UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
Escuela Nacional de Estudios Profesionales
" ARAGON "

SQL: EL LENGUAJE ESTANDAR PARA BASES DE DATOS RELACIONALES

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A:

GABRIEL SERRANO GARCIA

asesor: ing. ernesto peñaloza romero

ENEP ARAGON

San Juan de Aragón, Edo. de México

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

A mis padres Domingo y Evelia :

Por ser padres.
Por todo el cariño, apoyo y comprensión que me han dado.
Por hacer posible mi formación profesional.

A mis hermanos Federico y Elsy :

Por su cariño.

A Estela :

Por su amor incondicional.
Por ser como es : un alma modelo.

Gracias a mis revisores:

Ing. Ernesto Peñaloza Romero. Por todo el tiempo dedicado a esta tesis. Por sus consejos y ayuda en el desarrollo de la misma.

Ing. Juan Gastaldi Pérez e Ing. Silvia Vega Muytoy. Por revisar mi tesis. Por la ayuda que siempre están dispuestos a ofrecer, no sólo a mi, sino a cualquier persona que la necesite.

Ing. Fernando Flores Zavaleta e Ing. Blanca Estela Cruz Luevano. Por revisar la tesis y por su amistad.

ÍNDICE.

INTRODUCCIÓN.....1

1. PANORÁMICA GENERAL.....3
1.1. El lenguaje SQL.....3
1.2. Breve historia del SQL.....6
1.2.1. Los primeros productos.....6
1.2.2. El mito de la portabilidad.....8
1.3. Características y beneficios de SQL.....10
1.4. El impacto de SQL.....12
1.4.1. SQL en IBM.....12
1.4.2. SQL en minicomputadores.....13
1.4.3. SQL en sistemas UNIX.....13
1.4.4. SQL y el procesamiento de transacciones.....14
1.4.5. SQL en computadores personales.....15
1.5. El futuro del SQL.....16
1.5.1. Bases de datos distribuidas.....17
1.5.2. Bases de datos orientadas a objetos.....18
1.6. Conceptos básicos de SQL/Bases de datos relacionales.....19
1.6.1. Bases de datos.....19
1.6.2. Sistemas de manejo de bases de datos (DBMS).....20
1.6.3. Objetivos de un DBMS.....21
1.6.4. Enfoque jerárquico.....25
1.6.5. Enfoque de red.....28
1.6.6. Enfoque relacional.....30
1.6.6.1. Base de datos Ejemplo.....31
1.6.6.2. Tabla.....32
1.6.6.3. Clave Primaria.....33
1.6.6.4. Relaciones.....34
1.6.6.4.1. Relación uno a muchos.....35
1.6.6.4.2. Relación uno a uno.....35
1.6.6.4.3. Relación mucho a muchos.....35
1.6.6.5. Claves foráneas.....36
1.6.6.6. Valores NULL.....37

1.7. Diseño de bases de datos relacionales. 37
1.7.1. Primera Forma Normal 39
1.7.1.1. Derivación y anomalías en las relaciones 1FN. 40
1.7.1.2. Normalización de la relación 1FN. 42
1.7.2. Segunda Forma Normal (2FN). 46
1.7.2.1. Anomalías de relaciones 2FN. 46
1.7.2.2. Normalización de una relación 2FN. 47
1.7.3. Tercera Forma Normal (3FN). 49
1.7.4. Cuarta Forma Normal (4FN). 52

2. ACCESO Y ACTUALIZACIÓN DE LA BASE DE DATOS. 57
2.1. Elementos de una sentencia. 57
2.2. Tipos de datos. 59
2.2.1. Datos en forma de caracteres 60
2.2.2. Datos numéricos. 61
2.2.3. Datos fecha/hora. 62
2.3. Consultas simples. 63
2.3.1. La sentencia SELECT 63
2.3.2. Columnas calculadas 67
2.3.3. Eliminación de filas duplicadas 69
2.3.4. Cláusula WHERE. 71
2.3.5. Predicados 72
2.3.5.1. Predicados de comparación. 73
2.3.5.2. Predicado de rango (BETWEEN). 75
2.3.5.3. Predicado de pertenencia a un conjunto (IN). 77
2.3.5.4. Predicado de correspondencia con un patrón (LIKE). 78
2.3.5.4.1. Uso del caracter comodín signo de porcentaje (%). 79
2.3.5.4.2. Uso del caracter comodín signo de subrayado (_). 79
2.3.5.5. Predicado de valor nulo (IS NULL). 81
2.3.5.6. Predicados compuestos. 82
2.3.6. Ordenación de los resultados de una consulta (cláusula ORDER BY). 85

2.4. Consultas multitabla (composiciones). 88
2.4.1. Consideraciones SQL para consultas multitabla. 89
2.4.1.1. Nombres de columna cualificados. 89
2.4.1.2. Selecciones de todas las columnas. 90
2.4.1.3. Alias de tablas. 91
2.4.2. Composición de dos tablas. 91
2.4.3. Consulta padre / hijo 93
2.4.4. Join con criterio de selección de fila. 94
2.4.5. Join de tres tablas. 96
2.4.6. Producto cartesiano. 97
2.4.7. Join basadas en desigualdad. 99
2.5. Combinación de resultados de consulta (UNION). 100
2.5.1. Uniones y filas duplicadas. 102
2.5.2. Uniones y ordenación. 103
2.5.3. Uniones múltiples. 104
2.6. Consultas sumarias. 106
2.6.1. Funciones de columna. 107
2.6.1.1. Función de columna SUM, ... 107
2.6.1.2. Función de columna AVG, ... 108
2.6.1.3. Funciones de columna MAX y MIN. 109
2.6.1.4. Función de columna COUNT(*). 110
2.6.1.5. Eliminación de filas duplicadas (DISTINCT). 110
2.6.1.6. Valores NULL en funciones de columna. 111
2.6.2. Consultas agrupadas. Cláusula GROUP BY. 112
2.6.3. Condiciones de búsqueda de grupos. Cláusula HAVING. 114
2.7. Subconsultas. 116
2.7.1. Subconsultas en la cláusula WHERE. 117
2.7.2. Condiciones de búsqueda en subconsultas. 120
2.7.2.1. Test de comparación. 120
2.7.2.2. Test de pertenencia a un conjunto (IN). 121
2.7.2.3. Test de existencia (EXISTS) 122
2.7.2.4. Test cuantificados (ANY y ALL) 122
2.7.2.4.1. Test ANY. 123
2.7.2.4.2. Test ALL. 124
2.7.3. Subconsultas anidadas 125
2.7.4. Subconsultas en la cláusula HAVING. 127
2.8. Actualización de la base de datos. 128
2.9. Adición de datos. 128
2.9.1. La sentencia INSERT de una fila. 129
2.9.2. Inserción de valores NULL. 131
2.9.3. La sentencia INSERT multifila. 132

2.10. Supresión de datos. 133
2.10.1. La sentencia DELETE. 134
2.11. Modificación de datos. 135
2.11.1. La sentencia UPDATE 135

3. INTEGRIDAD DE DATOS Y PROCESAMIENTO DE TRANSACCIONES 137
3.1. Integridad de datos. 137
3.1.1. Validación de datos. 138
3.1.2. Integridad de entidad. 139
3.1.3. Integridad referencial. 140
3.1.4. Disparadores (triggers) 142
3.1.4.1. ¿Qué es un disparador? 143
3.1.4.2. Disparadores e integridad referencial. 144
3.2. Procesamiento de transacciones. 146
3.2.1. Unidad Lógica de trabajo 146
3.2.2. Sentencias COMMIT y ROLLBACK 149
3.2.3. Transacciones: ¿ Qué hace el DBMS ? 150
3.2.4. Transacciones y procesamiento multiusuario 152
3.2.5. Bloqueos (Locking) 153
3.2.5.1. Niveles de bloqueo. 153
3.2.5.2. Bloqueos compartidos y exclusivos 154
3.2.5.3. Interbloqueos 156

4. CREACIÓN Y ADMINISTRACIÓN DE UNA BASE DE DATOS BASADA EN SQL 158
4.1. Creación de una base de datos. 158
4.1.1. Creación de tablas. 159
4.1.1.1. Definiciones de columnas 160
4.1.1.2. Valores por omisión. 160
4.1.1.3. Definiciones de clave primaria y foránea. 161
4.1.1.4. Reglas de supresión: 162
4.1.1.5. Catalogo del sistema. 163
4.1.2. Eliminación de una tabla 163
4.1.3. Modificación de una definición de tabla 164
4.1.3.1. Adición y supresión de columnas. 165
4.1.3.2. Modificaciones de claves primaria y foránea 166
4.1.4. Índices. 166
4.1.4.1. Creación de índices 168
4.1.4.2. Eliminación de índices 168
4.1.5. Estructuras de bases de datos. 169

5.1.6. Recuperación de datos en SQL incorporado..... 215
5.1.6.1. Consultas monofila..... 215
5.1.6.1.1. Manejo de errores en la sentencia SELECT singular..... 216
5.1.6.1.2. Recuperación de valores NULL..... 217
5.1.7. Programación con cursores..... 220
5.1.7.1. Concepto y uso de cursores..... 221
5.1.7.1.1. Sentencia DECLARE CURSOR..... 222
5.1.7.1.2. Sentencia OPEN..... 223
5.1.7.1.3. Sentencia FETCH..... 224
5.1.7.1.4. La sentencia CLOSE..... 225
5.1.7.2. Modificación de datos utilizando cursores..... 228
5.1.7.2.1. Cláusula FOR UPDATE..... 229
5.1.7.2.2. Sentencia UPDATE con cursor..... 230
5.1.7.2.3. Sentencia DELETE con cursor,..... 230
5.1.8. Cursores y procesamiento de transacciones..... 233
5.2. SQL Dinámico..... 234
5.2.1. SQL estático y SQL dinámico..... 236
5.2.2. Ejecución dinámica de una sentencia..... 237
5.2.3. Ejecución dinámica en dos pasos..... 239
5.2.3.1. Sentencia PREPARE,..... 239
5.2.3.2. Sentencia EXECUTE,..... 240
5.2.3.3. EXECUTE con USING..... 240
5.2.4. Consultas dinámicas..... 242
5.2.4.1. SQL dinámico para SELECT de lista fija..... 242
5.2.4.2. SQL dinámico para SELECT de lista variable..... 245
5.2.4.3. La sentencia DESCRIBE..... 247
5.2.4.4. La sentencia FETCH dinámica..... 248

Apéndice A. 253
Las 12 reglas de Codd para un DBMS relacional..... 253

Apéndice B. 257

Bibliografía. 258

INTRODUCCIÓN.

SQL es un lenguaje que permite organizar, recuperar y manejar datos almacenados en una base de datos.

Durante los últimos años, SQL se ha convertido en el lenguaje de base de datos estándar. Muchos productos para el manejo de bases de datos que circulan en el mercado soportan SQL, que puede ejecutarse tanto en computadoras personales como en grandes computadores.

El presente trabajo proporciona un tratamiento completo del lenguaje SQL y puede ser de utilidad para usuarios de la base de datos, para el diseñador de la misma, para programadores de aplicaciones y para administradores de la base de datos. SQL presenta funciones para cada uno de ellos, y en los capítulos dos a cinco se explican detalladamente:

Incluso puede ser de utilidad para los directores del área informática que desean conocer acerca del poder de SQL, ya que se presenta en su primer capítulo una breve historia del SQL, la características de SQL el impacto en el mercado y el futuro del lenguaje.

♦ Funciones para el usuario de la base de datos:

- *Recuperación de datos.* Permite a un usuario o programa de aplicación recuperar datos almacenados en la base de datos y utilizarlos.

- *Actualización de datos.* Permite a un usuario o programa de aplicación modificar los datos previamente almacenados.

- *Inserción de datos.* Permite a un usuario o programa la inserción de nuevos datos.

- *Borrado de datos.* Permite la supresión de datos antiguos en la base de datos.

♦ Funciones para el diseñador de la base de datos

- *Diseño de la base de datos/Creación de tablas.* Permite a un usuario definir la estructura y organización de los datos y las relaciones entre ellos.

- *Integridad referencial.* SQL define restricciones de integridad en la base de datos , protegiéndola de actualizaciones inconsistentes y fallos del sistema.

♦ Funciones para el programador de aplicaciones de la base de datos

- *SQL incorporado en lenguajes de tercera y cuarta generación.* SQL puede funcionar como un *sublenguaje incorporado* para extender el poder del lenguaje y utilizarlo en el acceso a la base de datos.

- *Diseño y formas sencillas para la interfaz con el usuario.* SQL proporciona los resultados de las consultas en forma de reportes.

♦ Funciones para el administrador de la base de datos

- *Respaldo y recuperación de la base de datos*

- *Performance y seguridad.* SQL puede ser utilizado para restringir la capacidad de un usuario, protegiendo los datos frente a accesos no autorizados.

Los capitulos dos a cuatro muestran en forma completa el lenguaje SQL, con muchos ejemplos para tratar de clarificar los conceptos de SQL

El último capítulo explica la programación avanzada con SQL. Discute el SQL incorporado a otros lenguajes de programación. Se muestran ejemplos en lenguaje C y COBOL, dos de los lenguajes más utilizados en el ambiente informático.

SQL es una herramienta potente y útil para enlazar los datos de una base de datos relacional con los usuarios y sistemas de información.

Por lo anterior descrito SQL es un lenguaje completo de control e interacción con un DBMS.

SQL es un lenguaje potente y a la vez muy sencillo de aprender. SQL se ha convertido en el lenguaje de bases de datos estándar para la utilización de bases de datos relacionales.

1.PANORÁMICA GENERAL.

1.1. El lenguaje SQL.

SQL (Structured Query Language) es una herramienta para la organización y manejo de datos almacenados en una base de datos. Como su nombre lo indica SQL es un lenguaje informático utilizado para interaccionar con la base de datos. SQL es un lenguaje para bases datos relacionales.

Es un lenguaje con sentencias que parecen sencillas frases en ingles, lo que lo hace un lenguaje fácil de aprender. Estas sentencias pueden ser ejecutadas en forma interactiva *(on-line)* en la terminal, proporcionando a los usuarios consultas rápidas de los datos almacenados. Así mismo puede ser usado en forma *incorporada* a otro lenguaje como C, COBOL, FORTRAN, APL, Assembler de IBM, BASIC, PL/1, Prolog, Ada, y Pascal. El lenguaje en el que SQL se puede incorporar varía dependiendo del fabricante.

Aunque originalmente SQL fue concebido como una herramienta para la consulta de datos, SQL es mucho más que eso, proporcionando a sus usuarios diversas funciones, entre las que se incluyen :

♦ Funciones para el usuario de la base de datos:

- *Recuperación de datos.* Permite a un usuario o programa de aplicación recuperar datos almacenados en la base de datos y utilizarlos.

- *Actualización de datos.* Permite a un usuario o programa de aplicación modificar los datos previamente almacenados.

- *Inserción de datos.* Permite a un usuario o programa la inserción de nuevos datos.

- *Borrado de datos.* Permite la supresión de datos antiguos en la base de datos.

♦ Funciones para el diseñador de la base de datos

- *Diseño de la base de datos/Creación de tablas.* Permite a un usuario definir la estructura y organización de los datos y las relaciones entre ellos.

- *Reglas del negocio/Integridad referencial.* SQL define restricciones de integridad en la base de datos , protegiéndola de actualizaciones inconsistentes y fallos del sistema.

♦ Funciones para el programador de aplicaciones de la base de datos

- *SQL incorporado en lenguajes de tercera y cuarta generación.* SQL puede funcionar como un *sublenguaje incorporado* para extender el poder del lenguaje y utilizarlo en el acceso a la base de datos.

- *Diseño y formas sencillas para la interface con el usuario.* SQL proporciona los resultados de las consultas en forma de reportes.

♦ Funciones para el administrador de la base de datos

- *Respaldo y recuperación de la base de datos*

- *Performance y seguridad.* SQL puede ser utilizado para restringir la capacidad de un usuario, protegiendo los datos frente a accesos no autorizados.

Una de las principales tareas de un sistema informatico es el manejo y almacenamiento de datos. Para ocuparse de esta tarea, programas especializados conocidos como *sistemas de manejo de bases* de datos comenzaron a aparecer a finales de los setenta y principios de los setenta. Un sistema de manejo de base de datos, o DBMS (*Database Management System*), ayudaba a los usuarios del computador a organizar y estructurar sus datos. Aunque inicialmente los sistemas de manejo de bases de datos fueron desarrollados para sistemas de maxicomputadores, su popularidad se ha extendido a minicomputadores, computadoras personales y estaciones de trabajo.

Durante los últimos años se ha popularizado un tipo especifico de DBMS, llamado sistema de manejo de bases de datos *relacional* (DBMSR). Las bases de datos relacionales organizan los datos en una forma tabular sencilla y proporcionan muchas ventajas sobre las bases de datos anteriores.

La siguiente figura muestra como funciona SQL. El sistema informatico de la figura muestra una base de datos y el programa que controla la base de datos o DBMS.

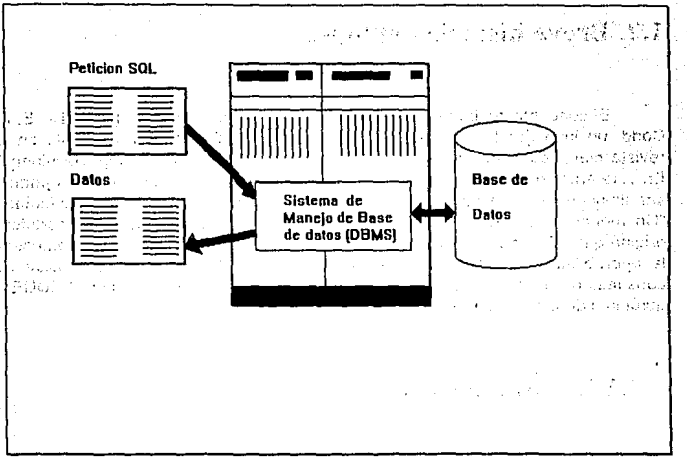


Fig. 1.1. Acceso de SQL a base de datos.

Cuando se necesita recuperar datos de la base de datos, se utiliza el lenguaje SQL para efectuar la petición. El DBMS procesa la petición, recupera los datos solicitados y los devuelve. Al proceso de solicitar datos de la base de datos y recibir los resultados de denomina *consulta (query)* a la base de datos; de donde proviene el nombre de Structured *Query* Language.

Por lo anterior descrito SQL es un lenguaje completo de control e interacción con un DBMS.

SQL es un lenguaje potente y a la vez muy sencillo de aprender. SQL se ha convertido en el lenguaje de bases de datos estándar para la utilización de bases de datos relacionales.

1.2. Breve historia del SQL.

El concepto de bases de datos relacional fue desarrollado por el Dr. E. F: Codd, un investigador de IBM, que en junio de 1970 publicó un artículo en la revista científica "Communications of the Associaton for Computing Machinery". En este articulo esquematizaba una teoría matemática de como los datos podían ser almacenados y manipulados usando una estructura tabular. El artículo titulado "Un modelo relacional de datos para grandes bancos de datos compartidos" originó que IBM realizara un importante proyecto de investigación para demostrar la operabilidad del concepto relacional, e incluía trabajos sobre lenguajes de consultas de bases de datos. Uno de estos lenguajes fue denominado SEQUEL, acrónimo de Structured English Query Languaje.

1.2.1. Los primeros productos.

En 1978 y 1979 al implementarse System/R en una serie de instalaciones de clientes de IBM para su evaluación, SEQUEL fue renombrada a SQL ó Structured Query Language (Lenguaje Estructurado de Consultas). A pesar de haber sido renombrado la pronunciación de SEQUEL siguió usándose y continua hoy en uso. En 1979 el proyecto de investigación concluyó dando como resultado la factibilidad de las bases de datos relacionales y que podían ser la base de un producto comercial útil.

Pronto este tipo de bases trajo la atención de un grupo de ingenieros de California que intuyeron un amplio mercado comercial para las bases de datos relacionales y en 1977 formaron una compañía llamada Relational Software, para construir un DBMS basado en SQL. El producto, de nombre Oracle, apareció en 1979 y se convirtió en el primer DBMS relacional comercialmente disponible. Oracle se adelanto dos años al primer producto de IBM y se ejecutaba en minicomputadores VAX. La empresa actualmente se llama Oracle Corporation y sigue siendo un vendedor importante de DBMS relacionales.

Profesores en los laboratorios informáticos de Berkeley de la Universidad de California estuvieron también investigando las base de datos relacionales, a mediados de los setenta. También ellos construyeron un prototipo de un DBMS relacional , y llamaron a su sistema Ingres. Muchos de los expertos de bases de datos comenzaron a interesarse en las bases de datos relacionales a consecuencia del proyecto Ingres, incluyendo a Britton-Lee y Sybase , el creador del producto SQL Server de Ashton-Tate/Microsoft.

En 1980 varios profesores abandonaron Berkeley y fundaron Relational Technology, que en 1989 seria renombrada Ingres Corporation, donde en 1981 anunciaron la versión comercial de Ingres.

Con Oracle e Ingres en la competencia de productos comerciales, IBM anuncio SQL/DS (SQL Data System) en 1981 y comenzó a distribuir su producto un año después.

En 1983 IBM introdujo también Database 2 (DB2), otro DBMS relacional para sus sistemas maxicomputadores, que operaba bajo el sistema operativo MVS, utilizado en los centros de datos de grandes maxicomputadores. La primera revisión de DB2 comenzó a entregarse en 1985, y DB2 se ha convertido desde entonces en el principal producto relacional de IBM, y el lenguaje de SQL para DB2 se ha convertido de hecho en el estándar entre los lenguajes de bases de datos.

Durante la ultima mitad de los ochenta, SQL y las bases de datos relacionales fueron rápidamente aceptadas como la tecnología de bases de datos del futuro. El rendimiento de los productos de bases de datos relacionales mejoró enormemente . Ingres y Oracle en particular, mejoraron substancialmente en cada versión, aumentando dos o tres veces el rendimiento de la versión anterior.

Un importante paso para la aceptación de SQL es el surgimiento de estándares SQL, que generalmente significa el estándar oficial adoptado por la American National Standards Institute (ANSI) y la International Standards Organization (ISO); aunque existen otros estándares SQL importantes como el definido por la System Application Architecture (SAA) de IBM y el estándar X/OPEN para SQL bajo UNIX.

Finalmente, al hacerse los computadores personales mas potentes y conectarse en redes de área local, comenzaron a necesitar un manejo de base de datos mas sofisticada. Los vendedores de bases de datos para PC adoptaron SQL como la solución a sus necesidades.

Cuando la industria informática entro en los noventa, las instalaciones SQL aumentaron en varios cientos de miles. SQL estaba claramente establecido como el lenguaje estándar de base de datos relacionales.

La Tabla 1 muestra algunos de los sistemas de manejo de bases de datos basados en SQL mas populares y sobre que tipos de sistemas operativos trabajan.

1.2.2. El mito de la portabilidad.

La existencia de estándares de SQL generó aseveraciones exageradas acerca de la portabilidad de SQL de que podía ser utilizado en cualquier sistema de manejo de base de datos basado en SQL. Las diferencias existentes entre los dialectos SQL de cada vendedor son suficientemente significativos para que una aplicación tenga que ser cambiada al tener que emigrar de una base de datos a otra.

Las diferencias entre los diferentes dialectos de SQL incluyen:

- *Códigos de error.* Cada implementación comercial tiene sus propios códigos de error.

- *Tipos de datos.* Los tipos de datos difieren entre un fabricante y otro, por ejemplo las cadenas de caracteres de longitud variable, los datos monetarios y los formatos de fecha y hora.

- *Tablas del sistema.* Estas tablas proporcionan información de la estructura de la propia base de datos. Y cada vendedor tiene su propia estructura.

- *Interfaz de programa.* Cada producto utiliza su propia técnica de interfaz con el programa de aplicación.

♦ *Diferencias semánticas* .Debido a que el estándar SQL especifica ciertos detalles como "definidos por el fabricante", es posible encontrar una misma consulta de dos implementaciones de SQL diferentes que arrojen resultados distinto. Estas diferencias ocurren en el manejo de valores NULL y la eliminación de filas duplicadas

♦ *Secuencias de ordenación.* Los resultados de una consulta serán diferentes si se ejecutan en un computador personal (con caracteres ASCII) o en un maxicomputador (con caracteres EBCDIC).

DBMS	Sistema en el que trabaja.
DB2	Maxicomputadores IBM bajo MVS.
SQL/DS	Maxicomputadores IBM bajo VM y DOS/VSE.
Oracle	Maxicomputadores, minicomputadores y PCs.
Ingres	Minicomputadores y PCs.
Sybase	Minicomputadores y LANs.
Informix-SQL	Minicomputadores y PCs basados en UNIX.
Unify	Minicomputadores basados en UNIX
OS/2 Extended Edition	Sistemas PS/2 de IBM basados en OS/2.
SQL Server	PC LANs basados en OS/2.
SQL Base	PC LANs basados en OS/2 y DOS.
dBASE IV	PCs y PC LANs.

Tabla 1.1. Principales DBMS basados en SQL.

1.3. Características y beneficios de SQL.

Algunas de las principales características de SQL son listadas a continuación:

- *Su independencia de los vendedores.* Una base de datos basada en SQL y los programas que la utilizan pueden convertirse de un DBMS a otro con un mínimo esfuerzo y poco reentrenamiento del personal. En el mercado han comenzado a aparecer herramientas que funcionan en varios productos de DBMS, tales como escritores de informes y generadores de aplicación. La independencia entre los vendedores proporciona es una de las razones mas importantes de su popularidad.

- *Portabilidad a través de sistemas de informáticos.* Los vendedores de SQL ofrecen sus productos que van desde computadores personales hasta redes de área local, minicomputadores y maxicomputadores. Las aplicaciones basadas en SQL que comienzan en sistemas monousuario pueden ser transferidas a sistemas mayores de minicomputadores y maxicomputadores cuando crecen. Los computadores personales pueden ser usados para construir sistemas prototipos basados en SQL antes de transferirlas a un sistema multiusuario.

- *Estándares SQL.* El hecho de que el American National Standards Institute (ANSI) y la International Standards Organization (ISO) hayan publicado conjuntamente un estándar oficial de SQL, así como en Europa el estándar X/OPEN, para aplicaciones basadas en UNIX ha servido como sello de aprobación de SQL, ha dado confianza a los compradores, acelerando su aceptación en el mercado.

- *Fundamento relacional.* SQL es un lenguaje para bases de datos relacionales y conjuntamente con las bases de datos de tipo relacional, se ha popularizado. La estructura tabular, de filas y columnas de una base de datos relacional, hace que el lenguaje SQL se haga simple de entender.

- *Estructura de alto nivel en ingles.* Las sintaxis de las sentencias SQL parecen sencillas frases en ingles, lo que lo hace mas fácil de entender. Dependiendo del vendedor, los nombres de las columnas y las tablas pueden ser largos y descriptivos.

- *Consultas interactivas.* SQL es un lenguaje de consultas interactivas que proporciona a los usuarios acceso rápido a los datos almacenados. Utilizando SQL de esta manera un usuario puede obtener respuestas incluso a cuestiones complejas en cuestiones de minutos o segundos, en contraste a los días o semanas que tardaría en programar un programa en COBOL, por ejemplo. Esta característica hace que los datos sean mas accesibles y pueden ser utilizados para tomas de decisiones oportunas y documentadas.

- *Acceso a la base de datos mediante programas* . SQL puede trabajar en modo interactivo o en modo programado, utilizando las mismas sentencias en los dos modos de acceso, de modo que las partes del programa donde se requiera SQL para accesar la base de dato pueden ser ejecutadas y probadas en modo interactivo para después incorporarse al programa de aplicación.

- *Vistas múltiples de datos.* Utilizando SQL, el creador de la base de datos puede dar diversas vistas a los diferentes usuarios. De acuerdo a la capacidad de cada usuario se le puede permitir accesar cierta información y prohibirle el acceso a otra. Ademas los datos procedentes de varias tablas pueden combinarse y presentarse al usuario como una sola tabla. Esta característica cumple una doble función al presentar información personalizada a cada usuario y mejorar la seguridad de la base de datos.

- *Lenguaje completo de base de datos.* Aunque inicialmente SQL fue diseñado como un lenguaje de consulta, su potencia fue desarrollada y se ha convertido en un lenguaje completo y consistente para la creación de bases de datos, manejo de su seguridad, actualización de datos, uso compartido de los mismos entre varios usuarios concurrentes.

- *Definición dinámica de datos.* Utilizando SQL, la estructura de una base de datos puede ser modificada y ampliada dinámicamente mientras los usuarios la están accesando. Este es un avance importante ya que los demás lenguajes impiden el acceso a la base de datos mientras su estructura esta siendo modificada.

1.4. El impacto de SQL.

Como lenguaje estándar de las bases de datos relacionales, SQL ha tenido un fuerte impacto en todas las áreas del mercado informático. IBM lo ha adoptado como tecnología unificadora de bases de datos para su línea de productos. Todos los vendedores de minicomputadores ofrecen bases de datos basadas en SQL, y las bases de datos basadas en SQL dominan el mercado de los sistemas informáticos en UNIX. SQL también está influyendo en el mercado de las bases de datos para computadores personales, a la vez que estos dan paso a redes de computadoras personales y a OS/2. Actualmente SQL emerge con gran fuerza en el proceso de transacciones en línea.

1.4.1. SQL en IBM.

IBM cuenta con cuatro productos estratégicos en base de datos para cada una de sus familias de computadores :

- *DB2* . El manejador de bases de datos relacional insignia de IBM. Corre en maxicomputadores IBM que corren en MVS.

- *SQL/DS.* Es el sistema de bases de datos relacional para VM, otro sistema operativo de los maxicomputadores IBM.

- *SQL/400.* Una implementación SQL para sistemas de minicomputadores de IBM que soporta la base de datos relacional interna de AS/400.

- *OS/2 Extended Edition.* Versión mejorada del sistema operativo para computadoras personales de IBM, incluye un manejador de base de datos basado en SQL.

1.4.2. SQL en minicomputadores.

Los minicomputadores fueron uno de los mercados donde inicialmente se concentraron los sistemas de bases de datos basados en SQL. Oracle e Ingres fueron comercializados inicialmente en minicomputadores VAX/VMS de Digital, y aunque han migrado a otras plataformas, VAX/VMS sigue siendo parte muy importante de sus ventas. Sybase, un sistema de bases de datos más reciente, especializado en procesamiento de transacciones en línea, también se destino a VAX como plataforma inicial.

Los vendedores de minicomputadores han desarrollado también sus propias bases de datos relacionales propietarias caracterizadas por SQL. Hewlett-Packard ofrece Allbase, una base de datos que soporta su dialecto HPSQL. Además muchos de los vendedores de minicomputadores revenden bases de datos relacionales procedentes de vendedores de bases de datos independientes. Por ejemplo, Digital ofrece Ingres para sus productos VAX basados en UNIX, y Oracle está disponible en varias docenas de vendedores de minicomputadores.

1.4.3. SQL en sistemas UNIX.

UNIX se popularizo en los ochenta como sistema operativo estándar independiente de los vendedores. Corre en varios sistemas de computadoras, desde estaciones de trabajo a maxicomputadores.

En la mayoría de los sistemas UNIX se encuentran disponibles Ingres y Oracle, que son adaptaciones especiales para UNIX. Existen otras dos: Informix y Unify, que fueron escritas especialmente para UNIX, así como una quinta opción de Sybase.

1.4.4. SQL y el procesamiento de transacciones.

SQL y las bases de datos relacionales originalmente tuvieron poca aceptación en las aplicaciones de procesamiento de transacciones en línea (OLTP). Fue hasta 1986 cuando un nuevo vendedor Sybase introdujo una nueva base de datos basada en SQL, diseñada especialmente para aplicaciones OLTP. EL manejador de base de datos de Sybase corría en minicomputadores VAX/VMS y en estaciones de trabajo Sun, y se centraba en obtener el máximo rendimiento en línea.

En abril de 1988 IBM entro al OLTP relacional con DB2 Versión 2, cuyos programas de prueba demostraron que la nueva versión operaba arriba de 250 transacciones por segundo en grandes maxicomputadores.

Debido a los avances en la tecnología para las bases de datos relacionales, los diferentes productos SQL han reportado frecuentemente tasas de transacciones cada vez mas altas, debido también, a la mayor potencia en el hardware.

1.4.5. SQL en computadores personales.

Desde el inicio de las computadoras personales las bases de datos han sido populares en las computadoras personales. El producto mas popular ha sido dBASE de Ashton-Tate. Aunque las bases de datos para PC´s presentan las bases de datos en una forma tabular, les falta la potencia de un DBMS relacional y un lenguaje de base de datos relacional como SQL.

Las primeras bases de datos basadas en SQL para computadores personales fueron versiones de los principales productos para minicomputadores que dificilmente se ajustaban a los recursos de las computadoras personales : Profesional Oracle lanzado a la venta en 1984, requería dos megabytes para correr de memoria para correr en una IBM PC, y Oracle para Macintosh, anunciado en 1988, requería igualmente dos megabytes, así mismo surgieron versiones de Ingres para MS-DOS, Informix-SQL anunció en 1986 una versión para PC. En el mismo año Gupta Technologies, una empresa fundada por un ex director de Oracle presentó SQLBase, una base de datos para redes de área local PC.

Cuando IBM y Microsoft anunciaron en abril de 1987 OS/2, además de OS/2, IBM anunció un OS/2 Extended Edition (OS/2 EE) propietario con una base de datos SQL interna; con esto, IBM manifestó que SQL era tan importante que pertenecía al sistema operativo del computador.

Ya que Microsoft era fabricante y distribuidor de OS/2 de otros fabricantes de computadores personales, Microsoft necesitaba una alternativa a Extended Edition. La manera como respondió Microsoft fue comprando la licencia del DBMS Sybase, que había sido desarrollado para VAX, y comenzó a portarlo al OS/2. En enero de 1988 se anuncio que venderían conjuntamente con Ashton-Tate el nuevo producto resultante: SQL Server.

El lanzamiento de SQL Server y la introducción de OS/2 Extended Edition atrajo la atención hacia el potencial de SQL sobre redes de área local basados en OS/2. Esto hizo que a finales de 1989 aparecieran una gran cantidad de productos enfocados a servidores de bases de datos OS/2:

- *OS/2 Extended Edition.*

- *SQL Server de Microsoft.*

♦ *Oracle Server para OS/2.*

♦ *SQL Base de Gupta.*

La arquitectura cliente/servidor también estimuló el desarrollo de aplicaciones frontales para los nuevos servidores; quizá el más significativo de estos frontales resultó dBASE IV que se anunció que funcionaría como frontal de SQL Server y de OS/2 Extended Edition. Cada vendedor ofertó sus propios productos frontales, incluyendo herramientas de desarrollo de aplicaciones, consultas orientadas a ventanas, facilidades de entrada y modificación de datos y otras.

1.5. El futuro del SQL.

SQL es una de las tecnologías más importantes que conforman y conducen el mercado informático hoy en día. SQL se ha convertido en el lenguaje estándar de bases de datos relaciones. Las evidencias muestran claramente la importancia del SQL:

- Las bases de datos más importantes de IBM, están basadas todas en SQL. Desde maxicomputadores hasta computadores personales.
- Los principales vendedores de DBMS independientes ofrecen o pretenden ofrecer productos basados en SQL.
- El más importante vendedor de bases de datos para PC, Ashton-Tate, se ha inclinado ante la presión del mercado y ha insertado SQL en dBASE IV. Por otra parte Microsoft se unió con Sybase para presentar SQL Server de Microsoft y en la versión de Foxpro para Windows se incorporó un módulo para realizar consultas SQL.

1.5.1. Bases de datos distribuidas.

Una de las principales tendencias de la computación en los últimos años ha sido el pasar de grandes computadores centralizados a redes distribuidas de sistemas informáticos. Los sistemas en maxicomputadores corporativos están pasando a sistemas departamentales más pequeños.

A medida que los sistemas crecen, los computadores y las redes de computadores se extienden a través de las organizaciones. Los datos se extienden a través de varios sistemas, cada uno de ellos con su propio DBMS. Con frecuencia los diferentes sistemas informáticos proceden de diferentes fabricantes.

Los principales vendedores de DBMS están orientando sus esfuerzos hacia las bases de datos distribuidos. Idealmente, entre otras cosas se desea conseguir :

- *Transparencia de ubicación.* El usuario no se debería de preocupar acerca donde se encuentran físicamente los datos. El DBMS debe de mostrar todos los datos como si fueran locales, sin importar que se encuentre en otro sistema, aún si se encuentra situado a kilómetros de donde se hace la petición.
- *Sistemas heterogéneos.* El DBMS debería soportar datos almacenados en sistemas diferentes, con diferentes arquitecturas, como PC, estaciones de trabajo, minicomputadores y maxicomputadores.
- *Consultas actualizaciones y transacciones distribuidas.* El usuarios debería de ser capaz de actualizar y consultar cualquier dato de cualquier tabla tanto si la tabla esta en el sistema local o se encuentra en un sistema remoto. Se debe mantener la integridad de la base de datos (distribuida).

Ningún producto DBMS actual se acercan a este ideal. Las versiones posteriores de los productos SQL, prometen acercarse más al objetivo de proporcionar acceso universal y transparente en bases de datos distribuidas.

Gran parte de la investigación académica en el área de las bases de datos durante los últimos años se ha concentrada en las llamadas *bases de datos orientadas a objetos*. Algunos estudiosos de las bases de datos proclaman que son la siguiente generación de las bases de datos y se pronostica un serio reto a las bases de datos relacionales hacia el final de siglo.

A diferencia del modelo relacional, donde el artículo de Codd proporciona una clara definición de una base de datos relacional, no existe un acuerdo sobre qué es una base de datos orientada a objetos. Cuando se refieren a ésta, se refieren a una base de datos que utiliza los mismos principios organizacionales que la programación orientada a objetos:

- *Objetos.* En una base de datos orientada a objetos, cualquier elemento de la base es considerado y manipulado como un objeto. La organización tabular de fila/columna de una base de datos relacional es sustituida por colecciones de objetos.

- *Clases.* Las bases de datos orientadas a objetos sustituyen la noción relacional con una noción jerárquica de clases y subclases. Por ejemplo una clase denominada VEHICULOS se puede descomponer en las subclases COCHES y BOTES.

- *Herencia.* Los objetos heredan las características de su clase y de todas las clases de nivel superior a la que pertenecen. Por ejemplo una característica de VEHICULOS es que sirven para transportarse, las clases COCHES y BOTES, heredan esta característica.

Las bases de datos orientadas a objetos se adecuan a aplicaciones que implican tipos de datos complejos, como documentos o gráficos. Por ejemplo un documento puede representarse como un único objeto.

Los proponentes declaran que las bases de datos orientadas a objetos reemplazarán en el futuro al modelo relacional de bases de datos. Los críticos dicen que las bases de datos orientadas a objetos son simplemente una reformulación de las bases de datos jerárquicas, con una nueva terminología.

Las bases de datos orientadas a objetos jugarán probablemente un papel creciente en mercados especializados, como el procesamiento de documentos e interfaces de usuario gráficas. Sin embargo es poco probable que tengan impacto importante en aplicaciones de procesamiento de datos durante los próximos años.

Realmente SQL esta demasiado encarrilado para que lo pueda sustituir algún DBMS orientado a objetos en algún momento cercano. Sin embargo siempre se dice lo mismo al aparecer un nuevo producto en el mercado. El desarrollo de las bases de datos orientadas a objetos si amenaza el dominio continuo de SQL en las próximas décadas. Sin embargo, lo dicho hasta aquí de la tendencia de las bases de datos orientadas a objetos son solamente especulaciones que el tiempo aclarará.

1.6. Conceptos básicos de SQL/Bases de datos relacionales.

1.6.1. Bases de datos.

A menudo los términos de "base de datos" y "archivo" se emplean como si fueran sinónimos, sin embargo existen diferencias importantes entre ellos. Un archivo es un conjunto de registros en el que cada registro esta formado por múltiples elementos llamados campos. Una base de datos, normalmente abarca un gran número de archivos y ademas contiene las relaciones existentes entre los registros de los diferentes archivos que conforman la base.

También podemos definir una base de datos como un conjunto de archivos creados con un sistema de manejo de base de datos, o DBMS (*Database Management System).* La base de datos contiene la definición de cada campo de los archivos que la conforman de tal manera que al estar disponibles para los usuarios se puedan eliminar los datos redundantes o al menos minimizarlos.

1.6.2. Sistemas de manejo de bases de datos (DBMS).

Un sistema de manejo de bases de datos es la porción mas importante del software de un sistema de base de datos. Las siglas DBMS (*Database Management System)* se utilizaran a lo largo de este trabajo para referirnos al sistema de manejo de bases de datos.

Un DBMS es una colección de numerosas rutinas de software con una tarea especifica. Las funciones principales de un DBMS son:

- Creación y organización de la base de datos.
- Establecer y mantener las trayectorias de acceso a la base de datos de tal manera que los datos en cualquier parte de la base se puedan accesar rápidamente.
- Manejar los datos de acuerdo con las peticiones de los usuarios.
- Mantener la integridad y la seguridad de los datos.
- Registrar el uso de la base de datos.

El DBMS interpreta las peticiones del usuario para recobrar información de la base; es decir un DBMS funciona como interface entre las peticiones del usuario y la base de datos. Los accesos a la base de datos pueden teclearse desde una terminal o codificarse en lenguajes de alto nivel. Una petición de consulta deberá de pasar varias etapas en el software y en el sistema operativo antes de que se pueda accesar la base de datos física. La siguiente figura muestra al DBMS como interface entre las peticiones de usuario y la base de datos.

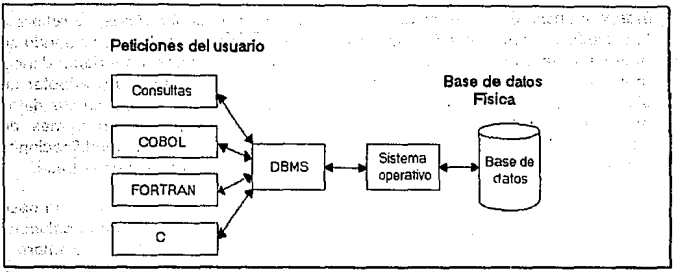


Fig. 1.2. El DBMS como interface entre las peticiones de usuario y la base de datos.

El DBMS responde a un petición de datos llamando a los subprogramas apropiados, cada uno de los cuales ejecuta su función para interpretar la petición y para localizar los datos solicitados en la base y presentarlos en el orden solicitado. De esta manera el DBMS protege al usuario de la tediosa tarea de organizar el almacenamiento de los datos y posteriormente recuperarlos. El DBMS permite a los usuarios el acceso a datos en cualquier parte de la base sin necesidad de conocer su organización dentro del dispositivo de almacenamiento.

El papel que juega el DBMS es análogo a la de un vendedor en una farmacia. El cliente que llega a la farmacia solicita las medicinas deseadas anotadas en una receta médica. El vendedor lee la receta, busca las medicinas en su bodega y las entrega al cliente; el cliente, que es el que solicita las medicinas no tiene idea de dónde y cómo están almacenadas las medicinas en la bodega. Similarmente el usuario solamente necesita solicitar los datos y el DBMS se encargará de proporcionarlos.

1.6.3. Objetivos de un DBMS.

Los objetivos que trata de alcanzar el sistema manejador de base de datos son :

- *Independencia lógica de los datos.* La *Independencia de los datos*, se refiere a la protección contra modificaciones de los programas de aplicación cuando se altera la organización física o la estructura lógica de la base. La Independencia lógica de los datos es uno de los motivos más importantes para adoptar un sistema base de datos. El DBMS alcanza la independencia lógica de sus datos al separar las definiciones de los datos de éstos en los programas de aplicación. Protegiendo de esta manera a los programas de modificaciones excesivas debidas a cambios en las estructuras de los archivos de la base.

  Por ejemplo, alguna columna podría eliminarse de la definición de la base de datos, pero el programa que utilice la tabla en donde se eliminó la columna, puede seguir funcionando correctamente sin modificación alguna, siempre y cuando los campos eliminados no sean referenciados dentro del programa.

- *Minimizar la redundancia de los datos.* Una característica de un sistema de base de datos es que todos los datos de los distintos programas y usuarios se unen e integran en la misma base para ser compartidos por todos. Minimizando así la redundancia de los datos.

  Por ejemplo la misma tabla donde se tienen almacenados todos los nombres de las materias puede ser utilizado, lo mismo por el Jefe de Carrera, para formar los grupos, que por el área de Servicios Escolares para imprimir los comprobantes de inscripción a los alumnos.

- *Integridad de los datos.* Esto se refiere a las medidas de seguridad para mantener correctos los datos en la base. Las áreas más importantes en lo que concierne a la seguridad en la integridad de los datos son :

  a) *Validación de los datos de entrada.* La persona que maneja la base de datos puede especificar condiciones para la validación o comprobación de la consistencia, además de algunas medidas de integridad que son proporcionadas automáticamente por el sistema, por ejemplo:

  1. *Validación del tipo de datos:* .El DBMS manda un mensaje de error cuando el dato de entrada es inconsistente con el tipo definido en la base de datos (caracter, decimal, entero, etc.).

2. *Validación del valor de los datos.* El contenido de cierto campo de entrada puede validarse para un rango de valores definidos previamente. Por ejemplo podríamos tener un campo llamado ESTACIÓN que aceptara solamente las cuatro estaciones del año : "Primavera", "Verano", "Otoño" e "Invierno".

3. *Validación sobre los valores de las claves primaria y secundaria.* Cada registro de la base de datos puede estar representado por un valor único. En la especificación de la base de datos, a cada tipo de registro se le designa como clave principal un campo o una combinación de campos; esta clave se puede usar para identificar de manera única a cada registro en el archivo.

   El DBMS asegura que el valor de la clave primaria sea único y no nulo. En caso de que el valor de entrada contenga un valor duplicado o nulo generará un mensaje de error.

   En la definición de la base de datos se debe de especificar si los valores de clave secundaria admitirá valores repetidos. Si no permite duplicidad el DBMS realiza las mismas validaciones que para la clave primaria.

4. *Integridad referencial.* La idea principal de la integridad referencial es asegurar que no existan registros hijo si no existe el padre correspondiente. Tratemos de explicar esto : supongamos que tenemos en una base de una editorial dos archivos TÍTULOS, que contiene los títulos que la editorial distribuye y VENTAS, si tratamos de capturar una venta de un título que no existe el DBMS nos mandará un mensaje de error. Análogamente si damos de baja un registro en TITULO debemos de dar de baja toda las ventas asociadas a él. Sin embargo pueden existir registros padre sin necesidad de algún hijo asociado a él. Siguiendo con nuestro ejemplo: podemos tener títulos que no tengan ningún pedido en VENTAS.

b) Interferencia debido a la concurrencia. En el ambiente de bases de datos es común que varios usuarios traten de usar simultáneamente la misma de base de datos, y el sistema tratará de cumplir tantas peticiones como le sea posible en base a un sistema compartido. Sin embargo uno de los problemas de concurrencia es que los programas concurrentes pueden interferir entre sí durante una actualización simultánea del mismo registro.

Supongamos que tenemos dos usuarios de la base de datos de una editorial: el usuario 1 trata de actualizar un título de un libro en el archivo de TÍTULOS, al mismo tiempo el usuario 2 accesa el mismo registro del archivo para actualizar el precio. Para empezar los dos programas leen el mismo registro de la base de datos y lo cargan a su memoria y efectúan sus cambios a los respectivos campos en la memoria. Primero el usuario 1 efectúa la actualización del título del libro en la base de datos, cuando usuario 2 actualice el precio del libro cancelará la operación del usuario 1 ya que en la memoria aún guarda el título anterior.

Una manera de prevenir tales interferencias es secuenciar las actualizaciones, es decir, programar el sistema de tal manera que complete una actualización antes de efectuar la siguiente.

Cada DBMS tiene diferentes enfoques y grados de sofisticación para reforzar la integridad de datos. Idealmente el DBMS, sin necesidad del programador, debe evitar la concurrencia y procurar mantener la integridad referencial; un DBMS con mecanismos automáticos para la integridad muy sofisticado sería muy costoso de implantar y su realización llevaría mucho tiempo.

Si en la base son detectados errores, el DBMS proporciona mecanismos para restaurar la base.

- *Seguridad de los datos.* Debido a que la base es compartida. El DBMS protege los datos de accesos y actualizaciones no autorizadas.

Se puede controlar que sólo ciertas porciones de la base sean accesibles para los usuarios y/o programadores.

- *Eficiencia.* La meta general de los sistemas de bases de datos es proporcionar respuesta rápida y económica a las peticiones de los usuarios. Para establecer la eficiencia en la base de datos, es importante tener trayectorias de acceso para la recuperación y el control de los datos. Una de las herramientas para el control de la base es el diccionario de datos que contiene documentación acerca de la base, éste diccionario puede estar aislado del sistema DBMS o formar parte de él.

1.6.4. Enfoque jerárquico.

El DBMS de enfoque jerárquico, usa estructuras de árbol para representar arreglos lógicos.

Familiaricémonos un poco con la terminología típica para estructuras jerárquicas:

a) *Nodo.* Es una entidad que ocupa una posición en una estructura de datos. Un nodo puede compararse con un archivo conceptual. Contiene sólo un tipo de registro.

b) *Raíz.* Un árbol sólo tiene un nodo raíz. Es el nodo sobre el nivel más alto de una estructura de árbol.

c) *Hojas.* Aquellos nodos que no tienen ningún subordinado se llaman hojas. Es el nivel más bajo en la estructura de árbol.

d) *Padre e hijo.* Con excepción del nodo raíz, cada nodo esta conectado con un solo nodo en el nivel superior. El nodo de nivel más alto se llama *padre* y el subordinado se llama *hijo.*

Las estructuras de árbol ocurren de manera natural en muchas organizaciones, porque algunas entidades tienen orden jerárquico intrínseco. Por ejemplo una Universidad puede ofrecer diferentes carreras. Cada carrera tiene distintas materias y cada materia cierta cantidad de alumnos inscritos.

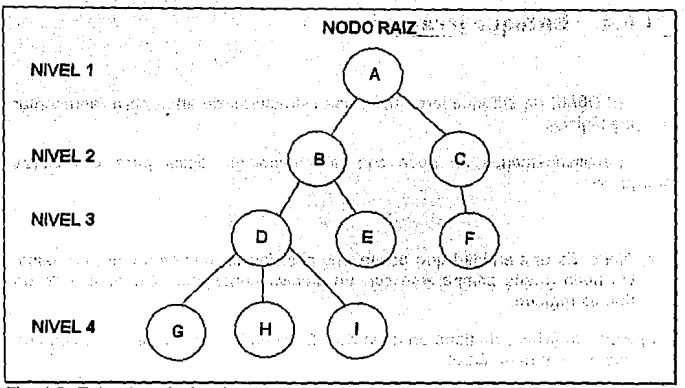


Fig. 1.3. Estructura jerárquica.

Un árbol se compone de *nodos*. Cada nodo contiene datos que constituyen un tipo de registro. En la figura 3 observamos: El nodo único en el nivel 1 es un nodo *raíz*. Los nodos E, F, G, H e I, sin ningún nodo subordinado se llaman hojas. Con excepción de la raiz cada nodo tiene solo un nodo *padre*.

La figura 4. muestra una estructura jerárquica con registros de ejemplo para la Escuela Nacional de Estudios Profesionales Aragón. La figura muestra solo dos carreras: computación y electrónica; y tres materias: Sistemas Operativos, Programación Estructurada y Circuitos Lógicos. En cada materia tenemos inscritos un determinado número de alumnos.

- Las tipos de registro en el enfoque jerárquico están dispuestas en estructura de árbol.
- Los diferentes tipos de registro en un archivo jerárquico están enlazados por medio de relaciones uno-a-muchos, pero la relación muchos-a-muchos no puede ser enlazados directamente. En la figura 4 la relación Escuela-Carrera es uno a muchos ya que una escuela ofrece varias carreras.
- Cada nodo consta de uno o más datos.

- Las ocurrencias de los padres pueden tener distintos número de ocurrencias de los hijos. En la figura, la materia de Programación Estructurada tiene 2 alumnos mientras que Sistemas Operativos tiene 4 alumnos.

- Un registro hijo no puede existir si no existe un registro padre. Por ejemplo un estudiante no puede inscribirse en, digamos Álgebra Lineal si Álgebra Lineal no es una materia que se ofrezca en la E.N.E.P. Aragón.

- Cuando se elimina un registro padre, también se deben borrar todos los registros hijos. Por ejemplo si se elimina la carrera de electrónica también se debe eliminar la materia de Circuitos Lógicos y a su vez a los alumnos inscritos en dicha materia.

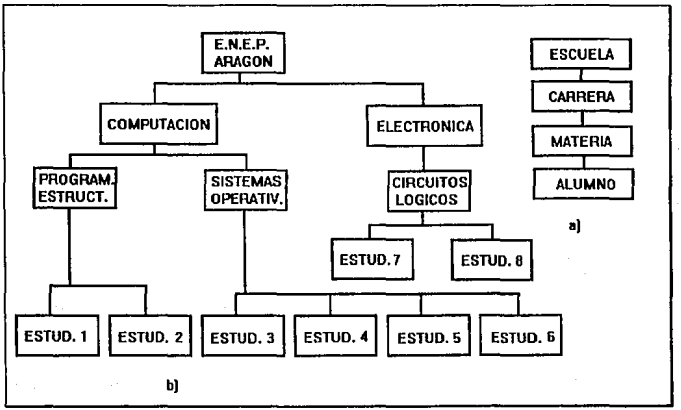


Fig. 1.4. Ejemplo de una base de datos jerárquica.

1.6.5. Enfoque de red.

La sencilla estructura de una base de datos jerárquica sufrió modificaciones al tratar de almacenar datos con una estructura más compleja. Una estructura de datos en red, abarca más que la estructura de árbol porque un nodo hijo puede tener mas de un padre. En otras palabras, la restricción de que un árbol jerárquico cada hijo puede tener solo un padre, se hace menos rígida

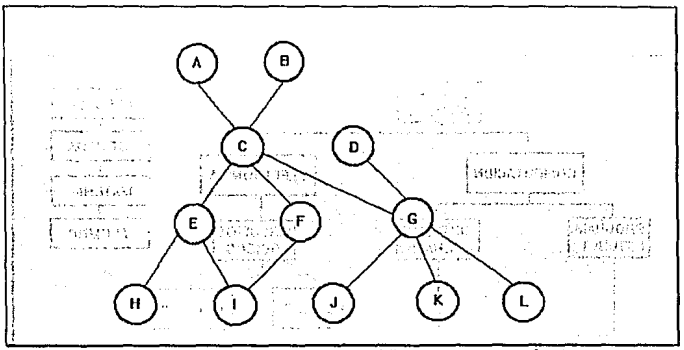


Fig. 1.5. Estructura de red en la cual un nodo hijo puede tener más de un padre.

En la figura observamos que A y B son nodos padre de C, mientras que C y D son padres de G.

Por ejemplo en una base de datos para procesar transacciones en una casa de cambio, una factura de una venta de dólares podría participar en varias relaciones padre/hijo diferentes, ligándolo al cliente que lo solicitó, al vendedor que lo aceptó y al producto ordenado como se muestra en la Figura 6.

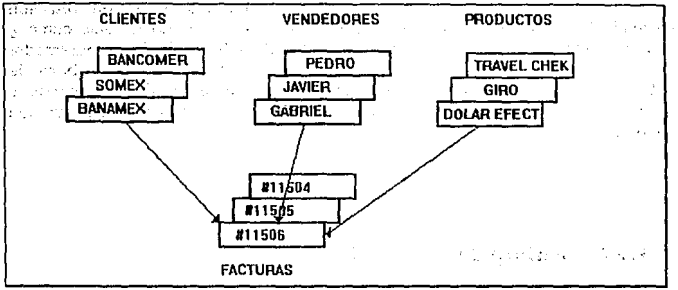


Fig. 1.6. Múltiples relaciones padre/hijo.

Para el desarrollo de aplicaciones que requerían este tipo de estructura fue desarrollado el nuevo modelo de datos en red. En la terminología de bases de datos en red la relación padre/hijo es conocida como *conjuntos.*

Para un programador, navegar a través de la base de datos en red es muy parecido a hacerlo en una base de datos jerárquicos. Un programa de aplicación puede:

- Hallar un registro padre mediante una clave (por ejemplo el # de vendedor).
- Descender al primer hijo en un conjunto en particular (la primera factura de un vendedor).
- Moverse lateralmente de un hijo al siguiente dentro del conjunto (la siguiente factura efectuada por el mismo vendedor).
- Ascender desde un hijo otro padre (el cliente al que se le vendieron los dólares).

A pesar de las mejoras introducidas con la nueva estructura en red, las bases de datos en red también presentaban sus desventajas. Al igual que las bases de datos jerárquicas, resultan muy rígidas. Las relaciones de conjunto y la estructura de los registros tienen que ser especificadas de antemano. Modificar su estructura requiere típicamente la reconstrucción de la base de datos completa.

Tanto las bases de datos jerárquicas como las bases en red son herramientas para los programadores. Para responder preguntas tales como ¿ cual vendedor hizo el mayor número de ventas durante el dia ? el programador tiene que escribir un programa que recorra su camino a través de la base de datos. El desarrollo del programa con frecuencia dura varios días o semanas, y para el momento en que es concluido la información tal vez ya no sea tan importante.

1.6.6. Enfoque relacional.

En el sistema de base de datos relacional, una estructura lógica se representa por medio de tablas bidimencionales llamadas *relaciones* (una relación es equivalente en concepto a un archivo).

La principal ventaja del enfoque relacional está en la simplicidad de su representación lógica de la base de datos y en la flexibilidad para establecer relaciones de datos por medio de campos de conexión. Todas las entidades están representadas como tablas separadas y no están colocadas en una jerarquía fija como en el caso de las estructuras de árbol o red.

El modelo de datos relacional fue descrito por primera vez por el Dr. Codd en 1970. Este modelo elimina las estructuras explícitas padre/hijo de las bases de datos de red y jerárquica. Los primeros sistemas de manejo de base de datos relacionales fallaron en implementar algunas partes del modelo de Codd. Conforme el concepto relacional crecía en popularidad , muchas bases de datos que se llamaban a sí mismas "relacionales" no lo eran.

En respuesta la corrupción de el concepto del término relacional, el Dr. Codd escribió un artículo en 1985 con doce reglas que debería cumplir cualquier base de datos que quisiera llamarse relacional. Las doce reglas de Codd fueron aceptadas desde entonces como la definición de un DBMS verdaderamente relacional Estas reglas son descritas en el apéndice A (con una breve explicación de cada una de ellas); sin embargo a continuación se presenta una definición mas informal:

Una base de datos relacional es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas.

1.6.6.1. Base de datos Ejemplo.

A continuación se describe el contenido de las tablas de la base de datos de la Escuela Nacional de Estudios Profesionales (E.N.E.P) la cual se utilizará a lo largo de todo este trabajo de tesis, y proporcionará la base para la mayoría de los ejemplos. El apéndice B describe la estructuras de las tablas que conforman esta base de datos así como los datos que contienen.

- MATERIA. Contiene toda la información acerca de las materias impartidas en la E.N.E.P. los créditos que se dan al cursarla, el precio del laboratorio en caso de llevar y la carrera a la cual pertenece.
- CARRERA. Tiene el registro de cada departamento académico (o carrera). El edificio donde se encuentra y el número de empleado académico que corresponde al jefe de carrera o director del departamento.
- GRUPO. Tiene un registro por cada grupo que es ofrecido en cada materia. Tiene el número de grupo, el número de empleado académico del maestro que imparte la clase, el día, la hora y el salón el que se imparte.
- INSCRIPC. Contiene un registro que hace constar que un alumno esta inscrito en una clase. Contiene el número de curso, el departamento al que pertenece, número de cuenta del estudiante, fecha y hora de la inscripción.
- ACADEMIC. Contiene un registro por cada miembro académico de la Escuela. Contiene su número de empleado, nombre, dirección, fecha de contratación, número de ayudantes asignados a él, sueldo y el departamento al que pertenece.
- ESTUDIAN. Contiene datos generales de los alumnos. Contiene su número de cuenta, nombre, dirección y carrera que cursa.

A continuación se presenta la estructura de nuestra base de datos de ejemplo:

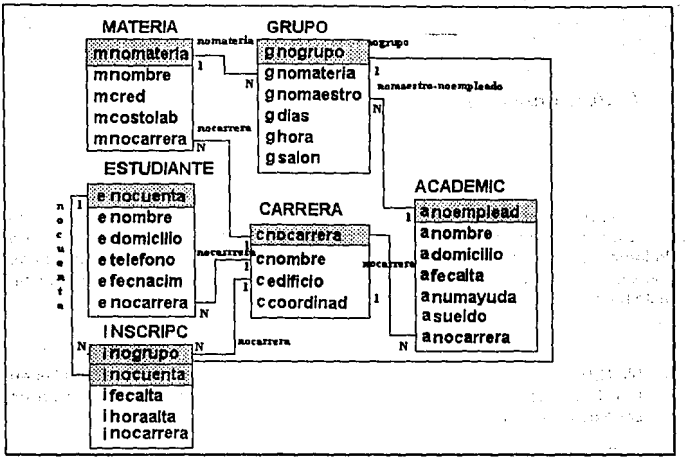


Fig. 1.7. Estructura de la base de datos de la E.N.E.P. que sirve de ejemplo en el presente trabajo de tesis.

1.6.6.2. Tabla.

Una tabla también es conocida como una *relación.* Una tabla es un archivo conceptual que consta de ocurrencias con la misma composición de campos, en otras palabras, es una disposición rectangular de filas y columnas de los valores de los datos. Cada tabla tiene un nombre único asignado. La siguiente figura representa la tabla MATERIA que forma parte de la base de datos de la E.N.E.P.

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.00	32
0134	Sistemas Digitales	8	50.00	32
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32
0559	Memorias y Periféricos	10	100.00	32
0561	Microcomputadoras	10	500.00	32
0028	Análisis Dinámico de Máquinas	8	100.00	38
0130	Elementos de Máquinas	8	50.00	38
0024	Circuitos Eléctricos	10	150.00	40
0138	Dispositivos Electrónicos	10	90.00	40

Fig. 1.8. Tabla MATERIA.

Observamos que cada ***fila*** representa una entidad física, es decir toda la información contenida en una fila se refiere a una sola materia. También vemos que todas las filas de la tabla contienen el mismo número y tipo de columnas.

Cada ***columna*** vertical de la tabla MATERIA representa un elemento de datos que esta almacenado en la base para cada materia. Por ejemplo, la columna MCOSTOLAB contiene el costo del laboratorio para cada materia. La columna MNOCARRERA contiene el número de departamento que hace referencia a la carrera a la que pertenece la materia.

Cada columna puede contener un solo valor por cada fila, es decir no se permiten los arreglos de valores en una columna.

Cada columna en una tabla puede ser de un solo tipo (numérico, caracter fecha... etc.) para todas las filas. Por ejemplo todos los valores que tiene la columna MCOSTOLAB son cantidades de dinero, los valores de la columna MCRED son valores enteros que representan los créditos que obtiene un alumno al cursar alguna materia.

El conjunto de valores que puede contener una columna se conoce como ***dominio***. Por ejemplo la columna de créditos solo puede contener valores positivos entre el cuatro y el 10, y ese es su dominio. El dominio de la columna NOMBRE es el conjunto de todas las materias impartidas en la E.N.E.P.

Cada columna de una tabla debe de estar referenciada por un *nombre de columna.* que se escribe generalmente como encabezado en la parte superior de la columna. Aunque las columnas de una tabla deben de tener todas nombres diferentes, puede ser que una columna tenga el mismo nombre en otra tabla.. Por ejemplo el nombre de tabla NOMBRE puede ser utilizado en la tabla MATERIA para describir el nombre de la materia, y el mismo nombre de columna NOMBRE puede servir en la tabla ACADEMIC para describir el nombre del empleado.

Aunque no existe un límite de columnas para una tabla, la mayoría de los productos manejan 255. Para que una tabla exista debe de contener al menos una columna. El orden de las columnas queda determinado en el momento de construirlas.

Una tabla puede contener cualquier número de filas inclusive cero, que es el caso cuando se acaba de construir la tabla, contiene la estructura pero simplemente no tiene datos. Muchos productos SQL no limitan el número de filas que puede contener una tabla y los que lo hacen tienen generalmente un límite muy alto (dos millones o más).

1.6.6.3. Clave Primaria.

Una o más columnas con valores que son únicos en la tabla y pueden ser usados para identificar las filas de esa tabla de manera única es conocida como *llave o clave primaria.* La llave primaria es un valor que nunca es nulo.

Observando la tabla MATERIA vemos que tanto MNOMATERIA como MNOMBRE pueden servirnos como llave primaria para la tabla, pero como el nombre es un poco largo y difícil de recordar se opta por escoger MNOMATERIA como llave primaria.

Consideremos la ambigüedad que se produce cuando una tabla contiene dos o mas registros que no pueden distinguirse unos de otros. Usando MNOMATERIA como la clave primaria, establecemos que un número de materia puede utilizarse únicamente para identificar un curso.

En la tabla INSCRIPC que contiene el registro de un alumno en determinado grupo, observamos que la clave primaria es una combinación de dos columnas. INOGRUPO y INOCUENTA, INOGRUPO identifica un grupo en el que se imparte alguna materia, pero por cada alumno inscrito en él se tendrá que repetir el mismo valor; por otro lado INOCUENTA se repetirá cuándo el mismo alumno este inscrito en varios grupos a la vez, por lo que debe utilizarse una combinación de ambas columnas para identificar cada fila de manera única.

1.6.6.4. Relaciones.

Las bases de datos relacionales, a diferencia de las jerárquicas y de red no almacenan en la base de datos punteros para identificar las ligas existentes entre dos tablas. Aunque estas relaciones siguen existiendo en la base de datos relacional.

Por ejemplo: en la base de datos ejemplo cada uno de los grupos tiene asignado un número de materia, por lo que hay una relación entre estas dos tablas. Se podría alegar que esta es una relación padre/hijo como las que no encontramos en las bases de datos relacionales, y en efecto, diré que es una relación padre/hijo, pero la diferencia existente es que esta relación no esta representada en forma física en la base de datos. En lugar de ello, la relación esta representada por *valores de datos comunes* almacenados en las dos tablas. Todas las relaciones de una base de datos relacional están representadas de ese modo.

En cualquier DBMS existen tres tipos de relaciones de entidades:

uno-a-muchos, uno-a-uno y muchos a muchos.

1.6.6.4.1.Relación uno a muchos.

Se dice que una relación es uno a muchos si las filas de una tabla está relacionada con varias filas de otra tabla. Pensemos en la tabla GRUPO y en la tabla INSCRIPC. La relación entre estas dos tablas es de uno-a-muchos porque cada grupo tiene muchas filas en la tabla de inscripciones, que representan alumnos dados de alta en ese grupo.

1.6.6.4.2.Relación uno a uno.

La relación uno-a-uno es un caso particular de la de uno-a-muchos. Con una relación uno-a-uno, la fila de una tabla se puede enlazar a solo una fila de otra tabla. Por ejemplo a cada clave de materia en la tabla MATERIA corresponde una sola carrera en la tabla CARRERA.

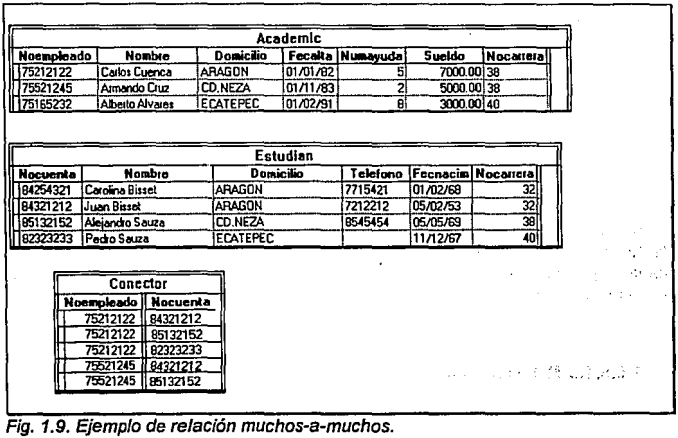
1.6.6.4.3.Relación mucho a muchos.

Una relación muchos-a-muchos sucede cuando se puede asociar una fila en una tabla con muchas filas en otra tabla y viceversa. Por ejemplo un maestro enseña a muchos alumnos y un alumno puede tener varios maestros.

Una relación muchos-a-muchos no se puede implantar directamente. Por ejemplo en la tabla INSCRIPC no se pueden determinar los maestros de los alumnos inscritos ni viceversa.

Aunque la relación muchos-a-muchos no se puede implantar directamente entre dos tablas, se puede reducir a relaciones uno-a-muchos utilizando una tabla que sirva de conexión.

Una fila en la tabla conectora contiene una llave principal por cada dos registros que une. La siguiente figura muestra el uso de la tabla conectora:



Academic

Noempleado	Nombre	Domicilio	Fecalta	Numayuda	Sueldo	Nocarrera
75212122	Carlos Cuenca	ARAGON	01/01/82	5	7000.00	38
75521245	Armando Cruz	CD.NEZA	01/11/83	2	5000.00	38
75165232	Alberto Alvarez	ECATEPEC	01/02/91	8	3000.00	40

Estudian

Nocuenta	Nombre	Domicilio	Telefono	Fecnacim	Nocarrera
84254321	Carolina Bisset	ARAGON	7715421	01/02/68	32
84321212	Juan Bisset	ARAGON	7212212	05/02/53	32
85132152	Alejandro Souza	CD.NEZA	8545454	05/05/69	38
82323233	Pedro Souza	ECATEPEC		11/12/67	40

Conector

Noempleado	Nocuenta
75212122	84321212
75212122	85132152
75212122	82323233
75521245	84321212
75521245	85132152

Fig. 1.9. Ejemplo de relación muchos-a-muchos.

Las columnas de que contiene la tabla conectora corresponden a las claves principales de las tablas INSCRIPC y ACADEMIC. En la tabla CONECTOR se ve que estudiantes son alumnos de que profesor y los profesores que dan clase a cada alumno.

1.6.6.5. Claves foráneas.

Un campo que sirve de conexión con otra tabla se llama clave foránea si es "foráneo" a la clave principal. En la figura 7 observamos en la tabla MATERIA que la columna CNOMATERIA es la llave principal, pero la columna CNOCARRERA hace referencia a la tabla de CARRERA, por lo que esta columna, en la tabla de MATERIA es una llave foránea ya que hace referencia a una llave primaria de otra tabla.

Así como una combinación de columnas pueden servir como llave principal, una llave foránea puede ser una combinación de columnas. De hecho, la clave foránea siempre será una combinación de columnas cuando haga referencia a la clave primaria de otra tabla que este compuesta por dos o más columnas. Una tabla puede contener más de una clave foránea.

1.6.6.6. Valores NULL.

Puesto que las bases de datos son generalmente un modelo de una situación real, ciertos datos pueden inevitablemente faltar, ser desconocidos o no ser aplicables. En la base de datos que utilizaremos de ejemplo sería normal en la tabla ESTUDIAN que al capturar los datos de un nuevo estudiante éste no tuviera teléfono, por lo que este dato quedaría sin capturar.

SQL soporta explícitamente los datos que faltan, son desconocidos o son inaplicables, a través del concepto *valor nulo*. Un valor nulo es un indicador que dice a SQL que el dato falta o no es aplicable. Por ejemplo si en la tabla ACADEMIC damos de alta un nuevo profesor del cual todavía no ha sido especificado un sueldo, el campo sueldo contendría un valor NULL, lo cual no significa que el profesor tenga un sueldo de cero pesos, lo que significa es que el sueldo aún no ha sido determinado.

En ocasiones los valores NULL requieren un manejo especial por parte del DBMS. Por ejemplo si el usuario requiere saber la suma total de sueldos en la tabla ACADEMIC. En las sentencias SQL que analizaremos en el siguiente capítulo veremos como decirle al DBMS la acción que debe seguir al encontrar un valor NULL, esta acción puede ser, por ejemplo: no tomar en cuenta la columna, darle un valor de cero, o inclusive darle un valor arbitrario.

1.7. Diseño de bases de datos relacionales.

El proceso de diseño de la base de datos comienza con los requerimientos conceptuales de varios usuarios. También se pueden especificar requerimientos de algunas aplicaciones que comenzarán en un futuro. Estos requerimientos de los usuarios están integrados en un solo enfoque llamado *modelo conceptual.* Este modelo representa las entidades y sus relaciones; y nos da la capacidad de visualizarlas sin tener que ocuparnos de su almacenamiento físico.

El modelo conceptual se traduce posteriormente a un modelo de datos compatible con el DBMS elegido. Es posible que las relaciones entre las entidades, tal como quedaron en el modelo conceptual no sean totalmente compatibles con el DBMS seleccionado. En tales casos se deberán de hacer modificaciones al modelo conceptual para eliminar estas limitaciones.

La nueva versión del modelo conceptual que puede adaptarse al DBMS es conocido como *modelo lógico.* Aunque cabe hacer notar que para algunos autores el modelo conceptual y el lógico es el mismo ya que las diferencias son mínimas o no existen.

El modelo lógico al transferirse a un almacenamiento físico, como un disco o una cinta se le conoce como *modelo físico.* El modelo físico toma en cuenta la distribución de los datos, los métodos de acceso y las técnicas de clasificación. El modelo físico también es conocido como *modelo interno.*

Es importante hacer notar que aunque para desarrollar el modelo conceptual de la base de datos, se usan términos relacionales, el modelo que resulta al utilizar esta metodología es independiente del enfoque que se le dé al implantar la base físicamente; es decir, el modelo conceptual resultante puede ser implantado en un modelo relacional, jerárquico o de red.

El concepto principal, tomado del modelo relacional utilizado en el desarrollo del modelo conceptual, es el proceso de normalización, es decir el proceso de agrupar los campos de datos en tablas que representan sus entidades y sus relaciones. La teoría de normalización está basada en la observación de que un cierto conjunto tiene mejores propiedades en un medio de inserción, actualización y supresión, que las que tendrían otros conjuntos de relaciones conteniendo los mismos datos.

Para asegurar el correcto funcionamiento de la base de datos lógica es que utilizamos el proceso de normalización. Aunque una base de datos no normalizada puede funcionar, puede causar algunos problemas cuando los programas de aplicación tratan de actualizar la base de datos.

El proceso de normalización identifica los datos redundantes que puedan existir en la estructura lógica, determina claves únicas para el acceso a los datos y ayuda a establecer las relaciones necesarias entre los elementos de datos.

Para el diseñador experto de bases de datos, derivar entidades o registros de tipo conceptual de un grupo de datos se puede hacer intuitivamente. Sin embargo esta intuición no esta presente en los principiantes, y es difícil encontrarla que surja espontáneamente cuando el diseño es muy complejo. La teoría de normalización proporciona un procedimiento riguroso para el diseño de bases de datos. Una base de datos mal diseñada puede funcionar inicialmente, pero puede mostrar anomalías en el almacenamiento cuando se presenten operaciones de inserción, borrado y actualización de datos. La teoría de la normalización ayuda a reconocer las cualidades no deseadas en una tabla y la forma de corregirlas.

Con el procedimiento de normalización, un archivo conceptual se representa como una tabla de dos dimensiones llamada relación o tabla, que es la forma mas simple para representar los datos.

Es importante hacer notar que el proceso de normalización se utiliza para llegar al modelo de datos conceptual; sin embargo la base de datos conceptual que se deriva de este procedimiento, puede modificarse como para su implantación en un DBMS jerárquico, relacional o de red.

Una relación no normalizada es un relación que contiene varias ocurrencias de algunos valores en cualquiera de sus campos. El primer paso de la normalización consiste en transformar los campos de datos a una tabla de dos dimensiones en donde sólo exista un valor para cada intersección de una columna con una fila, es decir no existan arreglos en un campo.

Las relaciones normalizadas se agrupan en cuatro categorías llamadas *formas normales (FN)*. Siendo cada nivel una descomposición más completa de la del nivel anterior. La meta final del proceso de normalización es la agrupación de todos los atributos (o campos, columnas) de una base de datos en relaciones adecuadas para que la base de datos se pueda almacenar con el mínimo de datos redundantes, además de quitar las cualidades indeseables de una relación que puedan ocasionar anomalías en el almacenamiento cuando se efectúen operaciones de actualización en la base de datos.

El proceso de normalización empieza con la combinación de todos los datos de la base en una relación, la que a su vez se descompone en dos o más relaciones más pequeñas. Se efectúan descomposiciones sucesivas de las relaciones intermedias hasta que todas las relaciones obtenidas pertenecen a la cuarta forma normal (4FN).

A continuación se describe completamente el proceso de normalización.

1.7.1. Primera Forma Normal.

Se dice que una relación esta en la primera forma normal (1FN) si todos los campos en cada registro contienen un solo valor tomado de sus dominios respectivos. Como sabemos el rango de valores permitidos para un campo se denomina *dominio*.

La siguiente figura no es una primera forma normal, porque en los campos CANTIDAD y FECHA existe más de un valor. Por ejemplo para el CLIENTE 100 el campo FECHA contiene los valores 01/01/94 y 15/02/94, lo mismo que el campo CANTIDAD contiene los valores 1 y 2 para el mismo cliente. Por lo tanto se dice que la relación ORDENES no está normalizada.

NO_CTE	NOMBRE	CIUDAD	CARGO_ ENVIO	PRECIO_ UNIDAD	NO_INV	CANT	FECHA
100	JUAN	MÉXICO	0.75	1.00	10	1	01/01/94
						2	15/02/94
200	MARÍA	TOLUCA	1.00	2.00	11	1	15/01/94
				1.00	10	1	16/01/94
				4.00	15	3	17/01/94
300	JOSÉ	MÉXICO	0.75	3.00	13	1	02/01/94
				2.00	11	2	16/01/94
400	ROSA	MÉRIDA	2.00	4.00	15	1	06/01/94

Fig. 1.10 Relación sin normalizar.

1.7.1.1. Derivación y anomalías en las relaciones 1FN.

Una relación sin normalizar se puede normalizar con la creación de un registro nuevo para cada uno de los distintos valores en un campo. El ejemplo de la figura anterior es una manera de guardar la información de los pedidos de una compañía, en forma tabular y sin duplicidad de datos. Pero es mejor crear registros para reemplazar los espacios que no contienen valores en los campos NO_CTE, NOMBRE, CIUDAD CARGO_ENVIO, PRECIO_UNIDAD, NO_INV con los valores apropiados. Como lo muestra la siguiente figura:

NO_CTE	NOMBRE	CIUDAD	CARGO_ ENVIO	PRECIO_ UNIDAD	NO_INV	CANT	FECHA
100	JUAN	MÉXICO	0.75	1.00	10	1	01/01/94
100	JUAN	MÉXICO	0.75	1.00	10	2	15/02/94
200	MARÍA	TOLUCA	1.00	2.00	11	1	15/01/94
200	MARÍA	TOLUCA	1.00	1.00	10	1	16/01/94
200	MARÍA	TOLUCA	1.00	4.00	15	3	17/01/94
300	JOSÉ	MÉXICO	0.75	3.00	13	1	02/01/94
300	JOSÉ	MÉXICO	0.75	2.00	11	2	16/01/94
400	ROSA	MÉRIDA	2.00	4.00	15	1	06/01/94

Fig. 1.11 Primera forma normalizada de la tabla ORDENES.

La relación presentada en esta figura está en la primera forma normal porque cada campo en un registro contiene sólo un valor.

Veamos algunas anomalías de almacenamiento que presenta esta forma normalizada:

- *Anomalías de inserción en la relación 1FN.* No se puede introducir un nuevo artículo del inventario en la relación ORDENES, a menos que el artículo se haya vendido cuando menos una vez. De otra manera atributos como NO_CTE y FECHA quedarían sin información. De manera similar no se puede introducir un cliente nuevo si no ha comprado algo.

  Las anomalías de inserción que se describieron se deben a la dependencia funcional de algunos campos no-clave en un subconjunto de la clave principal en lugar de ser dependientes de toda la clave. El concepto de *dependencia funcional* fue tomado de las matemáticas. Se dice que Y es función de X, Y =f(X), si el valor de Y está siempre determinado por el valor de X. Aplicando la misma terminología decimos que el campo NOMBRE y CIUDAD sólo son funcionalmente dependientes de NO_CTE. Por lo que si damos de alta un cliente los campos NO_INV y FECHA quedarían sin definir. Para evitar anomalías de inserción en este ejemplo, la información del cliente y la del inventario se debe guardar en relaciones separadas.

- *Anomalías en la eliminación en la relación 1FN.* Siguiendo con el mismo ejemplo: Si un día se decidiera dejar de vender el artículo 15, tendríamos que eliminar los registros que contengan el artículo 15. Pero el borrado del registro que contenga el artículo 15 no sólo quitaría la información referente al artículo, sino que también la información de un cliente (Rosa). De igual manera el dar de baja un cliente eliminaría la información del artículo si éste fue comprado únicamente por el cliente que estamos dando de baja. Estas anomalías pueden ser eliminadas si separamos la información referente al cliente y la referente al inventario en relaciones separadas.

- *Anomalías en la actualización en la relación 1FN.* Al pasar de una forma no normalizada a la 1FN, algunos datos se duplicaron. Esta duplicidad generará problemas al tratar de actualizar los datos. Por ejemplo si la cliente María se muda de la ciudad de Toluca a la ciudad de México tendríamos que modificar cada registro que contenga a María; de otra manera tendríamos datos inconsistentes. Aquí también se hace evidente que necesitamos un método para separar nuestra relación ORDENES en relaciones más pequeñas.

1.7.1.2. Normalización de la relación 1FN.

Como anteriormente se describió, las anomalías de almacenamiento se atribuyen a la presencia de campos no-clave que no son total y funcionalmente dependientes de la clave principal. Las anomalías presentadas en la relación 1FN se pueden eliminar con el siguiente procedimiento:

1) Quitar de la relación 1FN todos los campos no-clave que no sean totalmente dependientes de la clave primaria.

2) Guardar los campos no-clave que fueron quitados en relaciones nuevas y adecuadas.

Veamos el procedimiento paso a paso que seguiremos para dividir nuestra relación ORDENES en varias relaciones mas pequeñas a este procedimiento se le conoce como proceso de normalización:

PASO 1. Escoger una clave primaria que pueda representar de manera única cada registro en la relación:

Notamos que para la relación ORDENES los campos NO_CTE, NO_INV y FECHA identifican de manera única cada registro. La combinación de los tres campos es nuestra llave primaria.

PASO 2. Construir un diagrama de dependencia funcional describiendo las relaciones entre los atributos.

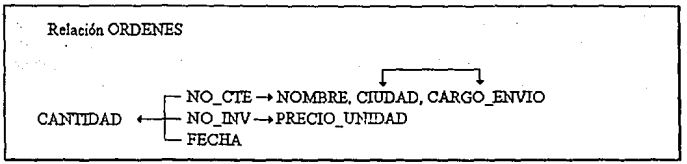


Fig.1.12. Diagrama de dependencia funcional.

En la figura mostramos el diagrama de dependencia funcional el sentido de la flecha determina cual campo es funcionalmente dependiente del otro. Por ejemplo para simbolizar que PRECIO_UNIDAD es dependiente de NO_INV mostramos:

NO_INV ⟶ PRECIO_UNIDAD

La figura del diagrama indica que solo CANTIDAD es funcionalmente dependiente de una combinación de NO_CTE, NO_INV y FECHA. Los atributos NOMBRE, CIUDAD y CARGO_ENVIO son funcionalmente dependientes de NO_CTE, mientras que PRECIO_UNIDAD es funcionalmente dependiente de NO_INV. Por lo que tenemos detectados los atributos que no son total y funcionalmente dependientes de la clave primaria.

PASO 3. Dividir la relación 1FN de tal manera que todos los campos no-clave en cada relación dividida sean total y funcionalmente dependientes de la clave primaria. Del diagrama de dependencia funcional observamos que la relación ORDENES se puede descomponer en las relaciones siguientes:

CLIENTE (NO_CTE, NOMBRE, CIUDAD, CARGO_ENVIO)

INVENTARIO(NO_INV, PRECIO_UNIDAD)

PEDIDOS(NO_CTE, NO_INV, FECHA, CANTIDAD)

La información que contiene cada tabla se muestran a continuación:

Relación CLIENTE

NO_CTE	NOMBRE	CIUDAD	CARGO_ ENVIO
100	JUAN	MEXICO	0.75
200	MARIA	TOLUCA	1.00
300	JOSE	MEXICO	0.75
400	ROSA	MERIDA	2.00

Relación INVENTARIO.

NO_INV	PRECIO_ UNIDAD
10	1.00
11	2.00
13	3.00
15	4.00

Relación PEDIDOS

NO_CTE	INV_NO	FECHA	CANTIDAD
100	10	01/01/94	1
100	10	15/02/94	2
200	11	15/01/94	1
200	10	16/01/94	1
200	15	17/01/94	3
300	13	02/01/94	1
300	11	16/01/94	2
400	15	06/01/94	1

Fig. 1.13. Contenido de las relaciones derivadas de la relación 1FN.

PRUEBAS DE ELIMINACIÓN DE ANOMALÍAS, EN 1FN. Observamos que podemos dar de alta un cliente en la relación **CLIENTE** sin que compre algo; por otro lado podemos dar de baja un artículo de **INVENTARIO** sin afectar la información del cliente que lo compró. Por último si un cliente cambia de residencia sólo tendremos que actualizar un solo registro.

Una vez que llegamos a la primera normalización, el proceso de normalización debe seguir; examinando cada una de las relaciones derivadas para buscar otras anomalías. Nuevamente se construye un diagrama de dependencia funcional para cada relación para determinar a que forma normal pertenece la relación y decidir si es necesaria una mayor descomposición.

1.7.2. Segunda Forma Normal (2FN).

Se dice que una relación pertenece a la segunda forma normal si es 1FN y cada atributo no-clave de la relación es total y funcionalmente dependiente de su clave principal.

Las relaciones derivadas mostradas en la figura anterior son de la segunda forma normal. Pero aún cuando las anomalías de almacenamiento de 1FN quedan eliminadas cuando se alcanza la 2FN, pueden surgir otros tipos de errores.

1.7.2.1. Anomalías de relaciones 2FN.

Una relación 2FN puede presentar anomalías de almacenamiento si cualquiera de sus campos no-clave *depende transitivamente* de la clave primaria. Decimos que un campo depende transitivamente de la clave primaria si es funcionalmente dependiente de un campo no-clave, el cual depende directamente de la clave principal; dicho de otra manera: depende indirectamente de la clave primaria. En la relación **CLIENTE** que obtuvimos en la **1FN** el campo **CARGO_ENVIO** es funcionalmente dependiente de **CIUDAD**. Veamos de que manera nos afecta esto a la hora de almacenar los datos.

- *Anomalías de inserción.* El precio de entrega CARGO_ENVIO no se puede dar de alta a menos de que un cliente exista en la ciudad en que queremos dar de alta el nuevo cargo. Si tratáramos de dar de alta un nuevo cargo sin que exista el cliente entonces NO_CTE y NOMBRE quedarían con valores nulos, siendo la llave principal NO_CTE. Este error es debido a la dependencia funcional del campo no-clave CARGO_ENVIO de otro atributo no-clave CIUDAD en la misma relación. Esta anomalía se elimina si el campo CARGO_ENVIO se registra en otra relación.

- *Anomalías de eliminación.* Suponga que el cliente Rosa cierra su cuenta y nosotros deseamos eliminar su registro de la relación CLIENTE, por ser el único cliente de Mérida también tendremos que perder la información de CARGO_ENVIO para la ciudad de Mérida. Esta anomalía nuevamente se le atribuye a la dependencia funcional de CARGO_ENVIO con respecto a CIUDAD.

- *Anomalías de actualización.* En la misma relación CLIENTE Juan y José son de México, quiere decir que para actualizar el campo CARGO_ENVIO tendremos que actualizarlo en los dos registros.

1.7.2.2. Normalización de una relación 2FN.

Los errores de almacenamiento en una relación 2FN son causadas por la dependencia transitiva. Recordemos que un campo depende transitivamente de la clave primaria si es funcionalmente dependiente de un campo no-clave, y éste último depende directamente de la clave principal. Por lo tanto si almacenamos los campos no-clave que son transitivamente dependientes de la clave primaria en una nueva relación, solucionaremos los errores de almacenamiento en la 2FN. Para normalizarla seguiremos los siguientes pasos:

PASO 1. Examinar cada atributo no-clave de la relación CLIENTE para determinar si es funcionalmente dependiente de otra no-clave. Al realizar un diagrama de dependencia funcional (ver figura) observamos que el campo CARGO_ENVIO es funcionalmente dependiente de CIUDAD, pero solo es transitivamente dependiente de NO_CTE.

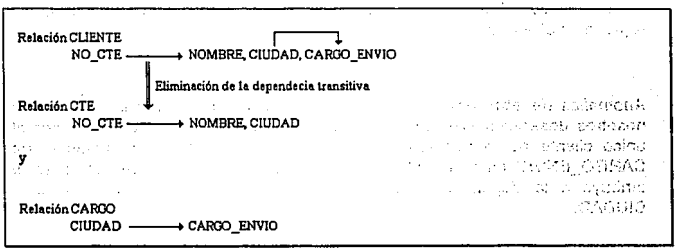


Fig. 1.14. Normalización de la relación 2FN CLIENTE.

PASO 2. Crear una nueva relación para almacenar la no-clave transitivamente dependiente CARGO_ENVIO y su llave CIUDAD. Por lo que la relación 2FN se divide en dos relaciones, CARGO y CTE. El campo CIUDAD aparece en las dos relaciones y sirve de campo de conexión. La siguiente figura muestra los contenidos de las nuevas relaciones CTE y CARGO.

Relación CTE

NO_CTE	NOMBRE	CIUDAD
100	JUAN	MEXICO
200	MARIA	TOLUCA
300	JOSE	MEXICO
400	ROSA	MERIDA

Relación CARGO

CIUDAD	CARGO_ENVIO
MEXICO	0.75
TOLUCA	1.00
MERIDA	2.00

Fig. 1.15. Contenido de las tablas CTE y CARGO

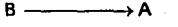
PRUEBAS DE ELIMINACIÓN DE ANOMALÍAS, EN 2FN. En la figura observamos que podemos introducir el CARGO_ENVIO para una ciudad aunque no exista un cliente para esa ciudad. Si deseamos dar de baja un cliente podemos hacerlo sin que se pierda la información del campo CARGO_ENVIO, aún cuando el cliente sea el único que vive en ésa ciudad. El valor de CARGO_ENVIO se almacena una sola vez, evitando así inconsistencia de datos al actualizar parcialmente datos redundantes.

Siguiendo con el proceso de normalización revisamos las nuevas relaciones y observamos si contienen alguna relación indeseable entre sus atributos, en caso de ser necesario se continúan separaciones hasta que no existan relaciones indeseables en ninguna de las relaciones derivadas. Las relaciones obtenidas en los pasos anteriores INVENTARIO y PEDIDOS no encontramos ninguna dependencia transitiva y por ello no les hizo ningún cambio. Como resultado obtuvimos cuatro relaciones: INVENTARIO, PEDIDOS, CTE y CARGO.

1.7.3. Tercera Forma Normal (3FN).

Decimos que una relación es 3FN si es 2FN y ningún atributo no-clave en la relación es funcionalmente dependiente de algún otro atributo no-clave. Dado que en la normalización anterior eliminamos la dependencia transitiva de CARGO_ENVIO con CIUDAD, Las cuatro relaciones derivadas pertenecen a la 3FN. En la mayoría de los casos el proceso de normalización queda completo cuando todas las relaciones derivadas son de 3FN.

Codd definió la 3FN originalmente en 1972. Posteriormente la corrigió y a la nueva definición se le conoce como la forma normal Boyce/Codd (BCFN); una forma normal es BCFN si cada determinante en la relación es una clave aspirante. Recordemos que si el atributo A es funcionalmente dependiente del atributo B si el valor de A está determinado por el valor de B, es decir :



Dicho de otra manera B determina a A, por lo que B es un *determinante* de A. Por otro lado una clave aspirante es un atributo o un grupo de atributos cuyo contenido puede representar de manera única a cada registro de una relación. Si en una relación existe más de una clave aspirante, una de las claves aspirantes se designa como la clave primaria.

En la mayoría de los casos, cuando una relación es 3FN también es BCFN. En el ejemplo anterior las cuatro relaciones en la 3FN INVENTARIO, PEDIDOS, CTE y CARGO son 3FN y BCFN. Por ejemplo, la relación CTE es 3FN porque ningún no-clave depende transitivamente de otra clave. También es BCFN porque cada determinante es clave aspirante; de hecho si nos fijamos hay un solo determinante: NO_CTE, que es a su vez clave principal. Sin embargo una relación 3FN no necesariamente es BCFN. Esta situación no es común y sucede cuando dos claves aspirantes sobrepuestas están contenidas en una relación.

Tomemos como ejemplo la relación PROY-MAQ-PROG, la cual se muestra en la siguiente figura:

NO_PROY	MAQ	PROGRAMADOR
P01	PC	JUANA
P01	APPLE	PEDRO
P15	HP 3000	CARLOS
P20	PC	JUANA
P20	PC	GLORIA

Fig. 1.16. Relación PROY-MAQ-PROG.

La relación contiene dos claves aspirantes : con el número de proyecto NO_PROY y el tipo de computadora MAQ determinamos el campo PROGRAMADOR, y NO_PROY - PROGRAMADOR es determinante de la máquina MAQ utilizada en el proyecto, de acuerdo a las siguientes reglas:

1) Un programador se especializa en un sólo tipo de computadora.

2) Un proyecto puede usar distintos tipos de máquinas, como por ejemplo el proyecto P01 asignado a APPLE y a una PC.

Si designamos como clave primaria NO_PROY-MAQ, el diagrama de dependencia funcional que se muestra a continuación define las reglas antes mencionadas:

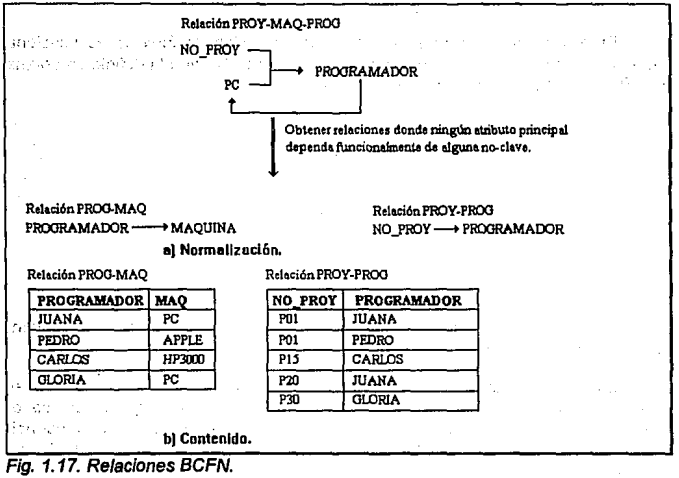


Fig. 1.17. Relaciones BCFN.

Observando la relación PROY-MAQ-PROG concluimos que es 3FN porque ninguna no-clave es dependiente de otra no-clave . Sin embargo la relación no es BCFN porque PROGRAMADOR no es una clave aspirante y es un determinante del tipo de máquina MAQ, que forma parte de la clave principal.

Uno de los problemas que se generan es que si por ejemplo, se tratara de dar de alta un nuevo programador especialista en APPLE el campo NO_PROY quedaría sin definir, y dado que es atributo principal no puede quedarse con un valor nulo.

Otra anomalía que se presenta : al tratar de eliminar un proyecto, se pude eliminar la información de la especialidad del programador, si éste tiene asignado únicamente el proceso que se desea dar de baja. Por ejemplo al tratar de eliminar el P30 se eliminará la información de Gloria y que es especialista en PC's.

Las anomalías 3FN suceden porque uno de los atributos principales depende de un campo no-clave. El problema se elimina al derivar dos relaciones BCFN. Como se mostró en la figura anterior.

En el ejemplo anterior se vio como la definición de 3FN no es suficiente cuando un atributo principal depende de un no-clave. Aunque la definición original de 3FN es adecuado en la mayoría de los casos.

1.7.4. Cuarta Forma Normal (4FN).

Una relación es cuarta forma normal (4FN) si es BCFN y no contiene dependencias multivalores.

Dada una relación, el atributo A de ésta relación se dice que es dependiente de multivalores (DMV) del atributo B si un rango específico de valores de A está determinado por un valor particular de B. La dependencia multivalores es un caso especial de la dependencia funcional. La DMV de A en B se expresa:

$$B \longrightarrow\!\!\!\!\!\rightarrow A$$

La flecha doble indica que B define valores múltiples en A. Considérese la relación CALZADO de la siguiente figura. Para cada modelo de zapato existen diferentes tamaños y diferentes colores. Los atributos COLOR y MEDIDA son dependientes multivalores de MODELO. En otras palabras cualquier valor de MODELO determina un conjunto de valores para los campos MEDIDA y COLOR. Por otro lado el atributo PRECIO es constante sin importar el tamaño ni el color, sólo es dependiente del MODELO.

La clave principal de ésta relación consta de MODELO, COLOR y MEDIDA.

El problema con la dependencia de multivalores es evidente: la redundancia de valores. Por ejemplo el modelo I1 se repite ocho veces ya que éste modelo existe en cuatro tallas y dos colores, y mientras existan más tallas y colores más número de combinaciones habrá y la redundancia será mayor.

MODELO	PRECIO	COLOR	MEDIDA
I1	2.00	NEGRO	1
I1	2.00	NEGRO	2
I1	2.00	NEGRO	3
I1	2.00	NEGRO	4
I1	2.00	CAFÉ	1
I1	2.00	CAFÉ	2
I1	2.00	CAFÉ	3
I1	2.00	CAFÉ	4
I2	4.00	BLANCO	1
I2	4.00	BLANCO	2
I2	4.00	VINO	1
I2	4.00	VINO	2
I2	4.00	CAFÉ	1
I2	4.00	CAFÉ	2

Fig. 1.18. Relación ZAPATOS.

La redundancia de los datos causada por la dependencia multivalores se puede eliminar con cualquiera de los dos métodos que a continuación se exponen:

1) Crear una nueva relación de cada atributo DMV

Cuando el número de valores repetidos en un DMV es muy grande, se puede crear una nueva relación para el atributo DMV y su clave principal. Siguiendo con nuestro ejemplo, y suponiendo que el número de medidas es muy grande o indefinido, entonces creamos una nueva relación como se muestra en la siguiente figura.

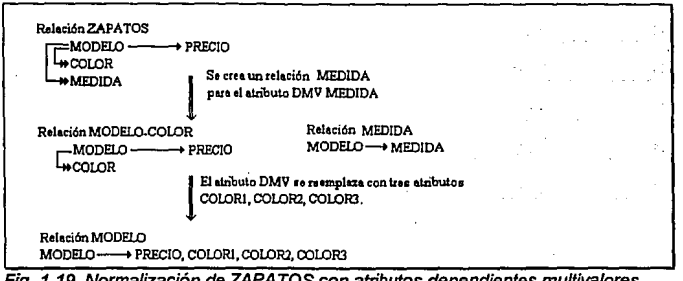


Fig. 1.19. Normalización de ZAPATOS con atributos dependientes multivalores.

2) Reemplazando un atributo DMV con varios atributos.

Si la cantidad de valores distintos en un atributo DMV es un número específico y pequeño, entonces cada uno de los valores del atributo DMV se puede representar por un atributo dentro de una misma relación. Por ejemplo si los colores de ningún modelo exceden de tres, el atributo COLOR se puede representar por un COLOR1, COLOR2 y COLOR3 como se muestra en la anterior figura,

El contenido de las relaciones resultantes se muestra a continuación.

Relación MODELO

MODELO	PRECIO	COLOR1	COLOR2	COLOR3
I1	2.00	NEGRO	CAFE	— —
I2	4.00	BLANCO	VINO	CAFE

Relación MEDIDA

MODELO	MEDIDA
I1	1
I1	2
I1	3
I1	4
I2	1
I2	2

Fig. 1.20. Contenido de las relaciones MODELO Y MEDIDA.

Aunque también podríamos haber optado por tener otra relación COLOR como se ejemplifica a continuación.

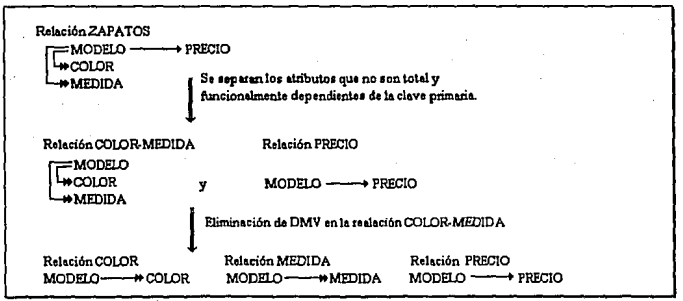


Fig. 1.21. Derivación de relaciones 4FN eliminando dependencias multivalores.

El contenido de las nuevas relaciones se muestra en la siguiente figura.

Relación PRECIO

MODELO	PRECIO
I1	2.00
I2	4.00

Relación COLOR

MODELO	COLOR
I1	NEGRO
I1	CAFE
I2	BLANCO
I2	VINO
I2	CAFE

Relación MEDIDA

MODELO	MEDIDA
I1	1
I1	2
I1	3
I1	4
I2	1
I2	2

Fig. 1.22. Contenido de las relaciones PRECIO, COLOR y MEDIDA.

La diferencia básica entre los dos planteamientos está en la manera que se almacenan los distintos valores para un atributo DMV. Con el primer método se crea una nueva relación para un atributo DMV, y los valores que ocurren en este atributo se agregan en forma vertical. Por otro lado, en el segundo método , los distintos valores se agregan horizontalmente en un número fijo de campos. La limitante de éste método es que no se reserva espacio para un número infinito de valores. Por ejemplo al tener tres atributos para el color, es decir COLOR1, COLOR2 y COLOR3, tenemos que estar seguros que los colores de un modelo de calzado nunca serán mayor a tres. Cabe señalar que al separar un atributo en una nueva relación tenemos redundancia de valores, por ejemplo el la relación MEDIDA el campo MODELO se repite al separarse de la relación ZAPATOS.

La forma de almacenamiento en este caso deja a criterio del diseñador de la base dependiendo de la aplicación de la base de datos que esta normalizando.

2. ACCESO Y ACTUALIZACIÓN DE LA BASE DE DATOS.

2.1. Elementos de una sentencia.

Las sentencias son usadas por SQL para pedir al DBMS que realice las operaciones deseadas sobre las tablas de la base de datos, tales como la creación, la recuperación de los datos o la inserción de nuevos datos.

Las sentencias de SQL contienen los siguientes componentes:

1) Verbo. Todas las sentencias SQL comienzan con un verbo, una palabra reservada que describe lo que la sentencia hace.

   Ejemplo: SELECT, INSERT, DELETE, UPDATE.

2) Cláusulas. Una cláusula nos puede decir sobre que datos actúa el verbo, o proporciona detalles de lo que la sentencia hace. Las cláusulas comienzan también con palabras reservadas como WHERE, INTO, FROM y GROUP BY.

3) Nombres de tablas y columnas. Son nombres que fueron dados por el creador de las tablas. Son asignados cuando se define una nueva tabla, para lo cual se utilizan a su vez sentencias SQL: la sentencia CREATE.

   Con el permiso adecuado también se pueden accesar tablas de otros usuarios, utilizando la notación completa para referirse a una tabla separando el nombre del propietario del nombre de la tabla mediante un punto. Por ejemplo si existiera una base nombrada DERECHO y quisiéramos accesar la tabla MATERIA de esa base, escribiríamos:

   DERECHO.MATERIA

Cuando una sentencia SQL requiere un nombre de columna con solo mencionar su nombre SQL sabe que pertenece a la tabla que estamos utilizando. Sin embargo cuando usamos varias tablas que contienen columnas con el mismo nombre, para diferenciarlos utilizamos la sintaxis completa para referirnos a una columna en especial, separando el nombre de la tabla a la que pertenece del nombre de la columna mediante un punto. Por ejemplo para diferenciar al campo NOMBRE de la tabla MATERIA del campo NOMBRE de la tabla ESTUDIAN, escribimos:

MATERIA.NOMBRE

ESTUDIAN.NOMBRE

También podemos hacer referencia a una columna de una tabla que pertenece a otra base de datos utilizando la notación completa para tablas y columnas. Veamos el ejemplo anterior, pero refiriéndonos a la base de DERECHO:

DERECHO.MATERIA.NOMBRE

DERECHO.ESTUDIAN.NOMBRE

Los nombres de tabla completos y los nombres de columna completos pueden utilizarse en cualquier parte donde la sentencia lo requiera.

4) Constantes. También conocidas como literales. Son secuencias de caracteres (letras, números, signos) que representan un valor. Cuando éste no es un número debe de ir entre comillas. Por ejemplo:

1253, "Electrónica II".

La siguiente figura muestra los elementos de una sentencia en SQL.

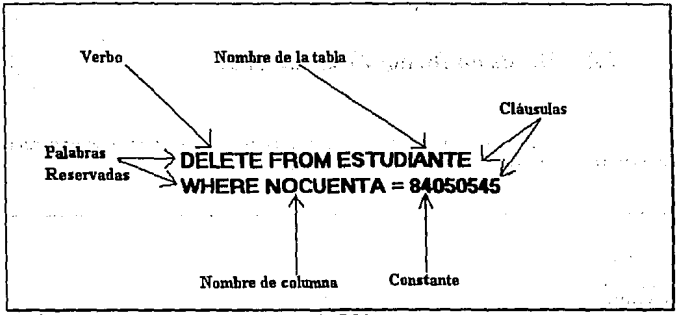


Fig. 2.1. Elementos de una sentencia SQL.

2.2. Tipos de datos.

EL SQL acepta varios tipos de datos. Cada tipo de dato cae en una de las siguientes tres categorías:

1) Cadenas de caracteres.

2) Numéricos.

3) Fechas y horas.

A continuación damos una descripción de cada categoría y tipos de datos asociados.

La mayoría de los productos SQL comerciales ofrecen su propio conjunto de tipos de datos en la siguiente sección se mencionan los tipos de datos de los productos SQL más populares.

2.2.1. Datos en forma de caracteres.

Existen varios tipos diferentes de datos expresados en forma de cadena de caracteres, Estos se muestran a continuación:

TIPO DE DATOS	DB2 y SQL/DS	ORACLE	INGRES	INFORMIX	SQL SERVER	dBASE IV	SQL BASE
Caracteres de longitud fija	CHARACTER(n)	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)	
Caracteres de longitud variable	VARCHAR(n)		VARCHAR(n)		VARCHAR(n)		VARCHAR(n)
Texto Largo	LONG VARCHAR	LONG			TEXT		LONG VARCHAR

Fig. 2.2. Tipos de datos caracter.

Los tipos de datos de caracteres de longitud fija CHARACTER(n) Y CHAR(n) tendrán siempre la misma longitud (n). Esto significa que el sistema anexa espacios en blanco no significativos.

Los valores almacenados como columnas VARCHAR(n), de longitud variable, contendrán únicamente los caracteres especificados en la operación de inserción. El sistema no añade ningún espacio en blanco. Los valores de la columna pueden tener diferentes longitudes que no excedan un máximo de (n)..

El tipo de datos de texto largo LONGVARCHAR, LONG y TEXT se usan para almacenar cadenas de caracteres largas.

2.2.2. Datos numéricos.

Contienen datos que se usaran en operaciones aritméticas. Los tipos de datos numéricos de los diferentes productos SQL más comerciales se presentan a continuación:

TIPO DE DATOS	DB2 y SQL/DS	ORACLE	INGRES	INFORMIX	SQL SERVER	dBASE IV	SQL BASE
Entero	SMALLINT INTEGER		INTEGER1 INTEGER2 INTEGER3	SMALLINT INTEGER	TYNYINT SMALLINT INT	SMALLINT INTEGER	SMALLINT INTEGER NUMBER
Decimal	DECIMAL(p,s)	NUMBER(p,s)		DECIMAL(p,s)		DECIMAL(p,s)	DECIMAL(p,s)
Monetario			MONEY	MONEY(p,s)	MONEY		
Coma Flotante	FLOAT(p) REAL DOUBLE PRESICION		FLOAT4 FLOAT8	SMALL FLOAT FLOAT	FLOAT	FLOAT(p,s)	FLOAT(p) REAL DOUBLE PRESICION

Fig. 2.3 Tipos de datos numéricos.

Los diferentes tipos de datos enteros existentes en cada producto definen cada uno de ellos distintos rangos de valores. Por ejemplo el tipo SMALLINT de DB2, INFORMIX, dBASE IV y SQL BASE permite un entero binario de 16 bits; es decir acepta valores entre -32768 y +32767; por otro lado el INTEGER, INT e INTEGER2 son enteros binarios de 32 bits y acepta valores enteros entre -2147483648 y +2147483647.

Los tipos de dato numérico DECIMAL(p,s) y NUMBER(p,s) son números decimales con precisión p y escala n. La precisión es el numero total de dígitos y la escala es el número de dígitos de la parte fraccional y no debe ser mayor a la precisión.

Los tipos de datos monetarios se almacenan generalmente como número decimal o en coma flotante. El tener este tipo de datos permite al DBMS formatear adecuadamente los importes monetarios cuando son visualizados.

Los números de coma flotante se utilizan para guardar números científicos que pueden ser muy grandes, pero pueden producir errores de redondeo en los cálculos.

2.2.3. Datos fecha/hora.

Algunos productos SQL soportan valores para fechas y horas y se presentan a continuación:

TIPO DE DATOS	DB2 y SQL/DS	ORACLE	INGRES	INFORMIX	SQL SERVER	dBASE IV	SQL BASE
DATE TIME TIME STAMP	DATE	DATE	DATE	DATE	DATETIME	DATE	DATE TIME DATETIME

Fig. 2.4 Tipos de datos de fecha.

Los tipos de datos de fecha y tiempo varían de acuerdo al producto SQL y al formato utilizado es decir una misma fecha puede desplegarse como mm/dd/aa, aa/mm/dd, mm-dd-aa... dependiente del formato que se elija al momento de desplegarse; aunque generalmente se almacena como yyyymmdd. El tipo de dato **TIMESTAMP** es en realidad un instante puede almacenarse como "yyyymmddhhmmssnnnnnn"

Algunos productos SQL ofrecen tipos de datos adicionales, por ejemplo:

Datos Booleanos. Algunos productos SQL por ejemplo dBase IV, Sybase y SQL Server, soportan los valores lógicos (TRUE o FALSE) como un tipo de dato.

Flujos de datos no estructurados. Algunos productos, por ejemplo Oracle permite almacenar y recuperar secuencias de bytes de longitud variable sin estructurar. Los valores que contienen estas columnas son imágenes de video comprimidas, código ejecutable y otros datos son estructurar.

2.3. Consultas simples.

En este capítulo se introducen los conceptos fundamentales y la estructura de la sentencia SELECT. Esta es la sentencia que se usa de forma más frecuente en SQL. Su propósito es la obtención de datos de la base de datos. Se verá como mostrar en pantalla sólo algunas columnas y seleccionar filas específicas de una tabla. La sentencia SELECT es la más potente y compleja de las sentencias SQL. A pesar de las muchas opciones permitidas por la sentencia SELECT en éste capítulo se presenta un formato simplificado que permite realizar consultas sencillas sobre una sola tabla y en secciones posteriores se elaborarán consultas más complejas.

Debe recordarse que los ejemplos realizados a lo largo de este trabajo de tesis se basan en la base de datos ejemplo que se describió en el capítulo anterior. Se puede hacer referencia al contenido de cada tabla en el apéndice B.

2.3.1. La sentencia SELECT.

La sentencia SELECT recupera datos de una base de datos y los regresa en forma de resultados de la consulta. Si la sentencia SELECT se escribe para realizar una consulta en forma interactiva, el DBMS visualiza los resultados de la consulta en forma tabular sobre la pantalla del computador. Si un programa envía una consulta al DBMS utilizando SQL, una tabla con los resultados de la consulta es devuelta al programa. En cualquiera de los dos casos los resultados siempre tienen el mismo formato de fila/columna al igual que las tablas de la base de datos.

A continuación se presenta una sintaxis simplificada de la sentencia SELECT:

```
SELECT    lista de selección
FROM      lista de tablas
```

La cláusula SELECT lista los campos a recuperar por la sentencia SELECT. La lista de selección puede incluir:

- Columnas de la base de la base de datos. Cuando un nombre de columna aparece en la lista de selección, SQL toma el valor de esa columna de cada fila de la tabla especificada en la cláusula WHERE y la coloca en la fila correspondiente de los resultados de la consulta.
- Expresiones SQL. Columnas que se calculan cuando se efectúa la consulta.
- Una constante, la cual en caso de especificarse aparecerá en todos los renglones del resultado.

La cláusula FROM lista las tablas que contienen los datos que se desean recuperar por la consulta. Por ahora sólo se describen las consultas que emplean una sola tabla.

Ejemplo: Mostrar todas las columnas de la tabla MATERIA:

```
SELECT *
FROM Materia
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.0032	
0134	Sistemas Digitales	8	50.0032	
0119	Estructuras de Datos	8	0.0032	
0056	Estructuras Discretas	7	0.0032	
0559	Memorias y Perifericos	10	100.0032	
0561	Microcomputadoras	10	500.0032	
0028	Analisis Dinamico de Máquinas	8	100.0038	
0130	Elementos de Máquinas	8	50.0038	
0024	Circuitos Electricos	10	150.0040	
0138	Dispositivos Electronicos	10	90.0040	

El asterisco que aparece a continuación de la cláusula SELECT indica que deseamos todas las columnas. La secuencia de salida de las columnas está determinada por la forma en que fueron definidas en el momento de su creación.

La cláusula FROM *Materia* le dice a SQL que se va a utilizar la tabla MATERIA. En caso de ser necesarias más tablas éstas van separadas por comas.

La sentencia anterior también puede ser escrita en una sola línea y el resultado será el mismo: Pero para mayor legibilidad se acostumbra escribir la cláusula FROM en una línea distinta.

Ejemplo: Mostrar el nombre y la clave de las materias que existen:

```
SELECT Mnomateria, Mnombre
FROM Materia
```

Resultado:

MNOMATERIA	MNOMBRE
0076	Bases de Datos
0134	Sistemas Digitales
0119	Estructuras de Datos
0056	Estructuras Discretas
0559	Memorias y Perifericos
0561	Microcomputadoras
0028	Analisis Dinamico de Máquinas
0130	Elementos de Máquinas
0024	Circuitos Electricos
0138	Dispositivos Electronicos

El orden en que se soliciten las columnas en la cláusula SELECT no necesitan ser el mismo en que se encuentre físicamente. Es decir podemos solicitar que el resultado se presente en un orden diferente al que se encuentra en la tabla.

Ejemplo: Ejecutar el mismo ejemplo anterior, solicitando el nombre de la materia seguido de la clave de la materia.

```
SELECT Mnombre, Mnomateria
FROM Materia
```

Resultado:

MNOMBRE	MNOMATERIA
Bases de Datos	0076
Sistemas Digitales	0134
Estructuras de Datos	0119
Estructuras Discretas	0056
Memorias y Perifericos	0559
Microcomputadoras	0561
Analisis Dinamico de Máquinas	0028
Elementos de Máquinas	0130
Circuitos Electricos	0024
Dispositivos Electronicos	0138

2.3.2. Columnas calculadas.

Además de las columnas cuyos valores provienen directamente de la base de datos, una columna de una consulta puede contener *columnas calculadas* cuyos valores se calculan a partir de los valores de los datos almacenados. Si se desea una columna calculada en la lista de selección se escribe una expresión, la cual puede contener sumas, restas, multiplicaciones y divisiones. Naturalmente que las columnas referenciadas en ésta expresión deberán ser numéricas de otro modo SQL marcará un error.

Ejemplo: Mostrar las materias y el costo de laboratorio y una ayuda voluntaria para la compra de material, esta ayuda se calcula en base al diez por ciento del costo del laboratorio.

```
SELECT Mnomateria, Mcostolab, Mcostolab * .10
FROM   Materia
```

MNOMATERIA	MCOSTOLAB	COSTOLABX1
0076	100.00	10.000
0134	50.00	5.000
0119	0.00	0.000
0056	0.00	0.000
0559	100.00	10.000
0561	500.00	50.000
0028	100.00	10.000
0130	50.00	5.000
0024	150.00	15.000
0138	90.00	9.000

En la tabla de resultado anterior la tercera columna se calcula utilizando los valores de MCOSTOLAB multiplicados por .10. En la tercera columna del resultado notamos que el titulo es MCOSTOLAB*1, éste varía dependiendo del producto SQL que utilicemos, algunos ponen como encabezado la expresión (o parte de la expresión dependiendo de los caracteres que acepte en el nombre de columna) y otros ponen como nombre de columna *expr_n* donde n indica el número consecutivo de la expresión dentro de la cláusula SELECT.

También se pueden utilizar constantes en la lista de selección. Esto puede utilizarse para imprimir resultados que se puedan interpretar más fácilmente.

Ejemplo: Listar los créditos de cada materia.

```
SELECT Mnombre, "tiene", Mnred, "créditos"
FROM   Materia
```

Resultado

MNOMBRE	TIENE	MCRED	CREDITOS
Bases de Datos	tiene		8creditos
Sistemas Digitales	tiene		8creditos
Estructuras de Datos	tiene		8creditos
Estructuras Discretas	tiene		7creditos
Memorias y Perifericos	tiene		10creditos
Microcomputadoras	tiene		10creditos
Analisis Dinamico de Máquinas	tiene		8creditos
Elementos de Máquinas	tiene		8creditos
Circuitos Electricos	tiene		10creditos
Dispositivos Electronicos	tiene		10creditos

La segunda y cuarta columna consiste en la misma cadena de texto para todas las filas.

2.3.3. Eliminación de filas duplicadas.

Si realizamos una consulta que incluya la clave primaria en la lista de selección cada fila del resultado será única, en caso contrario se podrían presentarse filas duplicadas.

Ejemplo: Listar la clave de las carreras de todas las materias de que se imparten.

```
SELECT Mnocarrera
FROM   Materia
```

Resultado:

MNOCARRERA
32
32
32
32
32
32
38
38
40
40

Se observa que cada materia tiene un número de carrera y por lo tanto el campo NOCARRERA se repite por cada materia que se imparta en la carrera.

Para eliminar la duplicidad de registros en el resultado de las consultas se inserta la palabra reservada DISTINCT en la sentencia SELECT justo antes de la *lista de selección:*

Sintaxis simplificada:

SELECT [DISTINCT] lista de selección
FROM lista de tablas

Ejemplo: Listar la clave de las carreras de todas las materias de que se imparten. Eliminar los valores duplicados.

SELECT DISTINCT Mnocarrera
FROM Materia

Resultado:

MNOCARRERA
32
38
40

Conceptualmente, SQL efectúa esta consulta generando primeramente un conjunto completo de resultados y eliminando posteriormente las filas que son duplicados exactos de alguna otra.

La palabra reservada DISTINCT sólo elimina filas duplicadas. En este ejemplo una fila consiste en una sola columna. Una fila se considera duplicada de otra si todos los valores de las columnas en la fila coinciden con los valores de las columnas de otra fila.

2.3.4. Cláusula WHERE.

En la práctica las bases de datos tienen tablas que contienen demasiadas filas para ser examinadas completamente por el usuario. Generalmente se desea analizar una parte de las filas de una tabla.

La cláusula WHERE determina exactamente cuales filas deben ser recuperadas para ser examinadas.

Sintaxis simplificada:

```
SELECT    lista de selección
FROM      lista de tablas
WHERE     condiciones de búsqueda
```

Ejemplo: Obtener toda la información sobre una materia donde el costo de laboratorio sea de 100.00

```
SELECT *
FROM Materia
WHERE Mcostolab = 100
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.0032	
0559	Memorias y Perifericos	10	100.0032	
0028	Analisis Dinamico de Máquinas	8	100.0038	

Cuando existe una cláusula WHERE en una sentencia SELECT, SQL recorre una a una las filas de las tablas especificadas y evalúa la condición. Por cada fila el resultado puede producir uno de tres resultados: FALSO, VERDADERO o DESCONOCIDO. El resultado DESCONOCIDO se produce cuando la condición de búsqueda actúa sobre un valor NULL. En el ejemplo todas las materias que no tienen asignado el costo de laboratorio produjeron DESCONOCIDO como resultado de la evaluación de búsqueda. Solo si el resultado de la evaluación de la condición resulta VERDADERO la fila es seleccionada y mostrada en el resultado.

2.3.5. Predicados

SQL ofrece una amplia variedad de condiciones de búsqueda. Las condiciones de búsqueda también son llamadas *predicados*. Un ***predicado*** expresa una condición entre valores y según sean éstos puede resultar VERDADERO, FALSO y DESCONOCIDO.

Los predicados se especifican en la cláusula WHERE, y en otras cláusulas que se verán posteriormente. Un predicado especificado en la cláusula WHERE da lugar a que se seleccionen todas aquellas filas en las que se tome el valor VERDADERO. En las secciones siguientes se describen los diferentes tipos de predicados soportados por SQL.

2.3.5.1. Predicados de comparación.

En un predicado de comparación SQL calcula y compara los valores de dos expresiones SQL por cada fila de datos. Las expresiones pueden ser tan simples como una columna o una constante o pueden ser operaciones aritméticas complejas. A continuación se presentan los modos de comparación:

Predicado	El predicado es VERDADERO si y solo si
x = y	La expresión x es igual a la expresión y
x <> y	La expresión x es diferente a la expresión y
x < y	La expresión x es menor a la expresión y
x > y	La expresión x es mayor a la expresión y
x >= y	La expresión x es mayor o igual a la expresión y
x <= y	La expresión x es menor o igual a la expresión y

Si alguno de los dos comparandos x o y, o ambos, contienen algún valor nulo, el predicado toma el valor DESCONOCIDO.

Ejemplo: Seleccionar la clave de la materia, la carrera a la cual pertenece y el costo del laboratorio de los cursos donde su laboratorio sea menor a $100.

```
SELECT Mnomateria, Mnocarrera, Mcostolab
FROM   Materia
WHERE Mcostolab < 100.00
```

MNOMATERIA	MNOCARRERA	MCOSTOLAB
0134	32	50.00
0119	32	0.00
0056	32	0.00
0130	38	50.00
0138	40	90.00

Ejemplo: Seleccionar la clave de la materia, la carrera a la cual pertenece y el costo del laboratorio de los cursos donde su laboratorio sea mayor o igual a $100.

```
SELECT Mnomateria, Mnocarrera, Mcostolab
FROM   Materia
WHERE Mcostolab >= 100.00
```

Resultado:

MNOMATERIA	MNOCARRERA	MCOSTOLAB
0076	32	100.00
0559	32	100.00
0561	32	500.00
0028	38	100.00
0024	40	150.00

Comparando los dos últimos ejemplos se nota que las condiciones de búsqueda son complementarias (menores a 100 y mayores - iguales a 100) la suma de los registros obtenidos en ambas consultas da como resultado el total de registros existentes en la tabla MATERIA. Si existiera un valor NULL en alguna fila ésta no sería tomada en cuenta en ninguna de las dos condiciones de búsqueda, por dar como resultado de la evaluación DESCONOCIDO.

2.3.5.2. Predicado de rango (BETWEEN).

La palabra reservada BETWEEN identifica un rango de valores. Su uso implica tres expresiones SQL . La primera define el valor a comprobar la segunda y la tercera son los extremos superior e inferior del rango.

El rango de selección incluye a los valores extremos.

Su sintaxis es:

expresión1 [NOT] BETWEEN expresión2 AND expresión3

Ejemplo: Obtener el nombre del curso y el costo del laboratorio de todas las materias donde el costo del laboratorio fluctúe entre 100 y 200 nuevos pesos (incluyendo ambos valores)

```
SELECT Mnombre, Mcostolab
FROM   Materia
WHERE Mcostolab BETWEEN 100.00 AND 200.00
```

Resultado

MNOMBRE	MCOSTOLAB
Bases de Datos	100.00
Memorias y Perifericos	100.00
Analisis Dinamico de Máquinas	100.00
Circuitos Electricos	150.00

El ejemplo anterior se podría haber escrito de la siguiente manera:

```
SELECT Mnombre, Mcostolab
FROM   Materia
WHERE Mcostolab >= 100.00 AND Mcostolab <=200.00
```

Esta solución produce los mismos resultados del ejemplo anterior, aunque se tuvo que repetir el nombre del campo MCOSTOLAB por lo que usar BETWEEN realmente sólo es una manera más elegante y sencilla de escribir la misma condición de búsqueda.

El predicado NOT BETWEEN se usa para seleccionar filas, cuando un valor para una columna se encuentra fuera del rango especificado. Los límites inferior y superior se excluyen en la selección de filas.

Ejemplo : Obtener el nombre de la materia y el costo del laboratorio. de todos los cursos que tengan un MCOSTOLAB menor de $100 o mayores a $200

```
SELECT Mnombre, Mcostolab
FROM   Materia
WHERE Mcostolab BETWEEN 100.00 AND 200.00
```

Resultado:

MNOMBRE	MCOSTOLAB
Sistemas Digitales	50.00
Estructuras de Datos	0.00
Estructuras Discretas	0.00
Microcomputadoras	500.00
Elementos de Máquinas	50.00
Dispositivos Electronicos	90.00

BETWEEN también puede utilizarse para rangos con cadenas de caracteres.

Ejemplo: Obtener todas las materias que sus nombres empiecen con B, C, o D

```
SELECT *;
FROM Materia;
WHERE Mnombre BETWEEN "B" AND "Dz"
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.0032	
0024	Circuitos Electricos	10	150.0040	
0138	Dispositivos Electronicos	10	90.0040	

2.3.5.3. Predicado de pertenencia a un conjunto (IN).

Este predicado permite pedir al sistema que seleccione una fila, si una determinada columna coincide con algún valor de los que se especifican en una lista.

Ejemplo: Seleccionar el número de curso, el nombre y los créditos para los cursos que tengan 8, ó 10 créditos.

SELECT Mnomateria, Mnombre, Mcred
FROM Materia
WHERE Mcred IN (10,8)
Resultado:

MNOMATERIA	MNOMBRE	MCRED
0076	Bases de Datos	8
0134	Sistemas Digitales	8
0119	Estructuras de Datos	8
0559	Memorias y Perifericos	10
0561	Microcomputadoras	10
0028	Analisis Dinamico de Máquinas	8
0130	Elementos de Máquinas	8
0024	Circuitos Electricos	10
0138	Dispositivos Electronicos	10

Utilizando NOT IN se seleccionan las filas donde un valor de una columna determinada no corresponde con ninguno de los valores especificados en la lista.

Ejemplo: Seleccionar el número de curso, el nombre y los créditos para los cursos que tengan créditos diferentes de 8 ó 10.

SELECT Mnomateria, Mnombre, Mcred
FROM Materia
WHERE Mcred NOT IN (8,10)

Resultado:

MNOMATERIA	MNOMBRE	MCRED
0056	Estructuras Discretas	7

Nuevamente tanto la frase IN como NOT IN es superflua y sólo representan una forma más sencilla de escribir la condición de búsqueda. Dado que el ejemplo anterior se podría haber escrito con cualquiera de las siguientes cláusulas WHERE :

WHERE Mcred <> 8 AND Mcred <> 10

WHERE NOT (Mcred = 8 OR Mcred = 10)

WHERE NOT Mcred = 8 AND NOT Mcred = 10

2.3.5.4. Predicado de correspondencia con un patrón (LIKE).

La palabra reservada LIKE es usada para seleccionar filas en la que el contenido de una columna de texto sea similar (no necesariamente idénticos) a un cierto texto particular. Solo es usado con columnas de tipo caracter, y algunos productos SQL soportan LIKE en datos *datetime*.

La palabra reservada LIKE se utiliza en la cláusula WHERE en lugar del operador de comparación. Su sintaxis es:

WHERE columna [NOT] LIKE patrón

El patrón es una cadena de caracteres que va entre comillas con el que va ser comparada la columna.

El test de correspondencia con un patrón puede ser utilizado para recuperar los datos en base a una correspondencia parcial con una cadena de caracteres. Esto se realiza usando dos caracteres especiales, a modo de comodines, que formaran parte del patrón. Estos caracteres son el tanto por ciento (%) y el subrayado (_). Su uso se describe a continuación:

2.3.5.4.1. Uso del caracter comodín signo de porcentaje (%).

El comodín signo de porcentaje puede coincidir con cualquier cadena de caracteres de cualquier longitud, inclusive puede ser una cadena vacía de longitud cero.

Ejemplo : Buscar todas las materias que sus títulos estén relacionados con máquinas.

```
SELECT Mnomateria, Mnombre, Mnocarrera;
FROM Materia;
WHERE Mnombre LIKE "%Máquina%"
```

MNOMATERIA	MNOMBRE	MNOCARRERA
0028	Analisis Dinamico de Máquinas	38
0130	Elementos de Máquinas	38

La cadena patrón "%Máquina%" contiene el símbolo del tanto por ciento al principio y al final, esto quiere decir que cualquier número (inclusive cero) de caracteres que se encuentre antes o después de la cadena "Máquina" hará coincidir la columna con el patrón de caracteres.

Si quitamos el primer comodín la columna debe de empezar con la cadena "Máquina" y las dos filas que fueron seleccionadas en el ejemplo anterior no hubieran cumplido con la condición de búsqueda.

2.3.5.4.2. Uso del caracter comodín signo de subrayado (_).

El caracter comodín subrayado (_) se corresponde con cualquier caracter simple. En el ejemplo anterior, si no se esta seguro de que la palabra "Máquina" se escribió con acento o sin acento se podría utilizar la siguiente consulta:

SELECT Mnomateria, Mnombre, Mnocarrera;
FROM Materia;
WHERE Mnombre LIKE "%M_quina%"

En este caso estas dos cadenas que se encuentren en cualquier posición de la columna MNOMBRE cumplirán el patrón.

Máquina, Máquina

Los caracteres comodines pueden aparecer en cualquier lugar de la cadena del patrón y puede haber varios caracteres comodines dentro de una misma cadena patrón.

Cuando el predicado LIKE se aplica a una columna que contenga un valor NULL el valor que devuelve el test es DESCONOCIDO por lo que la fila no es seleccionada.

Solamente por hacer una referencia de los comodines, diremos que son parecidos a los utilizados en el MS-DOS de las computadoras personales. En MS-DOS el asterisco (*) tiene la función del signo de porcentaje (%) en SQL y el signo de interrogación se utiliza en lugar del subrayado (_).

Se pueden localizar cadenas que no se ajusten a un patrón utilizando el formato NOT LIKE.

Ejemplo: Obtener los cursos que no tengan en su nombre una letra "i" en la segunda posición.

```
SELECT Mnomateria, Mnombre
FROM   Materia
WHERE Mnombre NOT LIKE  "_i%"
```

Resultado:

MNOMATERIA	MNOMBRE
0076	Bases de Datos
0119	Estructuras de Datos
0056	Estructuras Discretas
0559	Memorias y Perifericos
0028	Analisis Dinamico de Máquinas
0130	Elementos de Máquinas

2.3.5.5. Predicado de valor nulo (IS NULL).

Los valores NULL crean una lógica de tres valores. Para una fila determinada, el resultado de una condición de búsqueda puede ser CIERTO, FALSO o DESCONOCIDO (TRUE, FALSE y NULL). Un sistema de tres valores es más complejo y requiere mayor atención al introducir sentencias SQL e interpretar los resultados de la consulta.

A veces es útil comprobar explícitamente los valores NULL en una condición de búsqueda y manejarlos directamente, SQL proporciona un predicado especial de valor nulo (IS NULL). Su sintaxis es:

nombre de la columna IS [NOT] NULL

Ejemplo: Encontrar los grupos que aun no tienen un maestro asignado.

```
SELECT *
FROM   Grupo
WHERE  Gnomaestro IS NULL
```

Resultado:

GNOGRUPO	GNOMATERIA	GNOMAESTRO	GDIAS	GHORA	GSALON
1158	0134	NULL	MAJU	07:00 8:30	A521

La forma negada del predicado de valor nulo (IS NOT NULL) encuentra las filas con columnas que no contienen valores nulos:

Ejemplo: Reportar los grupos que ya tienen asignado su profesor.

```
SELECT *
FROM   Grupo
WHERE  Gnomaestro IS NOT NULL
```

Resultado:

GNOGRUPO	GNOMATERIA	GNOMAESTRO	GDIAS	GHORA	GSALON
1157	0076	72654545	LUMIVI	11:30 13:30	A211
1159	0119	75656566	SA	07:00 12:00	A121

A diferencia de los predicados vistos anteriormente. el predicado de valor nulo no puede producir un resultado NULL. Siempre será FALSO o CIERTO.

Se debe utilizar explícitamente el predicado de valores nulos para comprobar los valores NULL en una columna, es decir la palabra NULL no se puede emplear de manera directa en una comparación. El siguiente ejemplo marcaría un error de sintaxis, dado que NULL es una palabra reservada.:

```
SELECT *
FROM  Grupo
WHERE  Gnomaestro = NULL
```

2.3.5.6. Predicados compuestos.

Los predicados compuestos son combinaciones de otros predicados, simples, con los operadores lógicos AND, OR y NOT..

La palabra clave OR se utiliza para seleccionar filas donde se cumpla cualquiera de los predicados simples conectados con OR.

Ejemplo: Obtener la información de los cursos de las carreras de Eléctrica y Computación.

```
SELECT *;
FROM Materia
WHERE Mnocarrera = "32"
      OR Mnocarrera = "38"
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.00	32
0134	Sistemas Digitales	8	50.00	32
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32
0559	Memorias y Perifericos	10	100.00	32
0561	Microcomputadoras	10	500.00	32
0028	Analisis Dinamico de Máquinas	8	100.00	38
0130	Elementos de Máquinas	8	50.00	38

La condición AND se utiliza para combinar dos condiciones de búsqueda que deban ser ciertas simultáneamente:

Ejemplo: Encontrar la información sobre todas las materias de computación donde el laboratorio cueste mas de N$100.00.

```
SELECT *

FROM Materia
WHERE Mnocarrera = "32"
      AND Mcostolab > 100
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0561	Microcomputadoras	10	500.00	32

La palabra clave NOT se usa para seleccionar filas en donde la condición de búsqueda es falsa:

Ejemplo : Obtener el número, el nombre y la carrera de las materias que no sean de computación.

```
SELECT Mnomateria, Mnombre, Mnocarrera
FROM   Materia
WHERE  NOT Mnomateria = "32"
```

Resultado:

MNOMATERIA	MNOMBRE	MNOCARRERA
0028	Analisis Dinamico de Máquinas	38
0130	Elementos de Máquinas	38
0024	Circuitos Electricos	40
0138	Dispositivos Electronicos	40

El operador NOT puede colocarse antes de cualquier expresión condicional válida.

Hay que evitar cometer el error común de poner el NOT antes del operador de comparación. La siguiente cláusula no es legal:

```
WHERE  Mnomateria NOT = "32"
```

Siempre que una cláusula WHERE contenga más de dos condiciones que estén conectadas por diferentes operadores booleanos, el sistema debe decidir el orden de ejecución. Si la cláusula WHERE no tiene ningún paréntesis, el sistema sigue la siguiente secuencia:

- NOT
- AND
- OR

Cuando se incluyen paréntesis estos serán los primeros en ser evaluados. El orden de evaluación de los paréntesis es de izquierda a derecha y en caso de existir paréntesis anidados se evalúan primero los mas internos.

Ejemplo: Seleccionar toda la información sobre las materias de Computación que su laboratorio cueste N$0.00 o tenga 8 créditos.

```
SELECT *
FROM Materia
WHERE Mnocarrera = "32"
AND (Mcostolab = 0
OR Mcred = 8)
```

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0076	Bases de Datos	8	100.00	32
0134	Sistemas Digitales	8	50.00	32
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32

De cualquier manera para asegurar la portabilidad es recomendable usar los paréntesis. La siguiente figura presenta las tablas de verdad de los operadores booleanos.

AND	CIERTO	FALSO	DESCONOCIDO
CIERTO	CIERTO	FALSO	DESCONOCIDO
FALSO	FALSO	FALSO	DESCONOCIDO
DESCONOCIDO	DESCONOCIDO	DESCONOCDO	DESCONOCIDO

OR	CIERTO	FALSO	DESCONOCIDO
CIERTO	CIERTO	CIERTO	CIERTO
FALSO	CIERTO	FALSO	DESCONOCIDO
DESCONOCIDO	CIERTO	DESCONOCDO	DESCONOCIDO

NOT	CIERTO	FALSO	DESCONOCIDO
	FALSO	CIERTO	DESCONOCIDO

Fig. 2.5 Tablas de verdad de los operadores booleanos AND, OR y NOT.

2.3.6. Ordenación de los resultados de una consulta (cláusula ORDER BY).

SQL puede ordenar los resultados de una consulta incluyendo la cláusula ORDER BY en la sentencia SELECT.

Sintaxis simplificada:

SELECT lista de selección
FROM lista de tablas
[WHERE lista de condiciones]
[ORDER BY nombre de columna [ASC/DESC] [,...]]

Ejemplo: Visualizar toda la tabla MATERIA. Ordenar la salida en forma ascendente por el costo del laboratorio.

SELECT *;
FROM Materia;
ORDER BY Mcostolab

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32
0134	Sistemas Digitales	8	50.00	32
0130	Elementos de Máquinas	8	50.00	38
0138	Dispositivos Electronicos	10	90.00	40
0076	Bases de Datos	8	100.00	32
0559	Memorias y Perifericos	10	100.00	32
0028	Analisis Dinamico de Máquinas	8	100.00	38
0024	Circuitos Electricos	10	150.00	40
0561	Microcomputadoras	10	500.00	32

A menos que se especifique lo contrario en la cláusula ORDER BY la ordenación se hará en forma ascendente. Se podía haber escrito en forma explícita la ordenación ascendente. La cláusula ORDER BY quedaría de la siguiente manera:

ORDER BY Mcostolab ASC

En la columna MCOSTOLAB existen valores duplicados. La forma en que son presentadas las filas con MCOSTOLAB duplicadas no se puede determinar, además de que varia en los diferentes productos SQL. Generalmente son mostradas en el orden que el DBMS las accesa.

La ordenación que se obtiene es únicamente a la salida, no se afecta físicamente el orden de los datos en la tabla.

Para solicitar los datos en secuencia descendente usamos el parámetro DESC de la cláusula ORDER BY.

Ejemplo: Visualizar toda la tabla MATERIA. Ordenar la salida en forma descendente por el costo del laboratorio, en caso de haber costos de laboratorio iguales ordenarlos por el nombre de la materia, también en orden descendente.

```
SELECT *;
FROM Materia;
ORDER BY Mcostolab DES, Mnombre DES
```

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0561	Microcomputadoras	10	500.00	32
0024	Circuitos Electricos	10	150.00	40
0559	Memorias y Perifericos	10	100.00	32
0076	Bases de Datos	8	100.00	32
0028	Analisis Dinamico de Máquinas	8	100.00	38
0138	Dispositivos Electronicos	10	90.00	40
0134	Sistemas Digitales	8	50.00	32
0130	Elementos de Máquinas	8	50.00	38
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32

La primera especificación de la ordenación MCOSTOLAB es la clave de ordenación *mayor*, las que le siguen (MNOMBRE en éste caso) son progresivamente claves de ordenación *menores*, que son utilizados en caso de que la clave *mayor* se encuentre repetida.

La terminología clave de ordenación *mayor - menor* fue escogida entre varias formas de referirse al mismo par de conceptos, también es frecuente encontrar : *clave primaria - clave secundaria, campo de ordenación de primer nivel - campo de ordenación de segundo nivel, secuencia de ordenación MNOMBRE dentro de MCOSTOLAB* ... etcétera.

La cláusula ORDER BY puede también referenciar a una columna usando su posición relativa en la salida de datos que se designa en la cláusula SELECT. Si la columna de resultados de la consulta utilizada para la ordenación es una columna calculada, no tiene nombre que se pueda emplear para especificarla en la cláusula ORDER BY, en este caso se emplea el número de columna en lugar del nombre.

Ejemplo: Listar la clave de la materia, su nombre y el costo de laboratorio aumentado un diez por ciento. Ordenar en orden ascendente por el costo de laboratorio aumentado un diez por ciento.

```
SELECT Mnomateria, Mnombre, Mcostolab * 1.10
FROM Materia
ORDER BY 3
```

Resultado:

MNOMATERIA	MNOMBRE	EXP_3
0119	Estructuras de Datos	0.0000
0056	Estructuras Discretas	0.0000
0134	Sistemas Digitales	55.0000
0130	Elementos de Máquinas	55.0000
0138	Dispositivos Electronicos	99.0000
0076	Bases de Datos	110.0000
0559	Memorias y Perifericos	110.0000
0028	Analisis Dinamico de Máquinas	110.0000
0024	Circuitos Electricos	165.0000
0561	Microcomputadoras	550.0000

2.4. Consultas multitabla (composiciones).

Muchas consultas útiles solicitan datos procedentes de dos o mas tablas en la base de datos. Por ejemplo las siguientes dos peticiones de ejemplo deben extraer información de dos y tres tablas:

- Para cada empleado académico se necesitan saber todos sus datos y el nombre de la carrera para la cual trabaja.

Esta consulta requiere accesar la tabla ACADEMIC para accesar los datos del académico y la tabla CARRERA para accesar el nombre de la carrera.

- Por cada grupo requerimos saber el nombre número de grupo, el nombre de la materia que se imparte y el nombre del profesor asignado.

Esta consulta requiere el acceso a tres tablas: GRUPO, ACADEMIC, y CARRERA, las dos últimas para extraer el nombre del maestro y de la materia respectivamente.

La operación que nos permite que una única sentencia SELECT pueda hacer referencia a columnas de mas de una tabla es conocida es el lenguaje relacional como una operación *join* (composición). La operación join permite que una consulta especifique la fusión de columnas desde dos o mas tabla y relacione los valores que encuentre en las columnas de cada tabla.

2.4.1. Consideraciones SQL para consultas multitabla.

La sintaxis de las consultas multitabla no requieren ninguna característica especial del lenguaje, aparte de las descritas para las consultas monotabla. Sin embargo algunas consultas multitabla no pueden ser expresadas sin algunas características adicionales del lenguaje SQL que se describen en las siguientes secciones.

2.4.1.1. Nombres de columna cualificados.

Como se explicó en el capítulo 1, cuando una base de datos tiene nombres de columnas que coincidan en varias tablas y dos o más de estas tablas se utilizan en una operación join, es necesario utilizar en nombre de columna cualificado para identificar las columnas. Un nombre cualificado especifica el nombre de la columna y la tabla que contiene esa columna.

La sintaxis para cualificar la una columna es:

Nombre de tabla. Nombre de Columna

Un nombre de columna cualificado puede ser utilizado en una sentencia SELECT en cualquier lugar en donde se permite un nombre de columna. La tabla especificada es el nombre de columna debe, naturalmente encontrarse listada en la cláusula FROM.

Usar nombres de columna cualificados en una consulta multitabla es siempre una buena medida. La desventaja es que el texto de la petición se hace más largo.

2.4.1.2. Selecciones de todas las columnas.

Como se recuerda, para seleccionar todas las columnas de una tabla designada en la cláusula FROM utilizamos SELECT *. En una consulta multitabla un asterisco (*) selecciona todas las columnas de todas las tablas que se encuentren en la cláusula FROM.

En la mayoría de productos SQL el asterisco puede ser cualificado con el nombre de una tabla. Por ejemplo para recuperar todas las columnas de la tabla materia y ademas el grupo donde se imparte escribiríamos:

```
SELECT Materia.*, Grupo.Gnogrupo
```

2.4.1.3. Alias de tablas.

Los alias de tablas se utilizan cuando deseamos cambiar o simplificar el nombre de una tabla, por ejemplo cuando el nombre de la tabla es muy grande o se esta utilizando una tabla que pertenece a otro usuario.

La sintaxis para darle un alias a una tabla en la cláusula FROM es la siguiente:

FROM tabla alias,[tabla alias, ...]

Ejemplo: Seleccionar los campos MNOMATERIA y MNOMBRE de la tabla MATERIA, para la materia con clave "0076".

```
SELECT M.mnomateria, M.mnombre
FROM Materia M
WHERE M.nomateria = "0076"
```

Aquí se utilizó el alias "M" para la tabla MATERIA, por lo que donde debe aparecer el nombre MATERIA se substituye éste por su alias "M".

La única condición al usar múltiples alias en una cláusula FROM es que todos los alias que aparezcan deben de ser diferentes.

2.4.2. Composición de dos tablas.

Desde un punto de vista conceptual, la composición de dos tablas es la concatenación de filas de las dos tablas, en donde los valores de una columna de la primera tabla coinciden con valores de una columna de la segunda tabla. El resultado de una operación join es una nueva tabla que tiene una fila por cada valor coincidente entre las dos tablas originales. Tratemos de explicar esta operación con un ejemplo.

La siguiente figura muestra dos tablas que serán unidas para formar una tercera tabla:

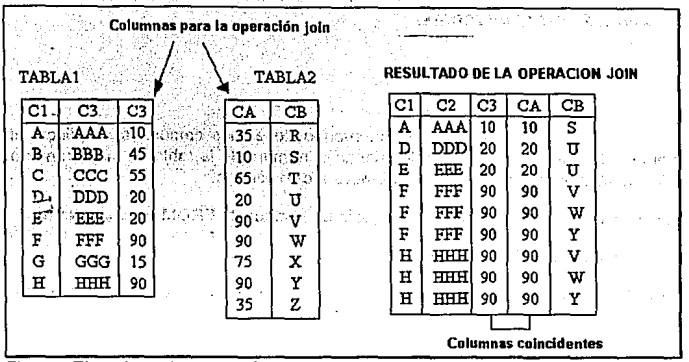


Fig. 2.6 Ejemplo de la operación join.

El primer paso para una operación join es especificar la columna de cada tabla que servirá para la composición, en el ejemplo se especifican la columna C3 de TABLA1, y la columna CA, de TABLA2. Con esto se especifica que , siempre que el valor C3 de una fila en TABLA1 sea igual al valor CA de una fila en TABLA2, se crea una fila en la tabla de resultado. Se observa en la tabla de resultados que:

- En TABLA, las filas con valores C3 de 45, 55 y 15 no coinciden con ningún valor en TABLA2 por lo que no aparecen en la tabla resultante.
- En TABLA1 hay dos filas con un valor C3 de 90. Cada una de ellas coincide con tres valores de la columna CA de TABLA2. Esto hace un total de seis combinaciones por lo que se producen seis filas en la tabla de resultado.

Generalmente una clave externa en una tabla, y su correspondiente clave primaria de otra tabla, son especificadas como las columnas que sirven de liga para una operación join. Sin embargo esto no es necesario. Cualquier par de columnas de dos tablas con tipos de datos que puedan compararse se pueden utilizar para realizar una operación join.

Las operaciones join son el fundamento para el procesamiento de consultas multitabla en SQL

He aquí la sentencia SELECT que corresponde a la operación join que se utilizó de ejemplo:

```
SELECT C1, C2, CA, CB
FROM   TABLA1, TABLA2
WHERE  C3 = CA
```

Esta sentencia SELECT tiene el mismo aspecto que las consultas que se han realizado hasta ahora, pero tiene dos características nuevas. En primer lugar la cláusula FROM tiene dos tablas en lugar de una. En segundo lugar la condición de búsqueda:

```
WHERE  C3 = CA
```

compara las columnas de dos tablas diferentes. A estas dos columnas se les conoce como columnas join o columnas de emparejamiento para las dos tablas.

En caso de existir valores NULL en las columnas join, estos valores no coinciden con ningún otro valor, inclusive un valor NULL no coincide con otro valor NULL.

2.4.3. Consulta padre / hijo

Las consultas multitabla más comunes implican a dos tablas que tienen una relación natural padre/hijo.

Podemos recordar del capítulo anterior que las claves foráneas y las claves primarias crean relaciones padre/hijo en la base de datos. El hijo es la tabla que contiene la clave foránea, la clave que contiene la clave primaria es el padre de la relación. Para realizar la relación padre/hijo en una consulta debe especificarse una condición de búsqueda que compare la clave foránea y la clave primaria.

Ejemplo: Listar todas las materias e incluir el nombre de la carrera a la que pertenece cada una de ellas.

```
SELECT Materia.*, Cnombre
FROM Materia, Carrera
WHERE Cnocarrera = Mnocarrera
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA	CNOMBRE
0076	Bases de Datos	8	100.00	32	Computación
0134	Sistemas Digitales	8	50.00	32	Computación
0119	Estructuras de Datos	8	0.00	32	Computación
0056	Estructuras Discretas	7	0.00	32	Computación
0559	Memorias y Periféricos	10	100.00	32	Computación
0561	Microcomputadoras	10	500.00	32	Computación
0028	Análisis Dinámico de Máquinas	8	100.00	38	Eléctrica
0130	Elementos de Máquinas	8	50.00	38	Eléctrica
0024	Circuitos Eléctricos	10	150.00	40	Mecánica
0138	Dispositivos Electrónicos	10	90.00	40	Mecánica

SQL no requiere que las columnas para la operación join sean incluidas en los resultados de una consulta multitabla. Con frecuencia sin omitidos porque generalmente las claves primarias y las claves foráneas suelen ser números de identificación (como el número de carrera en el ejemplo anterior).

2.4.4. Join con criterio de selección de fila.

Realmente en pocas ocasiones deseamos ver todas las filas y columnas de un resultado join. La condición de búsqueda que especifica las columnas join puede combinarse con otros predicados para restringir el contenido del resultado.

Ejemplo: Listar todas las materias de Computación y Mecánica, incluir el nombre de la carrera.

```
SELECT Materia.Mnomateria, Materia.Mnombre, Materia.Mcred, Materia.Mcostolab,
            Carrera.Cnombre
FROM Materia, Carrera
WHERE Carrera.Cnocarrera = Materia.Mnocarrera
      AND (Materia.Mnocarrera = "32"
      OR (Materia.Mnocarrera = "40"))
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	CNOMBRE
0076	Bases de Datos	8	100.00	Computación
0134	Sistemas Digitales	8	50.00	Computación
0119	Estructuras de Datos	8	0.00	Computación
0056	Estructuras Discretas	7	0.00	Computación
0559	Memorias y Periféricos	10	100.00	Computación
0561	Microcomputadoras	10	500.00	Computación
0024	Circuitos Eléctricos	10	150.00	Mecánica
0138	Dispositivos Electrónicos	10	90.00	Mecánica

El siguiente ejemplo hace realiza la operación join y sólo visualiza columnas de una tabla (ACADEMIC). Esto no es común. El objetivo de la operación join es permitir la selección de filas basadas en la información que se encuentra en una segunda tabla:

Ejemplo: Encontrar los maestros que tengan su coordinación en el edificio A202

Resultado:

```
SELECT Academic.anoemplead, Academic.anombre
FROM Academic, Carrera
WHERE Carrera.cnocarrera = Academic.anocarrera
      AND Carrera.cedificio = "A202"
```

Resultado:

ANOEMPLEAD	ANOMBRE
72654545	Juan Méndez
75656566	Pedro Benítez

Se observa que la consulta sólo visualiza datos de la tabla ACADEMIC, pero la operación join es necesaria para determinar que coordinaciones de carrera están localizadas en el edificio "A202".

2.4.5. Join de tres tablas.

Veamos el siguiente ejemplo donde requerimos la operación join de tres tablas:

Ejemplo: Listar los cursos que ya tienen asignado profesor, visualizando el nombre del profesor, su sueldo y el nombre de la materia, ordenar los resultados por el nombre del profesor.

```
SELECT Academic.anombre, Academic.asueldo, Grupo.gnogrupo,
      Materia.mnombre
FROM Grupo, Academic, Materia
WHERE Academic.anoemplead = Grupo.gnomaestro
      AND Grupo.gnomateria = Materia.mnomateria
ORDER BY Academic.anombre
```

Resultado:

ANOMBRE	ASUELDO	GNOGRUPO	MNOMBRE
Alberto Alvares	3000.00	2504	Circuitos Eléctricos
Alberto Alvares	3000.00	2501	Dispositivos Electrónicos
Armando Cruz	5000.00	2706	Elementos de Máquinas
Carlos Cuenca	7000.00	2705	Análisis Dinámico de Máquinas
Juan Méndez	3800.00	1157	Bases de Datos
Pedro Benítez	5200.00	1159	Estructuras de Datos

En este ejemplo fue necesario visualizar MNOMBRE de la tabla MATERIA, el número de grupo GNOGRUPO de la tabla GRUPO y el nombre del profesor de la tabla ACADEMIC: Esto requirió el join de tres tablas. Dependiendo del producto SQL, es el número de tablas que se pueden especificar en una operación join. Algunos productos como DB2/SQL de IBM ponen como límite un máximo de 16 tablas y algunos otros como SQL Server de Sybase no ponen límite a las tablas que operen en una operación join.

Una operación join de tres tablas requiere de dos condiciones join. Las tablas ACADEMIC y GRUPO se relacionan mediante la condición:

-- WHERE Academic.anoemplead = Grupo.gnomaestro

y las tablas GRUPO y MATERIA se relacionan con la siguiente condición:

AND Grupo.gnomateria = Materia.mnomateria

2.4.6. Producto cartesiano.

El siguiente ejemplo tiene por objeto mostrar el *producto cartesiano* (cross product) entre dos tablas. En una operación join donde no se pone una cláusula WHERE, cada fila de la primera tabla se empareja con cada fila de la segunda tabla. Esto ocurre debido a que la condición que restringe el resultado de la operación join no esta presente.

Ejemplo: Formar el producto cartesiana de las tablas GRUPO y CARRERA

SELECT *
FROM Academic, Estudian

Resultado:

ANOEMPLEAD	ANOMBRE	ENOCUENTA	ENOMBRE
75212122	Carlos Cuenca	84254321	Carolina Bisset
75212122	Carlos Cuenca	84321212	Juan Bisset
75212122	Carlos Cuenca	84254321	Alejandro Sauza
75212122	Carlos Cuenca	82323233	Pedro Sauza
75521245	Armando Cruz	84254321	Carolina Bisset
75521245	Armando Cruz	84321212	Juan Bisset
75521245	Armando Cruz	84254321	Alejandro Sauza
75521245	Armando Cruz	82323233	Pedro Sauza
75165232	Alberto Alvares	84254321	Carolina Bisset
75165232	Alberto Alvares	84321212	Juan Bisset
75165232	Alberto Alvares	84254321	Alejandro Sauza
75165232	Alberto Alvares	82323233	Pedro Sauza
72654545	Juan Méndez	84254321	Carolina Bisset
72654545	Juan Méndez	84321212	Juan Bisset
72654545	Juan Méndez	84254321	Alejandro Sauza
72654545	Juan Méndez	82323233	Pedro Sauza
75656566	Pedro Benítez	84254321	Carolina Bisset
75656566	Pedro Benítez	84321212	Juan Bisset
75656566	Pedro Benítez	84254321	Alejandro Sauza
75656566	Pedro Benítez	82323233	Pedro Sauza

Se observa que se generaron todas las combinaciones posibles de filas. Es posible que el número de filas generado sea muy elevado. Por ejemplo si las tablas que participan en una operación join sin condición tiene mil filas cada una el producto cartesiano tendría un millón de filas. El producto cartesiano es realmente una operación que raramente se utilizará.

El producto cartesiano es el número de veces que SQL tiene que evaluar la condición de búsqueda para la selección de filas.

2.4.7. Join basadas en desigualdad.

El término join se aplica a cualquier consulta que combina datos de dos tablas mediante comparación de los valores en una pareja de columnas de ambas tablas. Aunque las operaciones join basadas en la igualdad entre columnas correspondiente (equijoin) son las más habituales, SQL permite también componer tablas basándose en otros operadores de comparación.

Ejemplo: Se desea listar los laboratorios, que los profesores no puedan pagar con el cinco por ciento de su salario mensual. Listar el nombre del profesor y las materias que no puedan pagar.

```
SELECT Academic.anoemplead, Academic.anombre, Academic.asueldo,;
       Materia.mnomateria, Materia.mcostolab;
FROM Academic, Materia;
WHERE Materia.mcostolab > Academic.asueldo*0.05
```

Resultado:

ANOEMPLEAD	ANOMBRE	ASUELDO	MNOMATERIA	MCOSTOLAB
75212122	Carlos Cuenca	7000.000561		500.00
75521245	Armando Cruz	5000.000561		500.00
75165232	Alberto Alvares	3000.000561		500.00
72654545	Juan Méndez	3800.000561		500.00
75656566	Pedro Benítez	5200.000561		500.00

2.5. Combinación de resultados de consulta (UNION).

La operación UNION nos permite combinar los resultados de dos o mas consultas en una única tabla de resultados totales.

La sintaxis para la operación UNION:

SENTENCIA SELECT
UNION [ALL]
SENTENCIA SELECT
[UNION [ALL]
SENTENCIA SELECT ...]

A continuación se presenta un ejemplo para ilustrar esta capacidad de SQL.

Suponga que se desea listar todos los alumnos y todos los maestros. A los alumnos, marcarlos con una "E" y a los maestros marcarlos con una "A". Esta sencilla solicitud se puede satisfacer ejecutando dos sentencias SELECT:

1. Seleccionar todos los números de cuenta y los nombres de los estudiantes:

SELECT Academic.anoemplead, Academic.anombre, "A"
FROM Academic

2. Seleccionar a todos los números y nombres de los maestros.

SELECT Estudian.enocuenta, Estudian.enombre, "E"
FROM Estudian

La operación UNION produce una única tabla de resultados que combina las filas de la primera consulta con las filas de los resultados de la segunda consulta. La sentencia SELECT que realiza la operación UNION de el ejemplo, es la siguiente:

```
SELECT Academic.anoemplead, Academic.anombre, "A"
FROM Academic
UNION
SELECT Estudian.enocuenta, Estudian.enombre, "E"
FROM Estudian
```

Resultado:

ANOEMPLEAD	ANOMBRE	EXP 3
72654545	Juan Méndez	A
75165232	Alberto Alvares	A
75212122	Carlos Cuenca	A
75521245	Armando Cruz	A
75656566	Pedro Benítez	A
82323233	Pedro Sauza	E
84254321	Alejandro Sauza	E
84254321	Carolina Bisset	E
84321212	Juan Bisset	E

Existen varias restricciones sobre las tablas que pueden combinarse con una operación UNION:

- Ambas tablas deben contener el mismo número de columnas
- Cada columna en el resultado del de un SELECT debe de ser del mismo tipo que la correspondiente columna en el otro SELECT.
- Solo el SELECT final puede tener una cláusula ORDER BY, si es incluida afecta al resultado de la UNION.
- La operación UNION no se puede usar para unir subconsultas. Las subconsultas se estudian posteriormente en este mismo capítulo.

Se observa en el ejemplo que dado que los nombres de las dos consultas no son iguales el query presenta los nombres de las columnas de la primera sentencia SELECT, podemos cambiar el encabezado de las columnas mediante en empleo de alias de columna.

2.5.1. Uniones y filas duplicadas.

Dado que la operación UNION combina las filas de dos resultados, la tendencia sería de producir filas duplicadas, por omisión la operación UNION elimina las filas duplicadas como parte de su procesamiento.

Si se desea retener las filas duplicadas en una operación UNION, se puede especificar la palabra reservada ALL a continuación de la palabra UNION

Ejemplo : Presentar las carreras a las que pertenecen los alumnos y las carreras en donde trabajan los académicos.

```
SELECT Academic.anocarrera
FROM Academic
UNION
SELECT Estudian.enocarrera
FROM Estudian
```

Resultado:

ANOCARRERA
32
38
40

Ejemplo : Presentar todas las carreras a las que pertenecen los alumnos y las carreras en donde trabajan los académicos.

```
SELECT Anocarrera
FROM Academic
UNION ALL
SELECT Enocarrera
FROM Estudian
```

Resultado:

ANOCARRERA
38
38
40
32
32
32
32
38
40

Nótese que el manejo de duplicados por omisión en la sentencia SELECT y en la operación **UNION** son opuestos. En la sentencia **SELECT** por omisión es seleccionar todas la filas, si se desea eliminar las filas duplicadas se tiene que especificar **SELECT DISTINCT**. En el caso de la operación **UNION** las filas duplicadas se omiten por default, en caso de desearse la duplicidad de filas se debe usar **UNION ALL**.

El proceso de eliminación de filas duplicadas consume mucho tiempo, especialmente si el resultado contiene muchas filas. Si de antemano se sabe que una consulta no va a tener filas duplicadas en la operación **UNION**, es recomendable utilizar **UNION ALL**, ya que la consulta se ejecutara mas rápidamente.

2.5.2. Uniones y ordenación.

La cláusula ORDER BY tiene algunas restricciones cuando se utiliza con una operación **UNION**:

- La cláusula **ORDER BY** solo puede aparecer una vez, y debe ser la ultima cláusula de toda la sentencia.
- La cláusula **ORDER BY** debe referenciar a una columna por su número de columna relativo en la cláusula **SELECT**. Esto es debido a que las columnas de las dos cláusulas **SELECT** pueden ser diferentes.

Ejemplo: Listar los números y nombres de los maestros y alumnos de la carrera de Computación. Ordenarlos por nombre en orden ascendente.

```
SELECT Academic.anoemplead, Academic.anombre
FROM Academic
WHERE Academic.anocarrera = 32
UNION
SELECT Estudian.enocuenta, Estudian.enombre
FROM Estudian
WHERE Estudian.enocarrera = 32
ORDER BY 2
```

Resultado:

ANOEMPLEAD	ANOMBRE
84254321	Carolina Bisset
84321212	Juan Bisset
72654545	Juan Méndez
75656566	Pedro Benítez

2.5.3. Uniones múltiples.

La operación **UNION** puede utilizarse para combinar tres o mas conjuntos de resultados como se muestra en la siguiente figura:

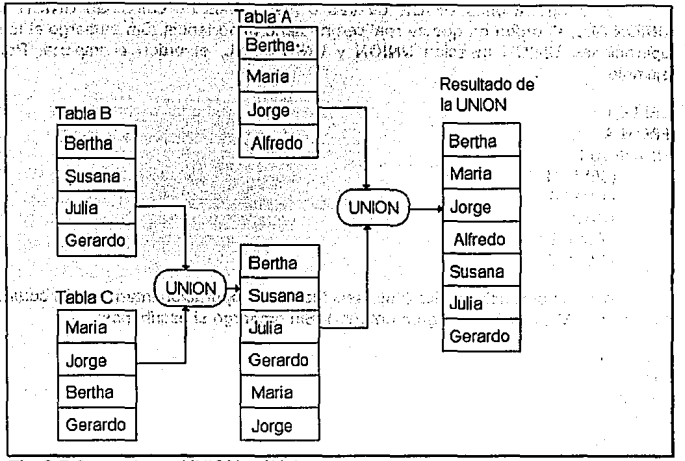


Fig. 2.7 Operaciones UNION múltiples.

La UNION de la TABLA B y de la TABLA C, produce una tabla intermedia que luego se combina con la TABLA A en otra operación UNION. El ejemplo que se utiliza en la figura se escribe:

```
SELECT *
FROM A
UNION
      (SELECT *
      FROM B
      UNION
      SELECT *
      FROM C)
```

El resultado se muestra en la figura anterior.

Los paréntesis que aparecen en la consulta indican que operación UNION debe ser realizada en primer lugar. Realmente en el ejemplo no existe diferencia alguna si se efectúan en otro orden el resultado será el mismo.

Si la operaciones UNION de tres o mas tablas no combinan UNION y UNION ALL, el orden en que se realicen no tiene importancia. Sin embargo si las operaciones UNION mezclan UNION y UNION ALL, el orden si importa. Por ejemplo:

```
SELECT *
FROM A
UNION ALL
      (SELECT *
      FROM B
      UNION
      SELECT *
      FROM C)
```

Aquí se producirían diez filas (seis filas de la operación interna, más cuatro filas de la TABLA A, Ver la figura anterior). Sin embargo si escribimos:

```
(SELECT *
FROM A
UNION ALL
SELECT *
FROM B)
UNION
SELECT *
FROM C
```

Entonces se producen las mismas siete filas que se muestran en la figura anterior, dado que la operación UNION externa elimina las filas duplicadas.

2.6. Consultas sumarias.

No en todas las peticiones de información se requieren todas las filas de detalle. Por ejemplo imagine que deseamos saber el número total de ayudantes de profesor que existen en cada carrera, en el resultado no se desea ver el detalle de cuantos ayudantes tiene cada profesor.

SQL ofrece la posibilidad de formar grupos a partir de las filas seleccionadas de una tabla. SQL soporta este tipo de consultas sumarias mediante funciones de columna y mediante las cláusulas GROUP BY y HAVING de la cláusula SELECT, que se describen en las secciones siguientes.

2.6.1. Funciones de columna.

Una función de columna es utilizada para explorar una columna de valores seleccionados y realizar un cálculo basado en esos valores, produciendo un único dato en la salida. Las funciones de columna son: AVG, SUM, MAX, MIN, y COUNT.

2.6.1.1. Función de columna SUM,

La función columna SUM() calcula la suma de una columna de valores. El tipo de datos de esta columna debe ser numérico. El tipo de datos a la salida es del mismo tipo de la columna.

Ejemplo: ¿Cual es la suma de todos los costos de laboratorios?.

```
SELECT SUM(Materia.mcostolab);
FROM Materia
```

Resultado:

```
SUM_MCOSTO
    1140.00
```

2.6.1.2. Función de columna AVG,

La función de columna AVG() calcula el promedio de una columna de valores, los cuales deben de ser de tipo numérico. Dado que el resultado de una función AVG() se realiza sumando todos los valores y dividiendo entre el número de filas, el resultado puede tener un tipo de dato distinto al de la columna. Por ejemplo al aplicar la función de columna AVG() a una columna con valores de tipo entero, el resultado será un número decimal o un número de coma flotante, dependiendo del producto SQL que se use.

Ejemplo: ¿Cual es el valor promedio de todos los laboratorios?.

```
SELECT AVG(Mcostolab)
FROM Materia
```

Resultado:

```
AVG_MCOSTO
    114.00
```

Dado que el resultado es un valor calculado y no un valor almacenado, el nombre de la columna de resultado será determinado por el DBMS del producto SQL que se use.

En este ejemplo y el anterior se utilizo toda la tabla, sin embargo esto no es imprescindible. Una función de columna podría aplicarse a un subconjunto de filas, usando la cláusula WHERE.

2.6.1.3. Funciones de columna MAX y MIN.

Las funciones de columna MAX() y MIN() determinan los valores mayor y menor de una selección de columnas, respectivamente. Ambas funciones se pueden aplicar a columnas numéricas, de cadenas de caracteres o de fecha/hora. Las funciones de columna MAX() y MIN() producen un resultado expresado en el mismo tipo de datos que la columna a la que se aplica.

Ejemplo: ¿Cual es el valor mínimo y máximo del costo de laboratorio de todos los cursos ofrecidos?

```
SELECT MIN(Materia.mcostolab), MAX(Materia.mcostolab);
FROM Materia;
WHERE Materia.mnomateria = "32"
```

Resultado:

MIN_MCOSTO	MAX_MCOSTO
0.00	500.00

Cuando se usan las funciones de columna MAX() y MIN() con tipos de datos de caracter, se usa el código de ASCII o el EBCDIC (dependiendo el tipo de computador donde se corre la consulta) para determinar los valores máximos y mínimos.

Ejemplo: Obtener el primer nombre de materia y el último nombre de materia que aparecería en la tabla MATERIA si se ordenara en forma alfabética.

Resultado:

MIN_MNOMBR	MAX_MNOMBR
Análisis Dinámico de Máquinas	Sistemas Digitales

2.6.1.4. Función de columna COUNT(*).

La función COUNT(*), determina el número de filas seleccionadas por una sentencia SELECT. Simplemente cuenta el número de filas que cumplen con el criterio de selección. El resultado siempre es un número entero.

Ejemplo: ¿Cuantos cursos de Computación están registrados?

```
SELECT COUNT(*)
FROM Materia
WHERE Materia.mnocarrera = "32"
```

Resultado:

```
CNT
 6
```

2.6.1.5. Eliminación de filas duplicadas (DISTINCT).

Todas las funciones de columna permiten la palabra clave DISTINCT para que solamente utilicen los valores únicos. Cualquier valor en una columna que aparezca en mas de una fila, se usara solo una vez si se emplea DISTINCT.

La segunda variación que encontramos en la función de columna COUNT nos permite examinar una columna de una selección de filas, para determinar cuantos valores únicos aparecen en esa columna.

Ejemplo: ¿Cuantas carreras ofrecen alguna materia?

```
SELECT COUNT(DISTINCT Materia.mnocarrera);
FROM Materia
```

Resultado:

DCNT
3

El uso de **DISTINCT** en las funciones de columna MAX() y MIN() no altera el resultado.

2.6.1.6. Valores NULL en funciones de columna.

El manejo de los valores **NULL** en las funciones de columna se rige bajo las siguientes reglas:

- Si alguno de los valores de la columna es **NULL**, éste será ignorado para el propósito de cálculos en la función de columna.
- Si todos los datos son **NULL**, la función de columna **COUNT()** devuelve un cero. Las demás funciones de columna devuelven un valor **NULL**.
- Si la columna esta vacía todas las funciones devuelven un valor de cero.
- El uso de la función **COUNT(*)** cuenta las filas seleccionadas y le afectan los valores **NULL** en las columnas.

Ejemplo: Considere la siguiente tabla y sume los valores de sus columnas.

Tabla1

COL1	COL2
10	20
NULL	30

```
SELECT SUM(Col1), SUM(Col2)
FROM Tabla1
```

Resultado:

SUM_COL1	SUM_COL2
10	50

2.6.2. Consultas agrupadas. Cláusula GROUP BY.

SQL ofrece la posibilidad de formar grupos a partir de todas las filas de una tabla en una sola consulta, y después aplicar las funciones de columna a cada grupo. Cuando se incluye una cláusula GROUP BY en una consulta, todas las filas seleccionadas son agrupadas por un valor común de una columna especifica. La función de columna que se especifique actúa para cada grupo.

Sintaxis simplificada:

SELECT [DISTINCT] lista de selección
FROM [lista de tablas]
[WHERE condiciones de búsqueda]
[GROUP BY [ALL] columna/expresión]
[ORDER BY columna/no relativo en la lista de selección/expresión [ASC/DES]]

Ejemplo: Para cada una de las carreras obtener el promedio del costo de todos sus laboratorios.

```
SELECT Materia.mnocarrera, AVG(Materia.mcostolab)
FROM Materia
GROUP BY Materia.mnocarrera
```

Resultado:

MNOCARRERA	AVG MCOSTO
32	125.00
38	75.00
40	120.00

Esta consulta agrupa tres conjuntos de filas, uno para cada materia. A cada grupo se le aplica la función de columna AVG() y reporta el promedio de cada grupo.

Nótese que sin especificarse una cláusula ORDER BY el resultado se muestra ordenado, esto es debido a que internamente el sistema realiza la ordenación para separar las filas en grupos.

La cláusula WHERE puede usarse para seleccionar las filas antes de la formación de grupos.

Ejemplo: Para cada carrera que ofrezca alguna materia determinar la suma de los costos de laboratorio que tengan 8 créditos. Visualizar la salida en forma descendente por el número de carrera.

```
SELECT Mnocarrera, SUM(Mcostolab)
 FROM Materia;
 WHERE Mcred = 8
 GROUP BY Mnocarrera
 ORDER BY Mnocarrera DESC
```

Resultado:

MNOCARRERA	SUM MCOSTO
38	150.00
32	150.00

El número de carrera "40" no aparece en el resultado dado que no tiene materias con 8 créditos. La cláusula WHERE se aplica a filas individuales y no a grupos, y se realiza antes de formar éstos.

SQL puede agrupar resultados de consulta en base a contenidos de dos o mas columnas.

Ejemplo: Listar el máximo costo de laboratorio, dependiendo el número de créditos y separando las carreras.

```
SELECT Materia.mcred, Materia.mnocarrera, MAX(Materia.mcostolab)
 FROM Materia
 GROUP BY Materia.mcred, Materia.mnocarrera
```

Resultado:

MCRED	MNOCARRERA	MAX_MCOSTO
	732	0.00
	832	100.00
	838	100.00
	1032	500.00
	1040	150.00

2.6.3. Condiciones de búsqueda de grupos. Cláusula HAVING.

Por hacer una comparación la cláusula WHERE tiene la misma función que la cláusula HAVING, con la diferencia de que la primera se utiliza para seleccionar y rechazar filas individuales, mientras que la cláusula HAVING es utilizada para la selección o rechazo de grupos de filas.

Sintaxis simplificada:

```
SELECT [DISTINCT] lista de selección
FROM [lista de tablas]
[WHERE condiciones de búsqueda]
[GROUP BY [ALL] columna/expresión]
[HAVING condiciones de búsqueda]
[ORDER BY columna/no relativo en la lista de selección/expresión [ASC/DES]]
```

Ejemplo: Obtener el número de la carrera y el valor promedio del costo de laboratorio, para todas las carreras en los que la media exceda de $100

```
SELECT Mnocarrera, AVG(Mcostolab)
 FROM Materia
 GROUP BY Mnocarrera
 HAVING AVG(Mcostolab) > 100
```

Resultado:

MNOCARRERA	AVG MCOSTO
32	125.00
40	120.00

La cláusula HAVING solo se puede aparecer si se encuentra presente la cláusula ORDER BY, y se coloca inmediatamente después de ésta. La condición especificada en la cláusula HAVING, contiene una referencia la valor de una función de columna. Esto ocurre siempre, ya que si no lo hace la condición se refiere a filas individuales, y podría especificarse la condición en la cláusula WHERE.

En el ejemplo el costo promedio de toda la carrera "38" es menor a cien por lo que la carrera no aparece en el resultado.

Una sentencia SELECT puede tener, tanto cláusulas WHERE como cláusulas HAVING. La cláusula WHERE selecciona inicialmente filas para su inclusión en los grupos, y la cláusula HAVING selecciona ciertos grupos para que aparezcan en el resultado.

Ejemplo: Obtener el número de la carrera y el valor promedio del costo de laboratorio, para todas las carreras en los que la media exceda de $100. Seleccionar solamente las filas donde el costo del laboratorio sea mayor a $0.

```
SELECT Materia.mnocarrera, AVG(Materia.mcostolab);
 FROM Materia;
 WHERE Materia.mcostolab > 0;
 GROUP BY Materia.mnocarrera;
 HAVING AVG(Materia.mcostolab) > 100
```

Resultado:

MNOCARRERA	AVG MCOSTO
32	187.50
40	120.00

2.7. Subconsultas.

Una subconsulta permite utilizar el resultado de una consulta como parte de otra. Es decir una sentencia SELECT anidada. Muchas de las subconsultas que se presentan en esta sección se pudieron haber resuelto empleando la operación join. Sin embargo el uso de subconsultas es considerado, con frecuencia para proporcionar soluciones más sencillas. Hay algunas circunstancias donde una subconsulta se hace necesaria.

Las subconsultas hacen más fácil la escritura de sentencia SELECT, ya que permiten descomponer una consulta en partes y escribirla en una sola sentencia SELECT.

Examinemos una consulta sencilla que no podría ser expresada adecuadamente en una sentencia SELECT sencilla.

Visualizar el número y nombre del curso (o cursos) con la tarifa más elevada. La forma de saber cual es la tarifa máxima es la siguiente:

```
SELECT MAX(Costolab)
FROM Materia
```

El sistema regresa el valor de $500, lo que nos permite escribir la siguiente sentencia:

```
SELECT Mnocarrera, Mnombre
FROM Materia
WHERE Costolab = 500
```

Para tener el resultado de la consulta propuesta se requirió de la ejecución de dos sentencias SELECT independientes.

Recuerde que la siguiente sentencia no funciona:

```
SELECT Mnocarrera, Mnombre, MAX(Costolab)
FROM Materia
```

Dado que la sentencia SELECT tiene una función de columna, las otras columnas Mnocarrera, Mnombre deben de estar referenciadas en una cláusula GROUP BY.

Veamos la sintaxis de una subconsulta:

(SELECT [DISTINCT] lista de selección
FROM [lista de tablas]
[WHERE condiciones de búsqueda]
[GROUP BY columna/expresión]
[HAVING condiciones de búsqueda])

Una subconsulta esta siempre va encerrada entre paréntesis. Si se observa, la sintaxis de una subconsulta es similar al de una sentencia SELECT con cláusulas FROM, WHERE, HAVING, GROUP BY y HAVING. Existen algunas diferencias entre una subconsulta y una sentencia SELECT:

- Una subconsulta solo debe producir una columna de datos como resultado, es decir que una subconsulta solo tiene un elemento en la cláusula SELECT..
- La cláusula ORDER BY no puede ser especificada en una subconsulta.
- Los resultados de una subconsulta no son visibles. Esta es la razón por la que la cláusula ORDER BY no tiene sentido en una subconsulta
- Una subconsulta no puede ser la UNION de varias sentencias diferentes, solo se permite una sentencia SELECT.

2.7.1. Subconsultas en la cláusula WHERE.

En el siguiente ejemplo la cláusula WHERE examina la subconsulta para encontrar un valor desconocido. Después de evaluar la subconsulta, esta devuelve un único valor a la cláusula WHERE de la consulta principal, que entonces es evaluada para encontrar las filas que serán seleccionadas para darse en el resultado.

Ejemplo: Visualizar el número y nombre del curso (o cursos) con la tarifa más elevada.

```
SELECT Mnomateria, Mnombre, Mcostolab
 FROM Materia
 WHERE Mcostolab =
 (SELECT MAX(Mcostolab)
  FROM Materia)
```

Resultado:

MNOMATERIA	MNOMBRE	MCOSTOLAB
0561	Microcomputadoras	500.00

En el ejemplo la consulta realiza primero la subconsulta, encontrando que el máximo valor de un laboratorio es $500 y este valor lo substituye en la cláusula WHERE, quedando como sigue:

```
SELECT Mnomateria, Mnombre, Mcostolab;
 FROM Materia;
 WHERE Mcostolab = 500
```

La consulta principal se ejecuta entonces y se visualiza el resultado.

Para delimitar más aun el resultado de la consulta principal, la subconsulta puede contener una cláusula WHERE que excluya o incluya ciertas filas.

Ejemplo: Visualizar el número, nombre y costo de laboratorio de los cursos con tarifa mínima, excluyendo los de tarifa $0.

```
SELECT Mnomateria, Mnombre, Mcostolab;
 FROM Materia;
 WHERE Mcostolab =;
 (SELECT MIN(Mcostolab);
  FROM Materia;
  WHERE NOT Mcostolab = 0)
```

Resultado:

MNOMATERIA	MNOMBRE	MCOSTOLAB
0134	Sistemas Digitales	50.00
0130	Elementos de Máquinas	50.00

En la siguiente consulta - ejemplo se utiliza la misma condición tanto en la consulta principal como en la subconsulta.

Ejemplo: Visualizar el número y el nombre de los cursos de Computación con la mínima tarifa:

```
SELECT Mnomateria, Mnombre, Mcostolab
 FROM Materia
 WHERE Mnocarrera = "32"
 AND Mcostolab =
 (SELECT MIN(Mcostolab)
  FROM Materia
  WHERE Mnocarrera = "32"
  AND NOT Mcostolab = 0)
```

Resultado:

MNOMATERIA	MNOMBRE	MCOSTOLAB
0134	Sistemas Digitales	50.00

Pareciera que la condición de búsqueda MNOCARRERA = "32" es redundante, pero no es así. Considérese el caso de eliminar la condición de la cláusula WHERE de la consulta principal. La sentencia visualizaría todos los cursos, de cualquier carrera, con la tarifa mínima en la carrera de Computación, es decir $50.

Si se elimina la condición de búsqueda en la subconsulta se elige el costo de laboratorio mínimo para cualquier carrera y se aplica este costo a Computación. Si el costo mínimo en la subconsulta no es de Computación el resultado será una tabla vacía.

Este ejemplo nos mostró que debemos tener cuidado al escribir las consultas, aunque SQL es sencillo , la lógica de una consulta puede ser compleja.

Veamos ahora un ejemplo de la consulta y la subconsulta principal hacen referencia a dos tablas diferentes.

Ejemplo: Seleccionar el número, nombre y tarifa de cualquier curso que tenga una tarifa mayor o igual al cinco por ciento del sueldo de cualquier maestro.

```
SELECT Materia.mnomateria, Materia.mnombre, Materia.mcostolab
 FROM Materia
 WHERE Materia.mcostolab >=
 (SELECT MIN(Academic.asueldo * .05)
  FROM Academic)
```

Resultado:

MNOMATERIA	MNOMBRE	MCOSTOLAB
0561	Microcomputadoras	500.00
0024	Circuitos Eléctricos	150.00

La lógica determina cual es el cinco por ciento del sueldo del maestro peor pagado y después que cursos están igual o por arriba de este valor.

2.7.2. Condiciones de búsqueda en subconsultas.

Una subconsulta forma parte de una condición de búsqueda, ya sea en la cláusula WHERE o en la cláusula HAVING. En las siguientes secciones se describe las condiciones de búsqueda en subconsultas.

2.7.2.1. Test de comparación.

Este test se ha utilizado en los ejemplos de subconsulta precedentes. El test de comparación empleado en subconsultas ofrece los operadores que ya conocemos: =, <>, <, <=, >, >= , que ya se habían utilizado en consultas simples.

La subconsulta que se especifique con este test debe producir una sola fila de resultado. Si la subconsulta produce mas de una fila SQL produce un error. Si la subconsulta no produce filas o devuelve un valor NULL el test de comparación devuelve un NULL.

2.7.2.2. Test de pertenencia a un conjunto (IN).

El uso de IN permite a la subconsulta devolver varios valores. También permite soluciones SQL alternativas a problemas que podrían resolverse utilizando la operación join.

IN copara un único valor de datos con una columna de valores producidos por la subconsulta y devuelve un VERDADERO si el valor coincide con uno de los valores de la columna.

Ejemplo: Reportar el nombre y el sueldo de cada maestro que trabaja en la carrera que su coordinación esta en el edificio A501.

```
SELECT Academic.anombre, Academic.asueldo;
FROM Academic;
WHERE Academic.anocarrera IN;
(SELECT Carrera.cnocarrera;
FROM Carrera;
WHERE Carrera.cedificio = "A501")
```

Resultado:

ANOMBRE	ASUELDO
Carlos Cuenca	7000.00
Armando Cruz	5000.00
Alberto Alvares	3000.00

La lógica de esta consulta es la consulta es que la subconsulta examina la tabla CARRERA para determinar que carreras tienen su coordinación en el edificio A501. Entonces la consulta principal examina la tabla ACADEMIC para localizar los maestros que trabajan en las carreras que resultaron de la subconsulta.

Como diferencia de las subconsultas anteriores, ésta devuelve múltiples filas, por lo que el WHERE de la consulta principal utiliza la palabra clave IN.

El formato del test IN funciona igual al que se describió para consultas simples, con la diferencia de que el conjunto de valores en una subconsulta esta dado por el resultado de la subconsulta, mientras que en una consulta simple los valores se dan explícitamente.

2.7.2.3. Test de existencia (EXISTS).

Este test comprueba si una subconsulta produce alguna fila de resultados.

Ejemplo: Lista los grupos que tengan profesor dado de alta en la tabla ACADEMIC.

```
SELECT Grupo.gnogrupo, Grupo.gnomaestro;
 FROM Grupo;
 WHERE EXISTS;
 (SELECT *;
  FROM Academic;
  WHERE Academic.anoemplead = Grupo.gnomaestro)
```

Conceptualmente, SQL procesa esta consulta recorriendo la tabla GRUPO y efectuando la consulta para cada GNOMAESTRO, si existe el número de maestro en la tabla ACADEMIC, es decir si el número de filas de la subconsulta es diferente de cero, entonces el test EXISTS es VERDADERO.

Como en todos los tipos de test de SQL, se puede invertir la lógica de EXISTS utilizando la forma NOT EXISTS, en este caso el TEST es VERDADERO cuando la subconsulta no produce filas y FALSO cuando si lo hace.

Cuando se utiliza EXISTS la subconsulta siempre tiene una referencia externa que enlaza la subconsulta a la fila que actualmente está siendo examinada por la consulta principal. En el ejemplo nótese que dentro de la subconsulta no se declaro la tabla GRUPO en la cláusula FROM, esto no es necesario ya que fue declarada en la sentencia exterior.

2.7.2.4. Test cuantificados (ANY y ALL)

Las palabras ANY y ALL pueden utilizarse con las condiciones WHERE que hacen referencias a subconsultas. Ambos test comparan un valor de dato con el conjunto de valores producidos por la subconsulta.

Sin embargo, cualquier consulta que se resuelva utilizando ANY y ALL puede resolverse siempre de alguna otra manera que no tenga esas palabras.

2.7.2.4.1.Test ANY.

El test ANY puede utilizarse con cualquiera de los operadores estándar de comparación. SQL utiliza el operador de comparación especificado para comparar el valor de test con cada valor de datos en la columna resultado de la subconsulta. La condición regresa VERDADERO si la condición se cumple con cualquiera de los valores devueltos por la subconsulta.

Ejemplo: Reportar el nombre y sueldo de cualquier maestro que no pertenezca a la carrera de computación.

```
SELECT Academic.anombre, Academic.asueldo
 FROM Academic
 WHERE Academic.anocarrera = ANY
 (SELECT Carrera.cnocarrera
 FROM Carrera
 WHERE Carrera.cnocarrera <> 32)
```

Resultado:

ANOMBRE	ASUELDO
Carlos Cuenca	7000.00
Armando Cruz	5000.00
Alberto Alvares	3000.00

La subconsulta devuelve los números de las carreras que son diferentes a 32, que es el número de carrera de Computación.

Se dice que algunos expertos han discutido que ANY y ALL, no deban formar parte de SQL, ya que, como se mencionó, cualquier consulta puede resolverse sin el uso de ANY.. Por ejemplo la consulta anterior se pudo haber resuelto de la siguiente manera:

```
SELECT Academic.anombre, Academic.asueldo
 FROM Academic
 WHERE Academic.anocarrera IN
 (SELECT Carrera.cnocarrera
 FROM Carrera
 WHERE Carrera.cnocarrera <> 32)
```

La palabra reservada SOME puede utilizarse en lugar de ANY, tienen el mismo significado, pero algunos productos SQL no soportan el uso de SOME.

2.7.2.4.2.Test ALL.

El test ALL, como ANY, puede utilizarse con cualquiera de los operadores de comparación estándar cuando una subconsulta devuelve varios valores. Cuando se usa ALL, la condición evaluada es VERDADERA, si la expresión es cierta para todos los valores regresados por la subconsulta.

Ejemplo: Seleccionar el número de materia, el nombre y el costo de laboratorio para aquellas materias donde el laboratorio tenga un costo menor, al cinco por ciento de todos los salarios de los maestros.

```
SELECT Materia.mnomateria, Materia.mnombre, Materia.mcostolab;
 FROM Materia;
 WHERE Materia.mcostolab < ALL;
 (SELECT (Academic.asueldo * .05) ;
 FROM Academic)
```

Resultado

MNOMATERIA	MNOMBRE	MCOSTOLAB
0076	Bases de Datos	100.00
0134	Sistemas Digitales	50.00
0119	Estructuras de Datos	0.00
0056	Estructuras Discretas	0.00
0559	Memorias y Periféricos	100.00
0028	Análisis Dinámico de Máquinas	100.00
0130	Elementos de Máquinas	50.00
0138	Dispositivos Electrónicos	90.00

Obsérvese que si el valor de MCOSTOLAB es menor al cinco por ciento de todos los sueldos, también es menor al cinco por ciento del menor de los sueldo. Esta observación da pie a la siguiente alternativa:

```
SELECT Materia.mnomateria, Materia.mnombre, Materia.mcostolab;
FROM Materia;
WHERE Materia.mcostolab < ;
(SELECT MIN(Academic.asueldo * .05) ;
FROM Academic)
```

Con lo que también se ve que la consulta es posible resolverla sin el uso de ALL.

Los siguientes comentarios son acerca de la lógica de ANY y ALL, cuando son utilizados con ciertos operadores de comparación:

- "= ANY" es equivalente a "IN". Pero "NOT = ANY" no significa lo mismo que "NOT IN". La explicación viene dada en el siguiente comentario.
- "NOT = ANY" tiene poca aplicación:

  Por ejemplo: WHERE COL1 NOT = ANY (2,3,4) Siempre será falsa,

  ó WHERE COL1 NOT = ANY (2). Es igual a WHERE COL1 <> 2.
- "= ALL" tiene poca aplicación:

  Por ejemplo: WHERE COL1 = ALL (2,3,4) Siempre será falsa,

  ó WHERE COL1 = ALL (2). Es igual a WHERE COL1 = 2.

Estos comentarios confirman aún mas que el uso de ANY y ALL debe de ser moderado. Desde un punto de vista general ambos test son superfluos, pero siempre es bueno tener alternativas en la solución de una consulta.

2.7.3. Subconsultas anidadas.

Las consultas vistas hasta ahora, han sido consultas con una subconsulta. Del mismo modo que se puede emplear una subconsulta dentro de una consulta principal, se puede utilizar una subconsulta dentro de otra subconsulta

Veremos que no hay nada nuevo en la sintaxis. Sin embargo la lógica de la consulta se hace más compleja.

Ejemplo: Visualizar el número y nombre del académico que sea coordinador de la carrera que tenga materias con 8 créditos:

```
SELECT Academic.anoemplead, Academic.anombre
 FROM Academic
 WHERE Academic.anoemplead IN
 (SELECT Carrera.ccoord
  FROM Carrera
  WHERE Carrera.cnocarrera IN
  (SELECT Materia.mnocarrera
   FROM Materia
   WHERE Materia.mcred = 8)
 )
```

Resultado:

ANOEMPLEAD	ANOMBRE
75521245	Armando Cruz
75656566	Pedro Benitez

El sistema ejecuta la subconsulta mas interna, en este caso devuelve los números de las carreras con materias de 8 créditos. Por lo que la consulta se reduce a:

```
SELECT Academic.anoemplead, Academic.anombre
 FROM Academic
 WHERE Academic.anoemplead IN
 (SELECT Carrera.ccoord
  FROM Carrera
  WHERE Carrera.cnocarrera IN
  (32,38)
 )
```

Este resultado todavía contiene una subconsulta. La subconsulta es evaluada y regresa los números de los coordinadores de las carreras 32,38, y por ultimo con estos números se produce el resultado que se mostró.

La misma técnica utilizada puede servir para construir consultas de mas de tres niveles. Al incrementar el número de niveles, la consulta consume mucho más tiempo. La consulta también es mas difícil de leer, comprender y mantener al aumentar el número de niveles en la subconsulta.

Muchos productos SQL restringen el número de niveles de subconsultas a un número relativamente pequeño (3 ó 4) argumentando que cualquier consulta, por muy difícil que sea no requiere más de tres niveles, sin que se pueda resolver por un método alternativo, por ejemplo utilizando la operación join.

2.7.4. Subconsultas en la cláusula HAVING.

Aunque la mayoría de las subconsultas se encuentran en la cláusula WHERE, también pueden utilizarse en la cláusula HAVING, funcionando como parte de la selección de grupos. La sintaxis de la subconsulta es la misma en ambos casos.

Ejemplo: Reportar el número de carrera y el costo promedio de laboratorios de la carrera, para las carreras con un promedio del costo de laboratorio por abajo del promedio de todos los laboratorios.

```
SELECT Mnocarrera, AVG(Mcostolab)
 FROM Materia
 GROUP BY Mnocarrera
 HAVING AVG(Mcostolab) <
 (SELECT AVG(Mcostolab)
 FROM Materia)
```

Resultado:

MNOCARRERA	AVG_MCOSTO
38	75.00

Dado que deseamos ver la tarifa promedio por departamento, es necesario establecer grupos por carrera. Como sólo queremos visualizar los grupos donde la media es menor a la media total, se requiere que la cláusula HAVING compare las medias de grupo con la total. La subconsulta determina la media total y la sustituye en la cláusula HAVING., quedando así una consulta simple.

2.8. Actualización de la base de datos.

SQL es un lenguaje completo de manipulación de datos que se utiliza no solamente para realizar consultas, sino que también es usado para actualizar los datos de la base de datos. Comparando la complejidad de las sentencias utilizadas para la actualización de la base de datos con la sentencia SELECT, las primeras son extremadamente sencillas.

Esta sección se centra en las sentencias de actualización de datos, las sentencias que se utilizan para este propósito son las siguientes:

- La sentencia INSERT se usa para adicionar filas en una tabla. La tabla puede estar vacía o contener información.
- La sentencia UPDATE es utilizada para modificar los datos existentes en las tablas.
- La sentencia DELETE que elimina una o mas filas de una tabla.

2.9. Adición de datos.

Una nueva fila de datos se añade a una base de datos relacional cuando una nueva entidad representada por una fila "aparece en el mundo real". Por ejemplo:

- Cuando se contrata un nuevo maestro.
- Cuando un nuevo alumno entra a la Universidad.
- Cuando se sobrepasa el cupo de alumnos en un grupo y se da de alta un nuevo grupo.

En cada caso se añade una nueva fila para mantener la base de datos como un modelo preciso de los que sucede en el mundo real. La unidad mas pequeña que puede adicionarse a una base de datos relacional es una fila.

La sentencia INSERT puede presentar dos formas. La primera forma se utiliza para insertar una sola fila en una tabla. La segunda forma permite la inserción de múltiples registros, que ya existan en alguna otra tabla.

Todos los productos SQL añaden utilerías externas para la carga masiva de información. Estas utilerías no forman parte del lenguaje SQL, pero sirven para cargar inicialmente la base de datos o transferir información desde otro sistema informático.

2.9.1. La sentencia INSERT de una fila.

La sentencia INSERT de una fila , añade una nueva fila a una tabla.

Sintaxis simplificada:

INSERT INTO nombre de tabla [nombres de columna]
VALUES (constante / NULL)

La cláusula INTO indica la tabla que contendrá la nueva fila, la cláusula VALUES especifica los valores que contendrán las columnas de la nueva fila.

Ejemplo: Supongamos que la carrera de Computación ofrece una nueva materia con el nombre de "PROGRAMACIÓN CON SQL" con la clave 0077. El curso tiene 6 créditos y tiene un laboratorio con valor de $100. Insertar el nuevo registro en la tabla correspondiente.

```
INSERT INTO Materia
VALUES ("0177", "Programación con SQL", 6, 100.00, 32)
```

El sistema mostrará un mensaje que indica que se ha insertado una fila satisfactoriamente ó en su defecto que se ha rechazado y las razones del rechazo. Suponiendo que la inserción del ejemplo sea un éxito, se puede verificar con una sentencia SELECT:

```
SELECT * FROM Materia WHERE Mnomateria = "0177"
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB	MNOCARRERA
0177	Programación con SQL	6	100.00	32

En el ejemplo se omitió la lista de columnas, cuando esto pasa SQL genera automáticamente una lista formada por todas las columnas de la tabla, es secuencia de izquierda a derecha. Esta es la misma secuencia que se genera cuando se utiliza la sentencia SELECT *. El mismo ejemplo también se pudo haber escrito de la siguiente manera:

```
INSERT INTO Materia (Mnomateria, Mnombre, Mcred, Mcostolab, Mnocarrera)
VALUES ("0177", "Programación con SQL", 6, 100.00, 32)
```

y se obtendría el mismo resultado.

La especificación de las columnas, permite cambiar la secuencia de éstas, siempre y cuando también se den los valores en la misma secuencia.

El nombre de la tabla que a la que se hace referencia después de INSERT INTO debe de ser una tabla que ya exista.

La cláusula VALUES va seguida los valores que van a ser colocados en las columnas de la nueva fila. El ejemplo muestra que los valores que se dan en la cláusula VALUES se insertan, haciendo corresponder los datos con las columnas.

El tipo de dato de cada valor debe de ser compatible con el tipo de dato correspondiente de la columna, en caso contrario se produce un error. Los valores de tipo caracter se deben de encerrar entre apóstrofes (') o comillas (").

Dado que el modelo relacional no hace suposiciones acerca del orden de las filas dentro de una tabla, no será necesario especificar donde queremos colocar la fila

El nombre de tabla especificado en la sentencia INSERT es normalmente un nombre de tabla no cualificado, que especifica una tabla propia. Para insertar datos en una tabla propiedad de otro usuario se puede especificar un nombre de tabla cualificado. Recordemos, del capítulo 1 que la sintaxis para un nombre de tabla cualificado es :

```
usuario.tabla[.columna]
```

Los interesantes temas de seguridad e integridad referencial serán tratados en un capítulo posterior.

2.9.2. Inserción de valores NULL.

Pueden darse casos en los que al dar de alta una nueva fila algunos de los datos sean desconocidos, pero se desea dar de alta los valores conocidos.

SQL asigna automáticamente un valor NULL a cualquier columna cuyo nombre falte en la lista de columnas en la sentencia INSERT.

Ejemplo: Insertar una fila en la tabla MATERIA. La nueva materia es "SQL DINÁMICO", la clave del curso es 0178 y es dada de alta en la carrera de Computación. Se supone que los créditos y el costo de laboratorio son desconocidos.

```
INSERT INTO CURSO
VALUES ("0178", "SQL Dinámico", NULL, NULL, 32)
```

Este ejemplo no especifica los nombres de columna, por lo que la cláusula VALUES debe especificar los valores de columna en la secuencia de izquierda a derecha. Los valores que no son conocidos, en este caso MCRED y MCOSTOLAB, se especifican con la palabra clave NULL. La palabra clave NULL puede utilizarse tanto en columnas numéricas como de tipo caracter e indica un valor desconocido.

Al construir una nueva tabla se puede declarar una columna para que no acepte valores nulos, en este caso se le tendría que dar un valor, por ejemplo cero para columnas de tipo de dato numérico y blancos para columnas de tipo caracter. (Ver capítulo 4, para la creación de tablas).

Cualquier columna no especificada en la sentencia INSERT se le asigna un valor nulo o un valor por defecto.

Ejemplo: Repetir el ejemplo anterior, especificando las columnas.

```
INSERT INTO Materia (Mnomateria, Mnombre, Mnocarrera)
VALUES ("0178", "SQL Dinámico", 32)
```

2.9.3. La sentencia INSERT multifila.

La segunda forma de la sentencia INSERT permite la inserción de múltiples registros, que ya existan en alguna otra tabla. En esta forma , los valores de los datos para las nuevas filas no son especificados explícitamente dentro del texto de la sentencia. En su lugar , la fuente de información para las nuevas filas es una consulta especificada dentro de la sentencia INSERT.

Sintaxis simplificada:

INSERT INTO nombre de tabla [(nombres de columnas)] consulta

Para el siguiente ejemplo supóngase que se construyó una nueva tabla llamada CMATERIA que tendrá las materias de la carrera de Computación, que tiene los campos CNOMATERIA, CNOMBRE, CCRED, CCOSTOLAB, y tienen las mismas definiciones respectivas a las columnas en la tabla MATERIA que concentra las materias de todas las carreras.

Ejemplo: Copiar el numero de curso, nombre, créditos y costo de laboratorio para los cursos de Computación de la tabla MATERIA en la tabla CMATERIA.

```
INSERT INTO Cmateria
    SELECT Mnomateria, Mnombre, Mcred, Mcostolab
    FROM Materia
    WHERE Mnocarrera = 32
```

Al igual que la primera forma de INSERT, ésta comienza con una cláusula INSERT INTO, y a diferencia de la primera, no tiene cláusula VALUES. En lugar de esto tiene la sentencia SELECT que sigue las reglas estudiadas anteriormente. Aunque existen algunas restricciones:

- La consulta no puede tener la cláusula ORDER BY. No tiene caso ordenar los resultados de la consulta so van a ser insertados en una tabla que es desordenada.
- La consulta no puede tener la operación UNION.

- La consulta no puede hacer referencia a la misma tabla donde se van a insertar los datos. Con esto se descarta la posibilidad de insertar parte de una tabla en si misma.

- Los resultados de la consulta deben contener el mismo numero de columnas que la lista de la cláusula INSERT INTO (o de la lista completa de columnas si se omite).

- Los tipos de datos de las columnas del resultado de la consulta deben de ser compatibles columna a columna con los especificados en la cláusula INSERT INTO.

El ejemplo no nombra explícitamente, las columnas de la tabla receptora, pero el resultado sería el mismo si se hubieran nombrado:

```
INSERT INTO Cmateria (Cnomateria, Cnombre, Ccred, Ccostolab)
    SELECT Mnomateria, Mnombre, Mcred, Mcostolab
    FROM Materia
    WHERE Mnocarrera = 32
```

2.10. Supresión de datos.

Así como una nueva fila de datos se añade a una base de datos relacional cuando una nueva entidad representada por una fila "aparece en el mundo real", entonces una fila de datos se suprime cuando la entidad representada por la fila "desaparece del mundo real. Por ejemplo:

- Cuando se da de baja un maestro.

- Cuando un alumno causa baja en la Universidad.

- Cuando se actualiza el plan de estudios y se dan de baja algunas materias.

En cada caso la fila se suprime para mantener la base de datos como un modelo preciso de los que sucede en el mundo real. La unidad mas pequeña que puede suprimirse en una base de datos relacional es una fila.

2.10.1. La sentencia DELETE.

La sentencia DELETE se usa para eliminar una fila o un grupo de filas de una tabla. La cláusula FROM especifica la tabla de donde se borraran las filas. La cláusula WHERE especifica que filas serán borradas.

Sintaxis:

DELETE FROM nombre de la tabla
WHERE condición de búsqueda

Ejemplo: Borrar todos los registros de la tabla MATERIA que su número de curso comiencen con "00"

```
DELETE
FROM Materia
WHERE Mnomateria LIKE "00%"
```

Como comentario a lo que se verá en el capítulo 3 de integridad referencial: si la tabla MATERIA tiene ligas con otras tablas, por ejemplo con los grupos, SQL no permitirá dar de baja los registros de MATERIA, sin antes dar de baja los grupos donde se imparte esta clase.

La cláusula WHERE puede utilizarse para identificar muchos registros, y por lo tanto hay que poner mucha atención en cuanto a la condición que se codifique, para evitar que borren registros de forma errónea. La cláusula WHERE se codifica igual que en la sentencia SELECT.

Si se desea borrar un solo registro hay que asegurarse que la cláusula WHERE identifique al registro por su llave primaria o por alguna columna que identifique de manera única la fila.

La cláusula WHERE es opcional. Sin embargo si no se incluye, todos los registros de la tabla serán borrados.

Ejemplo: Borrar las filas de la tabla CMATERIA.

```
DELETE
FROM Cmateria
```

Debido al gran daño que puede producir una sentencia DELETE como esta, es importante especificar siempre una condición de búsqueda y tener cuidado con seleccionar las filas que realmente se desean.

2.11. Modificación de datos.

Los valores de los datos son modificados en una base de datos cuando se producen cambios en el mundo real. Por ejemplo:

- Cuando un maestro cambia su domicilio o es ascendido.
- Cuando un alumno cambia de carrera o domicilio.
- Cuando se actualiza el plan de estudios y se cambia el nombre de algunas materias.

La unidad mas pequeña que puede modificarse en una base de datos es una columna de una fila.

2.11.1. La sentencia UPDATE.

La sentencia UPDATE puede usarse para cambiar cualquier valor en un tabla. Modifica los valores de una o mas columnas en las filas seleccionadas de una tabla única.

Sintaxis:

UPDATE nombre de tabla SET nombre de columna = expresión [,nombre de columna = expresión[,...]]
WHERE condición de búsqueda

La cláusula SET identifica las columnas que van a ser modificadas. La cláusula WHERE identifica las filas que se van a modificar.

Ejemplo: Cambiar la tarifa del laboratorio en la materia 0076 a $150.00.

```
UPDATE Materia
SET Mcostolab = 150.00
WHERE Mnomateria = 0076
```

La cláusula SET Mcostolab = 150.00 significa que MCOSTOLAB se fijará en 150.00 en las filas seleccionadas por la cláusula WHERE. El valor que se especifique en SET puede ser una expresión, puede ser también la palabra clave NULL para especificar valores desconocidos.

La cláusula SET puede referenciar muchas columnas, separándolas por comas.

WHERE puede especificar cualquier condición de búsqueda. Los cambios que se efectúen por la cláusula SET , solo afectarán los registros que cumplan la condición especificada en la cláusula WHERE. La cláusula WHERE se codifica igual que en la sentencia SELECT

La cláusula WHERE es opcional. Sin embargo la ausencia de ella hará que en la tabla se cambie cada registro.

Ejemplo: Ponerle 9 créditos a todas las materias de la tabla CMATERIA.

```
UPDATE Cmateria
SET CCRED = 9
```

Debido al gran daño que puede producir una sentencia UPDATE como esta, es importante especificar siempre una condición de búsqueda y tener cuidado con seleccionar las filas que realmente se desean.

3.INTEGRIDAD DE DATOS Y PROCESAMIENTO DE TRANSACCIONES.

3.1. Integridad de datos.

El termino integridad de datos se refiere a medidas de seguridad usados para mantener correctos los datos en la base de datos.

Cuando los datos de una base de datos son modificados mediante las sentencias INSERT, DELETE o UPDATE, los datos almacenados pueden ser inconsistentes de muchas maneras, por ejemplo:

- Se pueden adicionar datos que no son validos, por ejemplo un alumno dado de alta en un grupo que no existe.
- Al modificar un valor pueden darse valores inexistentes como asignar a un grupo un número de maestro inexistente.
- Los cambios que se hayan realizado pueden perderse total o parcialmente en una falla de energía eléctrica.

Una de las funciones más importantes de un DBMS relacional es preservar lo mas que se pueda la integridad de sus datos almacenados.

Para preservar la consistencia de los datos almacenados, DBMS relacional impone restricciones para la inserción, supresión y actualización de datos. Las restricciones que suelen encontrarse en los diferentes productos SQL se describen en las siguientes secciones.

El DBMS proporciona recursos que refuerzan la integridad de los datos sin necesidad de programación extra por parte del usuario de la base. Por ejemplo al crear la base de datos pueden crearse criterios de integridad de tal manera que el DBMS efectué automáticamente comprobaciones de validez y consistencia

3.1.1. Validación de datos.

Quien maneja la base puede especificar en el esquema ciertas validaciones ademas de las proporcionadas por el sistema. En el esquema se representa una visión organizacional de la base y consta de definiciones de datos y sus relaciones. Algunos ejemplos de las validaciones de datos:

- *Validación del tipo de datos.* Los valores de cada elemento de entrada debe de coincidir con sus características descritas en el esquema (caracter, entero, flotante, etc.), de otra manera el DBMS mandará el mensaje de error apropiado.

- *Validación del valor de los datos.* El contenido de un campo de entrada puede validarse para cierto rango de valores. Por ejemplo la fecha de alta de un maestro no puede ser mayor a la fecha en que se da de alta el registro. Otro ejemplo es que el numero de créditos de una materia no debe ser menor a 5 créditos ni mayor a 12 créditos.

- *Validación de los valores de las claves primarias y foráneas.* En la especificación de un esquema , a cada tabla se le designa una llave primaria para identificar de manera única cada fila de la tabla.

  El DBMS valida que el valor de la clave primaria sea único y no sea nulo. Mientras se introduce cada fila en la base de datos, el DBMS valida el valor de la clave primaria, si tiene un valor ya existente o se intenta dejar sin información, el DBMS genera un mensaje de error.

  Cuando se declaran la llave foránea, el usuario puede especificar si se permiten valores duplicados en tal clave. Si no se permiten el DBMS realiza el mismo chequeo de unicidad como lo hace para la llave primaria.

- *Validación de datos requeridos.* Algunas columnas en una base de datos deben de contener un valor válido, es decir no se permiten valores NULL. Por ejemplo en la tabla MATERIA se debe de identificar siempre a que carrera pertenece cada materia por lo que la columna no debe aceptar que se quede sin definir dicha columna.

3.1.2. Integridad de entidad.

Una fila en una tabla, corresponde generalmente, a un ejemplo de un tipo de entidad que la base de datos está modelando. Una entidad puede ser descrita como cualquier objeto identificable. Por ejemplo en la tabla MATERIA cada registro corresponde a una materia concreta ofrecida por la Escuela Nacional de Estudios Profesionales.

Debido a cada curso es identificable de manera única en el mundo real, es aconsejable que cada fila correspondiente en la tabla MATERIA sea identificable por alguna columna o grupo de columnas de manera única. A esto se le llama integridad de entidad y es el propósito de definir una clave primaria.

Considérese la ambigüedad que se produce cuando una tabla contiene dos o mas registros que no pueden diferenciarse uno de otro. Existiría una perdida de integridad de entidad debido a que la correspondencia, uno a uno, entre la entidad "materia" y su correspondiente fila se destruiría.

Identificando la columna MNOMATERIA como la clave primaria, se establece que un número de curso puede identificar una sola materia. Especificar MNOMATERIA como clave primaria hace que el sistema rechace cualquier valor duplicado o NULL en ésta columna. Por lo que cualquier sentencia que contenga la cláusula WHERE MNOMATERIA = *valor* solo seleccionará una fila.

También a veces es necesario exigir valores únicos en una columna que no es llave primaria. Por ejemplo pude existir la restricción de que el edificio de la coordinación no pueda ser utilizado para dos carreras a la vez. Entonces sería recomendable declarar la columna CEDIFICIO con restricción de unicidad. De este modo cualquier intento que viole esta restricción producirá un error.

3.1.3. Integridad referencial.

Antes de explicar la integridad referencial son necesarios algunos conceptos:

- *Tabla padre:* Una tabla que se referencia mediante alguna llave foránea. La tabla CARRERA es una tabla porque se referencia mediante la clave foránea MNOCARRERA de la tabla MATERIA. Una clave primaria, como CNOCARRERA en la tabla CARRERA debe ser especificada en la tabla padre. Esto es debido a que cualquier valor en la clave foránea (MNOCARRERA) debe coincidir con algún valor en la llave primaria de la tabla padre.

- *Tabla hijo.* Una tabla que contiene una clave foránea. La tabla MATERIA que contiene MNOMATERIA como llave foránea es una tabla dependiente. Depende de CARRERA que es la tabla padre. Una fila de una tabla dependiente no puede existir a menos que el valor de su clave foránea exista en la tabla padre.

La idea básica de la integridad referencial es asegurar que no existan filas en tablas hijo sin que exista una referencia en una tabla padre. Existen tipos de actualizaciones de bases de datos que pueden corromper la integridad referencial de las relaciones padre/hijo en una base de datos. Estas situaciones son:

- *Inserción de una nueva fila en la tabla hijo.* Cuando se inserta una nueva fila en la tabla hijo, su valor de clave foránea debe coincidir con algún valor en la llave primaria de la tabla padre. Si el valor de la clave foránea no coincide con la llave primaria, la inserción de la fila corromperá la información ya que habrá un hijo sin su padre. Si se desea insertar una fila en una tabla padre nunca representará un problema, ya que si no hace referencia a una llave foránea de una tabla hijo, solamente se convertirá en un padre sin hijos.

  En la base de datos ejemplo, al dar de alta una nueva materia en la tabla MATERIA, la materia debe pertenecer a una carrera ya existente en la tabla CARRERA. En caso contrario la fila queda sin su padre.

- *Actualización de una llave foránea en una tabla hijo.* Aquí se presenta una variación del problema anterior, si se desea actualizar el valor de la llave foránea, ésta debe de coincidir con un valor de llave primaria en la tabla padre.

Si se modifica una fila en la tabla MATERIA se debe verificar que el nuevo valor de la columna MNOCARRERA coincida con algún valor de la clave primaria CNOCARRERA de la tabla CARRERA.

- *Supresión de una fila en una tabla padre.* Si una fila en una tabla padre que tiene uno o mas hijos se suprime, las filas hijo quedarán huérfana. Los valores de llave foránea en éstas filas ya no corresponderán con ningún valor en la tabla padre. Por otro lado, suprimir una fila en la tabla hijo no representara problema, simplemente el padre de ésta fila tendrá un hijo menos.

  Al eliminar una fila de la tabla CARRERA las filas de la tabla hijo, por ejemplo la tabla MATERIA, que hacían referencia a la fila que se borro quedarán huérfanas.

- *Actualización de la llave primaria en una fila padre.* Este es una variación del problema anterior, ya que si se modifica una llave primaria en una tabla padre todos los hijos actuales de esta fila quedarán sin referencia al padre.

  Al modificar la clave primaria en una fila de la tabla CARRERA se pueden quedar huérfanas todos los hijos actuales de esta fila quedarán sin referencia.

El DBMS se ocupa de cada uno de estos problemas. El primer problema se maneja comprobando los valores de las columnas que son llaves foráneas antes de permitir la inserción, Si no coinciden con un valor de llave primaria, se genera un mensaje de error y la sentencia INSERT se rechaza.

El segundo problema se trata de manera similar comprobando el valor de la clave primaria antes de permitir la actualización. Si el valor de la columna de llave foránea no coincide con un valor de llave primaria, se genera un mensaje de error y la sentencia UPDATE se rechaza.

El problema de suprimir una fila de una tabla padre se maneja impidiendo que la fila sea suprimida en caso de que tenga hijos. Hasta que los hijos sean asignados a otro padre o sean eliminados la sentencia DELETE podrá ser válida.

El problema de actualización de la clave primaria en la tabla padre. También se maneja por prohibición. Antes de efectuar la modificación, el DBMS efectúa comprobaciones para asegurarse que no haya filas hijo que tengan valores de llave foránea que hagan referencia al valor que se actualiza. Si existen tales hijos la sentencia UPDATE es rechazada.

3.1.4. Disparadores (triggers)

Muchas de las cuestiones de integridad de datos tienen que ver con las reglas y procedimientos de una organización. Por ejemplo en la base de datos ejemplo, podrían existir políticas como:

- No se permite que algún maestro tenga un sueldo mayor a cualquiera de los coordinadores de carrera
- No se permite que existan cuatro grupos de la misma materia.
- Un coordinador solo puede impartir dos clases.

Estas políticas de la escuela caen fuera del ámbito de SQL. El DBMS toma la responsabilidad de almacenar y organizar los datos y asegurar su integridad básica, pero implementar las reglas de una empresa u organización es responsabilidad de los programas de aplicación que acceden a las bases de datos.

Dejar la responsabilidad de implementar las reglas de la empresa en manos de los programadores de aplicaciones tiene varias desventajas:

- *Duplicación de esfuerzo.* Si varios programas realizan actualizaciones en una tabla, cada uno de ellos deberá de contener el código de las reglas implicadas con esta tabla.
- *Problemas de mantenimiento.* Cuando hay una modificación en la regla se deben modificar todos los programas que contienen el código de la regla que se esta modificando.
- *Falta de consistencia.* Si varios programas realizan actualizaciones en una tabla, y estos son realizados por varios programadores la regla puede omitirse o variar en alguno de los programas de aplicación.

El DBMS Sybase fue el primero en introducir el concepto de *trigger (disparador)* para la inclusión de reglas en una base de datos relacional.

3.1.4.1. ¿Qué es un disparador?

Un disparador es un procedimiento especial que entra en acción cuando se cambia el contenido de una tabla. Las sentencias que pueden disparar el procedimiento son DELETE, INSERT o UPDATE.

La principal ventaja de los disparadores es que son automáticos. La acción que ejecuta el disparador es especificada mediante sentencias SQL. El disparador es ejecutado una vez por cada sentencia disparadora, y se dispara inmediatamente después de que a la sentencia es completada. El procedimiento del disparador y la sentencia son tratados como una sola transacción que puede ser cancelada en caso de que se detecte un error.

Cuando se crea un disparador, se especifica la tabla y la sentencia con que se disparará, después se especifica la acción que el disparador tomará.

Para comprender mejor como funciona un disparador veamos el siguiente ejemplo simple:

Ejemplo: Al dar de alta un nuevo maestro mandar un mensaje recordando que un maestro no puede tener un sueldo mayor a cualquiera de los académicos.

```
CREATE TRIGGER Altamaestro
ON Academic
FOR INSERT, UPDATE
AS
PRINT "Recuerda que el sueldo no debe exceder el sueldo de algún coordinador"
```

La cláusula CREATE crea el disparador y le da un nombre, en este caso ALTAMAESTRO. La cláusula ON indica la tabla que activa al disparador. La cláusula FOR especifica cuales sentencias de modificación activaran el disparador. Para el ejemplo un INSERT o UPDATE activara el disparador.

El dialecto SQL de Sybase, llamado Transact-SQL, se extiende para el soporte de los disparadores como son test IF/THEN/ELSE, llamadas de procedimientos y sentencias PRINT que visualizan mensajes para el usuario.

Las sentencias que puede ejecutar un disparador pueden ser cualquiera de las que ya se conocen, inclusive pueden incluirse sentencias INSERT, UPDATE y DELETE.

3.1.4.2. Disparadores e integridad referencial.

Los disparadores proporcionan un medio alternativo de implementar restricciones de integridad referencial.

Dos tablas especiales son usadas en las sentencias de los disparadores. la tabla *deleted* y la tabla *inserted.* Estas tablas no pueden ser modificadas, y solo pueden utilizarse en sentencias SELECT. Las dos tablas tienen los mismos nombres de columna que la tabla afectada.

Cuando una fila es afectada por las sentencias DELETE o UPDATE, las lineas son removidas de la tabla afectada y son adicionadas en la tabla *deleted.* La tabla *deleted* y la tabla ordinaria no tienen filas en común.

La tabla *inserted* contiene copias de las filas afectadas por sentencias INSERT y UPDATE, durante la ejecución de estas sentencias las nuevas filas son adicionadas en la tabla original y en la tabla *inserted.*

Como se ve una sentencia UPDATE borra la fila afectada y posteriormente inserta una nueva fila ya modificada, por lo que afecta a las dos tablas especiales: *inserted* y *deleted.*

Ejemplo: Crear un disparador para la tabla MATERIA para que al insertar o modificar una materia verifique que la carrera exista.

```
CREATE TRIGGER Act_Materia
ON Materia
FOR INSERT, UPDATE
AS IF ((COUNT (*)
        FROM Carrera, Inserted
        WHERE Carrera.Cnocarrera = Inserted.Mnocarrera) = 0)
        BEGIN
                PRINT "La oficina no existe"
                ROLLBACK TRANSACCIÓN
        END
```

En este ejemplo se busca que el numero de carrera insertado en la tabla *inserted* sea igual al numero de carrera de la tabla CARRERA, en el caso de que no se encuentren filas coincidentes, la función COUNT(*), que cuenta las filas seleccionadas, regresará cero y cancelara la transacción mediante la sentencia ROLLBACK TRANSACCIÓN. El manejo de transacciones se estudiara en el siguiente capitulo.

Los disparadores pueden servir para cuidar la integridad referencial de las siguientes maneras:

- Al insertar una fila que contenga llaves foráneas, validar que para cada una de ellas exista su correspondiente llave primaria en otra tabla, y cancelar la inserción en caso de que no se cumpla esta condición.

- Al actualizar una fila con llaves foráneas. Aquí se presenta una variación del problema anterior, si se desea actualizar el valor de la llave foránea, ésta debe de coincidir con un valor de llave primaria en la tabla padre, en caso contrario, el disparador cancela la operación de modificación.

- Al suprimir una fila en una tabla padre, se puede ordenar al trigger varias acciones, por ejemplo mandar un mensaje para que borre los hijos relacionados, o directamente borrarlos. También se puede ordenar por medio del disparador que se reasignen a un nuevo padre.

- Al actualizar la llave primaria en una fila padre, se presenta el mismo problema anterior, ya que si se modifica una llave primaria en una tabla padre todos los hijos actuales de esta fila quedarán sin referencia al padre. Se puede especificar en el disparador cualquiera de las acciones que se sugirieron en el anterior problema.

El presente trabajo no profundiza demasiado sobre el tema de los disparadores ya que actualmente solamente Sybase y SQL Server disponen de éstos , aunque varios vendedores han externado planes para añadir disparadores a sus productos.

La principal ventaja de los disparadores es que las *reglas comerciales* o también llamadas *reglas del negocio* pueden almacenarse en las bases de datos y ser forzadas consistentemente en cada actualización en la base de datos. El uso de disparadores reduce la complejidad de los programas, aunque también aumenta la complejidad de la base de datos.

3.2. Procesamiento de transacciones.

En esta sección se vera la manera de conservar la integridad de los datos a través del procesamiento de transacciones. Antes de explicar el procesamiento de transacciones es necesario introducir el concepto de transacción, para posteriormente presentar dos nuevas sentencias de SQL COMMIT y ROLLBACK para forzar que varias sentencias de actualización sean tratadas como una transacción.

3.2.1. Unidad Lógica de trabajo

A una transacción suele llamársele también Unidad Lógica de trabajo. Una transacción es una secuencia de operaciones que modifican los datos, pasándolos de un estado a otro. Esta definición es la que se utiliza en este capítulo y no debe generar confusiones con el concepto de transacción en los sistemas de tiempo real donde una transacción es un acceso a la base de datos.

Una transacción es una secuencia de actualizaciones sobre las tablas, que se debe realizar completa o no realizarse en absoluto, pero nunca a medias, porque así los datos quedarían en forma inconsistente.

Para ejemplificar el proceso de una transacción se presenta un ejemplo. Supóngase que al eliminar de la tabla MATERIA una fila que contiene el número de curso 0076 que equivale a la materia de Bases de Datos. Al mismo tiempo si la materia es dada de baja se debe dar de baja su correspondiente grupo en la tabla GRUPO. Al dar de baja el grupo también hay que dar de baja a todos los alumnos inscritos en él, y que están dados de alta en la tabla INSCRIPC.

Ejemplo: Dar de baja la materia 0076 de la tabla MATERIA. Dar de baja sus grupos y los alumnos inscritos a éstos.

```
DELETE
FROM Inscripc
WHERE Inogrupo =
      (SELECT Gnogrupo
      WHERE Gnomateria = "0076")

DELETE
FROM Grupo
WHERE Gnomateria = "0076"

DELETE
FROM Materia
WHERE Mnomateria = "0076"
```

La primera sentencia borra a los alumnos inscritos en los grupos que dan la clase 0076, la cláusula WHERE utiliza una subconsulta de la tabla GRUPO para obtener el grupo donde se imparte la materia. La segunda sentencia da de baja los grupos donde se imparte la materia. La tercera sentencia DELETE se encarga de dar de baja la materia con número 0076.

Puede ser que se hayan definido disparadores para las tablas del ejemplo donde se restrinja borrar sus filas si aún tienen hijos. Por ejemplo evitar borrar filas de la tabla MATERIA si la materia tiene grupos hijos en la tabla GRUPO, o el disparador también puede evitar borrar filas de la tabla GRUPO si tiene alumnos inscritos en la tabla INSCRIPC.

El orden de borrado del ejemplo es borrar primero los hijos y por último la información de la tabla padre. Toda las sentencias anteriores se pueden declarar en un disparador que se active al tratar de borrar una fila en la tabla MATERIA.

En el ejemplo se muestran tres sentencias DELETE independientes. Sin embargo desde el punto de vista del usuario, estos cambios constituyen una única unidad de trabajo lógica: borra la materia 0076 de la tabla MATERIA y todas las referencias que haya a esta materia en otras tablas. Para SQL son varias sentencias, puesto que necesita cambiar varias tablas debido a las restricciones de borrado para la tabla MATERIA. En este mismo capítulo se verá como enlazar varias sentencias para formar una transacción.

Tal vez quedaría la duda de porque se tienen que enlazar las sentencias DELETE del ejemplo, ¿porque no ejecutar simplemente, cada sentencia?. La respuesta esta con problemas que quedan fuera del control del usuario.

Supóngase que existe una falla en la corriente eléctrica en el momento que se ejecuta la cualquiera de las tres sentencias DELETE ¿ qué pasaría en la base de datos ?. Si esto ocurre no se puede saber que fue lo que ocurrió en la base de datos. Se podría revisar el contenido de las tablas, sin embargo esto no es practico, especialmente si las tablas son muy grandes.

Lo que se desea lograr con al procesar las tres sentencias como una unidad de trabajo lógica, es lograr una situación de todo o nada. Es decir que las tres sentencias se ejecuten o empezar de nuevo. Esto supone un esfuerzo extra, pero es el precio de la integridad de la base de datos. Esta situación de todo o nada, es lo que proporciona el concepto de transacción o Unidad Lógica de Trabajo.

El DBMS es responsable de mantener el compromiso de todo o nada, incluso si el programa aborta, o se produce una falla de hardware a mitad de la transacción, de tal manera que en la base de datos nunca existirá una transacción parcial.

Si las sentencias son ejecutadas en forma interactiva *(on-line)* en la terminal, cada sentencia es tratada como una transacción. Por lo general los productos SQL, pueden trabajar en dos modos cuando se trabaja en forma interactiva, una es que el usuario tenga que escribir COMMIT cada vez que quiere que sus actualizaciones se reflejen en la base de datos, o que el sistema de manera automática tome por default un COMMIT después de cada sentencia de actualización.

Cuando las sentencias de actualización se ejecutan en forma incorporada dentro de un programa, mientras no se le comunique otra cosa, el DBMS considera que todas las sentencias de actualización forman una transacción . Cuando el DBMS recibe del programa la primera petición de actualización, inicia una transacción. Cuando el programa termina, el DBMS termina la transacción . Si el programa termina correctamente, todos los cambios realizados en los datos por las actualizaciones, se ejecutan y permanecen. Si el programa termina porque ha tenido alguna falla, el DBMS detecta la situación, y deshace todas las actualizaciones realizadas en la transacción hasta ese momento, con lo que los datos quedan como estaban antes de comenzar la ejecución del programa.

3.2.2. Sentencias COMMIT y ROLLBACK.

Al principio de cada programa de aplicación se establece automáticamente un inicio de transacción y, de la misma manera al finalizar el programa se establece automáticamente un fin de transacción, de tal manera que todas las sentencias de actualización que realiza el programa son tratadas como una Unidad Lógica de Trabajo.

Si se desea, el usuario puede dividir las sentencias de actualización de su programa en varias transacciones mediante las sentencias SQL, COMMIT WORK y ROLLBACK WORK. En circunstancias normales, una transacción concluye al ejecutarse:

- Una sentencia COMMIT WORK, que informa al sistema de que se aceptan todos los cambios hechos dentro de la transacción y de que deberían aplicarse a la base de datos.
- Una sentencia ROLLBACK WORK, informa al sistema de que se ha localizado una situación inaceptable y que deben ignorarse todos los cambios realizados desde el principio de la transacción. El DBMS restaura la base de datos a como estaba antes de que la transacción comenzara.

Ejemplo : La materia 0076 se ha dejado de impartir en la escuela, eliminarla de la tabla MATERIA y eliminar también las referencias a este curso en las tablas GRUPO e INSCRIPC. Enlazar los cambios dentro de una transacción sencilla, que se lleve a cabo una vez que se han llevada a cabo todas las sentencias DELETE se hallan ejecutado correctamente. Suponer que se ejecuta en SQL programado en un programa de aplicación..

```
{punto de inicio de transacción establecido automáticamente al correr el programa}
DELETE
FROM Inscripc
WHERE Inogrupo =
      (SELECT Gnogrupo
      WHERE Gnomateria = "0076")

DELETE
FROM Grupo
WHERE Gnomateria = "0076"

DELETE
FROM Materia
WHERE Mnomateria = "0076"

COMMIT WORK.
```

3.2.3. Transacciones: ¿ Qué hace el DBMS ?.

Esta sección es un poco técnica y puede no leerse si el lector del presente trabajo así lo desea, en la presente sección se trata de explicar como es que el DBMS puede deshacer los cambios efectuados a una base de datos, especialmente si hay una falla en el sistema a la mitad de una transacción. Aunque las técnicas entre los diferentes productos varían, casi todas se basan en un *registro de transacción (transaction log)*, como se muestra en la siguiente figura:

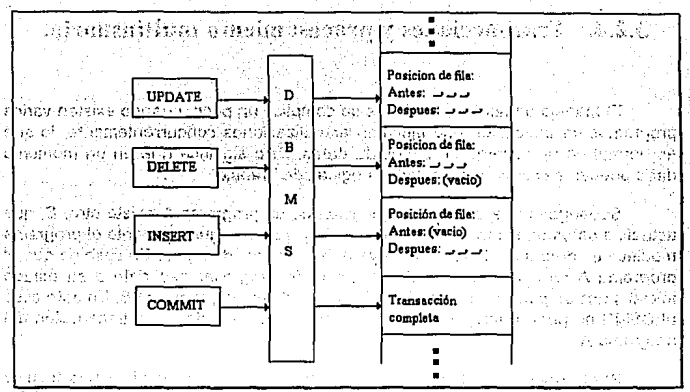


FIG. 3.1 Registro de transacción.

Cuando un usuario ejecuta una sentencia de modificación , el DBMS escribe automáticamente el registro de transacción mostrando dos copias de cada fila afectada por la sentencia. Una copia muestra un reflejo de la fila antes de la actualización , y la otra copia muestra la fila después de la modificación.

Si el usuario ejecuta posteriormente una sentencia COMMIT, el fin de la transacción se anota en el registro de transacción. Si el usuario ejecuta una sentencia ROLLBACK, el DBMS examina el registro de transacción para regresar las filas como se encontraban al inicio de la transacción. Utilizando el contenido de como estaban los registros antes, el DBMS restaura las filas a su estado original deshaciendo las modificaciones realizaron algunas sentencias de la transacción.

En caso de que haya una falla en el sistema, el DBMS revisa el registro de transacción, y busca las transacciones que no hayan finalizada correctamente, y actúa con estas como si hubiese encontrado un ROLLBACK, volviendo cada registro de la transacción a su estado original.

3.2.4. Transacciones y procesamiento multiusuario.

El manejo de las transacciones se complica un poco, cuando existen varios programas de aplicación que ejecuten actualizaciones concurrentemente, lo que es normal en un sistema de bases de datos. Esto significa que en un momento dado pueden existir varias Unidades Lógicas de Trabajo.

Supóngase que durante la ejecución de un programa A existe otro, B, que actualiza datos en las mismas tablas que A, y que en algún momento el programa modifica un dato que el programa A ya había modificado antes. En caso de que el programa A terminara mal, el DBMS no podría regresar ese dato a su estado inicial, pues se perdería la modificación realizada por el programa B. En este caso el DBMS no podría garantizar la ejecución de todo o nada de la transacción del programa A.

Para prevenir la situación anterior, los DBMS suelen emplear una técnica consistente en bloquear todos los datos que son modificados por una transacción, conforme se ejecuten sus sentencias de actualización. Esto quiere decir que estos datos son inaccesibles para las otras transacciones. Si alguna de las otras transacciones necesita acceder a estos datos, el DBMS la hará esperar hasta que la transacción que realizó el bloqueo termine. De esta manera un mismo dato no puede ser modificado en un momento dado por dos o más transacciones, con lo que el DBMS puede deshacer cualquier transacción conservando el compromiso de todo o nada.

Bloquear los registros que son modificados por una transacción nos lleva a otro problema: si una transacción dura mucho tiempo puede provocar largas esperas en otras transacciones, lo que puede ser inaceptable.

Para evitar estas situaciones, hay que descomponer la transacción asociada a un conjunto de transacciones mas cortas. Ya se estudio como hacerlo mediante sentencias COMMIT y ROLLBACK. Ambas indican el final de una transacción y el comienza de otra. Es una buena práctica hacer siempre las transacciones lo mas cortas posibles

3.2.5. Bloqueos (Locking)

Como se menciona en la sección anterior, para el manejo de transacciones concurrentes de mucho usuarios, los principales productos SQL utilizan sofisticadas técnicas de Bloqueo. A continuación se explican los diferentes niveles de Bloqueo que existen

3.2.5.1. Niveles de bloqueo.

El bloqueo puede ser implementado a varios niveles en la base de datos. El DBMS puede bloquear a *nivel base de datos*. Este bloqueo, aunque es sencillo no permitiría realizar otra transacción, lo que conduce a un rendimiento muy pobre.

Una forma mejorada de bloqueo es el bloqueo a *nivel de tabla*. El DBMS bloque las tablas que son accesadas por una transacción, mientras que otras transacciones pueden al mismo tiempo accesar otras tablas. Aun con este tipo de bloque se sigue teniendo un rendimiento inaceptable. Supóngase, en la base de datos de ejemplo, que es día de inscripciones y existen varias ventanillas para dar de alta a los alumnos en sus respectivos grupos, esto implicaría que al dar de alta un alumno se bloquearía la tabla INSCRIPC, y la otra ventanilla tendrá que esperar a que el programa de la primer ventanilla libere la tabla. El rendimiento en este caso

Muchos productos DBMS implementan el *bloqueo a nivel página*. En este tipo de bloqueo el DBMS bloquea un grupo de datos procedentes de una fracción de disco (*páginas*) conforme son accesados por una transacción. Habitualmente se utilizan páginas de 2KB, 4KB, y 16KB. Dado que una tabla extensa ocupará cientos, o miles de páginas, dos transacciones que accedan dos filas diferentes de una tabla, lo harán, generalmente a dos páginas diferentes, permitiendo a las dos transacciones ejecutarse en paralelo.

Algunos productos DBMS han pasado del nivel de página al bloqueo a *nivel de fila*. Este tipo de bloqueo permite que dos transacciones que accesan a dos filas diferentes puedan ejecutarse en forma paralela. Aunque parezca una posibilidad remota, puede ser que dos transacciones ocurran simultáneamente sobre una misma fila, sobre todo en tablas pequeñas como en la tabla CARRERA, de la base de datos de ejemplo. El bloqueo a nivel fila proporciona un alto nivel de ejecución de transacciones en paralelo.

Teóricamente es posible pasar del bloqueo a nivel fila al bloqueo a nivel de datos individuales. Sin embargo el recargo en el manejo del bloqueo a nivel de datos, es mayor que sus posibles ventajas. Ningún DBMS de SQL proporciona bloqueo a nivel de datos. De hecho la técnica utilizada por los diversos productos DBMS comerciales para el bloqueo son muy sofisticados. Unos de los mas sencillos se muestran en la siguiente sección.

3.2.5.2. Bloqueos compartidos y exclusivos.

La mayoría de los DBMS comerciales emplean mas de una técnica de bloqueo. Los más habituales son los bloqueos compartidos y los bloqueos exclusivos:

- Un *bloqueo compartido* se utiliza cuando en una transacción se encuentra una sentencia de lectura. El DBMS permite que una transacción concurrente acceda los mismos datos si los accesa de lectura.
- Un *bloqueo exclusivo* se utiliza cuando una transacción desea actualizar los datos en la base. Cuando una transacción tiene bloqueo exclusivo, ninguna otra transacción podrá accesar los datos ni aunque se trate de una consulta.

Para que la diferencia entre ambos tipos de bloqueo quede mas explícita se presenta una ejemplo que muestra como se manejan dos transacciones con las dos formas de bloqueos:

Cuando la transacción A de la figura accede los datos con la sentencia UPDATE se establece un inicio de transacción, bloqueando los datos que utiliza, ya sea para consulta o modificación. Al mismo tiempo la transacción B bloquea las partes de la base que ella necesita. Si la transacción B trata de acceder alguna parte que la transacción A tenga bloqueada, el DBMS congela la transacción B, haciéndole esperar hasta que los datos sean desbloqueados por la transacción A.

El DBMS desbloquea los datos de la transacción A, cuando el programa de aplicación termina, o la transacción se encuentra un COMMIT o un ROLLBACK. En el momento que los datos son desbloqueados, el DBMS permite continuar a la transacción B.

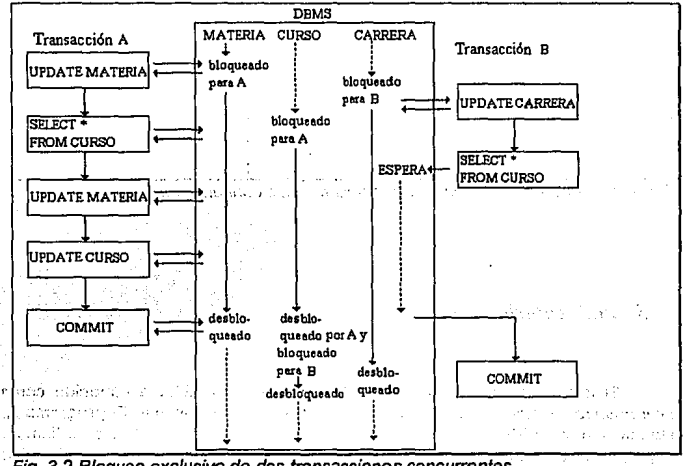


Fig. 3.2 Bloqueo exclusivo de dos transacciones concurrentes

En la figura anterior, una transacción realiza un bloqueo temporal sobre los datos que accesa, impidiendo que otras transacciones accesen (aunque sea de lectura) los datos bloqueados.

La siguiente figura, muestra las mismas transacciones de la figura anterior, pero utilizando bloqueos exclusivos y compartidos. Comparando ambas figuras se ve como se mejora el acceso a la base de dos transacciones concurrentes.

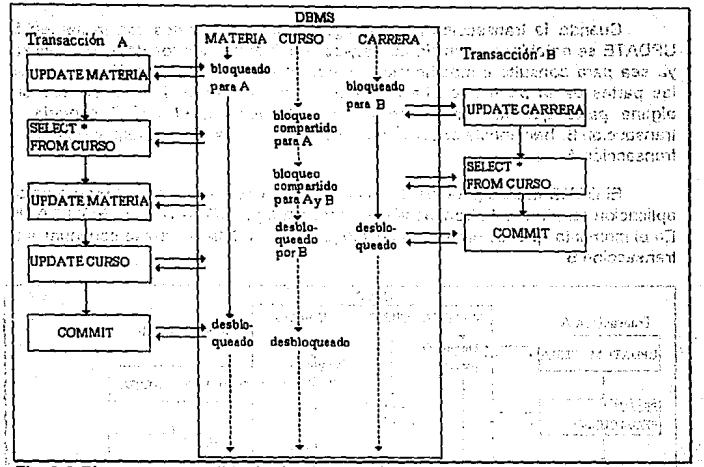


Fig. 2.3 Bloqueo compartido de dos transacciones concurrentes

3.2.5.3. Interbloqueos.

El uso de bloqueos por el DBMS produce un problema conocido como *interbloqueo (deadlock)*. La siguiente figura muestra el problema. El programa A, actualiza la tabla MATERIA, bloqeando por tanto parte de ella. Al mismo tiempo, el programa B, actualiza la tabla CURSO, bloqueando también parte de la tabla que accesa. Enseguida. el programa A trata de actualizar la tabla CURSO y el programa B, trata de actualizar la tabla MATERIA, en cada caso, se coincide que los programas tratan de accesar la parte bloqueada por el otro programa. Sin intervención alguna, cada programa esperará eternamente que el otro programa termine su transacción y libere los datos.

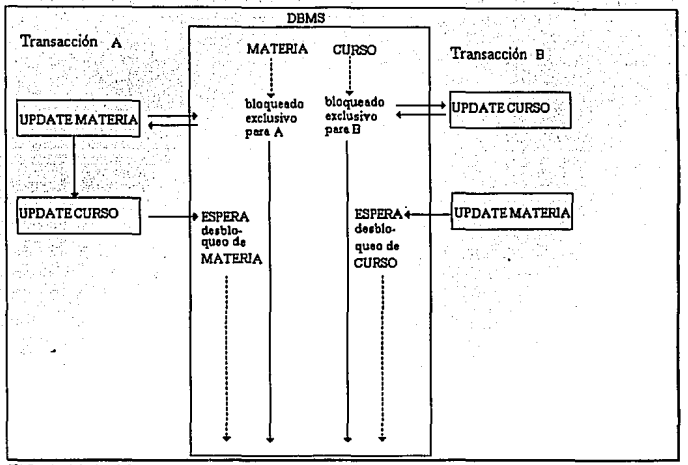


FIG. 3.4 Interbloqueos.

La situación que se presenta en la figura es un interbloqueo entre dos transacciones, pero pueden ocurrir situciones más complejas, donde se produzcan interbloqueos entre mas de dos transacciones.

El DBMS recurre a una lógica, donde periódicamente verifica los bloqueos que tienen las transacciones, cuando detecta un interbloqueo, el DBMS elige una transacción *perdedora* y le da ROLLBACK, liberando los bloqueos y permitiendo a la transacción *ganadora* proseguir. El programa perdedor recibe un código de error donde se le informa que ha sido perdedor de un interbloqueo.

En el SQL programado, el programa de aplicación debe estar preparado para manejar el código de error perdedor de un interbloqueo, y tomar las medidas adecuadas, ya sea informando al usuario del error o repitiendo la transacción.

Muchos productos de base de datos comerciales ofrecen facilidades avanzadas de bloqueo que van más alla de las proporcionadas por las transacciones SQL estándar. Sin embargo son específicas de cada producto y no se explicaran en este trabajo, que pretende dar un panorama del SQL estándar.

4. CREACIÓN Y ADMINISTRACIÓN DE UNA BASE DE DATOS BASADA EN SQL.

Otro papel no menos importante de SQL es que también permite definir la estructura y organización de una base de datos. En este capítulo se vera el lenguaje para la definición de datos y la creación de la base de datos. Se describirán las vistas, que es una característica de SQL que permite a los usuarios examinar los datos en organizaciones diferentes de como se encuentran en las tablas. Por ultimo se verá la seguridad que puede ofrecer SQL a los datos almacenados en la base.

4.1. Creación de una base de datos.

Generalmente en instalaciones DBMS para computadores grandes el administrador de la base de datos es el único responsable de la creación de nuevas base de datos. En instalaciones de DBMS para computadores más pequeños, los usuarios de la base pueden crear las bases de datos que utilicen.

Las técnicas usadas por los diferentes productos SQL varía ligeramente y siempre sería recomendable consultar los manuales del producto antes de proceder a crear la base de datos.

4.1.1. Creación de tablas.

La estructura mas importante de una base de datos relacional es una tabla. La sentencia CREATE TABLE se usa para crear una nueva tabla en la base de datos y la deja lista para aceptar datos. La sintaxis que se emplea se muestra a continuación, pero se explica en las siguientes secciones inmediatas:

CREATE TABLE nombre de tabla (definición de columna [, ...]
[definición de clave primaria]
[definición de clave secundaria]

definición de columna:

nombre de columna tipo de datos [NOT NULL [WITH DEFAULT]]

definición de llave primaria:

PRIMARY KEY (nombre de columna [, ...])

definición de llave foránea:

FOREIGN KEY nombre de relación (nombre de columna [, ...]) REFERENCES nombre de tabla
[ON DELETE {RESTRICT, CASCADE, SET NULL}]

La sintaxis parece compleja, ya que hay muchas partes de la definición que especificar y varias opciones para cada elemento. En la práctica, la creación de una nueva tabla es relativamente sencilla.

Aunque la longitud máxima para el nombre de la tabla no es constante en los diferentes productos, todos requieren que el nombre no sea una palabra reservada por el lenguaje SQL, y que comiencen con una letra que puede estar seguida de dígitos y el caracter subrayado.

4.1.1.1. Definiciones de columnas.

Las columnas de la nueva tabla constituyen el cuerpo de la sentencia CREATE TABLE. Los definiciones de columna se especifican inmediatamente después del nombre de la nueva tabla en una lista que se encierra entre paréntesis y se separa por comas. El orden de las definiciones determina el orden de izquierda a derecha de las columnas en la tabla. La definición de columnas consta de los siguientes elementos:

- *Nombre de columna.* Los nombres de columna deben de ser únicos dentro de una tabla dada, pero es posible que aparezca en varias tablas.
- *Tipo de datos.* Se especifica después del nombre de columna. Los tipos de datos que existen para los productos mas comerciales fueron presentados en el capítulo 2.
- *La cláusula NOT NULL o NOT NULL WITH DEFAULT.* Esta cláusula es opcional. Impide que aparezcan valores NULL en la columna.

4.1.1.2. Valores por omisión.

Existen circunstancias donde los valores nulos no son razonables, por ejemplo en un campo que sirva de llave primaria, o en la tabla ejemplo, cada materia debe pertenecer a una carrera por lo que el campo MNOCARRERA nunca debe ser NULL. La presencia de esta cláusula indica al sistema que rechace cualquier actualización de la tabla , que contenga un valor nulo en la columna donde se especifique.

Los productos SQL proporcionan un valor por default especifico para cada tipos de dato soportado. El valor por omisión de un dato numérico es cero, y el valor por omisión de un dato de tipo caracter es una cadena de blancos, y el valor por omisión de los datos tipo fecha y tiempo son la fecha y tiempo actual.

Al especificar NOT NULL WITH DEFAULT. Es similar a NOT NULL porque impide que aparezcan valores nulos en la columna donde se especifica. Sin embargo, una operación de actualización que contenga un valor nulo en la columna no es rechazada, en lugar de esto el valor nulo será cambiado automáticamente por el sistema por el valor por omisión.

Ejemplo: Crear la tabla Carrera de la base de datos ejemplo.

```
CREATE TABLE Carrera
        (Cnocarrera CHAR(4) NOT NULL,
        Cnombre VARCHAR(15) NOT NULL WITH DEFAULT,
        Cedificio VARCHAR(6) NOT NULL WITH DEFAULT,
        Ccoord CHAR(8) NOT NULL WITH DEFAULT,
```

4.1.1.3. Definiciones de clave primaria y foránea.

La sentencia CREATE TABLE, mediante las cláusulas PRIMARY KEY y FOREIGN KEY identifica la llave primaria de la tabla y las relaciones de esta tabla con otras tablas de la base de datos. Aunque hay algunos productos que no soportan estas cláusulas.

La cláusula PRIMARY KEY especifica la o las columnas que forman la clave primaria de la tabla. Se debe recordar del capítulo uno que una clave primaria es un valor que nunca es nulo y es único dentro de la tabla. Por esta razón, todas las columnas que forman esta llave deberán especificar NOT NULL en su definición.

La cláusula FOREIGN KEY especifica la clave foránea en la tabla y la relación que se crea con otra tabla en la base de datos. Recordemos también que una clave foránea es una columna, o un grupo de columnas, cuyo valor es igual al de alguna clave primaria en otra tabla.

Se puede especificar opcionalmente un nombre para la relación. Este nombre no es necesario para crear la clave foránea, pero si posteriormente se desea borrar esta clave foránea no será posible sin hacer referencia la nombre de la relación.

En la cláusula FOREIGN KEY, también se especifica una regla de supresión opcional para la relación (RESTRICT, CASCADE o SET NULL) si se omite la regla de supresión el default es RESTRICT.

Ejemplo: Crear la tabla MATERIA.

```
CREATE TABLE Materia
        (Mnomateria  CHAR(4) NOT NULL,
        Mnombre      VARCHAR(22) NOT NULL WITH DEFAULT,
        Mcred        SMALLINT,
        Mcostolab    DECIMAL (5,2),
        Mnocarrera   CHAR(4) NOT NULL WITH DEFAULT,
PRIMARY KEY (Mnomateria),
FOREIGN KEY (Mnocarrera)
        REFERENCES Carrera
        (ON DELETE RESTRICT)
```

El ejemplo muestra que las columnas MCRED y MCOSTOLAB aceptan valores nulos, en caso de que en una sentencia INSERT no se especifique el valor de MNOMBRE el valor por default será de espacios. La clave primaria en este caso es MNOMATERIA que no acepta valores nulos. La clave foránea en el ejemplo es MNOCARRERA que con la cláusula REFERENCES nos indica que el valor de la columna debe de ser igual a la llave primaria de la tabla CARRERA.

En el ejemplo, ON DELETE RESTRICT, le indica al sistema que rechace cualquier operación de DELETE de una fila para la tabla CARRERA, que tenga un valor en su llave primaria igual a uno o mas valores de MNOCARRERA en la tabla MATERIA. En el ejemplo se muestra, aunque no es el propósito principal, el uso de una clave foránea para cuidar la integridad referencial.

4.1.1.4. Reglas de supresión:

Las reglas de supresión que se especifican en la cláusula ON DELETE para las llaves foráneas. Estas reglas le dicen al DBMS que debe de hacer en caso de que se intente borrar una fila de la tabla padre. Las reglas permitidas son:

- RESTRICT que impide borrar una fila de la tabla padre si la fila tiene algún hijo en la tabla donde se esta especificando la restricción. Si se intenta ejecutar una sentencia DELETE que intente borrar una fila padre que aún tenga hijos, la sentencia será rechazada con su respectivo mensaje de error. En el ejemplo anterior, se da la restricción para que no se borren filas de la tabla CARRERA si su llave primaria CNOCARRERA coincide con la columna MNOCARRERA.

- CASCADE. Esta regla de supresión es peligrosa en su manejo, le dice al DBMS que cuando una fila de la tabla padre se suprima, también automáticamente se supriman todos los hijos de la tabla donde se especifica esta regla. En el ejemplo esto implicaría que al borrar una fila de la tabla CARRERA se borraran todas las filas de la tabla MATERIA donde se haga referencia a la fila borrada en la tabla padre.

- SET NULL. Indica al DBMS que cuando una fila padre sea suprimida, el valor de la llave foránea en la tabla hijo se le asigne un valor nulo. En el ejemplo, esto implicaría que al borrar una fila de la tabla CARRERA, el valor de la columna CNOCARRERA cambiaría a espacios dado que la columna tiene la cláusula NOT NULL WITH DEFAULT.

4.1.1.5. Catalogo del sistema.

La mayoría de los productos SQL, mantienen un catálogo que contiene la información sobre la existencia de tablas en la base de datos, columnas, claves primarias y secundarias, es decir, contiene toda la información de las tablas. Cuando se ejecuta una sentencia CREATE TABLE, el DBMS almacena la información de la nueva tabla en este catálogo, posteriormente si se cambia la estructura de la tabla, también se actualiza el catálogo del sistema.

Cuando el sistema procesa una consulta, examina este catalogo para comprobar la existencia de tablas y columnas referenciadas en la consulta. Este catalogo del sistema es presentado como un conjunto de tablas que como una tabla común puede referenciarse con cláusulas SELECT, la estructura de las tablas del sistema no se explican porque son especificas de cada producto SQL. Una vez que se conoce SQL es fácil consultar el manual de cualquier producto, conocer la estructura del catalogo del sistema y realizar las consultas deseadas.

4.1.2. Eliminación de una tabla.

La sentencia DROP TABLE permite eliminar de la base de datos una tabla que ya no es necesaria. Su sintaxis es muy simple:

Ejemplo: Eliminar la tabla MATERIA.

DROP TABLE Materia

La sentencia DROP borra todos los registros de la tabla y la definición de la tabla. Como se ve la sentencia DROP es substancialmente peligrosa, por lo que debe utilizarse con mucho cuidado. Si la tabla tiene índices asociados (que en este mismo capítulo se vera la manera de crearlos), se eliminarán automáticamente al borrar la tabla.

4.1.3. Modificación de una definición de tabla.

La sentencia ALTER TABLE nos puede ayudar para añadir una definición de columna a una tabla, definir una clave primaria, definir una clave foránea, eliminar una clave primaria o foránea.

Sintaxis:

ALTER TABLE nombre de la tabla
[ADD definición de columna]...
[definición de clave primaria]...
[definición de clave secundaria]...
[DROP PRIMARY KEY]...
[DROP FOREIGN KEY nombre de relación]...

Donde :

definición de columna:

nombre de columna tipo de datos [NOT NULL WITH DEFAULT]

definición de llave primaria:

PRIMARY KEY (nombre de columna [, ...])

definición de llave foránea:

FOREIGN KEY nombre de relación (nombre de columna [, ...]) REFERENCES nombre de tabla
[ON DELETE {RESTRICT, CASCADE, SET NULL}]

4.1.3.1. Adición y supresión de columnas.

Ejemplo: Adicionar dos nuevas columnas que contengan el peso y la estatura de cada estudiante en la tabla ESTUDIAN. Su estatura debe de darse en centímetros y su peso en kilogramos, se pide una precisión de un dígito después del punto decimal.

```
ALTER TABLE Estudian
    ADD Peso DECIMAL(4,1)
        NOT NULL WITH DEFAULT
    ADD Estatura DECIMAL(4,1)
        NOT NULL WITH DEFAULT
```

El adicionar columnas es el uso mas frecuente de la sentencia ALTER TABLE. La sintaxis es parecida a la utilizada en la sentencia CREATE TABLA, perro debe notarse que en la definición de columnas no se puede especificar únicamente NOT NULL , dado que al insertar una nueva columna, ésta contendría de inicio valores nulos, violando inmediatamente la regla y produciendo un error, por lo que sólo se puede usar NOT NULL WITH DEFAULT u omitirse para que asigne NULL en las nuevas columnas.

No existe una sentencia de SQL que permita eliminar columnas. Para realizar esto, se debe crear una nueva tabla con las columnas deseadas y copiar la información de la tabla original a la nueva tabla, ya sea mediante la sentencia INSERT de múltiples filas, o mediante alguna utilería que proporcione el fabricante del producto SQL especifico.

4.1.3.2. Modificaciones de claves primaria y foránea.

La sentencia ALTER TABLE también permite, cambiar, añadir o borrar las definiciones de las claves primaria y foránea de una tabla. La sintaxis para añadir una clave primaria o una clave foránea es igual a la que se utiliza en la sentencia CREATE TABLE y funciona del mismo modo. Si se desea eliminar una clave foránea, se tiene que hacer referencia al nombre de relación especificada en la sentencia CREATE TABLE, de otra manera la única manera de eliminar la clave foránea, será eliminar la tabla y volverla a construir, sin especificar la llave foránea que se desea eliminar.

Ejemplo: Supóngase que se creo la tabla ACADEMIC y se olvido definir como clave foránea la columna Anocarrera. Alterar la tabla y definir la relación ACAD_CARR.

```
ALTER TABLE Academic
FOREIGN KEY Acad_carr (Anocarrerra)
REFERENCES Carrera
```

Ejemplo: Eliminar la clave foránea que se definió en el ejemplo anterior.

```
ALTER TABLE Academic
DROP FOREIGN KEY Acad_carr
```

4.1.4. Índices.

Un índice es una estructura interna que el sistema puede usar para accesar rápidamente una o mas filas de una tabla, en base a los valores de una o más columnas.

Un índice de una base de datos es similar al índice que se puede encontrar en un libro. De la misma manera que el lector del libro acude al índice para localizar en que páginas se encuentra determinado tema, el DBMS leerá un índice para determinar las posiciones de las filas seleccionadas en una consulta SQL. Los índices ayudan al DBMS a procesar las sentencias que accesan datos de una manera mas eficiente.

Un índice de datos puede crearse para una o mas columnas. Veamos el siguiente ejemplo para tratar de explicar la función de los índices.

Ejemplo: Encontrar el número de cuenta y dirección del estudiante Juan Bisset.

```
SELECT Enocuenta, Enombre, Edomicilio
    FROM Estudian
    WHERE Enombre = "Juan Bisset"
```

Como se ve, para el usuario no existe ninguna diferencia, en la forma en que se accesa una tabla. La sentencia no indica si existe un índice para ENOMBRE o no y el DBMS tendrá una respuesta para la petición, ya sea que el índice exista o no exista.

En caso de que no existiera un índice para la columna ENOMBRE, el DBMS recorrería fila por fila, verificando si se cumple la condición. Para asegurarse de que no existen mas filas que cumplan la condición, el DBMS, deberá recorrer todas y cada una de las filas de la tabla donde se realiza la consulta.

Si existe un índice para el campo ENOMBRE, el DBMS realizará un esfuerzo mucho menor para producir el mismo resultado. El DBMS examina el archivo de índices hasta encontrar donde se cumpla la condición, y dado que los índices se encuentran ordenados ya sea en forma ascendente o descendente, el DBMS emplea técnicas de búsqueda que le facilitan el trabajo. Una vez que el DBMS encontró en el índice las filas que cumplen la condición, encontrarlas en la tabla es muy sencillo, dado que el índice le informa en que lugar del disco se encuentran las filas.

Si se sabe que se van a realizar frecuentemente consultas en una tabla por determinada columna, sería recomendable indexar la tabla por esa columna. Si las consultas por esa columna no son frecuentes no es conveniente indexar, dado que la indexación consume espacio adicional en disco. Otra de las desventajas de indexar una tabla, es que el índice debe de ser actualizado cada vez que se añade o borra una fila, o cuando la columna indexada es modificada. Esto significa recargos adicionales para el DBMS cuando realizan operaciones con las sentencias INSERT, DELETE y UPDATE.

El DBMS siempre crea automáticamente un índice para la clave primaria, ya que se supone que el usuario de la base, al elegir una llave primaria, utilizara esta frecuentemente en los accesos.

4.1.4.1. Creación de índices.

La sentencia CREATE INDEX se utiliza para crear índices. La sentencia le da un nombre al índice, especifica las columnas a indexar, el orden ascendente o descendente.

Sintaxis:

CREATE INDEX nombre de índice
ON nombre de tabla (nombre de columna [ASC/DES], [nombre de columna ...])

Ejemplo: Crear un índice en la columna ENOMBRE para la tabla ESTUDIAN. Llamar al índice XENOMBRE.

CREATE INDEX Xenombre
ON Estudian (Enombre)

La cláusula ON debe de referenciar a una tabla ya existente y a una columna de ésta tabla. En el ejemplo la tabla ESTUDIAN ya tiene registros . El DBMS explora la tabla para obtener los valores de ENOMBRE y las posiciones de las filas para crear el índice correspondiente. Es una buena práctica utilizar la sentencia CREATE INDEX antes de que la tabla contenga alguna fila.

Una vez creado el índice, el DBMS se encargara del resto, cada que se ejecute una sentencia de actualización, el DBMS actualizara los valores y posiciones en el índice.

4.1.4.2. Eliminación de índices.

La sentencia utilizada para eliminar un índice de una tabla es: DROP INDEX.

Sintaxis:

DROP INDEX nombre del índice

Ejemplo: Eliminar el índice del ejemplo anterior.

DROP INDEX Xenombre

4.1.5. Estructuras de bases de datos.

A estas alturas el lector del presente trabajo se estará preguntando como crear una base de datos, si ya se explicó como crear tablas e índices.

Aunque varios productos SQL proporcionan la misma estructura dentro de una única base de datos, hay mucha variación de como se organizan y estructuran las diferentes bases de datos en cada sistema informático particular.

Algunos productos suponen una base de datos única que almacena a todas las tablas del sistema, las tablas son diferenciadas por su propietario o creador. Otros pueden crear bases de datos múltiples, y cada base de datos se diferencia con un nombre especifico. Por último, también algunos productos soportan bases de datos dentro del contexto de directorios y subdirectorios del computador, de tal manera que para accesar una base de datos se debe especificar el directorio donde se encuentra.

Las variaciones que se presentan no cambian el modo de utilizar SQL para accesar los datos de la base. Solamente afectan el modo en que inicialmente se obtiene acceso a la base de datos.

Los productos que utilizan una arquitectura de base de datos única, contiene todas las tablas en una misma base y el usuario, con solo conectarse ya esta habilitado para accesar las tablas, donde el usuario que se conecta sea el dueño.

Los productos que no utilizan una arquitectura de base de datos única, soportan en su mayoría la sentencia CREATE DATABASE para crear una base de datos, la sentencia se ejecuta una vez para cada sistema de aplicación y la base de datos nueva contendía todas las tablas de dicho sistema.

Una vez creada la base de datos la manera de accesarla, varia también en los diferentes productos SQL. Estos son algunos ejemplos de como accesan a una base de datos algunos productos:

CONNECT base de datos (Ingres)

USE base de datos (SQL Server, Sybase)

DECLARE DATABASE alias de la base
FILENAME "camino en subdirectorio, nombre de base
(SQL VAX)

Como se ve, existe una gran variedad en el modo en que los diferentes productos DBMS organizan sus bases de datos y proporcionan acceso a ellas, por lo que no se profundiza en ningún producto en particular. Esta área de SQL es una de las menos estándar, y paradójicamente es la primera cosa que tiene que hacer el usuario para acceder o crear una base de datos. Estas diferencias muestran como el SQL no puede cambiar de un DBMS a otro transparentemente, aunque el proceso de conversión no es complejo generalmente.

4.2. Vistas.

Si un usuario tiene datos que desea que no sean accesados por otros usuarios, la definición y uso de vistas le permitirá que otros usuarios accesen solamente parte de una tabla, guardando la privacidad de parte de ella.

4.2.1. Concepto de vista.

Una vista es una sentencia SELECT a la que se designa un nombre y que se almacena como una tabla permanente del sistema. Dado que al ejecutar la sentencia se obtiene una tabla, nada impide que se puedan construir consultas sobre ella.

Una vista también es definida como una *tabla virtual*, cuyo contenido esta definido por una consulta. Se le denomina "tabla", porque el usuario percibe y manipula una vista SQL del mismo modo que se han manipulado las tablas a lo largo de este trabajo. Se le denomina "virtual", porque las filas y columnas de una vista no existen realmente. Las filas y columnas de una vista, son los resultados de la consulta que define la vista. La vista corresponde a los datos de una *tabla real*. El concepto de tabla real o tabla fuente se maneja en esta sección y se refiere a cualquier tabla que fue creada con la sentencia CREATE TABLE

Tal vez explicando como maneja el DBMS las vistas, quede más claro el concepto de vista.

4.2.2. Como maneja el DBMS las vistas.

Cuando se realiza una consulta respecto a una vista, el DBMS consulta su definición de la vista en el catálogo del sistema y extrae el nombre de la tabla real a la que hace referencia la vista y la sentencia SELECT que la define. El DBMS fusiona la consulta de la definición y la consulta que se realiza a la vista, traduciéndola en una nueva consulta equivalente, que es la que se ejecuta. De este modo el DBMS mantiene la ilusión de la vista mientras se mantiene la integridad de las tablas fuente.

El paso de definir y procesar vistas significa que cuando se ejecuta una consulta respecto a una vista siempre se aplica a su correspondiente tabla real. Por esta razón, el DBMS no tiene que almacenar una copia los datos para representar una vista.

Algunos productos, al accesar la definición de una vista, ejecutan automáticamente la sentencia SELECT que la define, y colocan el resultado en una tabla temporal. La consulta que se realice respecto a la vista se ejecuta sobre el resultado intermedio.

4.2.3. Ventajas y desventajas de las vistas.

4.2.3.1. Ventajas.

Las vistas se utilizan para tres fines principalmente:

1. Prohibir el acceso a datos confidenciales.
2. Simplificar la formulación de consultas complejas o repetitivas.
3. Aumentar la independencia de los programas con respectos a los datos.

A continuación se comenta un poco cada uno de ellos.

4.2.3.1.1.Protección a la confidencialidad.

Si una tabla contiene datos que no deben de ser accesados por algunos usuarios, se puede construir una vista en la que estos datos no aparezcan y autorizar a estos usuarios a trabajar con la vista en vez de la tabla, con lo que los datos no visualizados de ésta no podrán ser vistos por ellos, como si no existieran.

Se pueden esconder columnas completas, o filas que cumplan alguna condición. En algunos textos se le llama *vista vertical* a la vista en la que se esconden columnas completas, y *vista horizontal* a la vista en donde solamente se presentan algunas filas donde se cumpla alguna condición.

4.2.3.1.2. Facilidad de uso.

Supóngase que una consulta complicada (por ejemplo consultas multitabla y subconsultas) debe ser utilizada frecuentemente. Se puede formular la consulta y crear una vista con ella, de manera que la consulta complicada se convierte en simple SELECT sobre la vista, lo que ahorrará tiempo y posibles equivocaciones.

4.2.3.1.3. Independencia con respecto a los datos.

Supóngase que por alguna razón, es necesario reestructurar una tabla, cambiando el nombre de sus columnas, o desglosando su información en dos tablas. Entonces, todas las sentencias SELECT incluidas en los programas y que se refieran a esta tabla, deberán de ser revisadas y probablemente modificadas. Sin embargo, si se define una vista de manera que su nombre y columnas sean iguales a las de la tabla vieja, las sentencias seguirían valiendo. O si las sentencias SELECT se aplican todas a una vista, solamente será necesario cambiar la consulta que define a la vista.

4.2.3.2. Desventajas.

Así como las vistas presentan ventajas, también presentan algunas desventajas: rendimiento y restricciones de actualización.

Cuando se hace una consulta a una vista, el DBMS consulta la definición de ésta en el catálogo y ejecuta la sentencia SELECT. Si la vista se define mediante una consulta muy complicada, entonces, hasta la mas sencilla de las consultas a la vista podría tardar algo de tiempo.

Algunos DBMS no permiten la actualización de las vistas, es decir las crean "de solo lectura" . Otros DBMS solo permiten las actualizaciones en las vistas que no se definen con consultas multitabla. Sea cual sea el caso, existen restricciones de actualización para las vistas complejas, lo cual es una desventaja.

Las desventajas que existen significan que no se puede definir vistas para cada tabla real. Se deben de poner en una balanza las ventajas y las desventajas y decidir si se utiliza una vista o no.

4.2.4. Creación de vistas.

La sentencia CREATE VIEW se utiliza para definir el contenido de una vista SQL, que esta definida para alguna tabla real. Una vez que se establece una vista, se ejecuta una sentencia SELECT que define la vista. El DBMS asocia la vista con una tabla real.

La sentencia CREATE VIEW también puede dar un nombre a cada columna en la vista, que sea diferente a las columnas de la tabla de la consulta, si se especifica una lista de nombres de columna, debe de tener el mismo número de elementos que el número de columnas que produzca la consulta.

Sintaxis:

CREATE VIEW nombre de vista [(lista de nombres de columna)] AS consulta

En la lista de nombres de columna únicamente se especifican los nombres, ya que la definiciones y características de cada columna los toma de su correspondiente columna en la tabla real. Si se omite la lista de los nombres de columna, el sistema asume los mismos nombres de columna que los de la tabla fuente y se debe especificar en caso de que contenga columnas calculadas o la consulta sea un join de dos tablas que contengan columnas con idéntico nombre.

El siguiente ejemplo crea una vista que corresponde a un subconjunto de columnas y filas de la tabla MATERIA.

Ejemplo: Crear un vista denominada MCOMP, que contenga las filas que correspondan a la carrera de computación. La vista debe contener las columnas de la tabla MATERIA, con excepción de la columna MNOMATERIA, que sería redundante.

```
CREATE VIEW Mcomp AS
    SELECT Mnomateria, Mnombre, Mcred, Mcostolab
    FROM Materia
    WHERE Mnocarrera = 32
```

Si se ejecutara directamente la sentencia SELECT que define la vista, el resultado correspondería la contenido de la definición de la vista. El DBMS no ejecuta la sentencia SELECT, simplemente la guarda como parte de la definición de la vista.

Hay dos restricciones que existen para las cláusulas SELECT que definen una vista:

- La cláusula ORDER BY no puede estar incluida en la sentencia SELECT. Esto es debido a que por definición, las tablas y las vistas no deben tener una secuencia predefinida. La cláusula ORDER BY se puede utilizar en la sentencia SELECT que recupere los datos de una vista.
- La cláusula UNION no puede ser incluida en la sentencia SELECT. Esto es solamente debido a que en caso de crecer (en filas o columnas) las tablas que participan en la operación unión, se perdería el control del tamaño de la vista, recuerde que el número de filas de una operación unión es el resultado de la multiplicación del número de filas de ambas tablas.

4.2.4.1. Procesamiento de vistas.

Se pueden realizar consultas sobre las vistas de la misma manera como se han procesado las tablas a lo largo de este trabajo.

Ejemplo : Visualizar la vista Mcomp.

```
SELECT *
FROM Mcomp
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB
0076	Bases de Datos	8	100.00
0134	Sistemas Digitales	8	50.00
0119	Estructuras de Datos	8	0.00
0056	Estructuras Discretas	7	0.00
0559	Memorias y Periféricos	10	100.00
0561	Microcomputadoras	10	500.00

Como se ve, la sintaxis de la sentencia SELECT no cambio. Se hace referencia a la vista, simplemente mencionándola en la cláusula FROM. Generalizando, ninguna sentencia de manipulación de datos cambia al usarse en tablas o en vistas. Veamos otro ejemplo.

Ejemplo: Para las tarifas mayores a N$ 100 en la tabla Mcomp visualizar el nombre y tarifa. Clasificar el resultado en forma ascendente.

```
SELECT *
    FROM Mcomp
    WHERE Mcostolab >= 100
    ORDER BY Mnombre
```

Resultado:

MNOMATERIA	MNOMBRE	MCRED	MCOSTOLAB
0076	Bases de Datos	8	100.00
0559	Memorias y Periféricos	10	100.00
0561	Microcomputadoras	10	500.00

En el actual ejemplo se usa la cláusula WHERE y la cláusula ORDER BY para seleccionar filas especificas de la vista y ordenar el resultado de la consulta. Obsérvese que la consulta se realiza como si se tratara de una tabla. La consulta del ejemplo, también pudo haberse escrito de la siguiente forma. y produciría los mismos resultados:

```
SELECT *;
    FROM Materia
    WHERE Mnocarrera = "32"
        AND Mcostolab >= 100
    ORDER BY Mnombre
```

Sin embargo, esta sentencia es más compleja. En ocasiones el Administrador de la base de datos, solo permitirá al usuario accesar las vistas y no las tablas, para ejecutar la anterior sentencia se debe de conocer la tabla MATERIA.

4.2.4.2. Especificación de columnas para una vista.

En el siguiente ejemplo se ve como renombrar, explícitamente las columnas de una vista.

Ejemplo: Crear una vista denominada VCAROS con los laboratorios mas caros de todas la carreras, es decir que sean mayores o iguales a N$ 100.00. Las columnas correspondientes a MNOMATERIA, MNOMBRE Y.MCOSTOLAB, serán VNO_CARO, VNOM_CARO y VCOSTO_CARO

```
CREATE VIEW Vcaros
(Vno_Caro, Vnom_Caro, Vcosto_Caro)
AS SELECT  Mnomateria, Mnombre, Mcostolab
      FROM Materia
      WHERE Mcostolab >= 100
```

Los nombres de los campos de la vista empiezan con "V", para recordar que la tabla que se consulta es en realidad una vista, pues se puede olvidar y creerse que es una tabla, lo que podría generar problemas, si se desea actualizar, como se verá mas adelante en este mismo capítulo. La práctica de que las vistas y columnas de las vistas comiencen con "V", es una práctica útil, pero no necesaria.

Los nombres de columna deben referenciarse como son definidos en la vista. Si se intenta referenciar los nombres de columna como en la tabla real, se produciría un error.

Ejemplo: Visualizar la tabla VCAROS, donde el costo sea mayor a 100.

```
SELECT *
      FROM Vcaros
      WHERE Vcosto_caro > 100;
```

Resultado:

VNO_CARO	VNOM_CARO	VCOSTO_CARO
0024	Circuitos Eléctricos	150.00
0561	Microcomputadoras	500.00

4.2.4.3. Vista agrupada.

En la definición de la vista se puede incluir la cláusula ORDER BY. Cuando se usa esta cláusula, a la vista que se crea se le conoce como *vista agrupada.* Las vistas agrupadas tienen la misma función de las consultas agrupadas; agrupan filas y producen una fila por cada grupo. Una vista agrupada reúne los resultados para su consulta posterior.

Ejemplo; Crear una vista denominada VESTAD que tenga agrupadas las carreras y muestre la suma de los laboratorios, así como la media de los costos de los laboratorios.

```
CREATE VIEW Vestad
(Vnocarrera, Vtotal, Vpromedio)
        AS SELECT Mnomateria, SUM(Mcostolab), AVG(Mcostolab)
                FROM Materia
                GROUP Mnocarrera
```

El ejemplo muestra explícitamente los nombres de las columnas de las vistas. Esto es necesario debido a que las columnas que contienen datos calculados no se corresponden directamente. Las columnas calculadas son sumarios de las tablas reales, y por tanto requieren una substancial cantidad de procesamiento por parte del DBMS para mantener la ilusión de una vista agrupada

Esta vista no se puede actualizar ya que contiene datos calculados. Para generalizar, el DBMS no permite actualizaciones sobre ninguna vista definida con una cláusula ORDER BY, o con una columna que contenga una función. La razón parece ser obvia. ¿Que se supone que debería de hacer el DBMS al tratar de actualizar el valor promedio del costo de laboratorio para la carrera de Computación?. Dado que cada fila de una vista agrupada corresponde a un grupo de filas de la tabla real, no hay manera de trasladar la actualización de una vista agrupada a sus correspondientes tablas reales.

El siguiente ejemplo muestra de que manera nos puede servir definir una vista agrupada. La sentencia SELECT se simplifica al consultar la vista en lugar de la tabla real, y imposibilita a cometer un error al calcular las columnas cada vez que sea necesario.

Ejemplo: Seleccionar la tarifa media para cursos ofrecidos por las carreras Computación y Eléctrica..

```
SELECT Vnocarrera, Vpromedio
FROM Vestad
WHERE Vnocarrera = "32" OR
      Vnocarrera = "38"
```

Resultado:

VNOCARRERA	VPROMEDIO
32	125.00
38	75.00

4.2.4.4. Vistas compuestas,

Una *vista compuesta* es una vista cuya definición es una consulta SELECT con dos o más tablas. Las vistas compuestas simplifican las consultas multitabla. Una vez que se define la vista compuesta, las consulta multitabla se convierte en una consultas simple a la vista.

El usuario percibe las tablas que participan en la definición de la vista como una única tabla.

Ejemplo: Crear una vista que se llame Vjoin, que sea un join de las tablas MATERIA, ACADEMIC y GRUPO. Que contengan los cursos que ya tienen asignado profesor, seleccionando el nombre del profesor, su sueldo y el nombre de la materia.

```
CREATE VIEW Vjoin
AS
SELECT Academic.anombre, Academic.asueldo, Grupo.gnogrupo,
      Materia.mnombre
      FROM Grupo, Academic, Materia
      WHERE Academic.anoemplead = Grupo.gnomaestro
      AND Grupo.gnomateria = Materia.mnomateria
```

Una posterior consulta a la vista compuesta Vjoin, será muy simple. El usuario extraerá todos los datos de la vista en lugar de construir la consulta completa utilizando una operación join.

Ejemplo: Visualizar los cursos que ya tienen asignado profesor, seleccionando el nombre del profesor, su sueldo y el nombre de la materia, ordenar los resultados por el nombre del profesor.

```
SELECT *
FROM Vjoin
ORDER BY Anombre
```

ANOMBRE	ASUELDO	GNOGRUPO	MNOMBRE
Alberto Alvares	3000.00	2504	Circuitos Eléctricos
Alberto Alvares	3000.00	2501	Dispositivos Electrónicos
Armando Cruz	5000.00	2706	Elementos de Máquinas
Carlos Cuenca	7000.00	2705	Análisis Dinámico de Máquinas
Juan Méndez	3800.00	1157	Bases de Datos
Pedro Benitez	5200.00	1159	Estructuras de Datos

Naturalmente el DBMS tiene una carga de proceso igual al ejecutar la vista, que ejecutar una operación join sobre las tres tablas en cuestión. De hecho el DBMS debe de trabajar un poco más para procesar la vista que en el procesamiento de las tablas reales. Sin embargo para un usuario de la base de datos es mucho más fácil comprender y escribir una consulta basada en la vista que en las tablas reales.

4.2.4.5. Vista definida sobre otra vista.

La sentencia SELECT que define una vista puede referenciar a otra vista. El siguiente ejemplo crea una vista basada en la vista MCOMP, que a su vez esta basada en la tabla real CURSO.

Ejemplo: Crear una vista que se llame VCOMBARAT, que contenga las filas de la carrera de Computación con una tarifa menor a N$ 100. La vista solo contendrá las columnas de Mnombre y Mcostolab.

```
CREATE Vcombarat AS
SELECT Mnombre, Mcostolab
FROM Mcomp
WHERE Mcostolab < 100
```

Todos los datos requeridos por la vista VCOMBARAT se encuentran en la vista MCOMP. Puede darse el caso que el usuario no tenga acceso a la tabla Materia, pero si tenga acceso a la vista MCOMP. Por lo que le será necesario hacer referencia a la vista MCOMP.

Realmente no hay restricciones en el número de vistas que dependan de otras vistas, sin embargo se van haciendo mas complejas, y puede llegar el momento que se pierda el control sobre las condiciones de cada vista. Además de las restricciones de actualización sobre las vistas.

4.2.5. Actualización de las vistas.

En las operaciones de actualización realizadas sobre vistas, se presenta el siguiente problema. Si se especifica una sentencia INSERT, DELETE o UPDATE a una vista, las actualizaciones correspondientes se deben de traducir a operaciones equivalentes respecto a las tablas reales que corresponden a la vista. Esto no siempre es posible.

Si una vista tiene una funciones de columna, por ejemplo AVG, que calcula el promedio de un conjunto de datos, y se trata de actualizar a esta vista, el DBMS no permitirá la actualización; pues, ¿Como debe repercutir la actualización sobre el conjunto de valores que sirven para calcular el promedio?. Dado que cada fila de una vista agrupada corresponde a un grupo de filas de la tabla real, no hay manera de trasladar la actualización de una vista agrupada a sus correspondientes tablas reales. El mismo problema se presentaría al tratar de actualizar una vista con una cláusula ORDER BY, que también agrupa filas.

Actualmente no son actualizables las vistas que cumplen con alguna de las siguientes condiciones:

- Tienen filas no repetidas (DISTINCT en la cláusula SELECT.
- Tienen agrupamiento (GROUP BY o HAVING).
- Tienen funciones de columna (AVG(), SUM(), MAX(), etc).
- Se basan en otra vista que no es actualizable.
- La cláusula WHERE no debe incluir subconsultas.

4.2.6. Eliminación de una vista.

La sentencia que se utiliza para la eliminación de una vista es DROP VIEW y es muy sencilla.

Sintaxis:

DROP VIEW nombre de vista

Ejemplo: Eliminar la vista VJOIN de la base de datos.

DROP VIEW Vjoin

Cuando se borra una vista en la que se basa otra vista, todas lasa vistas que dependen de la vista eliminada son también borradas automáticamente.

Ejemplo: Eliminar la vista MCOMP y las vistas que dependan de ella.

DROP VIEW Mcomp

Este comando elimina MCOMP y VCOMBARAT, que depende de MCOMP. Si se usa la sentencia DROP TABLE para la tabla Materia, se eliminarían todas las vistas que dependan de la tabla

4.3. Seguridad SQL.

Un usuario no puede consultar o actualizar una tabla o una vista si no ha sido autorizado previamente para hacerlo. El manejo de autorizaciones es sistemas en computadores grandes, es responsabilidad de el administrador de la base de datos. Sin embargo en sistemas en computadores más pequeños o en el caso de vistas o tablas privadas de un usuario, éste debe de tener la posibilidad de autorizar a otros a usarlas.

SQL proporciona sentencias para especificar restricciones de seguridad. Para comprender la seguridad en SQL se tienen que tener en cuenta los siguientes conceptos:

- Los *usuarios*. Cada vez que el DBMS, consulta, inserta, suprime, crea o actualiza datos, lo hace a cuenta de algún usuario. El DBMS permitirá o prohibirá la acción dependiendo del usuario que requiera la operación.
- Los *objetos de la base de datos* son los elementos a los que se les puede aplicar seguridad. La seguridad SQL se aplica principalmente a tablas y vistas, sin embargo también se pueden proteger los programas de aplicación, o las bases completas. Cada usuario debe tener permiso para ciertos objetos, y también puede tener prohibidos otros.
- Los *privilegios* son acciones que un usuario tiene permitido efectuar para un objeto dado. Un usuario puede tener permisos para ciertas operaciones, por ejemplo para INSERT y SELECT, sin embargo puede tener prohibidos los privilegios de DELETE y UPDATE

4.3.1. Objetos de seguridad.

Las protecciones de seguridad de SQL se aplican a objetos de la base de datos. Aquí se verán los dos mas importantes: tablas y vistas. La mayoría de los productos SQL tienen otros objetos particulares cada uno, por ejemplo DB2 de IBM, los espacios de disco físicos para almacenar las tablas son tratados como objetos de seguridad, y así, otros productos SQL tienen sus propios objetos, los cuales no son explicados.

4.3.2. Privilegios.

Las acciones que un usuario puede efectuar sobre un objeto de la base de datos se denominan privilegios. Los privilegios que corresponden a tablas y vistas son:

- SELECT, que permite recuperar datos de una tabla o vista.
- INSERT permite insertar nuevas filas en una tabla o vista.
- DELETE que permite eliminar filas de una tabla o vista.
- UPDATE que permite actualizar las filas de una tabla o vista. Este privilegio puede restringirse a solo algunas columnas de la tabla o vista.
- ALTER permite usar la sentencia ALTER TABLE para cambiar la estructurar de una tabla.
- INDEX permite crear un índice para una tabla.

Recuérdese que un *nombre de tabla cualificado* especifica el nombre del propietario de la tabla junto con el nombre de la tabla, separados por un punto. Por ejemplo la tabla MATERIA del USER1 se referenciaría así:

USER1.MATERIA

Cuando se crea una nueva tabla con la sentencia CREATE TABLE el usuario que la crea se convierte en su propietario. El propietario de una tabla esta automáticamente autorizado a consultarla, actualizarla e incluso destruirla. Los demás usuarios no tienen inicialmente privilegios sobre la nueva tabla, el propietario debe de concederles los permisos necesarios.

Al construir una nueva vista, se debe tener permiso, por lo menos de consulta sobre las tablas reales que definen la vista. El propietario de una vista esta autorizado a consultarla y destruirla, pero no siempre a actualizarla. Si se está autorizado a actualizar las tablas o vistas reales, también tendrá los privilegios de actualización sobre la vista. Si ademas esta autorizado para transmitir sus privilegios sobre las tablas reales, también podrá transmitir las que tenga sobre la vista.

Para concretar estas ideas a continuación se verán los formatos de las sentencias de SQL que sirven para transmitir y revocar autorizaciones sobre tablas y vistas.

4.3.3. Concesión de privilegios.

El propietario de una tabla o vista, puede usar la sentencia GRANT de SQL, para transmitir a otros usuarios sus autorizaciones sobre ella.

Sintaxis:

GRANT lista de privilegios
    ON lista de objetos
    TO lista de usuarios
    [WITH GRANT OPTION]

Donde:

Lista de privilegios, puede ser:

SELECT
INSERT
DELETE
UPDATE [(lista de columnas)]
ALTER
INDEX.
ALL PRIVILEGES

Lista de objetos:

nombres de las tablas o vistas.

Lista de usuarios:

lista de los identificadores de los usuarios / PUBLIC

Ejemplo: Se desea otorgar a Juan permiso para visualizar la tabla MATERIA.

```
GRANT SELECT
ON Materia
TO Juan
```

Si se desea autorizar a todos los usuarios del DBMS se utiliza la palabra clave PUBLIC en lugar de la lista de usuarios a los que se desea conceder privilegios.

Ejemplo: Dar a todos los usuarios acceso de SELECT a la tabla CARRERA

```
GRANT SELECT
ON Carrera
TO PUBLIC
```

En caso de querer conceder todos los privilegios se utiliza ALL PRIVILEGES en lugar de la lista de privilegios.

Ejemplo: Dar todos los privilegios sobre la tabla MATERIA a Juan.

```
GRANT ALL PRIVILEGIES
ON Materia
TO Juan
```

El privilegio UPDATE puede restringirse a solamente unas columnas. Las columnas cobre las cuales se permiten realizar actualizaciones van después de la palabra clave UPDATE, y se encierran entre paréntesis.

Ejemplo: Permitir actualizar el costo del laboratorio y número de créditos en la tabla MATERIA, al usuario COORD1.

```
GRANT UPDATE (Mcostolab, Cred)
ON Materia
TO Coord1
```

Cuando el propietario de una tabla o vista concede privilegios a otro usuario, el usuario que recibe los privilegios no puede conceder, a su vez, los privilegios a otros usuarios. Si se desea permitir que otro usuario sea capaz de ello, se debe especificar WITH GRANT OPTION al final de la sentencia GRANT.

Ejemplo: Otorgar a COORD1 y COORD2 permisos para visualizar y actualizar el costo del laboratorio y número de créditos en la tabla MATERIA. Ellos podrán pasar los privilegios a otros usuarios.

```
GRANT SELECT, UPDATE (Mcostolab, Cred)
ON Materia
TO Coord1, Coord2
WITH GRANT OPTION
```

4.3.4. Retirar privilegios.

Cualquier privilegio otorgado puede ser retirados después, ejecutando la sentencia REVOKE. La sintaxis es similar a la sentencia GRANT.

Sintaxis:

```
REVOKE lista de privilegios
    ON lista de objetos
    FROM lista de usuarios
```

Donde:

Lista de privilegios, puede ser:
SELECT
INSERT
DELETE
UPDATE [(lista de columnas)]
ALTER
INDEX.
ALL PRIVILEGES

Lista de objetos:
nombres de las tablas o vistas.

Lista de usuarios:
lista de los identificadores de los usuarios / PUBLIC

Ejemplo: El usuario Coord1 ha abusado de su privilegio de actualización sobre la tabla Materia. Retirarle los privilegios de UPDATE, pero dejarle el de SELECT.

```
REVOKE UPDATE
ON Materia
FROM Coord1
```

Después de la ejecución de esta sentencia, el sistema rechazará cualquier sentencia UPDATE que trate de ejecutar el usuario COORD1. Al no especificar los nombres de las columnas después de la palabra UPDATE, el sistema revocara todos los privilegios UPDATE sobre todas las columnas.

Supóngase que COORD1, concedió privilegios UPDATE a otros usuarios, y estos a otros más; en este caso, la sentencia REVOKE se aplicara automáticamente a todos esos usuarios. El efecto recae sobre todos los usuarios que recibieron privilegios de UPDATE, para la tabla MATERIA, por el usuario COORD1.

También en la sentencia REVOKE son permitidos las palabras reservadas ALL PRIVILEGES y PUBLIC, que se estudiaron en la sentencia GRANT.

Ejemplo: Quitar todos los privilegios sobre la tabla Materia, que fueron concedidos a todos los usuarios.

```
REVOKE ALL PRIVILEGES
ON Materia
FROM PUBLIC
```

5.PROGRAMACIÓN CON SQL.

Los formatos y opciones de sentencias SQL que se han presentado hasta ahora, se pueden utilizar en peticiones que se introducen en forma interactiva, sin necesidad de conocer lenguajes adicionales a SQL. Esto hace posible que, con una breve capacitación, SQL pueda ser utilizado por usuarios no profesionales en el área de informática.

SQL ofrece la posibilidad de incluir sentencias de SQL dentro de un programa, lo que requerirá conocer un lenguaje de programación adicional. Si habla entonces de *SQL incorporado* a otro programa. Esta forma de uso implica también conocimientos de procesos auxiliares para la captura, depuración, compilación y preparación del programa, propios de la labor de programación y complementarios al SQL.

Esta parte se dedica a describir el uso de SQL incorporado y esta orientado a los programadores de aplicaciones. Se presupone que el programador esta relacionado con algún lenguaje de programación y con los conceptos relacionados con la puesta a punto de los programas.

Se considerarán únicamente dos lenguajes de programación para explicar el uso de SQL incorporado: COBOL y C. Aunque existen otros lenguajes donde se puede incorporar SQL, los conceptos que se verán son análogos en todos ellos y las diferencias son más bien del lenguaje de programación que de la forma de incorporar el SQL. Se eligió COBOL dado que es un lenguaje de programación ya muy generalizado en las empresas para sus aplicaciones administrativas. El lenguaje C se eligió dado que algunos productos SQL están ofreciendo aplicaciones y librerías SQL en este lenguaje. Cuando sea necesario ilustrar las características de ambos lenguajes se mostrarán ejemplos en los dos lenguajes, en caso contrario solamente en uno.

Por último, también se estudiará el *SQL dinámico*, que es una forma del SQL incorporado, pero más avanzada, que es utilizado para construir herramientas de base de datos de propósito general.

5.1. SQL Incorporado.

Como se ha comentado el SQL es un lenguaje de modo dual, se puede usar en modo interactivo y embebido en un lenguaje de aplicación.

Las mismas sentencias que se han visto también pueden utilizarse dentro de un programa, pero esta posibilidad plantea SQL nuevos requisitos, que llevan a ampliar el lenguaje SQL con nuevas sentencias y nuevas opciones en las sentencias ya conocidas, ampliaciones que no tienen sentido en el entorno interactivo. Por otra parte en el lenguaje en el que se incorporan sentencias de SQL, necesitara de algunos elementos de programación nuevos. Estos nuevos aspectos de SQL y los lenguajes de programación en los cuales se incorpora SQL, son los que se estudian en este capítulo, El conjunto de estas nuevos aspectos y la programación de SQL dentro de otro lenguaje se le denomina *SQL incorporado.*

El SQL incorporado requiere de un precompilador especial, proporcionado por el vendedor del producto SQL, que acepta el código fuente combinado, y lo convierte en código ejecutable.

La mayoría de los productos SQL utilizan un enfoque de SQL incorporado y soportan uno o más de los siguientes lenguajes de programación: ADA, C, COBOL, FORTRAN, PASCAL, y PL1. Estos son los lenguajes de programación que soportan SQL incorporado y que más frecuentemente se encuentran en los productos SQL, aunque también se pueden encontrar en algunos productos: APL, ASSEMBLER, BASIC, PROLOG, entre otros.

5.1.1. Usos interactivo e incorporado del SQL.

Cuando se utiliza SQL interactivo, el resultado es mostrado inmediatamente en la pantalla. En el caso de SQL incorporado la sentencia es ejecutada cuando el programa se ejecuta y la lógica de éste así lo decida. En ese momento manda las instrucciones SQL para que el DBMS las ejecute. El resultado es devuelto al programa por el DBMS, para que el programa pueda manipularlo como cualquier variable del programa.

Cuando se usa SQL en forma interactiva, realmente existe un programa que sirve de intermediario entre el usuario y el DBMS. Las instrucciones que escribe el usuario son mandadas por el programa intermediario y mandadas al DBMS para su ejecución. El DBMS regresa el resultado al programa y éste lo envía a su vez a la pantalla, donde el usuario puede verlo.

En ambos casos, en el SQL interactivo y en el SQL incorporado, el DBMS tiene que interactuar con un programa, del que recibe las peticiones de ejecución de sentencias SQL, y la que regresa el resultado de las mismas.

En la mayoría de los programas para ejecutar de manera interactiva las sentencias SQL, se siguen tres pasos:

- Correr el programa que reconoce las sentencias SQL.
- Escribir la sentencia SQL.
- Indicarle al programa que ejecute la sentencia.

Si el DBMS ejecuta la sentencia correctamente, el programa despliega el resultado o un mensaje de que fue satisfactoria la operación, en caso contrario se despliega el mensaje de error correspondiente.

La mayoría de los productos SQL proveen programas generadores de reportes, que complementan la capacidad de ejecución interactiva de SQL, ofreciendo una conjunto de opciones para la presentación de los resultados en forma impresa, incluso algunos productos venden herramientas adicionales para integrar los resultados en gráficos, procesadores de texto y hojas de cálculo.

Por otra parte la posibilidad de usar SQL incorporado proporciona a los programadores toda la potencia de SQL, simplificando la codificación de gran parte de la lógica y disminuyendo la posibilidad de cometer errores, con lo que aumenta notablemente la productividad en el desarrollo de aplicaciones.

5.1.2. Procesamiento de sentencias SQL.

Antes de explicar las técnicas que se emplean en el SQL programado, se verá como el DBMS procesa una sentencia SQL, veamos los pasos que sigue el DBMS:

1. El DBMS analiza la sentencia SQL. Es lo que se conoce como chequeo sintáctico. El DBMS analiza que la sentencia y las opciones de ésta sean validos.

2. El DBMS valida la sentencia. Los errores semánticos son detectados en este paso. Valida que las tablas, columnas y objetos en general existan, así como que los privilegios del usuario le permitan ejecutar la sentencia.

3. El DBMS optimiza la sentencia. Explora las alternativas que existen determina cual es la más optima y la elige.

4. El DBMS crea un plan de ejecución para la sentencia. Este plan es la representación binaria de los pasos que son necesarios para llevar a cabo la sentencia.

5. El DBMS ejecuta la sentencia mediante en plan de ejecución.

El consumo de CPU en cada paso varía, así, el análisis y la validación de la sentencia, típicamente se ejecuta rápidamente. La optimización es un proceso muy intenso que requiere acceso al catálogo de sistema de la base de datos. Para consultas complejas el DBMS examinará varias docenas de alternativas, sin embargo el tiempo que emplea el DBMS en la optimización se ve, por mucho, compensado con el tiempo que se tarda en ejecutar la sentencia.

Cuando la sentencia se ejecuta en modo interactivo, el DBMS recorre los pasos mencionados. Sin embargo cuando se ejecuta la sentencia en modo programado, el análisis, la validación y gran parte de la optimización es ejecutada en *el tiempo de compilación,* es decir cuando el programador compila su programa. Dejando unos pocos pasos para efectuarse en *tiempo de ejecución,* cuando el usuario ejecuta el programa. Todos los DBMS tratan de trasladar el mayor número de pasos el tiempo de compilación, para que una vez que se realiza el programa ejecutable no repita los pasos cada vez que sea ejecutado.

5.1.3. Sentencias SQL ejecutables.

Como los compiladores no entienden las sentencias SQL, hay que traducir éstas, antes de compilar un programa, mediante un *precompilador*, una herramienta de programación. El precompilador examina el programa, encuentra las sentencias SQL y las procesa. Se requiere un precompilador por cada lenguaje de programación donde se incorpore SQL.

El precompilador produce un programa fuente, despojado de las sentencias SQL. En su lugar, el precompilador sustituye llamadas a rutinas del DBMS. Algunas sentencias, llamadas *ejecutables*, son almacenadas por el precompilador en un módulo aparte, que tras un proceso de vinculación (BIND), que analiza, valida y optimiza las sentencias, se da lugar a un módulo llamado plan, en el que figuran todas las sentencias SQL ejecutables. El programa BIND se encarga de almacenar el plan en la base de datos, asignándole el nombre del programa de aplicación que creó el plan.

Las *sentencias ejecutables* son las que forman parte del plan y tienen alguna tarea con el DBMS. Existe otro tipo de sentencias SQL que son conocidas como *sentencias no ejecutables* y son tratadas, explícitamente por el precompilador. Las sentencias vistas hasta ahora son todas sentencias ejecutables. A manera de ejemplo se pueden citar las siguientes sentencias no ejecutables, que aún no han sido explicadas: DECLARE, INCLUDE, WHENEVER, EXEC SQL y END-EXEC.

Por otro lado, el programa fuente después de haber sido modificado por el precompilador, es sometido a los procesos habituales de compilación y preparación, hasta producir un programa ejecutable.

Cuando un programa se ejecuta, él y el plan se cargan en la memoria. Cada vez que el programa se topa con una sentencia SQL ejecutable, que se encuentra en el plan, se transfiere el control al DBMS, que ejecuta los pasos del plan y devuelve el resultado y el control al programa.

El programa ejecutable y el plan deberán ser coherentes, cada vez que el programa ejecutable transfiera el control al DBMS para ejecutar una sentencia SQL, la sentencia se debe de encontrar en el plan. Para asegurar esto el programa ejecutable y el plan, deben de construirse de la salida de la misma precompilación.

En particular, el precompilador maneja el análisis de las sentencias SQL, y la utilidad BIND se encarga de la verificación, la optimización y la generación del plan.

El proceso entrega dos productos: un programa ejecutable, almacenado en un archivo con el mismo formato que cualquier otro programa ejecutable; y un plan de aplicación ejecutable, almacenado dentro de la base de datos en el formato que el DBMS maneja.

La siguiente figura muestra el proceso para desarrollar un programa con SQL incorporado.

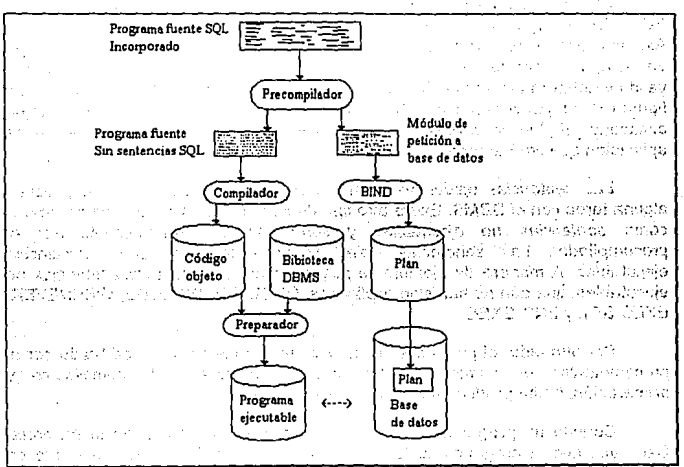


Fig. 5.1. Proceso de desarrollo de un programa con SQL incorporado.

5.1.4. Sentencias de SQL estáticas y dinámicas.

Como se vió, el optimizador es un componente muy importante de la herramienta BIND, es el responsable de decidir la estrategia de accesos físicos a emplear para obtener el resultado correspondiente a una sentencia SQL. Frecuentemente existen caminos alternativos para acceder a los mismos datos. El optimizador es el responsable de evaluar las diversas alternativas y elegir la más eficiente. Todas las sentencias deben de ser optimizadas antes de poder ser ejecutadas. El proceso de optimización puede ser costoso, pues se basa en algoritmos sofisticados, es recomendable evitar ejecuciones innecesarias del mismo.

Por las razones expuestas, es recomendable, siempre que sea posible, el proceso de optimización se debe hacer antes de la ejecución del programa, en el proceso de vinculación (BIND), en este caso los accesos se almacenan en el plan, en la base de datos.

Sin embargo esto no siempre es posible, si la sentencia es generada dinámicamente en el programa, por ejemplo que se le de oportunidad al usuario de escribir la condición de una cláusula, entonces en el paso de vinculación no se podrá optimizar la sentencia. En este caso no queda más remedio que optimizar la sentencias durante la ejecución del programa.

Las sentencias que se optimizan en el tiempo de ejecución se denominan sentencias de optimización dinámica, o para abreviar *sentencias dinámicas*. Por el contrario las que se optimizan en el tiempo de vinculación se denominan *sentencias estáticas*. La cualidad de que sean estáticas o dinámicas dependerá de la forma en que se codifiquen y utilicen por parte del programador.

La mayoría de las aplicaciones pueden resolverse con sentencias estáticas. Sin embargo hay aplicaciones donde se requieren sentencias dinámicas. Por ejemplo el programa que traduce las sentencias en forma interactiva, no tiene manera de saber cual es la siguiente sentencia que se escribirá, por tanto, no queda más remedio que optimizar las sentencias durante su ejecución.

En una sección próxima en este mismo capítulo se trata más extensamente cómo pasan los programas al DBMS una sentencia dinámica y cómo pedirle que la optimice y ejecute. Por el momento solo se analizan las sentencias estáticas.

5.1.5. Elementos de un programa con SQL incorporado.

La inclusión de sentencias SQL dentro de un programa requiere que se proporcionen algunos nuevos elementos de programación que permitan que existan dentro del programa sentencias escritas en un lenguaje ajeno al de programación y soporten la comunicación de peticiones y resultados con el DBMS. Estos nuevos elementos, son entre otros:

- *Delimitadores.* Son añadidos a la sintaxis de las sentencias SQL para distinguirlas dentro de un programa fuente.
- *Área de comunicación.* Es una área de datos definida en el programa, que es utilizada por el DBMS para recibir y regresar valores a éste.
- *Variables de programa.* Las variables que contienen los datos para interactuar con la base de datos. Contienen los valores de las columnas que se escriben o leen.

5.1.5.1. Delimitadores.

Los compiladores de los lenguajes de programación no entienden las sentencias SQL como instrucciones propias del lenguaje. Por lo que es necesaria encerrarlas entre delimitadores para distinguirlas dentro del programa fuente y que permitir posteriormente que el precompilador las reconozca.

Los *delimitadores* son uno a más caracteres que marcan el comienzo y final de una sentencia SQL en programa fuente y varían con el lenguaje de programación.

Lenguaje COBOL.

En COBOL, el delimitador de comienzo es EXEC SQL y el delimitador de final es END-EXEC. Los productos SQL utilizan estos delimitadores para la mayoría de sus lenguajes anfitriones.

Sintaxis:

```
EXEC SQL
     sentencia-SQL
END-EXEC
```

El EXEC SQL es tomado como una sola instrucción y sigue las reglas de las instrucciones en COBOL, debe de empezarse a escribir a partir de la columna 12.

La sentencia SQL puede escribirse entre las columnas 12 y 72, en una o varias lineas. Ademas los delimitadores y la sentencia se pueden escribir en una sola línea.

```
EXEC SQL  sentencia-SQL  END-EXEC
```

Lenguaje C.

En C el delimitador de comienza es EXEC SQL y el de fin es un punto y coma (;):

```
EXEC SQL
     sentencia-SQL;
```

El lenguaje C es un lenguaje que distingue mayúsculas y minúsculas. Sin embargo los identificadores de objetos de la base de datos, como pueden ser nombres de tablas, vistas, índices y columnas, así como las palabras reservadas de SQL deben de escribirse con mayúsculas.

Al igual que en COBOL el delimitador de inicio EXEC SQL debe de escribirse en una sola línea.

5.1.5.2. Área de comunicación del SQL.

Cuando se escribe una sentencia en SQL interactivo que provoca un error, el programa SQL interactivo visualiza un mensaje de error y aborta la sentencia, permitiendo volver a escribirla. En SQL incorporado, el manejo de errores pasa a ser responsabilidad del programa de aplicación.

El DBMS informa de los errores en tiempo de ejecución al programa de aplicación a través del Área de Comunicaciones del SQL (en inglés: SQLCA : SQL *Communication Area*).

El SQLCA se define dentro del programa, según como se definan los datos en cada lenguaje. Es un área en la que el DBMS, almacenará después de cada sentencia SQL, la información necesaria para que el programa sepa si la sentencia fue ejecutada correcta o incorrectamente. En el último de los casos proporciona información para conocer la causa que impidió su correcta ejecución.

Como el SQLCA esta definido en el programa, no hay ningún impedimento para accesar su contenido y manejar sus campos con las sentencias del lenguaje. Normalmente después de cada sentencia SQL, se incluyen en el programa sentencias para preguntar por los campos del SQLCA y saber si la sentencia concluyó bien o no.

La información que contiene cada campo es muy amplia, solamente se verá lo que contiene cada campo de manera genérica, pues los códigos de error se pueden encontrar en los manuales del producto y explicarlos aquí sería, efectivamente transcribir parte de un manual de errores. Los campos que definen al SQLCA son:

- SQLCAID. Campo alfanumérico de 8 caracteres, contiene la literal SQLCA.
- SQLCABC. Tiene por longitud una palabra (2 Bytes). Contiene la longitud de la SQLCA (136).
- SQLCODE. Mide una palabra. Es el código de retorno de la última sentencia SQL ejecutada. Es un número y permite saber si se realizó con éxito. Para ella se sigue el siguiente convenio:

  Si el código de retorno es cero:

  La sentencia se ejecutó correctamente.

Si el código de retorno es un valor negativo:

Se produjo un error serio que impidió la ejecución de la sentencia.

Si el código es un error positivo:

La sentencia se ejecutó, pero produjo un aviso, Un aviso puede ser por ejemplo que se produjo un truncamiento de los valores recuperados. O que no encontró ninguna fila que satisfaga la condición, sin embargo, la sentencia se ejecutó de manera satisfactoria.

- SQLERRML. Media palabra que contiene la longitud del campo SQLERRMC.
- SQLERRMC. Es un campo de longitud variable (70 caracteres máximo) que en caso de que se produzca un error, contiene la descripción de este error.
- SQLERRP. En caso de que se haya producido un error, contiene el nombre del módulo del DBMS donde se detectó. Es información de diagnostico para del DBMS y es específico en cada producto.
- SQLERRD. Es un vector de seis elementos, cada uno de los cuales es una palabra. Contiene varias cuentas. Por ejemplo el SQLERRD(3), contiene el número de filas modificadas por la última sentencia de actualización.
- SQLWARN. Es un vector de ocho elementos, cada uno de los cuales puede ser blanco o "W". En caso de contener una "W" significa que se detecto una anomalía que no impide la ejecución de la sentencia pero podría ser un error. Estos campos son conocidos como indicadores de condición de aviso (*warning conditions*).

  Aquí se mencionan los mas interesantes, contienen una "W" si:

  SQLWARN0: Si cualquiera de los restantes indicadores contiene una "W".

  SQLWARN1: Si el valor de una columna alfanumérica ha sido truncada al traerla al programa.

  SQLWARN2: Si hay valores nulos en una función de columna.

  SQLWARN3: Si el número de columnas seleccionadas con SELECT es diferente al número de variables que recibirán sus valores.

  SQLWARN4: Si una sentencia UPDATE o DELETE no contienen cláusula WHERE.

- SQLEXT. Ocho Bytes reservados por el sistema.

De los campos del SQLCA el más utilizado es el código de retorno SQLCODE.

La definición de esta área es obligatoria si el programa contiene sentencias SQL ejecutables. La forma en que se define depende de cada lenguaje. Dado que siempre que se use SQL incorporado se tendrá que definir el área de comunicación, SQL tiene la gentileza de facilitar al programador una sentencia especial para su definición, la sentencia INCLUDE.

5.1.5.3. Sentencia INCLUDE.

Permite incluir en un programa la definición de áreas de datos que se utilizan para intercambiar información entre el programa y el DBMS. Existen dos de estas áreas: la SQLCA y SQLDA. La segunda se describirá cuando se explique la programación de sentencias SQL dinámicas. De momento solo se considera SQLCA. Es una sentencia no ejecutable.

Sintaxis:

INCLUDE {SQLCA / SQLDA}

La sentencia, en el momento de precompilación será sustituida por la definición de la estructura de SQLCA, con la codificación adecuada al tipo de lenguaje que se utilice. El lugar donde se debe colocar la sentencia depende del lenguaje de programación utilizado.

Lenguaje COBOL.

La SQLCA se define en la WORKING STORAGE SECTION de la DATA DIVISION en cualquier lugar donde se pueda definir un registro. Recordar que el delimitador EXEC SQL debe de comenzar en la columna 12.

Ejemplo:

```
IDENTIFICATION DIVISION.
PROGRAM ID.
*************************
ENVIRONMENT DIVISION.
*************************
CONFIGURATION SECTION.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
*************************
DATA DIVISION.
FILE SECTION.
*
WORKING STORAGE SECTION.
*************************
*DECLARACIÓN DE SQLCA
*************************
        EXEC SQL INCLUDE SQLCA END-EXEC.
```

Cuando se precompila el programa, la sentencia INCLUDE se pone como comentario y es sustituida por la estructura que se muestra:

```
IDENTIFICATION DIVISION.
PROGRAM ID.
*************************
ENVIRONMENT DIVISION.
*************************
CONFIGURATION SECTION.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
*************************
DATA DIVISION.
FILE SECTION.
*
WORKING STORAGE SECTION.
*************************
*DECLARACIÓN DE SQLCA
*************************
*EXEC SQL INCLUDE SQLCA END-EXEC.
*************************
01 SQLCA.
   02 SQLCAID          PIC X(08).
   02 SQLCABC          PIC S9(09) COMP-4.
   02 SQLCODE          PIC S9(09) COMP-4.
   02 SQLERRML         PIC S9(09) COMP-4.
   02 SQLERRMC         PIC X(70).
   02 SQLERRP          PIC X(08).
   02 SQLERRD          OCCURS 6 TIMES
                       PIC S9(09) COMP-4.
   02 SQLWARN.
      03 SQLWARN0      PIC X(01).
      03 SQLWARN1      PIC X(01).
      03 SQLWARN2      PIC X(01).
      03 SQLWARN3      PIC X(01).
      03 SQLWARN4      PIC X(01).
      03 SQLWARN5      PIC X(01).
      03 SQLWARN6      PIC X(01).
      03 SQLWARN7      PIC X(01).
   02 SQLEXT           PIC X(08).
.....
```

La definición de la SQLCA en C no tiene posición fija. Sólo se debe tener en cuenta que debe estar colocada en el módulo fuente antes de cualquier sentencia SQL ejecutable.

Ejemplo:

```
#include stdio.h
#include stdlib.h
#include string.h
EXEC SQL INCLUDE SQLCA;
```

Cuando se precompila el programa, la sentencia INCLUDE se pone como comentario y es sustituida por la estructura que se muestra:

```
#include stdio.h
#include stdlib.h
#include stdio.h
#include string.h
/*EXEC SQL INCLUDE SQLCA:*/
struct sqlca {
        unsigned char  sqlcaid[8];
        long           sqlcabc;
        long           sqlcode;
        short          sqlerrml;
        unsigned char  sqlerrmc[70];
        unsigned char  sqlerrp[8];
        long           sqlerrd[6];
        unsigned char  sqlwarn[8];
        unsigned char  sqlext[8];
}
#define SQLCODE sqlca.sqlcode     /*código de estado SQL */
#define SQLWARN0 sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]
#define SQLWARN2 sqlca.sqlwarn[2]
#define SQLWARN3 sqlca.sqlwarn[3]
#define SQLWARN4 sqlca.sqlwarn[4]
#define SQLWARN5 sqlca.sqlwarn[5]
#define SQLWARN6 sqlca.sqlwarn[6]
#define SQLWARN7 sqlca.sqlwarn[7]
```

5.1.5.4. Sentencia WHENEVER.

Cuando el DBMS termina de ejecutar una sentencia SQL, el programa deberá preguntar por el valor de SQLCODE para verificar la correcta ejecución de una sentencia SQL incorporada.

Para comprobar el valor de SQLCODE se pueden emplear las sentencias del lenguaje. Por ejemplo IF...THEN de COBOL. Pero para un programador puede resultar tedioso estar verificando la correcta ejecución después de cada sentencia. Para simplificar el manejo de errores, SQL proporciona la sentencia WHENEVER, que facilita la comprobación después de cada sentencia SQL ejecutada.

La sintaxis de la sentencia WHENEVER es:

```
WHENEVER {SQLERROR/SQLWARNING/NOT FOUND}
         {CONTINUE/GOTO etiqueta}
```

Al procesar una sentencia WHENEVER, el precompilador incluye después de cada sentencia ejecutable las instrucciones necesarias para la comprobación y acción que en ella se indiquen. Estas instrucciones se incluyen en las sentencias que se encuentran después de la sentencia WHENEVER.

Se puede utilizar la sentencia WHENEVER para que el precompilador genere código en cualquiera de las tres siguientes condiciones :

- WHENEVER SQLWARNING, tiene efecto cuando el SQLCODE tiene un valor positivo, diferente de 100, que tiene un trato especial.
- WHENEVER SQLERROR, cuando se comprueba que el SQLCODE tiene un valor negativo.
- WHENEVER NOT FOUND, el precompilador genera código cuando el SQLCODE tiene un valor de 100. Este valor indica que se ha dado cuando el DBMS trata de recuperar datos en una consulta, y no encuentra datos que recuperar. Este uso de la sentencia WHENEVER es exclusivo de la sentencia SELECT.

Para cualquiera de las tres condiciones se pueden especificar la siguientes acciones:

- WHENEVER/CONTINUE, indica que se ignore si se da o no la condición, el programa continua a la siguiente sentencia del lenguaje anfitrión.
- WHENEVER/GOTO le dice al precompilador que genere código para dar una salto a la *etiqueta* especificada, en donde se supone que se tratará el error de la manera que el programador considere.

La sentencia WHENEVER es una orden al precompilador, y su efecto puede ser cambiado por la orden de otra sentencia WHENEVER que se encuentre posteriormente en el programa.

Lenguaje COBOL.

Ejemplo:

```
01 SALIDA.
   02 ERROR        PIC S9(9).
   05 MENSAJE      PIC X(70).
      .....
   PROCEDURE DIVISION.
      .....
      EXEC SQL
      WHENEVER SQLERROR GO TO TRAT-ERROR
      END EXEC
      .....
   TRAT-ERROR.
      MOVE SQLCODE TO ERROR
      MOVE SQLERRD  TO MENSAJE
      DISPLAY SALIDA
      STOP RUN.
```

El programa se irá al párrafo TRAT-ERROR cada vez que se produzca un error serio que impida la ejecución de la sentencia, desplegará el SQLCODE y el mensaje correspondiente a ese error, el programa termina al entrar en este párrafo.

Ejemplo.

```
#include stdio.h
#include stdlib.h
#include string.h
EXEC SQL INCLUDE SQLCA;
    .....
EXEC SQL WHENEVER SQLERROR goto error1;

EXEC SQL DELETE FROM MATERIA
    WHERE MCOSTOLAB = 100.00;

EXEC SQL DELETE FROM ACADEMIC
    WHERE ANOEMPLEAD = 75521245;

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL UPDATE ACADEMIC
    SET ASUELDO = ASUELDO * 1.30;

EXEC SQL WHENEVER SQLERROR goto error2;

EXEC SQL INSERT INTO MATERIA
        VALUES ("0777", "Sistemas Operativos", 8, 0.00, 32);
    .....
error1:
    printf("SQL DELETE error: %d\n",sqlca.sqlcode);
    exit();

error2:
    printf("SQL INSERT error: %d\n",sqlca.sqlcode);
    exit();
```

En este ejemplo, en cualquiera de las dos sentencias **DELETE** se da lugar a un salto a error1. Si se produce un error en la sentencia **UPDATE** es ignorado. Un error en la sentencia **INSERT** da lugar a un salto a error2. Como se ve, **WHENEVER/CONTINUE** sirve para cancelar el efecto de la sentencia **WHENEVER** anterior.

La sentencia WHENEVER hace que el manejo de errores sea mucho más simple en SQL incorporado y es más común en un programa de aplicación que la comprobación de SQLCODE con sentencias del lenguaje anfitrión.

5.1.5.5. Uso de sentencias SQL con variables del programa.

Dentro de una sentencia incorporada, puede especificarse el nombre de una variable, no SQL, definida en el programa. En muchas aplicaciones puede ser útil usar una o más variables en las sentencias SQL, por ejemplo si se desea cambiar el número de créditos de algunas carreras, el programa deberá pedir el número de materia, para que sea usado en la cláusula SELECT y posteriormente en la cláusula UPDATE.

Una *variable huésped* es una variable de programa declarada en el lenguaje anfitrión, que es referenciada en una sentencia de SQL incorporado. Para que el precompilador pueda reconocer en la sentencia que se usa una variable del programa y no crea que se trata de un objeto de la base de datos (tablas o columnas, por ejemplo) , la variable deberá de ir precedida por dos puntos (:). Al utilizar la variable fuera de la sentencia, se utiliza, como cualquier variable del programa, sin los dos puntos.

Las variables huésped pueden utilizarse en cualquier parte de la sentencia SQL donde pueda especificarse una constante.

Cuando se especifica una variable huésped en la cláusula VALUES de la sentencias INSERT, su contenido se inserta en la columna correspondiente de la tabla. Los valores recuperados mediante una sentencia SELECT se almacenan en variables huésped para que el programa los procese como crea conveniente. En resumen las variables huésped contienen los valores que el programa lee o escribe en las tablas.

Las variables huésped solamente pueden contener valores, no pueden contener nombres de objetos como tablas, vistas o columnas.

Algunos productos SQL, permiten definir estructuras de datos con variables huésped, de modo que en la sentencia SQL donde haya que especificar una lista de variables, se puede escribir en su lugar, el nombre de la estructura que los contiene. La estructura se especifica en la sentencia, de la misma manera que se especifica cualquier variable huésped.

Ejemplos:

1) Utilización de una variable huésped en una condición.

Lenguaje COBOL.

```
.....
MOVE 0076 TO NUM
EXEC SQL
     DELETE FROM MATERIA
     WHERE NOCARRERA = :NUM
END-EXEC.
**********************
*Obsérvese el uso de los dos puntos
*fuera de la sentencia y dentro de ella
**********************
.....
```

Lenguaje C.

```
.....
num = 0076;
EXEC SQL
     DELETE FROM MATERIA
     WHERE NOCARRERA = :num;
.....
```

2) Uso de una variable huésped en una expresión (COBOL).

Aumentar los sueldos del personal académico en un 30 por ciento.

```
MOVE 1.30 TO AUMENTO
EXEC SQL
UPDATE ACADEMIC
SET ASUELDO = ASUELDO * :AUMENTO
END EXEC
```

3) Utilización de una variable huésped en la cláusula VALUES.

Insertar una nueva materia en la carrera de Computación: Sistemas Operativos, con clave "0777", que valga 8 créditos y no tenga laboratorio.

```
MOVE 32 TO WCARR
   EXEC SQL
   INSERT INTO MATERIA
         VALUES ("0777", "Sistemas Operativos", 8, 0.00, :WCARR);
```

Para poder usar una variable del programa en una sentencia SQL, la variable debe estar previamente definida. Para ello se utilizan nuevas sentencias, que se ven a continuación.

5.1.5.6. Definición de variables huésped.

Las variables huésped se definen dentro del programa, como cualquier otra variable en el lenguaje de programación anfitrión, antes de que cualquier sentencia SQL las use. Deben de aparecer dentro de las sentencias SQL: **BEGIN DECLARE SECTION** y **END DECLARE SECTION**. Estas dos sentencias son propias del SQL incorporado y nos son ejecutables. Son indicaciones para el precompilador para que sepa que se trata la definición de una variable huésped.

Lenguaje COBOL.

La sección **DECLARE** debe aparecer en la **DATA DIVISION**, específicamente en la **WORKING STORAGE SECTION** o en la **LINKAGE SECTION**. Las variables huésped pueden recibir cualquier nombre válido en COBOL.

Lenguaje C.

La sección **DECLARE** puede aparecer en cualquier parte del programa, pero siempre antes de que se utilice la variable huésped en alguna sentencia SQL. Las variables huésped pueden recibir cualquier nombre válido en C.

5.1.5.7. Tipos de datos para las variables huésped

En el capítulo uno se vieron los tipos de datos que soportan los diferentes productos SQL. Por otro lado, en el programa hay que definir las variables huésped según la sintaxis de cada lenguaje y los tipos de datos soportados son a menudo bastante diferentes. En la siguiente tabla se muestran los tipos de datos de COBOL y C que corresponden a los tipos de datos en SQL.

SQL	C	COBOL
SMALLINT	short int	PIC S9(4) COMP
INTEGER	int	PIC S9(4) COMP
FLOAT	float	COMP -1
DOUBLE	double	COMP -2
DECIMAL(p,s)	double	PIC S9(p - s)V9(S) COMP-3
MONEY(p,s)	double	PIC S9(p - s)V9(S) COMP-3
CHAR(n)	char[n+1]	PIC X(n)

VARCHAR(n)	struct {
	short int;
	char [n+1]
	}
		01 X.
		02 LEN PIC S9(4)COMP
		02 DATA PIC X(n)

DATE	char x[12]	PIC X(11)
TIME	char x[28]	PIC X(27)

Tabla 5.1. Tipos de datos en SQL y en los lenguajes COBOL y C.

Lenguaje COBOL.

El tipo de datos VARCHAR(N) de SQL es equivalente en COBOL a una estructura con dos componentes, en uno la longitud y el otro el valor de la cadena. Realmente será una cadena de longitud fija, sólo que en el primer byte el DBMS regresa la longitud de la cadena.

Ejemplo: Insertar una nueva materia en la carrera de Computación: Sistemas Operativos, con clave "0777", que valga 8 créditos y no tenga laboratorio.

Las variables huésped se definen así:

```
DATA DIVISION.
*
WORKING STORAGE SECTION.
*****************************
     EXEC SQL BEGIN DECLARE SECTION END EXEC.
     01 WNOMATERIA          PIC X(04).
     01 WNOMBRE.
        02 WNOMBREL         PIC S9(4) COMP.
        02 WNOMBREV         PIC X(30).
     01 WCRED               PIC S9(02).
     01 WCOSTOLAB           PIC S9(07)V9(02) COMP-3.
     01 WNOCARRERA          PIC S9(02).
     EXEC SQL END DECLARE SECTION END EXEC.
```

La definición que se presenta en la siguiente tabla corresponde a la definición en SQL, de la tabla **MATERIA:**

Nombre del campo	Tipo
MNOMATERIA	CHAR(04)
MNOMBRE	VARCHAR(30)
MCRED	SMALLINT
MCOSTOLAB	DECIMAL(7,2)
MNOCARRERA	SMALLINT

Tabla 5.2 Definición de la tabla MATERIA.

Ejemplo: Insertar una nueva materia en la carrera de Computación: Sistemas Operativos, con clave "0777", que valga 8 créditos y no tenga laboratorio.

```
     PROCEDURE DIVISION:
     100-INICIO.
          MOVE "0777"              TO WNOMATERIA
          MOVE 19                  TO WNOMBREL.
          MOVE "Sistemas Operativos" TO WNOMBREV
          MOVE 8                   TO WCRED
          MOVE 0.00                TO WCOSTOLAB
          MOVE 32                  TO WNOCARRERA
```

Lenguaje C.

El lenguaje C se observa que un CHAR(n) en SQL corresponde a char[n+1], esto debido a en C la cadena de caracteres termina con un valor NULL.

En C se da el mismo caso para el tipo de dato SQL VARCHAR(n), la definición equivalente se muestra en la tabla 5.1. En la parte entera se guarda la longitud de la cadena y en la parte alfanumérica se retorna o manda la cadena.

El lenguaje C no soporta el manejo de decimales, se usa la conversión a coma flotante pero puede haber truncamiento de datos.

Los tipos de datos de fecha/hora se convierten a cadenas de caracteres en los formatos "dd-mmm-aaaa","hh:mm:ss" y "dd-mmm-aaaa hh:mm:ss:cccccc", este último para los productos que soportan el tipo TIMESTAMP.

5.1.5.8. Variables huésped y valores NULL.

La mayoría de los lenguajes de programación no soportan los valores NULL. Cuando se desea insertar un valor NULL en una columna, la variable correspondiente no puede tener un valor de cero o espacios, pues se insertaría ese valor y no el nulo deseado.

Para evitar el problema al almacenar y recuperar valores NULL, SQL permite que cada variable huésped tenga una *variable indicadora* asociada. Una variable indicadora tiene media palabra de longitud. La variable huésped y la variable indicadora, indican juntas un valor de acuerdo al siguiente convenio:

- Un valor de cero, o positivo en la variable indicadora indica que la variable huésped contiene un valor válido y que ese valor puede ser utilizado.
- Si la variable indicadora contiene un negativo significa que la variable huésped debería de tener un valor NULL, el valor de la variable huésped debe de ser despreciado.

Las variables indicadores, al igual que las variables, se usan dentro de las sentencias SQL, con su nombre precedido de dos puntos (:). Para definirse siguen las mismas reglas de las variables huésped.

Para establecer que una variable se utiliza como variable indicadora, hay que escribir el nombre de la variable a continuación del nombre de la variable huésped con la cual se asocia. La asociación variable huésped / variable indicadora se establece únicamente en la sentencia donde aparecen juntos.

Lenguaje COBOL.

Como la variable indicadora se define como un campo numérico de media palabra se define como PIC S9(04) COMP -4.

Ejemplo: Se ha dado de baja el laboratorio de la materia de "Bases de datos" y se desea actualizar el costo del laboratorio para que contenga un valor NULL.

```
DATA DIVISION.
     .....
WORKING STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END EXEC.
     01 WNOMATERIA          PIC X(04).
     01 WNOMBRE.
        02 WNOMBREL         PIC S9(4) COMP.
        02 WNOMBREV         PIC X(30).
     01 WCRED               PIC S9(02).
     01 WCOSTOLAB           PIC S9(07)V9(02) COMP-3.
*****************************
*INDICADOR PARA COSTOLAB
*****************************
     01 INDCOSTOLAB         PIC S9(04) COMP -4.
     01 WNOCARRERA          PIC S9(02).
     EXEC SQL END DECLARE SECTION END EXEC.
     .....
PROCEDURE DIVISION.
     .....
     MOVE -1                TO INDCOSTOLAB
     MOVE 14                TO WNOMBREL
     MOVE "Bases de datos"  TO WNOMBREV
     EXEC SQL
     UPDATE MATERIA
          SET MCOSTOLAB = :WCOSTOLAB :INDCOSTOLAB
          WHERE MNOMBRE = :WNOMBRE
     END-EXEC.
```

Es practica frecuente mandar para una columna con tipo de dato SQL VARCHAR, una variable de longitud fija, y el DBMS guardará el valor con la longitud de la variable huésped.

Recuérdese que la sentencia UPDATE admite especificar el valor nulo directamente, por lo que la sentencia mostrada en el ejemplo seria equivalente a:

```
EXEC SQL
UPDATE MATERIA
        SET MCOSTOLAB = NULL
        WHERE MNOMBRE = :WNOMBRE
END-EXEC.
```

Lenguaje C.

El mismo ejemplo se presenta ahora en lenguaje C.

```
        .....
EXEC SQL BEGIN DECLARE SECTION;
double wcostolab;
short int indcostolab;      /*indicador para costolab*/
struct {short int wnombrel;
        char wnombrev[30];
        }wnombre;
EXEC SQL END DECLARE SECTION;
        .....
indcostolab = -1
wnombrel = 14
wnombrev = "Bases de datos"
EXEC SQL
UPDATE MATERIA
        SET MCOSTOLAB = :wcostolab :indcostolab
        WHERE MNOMBRE = :wnombre;
```

En el ejemplo en COBOL se definió la estructura completa de la tabla MATERIA, en C solamente se difinieron las variables que se utilizan para el ejemplo, ésto es algo que queda a consideración del programador, y no afecta en la ejecución de la sentencia.

5.1.6. Recuperación de datos en SQL incorporado.

La recuperación de datos con SQL incorporado requiere de opciones especiales en la sentencia SELECT. La principal razón para éstas nuevas opciones es la disparidad entre el lenguaje SQL y los lenguajes de programación donde se incorpora: una consulta puede producir una tabla entera como resultado de una consulta, sin embargo, los lenguajes de programación solamente tienen capacidad de manejar registros (filas).

SQL divide las consultas en dos grupos:

- *Consultas de una fila.* En donde se espera que los resultados de la consulta tengas una sola fila de datos. Generalmente son consultas que se accesar por su llave primaria.
- *Consultas multitabla.* En donde se espera que los resultados de la consulta puedan contener cualquier número de filas, inclusive cero filas y una fila.

En SQL los dos tipos de consulta son manejados de manera diferente.

5.1.6.1. Consultas monofila.

Las sentencias SELECT cuyo resultado es una sola fila, requieren de una nueva cláusula, la cláusula INTO, la sintaxis de la SENTENCIA SELECT singular es la siguiente:

SELECT	lista de selección
INTO	lista de variables huésped
FROM	lista de tablas
WHERE	condición de búsqueda

En estas sentencias, dado que solamente se recupera una fila cada vez que se ejecuta, no pueden especificarse las cláusulas ORDER BY, GROUP BY ni HAVING.

La cláusula INTO va seguida de una lista de variables huésped, donde, tras ejecutar una sentencia SELECT con éxito, los valores de las columnas de la fila recuperada se corresponderán uno a uno con las variables huésped.

Los tipos de datos de las variable huésped especificadas en la cláusula INTO deben de ser compatibles con las correspondientes columnas del resultado, aunque pueden ser diferentes, en cuyo caso el DBMS realiza las conversiones necesarias.

5.1.6.1.1. Manejo de errores en la sentencia SELECT singular.

Como todas las sentencias SQL incorporado. La sentencia SELECT singular establece el valor de SQLCODE para indicar su estado al término de su ejecución :

- Si el resultado de ejecutar la sentencia SELECT es una sola fila. Las variables huésped designadas en la cláusula INTO tienen los valores recuperados y el SQLCODE tiene el valor de cero.
- Si el resultado es una tabla vacía el DBMS no asigna valores a las variable huésped y pone el código de error SQLCODE con valor de 100, que es el aviso especial NOT FOUND.
- Si el resultado es una tabla con más de una fila, el DBMS puede asignar a las variables huésped los valores de cualquiera de las filas recuperadas, pero almacena en SQLCODE un valor negativo.
- Si se produce un error grave las variables huésped no contienen información y el SQLCODE contiene un valor negativo.

5.1.6.1.2. Recuperación de valores NULL.

Para el manejo de valores NULL, SQL incorporado maneja variables indicadoras en la cláusula INTO, de la misma manera que se manejan en la cláusula VALUES de la sentencia INSERT.

Si una de las filas del resultado tiene valor NULL y la correspondiente variable huésped no tiene asociada una variable indicadora, el DBMS pone un código negativo en el SQLCODE y termina la ejecución de la sentencia. En caso de que si tenga una variable indicadora asociada, el DBMS regresa allí un valor negativo y no modifica el contenido de la variable huésped.

Aunque una columna del resultado no sea nula, su correspondiente variable huésped puede estar asociada con una variable indicadora. Entonces el DBMS le dará el valor de cero normalmente, aunque si la columna es truncada al ser asignada a la variable huésped, el DBMS almacena en la variable un número entero positivo que corresponde a la longitud del valor original.

Lenguaje COBOL.

Ejemplo En COBOL, leer el nombre de la materia con clave 0076.

```
.....
WORKING STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END EXEC.
    01 WNOMATERIA          PIC X(04).
    01 WNOMBRE             PIC X(30).
    EXEC SQL END DECLARE SECTION END EXEC.
    .....
PROCEDURE DIVISION.
    .....
    MOVE "0076"       TO WNOMATERIA
    EXEC SQL
    SELECT MNOMBRE
        INTO :WNOMBRE
        FROM MATERIA
        WHERE MNOMATERIA = :WNOMATERIA
    END-EXEC.
```

Obsérvese la variable WNOMBRE que es de longitud fija a pesar de que en SQL esta definida de tipo VARCHAR. El DBMS convierte de un tipo de datos a otro, truncando a rellenando con espacios, si el valor de la variable es mayor o menor a 30 caracteres, respectivamente.

Lenguaje C.

Se muestra el mismo ejemplo, ahora en lenguaje C.

```
.....
EXEC SQL BEGIN DECLARE SECTION;
char wnomateria[05];
char wnombre[30];
EXEC SQL END DECLARE SECTION;
.....
wnomateria = "0076"
EXEC SQL
SELECT MNOMBRE
    INTO :WNOMBRE
    FROM MATERIA
    WHERE MNOMATERIA = :wnomateria;
.....
```

Aquí se presenta un ejemplo adicional en COBOL, que tiene columnas calculadas y maneja variable indicadoras para el manejo de valores NULL.

Ejemplo encontrar los costo de laboratorio mínimo y máximo de la carrera de Computación, en la tabla MATERIA.

```
WORKING STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END EXEC.
     01 WCOSTOMIN          PIC S9(07)V9(02) COMP-3.
     01 INDCOSTOMIN        PIC S9(04) COMP -4.
     01 WCOSTOMAX          PIC S9(07)V9(02) COMP-3.
     01 INDCOSTOMAX        PIC S9(04) COMP -4.
     01 WNOCARRERA         PIC S9(02).
     EXEC SQL END DECLARE SECTION END EXEC.
     .....
PROCEDURE DIVISION.
     .....
     MOVE 32                    TO WNOCARRERA
     EXEC SQL
     SELECT MIN(COSTOLAB), MAX(COSTOLAB)
          INTO :WCOSTOMIN :INDCOSTOMIN,
               :WCOSTOMAX :INDCOSTOMAX
          FROM MATERIA
          WHERE MNOCARRERA = :WNOCARRERA
     END-EXEC.
     .....
```

Como ya se mencionó algunos productos SQL, permiten definir estructuras de datos con variables huésped, de modo que en la sentencia SQL donde haya que especificar una lista de variables, se puede escribir en su lugar, el nombre de la estructura que los contiene. El uso de las estructuras de datos como variables huésped no están soportadas por todos los productos SQL y además también esta restringido a ciertos lenguajes de programación, por ejemplo algunos lo soportan en lenguaje C y no en COBOL o lenguaje ensamblador.

Ejemplo: Aquí se muestra el uso de una estructura de datos como variable huésped. Seleccionar el nombre, los créditos y el costo de laboratorio del curso 0076.

```
main()
{
     EXEC SQL INCLUDE SQLCA;
     EXEC SQL BEGIN DECLARE SECTION;
     char wnomateria[05];
     struct {
          char wnombre[21];          /*nombre de la materia*/
          short int wcred;           /*número de créditos*/
          float wcostolab;           /*costo de laboratorio*/
     } inforep;
     short int ind_rep[3];           /*arreglo de variables para nulos*/
     EXEC SQL END DECLARE SECTION;
```

```
        /*Pide el número de la materia*/
        printf("introduzca la clave de la materia: ");
        scanf("%d",&wnomateria);
        EXEC SQL SELECT MNOMBRE, MCRED, MCOSTOLAB
                        INTO :inforep :ind_rep
                        FROM MATERIA
                        WHERE MNOMATERIA = :wnomateria;
        /*visualiza los datos recuperados*/
        if (sqlca.sqlcode = = 0){
                printf("Nombre: %s\n",inforep.wnombre);
                if (ind_rep[2] < 0)
                        printf("El costo de laboratorio tiene valor NULL\n);
                else
                        printf("Costolab: %f\n",inforep.wcostolab);
                printf("Créditos: %d\n",inforep.wcred);
        }
        else if (sqlca.sqlcode = = 100)
                printf("No existe la materia.\n");
            else
                printf("SQL error: %d\n",sqlca.sqlcode);
        exit();
}
```

5.1.7. Programación con cursores.

Para estudiar la sentencia SELECT en SQL incorporado, es necesario un nuevo concepto: *cursor*.

También se considerará como actualizar datos por medio del posicionamiento del cursor, y la forma en que la estructura del programa se ve afectada al utilizar los cursores.

Como son varias las sentencias nuevas para la programación con cursores, primero se explican y al final se muestra un ejemplo completo de ellas.

5.1.7.1. Concepto y uso de cursores.

Cuando el resultado de una sentencia SELECT tiene varias filas, SQL incorporado proporciona un modo para que el programa de aplicación procese los resultados una fila a la vez. SQL soporta esta capacidad mediante un nuevo concepto, llamado *cursor*, y la adición de nuevas sentencias al lenguaje SQL interactivo.

Cuando se ejecuta una sentencia SELECT, produce una tabla de resultado. Un cursor se comporta como un puntero que se posiciona sobre las filas de la tabla de resultado, y las puede recorrer una a una secuencialmente. El cursor se mueve a la siguiente cada vez que se ejecuta una sentencia FETCH, que se estudia un poco más adelante.

El cursor tiene, por tanto, tres componentes:

1) La tabla de resultado obtenida al ejecutar la sentencia SELECT.

2) Un orden establecido en sus filas, como se presentaría el resultado en forma interactiva.

3) Un puntero que señala una posición sobre la tabla resultado.

Por lo que un cursor se comporta como un fichero secuencial, en el que están almacenadas las filas de la tabla de resultado en un orden establecido. Por ello se procesa de manera similar a un archivo secuencial: hay que definirlo, abrirlo, leerlo y cerrarlo. Esto se logra con las siguientes sentencias de SQL incorporado:

- La sentencia DECLARE CURSOR. En ella se define el cursor. Dentro de la sentencia se incluye la sentencia SELECT con la que se construye la tabla de resultado. Se asocia un nombre de cursor con la consulta.

- La sentencia OPEN. Con ella se abre el cursor. Pide al DBMS que comience a ejecutar la consulta y generar la tabla de resultado en el orden del cursor. Posiciona el cursor antes de la primera fila de la tabla de resultados.

- La sentencia FETCH. La primera vez que se ejecuta avanza el apuntador del cursor a la primera fila de la tabla de resultados de la consulta y recupera los datos en las variables huésped para que el programa de aplicación pueda usarlos. Posteriores ejecuciones de la sentencia FETCH avanzan el cursor a

la fila siguiente y depositan los valores en las correspondientes variables huésped.

- La sentencia UPDATE. Con ella se pueden actualizar columnas de la fila sobre la que está posicionado el cursor.
- La sentencia DELETE permite borrar la fila señalada por el cursor.
- La sentencia CLOSE cierra el cursor. Finaliza el acceso a los resultados de la consulta y deshace la relación entre el cursor y los resultados.

A continuación se describen con mas detalle.

5.1.7.1.1.Sentencia DECLARE CURSOR.

La sentencia DECLARE CURSOR, define una consulta y asocia un nombre de cursor a los resultados de la consulta. Esta es una sentencia no ejecutable.

Sintaxis:

DECLARE nombre de cursor CURSOR FOR sentencia SELECT

El nombre del cursor debe ser un nombre SQL válido.

La sentencia SELECT de la sentencia DECLARE CURSOR define la consulta asociada con el cursor. La sentencia de selección puede ser cualquier sentencia SELECT de SQL interactivo. La única opción de SELECT que se ha visto para SQL programado es la opción INTO, ésta no se incluye en el formato y por lo que se ve no se especifican en el cursor las variables huésped de salida donde se deben depositar los valores recuperados. Estas variables se especifican en la sentencia FETCH que se discute después.

La consulta especificada puede incluir variables huésped, que se utilicen como entrada. Pueden aparecer en cualquier lugar de la consulta donde pueda aparecer una constante.

La sentencia DECLARE CURSOR no es ejecutable aunque contiene una sentencia SELECT que si es ejecutable. La sentencia SELECT se ejecuta hasta que el cursor se abre con la sentencia OPEN.

No existe un límite para el número de cursores que pueden ser definidos dentro de un programa, incluso puede haber cursores con la misma consulta que los defina, la única condición es que los nombres de cursor sean diferentes.

La sentencia DECLARE CURSOR debe de preceder a cualquier sentencia donde se haga referencia al cursor y el lugar donde se debe de definir dependerá del programa anfitrión empleado.

En COBOL puede declararse en cualquier posición de la WORKING STORAGE SECTION o LINKAGE SECTION de la DATA DIVISION, y también puede ir en la PROCEDURE DIVISION.

En C puede colocarse fuera o dentro del cuerpo principal (main()).

5.1.7.1.2.Sentencia OPEN.

La sentencia OPEN abre el cursor. Cuando se ejecuta la sentencia OPEN el DBMS ejecuta la consulta. Por lo que la sentencia OPEN ocasiona que el DBMS realice el mismo trabajo que con una sentencia SELECT interactiva.

Sintaxis:

OPEN nombre de cursor

El único parámetro de la sentencia OPEN es el nombre del cursor, que debe de haber sido declarado previamente mediante la sentencia DECLARE CURSOR. Si la consulta asociada tiene algún error, la sentencia OPEN produce un SQLCODE negativo.

El programa dispone de la tabla de resultados inmediatamente después de abrir el cursor y permanece disponible hasta que se cierre con la sentencia CLOSE. El DBMS también cierra todos los cursores abiertos cuando se detecta el final de una transacción con las sentencias COMMIT y ROLLBACK. Si el cursor ha sido cerrado, la sentencia OPEN puede utilizarse otra vez, y el DBMS ejecutará nuevamente la consulta.

5.1.7.1.3.Sentencia FETCH.

La sentencia FETCH recupera la siguiente fila de la tabla de resultados de la consulta con la que se define un cursor, para su proceso en el programa de aplicación. Debe de hacer referencia a un cursor previamente abierto mediante la sentencia OPEN.

Sintaxis:

FETCH nombre de cursor INTO lista de variables huésped

La sentencia FETCH extrae la fila de elementos de datos sobre la lista de variable que se especifican en la cláusula INTO. Cada variable puede ir asociada a una variable indicadora para el manejo de valores NULL. El comportamiento de la variable indicadora es idéntico a el descrito para la sentencia SELECT. El número de variable huésped debe de ser igual al número de columnas de la tabla de resultados, y además los tipos de datos deben de ser compatibles, columna a columna con los de resultados.

La siguiente figura muestra la forma en que se mueve el cursor a través de la tabla de resultados de la consulta:

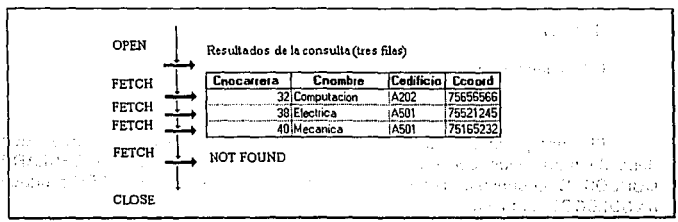


Fig. 5.2 . Posicionamiento del cursor.

- La sentencia OPEN posiciona el cursor antes de la primera fila de resultado. El cursor no apunta a ninguna fila.

- La sentencia FETCH avanza al cursor a la siguiente fila disponible, si existe alguna, ésta se convierte en la fila actual del cursor.
- Cuando el cursor esta posicionado en la última fila, el siguiente FETCH devuelve un aviso de NOT FOUND, es decir devuelve un código de 100 en el SQLCODE. En este caso el cursor tampoco tiene fila actual.
- La sentencia CLOSE finaliza el acceso a los resultados y cierra el cursor.

Si después de la sentencia OPEN la tabla de resultado estuviera vacía, la primera sentencia FETCH regresara el código de error NOT FOUND.

5.1.7.1.4.La sentencia CLOSE.

La sentencia CLOSE cierra la tabla de resultados de la consulta creada por la sentencia OPEN, dando por acabado el acceso al programa de aplicación. Su único parámetro es el nombre del cursor que debe ser un cursor previamente abierto por una sentencia OPEN.

Todos los cursores son cerrados automáticamente al finalizar una transacción.

Ejemplo: Aquí se muestra el uso del procesamiento multifila en lenguaje C. Seleccionar el nombre, los créditos y el costo de laboratorio de los cursos con un costo mayor a 100.

```
main()
{
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
    char wnombre[21];          /*nombre de la materia*/
    short int wcred;           /*número de créditos*/
    float wcostolab;           /*costo de laboratorio*/
    short int ind_costo        /*indicador de nulo para el costo*/
    EXEC SQL END DECLARE SECTION;

    /*Declara del cursor de la consulta*/
    EXEC SQL DECLARE currep CURSOR FOR
        SELECT MNOMBRE, MCRED, MCOSTOLAB
            FROM MATERIA
            WHERE MCOSTOLAB > 100.00
```

```
                ORDER BY MNOMBRE;
    /*Procesamiento de errores*/
    EXEC SQL WHENEVER SQLERROR goto error;
    EXEC SQL WHENEVER NOT FOUND goto termino;
    /*Abre el cursor*/
    EXEC SQL OPEN currep;
    /*Ciclo para leer los resultados*/
    for( ; ; ) {
        EXEC SQL FETCH currep
            INTO :wnombre, :wcred, :wcostolab :ind_costo;
        /*Visualiza datos*/
        printf("Nombre: %s\n",inforep.wnombre);
        if (ind_costo < 0)
            printf("El costo de laboratorio tiene valor NULL\n);
        else
            printf("Costolab: %f\n",inforep.wcostolab);
        printf("Créditos: %d\n",inforep.wcred);
    }
    error:
        printf("SQL error: %d\n",sqlca.sqlcode);
        exit();
    termino:
        /*Cierra cursor*/
        EXEC SQL CLOSE currep;
        exit();
}
```

El mismo ejemplo en COBOL.

```
WORKING STORAGE SECTION.
    .....
    EXEC SQL INCLUDE SQLCA END-EXEC.
    EXEC SQL BEGIN DECLARE SECTION END EXEC.
    01 WNOMBRE          PIC X(21).
    01 WCRED            PIC S9(02).
    01 WCOSTOLAB        PIC S9(07)V9(02) COMP-3.
    01 INDCOSTO         PIC S9(04) COMP -4.
    EXEC SQL END DECLARE SECTION END EXEC.
    .....
```

```
PROCEDURE DIVISION.
INICIO.
        EXEC SQL DECLARE CURREP CURSOR FOR
        SELECT MNOMBRE, MCRED, MCOSTOLAB
                FROM MATERIA
                WHERE MCOSTOLAB > 100.00
                ORDER BY MNOMBRE
        END-EXEC.
        EXEC SQL WHENEVER SQLERROR GO TO ERROR END-EXEC.
        EXEC SQL WHENEVER NOT FOUND GO TO TERMINO END-EXEC.
        EXEC SQL OPEN CURREP END-EXEC.

CICLO.
        EXEC SQL FETCH CURREP
                INTO :WNOMBRE, :WCRED, :WCOSTOLAB :IND_COSTO
        END-EXEC.
        DISPLAY "Nombre: " WNOMBRE
        IF INDCOSTO < 0
                DISPLAY "El costo de laboratorio tiene valor NULL"
        ELSE
                DISPLAY "Costolab: " WCOSTOLAB
        END-IF
        DISPLAY "Créditos: " WCRED.

ERROR.
        DISPLAY "SQL error: " SQLCODE
        STOP RUN.

TERMINO.
        EXEC SQL CLOSE CURREP END-EXEC.
        STOP RUN.
```

5.1.7.2. Modificación de datos utilizando cursores.

Cuando un cursor esta posicionado en una fila de la tabla de resultado, es posible modificar o borrar esa fila. Si la consulta extrae datos de una única tabla, y no es una consulta sumaria, el cursor apunta implícitamente a una fila de una tabla de la base de datos, ya que cada fila del resultado es extraída de una única fila de la tabla.

Mientras se inspecciona los datos, el usuario puede apuntar a datos que deberían ser modificados o borrados. El programa utiliza el cursor como puntero para indicar que fila desea ser actualizada o suprimida en la base de datos.

SQL soporta esta capacidad mediante versiones especiales de las sentencias DELETE y UPDATE, estas versiones son denominadas DELETE *posicionada* y UPDATE *posicionada*.

Para usar éstas sentencias, se tiene que tomar en cuenta algunas consideraciones para saber cuales tablas de resultado de un cursor es actualizable y cual no. La tabla de resultado de un cursor no será actualizable, si la sentencia SELECT que lo define tiene cualquiera de las siguientes características:

- Si existe una operación join, entre dos o más tablas.
- Si usa la cláusula DISTINCT en la sentencia SELECT.
- Tiene filas agrupadas, es decir usa GROUP BY o HAVING.
- Usa funciones de columna.
- Hace referencia a una vista no actualizable.
- Usa la cláusula ORDER BY.
- Usa una cláusula UNION.

Si la sentencia SELECT no tiene ninguna de las características anteriores, se dice que la tabla de resultado del cursor es actualizable.

5.1.7.2.1. Cláusula FOR UPDATE.

Esta cláusula pertenece al formato de la sentencia DECLARE CURSOR, y forma parte de la sentencia SELECT en la declaración del cursor.

Sintaxis:

FOR UPDATE OF lista de columnas

Esta cláusula solamente se puede especificar en la declaración del cursor si la tabla de resultado es actualizable.

Esta sentencia habilita al programa a que posteriormente se modifiquen columnas mediante la sentencia UPDATE.

Cualquier columna de la tabla de resultado se puede mencionar en la cláusula FOR UPDATE, incluso aunque no sea una de las seleccionadas en la sentencia SELECT.

Ejemplo:

```
EXEC SQL
DECLARE CURREP CURSOR FOR
        SELECT MNOMBRE, MCRED, MCOSTOLAB
        FROM MATERIA
        WHERE MCOSTOLAB > 100.00
        FOR UPDATE OF MNOCARRERA
END-EXEC.
```

Esta sentencia permite actualizar la columna MNOCARRERA de la tabla original.

5.1.7.2.2.Sentencia UPDATE con cursor.

Se utiliza para actualizar la fila sobre la que esté posicionado el cursor. Su sintaxis es la siguiente:

UPDATE nombre de tabla o vista
SET nombre de columna = expresión [,nombre de columna = expresión ...]
WHERE CURRENT OF nombre de cursor

El formato es igual al ya conocido, con la diferencia de que ahora no se especifica una cláusula WHERE, ya que la posición del cursor define perfectamente su posición.

La sentencia actualiza la fila sobre la que esta posicionado el cursor en un momento dado, modificando las columnas mencionadas en su cláusula SET, que se deben haber incluido en la declaración del cursor en la cláusula FOR UPDATE.

La actualización se realiza tanto en la tabla de resultados como en la tabla de la base de datos.

La sentencia no modifica la posición del cursor, que seguirá apuntando a la misma fila que apuntaba antes del UPDATE.

5.1.7.2.3.Sentencia DELETE con cursor,

Se utiliza para borrar la fila sobre la que se encuentra el cursor. Para procesar la sentencia, el DBMS localiza la fila de la tabla real que corresponde a la fila actual del cursor y suprime esa fila en la base. Al borrar la fila, el cursor apunta a un espacio vacío, antes de la siguiente fila, si es que existe, esperando ser avanzado por la sentencia FETCH.

Sintaxis:

DELETE FROM nombre de tabla o vista
WHERE CURRENT OF nombre cursor

Ejemplo: Bajar las tarifas en un 10 por ciento de las materias en donde su costo de laboratorio sea mayor a 100.

```
WORKING STORAGE SECTION.
     .....
     EXEC SQL INCLUDE SQLCA END-EXEC.
     EXEC SQL BEGIN DECLARE SECTION END EXEC.
     01 WCOSTOLAB          PIC S9(07)V9(02) COMP-3.
     01 INDCOSTO           PIC S9(04) COMP -4.
     EXEC SQL END DECLARE SECTION END EXEC.
     .....
PROCEDURE DIVISION.
INICIO.
     EXEC SQL DECLARE CURREP CURSOR FOR
     SELECT MCOSTOLAB
          FROM MATERIA
          WHERE MCOSTOLAB > 100.00
          ORDER BY MNOMBRE
          FOR UPDATE OF MCOSTOLAB
     END-EXEC.
     EXEC SQL WHENEVER SQLERROR GO TO ERROR END-EXEC.
     EXEC SQL WHENEVER NOT FOUND GO TO TERMINO END-EXEC.
     EXEC SQL OPEN CURREP END-EXEC.

CICLO.
     EXEC SQL FETCH CURREP
          INTO :WCOSTOLAB :IND_COSTO
     END-EXEC.
     EXEC SQL UPDATE MATERIA
          SET WCOSTOLAB = :WCOSTOLAB * .90
          WHERE CURRENT OF CURREP
     END-EXEC.
ERROR.
     DISPLAY "SQL error: " SQLCODE
     STOP RUN.
TERMINO.
     EXEC SQL CLOSE CURREP END-EXEC.
     STOP RUN.
```

El mismo programa del ejemplo anterior, ahora en C.

```
main()
{
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
    float wcostolab;            /*costo de laboratorio*/
    short int ind_costo         /*indicador de nulo para el costo*/
    EXEC SQL END DECLARE SECTION;

    /*Declara del cursor de la consulta*/
    EXEC SQL DECLARE currep CURSOR FOR
        SELECT MCOSTOLAB
            FROM MATERIA
            WHERE MCOSTOLAB > 100.00
            ORDER BY MNOMBRE;

    /*Procesamiento de errores*/
    EXEC SQL WHENEVER SQLERROR goto error;
    EXEC SQL WHENEVER NOT FOUND goto termino;

    /*Abre el cursor*/
    EXEC SQL OPEN currep;

    /*Ciclo para actualizar el costo de laboratorio*/
    for( ; ; ) {
        EXEC SQL FETCH currep
            INTO :wcostolab :ind_costo;
        EXEC SQL UPDATE MATERIA
            SET wcostolab = :wcostolab * .90
            WHERE CURRENT OF currep;
    }
    error:
        printf("SQL error: %d\n",sqlca.sqlcode);
        exit();
    termino:
        /*Cierra cursor*/
        EXEC SQL CLOSE currep;
        exit();
}
```

5.1.8. Cursores y procesamiento de transacciones.

En la mayoría de los productos SQL la terminación de una transacción implica el cierre de sus cursores. Es decir cuando se ejecuta una sentencia COMMIT o ROLLBACK, cierra todos los cursores que se encuentren abiertos en el programa en ese momento.

Al cerrar un cursor, se pierde la tabla de resultados y lógicamente se pierde la posición del cursor. Si fuera necesario, el programa deberá volverlo a abrirlo, pero el cursor aparecerá posicionado al principio de ésta y no donde se encontraba en el momento de que fue cerrado. El programa deberá tener en cuenta esto para no volver a procesar las mismas filas de la tabla de resultados.

La manera de posicionarse correctamente en el cursor después de que fue cerrado, podría ser la que se muestra en el siguiente ejemplo:

Ejemplo: Se desea actualizar todos los costos de laboratorio, aumentandolos en un 10%.

1) Declaración de un cursor que lee las materias en orden empezando por la materia "0000".

```
.....
MOVE 0 TO NUM
EXEC SQL
      DECLARE CURREP CURSOR FOR
            SELECT MNOCARRERA
            FROM MATERIA
            WHERE MNOCARRERA > :NUM
            ORDER BY MNOCARRERA
END-EXEC.
```

2) Abrir el cursor, leer sus filas de uno a uno y actualizarlas.

```
LECTURA.
      EXEC SQL OPEN CURREP END-EXEC.
      EXEC SQL WHENEVER SQLERROR GO TO ERROR END-EXEC.
      EXEC SQL WHENEVER NOT FOUND GO TO TERMINO END-EXEC.
```

```
CICLO.
    EXEC SQL FETCH CURREP INTO :NUM END EXEC.
    EXEC SQL
        UPDATE MATERIA
            SET MCOSTOLAB = MCOSTOLAB * 1.10
            WHERE MNOMATERIA = :NUM
    END-EXEC.
    .....
```

Se utilizo la sentencia UPDATE sobre la tabla MATERIA porque el cursor tiene la sentencia ORDER BY, que lo hace no actualizable.

Cada vez que se lee una fila de la tabla de resultados se almacena el número de la materia en la variable NUM, que sirve para cuando se vuelva a abrir el cursor, posicionarse en la siguiente fila no procesada.

3) En el transcurso del programa, tal vez aparezca un COMMIT. Después habrá necesidad de reabrir el cursor, que queda posicionado en la fila donde debe continuar el proceso.

```
EXEC SQL COMMIT WORK END-EXEC
EXEC SQL OPEN CURREP END-EXEC.
PERFORM CICLO.
```

Al ejecutar nuevamente la sentencia que define el cursor, solamente se encuentran en la tabla de resultado, aquellas fila que no han sido procesadas, esto debido a la condición de la consulta en la definición.

Si no se recuerdan conceptos como Unidad Lógica de Trabajo, transacción, bloqueo, interbloqueo y las sentencias COMMIT y ROLLBACK se recomienda regresar al capítulo tres, donde se explican ampliamente.

5.2. SQL Dinámico.

Recuérdese, del prin cipio del capítulo que como los compiladores no entienden las sentencias SQL, hay que traducir éstas mediante un *precompilador.* El precompilador produce un programa fuente, despojado de las sentencias SQL. En su lugar, el precompilador sustituye llamadas a rutinas del DBMS.

Algunas sentencias, llamadas *ejecutables*, son almacenadas por el precompilador en un módulo aparte, que tras un proceso de vinculación (BIND), que analiza, valida y optimiza las sentencias, se da lugar a un módulo llamado plan, en el que figuran todas las sentencias SQL ejecutables. El programa BIND se encarga de almacenar el plan en la base de datos, asignándole el nombre del programa de aplicación que creó el plan.

En particular, el precompilador maneja el análisis de las sentencias SQL, y la utilidad BIND se encarga de la verificación, la optimización y la generación del plan.

El optimizador es un componente muy importante de la herramienta BIND, es el responsable de decidir la mejor estrategia de accesos físicos a emplear para obtener el resultado correspondiente a una sentencia SQL.

Por las razones expuestas, es recomendable, siempre que sea posible, el proceso de optimización se debe hacer antes de la ejecución del programa, en el proceso de vinculación (BIND), en este caso los accesos se almacenan en el plan, en la base de datos.

Sin embargo esto no siempre es posible, si la sentencia es generada dinámicamente en el programa, por ejemplo que se le de oportunidad al usuario de escribir la condición de una cláusula, entonces en el paso de vinculación no se podrá optimizar la sentencia. En este caso no queda más remedio que optimizar la sentencias durante la ejecución del programa.

Las sentencias que se optimizan en el tiempo de ejecución se denominan sentencias de optimización dinámica, o para abreviar *sentencias dinámicas*. Por el contrario las que se optimizan en el tiempo de vinculación se denominan *sentencias estáticas*. La cualidad de que sean estáticas o dinámicas dependerá de la forma en que se codifiquen y utilicen por parte del programador.

La mayoría de las aplicaciones pueden resolverse con sentencias estáticas. Sin embargo hay aplicaciones donde se requieren sentencias dinámicas. Por ejemplo el programa que traduce las sentencias en forma interactiva, no tiene manera de saber cual es la siguiente sentencia que se escribirá, por tanto, no queda más remedio que optimizar las sentencias durante su ejecución.

Todos los conceptos explicados en esta sección ya se vieron al inicio del capítulo, si se desea ahondar mas en éstos se recomienda leerlos nuevamente, aquí se presentó un resumen para entender la diferencia entre *SQL Estático* y *SQL Dinámico*.

5.2.1. SQL estático y SQL dinámico.

El término *SQL estático* se emplea un programa contiene sentencias SQL que ya fueron optimizadas durante el proceso de vinculación, es decir la optimización se realiza antes del momento en que el programa se ejecuta.

El uso estático del SQL permite ejecutar una petición optimizada, sin necesidad de optimizarla cada vez que se ejecute.

El *SQL dinámico* es la herramienta adecuada para accesos a la base que no se pueden prever. Los programas que utilizan SQL dinámico, pueden tener los siguientes pasos en su procedimiento:

- Leer la petición del usuario en modo interactivo.
- Traducir la petición a una sentencia SQL si no se dio en este lenguaje. Por ejemplo el usuario del programa puede seleccionar menús y el programa armar la sentencia.
- Mandar la sentencia al DBMS, el cual se encarga de optimizarla, ejecutarla y regresar los resultados al programa.
- Presentar los resultados al usuario.

SQL estático es suficiente para escribir todos los programas típicamente requeridos por aplicaciones en SQL. SQL dinámico es considerado como un tema avanzado de SQL incorporado. Los conceptos que se manejarán son similares en los diferentes lenguajes donde se puede embeber el SQL, y en lo posterior solamente se presentarán ejemplos enfocados al lenguaje C.

SQL dinámico varía de un producto a otro ya que requiere mas interacción con el DBMS específico de cada producto, sin embargo los formatos solo varían en opciones de cada producto. A continuación se explican los formatos más usuales, si se desea conocer todas las opciones de cada sentencia se tendrá que hacer uso de los manuales de referencia correspondientes.

5.2.2. Ejecución dinámica de una sentencia.

La sentencia EXECUTE INMEDIATE transmite el texto de una sentencia de SQL al DBMS para su ejecución inmediata.

Sintaxis:

EXECUTE INMEDIATE :variable huésped

Donde la variable huésped contiene la sentencia que se va a vincular y ejecutar. El DBMS ejecuta la sentencia que está en la variable huésped, y regresa el valor de SQLCODE igual que en SQL estático. EXECUTE INMEDIATE sólo se puede usar para sentencias que no recuperan datos.

Ejemplo: El siguiente programa sirve para borrar filas de cualquier tabla, con la condición que el usuario especifique al correr el programa.

```
main()
{
        EXEC SQL INCLUDE SQLCA;
        EXEC SQL BEGIN DECLARE SECTION;
        char varsql[301];           /*Texto que contendrá la sentencia */
        EXEC SQL END DECLARE SECTION;

        char ntabla[101];           /*Variable para el nombre de tabla */
        char cond_whe[101];         /*Variable para la condición en WHERE */

        /*Empieza a construir la sentencia */
        strcpy(varsql, "DELETE FROM ");

        /*Pide el nombre de la tabla que se actualizará */
        /*y agrega el nombre de la tabla en la sentencia */
        printf("Dame el nombre de la tabla:   ");
        gets(ntabla);
        strcat(varsql, ntabla);

        /*Pide la condición para elegir filas y */
        /*Agrega la condición a la sentencia */
        printf("Dame condición de búsqueda: ");
        gets(cond_whe);
        strcat(varsql," WHERE ");
        strcat(varsql, cond_whe);

        EXEC SQL EXECUTE INMEDIATE :varsql;
        if (sqlca.sqlcode < 0)
                printf("SQL error: %d\n", sqlda.sqlcode);
        else
                printf("Terminó con éxito el DELETE");

        exit();
}
```

La sentencia EXECUTE INMEDIATE es la forma más sencilla del SQL dinámico pero es muy versátil, puede utilizarse con casi todas las sentencias de manejo de datos, exceptuando la sentencia SELECT, también puede procesar sentencias para la definición de objetos de la base de datos como CREATE, DROP, GRANT y REVOKE.

5.2.3. Ejecución dinámica en dos pasos.

La sentencia EXECUTE INMEDIATE sirve para ejecutar dinámicamente sentencias SQL en un solo paso. Para la ejecución de la sentencia el DBMS realiza el análisis, la validación, la optimización, genera el plan y lo ejecuta, todo cada vez que se ejecuta una sentencia, esto puede representar un fuerte recargo de proceso, sobre todo si el programa ejecuta muchas sentencias.

SQL ofrece un método alternativo para ejecutar las sentencias SQL, ahora en dos pasos, mediante sus sentencias PREPARE y EXECUTE.

5.2.3.1. Sentencia PREPARE,

La sentencia PREPARE solicita al precompilador la vinculación (BIND) sentencia SQL.

Sintaxis:

PREPARE nombre de sentencia FROM :variable huésped

En la *variable huésped* debe de estar almacenada una sentencia SQL válida. PREPARE, analiza, valida y optimiza la sentencia, y al resultado le da el nombre simbólico especificado en *nombre de sentencia* para su ejecución posterior en el programa.

El DBMS retiene la sentencia preparada y el nombre de la sentencia asociado hasta el final de la transacción actual, es decir hasta que el programa encuentre una sentencia ROLLBACK o COMMIT. Si se desea ejecutar posteriormente la misma sentencia dinámicamente en otra transacción posterior, la sentencia debe de ser preparada nuevamente.

La cadena que representa la sentencia SQL puede tener un *marcador de parámetro*, que se indica mediante un signo de interrogación (?), que sustituye a una constante, que posteriormente será suministrada por el usuario, cuando la sentencia sea ejecutada.

5.2.3.2. Sentencia EXECUTE,

Solicita el DBMS la ejecución de una sentencia previamente preparada con la sentencia PREPARE.

Sintaxis:

EXECUTE nombre de sentencia [USING lista de variables huésped]

Donde *nombre de sentencia* es el nombre de la sentencia previamente preparada que desea ejecutarse.

La sentencia EXECUTE no puede utilizarse para recuperar datos, ya que como se ve en su formato no tiene forma de manejar los resultados de las consultas.

5.2.3.3. EXECUTE con USING.

La manera más fácil de pasar valores a los parámetros de una sentencia SQL es mediante la especificación de una lista de variables huésped en la cláusula USING. Cuando se ejecuta la sentencia EXECUTE se sustituyen los valores de las variables huésped, por los marcadores de parámetro en el texto de la sentencia preparada. La correspondencia es uno a uno y deben de coincidir en número y en tipo.

Cada variable huésped puede tener una variable indicadora asociada para el manejo de valores NULL.

Ejemplo:

```
main()
{
        EXEC SQL INCLUDE SQLCA;
        EXEC SQL BEGIN DECLARE SECTION;
        char varsql[301];           /*Texto que contendrá la sentencia */
        float v_busca;              /*Valor del parámetro para búsqueda */
        float n_valor;              /*Valor del parámetro para actualizar */
        EXEC SQL END DECLARE SECTION;

        char ntabla[101];           /*Variable para el nombre de tabla */
        char col_busca[31];         /*Columna para la búsqueda */
        char col_actual[31];        /*Columna que se actualizará */
        char resp[31]:              /*Respuesta si o no del usuario */

        /*Pide el nombre de la tabla y columnas */
        printf("Dame el nombre de la tabla:   ");
        gets(ntabla);
        printf("Dame el nombre de la columna a buscar: ")
        gets(col_busca);
        printf("Dame el nombre de la columna a actualizar:   ");
        gets(col_actual);
        /*Construye la sentencia y pide al DBMS la vinculación */
        sprintf(varsql, "UPDATE %s SET %s = ? WHERE %s = ?" ,
                ntabla, col_actual, col_busca );
        EXEC SQL PREPARE sent_ej FROM :varsql;
        if (sqlca.sqlcode < 0) {
                printf( "PREPARE error: %d\n",  sqlda.sqlcode);
                exit();
        }
        /*Pide los parámetros al usuario y actualiza. */
        printf("Dame el valor de búsqueda para %s: ", col_busca );
        scanf("%f", &v_busca);
        printf("Dame el nuevo valor para %s: ", col_actual );
        scanf("%f", &n_valor);
        /*Pide al DBMS que ejecute la sentencia */
        EXECUTE sent_ej USING :v_busca, :n_valor;
        if (sqlca.sqlcode < 0) {
                printf("EXECUTE error: %d\n",  sqlda.sqlcode);
                exit();
        }
        printf("El programa de actualización ha terminado/n");
        exit()
}
```

El programa permite actualizar cualquier tabla que se pueda acceder por una columna numérica y permite actualizar cualquier columna también numérica. Es de propósito general, ya que pide el nombre de la tabla, la columna para la condición de búsqueda y la columna que se desea actualizar.

El programa construye la sentencia en la variable huésped *varsql*, y tiene dos marcadores de parámetros para pedir posteriormente el valor de la columna por la cual se accesa y el nuevo valor de la columna que se actualiza.

Una vez construido el texto de la sentencia , el programa pide al DBMS que ejecute la sentencia prepare. Posteriormente se piden los valores de los parámetros de la sentencia y ejecuta la sentencia.

Puede ser que se necesite actualizar varias filas de una tabla. El programa puede utilizar la sentencia EXECUTE repetidas ocasiones, suministrando diferentes valores de parámetros cada vez que la sentencia dinámica se ejecute, y solamente es necesario preparar la sentencia una sola vez.

5.2.4. Consultas dinámicas.

Las sentencia EXECUTE INMEDIATE, PREPARE y EXECUTE, como se han descrito, no soportan la ejecución dinámica de consultas. Para soportarlas, SQL combina las características de SQL dinámico con extensiones a las sentencias de procesamiento de consultas de SQL estático.

5.2.4.1. SQL dinámico para SELECT de lista fija.

En el caso de que se desee ejecutar una sentencia SELECT que seleccione un conjunto de columnas predefinido. Es decir, se conoce el número de columnas como las características de éstas, de modo que se pueden definir las variables para recibir los datos seleccionados.

En este caso, la utilidad radica en que se pueden variar los criterios de selección en cada ejecución.

El procedimiento usado para aplicaciones con consultas dinámicas, debe de:

- Elaborar el WHERE de la sentencia SELECT a partir de datos suministrados por el usuario.
- Declarar el cursor mediante DECLARE CURSOR.
- Preparar el SELECT asociándolo al cursor anterior.
- Abrir el cursor , recuperar las filas procesarlas, y cerrar el cursor. Esto se realiza con las sentencias ya conocidas OPEN, FETCH y CLOSE.

Ejemplo:

```
main()
{
        EXEC SQL INCLUDE SQLCA;
        EXEC SQL BEGIN DECLARE SECTION;
        char wnombre[21];          /*nombre de la materia */
        short int wcred;           /*número de créditos */
        float wcostolab;           /*costo de laboratorio */
        short int ind_costo        /*indicador de nulo para el costo */
        char varsql[301];          /*Texto que contendrá la sentencia */
        EXEC SQL END DECLARE SECTION;

        char cond_whe[101];        /*Variable para la condición en WHERE */

        /*Procesamiento de errores */
        EXEC SQL WHENEVER SQLERROR goto error;
        EXEC SQL WHENEVER NOT FOUND goto termino;

        /*Empieza a construir la sentencia */
        strcpy(varsql, "SELECT MNOMBRE, MCRED, MCOSTOLAB ");
        strcat(varsql," FROM MATERIA WHERE ");

        /*Pide la condición para elegir filas y agrega la condición a la sentencia */
        printf("Dame condición de búsqueda para la tabla MATERIA: ");
        gets(cond_whe);
        strcat(varsql, cond_whe);

        /*Declaración del cursor de la consulta */
        EXEC SQL DECLARE currep CURSOR FOR con_ej;
        EXEC SQL PREPARE con_ej FROM :varsql;
```

```
    EXEC SQL OPEN currep;    /*Abre el cursor */
    /*Ciclo para leer los resultados */
    for( ; ; ) {
        EXEC SQL FETCH currep
            INTO :wnombre, :wcred, :wcostolab :ind_costo;
        /*Visualiza datos */
        printf("Nombre: %s\n", inforep.wnombre);
        if (ind_costo < 0)
            printf("El costo de laboratorio tiene valor NULL\n);
        else
            printf("Costolab: %f\n", inforep.wcostolab);
        printf("Créditos: %d\n", inforep.wcred);
    }
    error:
        printf("SQL error: %d\n", sqlca.sqlcode);
        exit();
    termino:
        /*Cierra cursor */
        EXEC SQL CLOSE currep;
        exit();
}
```

En el ejemplo **DECLARE CURREP CURSOR FOR CON_EJ** declara un cursor llamado **CURREP** para la ejecución de una sentencia llamada **CON_EJ**. La sentencia **DECLARE CURSOR** en forma dinámica especifica la consulta indirectamente, mediante la especificación del nombre de sentencia asociado con la consulta mediante la sentencia **PREPARE**.

PREPARE CON_EJ FROM :VARSQL, indica que se prepare la sentencia que se encuentra en la variable **VARSQL** y que se le asigne el nombre de **CON_EJ**.

5.2.4.2. SQL dinámico para SELECT de lista variable.

En el caso de que se desee ejecutar una sentencia SELECT que seleccione un conjunto de columnas no predeterminado. Es decir, se desconoce el número de columnas del resultado, de modo que no se pueden definir las variables para recibir los datos seleccionados.

Para soportar el SQL dinámico de consultas de lista variable se usa un bloque de control llamado SQLDA *(SQL descriptor Area).* SQLDA tiene una entrada por cada columna que se selecciona, y tiene la siguiente información:

- Tipo de datos seleccionado.
- Dirección de memoria que contendrá los datos.
- Longitud del área de memoria necesaria.

El SQLDA es una estructura de datos de tamaño variable con dos partes:

- La parte fija esta localizada el comienzo del SQLDA. Sus campos identifican la estructura de datos como un SQLDA y especifican el tamaño de este SQLDA particular.
- La parte variable es un arreglo de una o más estructuras de datos SQLVAR. Cuando se utiliza SQLDA para transferir parámetros debe haber una estructura SQLVAR por cada parámetro.

A continuación se presenta la descripción del SQLDA para las bases de datos en IBM:

```
struct sqlda{
unsigned char       sqlaid[8];
long                sqldabc;
short               sqln;
short               sqld;
struct sqlvar       {
        short       sqltype;
        short       sqllen;
        short       *sqldata;
        short       *sqlind;
        unsigned char sqlname[32];
        }sqlvar[1];
}
```

Los campos de la estructura SQLVAR se describen a continuación:

- El campo SQLTYPE contiene un código de tipo de dato, que especifica el tipo de dato que se transfiere. El código varia en cada producto.
- El campo SQLLEN especifica la longitud de los datos transferidos. Contendrá un 2 para un entero de dos bytes y un 4 para un entero de cuatro bytes, cuando se transfieren cadenas, contiene el número de caracteres de la cadena.
- SQLDATA es un puntero al área datos que contiene el valor del parámetro. El DBMS usa este puntero para hallar el valor del dato cuando se ejecuta la sentencia SQL dinámica.
- SQLIND es un puntero a un entero de dos bytes que se utiliza como variable indicadora para el parámetro. Se utiliza para saber si el valor del parámetro es un valor NULL.

La técnica empleada en las consultas de lista variable es la siguiente:

1) Asignar en el programa un área de memoria para el SQLDA.

   El tamaño del área esta dado por el número máximo de columnas seleccionadas.

2) Construir una sentencia SELECT.

3) Declarar el cursor.

4) Preparar la sentencia.

5) Obtener en la SQLDA información descriptiva sobre la sentencia SELECT preparada, para lo cual se usa la nueva sentencia DESCRIBE, que sirve para pedirle al DBMS que devuelva en SQLDA el tipo de datos y longitud de cada columna seleccionada.

6) Con la información del SQLDA, adquirir dinámicamente memoria para las columnas resultantes de la consulta.

   Almacenar en SQLDA las direcciones de las áreas de memoria adquiridas.

7) Ejecutar la sentencia SQL indicando que se usa SQLDA en lugar de una lista de variables.

Existe la dificultad en el manejo de variables que implica disponer de facilidades para la obtención dinámica de memoria y manejo de direcciones. Estas posibilidades no se encuentran disponibles en todos los lenguajes de programación, aunque si se encuentran en C , en PL/1 y Assembler de IBM.

Antes de ver un ejemplo se necesita estudiar brevemente la nueva sentencia DESCRIBE y una nueva opción en FETCH: USING DESCRIPTOR.

5.2.4.3. La sentencia DESCRIBE.

La sentencia DESCRIBE es propia de las consultas dinámicas. Sirve para pedir al DBMS una descripción de una consulta dinámica. Es utilizada después de la sentencia PREPARE pero antes de la sentencia OPEN. El DBMS devuelve la descripción de la consulta en un SQLDA suministrado por el programa:

Sintaxis:

DESCRIBE nombre de sentencia INTO nombre de descriptor

Antes de transferir el SQLDA a la sentencia DESCRIBE, el programa debe rellenar el campo SQLN, informando al DBMS de lo grande que es el arreglo SQLVAR en este SQLDA en particular.

Como primer paso del procesamiento de DESCRIBE, el DBMS rellena el campo SQLD con el número de columnas de resultados de la consulta. Si el tamaño del arreglo SQLVAR es demasiado pequeño para contener todas las descripciones de columnas, el DBMS manda un error. En caso contrario, rellena una estructura SQLVAR por cada columna de resultados en orden de izquierda a derecha.

La estructura SQLNAME indica el nombre de la columna.

5.2.4.4. La sentencia FETCH dinámica.

La sintaxis para la sentencia FETCH dinámica:

FETCH nombre de cursor USING DESCRIPTOR nombre del descriptor

es una variación de la sentencia FETCH estática.

La sentencia FETCH avanza el cursor a la siguiente fila disponible de resultados y recupera los valores de sus columnas en las áreas de datos del programa.

Antes de usar la sentencia FETCH, es responsabilidad del programa de aplicación proporcionar los campos SQLDATA, SQLIND y SQLLEN de SQLVAR por cada columna, del siguiente modo:

- SQLDA debe apuntar al área de datos para el dato recuperado.
- SQLLEN debe especificar la longitud del área de datos a que apunta el campo SQLDATA.
- SQLIND debe apuntar a una variable indicadora para la columna. Si no se usa una variable indicadora SQLIND debería ponerse en cero.

Normalmente, el programa asigna un SQLDA, utiliza la sentencia DESCRIBE para obtener una descripción de los resultados, asigna almacenamiento para cada columna de los resultados y establece los valores de SQLDATA y SQLIND, todo antes de abrir el cursor. Este mismo SQLDA es usado por la sentencia FETCH.

Ejemplo: Programa con SELECT de lista variable.

```
main()
{
/* Programa de consulta de propósito general */
        EXEC SQL INCLUDE SQLCA;
        EXEC SQL INCLUDE SQLDA;
        EXEC SQL BEGIN DECLARE SECTION;
        char bufsent[301];          /*Texto que contendrá la sentencia */
        char sel_tabl[31];          /*Nombre de tabla para la consulta */
        char sel_col[31];           /*Nombres de las columnas para el SELECT */
        EXEC SQL END DECLARE SECTION;

        /* Se declara el cursor para la consulta con_ej dinámica */
        EXEC SQL DECLARE currep CURSOR FOR con_ej;

        /*Procesamiento de errores */
        EXEC SQL WHENEVER SQLERROR goto error;
        EXEC SQL WHENEVER NOT FOUND goto termino;
```

```
/*datos para el programa */
int             numcols = 0;  /* número de columnas */
struct sqlda    *p_sqlda      /* apuntador a SQLDA */
struct sqlvar   *p_sqlvar     /* apuntador a SQLVAR */
char            resp[101]             /* respuesta del usuario */

/*Pide la tabla que se desea consultar */
printf("Dame el nombre de la tabla que deseas consultar: ");
gets(sel_tabl);

/*Empieza a construir la sentencia */
strcpy(bufsent, "SELECT ");

/* Pide las columnas para la cláusula SELECT */
for ( ; ; ) {
        printf("Dame el nombre de la columna que deseas consultar: ");
        gets(sel_col);
        printf("Deseas consultar más columnas ? ")
        gets(resp);
        if (resp[0] == "y" {
                if (numcols++ > 0)
                        strcat(bufsent,", ");
                strcat(bufsent, sel_col);
        }
        else
        break;
}

/* Termina de armar la sentencia SELECT */
strcat(bufsent, "FROM ");
strcat(bufsent, sel_tabl);

/* Asigna SQLDA para la consulta */
p_sqlda = (SQLDA *) malloc(sizeof(SQLDA) + numcols * sizeof(SQLVAR));
p_sqlda -> sqln = numcols

/*Prepara la consulta y manda al DBMS que la describa */
EXEC SQL PREPARE con_ej FROM :bufsent;
EXEC SQL DESCRIBE con_ej INTO :*p_sqlda;

/* Con los SQLVAR, asigna memoria para cada columna */
for (i = 0; i < numcols; i++) {
        p_sqlvar = p_sqlda -> sqlvar + i;
        p_sqlvar -> sqldata = malloc(p_sqlvar -> sqllen);
        p_sqlvar -> sqlind = malloc(2);
}
```

```
/*El SQLDA esta preparado; Ciclo para leer los resultados */
EXEC SQL OPEN currep;              /*Abre el cursor */
for( ; ; ) {
        /* Recuperar datos dinámicamente */
        EXEC SQL FETCH currep
                USING DESCRIPTOR :*p_sqlda;
        printf("\n")
        /*Visualiza datos */
        for (i = 0; i < numcols; i++) {
                /* Imprime el nombre de la columna */
                p_sqlvar = p_sqlda + i;
                printf("Columna %s: ", *(p_sqlvar -> sqlname));
                /*Comprueba valores NULL */
                if (*(p_sqlvar -> sqlind)) != 0) {
                        printf("!es NULL\n");
                        continue;
                }
                /* Datos válidos. cada tipo de datos se trata en forma separada */
                /* Los códigos de dato de este ejemplo corresponden DB2 de IBM */
                /* y no se presentan todos los tipos de datos existentes */
                switch(*(p_sqlvar -> sqltype)){
                        case 448:
                        /* El dato es CHAR no requiere conversión */
                        puts(*(p_sqlvar -> sqldata));
                        break;
                        case 496:
                        /* El dato es entero de 4 bytes, convertirlo y mostrarlo */
                        printf("%d",*((int *)(p_sqlvar -> sqldata)));
                        break;
                        case 500:
                        /* El dato es entero de 2 bytes, convertirlo y mostrarlo */
                        printf("%d",*((short *)(p_sqlvar -> sqldata)));
                        break;
                }
        }
}
error:
        printf("SQL error: %d\n", sqlca.sqlcode);
        exit();
```

```
termino:
        /* Libera la memoria */
        for (i = 0; i < numcols; i++) {
        p_sqlvar = p_sqlda -> sqlvar + i;
        free(p_sqlvar -> sqllen);
        free(p_sqlvar -> sqlind);
        }
        EXEC SQL CLOSE currep;          /*Cierra cursor */
        exit();
}
```

El programa esta documentado de tal manera que puedan identificarse los pasos para realizar una consulta dinámica para lista variable.

APÉNDICE A.

Las 12 reglas de Codd para un DBMS relacional.

En un artículo publicado en 1985 en *Computerworld,* Ted Codd presentó doce reglas que debería de cumplir cualquier base de datos que deseara llamarse relacional. Estas doce reglas son presentadas a continuación y desde su publicación han sido tomadas como una definición semioficial de lo que es una base de datos relacional.

Desgraciadamente estas reglas son más un ideal que una realidad ya que actualmente ningún DBMS cumple totalmente con las doce reglas de Codd.

1. *La regla de la información.* Toda información de una base de datos relacional está representada explícitamente a nivel lógico y exactamente mediante un modo - mediante valores de tablas -.

   Como observamos esta es un definición informal de un base de datos relacional.

2. *Regla de acceso garantizado.* Todos y cada uno de los datos de una base de datos relacional se garantiza que sean lógicamente accesibles recurriendo a una combinación de nombre de tabla, valor de clave primaria y nombre de columna.

   Esta regla hace referencia a la forma de accesar los datos en la base de datos y menciona la importancia de la llave primaria. Para accesar un dato necesitamos el nombre de la tabla, el valor de la clave primaria para accesar la fila de la tabla, y por último el nombre de la columna para encontrar el dato individual que nos interesa.

3. *Tratamiento sistemático de valores nulos.* Los valores nulos (distinto de la cadena de caracteres vacía o de una cadena de caracteres en blanco y distinta de cero o de cualquier otro numero) se soportan en los DBMS completamente relacionales para representar la falta de información y la información inaplicable de un modo sistemático e independiente del tipo de datos.

Esta regla solicita soporte para que la base de datos maneje valores NULL en sus columnas. Un valor NULL es diferente de espacios para un valor alfanumérico y también es diferente de ceros para un valor numérico. Por ejemplo si a un empleado todavía no le ha sido capturado un sueldo no quiere decir que tenga un sueldo de cero pesos.

4. *Catálogo en línea dinámico basado en el modelo relacional.* La descripción de la base de datos se representa a nivel lógico del mismo modo que los datos ordinarios, de modo que los usuarios autorizados puedan aplicar a su interrogación el mismo lenguaje relacional que aplican a los datos regulares.

   Esta regla se refiere a que una base de datos contenga la información de la propia base de datos en forma de tablas que son conocidas como *tablas del sistema.* Es decir la base es autodescriptiva.

5. *Regla de sublenguaje completo de datos.* Un sistema relacional puede soportar varios lenguajes. Sin embargo debe haber al menos un lenguaje cuyas sentencias sean expresables, mediante alguna sentencia bien definida, como cadenas de caracteres, y que sea completa en cuanto al soporte de todos los puntos siguientes:

   - Definición de datos.
   - Definición de vista.
   - Manipulación de datos (interactiva y por programa)
   - Restricciones de integridad.
   - Seguridad.
   - Manejo de transacciones (comienzo, completación y vuelta atrás)

   Esta regla ordena que exista un lenguaje para bases de datos relacionales, que cumpla con los puntos antes mencionados, como lo cumple SQL. Aunque no se menciona que sea SQL el lenguaje a utilizar, éste se ha adoptado como un estándar para las bases de datos relacionales.

6. *Regla de actualización de vista.* Todas las vistas que sean teóricamente actualizables son también actualizables por el sistema.

Las vistas son *tablas virtuales* que presentan a diversos usuarios de la base varias vistas con diferentes estructuras, incluso con columnas de diferentes tablas. Esta regla es de las más difíciles de implementar en la práctica.

7. *Inserción, actualización y supresión de alto nivel.* La capacidad de manejar una relación de base de datos o una relación derivada como un único operando se aplica no solamente a la recuperación de datos, sino también a la inserción, actualización y supresión de datos.

   Esta regla pide que las filas puedan ser tratadas en conjuntos de filas para operaciones de inserción, borrado y actualización y no sea absolutamente necesario el recorrido fila a fila.

8. *Independencia física de datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cualquiera que sean los cambios efectuados ya sea a las representaciones de almacenamiento o a los métodos de acceso.

   En esta regla y en la siguiente se declara la independencia de los programas de usuario de los cambios físicos o lógicos de una base de datos, como pueden ser formas de acceso y cambios en la estructura de tablas en la base.

9. *Independencia lógica de datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cuando se efectúan sobre las tablas de la base cambios preservadores de la información de cualquier tipo que teóricamente permita alteraciones.

10. *Independencia de integridad.* Las restricciones de integridad específicas para una base de datos relacional particular deben ser definibles en el sublenguaje de datos relacional y almacenables en el catálogo, no en los programas de aplicación.

    Esta regla nos dice que el lenguaje relacional debe soportar las reglas de integridad que restringen la adición y actualización de registros en la base de datos.

11. *Independencia de distribución.* Un DBMS relacional tiene independencia de distribución.

Esta regla nos dice que el lenguaje de base de datos debe ser capaz de manejar datos distribuidos en otros sistemas informáticos.

12. ***Regla de no subversion.*** Si un sistema relacional tiene un lenguaje de bajo nivel (un solo registro cada vez), ese bajo nivel no puede ser utilizado para suprimir las reglas de integridad y las restricciones expresadas en el lenguaje relacional de nivel superior (varios registros a la vez).

Por último esta regla nos dice que de existir otro lenguaje diferente al relacional)por ejemplo COBOL. C, FORTRAN, etc.) éste no podrá alterar las reglas de integridad ni modificar la estructura relacional de la base de datos.

APÉNDICE B.

Las figuras muestran los datos de las tablas utilizadas en los ejemplos.

Academic

Anocomplead	Anombre	Adomicilio	Afecalta	Anumayuda	Asueldo	Anocarrera
75212122	Carlos Cuenca	ARAGON	01/01/92	5	7000.00	38
75521245	Armando Cruz	CD NEZA	01/11/83	2	5000.00	38
75165232	Alberto Alvarez	ECATEPEC	01/02/91	6	3000.00	40
72654545	Juan Mendez	ARAGON	01/02/95	2	3000.00	32
75656566	Pedro Benitez	CD NEZA	02/02/93	2	5200.00	32

Estudian

Enocuenta	Enombre	Edomicilio	Etelefono	Efecnacim	Enocarrera
84254321	Carolina Basset	ARAGON	7715421	01/02/68	32
84321212	Juan Basset	ARAGON	7212212	05/02/53	32
84254321	Alejandro Souza	CD NEZA	6545454	05/05/69	38
82323233	Pedro Souza	ECATEPEC		11/12/67	40

Grupo

Gnogrupo	Gnomateria	Gnomaestro	Gdias	Ghora	Gsalon
1157	0076	72654545	LUMMV	11.30 13.30	A211
1158	0134	NULL	MAJU	07.00 8.30	A521
1159	0119	75656566	SA	07.00 12.00	A121
2705	0028	75212122	MAJU	17.00 19.00	A525
2706	0130	75521245	LUMMV	17.00 18.30	A808
2504	0024	75165232	MAJU	17.00 19.00	A212
2501	0139	75165232	LUMMV	17.00 18.30	A213

Materia

Mnomateria	Mnombre	Mcred	Mcostetab	Mnocarrera
0076	Bases de Datos	8	100.00	32
0134	Sistemas Digitales	8	50.00	32
0119	Estructuras de Datos	8	0.00	32
0056	Estructuras Discretas	7	0.00	32
0559	Memorias y Periferico	10	100.00	32
0561	Microcomputadoras	10	500.00	32
0028	Analisis Dinamico de Maquinas	8	100.00	38
0130	Elementos de Maquinas	8	50.00	38
0024	Circuitos Electricos	10	150.00	40
0139	Dispositivos Electronicos	10	90.00	40

Inscribe

Inogrupo	Inocuenta	Ifecalta	Ihorasalta	Inocarrera
	84254321	01/02/94	1130	32
1158	84254321	01/02/94	1130	32
1157	84321212	02/02/94	1150	32
1158	84321212	02/02/94	1150	32
1157	85132152	15/02/94	1550	38
1158	85132152	15/02/94	1550	38
1157	82323233	16/02/94	1420	40
1159	82323233	16/02/94	1420	40

Carrera

Cnocarrera	Cnombre	Cedificio	Ccoord
32	Computacion	A202	75656566
38	Electrica	A501	75521245
40	Mecanica	A501	75165232

BIBLIOGRAFÍA.

Martyn Tim.
Hartley Tim.
DB2/SQL Manual para Programadores.
Serie J. Ranade IBM.
Mc. Graw Hill.
España, 1991.

Benavides Abajo J.
Olaizola Bartolomé J.M.
Rivero Cornelio E.
SQL Para Usuarios y Programadores.
Paraninfo, 1991.

Groff James R.
Weinberg. Paul N.
Aplique SQL.
Mc. Graw Hill.
México, 1992

Tsai Alice Y.
Sistemas de Base de Datos. Administración y Uso.
Prentice Hall.
México, 1991.

Atre Shakuntala.
Técnicas de Bases de Datos. Estructuración en Diseño y Administración.
Trillas.
México, 1991.

Schildt Herbert.
Programación en Lenguaje C.
Mc. Graw Hill.
México, 1987.

Philippakis Andreas S.
Kazmier Leonard J.
COBOL Estructurado. Tercera Edición.
Mc. Graw Hill.
México, 1990.

Manuales Técnicos de los RDBMS.