**Índice**



## Tutorial de PostgreSQL, Modelo Relacional y SQL en Español (Castellano)

## Sumario

Postgres, desarrollada originalmente en el Departamento de Ciencias de la Computación de la Universidad de California en Berkeley, fue pionera en muchos de los conceptos de bases de datos relacionales orientadas a objetos que ahora empiezan a estar disponibles en algunas bases de datos comerciales. Ofrece suporte al lenguaje SQL:2003, integridad de transacciones, y extensibilidad de tipos de datos. PostgreSQL es un descendiente de dominio público y código abierto del código original de Berkeley.

## Introducción

Este documento es el manual de usuario del sistema de mantenimiento de bases de datos PostgreSQL, originariamente desarrollado en la Universidad de California en Berkeley. PostgreSQL está basada en Postgres release 4.2. El proyecto Postgres, liderado por el Porfesor Michael Stonebraker, fue esponsorizado por diversos organismos oficiales u oficiosos de los EEUU: la Agencia de Proyectos de Investigación Avanzada de la Defensa de los EEUU (DARPA), la Oficina de Investigación de la Armada (ARO), la Fundación Nacional para la Ciencia (NSF), y ESL, Inc.

## ¿Qué es Postgres?

Los sistemas de mantenimiento de Bases de Datos relacionales tradicionales (DBMS,s) soportan un modelo de datos que consisten en una colección de relaciones con nombre, que contienen atributos de un tipo específico. En los sistemas comerciales actuales, los tipos posibles incluyen numéricos de punto flotante, enteros, cadenas de caracteres, cantidades monetarias y fechas. Está generalmente reconocido que este modelo será inadecuado para las aplicaciones futuras de procesado de datos. El modelo relacional sustituyó modelos previos en parte por su "simplicidad espartana". Sin embargo, como se ha mencionado, esta simplicidad también hace muy dificil la implementación de ciertas aplicaciones. Postgres ofrece una potencia adicional sustancial al incorporar los siguientes cuatro conceptos adicionales básicos en una vía en la que los usuarios pueden extender fácilmente el sistema

- tipos de datos
- funciones
- operadores
- funciones de agregado
- métodos de indexación
- lenguajes procedurales

Otras características aportan potencia y flexibilidad adicional:

- Consultas complejas (subconsultas, joins, etc.)
- Integridad referencial (claves primarias y foráneas)
- Restricciones (Constraints)
- Disparadores (triggers)
- Vistas (views)
- Reglas (rules)
- Integridad transaccional
- Control de concurrencia multiversión

Estas características colocan a Postgres en la categoría de las Bases de Datos identificadas como *objeto-relacionales*. Nótese que éstas son diferentes de las referidas como *orientadas a objetos*, que en general no son bien aprovechables para soportar lenguajes de Bases de Datos relacionales tradicionales. Postgres tiene algunas características que son propias del mundo de las bases de datos orientadas a objetos. De hecho, algunas Bases de Datos comerciales han incorporado recientemente características en las que Postgres fue pionera.

Además, debido a su licencia liberal, PostgreSQL puede ser usado, modificado, y distibuido por cualquier persona para cualquier propósito sin cargo alguno, ya sea para un fin privado, comercial o académico.

## Breve historia de Postgres

El Sistema Gestor de Bases de Datos Relacionales Orientadas a Objetos conocido como PostgreSQL (y brevemente llamado Postgres95) está derivado del paquete Postgres escrito en Berkeley. Con cerca de una década de desarrollo tras él, PostgreSQL es el gestor de bases de datos de código abierto más avanzado hoy en día, ofreciendo control de concurrencia multi-versión, soportando casi toda la sintaxis SQL (incluyendo subconsultas,

transacciones, y tipos y funciones definidas por el usuario), contando también con un amplio conjunto de enlaces con lenguajes de programación (incluyendo C, C++, Java, perl, tcl y python).

### El proyecto Postgres de Berkeley

La implementación del DBMS Postgres comenzó en 1986. Los conceptos iniciales para el sistema fueron presentados en *The Design of Postgres* y la definición del modelo de datos inicial apareció en *ThePostgres Data Model*. El diseño del sistema de reglas fue descrito en ese momento en *The Design of the Postgres Rules System*. La lógica y arquitectura del gestor de almacenamiento fueron detalladas en*The Postgres Storage System*.

Postgres ha pasado por varias revisiones importantes desde entonces. El primer sistema de pruebas fue operacional en 1987 y fue mostrado en la Conferencia ACM-SIGMOD de 1988. Lanzamos la Versión 1, descrita en *The Implementation of Postgres*, a unos pocos usuarios externos en Junio de 1989. En respuesta a una crítica del primer sistema de reglas (*A Commentary on the Postgres Rules System*), éste fue rediseñado (*On Rules, Procedures, Caching and Views in Database Systems*) y la Versión 2, que salió en Junio de 1990, lo incorporaba. La Versión 3 apareció en 1991 y añadió una implementación para múltiples gestores de almacenamiento, un ejecutor de consultas mejorado y un sistema de reescritura de reglas nuevo. En su mayor parte, las siguientes versiones hasta el lanzamiento dePostgres95 (ver más abajo) se centraron en mejorar la portabilidad y la fiabilidad.

Postgres forma parte de la implementación de muchas aplicaciones de investigación y producción. Entre ellas: un sistema de análisis de datos financieros, un paquete de monitorización de rendimiento de motores a reacción, una base de datos de seguimiento de asteroides y varios sistemas de información geográfica. También se ha utilizado como una herramienta educativa en varias universidades. Finalmente, Illustra Information Technologies (posteriormente absorbida por Informix) tomó el código y lo comercializó. Postgres llegó a ser el principal gestor de datos para el proyecto científico de computación Sequoia 2000 a finales de 1992.

El tamaño de la comunidad de usuarios externos casi se duplicó durante 1993. Pronto se hizo obvio que el mantenimiento del código y las tareas de soporte estaban ocupando tiempo que debía dedicarse a la investigación. En un esfuerzo por reducir esta carga, el proyecto terminó oficialmente con la Versión 4.2.

### Postgres95

En 1994, Andrew Yu y Jolly Chen añadieron un intérprete de lenguage SQL a Postgres. Postgres95 fue publicado a continuación en la Web para que encontrara su propio hueco en el mundo como un descendiente de dominio público y código abierto del código original Postgres de Berkeley.

El código de Postgres95 fue adaptado a ANSI C y su tamaño reducido en un 25%. Muchos cambios internos mejoraron el rendimiento y la facilidad de mantenimiento. Postgres95 v1.0.x se ejecutaba en torno a un 30-50% más rápido en el Wisconsin Benchmark comparado con Postgres v4.2. Además de corrección de errores, éstas fueron las principales mejoras:

- El lenguage de consultas Postquel fue reemplazado con SQL (implementado en el servidor). Las subconsultas no fueron soportadas hasta PostgreSQL (ver más abajo), pero podían ser emuladas enPostgres95 con funciones SQL definidas por el usuario. Las funciones agregadas fueron reimplementadas. También se añadió una implementación de la cláusula GROUP BY. La interfaz `libpq` permaneció disponible para programas escritos en C.
- Además del programa de monitorización, se incluyó un nuevo programa (`psql`) para realizar consultas SQL interactivas usando la librería GNU `readline`.
- Una nueva librería de interfaz, `libpgtcl`, soportaba clientes basados en Tcl. Un shell de ejemplo, pgtclsh, aportaba nuevas órdenes Tcl para interactuar con el motor Postgres95 desde programas `tcl`
- Se revisó la interfaz con objetos grandes. Los objetos grandes de Inversion fueron el único mecanismo para almacenar objetos grandes (el sistema de archivos de Inversion fue eliminado).
- Se eliminó también el sistema de reglas a nivel de instancia, si bien las reglas siguieron disponibles como reglas de reescritura.
- Se distribuyó con el código fuente un breve tutorial introduciendo las características comunes de SQL y de Postgres95.
- Se utilizó GNU make (en vez de BSD make) para la compilación. Postgres95 también podía ser compilado con un gcc sin parches (al haberse corregido el problema de alineación de variables de longitud doble).

### PostgreSQL

En 1996, se hizo evidente que el nombre "Postgres95" no resistiría el paso del tiempo. Elegimos un nuevo nombre, PostgreSQL, para reflejar la relación entre el Postgres original y las versiones más recientes con capacidades SQL. Al mismo tiempo, hicimos que los números de versión partieran de la 6.0, volviendo a la secuencia seguida originalmente por el proyecto Postgres.

Durante el desarrollo de Postgres95 se hizo hincapié en identificar y entender los problemas en el código del motor de datos. Con PostgreSQL, el énfasis ha pasado a aumentar características y capacidades, aunque el trabajo continúa en todas las áreas.

Las principales mejoras en PostgreSQL incluyen:

- Los bloqueos de tabla han sido sustituidos por el control de concurrencia multi-versión, el cual permite a los accesos de sólo lectura continuar leyendo datos consistentes durante la actualización de registros, y permite copias de seguridad en caliente desde pg_dump mientras la base de datos permanece disponible para consultas.
- Se han implementado importantes características del motor de datos, incluyendo subconsultas, valores por defecto, restricciones a valores en los campos (constraints) y disparadores (triggers).
- Se han añadido funcionalidades en línea con el estándar SQL92, incluyendo claves primarias, identificadores entrecomillados, forzado de tipos cadena literales, conversión de tipos y entrada de enteros binarios y hexadecimales.
- Los tipos internos han sido mejorados, incluyendo nuevos tipos de fecha/hora de rango amplio y soporte para tipos geométricos adicionales.
- La velocidad del código del motor de datos ha sido incrementada aproximadamente en un 20-40%, y su tiempo de arranque ha bajado el 80% desde que la versión 6.0 fue lanzada.

## Terminología

En la documentación siguiente, *sitio* (o *site*) se puede interpretar como la máquina en la que está instalada Postgres. Dado que es posible instalar más de un conjunto de bases de datos Postgres en una misma máquina, este término denota, de forma más precisa, cualquier conjunto concreto de programas binarios y bases de datos de Postgres instalados.

El *superusuario* de Postgres es el usuario llamado *`postgres`* que es dueño de los ficheros de la bases de datos y binarios de Postgres. Como superusuario de la base de datos, no le es aplicable ninguno de los mecanismos de protección y puede acceder a cualquiera de los datos de forma arbitraria. Además, al superusuario de Postgres se le permite ejecutar programas de soporte que generalmente no están disponibles para todos los usuarios. Tenga en cuenta que el superusuario de Postgres *no* es el mismo que el superusuario de Unix (que es conocido como *root*). El superusuario debería tener un identificador de usuario (*UID*) distinto de cero por razones de seguridad.

El *administrador de la base de datos* (*database administrator*) o DBA, es la persona responsable de instalar Postgres con mecanismos para hacer cumplir una política de seguridad para un site. El DBA puede añadir nuevos usuarios por el método descrito más adelante y mantener un conjunto de bases de datos plantilla para usar con`createdb`.

El `postmaster` es el proceso que actúa como una puerta de control (clearing-house) para las peticiones al sistema Postgres. Las aplicaciones frontend se conectan al `postmaster`, que mantiene registros de los errores del sistema y de la comunicación entre los procesos backend. El `postmaster` puede aceptar varios argumentos desde la línea de órdenes para poner a punto su comportamiento. Sin embargo, el proporcionar argumentos es necesario sólo si se intenta trabajar con varios sitios o con uno que no se ejecuta a la manera por defecto.

El backend de Postgres (el programa ejecutable `postgres` real) lo puede ejecutar el superusuario directamente desde el intérprete de órdenes de usuario de Postgres (con el nombre de la base de datos como un argumento). Sin embargo, hacer esto elimina el buffer pool compartido y bloquea la tabla asociada con un postmaster/sitio, por ello esto no está recomendado en un sitio multiusuario.

## Notación

"..." o `/usr/local/pgsql/` delante de un nombre de fichero se usa para representar el camino (path) al directorio home del superusuario de Postgres.

En la sinopsis, los corchetes ("y?") indican una expresión o palabra clave opcional. Cualquier cosa entre llaves ("{" y "}") y que contenga barras verticales ("|") indica que debe elegir una de las opciones que separan las barras verticales.

En los ejemplos, los paréntesis ("(" y ")") se usan para agrupar expresiones booleanas. "|" es el operador booleano OR.

Los ejemplos mostrarán órdenes ejecutadas desde varias cuentas y programas. Las órdenes ejecutadas desde la cuenta del root estarán precedidas por ">". Las órdenes ejecutadas desde la cuenta del superusuario de Postgres estarán precedidas por "%", mientras que las órdenes ejecutadas desde la cuenta de un usuario sin privilegios estarán precedidas por "$". Las órdenes de SQL estarán precedidas por"=>" o no estarán precedidas por ningún prompt, dependiendo del contexto.

## Copyrights y Marcas Registradas

*La traducción de los textos de copyright se presenta aquí únicamente a modo de aclaración y no ha sido aprobada por sus autores originales. Los únicos textos de copyright, garantías, derechos y demás legalismos que tienen validez son los originales en inglés o una traducción aprobada por los autores y/o sus representantes legales. .*

En ningún caso la Universidad de California se hará responsable de daños, causados a cualquier persona o entidad, sean estos directos, indirectos, especiales, accidentales o consiguientes, incluyendo lucro cesante que resulten del uso de este software y su documentación, incluso si la Universidad ha sido notificada de la posibilidad de tales daños.

La Universidad de California rehusa específicamente ofrecer cualquier garantia, incluyendo, pero no limitada únicamente a, la garantía implícita de comerciabilidad y capacidad para cumplir un determinado propósito. El software que se distribuye aquí se entrega "tal y cual", y la Universidad de California no tiene ninguna obligación de mantenimiento, apoyo, actualización, mejoramiento o modificación.

Unix es una marca registrada de X/Open, Ltd. Sun4, SPARC, SunOS y Solaris son marcas registradas de Sun Microsystems, Inc. DEC, DECstation, Alpha AXP y ULTRIX son marcas registradas de Digital Equipment Corp. PA-RISC y HP-UX son marcas registradas de Hewlett-Packard Co. OSF/1 es marca registrada de Open Software Foundation.

# SQL

Este capítulo apareció originariamente como parte de la tesis doctoral de Stefan Simkovics. (*Simkovics, 1998*).

SQL se ha convertido en el lenguaje de consulta relacional más popular. El nombre "SQL" es una abreviatura de *Structured Query Language* (Lenguaje de consulta estructurado). En 1974 Donald Chamberlain y otros definieron el lenguaje SEQUEL (*Structured English Query Language*) en IBM Research. Este lenguaje fue implementado inicialmente en un prototipo de IBM llamado SEQUEL-XRM en 1974-75. En 1976-77 se definió una revisión de SEQUEL llamada SEQUEL/2 y el nombre se cambió a SQL.

IBM desarrolló un nuevo prototipo llamado System R en 1977. System R implementó un amplio subconjunto de SEQUEL/2 (now SQL) y un número de cambios que se le hicieron a (now SQL) durante el proyecto. System R se instaló en un número de puestos de usuario, tanto internos en IBM como en algunos clientes seleccionados. Gracias al éxito y aceptación de System R en los mismos, IBM inició el desarrollo de productos comerciales que implementaban el lenguaje SQL basado en la tecnología System R.

Durante los años siguientes, IBM y bastantes otros vendedores anunciaron productos SQL tales como SQL/DS (IBM), DB2 (IBM), ORACLE (Oracle Corp.), DG/SQL (Data General Corp.), y SYBASE (Sybase Inc.).

SQL es también un estándar oficial hoy. En 1982, la American National Standards Institute (ANSI) encargó a su Comité de Bases de Datos X3H2 el desarrollo de una propuesta de lenguaje relacional estándar. Esta propuesta fue ratificada en 1986 y consistía básicamente en el dialecto de IBM de SQL. En 1987, este estándar ANSI fue también aceptado por la Organización Internacional de Estandarización (ISO). Esta versión estándar original de SQL recibió informalmente el nombre de "SQL/86". En 1989, el estándar original fue extendido, y recibió el nuevo nombre, también informal, de "SQL/89". También en 1989 se desarrolló un estándar relacionado llamado *Database Language Embedded SQL* (ESQL).

Los comités ISO y ANSI han estado trabajando durante muchos años en la definición de una versión muy ampliada del estándar original, llamado informalmente *SQL2* o *SQL/92*. Esta versión se convirtió en un estándar ratificado durante 1992: *International Standard ISO/IEC 9075:1992, Database Language SQL*. SQL/92 es la versión a la que normalmente la gente se refiere cuando habla de «SQL estándar». Se da una descripción detallada de SQL/92 en *Date and Darwen, 1997*. En el momento de escribir este documento, se encuentra disponible un nuevo estándar denominado informalmente como *SQL:2003*. Plantea hacer de SQL un lenguaje de alcance completo (e Turing-complete language), es decir, posibilitando todas las consultas computables, (por ejemplo consultas recursivas), entre otras características.

## El Modelo de Datos Relacional

Como mencionamos antes, SQL es un lenguaje relacional. Esto quiere decir que se basa en el *modelo de datos relacional* publicado inicialmente por E.F.Codd en 1970. Daremos una descripción formal del modelo de datos relacional más tarde (en *deDatos Formalidades del Modelo Relacional de Datos*), pero primero queremos dar una mirada desde un punto de vista más intuitivo.

Una *base de datos relacional* es una base de datos que se percibe por los usuarios como una *colección de tablas* (y nada más que tablas). Una tabla consiste en filas y columnas, en las que cada fila representa un registro, y cada columna representa un atributo del registro contenido en la tabla. *La Base de Datos de Proveedores y Artículos* muestra un ejemplo de base de datos consistente en tres tablas.

- SUPPLIER es una tabla que recoge el número (SNO), el nombre (SNAME) y la ciudad (CITY) de un proveedor.
- PART es una tabla que almacena el número (PNO) el nombre (PNAME), el precio (PRICE) y la cantidad en existencia (STOCK) de un artículo.
- SELLS almacena información sobre qué artículo (PNO) es vendido por qué proveedor (SNO). Esto sirve en un sentido para conectar las dos tablas entre ellas.

### La Base de Datos de Proveedores y Artículos

**Ejemplo 1. La Base de Datos de Proveedores y Artículos**

```
  SUPPLIER   SNO |  SNAME  |  CITY      SELLS   SNO | PNO
           -----+---------+--------           -----+-----
             1  |  Smith  | London              1  |  1
             2  |  Jones  | Paris               1  |  2
             3  |  Adams  | Vienna              2  |  4
             4  |  Blake  | Rome                3  |  1
             5  |  James  | London              3  |  3
                                                 4  |  2
 PART       PNO |  PNAME    | PRICE | STOCK     4  |  3
           -----+-----------+-------+-------     4  |  4
              1 | Tornillos | 10.00 |    20
              2 | Tuercas   |  8.00 |    16
              3 | Cerrojos  | 15.00 |    30
              4 | Levas     | 25.00 |    50
              5 | Arandela  | 50.00 |   100
```

Las tablas PART y SUPPLIER se pueden ver como *entidades* y SELLS se puede ver como una *relación* entre un artículo particular y un proveedor particular.

Como veremos más tarde, SQL opera en las tablas tal como han sido definidas, pero antes de ello estudiaremos la teoría del modelo relacional.

## Formalidades del Modelo Relacional de Datos

El concepto matemático que subyace bajo el modelo relacional es la *relación* de la teoría de conjuntos, la cual es un subconjunto del producto cartesiano de una lista de dominios. Esta relación de la teoría de conjuntos proporciona al modelo su nombre (no confundir con la relación del *Modelo Entidad-Relación*). Formalmente, un dominio es simplemente un conjunto de valores. Por ejemplo, el conjunto de los enteros es un dominio. También son ejemplos de dominios las cadenas de caracteres de longitud 20 y los números reales.

El *producto cartesiano* de los dominios *D1*, *D2*, ... *Dk*, escritos *D1* × *D2* × ... × *Dk* es el conjunto de las k-tuplas *v1*, *v2*, ... *vk*, tales que *v1 D1*, *v1 D1*, ... *vk Dk*.

Por ejemplo, cuando tenemos *k*=2, *D1*=`{0,1}` y *D2*=`{a,b,c}` entonces *D1* × *D2* es `{(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)}`.

Una Relación es cualquier subconjunto del producto cartesiano de uno o más dominios: *R D1* × *D2* × ... × *Dk*.

Por ejemplo, `{(0,a),(0,b),(1,a)}` es una relación; De hecho es un subconjunto de *D1* × *D2* mencionado antes.

Los miembros de una relación se llaman tuplas. Cada relación de algún producto cartesiano *D1* × *D2* × ... × *Dk* se dice que tiene nivel k y de este modo es un subconjunto de k-tuplas.

Una relación se puede ver como una tabla (como ya dijimos, recuerde *La Base de Datos de Proveedores y Artículos* donde cada tupla se representa como una fila y cada columna corresponde a un componente de la tupla. Dando nombres (llamados atributos) a las columnas, nos acercamos a la definición de un *esquema relacional*.

Un *esquema relacional* R es un conjunto finito de atributos *A1*, *A2*, ... *Ak*. Hay un dominio *Di*, para cada atributo *Ai*, 1 <= i <= k, de donde se toman los valores de los atributos. Entonces escribimos es esquema relacional como `R(A1, A2, ... Ak)`.

> Un *esquema relacional* es sólo un juego de plantillas mientras que una *relación* es un ejemplo de un *esquema relacional*. La relación consiste en las tuplas (y pueden ser vistas como una tabla); no así el esquema relacional.

### Dominios contra Tipos de Datos

Ya hemos hablado de *dominios* en la sección anterior. Recalcar que el dominio es, formalmente, un conjunto de valores (por ejemplo el conjunto de los enteros o el de los números reales). En términos de sistemas de base de datos, hemos hablado de *tipos de datos* más que de dominios. Cuando hemos definido una tabla, hemos tomado una decisión sobre qué atributos incluir. Adicionalmente, hemos decidido qué juego de datos deberá ser almacenado en valores de los atributos. Por ejemplo, los valores de `SNAME` de la tabla `SUPPLIER` serán cadenas de caracteres, mientras que `SNO` almacenará enteros. Definimos esto asignando un tipo de datos a cada atributo. El tipo de `SNAME` será VARCHAR(20) (este es el tipo SQL para cadenas de caracteres de longitud <= 20), el tipo de `SNO` será INTEGER. Con la asignación de tipos de datos, también habremos seleccionado un dominio para un atributo. El dominio de `SNAME` es el conjunto de todas las cadenas de caracteres de longitud <= 20, mientras el dominio de `SNO` es el conjunto de todos los números enteros.

## Operaciones en el Modelo de Datos Relacional

En la sección previa (*Formalidades del Modelo Relacional de Datos*) definimos la noción matemática del modelo relacional. Ahora conocemos como los datos pueden almacenarse utilizando un modelo de datos relacional, pero no conocemos qué podemos hacer con todas estas tablas para recuperar algo desde esa base de datos todavía. Por ejemplo, alguien podría preguntar por los nombre de todos los proveedores que vendan el artículo 'tornillo'. Hay dos formas diferentes de notaciones para expresar las operaciones entre relaciones.

- El *Álgebra Relacional* es una notación algebraica, en la cual las consultas se expresan aplicando operadores especializados a las relaciones.
- El *Cálculo Relacional* es una notación lógica, donde las consultas se expresan formulando algunas restricciones lógicas que las tuplas de la respuesta deban satisfacer.

### Álgebra Relacional

El *Álgebra Relacional* fue introducida por E.F.Codd en 1972. Consiste en un conjunto de operaciones con las relaciones.

- SELECCIÓN (): extrae *tuplas* a partir de una relación que satisfagan una restricción dada. Sea *R* una tabla que contiene un atributo *A*. $_{A=a}$(R) = {t R t(A) = a} donde `t` denota una tupla de *R* y `t(A)`denota el valor del atributo *A* de la tupla `t`.
- PROYECCIÓN (): extrae *atributos* (columnas) específicos de una relación. Sea `R` una relación que contiene un atributo `X`. $_{X}$(R) = {t(X) t R}, donde `t(X)` denota el valor del atributo `X` de la tupla `t`.
- PRODUCTO CARTESIANO (×): construye el producto cartesiano de dos relaciones. Sea `R` una tabla de rango (arity) $k_1$ y sea `S` una tabla con rango (arity) $k_2$. `R` × `S` es el conjunto de las $k_1 + k_2$-tuplas cuyos primeros$k_1$ componentes forman una tupla en `R` y cuyos últimos $k_2$ componentes forman una tupla en `S`.
- UNION (): supone la unión de la teoría de conjuntos de dos tablas. Dadas las tablas `R` y `S` (y ambas deben ser del mismo rango), la unión `R` `S` es el conjunto de las tuplas que están en `R, S` o en las dos.
- INTERSECCIÓN (): Construye la intersección de la teoría de conjuntos de dos tablas. Dadas las tablas `R` y `S`, `R` `S` es el conjunto de las tuplas que están en `R` y en `S`. De nuevo requiere que `R` y `S` tengan el mismo rango.
- DIFERENCIA ( or ): supone el conjunto diferencia de dos tablas. Sean `R` y `S` de nuevo dos tablas con el mismo rango. `R` - `S` Es el conjunto de las tuplas que están en `R` pero no en `S`.
- JUNTA NATURAL ( o |×|): conecta dos tablas por sus atributos comunes. Sea `R` una tabla con los atributos `A`,`B` y `C` y sea `S` una tabla con los atributos `C`,`D` y `E`. Hay un atributo común para ambas relaciones, el atributo`C`. R S = $_{R.A,R.B,R.C,S.D,S.E}$($_{R.C=S.C}$(R × S)). ¿Qué estamos haciendo aquí? Primero calculamos el producto cartesiano `R` × `S`. Entonces seleccionamos las tuplas cuyos valores para el atributo común `C` sea igual ($_{R.C = S.C}$). Ahora tenemos una tabla que contiene el atributo `C` dos veces y lo corregimos eliminando la columna duplicada.

**Ejemplo 2. Una Inner Join (Una Junta Interna)**

Veamos las tablas que se han producido evaluando los pasos necesarios para una join. Sean las siguientes tablas dadas:

```
        R   A | B | C      S   C | D | E
          ---+---+---        ---+---+---
           1 | 2 | 3          3 | a | b
           4 | 5 | 6          6 | c | d
           7 | 8 | 9
```

Primero calculamos el producto cartesiano `R` × `S` y tendremos:

```
      R x S   A | B | R.C | S.C | D | E
            ---+---+-----+-----+---+---
             1 | 2 |  3  |  3  | a | b
             1 | 2 |  3  |  6  | c | d
             4 | 5 |  6  |  3  | a | b
             4 | 5 |  6  |  6  | c | d
             7 | 8 |  9  |  3  | a | b
             7 | 8 |  9  |  6  | c | d
```

Tras la selección $_{R.C=S.C}$(R × S) tendremos:

```
              A | B | R.C | S.C | D | E
            ---+---+-----+-----+---+---
             1 | 2 |  3  |  3  | a | b
             4 | 5 |  6  |  6  | c | d
```

Para eliminar las columnas duplicadas S.C realizamos la siguiente operación: $_{R.A,R.B,R.C,S.D,S.E}(_{R.C=S.C}(R \times S))$ y obtenemos:

```
 A | B | C | D | E
---+---+---+---+---
 1 | 2 | 3 | a | b
 4 | 5 | 6 | c | d
```

- DIVISIÓN (÷): Sea R una tabla con los atributos A, B, C, y D y sea S una tabla con los atributos C y D. Definimos la división como: R ÷ S = {t $t_s$ S $t_r$ R tal que $t_r$(A,B)=t$t_r$(C,D)=$t_s$} donde $t_r$(x,y) denota una tupla de la tabla R que consiste sólo en los componentes x y y. Nótese que la tupla t consiste sólo en los componentes A y B de la relación R.

Dadas las siguientes tablas

```
R   A | B | C | D        S   C | D
  ---+---+---+---          ---+---
   a | b | c | d            c | d
   a | b | e | f            e | f
   b | c | e | f
   e | d | c | d
   e | d | e | f
   a | b | d | e
```

R ÷ S se deriva como

```
 A | B
---+---
 a | b
 e | d
```

Para una descripción y definición más detallada del Álgebra Relacional diríjanse a [*Ullman, 1988*] o [*Date, 1994*].

**Ejemplo 3. Una consulta utilizando Álgebra Relacional**

Recalcar que hemos formulado todos estos operadores relacionales como capaces de recuperar datos de la base de datos. Volvamos a nuestro ejemplo de la sección previa (*Operaciones en el Modelo de Datos Relacional*) donde alguien quería conocer los nombres de todos los proveedores que venden el artículo `Tornillos`. Esta pregunta se responde utilizando el álgebra relacional con la siguiente operación:

```
SUPPLIER.SNAME(PART.PNAME='Tornillos'(SUPPLIER  SELLS  PART))
```

Llamamos a estas operaciones una consulta. Si evaluamos la consulta anterior contra las tablas de nuestro ejemplo (*La Base de Datos de Proveedores y Artículos*) obtendremos el siguiente ejemplo:

```
 SNAME
-------
 Smith
 Adams
```

### Cálculo Relacional

El Cálculo Relacional se basa en la *lógica de primer orden*. Hay dos variantes del cálculo relacional:

- El *Cálculo Relacional de Dominios* (DRC), donde las variables esperan componentes (atributos) de las tuplas.
- El *Cálculo Relacional de Tuplas* The *Tuple Relational Calculus* (TRC), donde las variables esperan tuplas.

Expondremos sólo el cálculo relacional de tuplas porque es el único utilizado por la mayoría de lenguajes relacionales. Para una discusión detallada de DRC (y también de TRC) vea [*Date, 1994*] o [*Ullman, 1988*].

### Cálculo Relacional de Tuplas

Las consultas utilizadas en TRC tienen el siguiente formato: x(A) F(x) donde `x` es una variable de tipo tupla, `A` es un conjunto de atributos y `F` es una fórmula. La relación resultante consiste en todas las tuplas `t(A)` que satisfagan `F(t)`.

Si queremos responder la pregunta del ejemplo `*Una consulta utilizando Álgebra Relacional* utilizando TRC formularemos la siguiente consulta:

```
    {x(SNAME)  x  SUPPLIER
                      y  SELLS  z  PART (y(SNO)=x(SNO)
                      z(PNO)=y(PNO)
                      z(PNAME)='Tornillos')}
```

Evaluando la consulta contra las tablas de *La Base de Datos de Proveedores y Artículos* encontramos otra vez el mismo resultado de *Una consulta utilizando Álgebra Relacional*.

#### Álgebra Relacional contra Cálculo Relacional

El álgebra relacional y el cálculo relacional tienen el mismo *poder de expresión*; es decir, todas las consultas que se pueden formular utilizando álgebra relacional pueden también formularse utilizando el cálculo relacional, y viceversa. Esto fue probado por E. F. Codd en 1972. Este profesor se basó en un algoritmo ("algoritmo de reducción de Codd") mediante el cual una expresión arbitraria del cálculo relacional se puede reducir a la expresión semánticamente equivalente del álgebra relacional. Para una discusión más detallada sobre este punto, diríjase a [*Date, 1994*] y [*Ullman, 1988*].

Se dice a veces que los lenguajes basados en el cálculo relacional son de "más alto nivel" o "más declarativos" que los basados en el álgebra relacional porque el álgebra especifica (parcialmente) el orden de las operaciones, mientras el cálculo lo traslada a un compilador o interprete que determina el orden de evaluación más eficiente.

### El Lenguaje SQL

Como en el caso de los más modernos lenguajes relacionales, SQL está basado en el cálculo relacional de tuplas. Como resultado, toda consulta formulada utilizando el cálculo relacional de tuplas ( o su equivalente, el álgebra relacional) se pude formular también utilizando SQL. Hay, sin embargo, capacidades que van más allá del cálculo o del álgebra relaciona. Aquí tenemos una lista de algunas características proporcionadas por SQL que no forman parte del álgebra y del cálculo relacionales:

- Comandos para inserción, borrado o modificación de datos.
- Capacidades aritméticas: En SQL es posible incluir operaciones aritméticas así como comparaciones, por ejemplo A < B + 3. Nótese que ni + ni otros operadores aritméticos aparecían en el álgebra relacional ni en cálculo relacional.
- Asignación y comandos de impresión: es posible imprimir una relación construida por una consulta y asignar una relación calculada a un nombre de relación.
- Funciones agregadas: Operaciones tales como *promedio (average)*, *suma (sum)*, *máximo (max)*, etc. se pueden aplicar a las columnas de una relación para obtener una cantidad única.

#### SELECT

El comando más usado en SQL es la instrucción SELECT, que se utiliza para recuperar datos. La sintaxis básica es (ver sintaxis avanzada):

```
 SELECT [ALL|DISTINCT]
       { * | expr_1 [AS c_alias_1] [, ...
             [, expr_k [AS c_alias_k]]]}
FROM table_name_1 [t_alias_1]
     [, ... [, table_name_n [t_alias_n]]]
[WHERE condition]
[GROUP BY name_of_attr_i
          [,... [, name_of_attr_j]] [HAVING condition]]
[{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
[ORDER BY name_of_attr_i [ASC|DESC]
          [, ... [, name_of_attr_j [ASC|DESC]]]];
```

Ilustraremos ahora la compleja sintaxis de la instrucción SELECT con varios ejemplos. Las tablas utilizadas para los ejemplos se definen en: *La Base de Datos de Proveedores y Artículos*.

#### SELECT sencillas

Aquí tenemos algunos ejemplos sencillos utilizando la instrucción SELECT:

**Ejemplo 4. Query sencilla con cualificación**

Para recuperar todas las tuplas de la tabla PART donde el atributo PRICE es mayor que 10, formularemos la siguiente consulta:

```
SELECT * FROM PART
 WHERE PRICE > 10;
```

y obtenemos la siguiente tabla:

```
 PNO |  PNAME    | PRICE | STOCK
-----+-----------+-------+-------
   1 | Tornillos | 10.00 |    20
   2 | Tuercas   |  8.00 |    16
   3 | Cerrojos  | 15.00 |    30
   4 | Levas     | 25.00 |    50
   5 | Arandela  | 50.00 |   100
```

Utilizando "*" en la instrucción SELECT solicitaremos todos los atributos de la tabla. Si queremos recuperar sólo los atributos PNAME y PRICE de la tabla PART utilizaremos la instrucción:

```
 SELECT PNAME, PRICE
FROM PART
WHERE PRICE > 10;
```

En este caso el resultado es:

```
  PNAME      |  PRICE
-------------+--------
 Cerrojos    |   15
 Levas       |   25
 Arandelas   |   50
```

Nótese que la SELECT SQL corresponde a la "proyección" en álgebra relaciona, no a la "selección" (vea *Álgebra Relacional* para más detalles).Las cualificaciones en la clausula WHERE pueden también conectarse lógicamente utilizando las palabras claves OR, AND, y NOT:

```
 SELECT PNAME, PRICE
FROM PART
WHERE PNAME = 'Cerrojos' AND
      (PRICE = 0 OR PRICE < 15);
```

dará como resultado:

```
  PNAME      |  PRICE
-------------+--------
 Cerrojos    |   15
```

Las operaciones aritméticas se pueden utilizar en la lista de objetivos y en la clausula WHERE. Por ejemplo, si queremos conocer cuanto cuestan si tomamos dos piezas de un artículo, podríamos utilizar la siguiente consulta:

```
 SELECT PNAME, PRICE * 2 AS DOUBLE
FROM PART
WHERE PRICE * 2 < 50;
```

y obtenemos:

```
  PNAME      |  DOUBLE
-------------+---------
 Tornillos   |    20
 Tuercas     |    16
```

```
                    Cerrojos   |    30
```

Nótese que la palabra DOBLE tras la palabra clave AS es el nuevo título de la segunda columna. Esta técnica puede utilizarse para cada elemento de la lista objetivo para asignar un nuevo título a la columna resultante. Este nuevo título recibe el calificativo de "un alias". El alias no puede utilizarse en todo el resto de la consulta.

Si queremos ordenar los resultados, ordenamos por la clausula ORDER BY:

```
   SELECT PNAME, PRICE * 2 AS DOUBLE
  FROM PART
  WHERE PRICE * 2 < 50
  ORDER BY PRICE * 2;
```

y obtendremos:

```
   PNAME   | DOUBLE
-----------+--------
Tuercas   |  16.00
Tornillos |  20.00
Cerrojos  |  30.00
```

Notar que el orden es ascendente y determinado por la expresión PRICE * 2. Si quisieramos otro orden, podríamos utilizar cualquier otra expresión y la clausula DESC (descendente) y ASC (ascendente).

**Joins (Cruces)**

El siguiente ejemplo muestra como las *joins (cruces)* se realizan en SQL.

Para cruzar tres tablas SUPPLIER, PART y SELLS a través de sus atributos comunes, formularemos la siguiente instrucción:

```
   SELECT S.SNAME, P.PNAME
  FROM SUPPLIER S, PART P, SELLS SE
  WHERE S.SNO = SE.SNO AND
        P.PNO = SE.PNO;
```

y obtendremos la siguiente tabla como resultado:

```
                      SNAME | PNAME
                     -------+-------
                      Smith | Tornillos
                      Smith | Tuercas
                      Jones | Levas
                      Adams | Tornillos
                      Adams | Cerrojos
                      Blake | Tuercas
                      Blake | Cerrojos
                      Blake | Levas
```

En la clausula FROM hemos introducido un alias al nombre para cada relación porque hay atributos con nombre común (SNO y PNO) en las relaciones. Ahora podemos distinguir entre los atributos con nombre común simplificando la adicción de un prefijo al nombre del atributo con el nombre del alias seguido de un punto. La join se calcula de la misma forma, tal como se muestra en *Una Inner Join (Una Join Interna)*. Primero el producto cartesiano: SUPPLIER × PART × SELLS Ahora seleccionamos únicamente aquellas tuplas que satisfagan las condiciones dadas en la clausula WHERE (es decir, los atributos con nombre común deben ser iguales). Finalmente eliminamos las columnas repetidas (S.SNAME, P.PNAME).

Podemos obtener el mismo resultado usando la sintaxis propia de INNER JOIN, donde se cruza de a dos tablas sobre sus atributos que satisfagan la condición de la clausula ON:

```
   SELECT S.SNAME, P.PNAME
  FROM SUPPLIER S INNER JOIN SELLS SE ON S.SNO = SE.SNO AND
```

```
      INNER JOIN PART P ON P.PNO = SE.PNO;
```

A su vez, en este caso, podemos obtener el mismo resultado usando la sintaxis de NATURAL JOIN o junta natural ( o |×|) ya que los atributos comunes tienen el mismo nombre y tipo (se satisface la condición S.SNO = SE.SNO y P.PNO = SE.PNO implícitamente, al llamarse SNO y PNO en ambas tablas):

```
  SELECT S.SNAME, P.PNAME
FROM SUPPLIER S NATURAL JOIN SELLS SE
     NATURAL JOIN PART P;
```

Si quisieramos obtener todas las partes (PART) disponibles y el nombre del proveedor en el caso que sea provista por alguno (suponiendo que hay partes que no son provistas por ningún proveedor, en dicho caso no figuran en la relación SELLS), podríamos utilizar la sintaxis LEFT JOIN:

```
  SELECT P.PNAME, S.SNAME
FROM PART P LEFT JOIN SELLS SE ON P.PNO=SE.PNO
     LEFT JOIN SUPPLIER S ON S.SNO=SE.SNO;
```

Lo que nos devolvería son todas las filas que satisfagan el producto cartesiano de PART × SELLS (todas las que cumplan la condición de la clausula ON P.PNO=SE.PNO), más una copia de cada fila en la tabla izquierda (PART) para la cual ninguna fila de la tabla derecha (SELLS) cumplió la condición. Estas filas izquierdas son extendidas con valores nulos para las columnas de la tabla derecha. Lo mismo aplica para el resultado del LEFT JOIN entre el PART y SELLS con SUPPLIER:

```
   PNAME   | SNAME
-----------+-------
Tornillos | Smith
Tornillos | Adams
Tuercas   | Smith
Tuercas   | Blake
Cerrojos  | Adams
Cerrojos  | Blake
Levas     | Blake
Levas     | Jones
Arandela  |
```

En este caso, nos indica que la parte 'Arandela' no es provista por ningún proveedor (está en la relación PART pero no en SELLS).

Convenientemente, existe la junta derecha o RIGHT JOIN cuyo funcionamiento es similar, pero completando con las filas de la tabla derecha que no encontraron correspondencia on la tabla izquierda (al inverso que LEFT JOIN).

**Operadores Agregados**

SQL proporciona operadores agregados (como son AVG, COUNT, SUM, MIN, MAX) que toman el nombre de un atributo como argumento. El valor del operador agregado se calcula sobre todos los valores de la columna especificada en la tabla completa. Si se especifican grupos en la consulta, el cálculo se hace sólo sobre los valores de cada grupo (vean la siguiente sección).

**Ejemplo 5. Aggregates**

Si queremos conocer el coste promedio de todos los artículos de la tabla PART, utilizaremos la siguiente consulta:

```
  SELECT AVG(PRICE) AS AVG_PRICE
FROM PART;
```

El resultado es:

```
                      AVG_PRICE
                     -----------
                       21.6
```

Si queremos conocer cuantos artículos se recogen en la tabla PART, utilizaremos la instrucción:

```
  SELECT COUNT(PNO)
FROM PART;
```

y obtendremos:

```
                        COUNT
                       -------
                         5
```

Si queremos conocer cuantos artículos tenemos en existencia (STOCK), y su respectiva valorización (STOCK*PRICE), utilizaremos la instrucción:

```
  SELECT SUM(STOCK) AS QTY, SUM(STOCK*PRICE) AS AMOUNT
FROM PART;
```

y obtendremos:

```
 QTY | AMOUNT
-----+---------
216 | 7028.00
```

Notar que en este ejemplo estamos cambiando el nombre de las columnas (por defecto SUM), a QTY (cantidad) y AMOUNT (importe total).

**Agregación por Grupos**

SQL nos permite particionar las tuplas de una tabla en grupos. En estas condiciones, los operadores agregados descritos antes pueden aplicarse a los grupos (es decir, el valor del operador agregado no se calculan sobre todos los valores de la columna especificada, sino sobre todos los valores de un grupo. El operador agregado se calcula individualmente para cada grupo).

El particionamiento de las tuplas en grupos se hace utilizando las palabras clave **GROUP BY** seguidas de una lista de atributos que definen los grupos. Si tenemos **GROUP BY $A_1$, , $A_k$** habremos particionado la relación en grupos, de tal modo que dos tuplas son del mismo grupo si y sólo si tienen el mismo valor en sus atributos $A_1$, , $A_k$.

**Ejemplo 6. Agregados**

Si queremos conocer cuántos artículos han sido vendidos por cada proveedor formularemos la consulta:

```
  SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME;
```

y obtendremos:

```
                    SNO | SNAME | COUNT
                   -----+-------+-------
                     1  | Smith |   2
                     2  | Jones |   1
                     3  | Adams |   2
                     4  | Blake |   3
```

Demos ahora una mirada a lo que está ocurriendo aquí. Primero, la join de las tablas SUPPLIER y SELLS:

```
                  S.SNO | S.SNAME | SE.PNO
                 -------+---------+--------
                    1   |  Smith  |   1
                    1   |  Smith  |   2
                    2   |  Jones  |   4
                    3   |  Adams  |   1
                    3   |  Adams  |   3
                    4   |  Blake  |   2
```

```
   4   |  Blake  |   3
   4   |  Blake  |   4
```

Ahora particionamos las tuplas en grupos reuniendo todas las tuplas que tiene el mismo atributo en S.SNO y S.SNAME:

```
  S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
       |         |   2
-------------------------
   2   |  Jones  |   4
-------------------------
   3   |  Adams  |   1
       |         |   3
-------------------------
   4   |  Blake  |   2
       |         |   3
```

En nuestro ejemplo, obtenemos cuatro grupos y ahora podemos aplicar el operador agregado COUNT para cada grupo, obteniendo el resultado total de la consulta dada anteriormente.

Nótese que para el resultado de una consulta utilizando GROUP BY y operadores agregados para dar sentido a los atributos agrupados, debemos primero obtener la lista objetivo. Los demás atributos que no aparecen en la clausula GROUP BY se seleccionarán utilizando una función agregada. Por otro lado, no se pueden utilizar funciones agregadas en atributos que aparecen en la clausula GROUP BY.

**HAVING**

La clausula HAVING trabaja de forma muy parecida a la clausula WHERE, y se utiliza para considerar sólo aquellos grupos que satisfagan la cualificación dada en la misma. Las expresiones permitidas en la clausula HAVING deben involucrar funcionen agregadas. Cada expresión que utilice sólo atributos planos deberá recogerse en la clausula WHERE. Por otro lado, toda expresión que involucre funciones agregadas debe aparecer en la clausula HAVING.

**Ejemplo 7. Having**

Si queremos solamente los proveedores que venden más de un artículo, utilizaremos la consulta:

```
 SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME
HAVING COUNT(SE.PNO) > 1;
```

y obtendremos:

```
  SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  3  | Adams |   2
  4  | Blake |   3
```

**Subconsultas**

En las clausulas WHERE y HAVING se permite el uso de subconsultas (subselects) en cualquier lugar donde se espere un valor. En este caso, el valor debe derivar de la evaluación previa de la subconsulta. El uso de subconsultas amplía el poder expresivo de SQL.

**Ejemplo 8. Subselect**

Si queremos conocer los artículos que tienen mayor precio que el artículo llamado 'Tornillos', utilizaremos la consulta:

```
 SELECT *
FROM PART
```

```
  WHERE PRICE > (SELECT PRICE FROM PART
                 WHERE PNAME='Tornillos');
```

El resultado será:

```
                   PNO |  PNAME      |  PRICE
                  -----+-------------+--------
                    3  |  Cerrojos   |   15
                    4  |  Levas      |   25
```

Cuando revisamos la consulta anterior, podemos ver la palabra clave SELECT dos veces. La primera al principio de la consulta - a la que nos referiremos como la SELECT externa - y la segunda en la clausula WHERE, donde empieza una consulta anidada - nos referiremos a ella como la SELECT interna. Para cada tupla de la SELECT externa, la SELECT interna deberá ser evaluada. Tras cada evaluación, conoceremos el precio de la tupla llamada 'Tornillos', y podremos chequear si el precio de la tupla actual es mayor.

Si queremos conocer los artículos que no son vendidos por ningún proveedor, podemos utilizar la sintáxis LEFT JOIN explicada anteriormente o utilizaremos la siguiente consulta:

```
SELECT *
  FROM PART
  WHERE PNO NOT IN (SELECT PNO FROM SELLS);
```

El resultado será todos los artículos cuyo número (PNO) no figura en la tabla SELLS (no esta asociado a algún proveedor):

```
 PNO |  PNAME   | PRICE
-----+----------+-------
  5 | Arandela | 50.00
```

Si queremos conocer todos los proveedores que no venden ningún artículo (por ejemplo, para poderlos eliminar de la base de datos), utilizaremos:

```
   SELECT *
  FROM SUPPLIER S
  WHERE NOT EXISTS
            (SELECT * FROM SELLS SE
             WHERE SE.SNO = S.SNO);
```

En nuestro ejemplo, obtendremos un resultado:.

```
 SNO | SNAME |  CITY
-----+-------+--------
  5 | James | London
```

Nótese que utilizamos S.SNO de la SELECT externa en la clausula WHERE de la SELECT interna. Como hemos descrito antes, la subconsulta se evalúa para cada tupla de la consulta externa, es decir, el valor de S.SNO se toma siempre de la tupla actual de la SELECT externa.

**Unión, Intersección, Excepción**

Estas operaciones calculan la unión, la intersección y la diferencia de la teoría de conjuntos de las tuplas derivadas de dos subconsultas.

**Ejemplo 9. Union, Intersect, Except**

La siguiente consulta es un ejemplo de UNION:

```
   SELECT S.SNO, S.SNAME, S.CITY
  FROM SUPPLIER S
  WHERE S.SNAME = 'Jones'
  UNION
  SELECT S.SNO, S.SNAME, S.CITY
  FROM SUPPLIER S
  WHERE S.SNAME = 'Adams';
```

Dará el resultado:

```
  SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

Aquí tenemos un ejemplo para INTERSECT:

```
 SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
INTERSECT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 2;
```

que dará como resultado:

```
  SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
La única tupla devuelta por ambas partes de la consulta es la única que tiene $SNO=2$.
```

Finalmente, un ejemplo de EXCEPT:

```
 SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
EXCEPT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 3;
```

que dará como resultado:

```
  SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

**Definición de Datos (DDL)**

El lenguaje SQL incluye un conjunto de comandos para definición de datos.

**CREATE TABLE**

El comando fundamental para definir datos es el que crea una nueva relación (una nueva tabla). La sintaxis básica del comando CREATE TABLE es (ver sintaxis avanzada):

```
CREATE TABLE table_name
            (name_of_attr_1 type_of_attr_1
             [, name_of_attr_2 type_of_attr_2
             [, ...]]);
```

**Ejemplo 10. Creación de una tabla**

Para crear las tablas definidas en *La Base de Datos de Proveedores y Artículos* se utilizaron las siguientes instrucciones de SQL:

```
 CREATE TABLE SUPPLIER
            (SNO   INTEGER PRIMARY KEY,
             SNAME VARCHAR(20) NOT NULL UNIQUE,
             CITY  VARCHAR(20) CHECK (CITY IN ('London', 'Paris', 'Rome', 'Vienna')));

CREATE TABLE PART
            (PNO   INTEGER PRIMARY KEY,
             PNAME VARCHAR(20) NOT NULL UNIQUE,
             PRICE DECIMAL(4 , 2) DEFAULT 1.00,
             STOCK INTEGER CHECK (STOCK>0));

CREATE TABLE SELLS
            (SNO INTEGER REFERENCES SUPPLIER(SNO),
             PNO INTEGER REFERENCES PART(PNO),
             PRIMARY KEY (SNO, PNO));
```

Notar que en este ejemplo estamos definiendo las claves primarias (PRIMARY KEY), que identifican cada fila unequivocamente:

- SNO es clave primaria (PK) de SUPPLIER
- PNO es clave primaria (PK) de PART
- SNO, PNO es clave primaria (PK compuesta) de SELLS (recordemos que las claves primarias no pueden ser nulas, por lo que nos aseguramos de que esten presentes ambos valores)

También definimos claves foráneas (FK o FOREIGN KEY) a SELLS sobre SUPPLIER y PART para integridad referencial (para no poder referenciar un proveedor o artículo inexistente)

Adicionalmente, definimos ciertas restricciones (CONSTRAINTS) sobre los datos:

- La ciudad del proveedor puede ser solo alguna de 'London', 'Paris', 'Roma', 'Viena'
- La cantidad en existencia (STOCK) debe ser 0 o positiva
- El nombre del artículo (PNAME) y del proveedor (SNAME) no deben ser nulos (NOT NULL) y no se pueden repetir (únicos o UNIQUE)

Las restricciones CHECK pueden ser cualquier expresión condicional que se deba cumplir sobre uno o más campos para que el registro se considere válido y pueda agregarse a la tabla.

Por último, definimos un valor por defecto para el precio (PRICE) de 1.00. En el caso que en el momento de agregar el registro no informemos el precio, se tomará 1.00 por defecto.

**Tipos de Datos en SQL**

A continuación sigue una lista de algunos tipos de datos soportados por SQL:

- INTEGER: entero binario con signo de palabra completa (31 bits de precisión).
- SMALLINT: entero binario con signo de media palabra (15 bits de precisión).
- NUMERIC (*p*[,*q*]): número decimal con signo de *p* dígitos de precisión, asumiendo *q* a la derecha para el punto decimal. (15 *p* *q*q 0). Si *q* se omite, se asume que vale 0.
- FLOAT: numérico con signo de doble palabra y coma flotante.
- CHAR(*n*): cadena de caracteres de longitud fija, de longitud *n*.
- VARCHAR(*n*): cadena de caracteres de longitud variable, de longitud máxima *n*.

**CREATE INDEX**

Se utilizan los índices para acelerar el acceso a una relación. Si una relación R tiene un índice en el atributo A podremos recuperar todas la tuplas $t$ que tienen $t$(A) = $a$ en un tiempo aproximadamente proporcional al número de tales tuplas $t$ más que en un tiempo proporcional al tamaño de R.

Para crear un índice en SQL se utiliza el comando CREATE INDEX. La sintaxis es:

```
 CREATE INDEX index_name
ON table_name ( name_of_attribute );
```

**Ejemplo 11. Create Index**

Para crear un índice llamado I sobre el atributo SNAME de la relación SUPPLIER, utilizaremos la siguiente instrucción:

```
  CREATE INDEX I
ON SUPPLIER (SNAME);
```

El índice creado se mantiene automáticamente. es decir, cada vez que una nueva tupla se inserte en la relación SUPPLIER, se adaptará el índice I. Nótese que el único cambio que un usuario puede percibir cuando se crea un índice es un incremento en la velocidad.

**CREATE VIEW**

Se puede ver una vista como una *tabla virtual*, es decir, una tabla que no existe *físicamente* en la base de datos, pero aparece al usuario como si existiese. Por contra, cuando hablamos de una *tabla base*, hay realmente un equivalente almacenado para cada fila en la tabla en algún sitio del almacenamiento físico.

Las vistas no tienen datos almacenados propios, distinguibles y físicamente almacenados. En su lugar, el sistema almacena la definición de la vista (es decir, las reglas para acceder a las tablas base físicamente almacenadas para materializar la vista) en algún lugar de los catálogos del sistema (vea *System Catalogs*). Para una discusión de las diferentes técnicas para implementar vistas, refiérase a *SIM98*.

En SQL se utiliza el comando **CREATE VIEW** para definir una vista. La sintaxis es:

```
  CREATE VIEW view_name
AS select_stmt
```

donde *`select_stmt`* es una instrucción select válida, como se definió en *wiki:TutorialPostgreSql#Select Select]*. Nótese que *`select_stmt`* no se ejecuta cuando se crea la vista. Simplemente se almacena en los *catálogos del sistema* y se ejecuta cada vez que se realiza una consulta contra la vista.Sea la siguiente definición de una vista (utilizamos de nuevo las tablas de *La Base de Datos de Proveedores y Artículos* ):

```
  CREATE VIEW London_Suppliers
    AS SELECT S.SNAME, P.PNAME
       FROM SUPPLIER S, PART P, SELLS SE
       WHERE S.SNO = SE.SNO AND
             P.PNO = SE.PNO AND
             S.CITY = 'London';
```

Ahora podemos utilizar esta *relación virtual* `London_Suppliers` como si se tratase de otra tabla base:

```
  SELECT *
FROM London_Suppliers
WHERE P.PNAME = 'Tornillos';
```

Lo cual nos devolverá la siguiente tabla:

```
                     SNAME | PNAME
                   -------+----------
                    Smith | Tornillos
```

Para calcular este resultado, el sistema de base de datos ha realizado previamente un acceso *oculto* a las tablas de la base SUPPLIER, SELLS y PART. Hace esto ejecutando la consulta dada en la definición de la vista contra aquellas tablas base. Tras eso, las cualificaciones adicionales (dadas en la consulta contra la vista) se podrán aplicar para obtener la tabla resultante.

**DROP TABLE, DROP INDEX, DROP VIEW**

Se utiliza el comando DROP TABLE para eliminar una tabla (incluyendo todas las tuplas almacenadas en ella):

```
  DROP TABLE table_name;
```

Para eliminar la tabla SUPPLIER, utilizaremos la instrucción:

```
  DROP TABLE SUPPLIER;
```

Se utiliza el comando DROP INDEX para eliminar un índice:

```
  DROP INDEX index_name;
```

Finalmente, eliminaremos una vista dada utilizando el comando DROP VIEW:

```
  DROP VIEW view_name;
```

### Manipulación de Datos (DML)

#### INSERT INTO

Una vez que se crea una tabla (vea *Create Table*), puede ser llenada con tuplas mediante el comando **INSERT INTO**. La sintaxis básica (ver sintaxis avanzada) es:

```
  INSERT INTO table_name (name_of_attr_1
                          [, name_of_attr_2 [,...]])
VALUES (val_attr_1
        [, val_attr_2 [, ...]]);
```

Para insertar la primera tupla en la relación SUPPLIER (de *La Base de Datos de Proveedores y Artículos*) utilizamos la siguiente instrucción:

```
  INSERT INTO SUPPLIER (SNO, SNAME, CITY)
VALUES (1, 'Smith', 'London');
```

Para insertar la primera tupla en la relación SELLS, utilizamos:

```
  INSERT INTO SELLS (SNO, PNO)
VALUES (1, 1);
```

Para insertar los datos en las tablas definidas en *La Base de Datos de Proveedores y Artículos* se utilizaron las siguientes instrucciones de SQL:

```
INSERT INTO supplier (sno, sname, city) VALUES
(1, 'Smith', 'London'),
(2, 'Jones', 'Paris'),
(3, 'Adams', 'Vienna'),
(4, 'Blake', 'Rome'),
(5, 'James', 'London');

INSERT INTO part (pno, pname, price, stock) VALUES
(1, 'Tornillos', 10, 20),
(2, 'Tuercas', 8, 16),
(3, 'Cerrojos', 15, 30),
(4, 'Levas', 25, 50),
(5, 'Arandelas', 50, 100);

INSERT INTO sells (sno, pno) VALUES
(1, 1),
(1, 2),
(2, 4),
(3, 1),
(3, 3),
(4, 2),
(4, 3),
(4, 4);
```

#### UPDATE

Para cambiar uno o más valores de atributos de tuplas en una relación, se utiliza el comando UPDATE. La sintaxis básica (ver sintaxis avanzada) es:

```
 UPDATE table_name
SET name_of_attr_1 = value_1
    [, ... [, name_of_attr_k = value_k]]
WHERE condition;
```

Para cambiar el valor del atributo PRICE en el artículo 'Tornillos' de la relación PART, utilizamos:

```
 UPDATE PART
SET PRICE = 15
WHERE PNAME = 'Tornillos';
```

El nuevo valor del atributo PRICE de la tupla cuyo nombre es 'Tornillos' es ahora 15.

**DELETE**

Para borrar una tupla de una tabla particular, utilizamos el comando DELETE FROM. La sintaxis básica (ver sintaxis avanzada) es:

```
 DELETE FROM table_name
WHERE condition;
```

Para borrar el proveedor llamado 'Smith' de la tabla SUPPLIER, utilizamos la siguiente instrucción:

```
 DELETE FROM SUPPLIER
WHERE SNAME = 'Smith';
```

### System Catalogs

En todo sistema de base de datos SQL se emplean *catálogos de sistema* para mantener el control de qué tablas, vistas, índices, etc están definidas en la base de datos. Estos catálogos del sistema se pueden investigar como si de cualquier otra relación normal se tratase. Por ejemplo, hay un catálogo utilizado para la definición de vistas. Este catálogo almacena la consulta de la definición de la vista. Siempre que se hace una consulta contra la vista, el sistema toma primero la *consulta de definición de la vista* del catálogo y materializa la vista antes de proceder con la consulta del usuario (vea *SIM98* para obtener una descripción más detallada). Diríjase a*DATE* para obtener más información sobre los catálogos del sistema.

### SQL Embebido

En esta sección revisaremos como se puede embeber SQL en un lenguaje de host (p.e. C). Hay dos razones principales por las que podríamos querer utilizar SQL desde un lenguaje de host:

- Hay consultas que no se pueden formular utilizando SQL puro (por ejemplo, las consultas recursivas). Para ser capaz de realizar esas consultas necesitamos un lenguaje de host de mayor poder expresivo que SQL.
- Simplemente queremos acceder a una base de datos desde una aplicación que está escrita en el lenguaje del host (p.e. un sistema de reserva de billetes con una interface gráfica escrita en C, y la información sobre los billetes está almacenada en una base de datos que puede accederse utilizando SQL embebido).

Un programa que utiliza SQL embebido en un lenguaje de host consiste en instrucciones del lenguaje del host e instrucciones de *SQL embebido* (ESQL). Cada instrucción de ESQL empieza con las palabras claves **EXEC SQL**. Las instrucciones ESQL se transforman en instrucciones del lenguaje del host mediante un *precompilador* (que habitualmente inserta llamadas a rutinas de librerías que ejecutan los variados comandos de SQL).

Cuando vemos los ejemplos de *Select* observamos que el resultado de las consultas es algo muy próximo a un conjunto de tuplas. La mayoría de los lenguajes de host no están diseñados para operar con conjuntos, de modo que necesitamos un mecanismo para acceder a cada tupla única del conjunto de tuplas devueltas por una instrucción SELECT. Este mecanismo puede ser proporcionado declarando un*cursor*. Tras ello, podemos utilizar el comando FETCH para recuperar una tupla y apuntar el cursor hacia la siguiente tupla.

Para una discusión más detallada sobre el SQL embebido, diríjase a [*Date and Darwen, 1997*], [*Date, 1994*], o [*Ullman, 1988*].

## Arquitectura

Antes de comenzar, debería comprender las bases de la arquitectura del sistema Postgres . Entendiendo como las partes de Postgres interactuan le hará el siguiente capítulo mucho más sencillo. En la jerga de bases de datos, Postgres usa un modelo cliente/sevidor conocido como "proceso por usuario". Una sesión Postgres consiste en los siguientes procesos cooperativos de Unix (programas):

- Un proceso demonio supervisor (`postmaster`),
- la aplicación sobre la que trabaja el usuario (frontend) (e.g., el programa`psql` ), y
- uno o más servidores de bases dedatos en segundo plano (el mismo proceso `postgres`).

Un único `postmaster` controla una colección de bases de datos dadas en un único host. Debido a esto una colección de bases de datos se suele llamar una instalación o un sitio. Las aplicaciones de frontend que quieren acceder a una determinada base de datos dentro de una instalación hacen llamadas a la libreria La libreria envía peticiones de usuario a través de la red al `postmaster` (*Como se establece una conexión*), el cual en respuesta inicia un nevo proceso en el servidor (backend) y conecta el proceso de frontend al nuevo servidor. A partir de este punto, el proceso de frontend y el servidor en backend se comunican sin la intervención del `postmaster`. Aunque, el `postmaster` siempre se está ejecutando, esperando peticiones, tanto los procesos de frontend como los de backend vienen y se van.

La libreria `libpq` permite a un único proceso en frontend realizar multiples conexiones a procesos en backend. Aunque, la aplicación frontend todavía es un proceso en un único thread. Conexiones multithread entre el frontend y el backend no están soportadas de momento en `libpq`. Una implicación de esta arquitectura es que el `postmaster` y el proceso backend siempre se ejecutan en la misma máquina (el servidor de base de datos), mientras que la aplicación en frontend puede ejecutarse desde cualquier sitio. Debe tener esto en mente, porque los archivos que pueden ser accedidos en la máquina del cliente pueden no ser accesibles (o sólo pueden ser accedidos usando un nombre de archivo diferente) el máquina del servidor de base de datos.

Tenga en cuenta que los servicios `postmaster` y postgres se ejecutan con el identificador de usuario del "superusuario" Postgres Note que el superusuario Postgres no necesita ser un usuario especial (ej. un usuario llamado "postgres"). De todas formas, el superusuario Postgres definitivamente no tiene que ser el superusuario de Unix ("root")! En cualquier caso, todos los archivos relacionados con la base de datos deben pertenecer a este superusuario Postgres. Postgres.

## Empezando

¿Cómo empezar a trabajar con Postgres?

Algunos de los pasos necesarios para usar Postgres pueden ser realizados por cualquier usuario, y algunos los deberá realizar el administrador de la base de datos. Este administrador es la persona que instaló el software, creó los directorios de las bases de datos e inició el proceso `postmaster`. Esta persona no tiene que ser el superusuario Unix ("root") o el administrador del sistema. Una persona puede instalar y usarPostgres sin tener una cuenta especial o privilegiada

Si está instalando Postgres, consulte las instrucciones de instalación en la Guía de Administración y regrese a esta guía cuando haya concluido la instalación.

Mientras lee este manual, cualquier ejemplo que vea que comience con el carácter "%" son órdenes que se escribirán en el la línea de órdenes de Unix. Los ejemplos que comienzan con el caracter "*" son órdenes en el lenguaje de consulta Postgres, Postgres SQL.

## Ejecución del Monitor Interactivo (psql)

Asumiendo que su administrador haya ejecutado adecuadamente el proceso `postmaster` y le haya autorizado a utilizar la base de datos, puede comenzar a ejecutar aplicaciones como usuario. Como mencionamos previamente, debería añadir `/usr/local/pgsql/bin` al "path" de búsqueda de su intérprete de órdenes. En la mayoría de los casos, es lo único que tendrá que hacer en términos de preparación.

Desde Postgres v6.3, se soportan dos tipos diferentes de conexión. El administrador puede haber elegido permitir conexiones por red TCP/IP, o restringir los accesos a la base de datos a través de conexiones locales (en la misma máquina). Esta elección puede ser significativa si encuentra problemas a la hora de conectar a la base de datos.

Si obtiene los siguientes mensajes de error de una orden Postgres (tal como `psql` o `createdb`):

```
% psql postgres
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections
   at 'UNIX Socket' on port '5432'?
```

o

```
% psql -h localhost postgres
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
   (with -i) connections at 'localhost' on port '5432'?
```

normalmente es debido a que (1) el `postmaster` no está en funcionamiento, o (2) está intentando conectar al servidor equivocado. Si obtiene el siguiente mensaje de error:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

Significa que el administrador ejecutó el `postmaster` mediante el usuario equivocado. Dígale que lo reinicie utilizando el superusuario de Postgres.

== Administrando una Base de datos = = Ahora que Postgres está ejecutándose podemos crear alguna base de datos para experimentar con ella. Aquí describimos las órdenes básicas para administrar una base de datos

La mayoría de las aplicaciones Postgres asumen que el nombre de la base de datos, si no se especifica, es el mismo que el de su cuenta en el sistema.

Si el administrador de bases de datos ha configurado su cuenta sin privilegios de creación de bases de datos, entonces deberán decirle el nombre de sus bases datos. Si este es el caso, entonces puede omitir la lectura de esta sección sobre creación y destrucción de bases de datos.

### Creación de una base de datos

Digamos que quiere crear una base de datos llamada mydb. Puede hacerlo con la siguiente orden:

```
% createdb mydb
```

Si no cuenta con los privilegios requeridos para crear bases de datos, verá lo siguiente:

```
% createdb mydb
NOTICE: user "su nombre de usuario" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres le permite crear cualquier número de bases de datos en un sistema dado y automáticamente será el administrador de la base de datos que creó. Los nombres de las bases de datos deben comenzar por un carácter alfabético y están limitados a una longitud de 32 caracteres. No todos los usuarios están autorizados para ser administrador de una base de datos. Si Postgres le niega la creación de bases de datos, seguramente es debido a que el administrador del sistema ha de otorgarle permisos para hacerlo. En ese caso, consulte al administrador del sistema.

### Acceder a una base de datos

Una vez que ha construido una base de datos, puede acceder a ella:

- Ejecutando los programas de monitorización de Postgres (por ejemplo `psql`) los cuales le permiten introducir, editar y ejecutar órdenes SQL interactivamente
- Escribiendo un programa en C usando la librería de subrutinas LIBPQ, la cual le permite enviar órdenes SQL desde C y obtener mensajes de respuesta en su programa. Esta interfaz es discutida más adelante en la *Guía de Programadores de PostgreSQL*

Puede que desee ejecutar `psql`, para probar los ejemplos en este manual. Lo puede activar para la base de datos mydb escribiendo la orden:

```
% psql mydb
```

Se le dará la bienvenida con el siguiente mensaje:

```
Welcome to the POSTGRESQL interactive sql monitor:
 Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
You are currently connected to the database: template1

mydb=>
```

Este prompt indica que el monitor está listo y puede escribir sus consultas SQL dentro de un espacio de trabajo mantenido por el monitor. El programa `psql` responde a los códigos de escape que empiezan por el carácter "\". Por ejemplo, puede obtener la ayuda acerca de la sintaxis de varias órdenes SQL Postgres escribiendo:

```
mydb=> \h
```

Una vez que haya terminado de introducir consultas, puede pasar el contenido del espacio de trabajo al servidor Postgres escribiendo:

```
mydb=> \g
```

Esto le dice al servidor que procese la consulta. Si termina su consulta con un punto y coma, la "\g" no es necesaria. `psql` procesará automáticamente las consultas terminadas con punto y coma. Para leer consultas desde un archivo, digamos myFile, en lugar de introducirlas interactivamente, escriba:

```
mydb=> \i nombreDelFichero
```

Para salir de `psql` y regresar a Unix escriba:

```
mydb=> \q
```

y `psql` terminará y volverá a la línea de órdenes. (Para conocer más códigos de escape, escriba **\h** en el prompt del monitor). Se pueden utilizar espacios en blanco (por ejemplo espacios, tabulador y el carácter de nueva línea) en las consultas SQL. Las líneas simples comentadas comienzan por "--". Lo que haya después de los guiones hasta el final de línea será ignorado. Los comentarios múltiples y los que ocupan más de una línea se señalan con "/* ... */"

#### Eliminando bases de datos

Si es el administrador de la base de datos mydb, puede eliminarla utilizando la siguiente orden Unix:

```
% dropdb mydb
```

Esta acción elimina físicamente todos los archivos Unix asociados a la base de datos y no pueden recuperarse, así que deberá hacerse con precaución.

### El Lenguaje de consultas

El lenguaje de consultas de Postgres Postgres es una variante del estándar SQL3 Tiene muchas extensiones, tales como tipos de sistema extensibles, herencia, reglas de producción y funciones. Estas son características tomadas del lenguaje de consultas original de Postgres (PostQuel). Ésta sección proporciona un primer vistazo de cómo usar Postgres SQL para realizar operaciones sencillas. La intención de este manual es simplemente la de proporcionarle una idea de nuestra versión de SQL y no es de ningún modo un completo tutorial acerca de SQL. Se han escrito numerosos libros sobre SQL, incluyendo [MELT93] and [DATE97]. Tenga en cuenta que algunas características del lenguaje son extensiones del estándar ANSI.