediante la cláusula **order by**, la cual debe aparecer siempre en último lugar de la sentencia **select**. Su sintaxis general es la siguiente:

```
order by [tabla|alias].columna [ASC|DESC]
         [, [tabla|alias].columna [ASC|DESC] ]
```

Como se puede ver, se puede ordenar el resultado por una o más columnas, tanto de forma ascendente como de forma descendente. Por defecto, si no se indica nada respecto del orden, éste se realiza ascendentemente. Cuando existe más de una columna en la cláusula **order by**, la ordenación se realiza de izquierda a derecha. Es decir, se ordena el resultado por la primera columna y si algunas filas coinciden en sus valores de la primera columna, se ordenan por la segunda y así sucesivamente.

La anterior sintaxis tiene dos excepciones:

- En lugar de una columna de una tabla se puede ordenar por una expresión o parte de una columna.
- En lugar de una columna de una tabla se puede indicar un número entero. En este caso se ordenará por la columna o expresión de la cláusula **select** cuya posición coincida con el mencionado número. Por ejemplo, si aparece un 3 en la cláusula **order by**, entonces se ordenará el resultado de la consulta por la tercera columna o expresión de la cláusula **select**.

A continuación se muestran distintos ejemplos de cláusulas de ordenación:

```
order by pr.nombre ;
order by pr.nombre, c.nombre;
order by to_char( f.fecha, 'mm' ), c.codcli;
order by 3, 1 desc, 2 asc ;
```

7.2 Operaciones algebraicas

En el lenguaje SQL se pueden realizar diversas operaciones algebraicas con el resultado de la ejecución de consultas. El estándar SQL proporciona operadores algebraicos de **unión**, **intersección** y **diferencia** de los resultados de consultas. Desafortunadamente, algunas implementaciones no proporcionan todos ellos, ofreciendo sólo el operador de unión.

La forma de uso de los operadores algebraicos es la siguiente:

```
sentencia_select ...
union | intersect | minus | except [all]
sentencia_select ...  [
union | intersect | minus | except [all]
sentencia_select ...  ]
[ order by ...  ] ;
```

Como se puede ver, con estos operadores se pueden encadenar tantas sentencias **select** como se quiera.

Todas las sentencias **select** deben devolver el mismo número de columnas y el mismo tipo de cada una de las columnas.

En cualquier caso, sólo puede haber una única cláusula **order by** al final. La ordenación se realiza sobre el resultado final.

A continuación se describen con detalle cada uno de los operadores mencionados.

7.2.1 Operador de unión

El operador **union** devuelve como resultado todas las filas que devuelve la primera sentencia **select**, más aquellas filas de la segunda sentencia **select** que no han sido ya devueltas por la primera. En el resultado no se muestran duplicados pues se utilizan algoritmos de eliminación de duplicados (mediante ordenación), por lo que el resultado también aparecerá ordenado.

El operador **union all** no elimina duplicados. En este caso, si una fila aparece *m* veces en la primera sentencia y *n* veces en la segunda, en el resultado aparecerá *m*+*n* veces.

Si se realizan varias uniones, éstas se evalúan de izquierda a derecha (o de primera a la última), a menos que se utilicen paréntesis para establecer un orden distinto.

- **Ejercicio**: Dadas las siguientes tablas **A** y **B**, calcular el resultado de **A union B** y **A union all B.**

Tabla A
10
10
20
20
50

Tabla B
0
10
10
20

Solución: El resultado se muestra a continuación:

A union B	A union all B
0	10
10	10
20	20
50	20
	50
	0
	10
	10
	20

- **Ejercicio**: Facturas para el cliente con código 291 o para el vendedor con código 495.
  Ayuda: Se puede realizar de dos formas: con el operador *union* o sin él (en este caso es más eficiente esto último).
  Solución:

```
select *
from   facturas
```

```
where  codcli = 291
or     codven = 495 ;

select *
from   facturas
where  codcli = 291
union
select *
from   facturas
where  codven = 495 ;
```

♦ **Ejercicio**: Códigos de pueblos donde hay clientes o donde hay vendedores.
Solución:

```
select codpue
from   clientes
union
select codpue
from   vendedores ;
```

7.2.2 Operador de intersección

El operador **intersect** devuelve como resultado las filas que se encuentran tanto en el resultado de la primera sentencia **select** como en el de la segunda sentencia **select**. En el resultado no se muestran duplicados.

El operador **intersect all** no elimina duplicados. En este caso, si una misma fila aparece *m* veces en la primera sentencia y *n* veces en la segunda, en el resultado esta fila aparecerá *min(m,n)* veces.

Si se realizan varias intersecciones, éstas se evalúan de izquierda a derecha (o de primera a última), a menos que se utilicen paréntesis para establecer un orden distinto.

La intersección tiene mayor prioridad, en el orden de evaluación, que la unión, es decir, **A union B intersect C** se evalúa como **A union (B intersect C)**.

♦ **Ejercicio**: Dadas las anteriores tablas **A** y **B**, calcular el resultado de **A intersect B** y **A intersect all B.**
Solución: El resultado se muestra a continuación:

A intersect B	A intersect all B
10	10
20	10
	20

♦ **Ejercicio**: Códigos de pueblos donde residen tanto clientes como vendedores.
Solución:

```
select codpue
from   clientes
intersect
select codpue
from   vendedores ;
```

7.2.3 Operador de diferencia

El operador **minus|except** devuelve como resultado las filas que se encuentran en el resultado de la primera sentencia **select** y no se encuentran en el resultado de la segunda sentencia **select**. En el resultado no se muestran duplicados.

El operador **minus all|except all** no elimina duplicados. En este caso, si una misma fila aparece *m* veces en la primera sentencia y *n* veces en la segunda, en el resultado esta fila aparecerá *max(m-n,0)* veces.

Si se realizan varias diferencias, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto. La diferencia tiene la misma prioridad, en el orden de evaluación, que la unión.

En algunos sistemas la diferencia se indica con la palabra **except**, mientras que en otros se indica con la palabra **minus**. Su funcionamiento es idéntico.

A diferencia de los otros operadores algebraicos, la diferencia de conjuntos no es un operador conmutativo.

- **Ejercicio**: Dadas las anteriores tablas **A** y **B**, calcular el resultado de **A minus B** y **A minus all B.**
  Solución: El resultado se muestra a continuación:

A minus B	A minus all B
50	20
	50

- **Ejercicio**: Códigos de pueblos donde no hay clientes.
  Solución:

```
select codpue
from   pueblos
minus
select codpue
from   clientes ;
```

7.2.4 Uso incorrecto de los operadores algebraicos

Cuando se realiza una operación algebraica de dos consultas, es obligatorio que coincida el número y tipo de las columnas devueltas por cada una de las consultas. En caso contrario, el sistema suele avisar con un error de ejecución.

A continuación se muestra un ejemplo de uso incorrecto debido a que no coincide ni el número ni el tipo de las columnas devueltas por ambas consultas.

```
select codpue, codpro
from   pueblos
minus
select nombre, codpue, codcli
from   clientes ;
```

Si en todas las consultas encadenadas con los operadores algebraicos se usan exactamente las mismas tablas, es mejor rescribir la sentencia recurriendo a los operadores lógicos **or**, **and** y **not**, los cuales simplifican enormemente la escritura y ejecución de las consultas.

A continuación se muestra un ejemplo de uso ineficiente de los operadores algebraicos. La siguiente sentencia devuelve aquellos clientes que se llaman o apellidan garcia (en su primer o segundo apellido) o cuyo código postal pertenece a la provincia de Alicante.

```
select *
from   clientes
where  nombre like '%garcia%'
union
select *
from   clientes
where  substr( codpostal, 1, 2 ) = '03' ;
```

Como se puede ver en la sentencia anterior, ambas consultas trabajan sobre las mismas tablas. Dicha sentencia se puede rescribir muy fácilmente mediante el operador **or**. Seguidamente se muestra el ejemplo anterior escrito de forma más eficiente:

```
select *
from   clientes
where  nombre like '%garcia%'
or     substr( codpostal, 1, 2 ) = '03' ;
```

7.2.5 Variantes de SQL y operadores algebraicos

Desafortunadamente, no todos los SGBD ofrecen todos los operadores algebraicos antes descritos. Algunas implementaciones de SQL, como por ejemplo Access 2000, suelen ofrecer el operador **union**, pero no el resto. Otras implementaciones, como Oracle 9i, ofrecen los operadores **union**, **intersect**, **minus** y **union all**, pero no los **intersect all** ni el **minus all**.

7.3 Ejercicios

- **Ejercicio 7.1**: Escribir una consulta que obtenga el código y nombre de aquellos pueblos donde residen al menos un vendedor o al menos un cliente. No eliminar del resultado los pueblos repetidos.
  Ayuda: Operación de unión sin eliminación de repetidos de dos consultas. En cada consulta se ha realizado una concatenación con la tabla pueblos para poder extraer el nombre del pueblo.
  Solución:

```
select codpue, p1.nombre
from   pueblos p1 join vendedores v using ( codpue )
union all
select codpue, p2.nombre
from   pueblos p2 join clientes c using ( codpue );
```

- **Ejercicio 7.2**: Escribir una consulta que obtenga el código y nombre de aquellos pueblos donde residen al menos un vendedor o al menos un cliente. Eliminar del resultado los pueblos repetidos.
  Ayuda: Unión de dos consultas. En cada consulta se ha realizado una concatenación con la tabla pueblos para poder extraer el nombre del pueblo.
  Solución:

```
select codpue, p1.nombre
from   pueblos p1 join vendedores v using ( codpue )
union
select codpue, p2.nombre
from   pueblos p2 join clientes c using ( codpue );
```

- **Ejercicio 7.3**: Escribir una consulta que obtenga el código y nombre de aquellos pueblos donde residen al menos un vendedor y al menos un cliente.
  Ayuda: Intersección de dos consultas. En cada consulta se ha realizado una concatenación con la tabla pueblos para poder extraer el nombre del pueblo.
  Solución:

```
select codpue, p1.nombre
from   pueblos p1 join vendedores v using ( codpue )
intersect
select codpue, p2.nombre
from   pueblos p2 join clientes c using ( codpue );
```

♦ **Ejercicio 7.4**: Escribir una consulta que obtenga el código y nombre de aquellos pueblos donde residen al menos un vendedor pero no reside ningún cliente.
Ayuda: Diferencia de dos consultas. En cada consulta se ha realizado una concatenación con la tabla pueblos para poder extraer el nombre del pueblo.
Solución:

```
select codpue, p1.nombre
from   pueblos p1 join vendedores v using ( codpue )
minus
select codpue, p2.nombre
from   pueblos p2 join clientes c using ( codpue );
```

♦ **Ejercicio 7.5**: Escribir una consulta que obtenga el código y descripción de aquellos artículos que nunca han sido vendidos en el mes de enero.
Ayuda: Diferencia de dos consultas.
Solución:

```
select a1.codart, a1.descrip
from   articulos a1
minus
select codart, a2.descrip
from   articulos a2 join lineas_fac l using ( codart )
                    join facturas f using ( codfac )
where  to_char( f.fecha, 'mm' ) = '01';
```

♦ **Ejercicio 7.6**: Escribir una consulta que muestre el código de cada artículo cuyo *stock* supera las 20 unidades, con un precio superior a 15 euros, y de los que no hay ninguna factura en el último trimestre del año pasado.
Ayuda: Uso del operador *minus*.
Solución:

```
select a.codart
from   articulos a
where  a.stock > 20
and    a.precio > 15
minus
select l.codart
from   lineas_fac l, facturas f
where  f.codfac = l.codfac
and    to_char( f.fecha, 'q' ) = '4'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1;
```

♦ **Ejercicio 7.7**: Vendedores y clientes cuyo nombre coincide (vendedores que a su vez han comprado algo a la empresa).
Ayuda: Intersección de dos consultas.
Solución:

```
select v.nombre
from   vendedores v
intersect
select c.nombre
from   clientes c ;
```

♦ **Ejercicio 7.8**: Escribir una consulta que muestre los códigos de los artículos tales que su *stock* esté por debajo del doble de su *stock* mínimo, y el número total de unidades vendidas sea mayor que 100.
Ayuda: Uso del operador *intersect*.
Solución:

```
select a.codart
from   articulos a
where  a.stock < a.stock_min * 2
```

```
intersect
select l.codart
from   lineas_fac l
group  by l.codart
having sum( l.cant ) > 100;
```

- **Ejercicio 7.9**: Escribir una consulta que obtenga la facturación mensual de cada mes y también la facturación anual tras el mes de diciembre de todos los años en los que la empresa está operando.
  Ayuda: Se realiza uniendo el resultado de dos consultas. La primera calcula la facturación para cada mes de cada año. La segunda calcula la facturación anual. La primera columna de ambas consultas es un código alfanumérico que consigue que la facturación anual siga a la del mes de diciembre.
  Solución:

```
select to_char( f.fecha, 'yyyy' ) ||
       to_char( f.fecha, 'mm' ) codigo,
       to_char( f.fecha, 'yyyy' ) anyo,
       to_char( f.fecha, 'mm' ) mes,
       sum( l.cant * l.precio ) facturacion
from   facturas f join lineas_fac l using ( codfac )
group  by to_char( f.fecha, 'yyyy' ), to_char( f.fecha, 'mm'
)
union
select to_char( f.fecha, 'yyyy' ) ||
       'ft' codigo,
       to_char( f.fecha, 'yyyy' ) anyo,
       '--' mes,
       sum( l.cant * l.precio ) facturacion
from   facturas f join lineas_fac l using ( codfac )
group  by to_char( f.fecha, 'yyyy' )
order  by 1 ;
```

7.4 Autoevaluación

- **Ejercicio 1**: Escribir una consulta que obtenga el código y nombre de aquellas provincias en las que no hubo ventas de los vendedores residentes en dichas provincias durante el año pasado.
- **Ejercicio 2**: Escribir una consulta que muestre el código y descripción de aquellos artículos que se han vendido alguna vez, pero nunca en la provincia de Castellón.
- **Ejercicio 3**: Escribir una consulta que muestre el nombre de cada provincia y el número de facturas realizadas a clientes de dicha provincia durante el año pasado. Si una provincia no tiene ninguna factura, debe aparecer con la cantidad cero.

8 CONCATENACIÓN EXTERNA DE TABLAS

Este capítulo aborda también la concatenación de tablas: una operación muy habitual y útil en el procesamiento de una base de datos que permite el acceso y extracción de información que se encuentra repartida en varias tablas. La concatenación externa de tablas es una variante de la concatenación interna tal que permite que no se pierdan filas de una tabla aunque no exista ninguna fila relacionada en la otra tabla.

8.1 Problemas de la concatenación interna

La concatenación interna es un método muy empleado en la recuperación y extracción de información distribuida entre varias tablas. Sin embargo, a veces este tipo de concatenación tiene un inconveniente cuando se aplica a bases de datos del mundo real, al contener éstas gran cantidad de valores nulos.

Vamos a verlo con un ejemplo. Supongamos que se desea mostrar el nombre de cada cliente junto al del pueblo en el que reside y supongamos que tenemos las dos tablas siguientes (sólo se muestran aquellas columnas con las que se va a trabajar).

Tabla CLIENTES

CODCLI	NOMBRE	CODPUE
101	Alberto	1000
102	Carlos	1000
103	Juan	
104	Pedro	1001

Tabla PUEBLOS

CODPUE	NOMBRE
1000	Cella
1001	Olocau
1002	Jumilla

A continuación se muestra las consultas con concatenación interna y el resultado obtenido con ellas. Se muestra en primer lugar la sentencia con la sintaxis estándar y después con la sintaxis tradicional.

```
select c.nombre cliente, p.nombre pueblo
from   clientes c join pueblos p using ( codpue ) ;

select c.nombre cliente, p.nombre pueblo
from   clientes c, pueblos p
where  c.codpue = p.codpue ;
```

CLIENTE	PUEBLO
Alberto	Cella
Carlos	Cella
Pedro	Olocau

Sin embargo, esta sentencia presenta un ligero inconveniente: en el resultado final no aparece ni el cliente Juan ni el pueblo de Jumilla. Ambos casos son similares, pero ligeramente distintos:

- El cliente Juan, cuyo código de pueblo es nulo, no aparece porque al realizar la concatenación no existe ningún pueblo cuyo código sea nulo. En realidad, no puede existir ningún pueblo cuyo código sea el valor nulo si se cumple la regla de integridad de entidades. Es decir, por muchas filas que se añadan a la otra tabla, el cliente nunca podrá aparecer a menos que se modifique su código de pueblo.
- El pueblo de Jumilla no aparece porque no existe ningún cliente domiciliado en dicho pueblo. Por tanto, al realizar la concatenación, como no existe ningún cliente cuyo código de pueblo es el 1002, este pueblo desaparece. En este caso, si se añadiera una fila (un cliente) en la otra tabla que residiera en Jumilla, este pueblo sí que aparecería.

En ocasiones resulta muy interesante que no se pierda ninguna fila de una u otra tabla al realizar la concatenación. En tales casos, se suele recurrir a la concatenación externa. Esta es muy similar a la interna sólo que evita que se pierdan filas que no están relacionadas. Estos requerimientos no son nada extraños, sino más bien muy habituales. Ejemplos de este tipo de requerimientos son los siguientes:

- Listado de clientes con sus direcciones completas. Si un cliente no tiene código de pueblo y se realiza una concatenación interna, éste no aparece en el listado, lo cual puede dar lugar a creer que no existe en la base de datos.
- Listado de facturas con sus importes y nombres de clientes. Si una factura no tiene código de cliente y se realiza una concatenación interna, ésta no aparece en el listado.
- Listado de artículos con las unidades vendidas durante el año pasado. Si un artículo no ha sido vendido, no aparecerá. En muchas ocasiones puede resultar interesante que sí aparezca dicho artículo, pero teniendo como venta 0 unidades.
- etc.

8.2 Concatenación externa de dos tablas

Existen ciertas variaciones en la sintaxis de la concatenación interna en los diferentes estándares e implementaciones de SQL. Prácticamente existen dos formas (sintaxis) distintas de realizar el mismo concepto de concatenación interna: la tradicional y la del estándar SQL-99.

No obstante, afortunadamente la mayor parte de los sistemas con sintaxis estándar (Oracle 9i, Access, etc.) aceptan la sintaxis tradicional sin más problemas.

Aunque este texto también describe la sintaxis tradicional, se recomienda el uso de la sintaxis estándar por su mayor legibilidad y, también, por su mayor potencia.

La sintaxis estándar de la concatenación externa es la siguiente:

- **A left [outer] join B**: El resultado contiene todas las filas de la tabla A. Las filas de la tabla A que se relacionan con alguna de las filas de la tabla B aparecen concatenadas en el resultado. Las filas de la tabla A que no se relacionan con ninguna fila de la tabla B aparecen en el resultado concatenadas con una fila de nulos.
- **A right [outer] join B**: El resultado contiene todas las filas de la tabla B. Las filas de la tabla B que se relacionan con alguna de las filas de la tabla A aparecen concatenadas en el resultado. Las filas de la tabla B que no se relacionan con ninguna fila de la tabla A aparecen en el resultado concatenadas con una fila de nulos.
- **A full [outer] join B**: El resultado contiene todas las filas de las tablas A y B. Realiza las dos operaciones anteriores simultáneamente.

Como se puede ver, la palabra **outer** es opcional. Como no hay posible confusión de las concatenaciones externas entre sí ni con la concatenación interna, en adelante no se usará esta palabra.

Seguidamente se describen con más detalle estos tres tipos de concatenación externa.

8.2.1 Concatenación externa por la izquierda: A left join B

Supongamos que se desea obtener un listado de los clientes con los nombres de sus respectivos pueblos, pero sin que se pierda ningún cliente, aunque éste tenga un nulo como código de pueblo. A continuación se muestran dos sentencias con concatenación externa, en primer lugar con sintaxis estándar y después con la tradicional, que resuelven dicho problema y el resultado obtenido:

```
select c.nombre cliente, p.nombre pueblo
from   clientes c left join pueblos p using ( codpue ) ;

select c.nombre cliente, p.nombre pueblo
from   clientes c, pueblos p
where  c.codpue = p.codpue (+);
```

CLIENTE	PUEBLO
Alberto	Cella
Carlos	Cella
Juan	
Pedro	Olocau

Como se puede ver, el cliente Juan ya aparece en el resultado. Como dicho cliente no tiene código de pueblo, su nombre de pueblo toma el valor nulo. Es decir, al realizar la concatenación externa, *es como si* el cliente Juan fuera concatenado a una fila virtual de la tabla pueblos con todos sus valores nulos.

La consulta con la sintaxis tradicional es idéntica a la respectiva sentencia con concatenación interna exceptuando el carácter + entre paréntesis. Estos tres caracteres, (+), deben aparecer en la restricción del producto cartesiano justo al otro lado de la tabla cuyas filas no se quieren perder. Es decir, como no se quiere que se pierda ningún cliente, el texto (+) debe ir en el lado de la tabla pueblos.

8.2.2 Concatenación externa por la derecha: A right join B

Supongamos que se desea obtener un listado de clientes con sus respectivos pueblos, pero sin que se pierda ningún pueblo, aunque no exista ningún cliente en dicho pueblo. A continuación se muestran dos sentencias con concatenación externa, en primer lugar con sintaxis estándar y después con la tradicional, que resuelven dicho problema y el resultado obtenido:

```
select c.nombre cliente, p.nombre pueblo
from   clientes c right join pueblos p using ( codpue ) ;

select c.nombre cliente, p.nombre pueblo
from   clientes c, pueblos p
where  c.codpue (+) = p.codpue ;
```

CLIENTE	PUEBLO
Alberto	Cella
Carlos	Cella
Pedro	Olocau
	Jumilla

Como se puede ver, el pueblo de Jumilla aparece en el resultado pese a que en la tabla de clientes no existe ninguno que resida allí. El nombre del cliente para el pueblo de Jumilla es nulo. Es decir, al realizar la concatenación externa, *es como si* el pueblo de Jumilla fuera concatenado a una fila virtual de la tabla clientes con todos sus valores nulos.

La sentencia con sintaxis tradicional es idéntica a la misma con concatenación interna exceptuando el carácter + entre paréntesis. Estos tres caracteres, (+), deben aparecer en el otro lado de la tabla cuyas filas no se quieren perder. Es decir, como no se quiere que se pierda ningún pueblo, el texto (+) debe ir en el lado de la tabla clientes.

8.2.3 Concatenación externa completa: A full join B

Si se desea que no se pierda ningún cliente ni tampoco ningún pueblo del resultado final, hay que recurrir a la concatenación externa completa. Normalmente este tipo de concatenación externa no es ofrecido por la mayor parte de sistemas que emplean la sintaxis tradicional. A continuación se muestra la sentencia con concatenación externa siguiendo la sintaxis estándar y el resultado obtenido:

```
select c.nombre cliente, p.nombre pueblo
from   clientes c full join pueblos p using ( codpue ) ;
```

CLIENTE	PUEBLO
Alberto	Cella
Carlos	Cella
Juan	
Pedro	Olocau
	Jumilla

8.2.4 Equivalencias y Ejemplos

Hay que decir que **A left join B** es completamente equivalente a **B right join A**. De manera análoga, **A right join B** es equivalente a **B left join A**.

- **Ejercicio**: Escribir una consulta que devuelva el código y fecha de cada factura junto al nombre del vendedor que realizó la factura. No se debe perder ninguna factura aunque no tenga código de vendedor.
  Solución:

```
select f.codfac, f.fecha, v.nombre
from   facturas f left join vendedores v using ( codven );

select f.codfac, f.fecha, v.nombre
from   vendedores v right join facturas f v using ( codven );
```

- **Ejercicio**: Escribir una consulta que devuelva el código de factura, línea, código y descripción para todas las líneas de la factura 100, aunque el código de artículo sea nulo.
  Solución:

```
select l.codfac, l.linea, codart, a.descrip
from   lineas_fac l left join articulos a using ( codart )
where  l.codfac = 100 ;

select l.codfac, l.linea, codart, a.descrip
from   articulos a right join lineas_fac l using ( codart )
where  l.codfac = 100 ;
```

8.3 Concatenación externa y agrupación

Siguiendo con el ejemplo de los clientes y pueblos, imaginemos que se desea mostrar el nombre de cada pueblo y el número de clientes residiendo en cada uno de ellos. Esta consulta puede realizarse con una simple concatenación de dos tablas y una agrupación. A continuación se muestra ésta y el resultado obtenido con las tablas anteriores:

```
select p.nombre pueblo, count( * ) NumCli
from   clientes c join pueblos p using ( codpue )
group  by codpue, p.nombre ;
```

PUEBLO	NUMCLI
Cella	2
Olocau	1

Si se realiza una concatenación interna como la de la sentencia anterior, entonces los pueblos que no tienen ningún cliente no aparecerán. Como se puede ver, el pueblo de Jumilla no aparece en el resultado dado que al realizar la concatenación interna se pierde.

Si en realidad se desea que aparezcan, pero con un cero al lado (pues en Jumilla hay cero clientes), habrá que recurrir a la concatenación externa. A continuación se muestra la sentencia con la concatenación externa y el resultado que obtiene:

```
select p.nombre pueblo, count( * ) NumCli
from   clientes c right join pueblos p using ( codpue )
group  by codpue, p.nombre ;
```

PUEBLO	NUMCLI
Cella	2
Olocau	1
Jumilla	1

¿Es éste el resultado deseado? Obviamente no porque dice que Jumilla tiene un cliente, cuando no es así. Así pues, ¿qué está fallando? Como el pueblo de Jumilla es concatenado con un cliente virtual todo a nulos, la función de grupo **count(*)** devuelve 1. Por tanto, el **count(*)** no sirve y hay que cambiarlo por algún otro operador que distinga un cliente real de un cliente ficticio todo a nulos. A continuación se muestra la sentencia final y el resultado deseado.

```
select p.nombre pueblo, count( c.codcli ) NumCli
from   clientes c right join pueblos p using ( codpue )
group  by codpue, p.nombre ;
```

PUEBLO	NUMCLI
Cella	2
Olocau	1
Jumilla	0

Como se puede ver, el único cambio ha sido sustituir el operador **count(*)** que devuelve 1 en el caso de Jumilla por el operador **count(c.codcli)** que devolverá cero en dicho caso pues el código del cliente para dicho pueblo es nulo.

8.4 Concatenación externa y unión

En la mayor parte de los casos, una concatenación externa puede ser implementada mediante una concatenación interna y una unión de consultas. Por ejemplo, si se desea mostrar el nombre de los clientes y a su lado el nombre del pueblo en el que residen, pero sin que se pierda ningún cliente, habría que recurrir a la siguiente sentencia:

```
select c1.nombre cliente, p.nombre pueblo
from   clientes c1 join pueblos p using ( codpue )
union
select c2.nombre cliente, NULL pueblo
from   clientes c2
where  c2.codpue is null ;
```

Como se puede ver, la primera consulta realiza una concatenación interna, con lo que pierde aquellos clientes cuyo código de pueblo es nulo. La segunda sentencia subsana este problema devolviendo justo aquellos clientes cuyo código de pueblo es nulo. Nótese que el número y tipo de las columnas de ambas consultas deben coincidir. Por ello, la segunda consulta devuelve también dos columnas.

- **Ejercicio**: Escribir una consulta que devuelva el nombre de cada cliente y el de su pueblo, pero sin que se pierda ningún pueblo en el listado. No se debe utilizar la concatenación externa.

  Solución:

  ```
  select c1.nombre cliente, p1.nombre pueblo
  from   clientes c1 join pueblos p1 using ( codpue )
  union
  ( select NULL cliente, p2.nombre pueblo
    from   pueblos p2
    except
    select NULL cliente, p3.nombre pueblo
    from   clientes c2 join pueblos p3 using ( codpue )
  );
  ```

  Explicación: La primera consulta devuelve los clientes y sus pueblos de residencia. La segunda consulta (toda la que está entre paréntesis) devuelve el nombre de aquellos pueblos que no tienen clientes con un nombre de cliente nulo. La solución final es la unión de ambas consultas. La forma de obtener el resultado de la segunda consulta (pueblos donde no hay clientes) es quitando a todos los pueblos aquellos pueblos donde sí hay clientes, lo cual se realiza con dos consultas unidas mediante el operador **except**.

8.5 Concatenación externa de tres o más tablas

La concatenación de tres o más tablas se realiza de manera análoga a la concatenación de dos tablas. Sólo que hay que tener en cuenta que en el caso general de procesamiento de *n* tablas, deben existir *n-1* operaciones de concatenación.

- **Ejercicio**: Escribir una consulta que devuelva el código y fecha de cada factura junto al nombre del vendedor que realizó la factura y al nombre del cliente destino de la factura. No se debe perder ninguna factura aunque no tenga código de vendedor ni código de cliente.

  Solución:

  ```
  select f.codfac, f.fecha, v.nombre vendedor, c.nombre cliente
  from   facturas f left join vendedores v using ( codven )
                    left join clientes c using ( codcli ) ;
  ```

♦ **Ejercicio**: Escribir una consulta que devuelva facturación realizada el primer día de enero en las tres provincias de la Comunidad Valenciana. Las provincias que no tienen facturación deben aparecer en el listado con cantidad nula o cero.
Ayuda: Uso de varias concatenaciones externas para que salgan las tres provincias.
Solución:

```
select pr.codpro, pr.nombre, sum( l.cant * l.precio )
from   lineas_fac l, facturas f, clientes c, pueblos p,
       provincias pr
where  f.codfac = l.codfac (+)
and    c.codcli = f.codcli (+)
and    p.codpue = c.codpue (+)
and    pr.codpro = p.codpro (+)
and    pr.codpro in ( '03', '12', '46' )
and    to_number( to_char(
         nvl( f.fecha, to_date('01-01-2002','dd-mm-yyyy') ),
         'ddd' ) ) = 1
group  by pr.codpro, pr.nombre ;
```

8.6 Ejercicios

♦ **Ejercicio 8.1**: Escribir una consulta que muestre el código y nombre de aquellos clientes con menos de 5 facturas (incluyendo aquéllos con ninguna).
Ayuda: Hay que realizar una concatenación externa de dos tablas y una simple agrupación.
Solución:

```
select codcli, c.nombre, count( f.codfac )
from   clientes c left join facturas f using ( codcli )
group  by codcli, c.nombre
having count( f.codfac ) < 5;
```

♦ **Ejercicio 8.2**: Escribir una consulta que muestre el código, descripción y cantidad de aquellos artículos tales que se han vendido menos de 100 unidades (incluyendo aquéllos que no se han vendido).
Ayuda: Hay que realizar una concatenación externa de dos tablas y una simple agrupación. Una labor adicional es convertir el resultado del operador **sum(l.cant)** al valor cero si vale nulo, principalmente en la cláusula **having**. Ello se debe a que en los artículos sin ventas no se cumple la condición del **having** si no se realiza previamente la conversión (pues de no hacer la conversión se estaría comparando **nulo < 100**, lo cual se evalúa a falso).
Solución:

```
select codart, a.descrip, coalesce( sum( l.cant ), 0 )
from   articulos a left join lineas_fac l using ( codart )
group  by codart, a.descrip
having coalesce( sum( l.cant ), 0 ) < 100 ;
```

♦ **Ejercicio 8.3**: Escribir una consulta que muestre el código y nombre de aquellas provincias con menos de 5 vendedores residiendo en ellas (incluyendo aquéllas que no tienen ningún vendedor).
Ayuda: Hay que realizar una concatenación externa de tres tablas y una agrupación. Si se considera que toda provincia tiene al menos un pueblo, bastaría con una concatenación interna y con una externa.
Solución:

```
select codpro, pr.nombre, count( v.codven )
from   vendedores v right join pueblos p using ( codpue )
                   right join provincias pr using ( codpro )
```

```
group  by codpro, pr.nombre
having count( v.codven ) < 5 ;
```

♦ **Ejercicio 8.4**: Escribir una consulta que para cada factura con un importe inferior a 3 euros, muestre su fecha, el nombre del cliente y su importe.
Ayuda: Uso de dos concatenaciones externas: la primera para que no se pierdan facturas si no tienen código de cliente asignado; la segunda para que no se pierdan facturas si no tienen líneas.
Solución:

```
select f.fecha, c.nombre, sum( l.cant * l.precio ) importe
from   lineas_fac l, facturas f, clientes c
where  f.codfac = l.codfac (+)
and    c.codcli (+) = f.codcli
group  by f.codfac, f.fecha, c.nombre
having nvl( sum( l.cant * l.precio ), 0 ) < 3 ;
```

♦ **Ejercicio 8.5**: Escribir una consulta que muestre el código y nombre de aquellos vendedores que han realizado ventas a menos de 10 clientes.
Ayuda: Hay que realizar una concatenación externa de dos tablas y una agrupación dado que un vendedor puede no tener facturas (acaba de entrar en la empresa) y una factura puede no tener cliente asignado en este momento.
Solución:

```
select codven, v.nombre, count( codcli )
from   vendedores v left join facturas f using ( codven )
group  by codven, v.nombre
having count( f.codcli ) < 10 ;
```

♦ **Ejercicio 8.6**: Escribir una consulta que muestre el código y descripción de aquellos artículos que han sido vendidos en menos de 5 provincias.
Ayuda: Hay que realizar una concatenación externa de 5 tablas y una agrupación.
Solución:

```
select codart, a.descrip, count( distinct p.codpro )
from   articulos a left join lineas_fac l using ( codart )
                  left join facturas f using ( codfac )
                  left join clientes c using ( codcli )
                  left join pueblos p using ( codpue )
group  by codart, a.descrip
having count( distinct p.codpro ) < 5 ;
```

♦ **Ejercicio 8.7**: Escribir una consulta que muestre el código y nombre de aquellos clientes cuyo descuento máximo aplicado en las facturas está por debajo del 5 %. Se deben incluir aquellos clientes cuyo descuento es nulo y también aquellos que no tienen facturas.
Ayuda: Hay que realizar una concatenación externa de dos tablas y una agrupación. La cláusula **having** debe usar la función **coalesce** para incluir aquellos clientes cuyos descuentos son siempre nulos.
Solución:

```
select codcli, c.nombre, max( f.dto )
from   clientes c left join facturas f using ( codcli )
group  by codcli, c.nombre
having coalesce( max( f.dto ), 0 ) < 5 ;
```

♦ **Ejercicio 8.8**: Escribir una consulta que muestre el código y nombre de aquellas provincias en las que se ha facturado en total menos de 6000 euros (a los clientes que residen en ellas). Deben incluirse las provincias sin facturación.

Ayuda: Hay que realizar una concatenación externa de 5 tablas y una agrupación. La cláusula **having** debe usar la función **coalesce** para incluir aquellas provincias cuya facturación es nula.

Solución:

```
select codpro, pr.nombre, sum( l.cant * l.precio )
from   provincias pr left join pueblos p using ( codpro )
                     left join clientes c using ( codpue )
                     left join facturas f using ( codcli )
                     left join lineas_fac l using ( codfac )
group  by codpro, pr.nombre
having coalesce( sum( l.cant * l.precio ), 0 ) < 6000 ;
```

8.7 Autoevaluación

- **Ejercicio 1**: Escribir una consulta que devuelva el código y descripción de aquellos artículos tales que el máximo descuento aplicado en sus ventas (líneas de facturas) es menor del 10 %. Se deben incluir también aquellos artículos cuyo descuento es cero o nulo.
- **Ejercicio 2**: Escribir una consulta que obtenga el código y nombre de aquellos pueblos de Castellón donde se ha facturado a los clientes residentes menos de 1000 euros.
- **Ejercicio 3**: Escribir una consulta que devuelva el código y nombre de aquellos pueblos de la provincia de Castellón cuyo número de clientes residentes sea menor que 5. La consulta debe devolver también el número de clientes en cada pueblo.

9 SUBCONSULTAS

Este capítulo presenta una parte de SQL muy importante: las subconsultas. Una subconsulta no es sino una consulta que aparece dentro de otra consulta. La posibilidad de insertar consultas dentro de otras consultas aumenta notablemente la potencia del lenguaje SQL. Hay que decir que algunos sistemas (en realidad muy pocos) carecen desafortunadamente de esta característica.

9.1 Introducción

Una subconsulta es una sentencia **select** que se utiliza dentro de otra sentencia **select**. La subconsulta obtiene unos resultados intermedios (uno o más datos) que se emplean en la consulta principal. Las subconsultas siempre deben estar escritas entre paréntesis. Las subconsultas pueden aparecer prácticamente en cualquier sitio de la consulta principal, aunque los lugares más habituales son las cláusulas **where** y **having**.

Las subconsultas se pueden clasificar en los siguientes tipos, atendiendo al número de filas y columnas que devuelven:

- Subconsultas que devuelven un único valor (una fila con una única columna).
- Subconsultas que devuelven una única fila o tupla con más de una columna.
- Subconsultas que devuelven un conjunto de filas (es decir, cero, una o varias filas).

Seguidamente se describen con más detalle y se presentan ejemplos de uso de estos tres tipos de subconsultas.

9.2 Subconsultas que devuelven un único valor

Una subconsulta que devuelve un solo valor puede ser usada prácticamente en cualquier lugar de una consulta principal aunque, como ya se ha dicho, los lugares más habituales son las cláusulas **where** y **having**. Aunque devuelva un solo valor, la subconsulta debe ser escrita entre paréntesis, como se ha comentado anteriormente. El valor devuelto por la subconsulta puede ser usado como un valor normal en una expresión, en una comparación, etc.

Una forma de usar la subconsulta es la siguiente: **expresión operador (subconsulta)**, donde el predicado se evalúa a verdadero si la comparación indicada por el operador (=, <>, >, <, >=, <=) entre el resultado de la expresión y el de la subconsulta es verdadero. En este predicado la subconsulta debe devolver un solo valor (una fila con una columna). Si la subconsulta devuelve más de un valor (una columna con varias filas o más de una columna), se produce un error de ejecución.

♦ **Ejercicio**: Escribir una consulta que devuelva el código y descripción del artículo más caro.
Solución:

```
select codart, descrip
from   articulos
where  precio = ( select max( precio )
                  from   articulos ) ;
```

Explicación: La subconsulta devuelve el precio más caro de todos los artículos. La consulta principal muestra la información solicitada de aquellos artículos cuyo precio es igual al precio obtenido en la subconsulta (el más caro).

♦ **Ejercicio**: Escribir una consulta que devuelva el código y descripción de aquellos artículos cuyo precio supera la media.

Solución:

```
select codart, descrip
from   articulos
where  precio > ( select avg( precio )
                  from   articulos ) ;
```

Explicación: La subconsulta obtiene el precio medio de los artículos. La consulta principal muestra aquellos artículos cuyo precio supera dicho precio medio.

- **Ejercicio**: Escribir una consulta que devuelva el código y nombre de aquellos clientes cuyo número de facturas es menor que la mitad del mayor número de facturas de un cliente.

  Solución:

```
select codcli, c.nombre
from   facturas f join clientes c using ( codcli )
group  by codcli, c.nombre
having count( * ) < ( select 0.5 * max( count( * ) )
                      from   facturas f2
                      group  by f2.codcli ) ;
```

  Explicación: La subconsulta calcula la mitad del mayor número de facturas de un cliente de la base de datos. La consulta principal muestra el código y nombre de aquellos clientes cuyo número de facturas es inferior al valor calculado por la subconsulta.

- **Ejercicio**: Escribir una consulta que devuelva el código, descripción y número total de unidades vendidas para cada artículo empleando una subconsulta en la cláusula **select**.

  Solución:

```
select a.codart, a.descrip,
       ( select sum( l.cant )
         from   lineas_fac l
         where  l.codart = a.codart ) SumCant
from   articulos a ;
```

  Explicación: En la sentencia anterior la subconsulta se encuentra en la misma cláusula **select**. La subconsulta devuelve la suma de unidades vendidas en las líneas de facturas para cada artículo. Aunque esta sentencia es correcta, requiere la ejecución de una subconsulta para cada artículo mostrado. En un determinado sistema la ejecución de la sentencia anterior cuesta el doble de tiempo que la siguiente sentencia, la cual usa una concatenación externa y una simple agrupación:

```
select codart, a.descrip, sum( l.cant ) SumCant
from   articulos a left join lineas_fac l using ( codart )
group  by codart, a.descrip ;
```

9.3 Subconsultas que devuelven una única fila

En este apartado se van a estudiar las subconsultas que devuelven una única fila con más de una columna. Si una subconsulta devuelve una única fila con una única columna, entonces se considera que devuelve un único valor. Este caso ha sido descrito en profundidad en el apartado anterior.

Una subconsulta que devuelve una única fila con más de una columna es usada habitualmente en predicados (en el **where** o en el **having**). Su forma de uso es la siguiente: **(expr1, expr2, ...) operador (subconsulta)**. En este caso la subconsulta debe devolver una sola fila y tantas columnas como las existentes entre paréntesis a la izquierda del operador. Es decir, el número de las columnas a ambos lados del operador debe coincidir. Las expresiones de la izquierda **expr1**, **expr2**,... se evalúan y la fila que forman se compara, utilizando un operador, con la fila que devuelve la subconsulta.

En la versión actual de la mayor parte de sistemas sólo se pueden utilizar los operadores = y <>. El predicado se evalúa a verdadero si el resultado de la comparación es verdadero para la fila devuelta por la subconsulta. En caso contrario se evalúa a falso.

Si la subconsulta no devuelve ninguna fila, se evalúa a nulo o desconocido (ambos términos son sinónimos).

Hay que tener en cuenta que una restricción, tanto en la cláusula **where** como en la **having**, se cumple si el resultado de su predicado es verdadero; si el predicado es falso o nulo, se considera que la restricción no se cumple.

Cuando se comparan dos filas, los atributos se comparan uno a uno según su posición en la fila: el primer atributo de la primera fila con el primer atributo de la segunda fila, el segundo atributo de la primera fila con el segundo atributo de la segunda fila etc.

Cuando no hay ningún valor nulo en ninguno de los atributos de ambas filas, comparar dos filas para decidir si son iguales o distintas es muy fácil: hasta un niño podría hacerlo. Si no existen valores nulos, el resultado de una comparación para decidir si dos filas son iguales o distintas sólo puede ser cierto o falso.

Cuando existe algún valor nulo, entonces el asunto se complica un poco ya que el valor nulo es un valor desconocido. Cuando existe algún valor nulo en una o en ambas filas, el resultado de una comparación para decidir si dos filas son iguales o distintas puede ser cierto, falso o desconocido (nulo).

Para decidir si dos filas son iguales o distintas se suele aplicar el método siguiente:

- Dos filas se consideran iguales si todos sus atributos no son nulos y son iguales uno a uno en ambas filas.
- Si existe al menos un valor nulo en una de las filas, éstas ya no pueden ser iguales, en todo caso pueden ser distintas o se puede obtener como resultado el valor desconocido.
- Dos filas se consideran distintas si al menos un atributo correspondiente de ambas filas es distinto y con valores no nulos en dicho atributo de ambas filas.
- En cualquier otro caso, el resultado del predicado es desconocido (nulo).

Si la subconsulta devuelve más de una fila, se produce un error de ejecución.

Cuando no existen valores nulos, el comportamiento antes descrito es el lógico y esperable. Cuando sí existen, el comportamiento también es el lógico si se tiene en cuenta que el valor nulo equivale a un valor desconocido. Por ejemplo, sabiendo que el valor nulo es un valor desconocido, la expresión **(4) = (null)** da como resultado el valor desconocido ya que no se puede decir que la expresión sea cierta ni que sea falsa puesto que la parte derecha tiene un valor desconocido, por lo que podría ser un 4 o no serlo.

♦ **Ejercicio**: Evaluar los siguientes predicados (determinar si devuelven cierto, falso o nulo).

```
1. ( 6, 1 ) = ( 6, 1 )
2. ( 6, 1 ) = ( 6, 5 )
3. ( 6, 1 ) = ( 6, null )
4. ( 6, null ) = ( 1, null )
5. ( 6, null ) = ( 6, null )
6. ( null, null ) = ( null, null )
7. ( 6, 1 ) <> ( 6, 1 )
8. ( 6, 1 ) <> ( 6, 5 )
9. ( 6, 1 ) <> ( 6, null )
10. ( 6, null ) <> ( 1, null )
11. ( 6, null ) <> ( 6, null )
12. ( null, null ) <> ( null, null )
13. ( null, null ) = ( subconsulta vacía )
14. ( null, null ) <> ( subconsulta vacía )
```

```
15. ( 6, 1 ) = ( subconsulta vacía )
16. ( 6, 1 ) <> ( subconsulta vacía )
```

Solución: T es cierto, F es falso y null es desconocido. Los resultados son los siguientes: 1. T, 2. F, 3. null, 4. F, 5. null, 6. null, 7. F, 8. T, 9. null, 10. T, 11. null, 12. null, 13. null, 14. null, 15. null y 16 null.

♦ **Ejercicio**: Escribir una consulta que devuelva el código y fecha de aquellas facturas para las cuales tanto su descuento como su iva son iguales a los máximos. Escribir en primer lugar una sentencia utilizando subconsultas que devuelvan un solo valor. Escribir en segundo lugar una sentencia que emplee subconsultas que devuelvan una tupla.
Solución:

```
select codfac, fecha
from   facturas
where  dto = ( select max( dto ) from facturas )
and    iva = ( select max( iva ) from facturas ) ;

select codfac, fecha
from   facturas
where  ( dto, iva ) = ( select max( dto ), max( iva )
                        from   facturas ) ;
```

Explicación: En el caso de la subconsulta que devuelve una fila, ésta calcula el máximo descuento y el máximo iva aplicado en las facturas. La consulta principal muestra el código y fecha de aquellas facturas para las cuales tanto el iva como el descuento son iguales a los máximos.

9.4 Subconsultas que devuelven un conjunto de filas

Si la subconsulta devuelve un conjunto de valores (cero, uno o más), entonces no se pueden comparar con un comparador tradicional (<, <=, =, etc.) debido a que estos operadores sólo sirven para comparar un valor con otro, pero no un valor con muchos. Por tanto, hay que recurrir a operadores especiales.

La consulta principal y la subconsulta que devuelve un conjunto de valores pueden unirse mediante diversos operadores, entre los cuales destacan los siguientes:

- Operador **in**. Este operador comprueba la pertenencia a un conjunto. La operación **a in b** devuelve cierto si **a** pertenece al conjunto **b** y falso en caso contrario. Obviamente la subconsulta debe ir en el lugar de **b**.
- Operador **exists**. La operación **exists(b)** devuelve cierto si el conjunto **b** contiene al menos un elemento (una fila).
- Operador **all**. La operación **a >= all(b)** devuelve cierto si **a** es mayor o igual que todos los elementos del conjunto **b**. Se puede emplear cualquier otro operador de comparación en lugar del antes mostrado.
- Operador **any**. La operación **a > any(b)** devuelve cierto si **a** es mayor que alguno de los elementos del conjunto **b**. Se puede emplear cualquier otro operador de comparación en lugar del antes mostrado.

Todos los operadores menos el operador **exists** admiten dos formas de funcionar: comparación de valores y comparación de filas. En la comparación de valores se pueden emplear los comparadores tradicionales: **=,<>,<,<=,>,>=**. En la comparación de filas sólo se pueden emplear los comparadores de igualdad y desigualdad: = y <>.

Enseguida se describen con más detalle y se muestran diversos ejemplos de su uso.

9.4.1 Operador in

En capítulos anteriores se ha descrito cómo el operador **in** se puede emplear para determinar si un elemento pertenece a un conjunto dado de valores. Ahora se va a ver cómo se puede usar una subconsulta en lugar de un conjunto dado de valores.

Este operador tiene dos formas habituales de uso: **expresión in (subconsulta)** y **(expr1, expr2,...) in (subconsulta)**. En el primer caso el operador determina si un valor pertenece a un conjunto de valores. En el segundo caso el operador determina si una tupla pertenece a un conjunto de tuplas. A continuación se describen ambos casos con más detalle.

Se comienza con el primer caso. El predicado **expresión in (subconsulta)** se evalúa a verdadero si el resultado de la expresión es igual a alguno de los valores de la columna devuelta por la subconsulta. El predicado se evalúa a falso si no se encuentra ningún valor en la subconsulta que sea igual a la expresión. Cuando la subconsulta no devuelve ninguna fila, también se evalúa a falso. Si el resultado de la expresión es un nulo (la parte izquierda del operador **in**), el predicado se evalúa a nulo. Si ninguno de los valores de la subconsulta es igual a la expresión y la subconsulta ha devuelto algún nulo, el predicado se evalúa a nulo.

Al igual que ocurría en el apartado anterior, cuando no existen valores nulos, este comportamiento es el lógico y esperable. Cuando sí existen, el comportamiento también es el lógico si se tiene en cuenta que el valor nulo equivale a un valor desconocido. Por ejemplo, partiendo de este punto está claro que la expresión **20 in (null)** da como resultado desconocido puesto que no se puede decir si el valor 20 está o no está en la parte derecha pues la parte derecha tiene un valor desconocido.

- **Ejercicio**: Evaluar los siguientes predicados (determinar si devuelven cierto, falso o nulo).

```
1. 20 in ( 10, 20, 30 )
2. 20 in ( 10, 19, 30 )
3. 20 in ( 10, 30, null )
4. 20 in ( 10, 20, null )
5. 20 in ( subconsulta vacía )
6. null in ( 10, 20, 30 )
7. null in ( 10, null, 30 )
```

  Solución: T es cierto, F es falso y null es desconocido. Los resultados son los siguientes: 1. T, 2. F, 3. null, 4. T, 5. F, 6. null, 7. null.

- **Ejercicio**: Código y nombre de aquellos pueblos donde hay algún cliente (sin emplear la concatenación).
  Solución:

```
select p.codpue, p.nombre
from   pueblos p
where  p.codpue in ( select c.codpue
                     from   clientes c ) ;
```

  Explicación: La subconsulta devuelve todos aquellos códigos de pueblos donde hay clientes. La consulta principal muestra cada pueblo si se halla entre los de los clientes. Reescribe la anterior sentencia utilizando una concatenación interna.

- **Ejercicio**: Código y nombre de aquellos vendedores que han realizado alguna factura con un iva del 16 %. Emplear una subconsulta y el operador **in**.
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  v.codven in ( select f.codven
```

```
from   facturas f
where  f.iva = 16 ) ;
```

Explicación: La subconsulta obtiene el código de aquellos vendedores que en alguna de sus facturas han aplicado un 16 % de iva. La consulta principal se limita a mostrar el código y nombre de aquellos vendedores cuyos códigos se hallan entre los códigos devueltos por la subconsulta.

La segunda forma de usar el operador **in** es la siguiente: **(expr1, expr2,...) in (subconsulta)**. En este predicado la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis en la parte izquierda de este predicado. Las expresiones de la izquierda **expr1**, **expr2**,... se evalúan y la fila que forman se compara con las filas de la subconsulta, una a una. El predicado se evalúa a verdadero si se encuentra alguna fila igual en la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila). Para determinar si dos filas son iguales o distintas, véase los apartados anteriores de este mismo capítulo.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador **in**, el predicado se evalúa a nulo.

- **Ejercicio**: Código y fecha de aquellas facturas en las que se ha comprado un mismo producto que en la factura 282 y en la misma cantidad.
  Solución:

```
select distinct f.codfac, f.fecha
from   lineas_fac l1 join facturas f using ( codfac )
where  f.codfac <> 282
and    ( l1.codart, l1.cant ) in ( select l2.codart, l2.cant
                                   from   lineas_fac l2
                                   where  l2.codfac = 282 );
```

9.4.2 Operador not in

Al igual que el anterior, este operador también tiene dos modos de uso: **expresión not in (subconsulta)** y **(expr1, expr2,...) not in (subconsulta)**. Se comienza describiendo el primero.

El predicado **expresión not in (subconsulta)** se evalúa a verdadero si la expresión es distinta de todos los valores de la columna devuelta por la subconsulta. También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila (¡mucho ojo!). Si se encuentra algún valor igual a la expresión, se evalúa a falso.

Si el resultado de la expresión es un nulo, el predicado se evalúa a nulo. Si la subconsulta devuelve algún nulo y todos los demás valores son distintos a la expresión, el predicado se evalúa a nulo.

- **Ejercicio**: Código y nombre de aquellos vendedores que no tienen ninguna factura. Emplear una subconsulta y el operador **not in**.
  Solución: (Esta solución no es del todo correcta)

```
select v.codven, v.nombre
from   vendedores v
where  v.codven not in ( select f.codven
                         from   facturas f ) ;
```

Como se puede ver, la subconsulta obtiene el código de todos aquellos vendedores que tienen al menos una factura. La consulta principal se limita a mostrar el código y nombre de aquellos vendedores cuyos códigos no se encuentran entre los códigos de la subconsulta.

La sentencia anterior tiene un único punto débil: cuando existen facturas cuyo código de vendedor es nulo. Vamos a verlo con las tablas rellenadas de la forma siguiente. Con estas tablas, el único vendedor que no tiene facturas es el vendedor con código 27 llamado Carlos.

Tabla FACTURAS

CODFAC	IVA	CODVEN
101	16	25
102	16	
103	7	26

Tabla VENDEDORES

CODVEN	NOMBRE
25	Juan
26	Pedro
27	Carlos

Con los datos anteriores, la subconsulta que devuelve los códigos de vendedores de las facturas obtendrá el siguiente conjunto: **(25, null, 26)**. Por tanto, la consulta principal comprueba si cada vendedor no se encuentra entre los anteriores con la expresión **codven not in (25, null, 26)**. Esta expresión será falsa para los vendedores 25 (Juan) y 26 (Pedro). Para el vendedor 27 (Carlos) esta expresión se evalúa a nulo, con lo cual para este vendedor tampoco se cumple la restricción y no aparece. Para el vendedor Carlos esta expresión, **27 not in (25, null, 26)**, se evalúa a nulo dado que no se puede decir ni que el vendedor esté entre los vendedores especificados ni que no esté dado que no aparece pero hay un valor nulo.

Una forma de evitar este comportamiento es insertar una restricción en la cláusula **where** de la subconsulta para eliminar los molestos valores nulos. La siguiente solución obtiene el resultado pedido incluso con la presencia de valores nulos y, en el caso de las tablas anteriores, devolverá el resultado esperado: 27, Carlos. Otra forma de resolver el problema de los valores nulos es usando la función **coalesce** para traducir el valor nulo por otro valor.

```
select v.codven, v.nombre
from   vendedores v
where  v.codven not in ( select f.codven
                         from   facturas f
                         where  f.codven is not null ) ;
```

- **Ejercicio**: Código y nombre de aquellos vendedores que no han realizado ninguna factura con un iva del 16 %. Emplear una subconsulta y el operador **not in**.
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  v.codven not in ( select f.codven
                         from   facturas f
                         where  f.iva = 16
                         and    f.codven is not null ) ;
```

- **Ejercicio**: Número de clientes que no tienen facturas.
  Solución:

```
select count( * )
from   clientes c
where  c.codcli not in ( select f.codcli
                         from   f.facturas
                         where  f.codcli is not null );
```

  Explicación: Nótese que en el ejemplo se ha incluido la restricción **codcli is not null** en la subconsulta porque la columna **codcli** de la tabla facturas acepta nulos, por lo

que podría haber alguno y eso haría que el predicado **not in** se evaluara a nulo para todos los clientes de la consulta principal.

El predicado **(expr1, expr2,...) not in (subconsulta)** se evalúa a verdadero si no se encuentra ninguna fila igual en la subconsulta. También se evalúa a verdadero si la subconsulta no devuelve ninguna fila (¡mucho ojo!). Si se encuentra alguna fila igual, se evalúa a falso.

La subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador **not in**. Las expresiones de la izquierda **expr1**, **expr2**,... se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador **not in**, el predicado se evalúa a nulo.

- **Ejercicio**: Códigos de aquellos clientes que no tienen facturas con iva y descuento como los de los clientes cuyos códigos varían entre 171 y 174, ambos inclusive.
  Solución:

```
select distinct codcli
from   facturas
where  ( coalesce( iva, 0 ), coalesce( dto, 0 ) )
       not in ( select coalesce( iva, 0 ), coalesce( dto, 0 )
                from   facturas
                where  codcli between 171 and 174 );
```

9.4.3 Operador any

La palabra **some** es sinónimo de **any**. Este operador también tiene dos formas de uso, dependiendo del número de columnas en la subconsulta. Se comienza por describir la versión para una sola columna.

Un uso de este operador es el siguiente: **expresión operador any (subconsulta)**. En este uso de **any** la subconsulta debe devolver una sola columna. El operador debe ser una comparación (=, <>, >, <, >=, <=).

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para alguno de los valores de la columna devuelta por la subconsulta. En caso contrario se evalúa a falso.

Si la subconsulta no devuelve ninguna fila, el predicado devuelve falso. Si ninguno de los valores de la subconsulta coincide con la expresión de la izquierda del operador y en la subconsulta se ha devuelto algún nulo, se evalúa a nulo.

El operador **in** es equivalente a **= any**.

- **Ejercicio**: Facturas con descuentos como los de las facturas sin iva.
  Solución:

```
select *
from   facturas
where  coalesce( dto, 0 ) =
         any( select coalesce( dto, 0 )
              from   facturas
              where  coalesce( iva, 0 ) = 0 );
```

Otro posible uso de este operador es el siguiente: **(expr1, expr2,...) operador any (subconsulta)**. En este uso de **any** la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador. Las expresiones de la izquierda **expr1**, **expr2**,... se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila. En la versión actual de la mayor parte de sistemas sólo se pueden utilizar los operadores = y <>. Véase en apartados anteriores todo lo relativo a la comparación de filas.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para alguna de las filas devueltas por la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila).

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser falso (será verdadero o nulo).

9.4.4 Operador all

Este operador también tiene dos formas de uso, dependiendo del número de columnas en la subconsulta. Se comienza por describir la versión para una sola columna.

El primer uso de este operador es el siguiente: **expresión operador all (subconsulta)**. En este uso la subconsulta debe devolver una sola columna. El operador debe ser una comparación (=, <>, >, <, >=, <=).

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todos los valores de la columna devuelta por la subconsulta. También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila (¡mucho ojo!). En caso contrario se evalúa a falso. Si la subconsulta devuelve algún nulo, el predicado se evalúa a nulo.

Vamos a ver un ejemplo. Supongamos que se desea mostrar el código y descripción de los artículos con mayor precio (sin recurrir a la función de columna **max**). Una solución sería la siguiente:

```
select a.codart, a.descrip
from   articulos a
where  a.precio >= all( select a2.precio
                        from   articulos a2 ) ;
```

Vamos a verlo con unos datos concretos. Supongamos la tabla de artículos que se muestra a continuación. La anterior sentencia irá evaluando cada artículo y lo mostrará si su precio es mayor o igual que todos los precios de la misma tabla. En este caso, el único artículo cuyo precio es mayor o igual que todos los precios de los artículos es aquél con código A3.

Tabla ARTICULOS

CODART	PRECIO
A1	4.00
A2	2.00
A3	5.00

La anterior sentencia funciona correctamente y devuelve el artículo o artículos cuyo precio es igual al máximo. Únicamente hay que tener en cuenta que si al menos un artículo tiene precio nulo, entonces no mostrará ningún artículo dado que el operador **all** devolverá desconocido para todos los artículos. Como la columna precio de la tabla artículos acepta nulos, ésta no es una situación excepcional.

Supongamos ahora la tabla de artículos que se muestra a continuación.

Tabla ARTICULOS

CODART	PRECIO
A1	4.00
A2	
A3	5.00

Con la anterior tabla, cuando se evalúa un artículo se realiza la siguiente comparación: **precio >= all(4.00, null, 5.00)**. Dado que hay un nulo, esta comparación devuelve siempre desconocido, con lo que la restricción no se cumple y ningún artículo saldrá en el resultado. De hecho, este es el resultado lógico si se considera que el valor nulo es el valor desconocido dado que no se puede decir qué artículo tiene mayor precio si uno de ellos es nulo o desconocido.

Probablemente éste no es el comportamiento deseado, sino que se quiere obtener el artículo o artículos cuyo precio es igual al máximo sin tener en cuenta los artículos con precio nulo.

La siguiente sentencia, que devuelve el artículo o artículos con mayor precio sin tener en cuenta los nulos, es muy parecida a la anteriormente propuesta. Hay que añadir una restricción en la cláusula **where** para eliminar los valores nulos en el resultado de la subconsulta. Otra posible solución sería convertir los valores nulos de la subconsulta en el precio 0.00 con la ayuda del operador **coalesce**. Seguidamente se muestra la solución con la restricción:

```
select a.codart, a.descrip
from   articulos a
where  a.precio >= all( select a2.precio
                        from   articulos a2
                        where  a2.precio is not null ) ;
```

Otro aspecto en el que hay que tener mucho cuidado con este operador es en si las subconsulta devuelve algún resultado o no. Si la subconsulta no devuelve ningún resultado, el operador **all** devuelve cierto, lo cual puede dar lugar a cierta confusión.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia con los datos anteriores?

```
select a.codart, a.descrip
from   articulos a
where  a.precio >= all( select a2.precio
                        from   articulos a2
                        where  a2.precio is not null
                        and    a2.codart like 'A6%' ) ;
```

  Solución: Trata de mostrar aquellos artículos cuyos precios superan a todos los precios de los artículos cuyo código comienza por A6. Pero como no hay ningún artículo con dicho código y con precio conocido, muestra todos los artículos pues la subconsulta no devuelve ninguna fila y, por tanto, el operador devuelve cierto.

- **Ejercicio**: Datos de las facturas en las que se ha aplicado el máximo descuento (sin utilizar la función de columna **max**).

  Solución:

```
select *
from   facturas
where  coalesce( dto, 0 ) >= all( select coalesce( dto, 0 )
                                  from   facturas );

select *
from   facturas
where  coalesce( dto, 0 ) >= all( select dto
                                  from   facturas
                                  where  dto is not null );
```

  Explicación: Si en la primera sentencia la subconsulta no utiliza la función **coalesce** para convertir los descuentos nulos en descuentos cero, la consulta principal no devuelve ninguna fila si hay nulos en el resultado de la subconsulta, dado que el predicado se evalúa a nulo.

El operador **not in** es equivalente a **<> all**.

El segundo uso de este operador es el siguiente: (**expr1, expr2,...) operador all (subconsulta)**. En este uso la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador.

Las expresiones de la izquierda **expr1**, **expr2**,... se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

En la versión actual de la mayor parte de sistemas sólo se pueden utilizar los operadores = y <>.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todas las filas devueltas por la subconsulta; cuando la subconsulta no devuelve ninguna fila también se evalúa a verdadero (¡mucho ojo!). En caso contrario se evalúa a falso.

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser verdadero (será falso o nulo).

- **Ejercicio**: Datos del cliente 162 si siempre ha comprado sin descuento y con 16 % de iva.

  Solución:

```
select *
from   clientes c
where  c.codcli = 162
and    ( 16, 0 ) = all (
         select coalesce( f.iva, 0 ), coalesce( f.dto, 0 )
         from   facturas f
         where  f.codcli = 162 ) ;
```

Cuando se emplean subconsultas en predicados, el SGBD no obtiene el resultado completo de la subconsulta, a menos que sea necesario. Lo que hace es ir obteniendo filas de la subconsulta hasta que es capaz de determinar si el predicado es verdadero.

9.4.5 Referencias externas

Hasta ahora, las subconsultas se han tratado de modo independiente y, para comprender mejor el funcionamiento de la sentencia, se ha supuesto que la subconsulta se ejecuta en primer lugar, sustituyéndose ésta en la cosulta principal por su valor.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select *
from   facturas
where  coalesce( dto, 0 ) = ( select max( dto )
                              from   facturas );
```

  Solución: En primer lugar la subconsulta obtiene el descuento máximo de las facturas, se sustituye la subconsulta por este valor y, por último, se ejecuta la consulta principal. El resultado final son los datos de aquellas facturas en las que se ha aplicado el mayor descuento.

En ocasiones sucede que la subconsulta se debe recalcular una vez para cada fila de la consulta principal, estando la subconsulta parametrizada mediante valores de columnas de la consulta principal. A este tipo de subconsultas se les llama subconsultas correlacionadas y a los parámetros de la subconsulta que pertenecen a la consulta principal se les llama referencias externas.

- **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select *
from   facturas f
where  0 < ( select min( coalesce( l.dto, 0 ) )
             from   lineas_fac l
             where  l.codfac = f.codfac ) ;
```

  Solución: Obtiene los datos de las facturas que tienen descuento en todas sus líneas:

En este caso existe una referencia externa y ésta es **f.codfac** ya que hace referencia a una columna de la consulta principal. Se puede suponer que la consulta se ejecuta del siguiente modo. Se recorre, fila a fila, la tabla de facturas. Para cada fila se ejecuta la subconsulta, sustituyendo **f.codfac** por el valor que tiene en la fila actual de la consulta principal. Es decir, para cada factura se obtiene el descuento mínimo en sus líneas. Si este descuento mínimo es mayor que cero, significa que la factura tiene descuento en todas sus líneas, por lo que se muestra en el resultado. Si no es así, la factura no se muestra. En cualquiera de los dos casos, se continúa procesando la siguiente factura: se obtienen sus líneas y el descuento mínimo en ellas, etc.

- **Ejercicio**: Escribir una sentencia que muestre el número de clientes a los que en sus facturas siempre se les ha aplicado un iva del 16 % y sin descuento.
  Solución:

```
select count( * )
from   clientes c
where  ( 16, 0 ) = all ( select coalesce( iva, 0 ),
                                coalesce( dto, 0 )
                         from   facturas f
                         where  f.codcli = c.codcli );
```

9.4.6 Operador exists

Los operadores **exists** y su contrario **not exists** suelen necesitar emplear las referencias externas, de ahí que hasta ahora no se hayan presentado. Ambos operadores pueden ser usados tanto en la cláusula **where** como en la cláusula **having**.

El operador **exists(subconsulta)** devuelve verdadero si la subconsulta devuelve al menos una fila. En caso contrario, es decir, si la subconsulta no devuelve ninguna fila, entonces el operador devuelve falso.

La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

Nótese que este operador no necesita terminar de ejecutar completamente la subconsulta, pues en cuanto se encuentre una fila puede devolver verdadero, sin terminar de obtener el resto de las filas.

Como el operador **exists** únicamente pretende determinar si existe al menos una fila o no, la cláusula **select** de la subconsulta no tiene mucha importancia. De hecho, si ponemos **select** * y la tabla empleada tiene muchas columnas, la ejecución se ralentizará devolviendo muchas columnas que en realidad no sirven para nada pues únicamente se quiere saber si existe al menos una fila. Por tanto, para acelerar la ejecución en la subconsulta suele escribirse una constante, como por ejemplo **select '*'**, para que devuelva un valor que ocupe poco espacio y no todas sus columnas.

- **Ejercicio**: Mostrar el código y nombre de aquellos pueblos donde hay clientes.
  Solución:

```
select p.codpue, p.nombre
from   pueblos p
where  exists( select '*'
               from   clientes c
               where  c.codpue = p.codpue ) ;
```

- **Ejercicio**: Mostrar el código y nombre de aquellos vendedores que tienen facturas con iva 16 (empleando una subconsulta y el operador **exists**).
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  exists( select '*'
```

```
                from   facturas f
                where  f.iva = 16
                and    f.codven = v.codven ) ;
```

Explicación: La subconsulta devuelve todas aquellas facturas con iva 16 para un determinado vendedor. La consulta principal recorre todos los vendedores y deja aquéllos para los cuales existen facturas con iva 16.

- **Ejercicio**: Mostrar el código y descripción de aquellos artículos vendidos alguna vez.
  Solución:

```
select a.codart, a.descrip
from   articulos a
where  exists( select '*'
               from   lineas_fac l
               where  l.codart = a.codart ) ;
```

9.4.7 Operador not exists

El operador **not exists(subconsulta)** devuelve falso si la subconsulta retorna al menos una fila y devuelve cierto si la subconsulta no retorna ninguna fila.

La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

En la ejecución de la subconsulta, en cuanto se devuelve la primera fila, se devuelve falso, sin terminar de obtener el resto de las filas.

Puesto que el resultado de la subconsulta carece de interés (sólo importa si se devuelve o no alguna fila), se suele escribir las consultas indicando una constante en la cláusula **select** en lugar de * o cualquier columna:

- **Ejercicio**: Mostrar el código y nombre de aquellos pueblos donde no hay clientes.
  Solución:

```
select p.codpue, p.nombre
from   pueblos p
where  not exists( select '*'
                   from   clientes c
                   where  c.codpue = p.codpue ) ;
```

- **Ejercicio**: Mostrar el código y nombre de aquellos vendedores que no tienen facturas con iva del 16 %.
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  not exists( select *
                   from   facturas f
                   where  f.iva = 16
                   and    f.codven = v.codven ) ;
```

  Explicación: La subconsulta devuelve todas aquellas facturas con iva 16 para un determinado vendedor. La consulta principal recorre todos los vendedores y deja aquellos para los cuales **no** existen facturas con iva 16. En este caso no hay problemas con el valor nulo.

9.5 Subconsultas en la cláusula from

Es posible incluir subconsultas en la cláusula **from**. En este caso no se utilizan para construir predicados, sino para extraer información de otras tablas. El resultado de la subconsulta se ve como otra tabla más de la que extraer información y usar en la consulta principal.

Existen diversas variantes en el uso de subconsultas dentro de esta cláusula. En PostgreSQL se debe dar un nombre a la tabla resultado mediante la cláusula **as**. En cambio, en Oracle no hay que emplear esta palabra.

♦ **Ejercicio**: ¿Qué realiza la siguiente sentencia?

```
select count( * ), max( ivat ), max( dtot )
from   ( select distinct coalesce( iva, 0 ) as ivat,
                         coalesce( dto, 0 ) as dtot
         from   facturas ) t ;
```

Solución: Cuenta las distintas combinaciones de iva y descuento y muestra el valor máximo de éstos. Nótese que que se han renombrado las columnas de la subconsulta para poder referenciarlas en la consulta principal. Esta consulta no la podemos resolver si no es de este modo ya que **count** no acepta una lista de columnas como argumento. Es conveniente renombrar las columnas de la subconsulta que son resultados de expresiones, para poder hacerles referencia en la consulta principal.

En capítulos anteriores se han usado funciones de columna sobre el resultado de funciones de grupo. Por ejemplo, el cálculo del mayor número de facturas realizado por un cliente puede realizarse de este modo: una función de grupo cuenta el número de facturas para cada cliente y la función de columna **max** devuelve el mayor valor de entre los obtenidos.

Desgraciadamente, algunos SGBD no permiten emplear funciones de columna sobre funciones de grupo. En este tipo de sistemas, problemas como el anterior se pueden resolver mediante el uso de subconsultas en la cláusula **from**.

♦ **Ejercicio**: Escribir una consulta que devuelva el número máximo de facturas que ha realizado un cliente. En primer lugar empléese una función de columna y una función de grupo; en segundo, una subconsulta en la cláusula **from**.

Solución:

```
select max( count( * ) )
from   facturas
group  by codcli ;
```

```
select max( NumFactPorCliente )
from   ( select count( * ) as NumFactPorCliente
         from   facturas
         group  by codcli ) as NumFacturas ;
```

♦ **Ejercicio**: Escribir una consulta (con una subconsulta en la cláusula **from**) que devuelva el mayor descuento medio aplicado en las facturas de un cliente.

Solución:

```
select max( dto_med )
from   ( select avg( dto ) as dto_med
         from   facturas
         group  by codcli );
```

♦ **Ejercicio**: Escribir una consulta que obtenga el código de las facturas en las que se ha comprado el artículo que actualmente es el más caro. Escribir en primer lugar una sentencia habitual que use subconsultas en la cláusula **where** y en segundo, una sentencia que use subconsultas en la cláusula **from**.

Solución:

```
select distinct codfac
from   lineas_fac l
where  l.codart in ( select codart
                     from   articulos
                     where  precio = ( select max( precio )
                                       from   articulos ) ) ;
```

```
select distinct l.codfac
from   lineas_fac l join articulos a using ( codart )
                    join ( select max( precio ) as precio
                           from   articulos ) t
                         on ( a.precio = t.precio );
```

9.6 Equivalencia de subconsulta y concatenación interna

En muchas ocasiones las consultas con subconsultas se pueden escribir como consultas multitabla con concatenación interna y viceversa.

- **Ejercicio**: Escribir una consulta que devuelva el código y nombre de aquellos vendedores que tienen al menos una factura con iva 16. Escribirla en primer lugar usando una subconsulta y en segundo, con una concatenación.
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  v.codven in ( select f.codven
                     from   facturas f
                     where  f.iva = 16 ) ;
```

```
select distinct codven, v.nombre
from   facturas f join vendedores v using ( codven )
where  f.iva = 16 ;
```

  Explicación: En el caso de la consulta multitabla, al realizar la concatenación interna de los vendedores con las facturas y restringir éstas a aquéllas con iva 16, todo vendedor que no tenga ninguna factura con iva 16 desaparecerá del resultado final, consiguiéndose así resolver el problema planteado.

9.7 Equivalencia de subconsulta y concatenación externa

En muchas ocasiones las consultas con subconsultas se pueden escribir como consultas multitabla con concatenación externa y viceversa.

- **Ejercicio**: Escribir una consulta que devuelva el código y nombre de aquellos clientes que no tienen facturas. Escribirla en primer lugar usando una subconsulta y en segundo, con una concatenación.
  Solución:

```
select c.codcodcli, c.nombre
from   clientes c
where  not exists( select '*'
                   from   facturas f
                   where  f.codcli = c.codcli );
```

```
select codcli, c.nombre
from   clientes c left join facturas f using ( codcli )
where  f.codfac is null ;
```

  Explicación: En el caso de la consulta multitabla, al realizar la concatenación externa de los clientes con las facturas, no se pierde ningún cliente. Pero si un cliente no tiene ninguna factura, habrá sido concatenado con una factura ficticia con sus valores todos a nulos. Por tanto, el resultado deseado se obtiene seleccionando aquellos clientes unidos a facturas ficticias. Una factura ficticia será aquella cuyo código de factura es nulo dado que una factura real no puede tener un código de factura nulo al ser ésta su clave primaria.

9.8 Operación "Todo"

Existe un tipo de consulta muy habitual en la extracción de información: determinar comportamientos que siempre se cumplen. Algunos ejemplos de este tipo de consultas son los siguientes: clientes a los que siempre se les ha aplicado descuento, vendedores que siempre han aplicado descuento, años en los que hay ventas en todos los meses de un determinado artículo, clientes para los que todas sus facturas tienen iva 16, artículos que siempre son vendidos en más de 5 unidades, etc.

Este tipo de consultas pueden resolverse de varias formas, siendo las más habituales las siguientes: con subconsultas aplicando el método de la doble negación o con el operador algebraico de diferencia.

Vamos a verlo en un ejemplo. Supongamos que se desea obtener el código y descripción de aquellos artículos que siempre se venden en cantidades superiores a 5 unidades (en cada línea de factura).

- **Ejercicio**: ¿La siguiente consulta devuelve el código y descripción de aquellos artículos que siempre se venden en cantidades superiores a 5 unidades?

```
select a.codart, a.descrip
from   articulos a
where  a.codart in ( select l.codart
                     from   lineas_fac l
                     where  l.cant > 5 ) ;
```

  Solución: No. La anterior consulta muestra aquellos artículos que alguna vez (en alguna factura) se han vendido en cantidad superior a 5 unidades.

Así pues, la anterior solución no sirve. Para resolver este problema se puede aplicar el método de la doble negación: un artículo se habrá vendido siempre en cantidades superiores a 5 unidades si se ha vendido alguna vez y nunca se ha vendido en cantidades inferiores o iguales a 5 unidades. La siguiente sentencia obtiene el resultado deseado:

```
select a.codart, a.descrip
from   articulos a
where  a.codart in ( select l.codart
                       from   lineas_fac l )
and    not exists( select '*'
                   from   lineas_fac l
                   where  l.codart = a.codart
                   and    l.cant <= 5 );
```

La primera subconsulta comprueba que el artículo ha sido vendido por lo menos una vez. La segunda subconsulta busca ventas con 5 o menos unidades. Por tanto, el artículo habrá sido vendido siempre en cantidades superiores a 5 unidades si ha sido vendido y no existen ventas del mismo con 5 o menos unidades.

El segundo método para resolver este tipo de problemas consiste en trabajar con la diferencia de consultas. En una primera consulta se extraen todas las ventas y a este conjunto se le quitan todos aquellos artículos que han sido vendidos en cantidades iguales a 5 o inferiores. De esta forma, en el resultado final sólo quedan artículos que han sido vendidos en cantidades superiores a 5 unidades.

```
select a.codart, a.descrip
from   lineas_fac l join articulos a using ( codart )
except
select a.codart, a.descrip
from   lineas_fac l join articulos a using ( codart )
where  l.cant <= 5 ;
```

Aunque no siempre se puede hacer, en ocasiones se puede escribir una consulta de este tipo usando simplemente funciones de agrupación. Por ejemplo, si se desea mostrar aquellos artículos que siempre han sido vendidos en cantidades superiores a las 5 unidades, podría calcularse el mínimo de las ventas y mirar si éste es superior a 5. Si el mínimo no es superior a 5, entonces es que en alguna ocasión se vendieron 5 o menos unidades y, por tanto, el artículo no debe aparecer.

```
select codart, a.descrip
from   lineas_fac l join articulos a using ( codart )
group  by codart, a.descrip
having min( l.cant ) > 5 ;
```

9.9 Equivalencia de sentencias

Algunos SGBD no son eficientes procesando consultas que tienen subconsultas anidadas con referencias externas; otros carecen por completo de esta posibilidad. Por tanto, es muy conveniente saber encontrar sentencias equivalentes que no utilicen subconsultas, si es posible.

Por ejemplo, la siguiente sentencia también obtiene los datos de las facturas que tienen descuento en todas sus líneas. Utiliza una subconsulta en la cláusula **from** y no posee referencias externas.

```
select *
from   facturas join
       ( select codfac
         from   lineas_fac
         group  by codfac
         having min( coalesce( dto, 0 ) ) > 0 ) lf
       using ( codfac );
```

Otra versión es la que se muestra a continuación:

```
select *
from   facturas
where  codfac in ( select codfac
                   from   lineas_fac
                   group  by codfac
                   having min( coalesce( dto, 0 ) ) > 0 ) ;
```

En muchas ocasiones una misma consulta de datos puede responderse mediante distintas sentencias **select** que utilizan operadores diferentes. Cada una de ellas dará, por lo general, un tiempo de respuesta diferente. La diferencia de tiempos entre sentencias que resuelven el mismo problema no es nimia. En algunos casos una sentencia puede obtener una velocidad 10 veces mayor que otra, aunque lo habitual es encontrar mejoras de velocidad de unas dos veces.

El que una sea sentencia sea más rápida en unas circunstancias no garantiza que vaya a serlo siempre: puede que al evolucionar el contenido de la base de datos, una sentencia, que era la mejor, deje de serlo porque las tablas hayan cambiado de tamaño o se haya creado o eliminado algún índice, etc.

Por todo lo anterior, frente a un problema es muy importante ser capaz de dar varias sentencias alternativas.

9.10 Ejercicios

- **Ejercicio 9.1**: Escribir una consulta que muestre el número de artículos con un precio superior al 55 % del precio máximo de los artículos.
  Ayuda: Uso de una subconsulta para calcular el precio máximo de los artículos.
  Solución:

```
select count( * )
from   articulos a
where  a.precio > ( select 0.55 * max( a2.precio )
                    from   articulos a2 );
```

♦ **Ejercicio 9.2**: Escribir una consulta que muestre el nombre de la provincia con mayor número de clientes.
Ayuda: La subconsulta debe calcular el número máximo de clientes de una provincia.
Solución:

```
select pr.nombre
from   clientes c, pueblos p, provincias pr
where  pr.codpro = p.codpro
and    p.codpue = c.codpue
group  by pr.codpro, pr.nombre
having count( * ) =
         ( select max( count( * ) )
           from   clientes c2, pueblos p2, provincias pr2
           where  pr2.codpro = p2.codpro
           and    p2.codpue = c2.codpue
           group  by pr2.codpro ) ;
```

♦ **Ejercicio 9.3**: Escribir una consulta que muestre el código y nombre de aquellos clientes que compraron en todos los meses del año (no necesariamente del mismo año).
Ayuda: La subconsulta cuenta el número de meses en que hay ventas para un determinado cliente.
Solución:

```
select c.codcli, c.nombre
from   clientes c
where  12 = ( select count( distinct to_char( f.fecha, 'mm' )
)
              from   facturas f
              where  c.codcli = f.codcli );
```

♦ **Ejercicio 9.4**: Escribir una consulta que muestre el código y nombre de aquellos vendedores cuya media mensual de facturas durante el año pasado fue inferior a 5.
Ayuda: La subconsulta calcula la media mensual de facturas para un vendedor. Ésta se calcula como el número total de facturas dividido por 12. La siguiente sentencia usa la sintaxis de Oracle.
Solución:

```
select v.codven, v.nombre
from   vendedores v
where  5>( select count( * ) / 12
           from   facturas f
           where  f.codven = v.codven
           and    to_number( to_char( f.fecha, 'yyyy' ) ) =
                  to_number( to_char( sysdate, 'yyyy' ) )-1
);
```

♦ **Ejercicio 9.5**: Escribir una consulta que muestre el código y fecha de las facturas con descuento para aquellos clientes cuyo código postal comienza por 12.
Ayuda: La subconsulta obtiene los códigos de clientes cuyo código postal comienza por 12. La consulta principal muestra aquellos clientes cuyo código se encuentra entre los encontrados. La consulta y la subconsulta se unen con el operador **in** o con el = **any**.
Solución:

```
select f.codfac, f.fecha
from   facturas f
```

```
where  coalesce( f.dto, 0 ) > 0
and    f.codcli in ( select c.codcli
                     from   clientes c
                     where  c.codpostal like '12%' );
```

♦ **Ejercicio 9.6**: Escribir una consulta que muestre el número de pueblos en los que no tenemos clientes.
Ayuda: Uso de una subconsulta con negación.
Solución:

```
select count( * )
from   pueblos
where  codpue not in ( select codpue
                       from   clientes ) ;
```

♦ **Ejercicio 9.8**: Escribir una consulta que muestre el número de artículos cuyo stock supera las 20 unidades, con un precio superior a 15 euros y de los que no hay ninguna factura en el último trimestre del año pasado.
Ayuda: Uso de una subconsulta (con o sin referencia externa) con negación simple.
Solución:

```
select count( * )
from   articulos a
where  a.stock > 20
and    a.precio > 15
and    a.codart not in
         ( select l.codart
           from   lineas_fac l, facturas f
           where  f.codfac = l.codfac
           and    to_char( f.fecha, 'q' ) = '4'
           and    to_number( to_char( f.fecha, 'yyyy' )) =
                    to_number( to_char( sysdate, 'yyyy' ) )-
1);

select count( * )
from   articulos a
where  a.stock > 20
and    a.precio > 15
and    not exists(
           select '*'
           from   lineas_fac l, facturas f
           where  f.codfac = l.codfac
           and    to_char( f.fecha, 'q' ) = '4'
           and    to_number( to_char( f.fecha, 'yyyy' ))=
                  to_number( to_char( sysdate, 'yyyy'))- 1
           and    l.codart = a.codart ) ;
```

♦ **Ejercicio 9.9**: Escribir una consulta que muestre el código y descripción de aquellos artículos que siempre se han vendido en los primeros tres meses del año.
Ayuda: Este es un ejercicio del tipo todo. Se puede resolver de tres modos: con el método de la doble negación, con la diferencia de conjuntos y con funciones de agrupación. En el primer modo se obtienen aquellos artículos que se han vendido al menos una vez y que no se han vendido en meses posteriores a marzo. En el segundo método se obtienen los artículos vendidos al menos una vez menos los artículos que se han vendido en un mes posterior a marzo. En el tercero se obtiene el máximo mes en que ha sido vendido un artículo y se comprueba que sea menor o igual a 3.
Solución:

```
select a.codart, a.descrip
from   articulos a
where  a.codart in ( select l.codart
                     from   lineas_fac l )
and    not exists( select '*'
```

```
                    from   lineas_fac l, facturas f
                    where  f.codfac = l.codfac
                    and    to_char( f.fecha, 'mm' ) > '03'
                    and    l.codart = a.codart ) ;


select codart, a.descrip
from   lineas_fac l join articulos a using ( codart )
minus
select codart, a.descrip
from   lineas_fac l join facturas f using ( codfac )
                    join articulos a using ( codart )
where  to_char( f.fecha, 'mm' ) > '03' ;


select codart, a.descrip
from   lineas_fac l join facturas f using ( codfac )
                    join articulos a using ( codart )
group  by codart, a.descrip
having max( to_number( to_char( f.fecha, 'mm' ) ) ) <= 3 ;
```

♦ **Ejercicio 9.10**: Escribir una consulta que muestre el código y la descripción de los artículos cuyo precio es mayor de 90,15 euros y se hayan vendido menos de 10 unidades (o ninguna) durante el año pasado.
Ayuda: La subconsulta debe calcular las ventas para el artículo actual.
Solución:

```
select a.codart, a.descrip
from   articulos a
where  a.precio > 90.15
and    10 > ( select nvl( sum( cant ), 0 )
              from   lineas_fac l, facturas f
              where  f.codfac = l.codfac
              and    to_number( to_char( fecha, 'yyyy' ) ) =
                     to_number( to_char( sysdate, 'yyyy' ) )
- 1
              and    l.codart = a.codart ) ;
```

♦ **Ejercicio 9.11**: Escribir una consulta que muestre el código y nombre de aquellos vendedores que siempre han realizado sus ventas a clientes de la misma provincia.
Ayuda: Una posible forma de resolver este problema es de la siguiente forma: la subconsulta calcula el número de provincias de residencia de los clientes a los que ha facturado un vendedor y la consulta principal comprueba que el número de provincias de los clientes atendidos por un vendedor es una.
Solución:

```
select v.codven, v.nombre
from   vendedores v
where  1 = ( select count( distinct p.codpro )
             from   facturas f, clientes c, pueblos p
             where  c.codcli = f.codcli
             and    p.codpue = c.codpue
             and    v.codven = f.codven ) ;
```

♦ **Ejercicio 9.12**: Escribir una consulta que muestre el nombre del cliente con mayor facturación.
Ayuda: Uso de subconsultas y agrupaciones. La subconsulta debe calcular la mayor facturación realizada por un cliente.
Solución:

```
select c.nombre
from   lineas_fac l, facturas f, clientes c
where  c.codcli = f.codcli
```

```
and    f.codfac = l.codfac
group  by c.codcli, c.nombre
having sum( cant * precio ) =
         ( select max( sum( cant * precio ) )
           from   lineas_fac l, facturas f
           where  f.codfac = l.codfac
           group  by f.codcli ) ;
```

♦ **Ejercicio 9.13**: Escribir una consulta que muestre el código, la descripción y el precio de los diez artículos más caros.
Ayuda: Uso de una subconsulta para contar cuántos artículos son más caros que el actual.
Solución:

```
select a.codart, a.descrip, a.precio
from   articulos a
where  10 > ( select count( * )
              from   articulos a2
              where  a2.precio > a.precio )
order  by 3 desc ;
```

♦ **Ejercicio 9.14**: Escribir una consulta que obtenga el código y nombre de aquellos clientes que durante el año pasado realizaron sus compras en meses consecutivos.
Ayuda: Un cliente habrá facturado en meses consecutivos de un determinado año si el mayor mes en el que realizó sus compras menos el menor mes en el que realizó sus compras es igual al número total de meses distintos en el que realizó compras menos uno. Ejemplo: un cliente habrá realizado sus compras en meses consecutivos si el último mes en que realizó sus compras fue agosto (mes 8), el primer mes en el que realizó sus compras fue junio (mes 6) y, además, realizó compras en un total de tres meses distintos.
Solución:

```
select codcli, c.nombre
from   clientes c join facturas f   using ( codcli )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c.nombre
having count( distinct to_char( f.fecha, 'mm' ) ) ) =
       max( to_number( to_char( f.fecha, 'mm' ) ) ) -
       min( to_number( to_char( f.fecha, 'mm' ) ) ) + 1 ;
```

♦ **Ejercicio 9.15**: Escribir una consulta que muestre el código y descripción de aquellos artículos que durante el año pasado fueron comprados siempre en cantidades pares (en las líneas de facturas).
Ayuda: Uso de la función *mod* para calcular el resto de la división entre 2. Operación del tipo todo: se puede resolver con minus o con doble negación.
Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
                   join facturas f   using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    mod( l.cant, 2 ) = 0
minus
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
                   join facturas f   using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    mod( l.cant, 2 ) != 0 ;
```

- **Ejercicio 9.16**: Escribir una consulta que obtenga el código y nombre de los clientes cuyas facturas han sido siempre inferiores a 1000 euros.
  Ayuda: Operación del tipo todo: se puede resolver con minus o con doble negación.
  Solución:

```
select codcli, c.nombre
from   clientes c join facturas f   using ( codcli )
                  join lineas_fac l using ( codfac )
group  by codcli, c.nombre
having sum( l.cant * l.precio ) < 1000.00
minus
select codcli, c.nombre
from   clientes c join facturas f   using ( codcli )
                  join lineas_fac l using ( codfac )
group  by codcli, c.nombre
having sum( l.cant * l.precio ) >= 1000.00 ;
```

9.11 Autoevaluación

- **Ejercicio 1**: Código y nombre de aquellos clientes de Castellón que durante el año pasado siempre han comprado artículos cuyo precio iguala o supera los 20 euros. Resolver el ejercicio de dos formas distintas.
- **Ejercicio 2**: Mes del año pasado en que se realizó una mayor facturación.
- **Ejercicio 3**: Vendedores que en todas y cada una de sus facturas del año pasado vendieron más de 5 artículos distintos. Resolver el ejercicio de dos formas distintas.
- **Ejercicio 4**: Código y nombre del pueblo de Castellón en el que más se ha facturado (a clientes residentes en él) durante el año pasado.

10 CREACIÓN Y ACTUALIZACIÓN DE LOS DATOS

En este capítulo se describen las operaciones SQL que permiten la creación y el borrado de tablas, así como la actualización de los datos que las tablas almacenan con las operaciones de inserción, modificación y borrado de filas. La definición de estas operaciones incluye conceptos ya descritos en los capítulos anteriores, como los **select**, la restricción de filas mediante la cláusula **where** o las subconsultas.

10.1 Creación de Tablas

La tabla es el elemento básico de SQL sobre la que actúan la mayoría de las sentencias de este lenguaje. Para crear una de estas estructuras se utiliza la operación **create table**, que tiene la siguiente sintaxis:

```
create table nombre_tabla (
  definic_campo1 [, definic_campo2 [, ..., definic_campoN ]...]]
  [, restriccion1 [, restriccion2, ... [ , restriccionM ] ...
]]);
```

Como se puede observar en la definición, existen dos tipos de elementos en la creación de una tabla, los campos y las restricciones, aunque sólo los primeros son obligatorios.

Por lo que respecta a la definición de los campos de la tabla, se realiza como se muestra a continuación:

```
nom_campo tipo [(dim [,tam])] [null | not null] [default valor]
```

donde se observa que resulta obligatorio especificar el nombre, que debe ser único en la tabla, y el tipo, mientras que es opcional indicar si acepta nulos o no y si va a tener un valor por defecto cuando se realice una inserción de una fila. Realmente si no se especifica lo contrario el campo aceptará nulos y el valor por defecto será nulo. Por lo que respecta a los tipos, se muestran seguidamente los más comunes:

- **character (dim)** hace referencia a una cadena de caracteres de dimensión máxima igual a **dim**. En ORACLE también se acepta el tipo **varchar2 (dim)**.
- **boolean** es un campo cuyos únicos valores válidos son **true** o **false**.
- **date** es un campo en donde se almacena una fecha.
- **time** es un campo en donde se almacena una hora de un día.
- **integer** es un campo que almacena números enteros de como máximo **dim** dígitos. En ORACLE también se utiliza el tipo **number (dim)**.
- **decimal (dim, tam)** es un campo que almacena números reales de hasta **dim** dígitos y como mucho **tam** dígitos decimales. En ORACLE resulta más común la utilización del tipo **number (dim, tam)**.

A continuación se muestran un par de ejemplos de creación de tablas en las que únicamente se incluyen la definición de los campos.

♦ **Ejercicio**: Creación de la tabla de provincias, sin restricciones.
Solución:

```
create table provincias (
    codpro character(2)  not null,
    nombre character(20) not null default ' ');
```

♦ **Ejercicio**: Creación de la tabla de artículos, sin restricciones.
Solución:

```
create table articulos (
    codart    character(8)  not null,
```

```
descrip   character(40) not null,
precio    decimal(7,2)  not null  default 0.0,
stock     integer(6),
stock_min integer(6) );
```

Las restricciones son los elementos del modelo relacional que permiten que éste funcione, ya que sin ellas una base de datos no es más que un mero almacén de información pero sin ningún tipo de conexión entre los datos. La definición de éstas presenta una misma estructura:

```
constraint nombre tipo parametros
```

Es importante destacar que el nombre definido para una restricción debe ser único para toda la base de datos, por lo que se aconseja tomar un criterio general que incluya el tipo de restricción, la tabla o tablas que involucra, y los campos correspondientes. En los ejemplos posteriores se utilizará un criterio posible. Seguidamente se muestra para los diferentes tipos de restricciones el valor de **tipo** y **parametros**.

- Definición de la clave primaria de una tabla.

```
primary key (campo1[, campo2, ... ] )
```

- Definición de una clave alternativa de una tabla.

```
unique (campo1[, campo2, ... ] )
```

- Definición de la clave ajena de una tabla sobre otra.

```
foreign key (campo1[, campo2, ... ] )
    references tabla_referida [(campo1[, campo2, ... ] )]
    on delete [ no action|set null|set default|cascade|restrict
]
    on update [ no action|set null|set default|cascade|restrict
]
```

- Definición de una restricción sobre las filas de la tabla.

```
ckeck (condicion)
```

De las restricciones mostradas, las dos primeras únicamente indican los campos que conforman la clave correspondiente, la última especifica condiciones que deben cumplir los valores que desean ser insertados o modificados en las filas de la tabla, y la tercera especifica qué atributos definen una clave ajena a otra tabla y las características necesarias para su correcto funcionamiento. De todas ellas, esta última es la que tiene aspectos adicionales a desarrollar.

- La tabla referida debe existir para poder ser incluida en la definición de la clave ajena, por lo que la definición de las tablas debe seguir un orden. Así, la tabla de **lineas_fac** no puede ser creada antes que la tabla de **artículos**.
- Una clave ajena debe tener el mismo número de atributos que la clave primaria de la tabla a la que referencia, y además deben corresponder en tipo y dimensión.
- Si la clave ajena es compuesta se recomienda especificar las columnas de la tabla actual y de la tabla referida, para asegurar que la correspondencia entre campos sea la adecuada.
- La cláusula **on delete** indica qué ocurrirá cuando se intente borrar en la tabla referida una fila cuya clave primaria aparece como valor de clave ajena en alguna fila de la tabla actual. Definiéndose cinco opciones,
  - **no action**, es decir, no hacer nada en la tabla actual, pero borrar la fila en la tabla referida, lo que puede provocar problemas de falta de integridad de los datos.
  - **set null**, se asigna el valor **null** en los campos que forman la clave ajena de aquellas filas que tengan como clave ajena el valor de la clave primaria que se desea borrar en la tabla referida, y posteriormente se borra esta fila.
  - **set default**, se asigna el valor por defecto en los campos que forman la clave ajena de aquellas filas que tengan como clave ajena el valor de la clave primaria que se desea borrar en la tabla referida, y posteriormente se borra esta fila.

  - **cascade**, se borran las filas que tienen como clave ajena el valor de la clave primaria que se desea borrar en la tabla referida antes de borrar éstas.
  - **restrict**, si existe alguna fila que tiene como clave ajena el valor de la clave primaria que se desea borrar en la tabla referida, la fila asociada no se borra.
- La cláusula **on update** quiere indicar que ocurrirá cuando se intente modificar en la tabla referida la fila a la que apunta una fila de la tabla actual. Se definen las mismas cinco opciones que antes.
  - **no action**, es decir, no hacer nada, pero modificar la fila en la tabla referida, lo que puede provocar problemas de falta de integridad de los datos.
  - **set null**, se asigna el valor **null** en los campos que forman la clave ajena de aquellas filas que tengan como clave ajena el valor de la clave primaria que se desea modificar en la tabla referida, y posteriormente se modifica esta fila.
  - **set default** se asigna el valor por defecto en los campos que forman la clave ajena de aquellas filas que tengan como clave ajena el valor de la clave primaria que se desea modificar en la tabla referida, y posteriormente se modifica esta fila.
  - **cascade**, se modifican las filas que tienen como clave ajena el valor de la clave primaria que se desea modificar en la tabla referida antes de modificar éstas.
  - **restrict**, si existe alguna fila que tenga como clave ajena el valor de la clave primaria que se desea modificar en la tabla referida, la fila asociada no se modifica.

Ahora se presentan los ejemplos comentados con anterioridad pero en los que ya se detallan las restricciones de cada tabla.

♦ **Ejercicio**: Creación de la tabla de provincias con definición de clave primaria.
Solución:

```
create table provincias (
    codpro character(2)  not null,
    nombre character(20) not null default ' ',
    constraint pk_provincias primary key (codpro));
```

♦ **Ejercicio**: Creación de la tabla de artículos con definición de clave primaria y restricciones sobre los campos precio, stock y stock_min para que sólo admitan números no nulos y positivos.
Solución:

```
create table articulos (
    codart    character(8)  not null,
    descrip   character(40) not null,
    precio    decimal(7,2)  not null default 0.0,
    stock     integer(6),
    stock_min integer(6),
    constraint pk_articulos primary key (codart),
    constraint ch_precio_articulos check (precio > 0.0),
    constraint ch_strockm_articulos check
                                    (coalesce(stock_min,0) >
0),
    constraint ch_stock_articulos check
                                    (coalesce(stock,0) > 0) );
```

♦ **Ejercicio**: Creación de la tabla de facturas con definición de clave primaria, claves ajenas y restricciones sobre los campos iva y dto, para que sólo admitan unos valores concretos.
Solución:

```
create table facturas (
    codfac   integer(6) not null,
    fecha    date       not null,
    codcli   integer(5),
    codven   integer(5),
```

```
iva      integer(2),
dto      integer(2),
constraint pk_facturas primary key (codfac),
constraint ch_iva_facturas
    check (coalesce(iva,0) in (0, 7, 16) ),
constraint ch_dto_facturas
    check (coalesce(dto,0) in (0, 10, 20, 40, 50) ) ,
constraint fk_fact_cli foreign key (codcli) ,
    references clientes
    on delete restrict on update cascade
constraint fk_fact_ven foreign key (codven) ,
    references vendedores
    on delete restrict on update cascade );
```

10.2 Borrado de Tablas

La operación **drop table** permite eliminar los datos almacenados en una tabla y su definición, de acuerdo a la siguiente sintaxis.

```
drop table nombre_tabla;
```

Obviamente no se puede borrar una tabla en cualquier momento, sino que sólo se podrá borrar si no existe ninguna otra tabla en la base de datos que la referencia, por lo que la eliminación de las tablas de una base de datos se debe realizar en un orden determinado. Es por ello que no es posible borrar la tabla **artículos** si no se ha borrado previamente la tabla **lineas_fac**.

10.3 Inserción de Datos

La operación **insert** permite la introducción de nuevas filas en una tabla de la base de datos. La sintaxis más sencilla de esta operación permite la introducción de una nueva fila en la tabla a partir de los valores escalares correspondientes, de acuerdo a la siguiente sintaxis:

```
insert into nombre_tabla [ (columna1, columna2, columna3, ... )
]
      values ( valor1, valor2, valor3, ... ) ;
```

Seguidamente se muestra un ejemplo sencillo de utilización de esta operación en el que se muestra que esta sintaxis permite más de una solución.

- **Ejercicio**: Introducir un nuevo artículo cuyo código es 'ARTXXX', su descripción es "Artículo de prueba 1", con un precio actual de 10,20 euros, un stock de 90 y un stock mínimo de 10.

  Solución1:

  ```
  insert into articulos
      values ('ARTXXX', 'Artículo de prueba 1' 10.20, 90, 10);
  ```

  Solución2:

  ```
  insert into articulos
                     (codart, descrip, precio, stock, stock_min
  )
      values ('ARTXXX', 'Artículo de prueba 1', 10.20, 90, 10);
  ```

Como se puede observar, esta operación requiere que se indique la tabla sobre la que se desea realizar la inserción de los datos y los escalares que se desean insertar, pero no es necesario indicar las columnas sobre las que se desea realizar la operación. Esta situación es válida pero debe ajustarse a una serie de condiciones,

1. Todo valor especificado dentro de la cláusula **values** se debe corresponder en tipo y dimensión con la columna que aparece en el mismo lugar, es decir, el valor1 con la columna1, el valor2 con la columna2, … El valor **null** es uno de los valores válidos que se puede especificar para cualquier tipo de columna.
2. Si en la definición de la tabla aparecen más columnas que las especificadas en la operación, el resto de columnas de la fila introducida toman su valor por defecto, si lo tuviera, o bien **null**.
3. Si no se especifican las columnas de la tabla, se asume el orden de las columnas especificado en su creación, pudiéndose especificar menos valores que columnas tenga la tabla.
4. En cualuiera de los casos se debe asegurar que el valor de la clave primaria introducida sea no nula y única, que los valores no nulos introducidos en las claves ajenas existen en la tabla referida, y que se cumplen el resto de características de los atributos.
5. Para evitar errores y para prevenir posibles cambios en la definición de la base de datos, es aconsejable especificar las columnas sobre las que se desea insertar información.

Hay que remarcar que la primera de las cuestiones es especialmente delicada para el caso de las columnas de tipo **date** o **time**, ya que existen diferentes formatos de visualización de los datos, pero sólo uno de ellos puede ser utilizado para introducirlos.

- **Ejercicio**: Introducir un nuevo artículo cuyo código es 'ARTYYY', su descripción es 'Artículo de prueba 2', con un precio actual de 10,20 euros, sin información sobre el stock y un stock mínimo de 10.

  Solución1:

```
insert into articulos
    values ('ARTYYY', 'Artículo de prueba 2', 10.20, NULL,
10);
```

  Solución2:

```
insert into articulos ( codart, descrip, precio, stock,
stock_min )
    values ('ARTYYY', 'Artículo de prueba 2', 10.20, NULL,
10);
```

  Solución3:

```
insert into articulos ( codart, descrip, precio, stock_min)
    values ('ARTYYY', 'Artículo de prueba 2', 10.20, 10);
```

- **Ejercicio**: Introducir un nuevo artículo cuyo código es 'ARTZZZ', su descripción es 'Artículo de prueba 3', con un precio actual de 10,20 euros y sin información sobre el stock ni el stock mínimo.

  Solución1:

```
insert into articulos
     values ('ARTZZZ', 'Artículo de prueba 3', 10.20);
```

  Solución2:

```
insert into articulos ( codart, descrip, precio)
     values ('ARTZZZ', 'Artículo de prueba 3', 10.20);
```

Otra sintaxis de la operación **insert** permite introducir más de una fila a la vez, utilizando datos existentes en la base de datos, tal y como sigue:

```
insert into tabla [ (columna1, columna2, columna3,... ) ]
     sentencia select ;
```

En este caso se ejecuta la sentencia **select** y el resultado no es visualizado sino que los valores resultantes se introducen en la tabla indicada. Como en la definición anterior, se debe especificar la tabla sobre la que se desea realizar la operación pero no es necesario especificar las columnas afectadas debiéndose cumplir las mismas normas que se han descrito con anterioridad, aunque existe alguna particularidad:

1. Si existe alguna columna contador, es decir, que su valor se calcula automáticamente como es el caso de la mayoría de códigos, el valor de estos campos no puede ser introducido a partir de los valores obtenidos en una sentencia **select** pero sí lo podrían ser si se realiza una inserción fila a fila.
2. Para una correcta introducción de datos se aconseja probar la sentencia **select** y analizar los resultados que se obtienen antes de definir la sentencia **insert**.

Para finalizar se especifican una serie de ejemplos de inserción de varias filas en tablas de uso específico.

♦ **Ejercicio**: Introducir las artículos cuyo precio sea menor de 1 euro y cuyo stock sea menor que 50 en la tabla **pedir_articulos (codigo, fecha, codart, stock)**, donde **codigo** es una columna contador**.**
Solución:

```
insert into pedir_articulos ( fecha, codart, stock )
     select sysdate, codart, stock
       from articulos
       where ( stock < 50 ) and ( precio < 1.0 );
```

10.4 Modificación de Datos

La operación **update** permite modificar los valores almacenados en las columnas de una tabla, de acuerdo a la siguiente sintaxis:

```
update nombre_tabla set columna1 = expr1 [, columna2 = expr2,
...]
    [ where condicion ] ;
```

donde se observa que se puede modificar una o más columnas de la tabla, y también que es posible definir una condición que deben cumplir las filas a modificar.

♦ **Ejercicio**: Modificar el stock mínimo de los artículos para fijarlo a la mitad del stock actual.
Solución:

```
update articulos set stock_min = stock / 2 ;
```

Seguidamente se indican una serie de aspectos a considerar, cuando se desea modificar las filas de una tabla:

1. Igual que ocurre en un **select**, la condición puede ser tan compleja como sea necesario e incluso puede incluir subconsultas.
2. Si en las expresiones aparecen referidas columnas de la propia tabla, en la modificación de una fila se toma el valor actual de la columna en esa fila, permitiéndose introducir también el valor **null** si la columna los aceptara.
3. Hay que tener cuidado cuando se modifican las claves primarias, ya que alguna restricción de la base de datos puede impedirlo.
4. También hay que considerar el caso de las claves ajenas, ya que si se introduce un valor no nulo, éste debe existir en la tabla referida.
5. No hay que olvidar el resto de restricciones existentes sobre la tabla que no pueden ser infringidas por esta operación.

Seguidamente se muestran unos ejemplos de la operación **update**, donde se muestra el estado de la tabla afectada antes y después de ejecutar la sentencia.

♦ **Ejercicio**: Incrementar los precios de la tabla artículos en un 10%.

Solución:

```
update articulos set precio = precio * 1.1 ;
```

Tabla ARTICULOS Original	
CODART	PRECIO
A1	1
A2	2
A3	2
A4	
A5	3

Tabla ARTICULOS Modificada	
CODART	PRECIO
A1	1.1
A2	2.2
A3	2.2
A4	
A5	3.3

- **Ejercicio**: Reducir los precios de la tabla artículos en un 10% y aumentar su stock mínimo en un 25%, de aquellos artículos cuyas ventas hayan sido menor de 100 euros en el último año, sin tener en cuenta descuentos ni iva.

  Solución:

```
update articulos
   set precio = precio * 0.9, stock_min = stock_min * 1.25
   where codart in
      ( select l.codart
         from lineas_fac l join facturas f using (codfac)
         where (to_number(to_char (fecha, 'YYYY')) =
                  to_number(to_char (fecha, 'YYYY')) - 1)
         group by l.codart
         having SUM(l.cant * l.precio) < 100 );
```

Tabla ARTICULOS Original		
CODART	PRECIO	STOCK_MIN
A1	1	150
A2	2	
A3	2	30
A4		250
A5	3	10

Tabla ARTICULOS Modificada		
CODART	PRECIO	STOCK_MIN
A1	0.9	188
A2	1.8	
A3	1.8	38
A4		313
A5	2.7	13

10.5 Borrado de Datos

La operación **delete** permite la eliminación de todas las filas de una tabla o bien aquellas que cumplan una determinada condición, de acuerdo a la siguiente sintaxis.

```
delete from nombre_tabla [ where condicion ] ;
```

En este caso hay que tener en cuenta que no todas las filas podrán ser borradas, sino que se borrarán únicamente aquellas que cumplan las restricciones existentes en la base de datos por la definición de las claves ajenas.

- **Ejercicio**: Borrar los artículos con un stock igual a 0.

  Solución:

```
delete from articulos
    where stock = 0 ;
```

11 MANEJO AVANZADO DE LOS DATOS

En el capítulo anterior se han analizado las cuestiones básicas referidas a la creación de los datos dejando de lado la posible modificación de la definición inicial de una tabla, así como aspectos que resultan fundamentales para el correcto funcionamiento de una base de datos, tales como la creación de vistas y de índices. Todos estos aspectos son introducidos en este capítulo.

11.1 Modificación de Tablas

La operación **alter table** permite modificar la definición de una tabla, con la incorporación o eliminación de campos y/o restricciones, y con la modificación de columnas, tal y como sigue,

```
alter table nombre_tabla
  [ add [column definic_campo | restriccion] |
    modify column definic_campo |
    drop [column nombre_campo | constraint nombre_restriccion]
];
```

Según la sintaxis expuesta se observa que en la eliminación de campos y restricciones requiere únicamente indicar su nombre, mientras que en el resto de casos es necesario especificar la definición completa del campo o la restricción.

La modificación de una tabla puede presentar diferentes problemas en función del tipo de operación y sobre que campo o restricción se realice. Seguidamente se enuncian las cuestiones que hay que tener en cuenta en esta operación,

1. No se puede borrar una columna que se utiliza en una restricción.
2. No se puede eliminar una restricción que sea referencia desde otra, como es el caso de la clave primaria de una tabla que sea referenciada mediante una clave ajena desde otra.
3. Antes de añadir una restricción de tipo **check** sobre una columna se debe asegurar que todos los valores de la columna en las diferentes filas cumplan la condición, o bien que la tabla esté vacía.
4. No se puede modificar una columna de la tabla a **not null** a no ser que todos los valores de la columna en las diferentes filas sean no nulos, o bien que la tabla esté vacía.
5. No se puede añadir una columna que sea **not null** en una tabla que no esté vacía.

Seguidamente se muestra algún ejemplo que muestra el uso correcto de esta operación, que en algún caso puede requerir la realización de varias operaciones sucesivas.

- **Ejercicio**: Incorporación de la columna importe en la tabla lineas_fac;
  Solución:

```
alter table lineas_fac add column importe decimal (8,2) ;

update table lineas_fac
  set importe = round( cant * precio *
                       ( 1.0 - coalesce( dto, 0 ) / 100.0 ), 2
);

alter table lineas_fac modify column importe not null;
```

Obviamente, esta operación es útil cuando se desea modificar el esquema de una base de datos y con ello almacenar nuevas informaciones, pero también es muy útil en la definición inicial de la base de datos especialmente en la definición de ciclos referenciales, ya que para que una tabla sea mencionada en una restricción **foreign key** ésta debe existir. Este hecho se puede observar en la definición de la tabla **vendedores**, donde es necesario utilizar la operación **alter table** para definir la clave ajena **codjefe**.

- **Ejercicio**: Creación de la tabla de vendedores.
  Solución:

```
create table vendedores (
    codven     integer(5)    not null,
    nombre     character(50) not null,
    direccion  character(50) not null,
    codpostal  character(6),
    codpue     character(5) not null,
    codjefe    integer(5),
    constraint pk_vendedores primary key (codven),
    constraint fk_ven_pue foreign key (codpue),
        references pueblos
        on delete restrict on update cascade );

alter table vendedores add
    constraint fk_ven_jefe foreign key (codjefe) ,
        references vendedores
        on delete restrict on update cascade ;
```

Esta situación relacionada con los ciclos referenciales también aparece en el borrado de tablas, debiéndose utilizar **alter table** para eliminar alguna de las claves ajenas que forman el ciclo y de este modo poder realizar el borrado de las tablas.

- **Ejercicio**: Borrado de la tabla de vendedores.
  Solución:

```
alter table vendedores drop constraint fk_ven_jefe;

drop table vendedores;
```

11.2 Creación de Vistas

En algunos casos el administrador de la base de datos puede mostrar una visión parcial de la base de datos a un conjunto de usuarios, o bien algunos usuarios requieren una versión simplificada de los datos almacenados. En ambos casos, resulta interesante la definición de una vista, que, básicamente, es una sentencia **select** a la que el administrador de la base de datos le ha dado un nombre, de acuerdo a la siguiente sintaxis:

```
create view nombre_vista as sentencia_select ;
```

Seguidamente se muestran ejemplos de ambos tipos de vistas, para una mejor comprensión de esta tipo de estructura.

- **Ejercicio**: Creación de una vista que muestre, únicamente, los códigos postales de los clientes de la provincia de Castellón.
  Solución:

```
create view codigos_clientes as
    select distinct codpostal from clientes
      where codpostal like '12%' ;
```

- **Ejercicio**: Creación de una vista que muestre para el año actual, la provincia, el vendedor de esa provincia y el importe vendido. (No tener en cuenta descuentos ni iva).
  Solución:

```
create view mejor_vendedor as
  select codpro, conven, sum(l.precio * l.cant) importe
    from ((((lineas_fac l join facturas f using (codfac))
                          join vendedores v using (codven))
                          join pueblos p using (codpue))
                          join provincias pr using (codpro))
    where (to_number(to_char (f.fecha, 'yyyy')) =
             to_number(to_char (sysdate, 'yyyy')) )
    group by codpro, codven
    having sum ( l.cant * l.precio) =
      (select max (sum (l1.cant * l1.precio) )
        from (((lineas_fac l1 join facturas f1 using
(codfac))
                              join vendedores v1 using
(codven))
                              join pueblos p1 using (codpue))
            where to_number(to_char (f1.fecha, 'yyyy')) =
                    to_number(to_char (sysdate, 'yyyy'))
              and p1.codpro = codpro
            group by codven);
```

Una vez definidas, las vistas pueden ser utilizadas en las sentencias **select**, exactamente igual a como se realizaría con una tabla de la base de datos, tal y como se muestra en los ejercicios siguientes:

- **Ejercicio**: Selección de los clientes de Castellón de los que tenemos información sobre su código postal.
  Solución:

```
select c.nombre, c.direccion
  from clientes c join codigos_clientes using (codpostal);
```

- **Ejercicio**: Mostrar los datos del mejor vendedor de la Comunidad Valenciana para el año actual. (No tener en cuenta descuentos ni iva).
  Solución:

```
select codven, v.nombre, v.direccion
  from vendedores v join mejor_vendedor mv using (codven)
  where mv.codpro in ('03', '12', '46') and
        mv.importe = (select max(importe)
                      from mejor_vendedor
                      where mv.codpro in ('03', '12',
'46'))
```

A pesar de lo comentado, una vista difiere de la tabla en diferentes aspectos, pero todos ellos se pueden resumir en el hecho que en la tabla se almacena información y en la vista se visualizan datos almacenados en una o más tablas y, en algún caso, tras un procesamiento, como es el caso de sentencias **select** que incluyan un agrupamiento y un posterior cálculo. Seguidamente se enumeran las diferencias más importantes entre una tabla y una vista:

1. Una tabla siempre debe tener una clave primaria y una vista no **tiene** porqué.
2. Las columnas de una tabla aparecen claramente definidas, mientras que no se conocen las características de los campos de una vista, es decir, si aceptan o no nulos o el tipo exacto de cada uno de ellos.
3. Sobre una tabla es posible realizar operaciones de actualización, como inserción, modificación y borrado, mientras que sobre las vistas no es posible realizar este tipo de operaciones directamente, ya que no almacenan información.

Sobre esta última diferencia hay que realizar algunos comentarios, ya que aún siendo cierta la afirmación, sí es posible realizar operaciones de actualización sobre algunos tipos de vista, denominadas vistas actualizables. En estos casos, no se actualizan los datos de la vista porque no existen, sino que se actualizan los datos de la tabla o tablas asociadas a la vista. Se enuncian a continuación las propiedades que deben cumplir las vistas para ser actualizables.

1. Una vista es actualizable si su definición incluye las claves primarias y los atributos que no aceptan nulos de todas las tablas asociadas.
2. Los campos de una vista podrán ser modificados si se obtienen directamente de uno solo de los campos de alguna de las tablas y si la clave primaria de dicha tabla está incluida en la vista.
3. Las vistas definidas con operaciones de conjuntos pueden sufrir operaciones **update** o **delete** pero no pueden sufrir operaciones **insert**, ya que no se puede determinar en cuál de todas las tablas se debe realizar la inserción.

11.3 Creación de Índices

La mejora del tiempo de acceso a la información es una de las tareas más importantes del administrador de la base de datos. Se pueden realizar diferentes acciones sobre los datos que permiten reducir el tiempo de ejecución de las operaciones de la base de datos, pero quizás la más sencilla y práctica es la definición de uno o más índices sobre las tablas de la base de datos, a través de la siguiente operación:

```
create [ unique ] [ clustered ] index nombre_indice
  on nombre_tabla ( column-name [ asc | desc ], ...)
```

De un modo sencillo se puede entender un índice como una versión resumida de una tabla en la que aparecen todos los valores de una o más columnas de la tabla así como una referencia a la fila correspondiente.

♦ **Ejercicio**: Definición de un índice sobre el código postal de los clientes.
Solución:

```
create index codpostal_clientes
  on clientes ( codpostal);
```

En la operación **create index** aparecen ciertas claúsulas cuyo significado se detalla a continuación,

- **unique**, indica que asociado a cada valor almacenado en el índice sólo puede aparecer un registro de la tabla, es decir, que en la tabla los valores no nulos asociados a las columnas que componen la clave de búsqueda no se repiten.
- **clustered**, el orden de almacenamiento de los registros de la tabla mantienen el mismo orden que el que define el índice, lo que aumenta el coste de actualización del fichero.
- **[asc | desc]**, indica el criterio de ordenación que se utilizará para cada columna que componen la clave de búsqueda, siendo **asc** el valor por defecto.

La característica más importante de estos ficheros es que sus filas están ordenadas respecto de las columnas de la tabla que contiene, y que se denominan clave de búsqueda. Este hecho permite utilizar algoritmos de búsqueda muy eficientes en la localización de un valor concreto de la clave de búsqueda, reduciendo de modo considerable el coste de las **select** que los utilicen. En cambio incrementa el coste de las operaciones **insert** y **delete**, así como los **update** sobre alguna columna de la clave de búsqueda, ya que en todos esos casos es necesario actualizar la tabla original y los índices asociados.

Por lo que respecta a su ubicación, hay que decir que la gestión de los índices es realmente efectiva cuando se almacenan en memoria principal, ya que permite utilizar los citados algoritmos de búsqueda para el acceso a la información y para la realización de actualizaciones con un coste mínimo. Este hecho no siempre es posible, debido al tamaño del índice o al número de índices de la base de datos, en cuyo caso el gestor de la base de datos almacena en memoria principal versiones reducidas de los índices del sistema que permiten acelerar el manejo de la información.

Dado que no es posible definir un número ilimitado de índices, ya que tendría un coste de gestión excesivo, el administrador de la base de datos debe seguir una serie de criterios básicos que asegure la creación de un número suficiente de índices. Seguidamente se enuncian algunos de los criterios que se puede seguir:

- Resulta aconsejable definir al menos un índice de tipo **unique** asociado a la clave primaria de todas las tablas de la base de datos.
- Se deben analizar las consultas que más tiempo consuman, para definir algún índice que acelere su ejecución.

Seguidamente se presentan dos ejemplos de creación de índices, ambos asociados a una clave primaria, pero en uno de ellos se obliga a definir un orden determinado en el almacenamiento de la información.

♦ **Ejercicio**: Definición de un índice sobre la clave primaria de clientes.
Solución:

```
create unique index ind_codcli_clientes
  on clientes ( codcli );
```

♦ **Ejercicio**: Definición de un índice sobre la clave primaria de provincias, que mantenga ordenada la tabla.
Solución:

```
create unique clustered index ind_codpro_provincias
  on provincias ( codpro );
```

12 SOLUCIÓN A LOS EJERCICIOS DE AUTOEVALUACIÓN

En este apartado se ofrece la solución a los ejercicios de autoevaluación propuestos en los capítulos anteriores. Hay que tener en cuenta que la solución a los ejercicios no siempre es única, es decir, un mismo ejercicio puede tener varias soluciones, siendo algunas de ellas muy distintas. Obviamente, algunas soluciones serán más eficientes y otras no, pero ello escapa a los objetivos de este libro. La probabilidad de poder encontrar distintas soluciones a un mismo ejercicio aumenta conforme se avanza y profundiza en SQL.

12.1 Soluciones a la autoevaluación del capítulo 2

- **Ejercicio 1**: Mostrar el código y nombre de aquellos vendedores cuyo jefe tiene el código 125.
  Solución:

```
select codven, nombre
from   vendedores
where  codjefe = 125;
```

- **Ejercicio 2**: Mostrar el código y descripción de aquellos artículos cuyo stock en el almacén supera los 100 euros.
  Solución:

```
select codart, descrip
from   articulos
where  precio * stock > 100.0 ;
```

- **Ejercicio 3**: Mostrar el código, sin que salgan repetidos, de los artículos vendidos en las facturas con código inferior a 100.
  Solución:

```
select distinct codart
from   lineas_fac
where  codfac < 100 ;
```

12.2 Soluciones a la autoevaluación del capítulo 3

- **Ejercicio 1**: Pueblos de la provincia de Castellón cuya primera y última letra coinciden.
  Solución:

```
select codpue, nombre
from   pueblos
where  codpro = '12'
and    upper( substr( nombre, 1, 1 ) ) =
       upper( substr( nombre, length( nombre ), 1 ) );
```

- **Ejercicio 2**: Se desea hacer una promoción especial de los artículos más caros (aquéllos cuyo precio supera los 10 euros). Mostrar el código, descripción, precio original y precio de promoción de los artículos. El precio de promoción se calcula de

la siguiente forma: Si el precio es menor de 20 euros, se aplica un 10 % de descuento en la promoción. Si es menor de 30 euros, se aplica un 20 %. Si es menor de 40 euros se aplica un 30 %. Si supera los 40 euros, se aplica un 40 %.
Solución:

```
select codart, descrip, precio,
       precio * case when precio < 20 then 0.9
                     when precio < 30 then 0.8
                     when precio < 40 then 0.7
                                      else 0.6
                end
from   articulos
where  precio > 10.00 ;
```

♦ **Ejercicio 3**: Código, fecha y código de cliente de las facturas de los diez primeros días del mes de febrero del año pasado.
Solución:

```
select codfac, fecha, codcli
from   facturas
where  to_number( to_char( fecha, 'dd' ) ) <= 10
and    to_char( fecha, 'mm' ) = '02'
and    to_number( to_char( fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

12.3 Soluciones a la autoevaluación del capítulo 4

♦ **Ejercicio 1**: Escribir una sentencia que calcule el número máximo de líneas en una factura.
Solución:

```
select max( linea )
from   lineas_fac ;
```

♦ **Ejercicio 2**: Escribir una sentencia que calcule el número de facturas sin descuento (cero o nulo); con descuento moderado (<= 10) y con descuento elevado (> 10).
Solución:

```
select sum( case when coalesce( dto, 0 ) = 0 then 1
                 else 0 end ) FacSinDto,
       sum( case when coalesce( dto, 0 ) <= 10 then 1
                 else 0 end ) FacDtoModerado,
       sum( case when coalesce( dto, 0 ) <= 10 then 0
                 else 1 end ) FacDtoElevado
from   facturas ;
```

♦ **Ejercicio 3**: Escribir una sentencia que calcule el número medio de unidades vendidas por factura.
Solución:

```
select sum( cant ) / count( distinct codfac )
from   lineas_fac ;
```

12.4 Soluciones a la autoevaluación del capítulo 5

- **Ejercicio 1**: Escribir una consulta que obtenga el importe de la factura más alta.
  Solución:

```
select max( sum( cant * precio ) )
from   lineas_fac
group  by codfac ;
```

- **Ejercicio 2**: Escribir una consulta que calcule el número de clientes a los que han realizado facturas los vendedores de la empresa.
  Solución:

```
select codven, count( distinct codcli )
from   facturas
group  by codven ;
```

- **Ejercicio 3**: Escribir una consulta que obtenga el número más alto de clientes que viven en el mismo pueblo.
  Solución:

```
select max( count( * ) )
from   clientes
group  by codpue ;
```

12.5 Soluciones a la autoevaluación del capítulo 6

- **Ejercicio 1:** Para aquellos clientes de la Comunidad Valenciana cuyo nombre comienza por la misma letra que comienza el nombre del pueblo en el que residen, mostrar el nombre del cliente, el nombre del pueblo y el número de artículos distintos comprados durante el último trimestre del año pasado. En el listado final sólo deben aparecer aquellos clientes cuya facturación en el mismo periodo superó los 6000 euros, sin considerar impuestos ni descuentos.
  Solución:

```
select codcli, c.nombre, p.nombre, count( distinct l.codart )
from   clientes c join pueblos p     using ( codpue )
                  join facturas f    using ( codcli )
                  join lineas_fac l  using ( codfac )
where  p.codpro in ( '03', '12', '46' )
and    upper( substr( c.nombre, 1, 1 ) ) =
       upper( substr( p.nombre, 1, 1 ) )
and    to_char( f.fecha, 'q' ) = '4'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c.nombre, p.nombre
having sum( l.cant * l.precio ) > 6000.00 ;
```

- **Ejercicio 2:** Artículos cuya descripción consta de más de 15 letras o dígitos que han sido comprados por más de 5 clientes distintos de la provincia de Castellón durante los últimos diez días del año pasado. En el listado final se debe mostrar el artículo y su descripción.
  Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
                   join facturas f   using ( codfac )
                   join clientes c   using ( codcli )
                   join pueblos p    using ( codpue )
where  length( a.descrip ) > 15
and    p.codpro = '12'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    to_char( f.fecha, 'mm' ) = '12'
and    to_number( to_char( f.fecha, 'dd' ) ) > 21
group  by codart, a.descrip
having count( distinct codcli ) > 5 ;
```

- **Ejercicio 3:** Código y nombre de aquellos pueblos cuya primera letra del nombre es la misma que la primera letra del nombre de la provincia, en los que residen más de 3 clientes y en los que se han facturado más de 1000 unidades en total durante el tercer trimestre del año pasado.

  Solución:

```
select codpue, p.nombre
from   pueblos p join provincias pr  using ( codpro )
                 join clientes c     using ( codpue )
                 join facturas f     using ( codcli )
                 join lineas_fac l   using ( codfac )
where  upper( substr( p.nombre, 1, 1 ) ) =
       upper( substr( pr.nombre, 1, 1 ) )
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    to_char( f.fecha, 'q' ) = '3'
group  by codpue, p.nombre
having count( distinct codcli ) < 3
and    sum( l.cant ) > 1000 ;
```

- **Ejercicio 4:** Para aquellos vendedores cuyo primer o segundo apellido terminan con 'EZ' (se asume que ningún nombre de pila termina con dicho sufijo), mostrar el número de clientes de su misma provincia a los que ha realizado alguna venta durante los 10 últimos días del año pasado. Mostrar el código y nombre del vendedor, además del citado número de clientes.

  Solución:

```
select codven, v.nombre
from   vendedores v join pueblos p1 on ( v.codpue=p1.codpue )
                    join facturas f using ( codven )
                    join clientes c using ( codcli )
                    join pueblos p2 on ( c.codpue=p2.codpue )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    to_char( f.fecha, 'mm' ) = '12'
and    to_number( to_char( f.fecha, 'dd' ) ) > 21
and    ( upper( v.nombre ) like '%EZ %'
or       upper( v.nombre ) like '%EZ' )
and    p1.codpro = p2.codpro
group  by codven, v.nombre ;
```

12.6 Soluciones a la autoevaluación del capítulo 7

- **Ejercicio 1**: Escribir una consulta que obtenga el código y nombre de aquellas provincias en las que no hubo ventas de los vendedores residentes en dichas provincias durante el año pasado.
  Solución:

```
select codpro, pr.nombre
from   provincias pr
minus
select codpro, pr.nombre
from   provincias pr join pueblos p  using ( codpro )
                     join vendedores v using ( codpue )
                     join facturas f using ( codven )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1 ;
```

- **Ejercicio 2**: Escribir una consulta que muestre el código y descripción de aquellos artículos que se han vendido alguna vez, pero nunca en la provincia de Castellón.
  Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
minus
select codart, a.descrip
from   articulos a join lineas_fac l using ( codart )
                   join facturas f using ( codfac )
                   join clientes c using ( codcli )
                   join pueblos p  using ( codpue )
where  p.codpro = '12' ;
```

- **Ejercicio 3**: Escribir una consulta que muestre el nombre de cada provincia y el número de facturas realizadas a clientes de dicha provincia durante el año pasado. Si una provincia no tiene ninguna factura, debe aparecer con la cantidad cero.
  Solución:

```
select codpro, count( * )
from   provincias pr join pueblos p  using ( codpro )
                     join clientes c using ( codpue )
                     join facturas f using ( codcli )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codpro
union
(
  select codpro, 0
  from   provincias
  minus
  select codpro, 0
  from   provincias pr join pueblos p  using ( codpro )
                       join clientes c using ( codpue )
                       join facturas f using ( codcli )
  where  to_number( to_char( f.fecha, 'yyyy' ) ) =
         to_number( to_char( sysdate, 'yyyy' ) ) - 1
) ;
```

12.7 Soluciones a la autoevaluación del capítulo 8

♦ **Ejercicio 1**: Escribir una consulta que devuelva el código y descripción de aquellos artículos tales que el máximo descuento aplicado en sus ventas (líneas de facturas) es menor del 10 %. En el resultado deben aparecer todos los artículos.
Solución:

```
select codart, a.descrip
from   articulos a left join lineas_fac l using ( codart )
group  by codart, a.descrip
having max( coalesce( l.dto, 0 ) ) < 10 ;
```

♦ **Ejercicio 2**: Escribir una consulta que obtenga el código y nombre de aquellos clientes que han facturado a menos de 10 vendedores distintos residentes en su misma provincia.
Solución:

```
select codcli, c.nombre, count( distinct codven )
from   clientes c left join facturas f using ( codcli )
                  left join vendedores v using ( codven )
                  join pueblos p1 on ( c.codpue = p1.codpue )
             left join pueblos p2 on ( v.codpue = p2.codpue )
where  p2.codpro is null or p1.codpro = p2.codpro
group  by codcli, c.nombre
having count( distinct codven ) < 10 ;
```

♦ **Ejercicio 3**: Escribir una consulta que devuelva el código y nombre de los pueblos de la provincia de Castellón sin clientes o cuyo número de clientes residentes sea menor que 5. La consulta debe devolver también el número de clientes en cada pueblo
Solución:

```
select codpue, p.nombre, count( codcli )
from   pueblos p left join clientes c using ( codpue )
where  p.codpro = '12'
group  by codpue, p.nombre
having count( codcli ) < 5 ;
```

12.8 Soluciones a la autoevaluación del capítulo 9

♦ **Ejercicio 1**: Código y nombre de aquellos clientes de Castellón que durante el año pasado siempre han comprado artículos cuyo precio iguala o supera los 20 euros. Resolver el ejercicio de dos formas distintas.
Solución:

```
select codcli, c.nombre
from   clientes c join pueblos p     using ( codpue )
                  join facturas f    using ( codcli )
                  join lineas_fac l  using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    p.codpro = '12'
and    l.precio >= 20.00
minus
select codcli, c.nombre
from   clientes c join pueblos p     using ( codpue )
                  join facturas f    using ( codcli )
```

```
                  join lineas_fac l  using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    p.codpro = '12'
and    l.precio < 20.00 ;


select codcli, c.nombre
from   clientes c join pueblos p     using ( codpue )
where  p.codpro = '12'
and    codcli in
       ( select f.codcli
         from   facturas f join lineas_fac l using ( codfac )
         where  to_number( to_char( f.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1 )
and    codcli not in
       ( select f.codcli
         from   facturas f join lineas_fac l using ( codfac )
         where  to_number( to_char( f.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         and    l.precio < 20.00
         and    f.codcli is not null ) ;
```

- **Ejercicio 2**: Mes del año pasado en que se realizó una mayor facturación.
  Solución:

```
select to_char( f.fecha, 'mm' )
from   facturas f join lineas_fac l using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by to_char( f.fecha, 'mm' )
having sum( l.cant * l.precio ) =
         ( select max( sum( l.cant * l.precio ) )
           from   facturas f join lineas_fac l using (codfac)
           where  to_number( to_char( f.fecha, 'yyyy' ) ) =
                  to_number( to_char( sysdate, 'yyyy' ) ) - 1
           group  by to_char( f.fecha, 'mm' ) );
```

- **Ejercicio 3**: Vendedores que en todas y cada una de sus facturas del año pasado vendieron más de 5 artículos distintos. Resolver el ejercicio de dos formas distintas.
  Solución:

```
select codven, v1.nombre
from   vendedores v1 join facturas f1   using ( codven )
                     join lineas_fac l1 using ( codfac )
where  to_number( to_char( f1.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codfac, codven, v1.nombre
having count( distinct l1.codart ) > 5
minus
select codven, v2.nombre
from   vendedores v2 join facturas f2   using ( codven )
                     join lineas_fac l2 using ( codfac )
where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codfac, codven, v2.nombre
having count( distinct l2.codart ) <= 5 ;


select v.codven, v.nombre
from   vendedores v
where  v.codven in (
```

```
        select f1.codven
        from   facturas f1
        where  to_number( to_char( f1.fecha, 'yyyy' ) ) =
               to_number( to_char( sysdate, 'yyyy' ) ) - 1 )
and     v.codven not in (
        select f2.codven
        from   facturas f2 join lineas_fac l2 using ( codfac )
        where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
               to_number( to_char( sysdate, 'yyyy' ) ) - 1
        and    f2.codven is not null
        group  by codfac, f2.codven
        having count( distinct l2.codart ) <= 5 );
```

- **Ejercicio 4**: Código y nombre del pueblo de Castellón en el que más se ha facturado (a clientes residentes en él) durante el año pasado.
  Solución:

```
select codpue, p.nombre
from   pueblos p join clientes c using ( codpue )
                 join facturas f using ( codcli )
                 join lineas_fac l using ( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    p.codpro = '12'
group  by codpue, p.nombre
having sum( l.cant * l.precio ) =
       ( select max( sum( l2.cant * l2.precio ) )
         from   pueblos p2 join clientes c2 using ( codpue )
                           join facturas f2 using ( codcli )
                           join lineas_fac l2 using ( codfac)
         where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         and    p2.codpro = '12'
         group  by codpue, p2.nombre );
```

13 EJERCICIOS AVANZADOS

En este apartado se presentan diversos ejercicios en SQL de un nivel medio o alto. Asimismo, se ofrece una posible solución a cada uno de los ejercicios, aunque hay que tener en cuenta que la mayor parte de ellos tienen múltiples soluciones, cada una con sus ventajas e inconvenientes. En algunos casos se presenta más de una solución.

- **Ejercicio 1**: Mostrar, ordenadamente, el código y el nombre de los vendedores que han vendido al menos los mismos artículos que el vendedor con código 230.
  Solución:

```
select v.codven, v.nombre
from   vendedores v
where  not exists(
       select '*'
       from   articulos a
       where  exists(
          select '*'
          from   facturas f join lineas_fac l using( codfac )
          where  f.codven = '230'
           and   l.codart = a.codart )
       and  not exists(
          select '*'
          from   facturas f join lineas_fac l using( codfac )
          where  f.codven = v.codven
           and   l.codart = a.codart ) )
order  by 2;


select v.codven, v.nombre
from   vendedores v
where  not exists(
         select '*'
         from   articulos a join lineas_fac l using( codart )
                           join facturas f   using( codfac )
         where  f.codven = '230'
         and    codart not in(
            select l.codart
            from   facturas f join lineas_fac l using( codfac
)
            where  f.codven = v.codven
            and    l.codart = codart ) )
order  by 2;
```

- **Ejercicio 2**: Mostrar, ordenadamente, los artículos de los que se han vendido menos de 10 unidades en la Comunidad Valenciana.
  Solución:

```
select codart, a.descrip
from   articulos a
minus
select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
                   join clientes c   using( codcli )
                   join pueblos p    using( codpue )
where  p.codpro in ( '03', '12', '46' )
group  by codart, a.descrip
```

```
having sum( l.cant ) >= 10
order  by 2;
```

♦ **Ejercicio 3**: Mostrar, ordenadamente, el código y el nombre de los 5 mejores vendedores, en importe facturado, del segundo semestre del año pasado. (No tener en cuenta en el cálculo del importe los descuentos ni el iva).

Solución:

```
select codven, v.nombre
from   vendedores v join facturas f1   using( codven )
                    join lineas_fac l1 using( codfac )
where  to_number( to_char( f1.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    to_number( to_char( f1.fecha, 'mm' ) ) > 6
group  by codven, v.nombre
having 5 >
      ( select count( count( * ) )
        from   facturas f2 join lineas_fac l2 using( codfac )
        where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
               to_number( to_char( sysdate, 'yyyy' ) ) - 1
        and    to_number( to_char( f2.fecha, 'mm' ) ) > 6
        group  by f2.codven
        having sum( l1.precio * l1.cant ) <
               sum( l2.precio * l2.cant ) )
order  by 2;
```

♦ **Ejercicio 4**: Indicar claramente el enunciado que corresponde a la siguiente consulta SQL.

```
select count( distinct max( codven ) )
from   clientes c join facturas f   using( codcli )
                  join vendedores v using( codven )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codven, codcli
having count( * ) > 2 ;
```

Solución: Número de vendedores con más de 2 facturas durante el año pasado a un mismo cliente.

♦ **Ejercicio 5**: Mostrar, ordenadamente, el código y el nombre de los clientes que sólo compran los lunes y martes.

Solución:

```
select c.codcli, c.nombre
from   clientes c
where  c.codcli in (
         select f.codcli
         from   facturas f join lineas_fac l using( codfac )
         where  to_number( to_char( f.fecha, 'd' ) ) <= 2
       )
and    c.codcli not in(
         select f.codcli
         from   facturas f join lineas_fac l using( codfac )
         where  to_number( to_char( f.fecha, 'd' ) ) > 2
       )
order  by 2;
```

♦ **Ejercicio 6**: Mostrar, ordenadamente, el código y el nombre de los vendedores que han facturado menos de 1000 euros, o que no han facturado nada. (No tener en cuenta en el cálculo del importe los descuentos ni el iva).
Solución:

```
select v.codven, v.nombre
from   vendedores v
minus
select codven, v.nombre
from   vendedores v join facturas f   using( codven )
                    join lineas_fac l using( codfac )
group  by codven, v.nombre
having sum( l.precio * l.cant ) >= 1000.00
order  by 2;
```

♦ **Ejercicio 7**: Mostrar, ordenadamente, el código y la descripción de los artículos que durante el año pasado se han vendido, pero a menos de 3 clientes.
Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
minus
select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codart, a.descrip
having count( distinct f.codcli ) >= 3
order  by 2;
```

♦ **Ejercicio 8**: Mostrar, ordenadamente , el código y el nombre del vendedor, o vendedores, que durante este trimestre ha vendido a más clientes.
Solución:

```
select codven, v.nombre
from   vendedores v join facturas f using( codven )
where  to_char( sysdate,'yyyyq' ) = to_char( f.fecha, 'yyyyq'
)
group  by codven, v.nombre
having count( distinct f.codcli ) =
         ( select max( count( distinct f2.codcli ) )
           from   facturas f2
           where  to_char( sysdate, 'yyyyq' ) =
                  to_char( f2.fecha, 'yyyyq' )
           group  by f2.codven )
order  by 2;
```

♦ **Ejercicio 9**: Mostrar, ordenadamente, el código y la descripción de los artículos cuyo precio es inferior a la media y que siempre se han vendido con un 16% de iva.
Solución:

```
select a1.codart, a1.descrip
from   articulos a1 cross join articulos a2
where  16 =
    (  select min( nvl( f.iva,0 ) )
       from   facturas f join lineas_fac l using(codfac)
```

```
       where  l.codart = a1.codart)
group  by a1.codart, a1.descrip, a1.precio
having a1.precio < avg( a2.precio )
order  by 2;

select codart, a1.descrip
from   articulos a1 join lineas_fac l using( codart )
                    join facturas f using( codfac )
where  a1.precio <
       ( select avg( a2.precio ) from  articulos a2)
group  by codart, a1.descrip
having min( nvl( f.iva,0 ) ) = 16
order  by 2;
```

♦ **Ejercicio 10**: Mostrar, ordenadamente, un listado de los clientes con su código, su nombre y la fecha de su primera factura con importe superior a 250 euros. (No tener en cuenta en el cálculo del importe los descuentos ni el iva).

Solución:

```
select c.codcli, c.nombre, f1.fecha
from   clientes c join facturas f1 on( c.codcli = f1.codcli )
                  join lineas_fac l1 using( codfac )
group  by c.codcli, c.nombre, codfac, f1.fecha
having sum( l1.precio * l1.cant ) > 250 and f1.fecha =
    (  select min( min( f2.fecha ) )
       from   facturas f2 join lineas_fac l2 using( codfac )
       where  c.codcli = f2.codcli
       group  by codfac, f2.fecha
       having sum( l2.precio * l2.cant ) > 250)
order  by 2;


select codcli, c.nombre, min( f1.fecha )
from   clientes c join facturas f1 using( codcli )
where  f1.codfac in
    ( select codfac
      from   facturas f2 join lineas_fac l using( codfac )
      group  by codfac
      having sum( l.precio * l.cant ) > 250 )
group  by codcli, c.nombre
order  by 2;
```

♦ **Ejercicio 11**: ¿Qué expresiones deben aparecer en el **`select`** y en **`group by`** de la siguiente sentencia, para que cumpla el enunciado "Número de pueblos en los que se han realizado más de 250 facturas"?

```
select ...
from   clientes c join facturas f using( codcli )
group  by ...
having  count( * ) > 250;
```

Solución: Para que el **`count(*)`** del **`having`** cuente facturas, es necesario agrupar a través del **`cli.codpue`**, que es lo que pondremos en el **`group by`**. Dado que no se quiere mostrar un valor para cada grupo, sino que se pretende contar el número de grupos, se debe aplicar la función **`count(*)`** sobre una función de grupo. Por todo esto, el resultado es el siguiente,

```
select count( count( * ) )
from   clientes c join facturas f using( codcli )
group  by c.codpue
having count( * ) > 250;
```

♦ **Ejercicio 12**: De las provincias en las que haya más de 25 clientes, mostrar su nombre y el nombre de los pueblos de dicha provincia en los que haya más de 5 clientes, ordenados respecto de la provincia y el pueblo.
Solución:

```
select pr.nombre, p1.nombre
from   provincias pr join pueblos p1 using( codpro )
                     join clientes c1 using( codpue )
where  codpro in
       ( select p2.codpro
         from   pueblos p2 join clientes c2 using( codpue )
         group  by p2.codpro
         having count( c2.codcli ) > 25 )
group  by pr.nombre, codpro, p1.nombre, codpue
having count( c1.codcli ) > 5
order  by 2, 1;
```

♦ **Ejercicio 13**: Mostrar, ordenadamente por la provincia, el código y el nombre de cada provincia, así como el número de líneas de pedido que se han hecho desde esa provincia, siempre y cuando no se haya hecho ningún pedido o el total de los pedidos incluya un número menor de 100 líneas.
Solución:

```
select codpro, pr.nombre, count( l.linea )
from   provincias pr left join pueblos p using( codpro )
                     left join clientes c using( codpue )
                     left join facturas f using( codcli )
                     left join lineas_fac l using( codfac )
group  by codpro, pr.nombre
having count( l.linea ) < 100
order  by 2;
```

♦ **Ejercicio 14**: ¿A qué consulta corresponde la siguiente sentencia SQL?. Responder sin ambigüedad.

```
select codpro, pr1.nombre
from   provincias pr1 join pueblos p1 using( codpro )
                      join clientes c1 using( codpue )
                      join facturas f1 using( codcli )
                      join lineas_fac l1 using( codfac )
group  by pr1.nombre, codpro, codcli
having sum( l1.cant * l1.precio ) > 1000
minus
select codpro, pr2.nombre
from   provincias pr2 join pueblos p2 using( codpro )
                      join clientes c2 using( codpue )
                      join facturas f2 using( codcli )
                      join lineas_fac l2 using( codfac )
group  by pr2.nombre, codpro, codcli
having sum( l2.cant * l2.precio ) <= 1000
order  by 2;
```

Solución: Código y nombre de provincias, ordenados por el nombre, que cumplen que alguno de sus clientes ha facturado, y que el total de la facturación de cada uno de los clientes de dicha provincia que ha comprado siempre es mayor de 1000 euros.

♦ **Ejercicio 15**: Mostrar el código y el nombre de los clientes que tengan la mayor proporción entre el importe que han facturado y el número de unidades que han

comprado, es decir, que sea máxima la división entre la suma del importe de todas sus compras y la suma de unidades adquiridas.

Solución:

```
select codcli, c.nombre
from   clientes c join facturas f   using( codcli )
                  join lineas_fac l using( codfac )
group  by codcli, c.nombre
having sum( l.precio * l.precio ) / sum( l.cant ) =
       ( select max( sum( l2.precio * l2.precio ) /
                sum( l2.cant ) )
         from   facturas f2 join lineas_fac l2 using( codfac
)
         group  by codcli );


select codcli, c.nombre
from   clientes c join facturas f   using( codcli )
                  join lineas_fac l using( codfac )
group  by codcli, c.nombre
having sum( l.precio * l.precio ) / sum( l.cant ) >= all
       ( select sum( l2.precio * l2.precio ) / sum( l2.cant )
         from   facturas f2 join lineas_fac l2 using( codfac
)
         group  by codcli  );
```

- **Ejercicio 16**: Mostrar el código y el nombre de los clientes que durante el año pasado han comprado alguno de los artículos que tenga mayor precio actual, junto con el código, la descripción y el número de unidades vendidas de cada uno de los artículos. Ordenar el listado de forma descendente respecto de este número, y de forma ascendente respecto del nombre del cliente y la descripción del artículo.

  Solución:

```
select codcli, c.nombre, codart, a.descrip, sum( l.cant )
Total
from   clientes c join facturas f   using( codcli )
                  join lineas_fac l using( codfac )
                  join articulos a  using( codart )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    a.precio = ( select max( precio ) from articulos )
group  by codcli, c.nombre, codart, a.descrip
order  by 5 desc, 2, 4 ;
```

- **Ejercicio 17**: Para todas las provincias con más de 500 pueblos, incluidas las que no tuvieron ninguna venta, mostrar su código, su nombre y el número total de unidades vendidas de artículos por parte de los vendedores de la provincia, todo ello ordenado respecto del código de provincia.

  Solución:

```
select codpro, pr.nombre, sum( l.cant ) Total
from   provincias pr join pueblos p         using( codpro )
                     left join vendedores v using( codpue )
                     left join facturas f   using( codven )
                     left join lineas_fac l using( codfac )
group  by codpro, pr.nombre
having count( distinct codpue ) > 500
order  by 1;
```

♦ **Ejercicio 18**: Indicar claramente el enunciado que corresponde a la siguiente consulta SQL.

```
select v1.codven, v1.nombre
from   vendedores v1 join vendedores v2
                           on( v1.codjefe = v2.codven )
                      join pueblos p
                           on( v2.codpue = p.codpue )
                      join provincias pr using( codpro )
where  codpro <> '12'
and    v1.codven in (
         select codven
         from   vendedores v3 join facturas f3 using( codven
)
         where  to_char( f3.fecha, 'dd' ) <= 20
       )
and    v1.codven not in (
         select codven
         from   vendedores v3 join facturas f3 using( codven
)
         where  to_char( f3.fecha, 'dd' ) > 20
       )
order by 2 ;
```

Solución: Código y nombre de los vendedores que siempre realizan sus ventas en los primeros 20 días de cada mes y cuyo jefe no es de la provincia de Castellón.

♦ **Ejercicio 19**: Mostrar, ordenadamente, los clientes cuyas facturas han superado siempre los 600 euros durante el año pasado (No tener en cuenta los descuentos ni el iva).

Solución:

```
select codcli, c1.nombre
from   clientes c1 join facturas f1   using( codcli )
                   join lineas_fac l1 using( codfac )
where  to_number( to_char( f1.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c1.nombre, codfac
having sum( l1.precio * l1.cant) > 600.00
minus
select codcli, c2.nombre
from   clientes c2 join facturas f2   using( codcli )
                   join lineas_fac l2 using( codfac )
where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c2.nombre, codfac
having sum( l2.precio * l2.cant) <= 600.00
order by 2, 1;
```

♦ **Ejercicio 20**: Mostrar, ordenadamente, el código y el nombre de los clientes de la provincia de Castellón que durante el último trimestre del año pasado realizaron facturas con vendedores de más de tres pueblos diferentes.

Solución:

```
select codcli, c.nombre
from   clientes c join pueblos p on( c.codpue = p.codpue )
                  join facturas f using( codcli )
                  join vendedores v using( codven )
where  p.codpro = '12'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
```

```
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    to_char( sysdate, 'q' ) = '4'
group  by codcli, c.nombre
having count( distinct v.codpue ) > 3
order  by 2;
```

♦ **Ejercicio 21**: ¿La siguiente sentencia SQL devuelve como resultado, de modo ordenado, el código y el nombre de los vendedores de la provincia de Castellón que han realizado facturas a clientes de más de tres provincias distintas? Responder sin ambigüedad, indicando como corregir la sentencia en caso de que no sea correcta.

```
select v1.codven, v1.nombre
from   vendedores v1 join pueblos p1 using( codpue )
where  p1.codpro = '12'
and    3 <
      (  select count( distinct p2.codpro )
         from   pueblos p2 join clientes using( codpue )
                           join facturas f using( codcli )
                           join vendedores v2 using( codven )
)
order  by 2;
```

Solución: No, dado que la consulta obtendría todos los vendedores si la empresa trabaja con clientes de más de tres provincias, o ningún vendedor en el caso de que se trabaje con clientes de tres o menos provincias. El fallo de la consulta se produce porque no existe una referencia externa que seleccione en la subconsulta las facturas de un determinado vendedor, y sobre estas facturas contar el número de provincias distintas de los clientes del vendedor. La introducción de esta referencia externa daría el siguiente resultado,

```
select v1.codven, v1.nombre
from   vendedores v1 join pueblos p1 using( codpue )
where  p1.codpro = '12'
and    3 <
      (  select count( distinct p2.codpro )
         from   pueblos p2 join clientes c using( codpue )
                           join facturas f using( codcli )
         where  v1.codven = f.codven )
order  by 2;
```

♦ **Ejercicio 22**: Mostrar, ordenadamente, el artículo, o artículos, más vendido en la provincia de Castellón durante el año pasado.

Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
                   join clientes c   using( codcli )
                   join pueblos p    using( codpue )
where  p.codpro = '12'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codart, a.descrip
having sum( l.cant ) = (
         select max( sum( l2.cant ) )
         from   lineas_fac l2 join facturas f2 using( codfac
)
                              join clientes c2 using( codcli
)
                              join pueblos p2  using( codpue
)
```

```
          where  p2.codpro = '12'
          and    to_number( to_char( f2.fecha, 'yyyy' ) ) =
                 to_number( to_char( sysdate, 'yyyy' ) ) - 1
          group  by l2.codart )
order by 2;
```

♦ **Ejercicio 23**: Mostrar, ordenadamente, los artículos que, habiendo sido vendido alguna vez, nunca se les haya aplicado ningún descuento.
Solución:

```
select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
group  by codart, a.descrip
having max( nvl( l.dto, 0 ) ) = 0
order  by 2 ;
```

♦ **Ejercicio 24**: ¿A qué consulta corresponde la siguiente sentencia SQL?. Responder sin ambigüedad.

```
select codart, a.descrip, avg( nvl( l.cant, 0 ) )
from   articulos a left join lineas_fac l using( codart )
where  upper( codart ) like 'IM2F%' and a.precio > 15
group  by codart, a.descrip
order  by 2;
```

Solución: Muestra, ordenadamente, los artículos cuyo precio es mayor de 15 euros y cuyo código empieza por IM2F, así como la cantidad media que aparece en las líneas de pedido que los incluye o cero si dichos artículos no aparecen en ningún pedido.

♦ **Ejercicio 25**: Mostrar el código y el nombre de los clientes de Castellón que han realizado facturas con vendedores de más de dos provincias distintas. El resultado debe quedar ordenado ascendentemente respecto del nombre del cliente.
Solución:

```
select codcli, c.nombre
from   clientes c join pueblos p using( codpue )
where  p.codpro = '12'
and    2 <
       ( select count( distinct p2.codpro )
         from   pueblos p2 join vendedores v using( codpue )
                           join facturas f   using( codven )
         where f.codcli = c.codcli )
order  by 2;
```

♦ **Ejercicio 26**: Mostrar el código y el nombre de los vendedores que en el primer trimestre de este año han facturado menos que la facturación media de los vendedores con facturación para ese mismo trimestre. El resultado debe quedar ordenado ascendentemente respecto del nombre del vendedor.
Solución:

```
select codven, v.nombre
from   vendedores v join facturas f using( codven )
                    join lineas_fac l using( codfac )
where  to_char( f.fecha, 'q' ) = '1'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) )
group  by codven, v.nombre
having sum( l.cant * l.precio ) <
```

```
       ( select avg( sum( l2.cant * l2.precio ) )
         from   vendedores v2 join facturas f2
using(codven)
                              join lineas_fac l2
using(codfac)
         where  to_char( f2.fecha, 'q' ) = '1'
         and    to_number( to_char( f2.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) )
         group  by codven )
order  by 2;
```

♦ **Ejercicio 27**: Mostrar, ordenadamente, el código de los clientes cuyas facturas del año pasado siempre superaron un número total de artículos pedidos de 50 unidades.
Solución:

```
select c.codcli
from   clientes c
where  c.codcli in(
         select codcli
         from   facturas f join lineas_fac l using (codfac)
         where  to_number( to_char( f.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         group  by codfac, f.codcli
         having sum( l.cant ) > 50
       )
and    c.codcli not in(
         select codcli
         from   facturas f join lineas_fac l using (codfac)
         where  to_number( to_char( f.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         group  by codfac, f.codcli
         having sum( l.cant ) <= 50
       )
order by 1;
```

♦ **Ejercicio 28**: Indicar claramente el enunciado que corresponde a la siguiente consulta SQL.

```
select codcli, c.nombre
from   clientes c join facturas f   using( codcli )
                  join lineas_fac l using( codfac )
                  join pueblos p    using( codpue )
where  p.codpro = '12'
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
and    12 =
       ( select count( count( * ) )
         from   facturas f2 join lineas_fac l2 using( codfac
)
         where  f2.codcli = codcli
         and    to_number( to_char( f2.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         group  by to_char( f2.fecha, 'mm' )
         having sum( l2.cant * l2.precio ) > 600 )
group  by to_char( f.fecha, 'yyyy' ), codcli, c.nombre
having sum( l.cant * l.precio ) > 12000
order  by 2;
```

Solución: Código y nombre de los clientes de Castellón, ordenados ascendentemente respecto de este último, tales que durante el año pasado tuvieron una facturación de al menos 12000 euros y, además, en cada uno de sus

meses del año pasado tuvieron una facturación superior a los 600 euros. (sin tener en cuenta descuentos ni impuestos).

- **Ejercicio 29**: Para todos los clientes de la base de datos que tengan menos de 10 facturas, mostrar su código, nombre, y número total de unidades que han comprado de los artículos cuyo stock actual está por debajo de las 50 unidades. Cuando un cliente no tiene facturas el número de unidades mostradas debe ser cero.
  Solución:

```
select codcli, c.nombre,
       sum( case when coalesce( a.stock, 0 ) < 50
                 then coalesce( l.cant, 0 )
                 else 0 end ) unidades
from   clientes c left join facturas     using( codcli )
                  left join lineas_fac l using( codfac )
                  left join articulos a  using( codart )
group  by codcli, c.nombre
having count( distinct codfac ) < 10 ;
```

- **Ejercicio 30**: Mostrar, ordenadamente, el código y el nombre de los vendedores cuyo importe facturado durante el año pasado supera en un 10% a la media de facturación de los vendedores en dicho año. Considerar sólo los vendedores que tienen facturas.
  Solución:

```
select codven, v.nombre
from   vendedores v join facturas f   using( codven )
                    join lineas_fac l using( codfac )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate,'yyyy' ) ) - 1
group  by codven, v.nombre
having sum( l.cant*l.precio ) >
       ( select 1.1 * avg( sum( l2.cant*l2.precio ) )
         from   vendedores v2 join facturas f2
using(codven)
                              join lineas_fac l2
using(codfac)
         where  to_number( to_char( f2.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         group  by codven  )
order by 2;
```

- **Ejercicio 31**: Mostrar, ordenadamente, el código y la descripción de los artículos que siempre que se han vendido a clientes de la provincia de Castellón ha sido en lotes de más de 5 unidades en una misma línea de factura.
  Solución:

```
select codart, a1.descrip
from   articulos a1 join lineas_fac l1 using( codart )
                    join facturas f1   using( codfac )
                    join clientes c1   using( codcli )
                    join pueblos p1    using( codpue )
where  p1.codpro = '12'
and    l1.cant > 5
minus
select codart, a2.descrip
from   articulos a2 join lineas_fac l2 using( codart )
                    join facturas f2   using( codfac )
                    join clientes c2   using( codcli )
                    join pueblos p2    using( codpue )
```

```
where  p2.codpro = '12'
and    l2.cant <= 5
order  by 2, 1;


select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
                   join clientes c   using( codcli )
                   join pueblos p    using( codpue )
where  p.codpro = '12'
group  by codart, a.descrip
having min( l.cant ) > 5
order  by 2, 1;
```

♦ **Ejercicio 32**: Mostrar, ordenadamente, el código y el nombre de los artículos cuya segunda letra de la descripción coincida con la primera letra del alfabeto, y que durante el segundo semestre del año pasado han sido comprados por más de 10 clientes diferentes.

Solución:

```
select a.codart, a.descrip
from   articulos a
where  upper( a.descrip ) like '_A%'
and    10 <
       ( select count( distinct f.codcli )
         from   facturas f join lineas_fac l using( codfac )
         where  to_number( to_char( f.fecha, 'mm' ) ) > 6
         and    to_number( to_char( f.fecha, 'yyyy' ) ) =
                to_number( to_char( sysdate, 'yyyy' ) ) - 1
         and    l.codart = a.codart )
order  by 2, 1;


select codart, a.descrip
from   articulos a join lineas_fac l using( codart )
                   join facturas f   using( codfac )
where  upper( substr( a.descrip, 2, 1 ) ) = 'A'
and    to_number( to_char( f.fecha, 'mm' ) ) > 6
and    to_number( to_char( f.fecha, 'yyyy' ) ) =
       to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codart, a.descrip
having count( distinct f.codcli ) > 10
order by 2, 1;
```

♦ **Ejercicio 33**: Indicar claramente el enunciado que corresponde a la siguiente consulta SQL.

```
select a.codart, count( * )
from   articulos a join lineas_fac l1 on (a.codart =
l1.codart)
where  l1.cant > ( select avg( l2.cant )  from lineas_fac l2
)
and    exists
        ( select *
          from   lineas_fac l3
          where  l3.codart = a.codart )
and    not exists
        ( select *
          from   lineas_fac l4
          where  l4.codart = a.codart
```

```
          and    a.precio <= l4.precio )
group  by a.codart
order  by 1;
```

Solución: Para cada artículo que se haya vendido alguna vez y que todas las veces que se vende es con un precio inferior a su precio actual, mostrar su código y el número de líneas de pedido de ese artículo donde la cantidad es superior a la media.

- **Ejercicio 34**: Mostrar, ordenadamente, el código y el nombre de los vendedores que siempre venden a clientes de su misma provincia. Considerar solamente los vendedores que tengan facturas.

  Solución:

```
select  v.codven, v.nombre
from   vendedores v
where  not exists
       ( select *
         from   facturas f join clientes c using( codcli )
                           join pueblos p1 using( codpue )
                      cross join pueblos p2
         where  f.codven= v.codven and v.codpue = p2.codpue
         and    p1.codpro <> p2.codpro )
and    exists ( select *
                from   facturas f
                where  f.codven= v.codven )
order  by 2, 1;


select codven, v.nombre
from   vendedores v join facturas f using( codven )
minus
select codven, v1.nombre
from   vendedores v1 join facturas f1 using( codven )
                     join clientes c2 using( codcli )
                     join pueblos p1 on (v1.codpue = p1.codpue
)
                     join pueblos p2 on (c2.codpue = p2.codpue
)
where  p1.codpro <> p2.codpro
order  by 2, 1;
```

- **Ejercicio 35**: Recuperar las facturas con mayor importe total. Para cada factura mostrar su código, el código del cliente, el código del vendedor, su importe total, y el número de artículos diferentes que contiene en sus líneas.

  Solución:

```
select codfac, f1.codcli, f1.codven,
       sum( l1.cant * l1.precio ), count( distinct l1.codart
)
from   facturas f1 join lineas_fac l1 using( codfac )
group  by codfac, f1.codcli, f1.codven
having sum( l1.cant * l1.precio ) =
        ( select max( sum( l2.cant * l2.precio ) )
          from   facturas f2 join lineas_fac l2 using( codfac
)
          group  by codfac )
order  by 1;
```

- **Ejercicio 36**: ¿A qué consulta corresponde la siguiente sentencia SQL?. Responder sin ambigüedad.

Solución:

```
select codcli, c.nombre
from   clientes c join facturas f using( codcli )
where  to_number( to_char( f.fecha, 'yyyy' ) ) =
         to_number( to_char( sysdate, 'yyyy' ) ) - 1
group  by codcli, c.nombre
having max( to_number( to_char( f.fecha, 'mm' ) ) ) -
          min( to_number( to_char( f.fecha, 'mm' ) ) ) + 1 =
        count( to_number( to_char( f.fecha, 'mm' ) ) )
order  by 2, 1;
```

Solución: Muestra, ordenadamente, el código y el nombre de los clientes con facturas el año pasado, que cumplen que sus compras se han concentrado en un único mes o en varios meses coinsecutivos.

BIBLIOGRAFÍA

La bibliografía que puede consultarse sobre los sistemas de bases de datos relacionales y, concretamente, sobre SQL es, afortunadamente, enorme. A continuación se presentan algunos de ellos, por orden inversa de fecha:

- "SQL in a nutshell".
  Kevin Kline, Daniel Kline, Brand Hunt.
  Segunda edición, O'Reilly, 2004.
- "Advanced SQL:1999 - Understanding Object-Relational and Other Advanced Features".
  Jim Melton.
  Morgan Kaufmann, 2003.
- "Introducción al SQL para usuarios y programadores".
  Enrique Rivero Cornelio et al.
  Thomson-Paraninfo, 2002.
- "Mastering Oracle SQL".
  Sanjay Mishra, Alan Beaulieu.
  O'Reilly, 2002.
- "SQL:1999. Understanding Relational Language Components".
  Jim Melton, Alan R. Simon.
  Morgan Kaufmann, 2002.
- "Aprendiendo MySQL en 21 días".
  Mark Maslakowski, Tony Butcher.
  Pearson Educación, 2001.
- "Oracle SQL. The essential reference".
  David C. Kreines.
  O'Reilly, 2000.
- "Sams teach yourself SQL in 10 minutes".
  Ben Forta.
  Sams, 1999.
- "SQL for Smarties. Advanced SQL Programming".
  Joe Celko.
  Morgan Kaufmann Publishers, 1999.
- "Guía LAN Times de SQL".
  James R. Groff, Paul N. Weinberg.
  Osborne McGraw-Hill, 1998.
- "SQL El lenguaje de las bases de datos relacionales. Guía de referencia rápida".
  John Viescas.
  Microsoft Press, Anaya Multimedia, 1991.
- "Aplique SQL".
  James R. Groff, Paul N. Weinberg.
  Osborne/McGraw-Hill, 1991.