



Introducción a Big Data

Fundación Telefónica Movistar
CURSO DE INTRODUCCIÓN A BIG DATA.

Modelado de datos

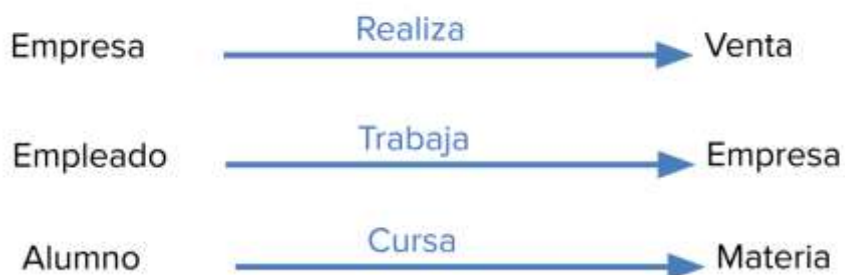
En la mayoría de los paradigmas se separan las abstracciones de datos que representan información y los procesos: abstracciones de control del flujo que manipulan esos datos

En el diseño lógico de datos definimos

- las **entidades** que representan partes de un sistema
- sus **relaciones**.

Qué son las Relaciones?

- Representan asociaciones del mundo real entre entidades.
- Se puede representar con un verbo o preposición que conecta dos entidades.



Ahora bien, cómo podemos comunicar nuestro Modelo de Datos?

- Prosa, poca y muy puntual.
- Diagrama de Entidad Relación DER.
- Código SQL – DDL Data Definition Language. (Metadata)₍₁₎



Modelo Conceptual

El modelo conceptual define las entidades y sus relaciones en base al negocio (es fácil de validar por el usuario). Es independiente de la tecnología.

Ejemplos

- un cliente tiene muchas sucursales,
- cada sucursal está ubicada en un domicilio.
- una materia de una facultad puede tener 0, 1 ó muchas correlativas.

Diseño lógico

Aquí definimos el esquema en el cual vamos a trabajar sin depender de una implementación física determinada. Ej: trabajamos en un esquema relacional sin decidir si lo implementaremos en Oracle, MySQL o SQLServer.

El modelo conceptual define las entidades y sus relaciones en base al negocio (es fácil de validar por el usuario). Es independiente de la tecnología.

Diseño físico

El diseño físico se implementa en un motor de base de datos que, además de soportar un determinado esquema, impone ciertas restricciones de la implementación, como el lenguaje de manipulación de datos (DML - data manipulation language) o el que nos permite crear las entidades (DDL - Data definition language).

El estándar para el modelo relacional es el SQL aunque cada DBMS implementa su propio subconjunto de instrucciones.

Modelado de datos

DER – Diagrama de Entidad Relación

- **Entidad:** cualquier cosa que tenga relevancia para nuestro sistema. Los proyectos, las tareas, los impuestos, las complejidades...
- **Atributos:** son las propiedades o características que describen una entidad. Un cliente tiene nombre y cuit, un auto tiene modelo, marca y color, etc.
- **Atributos multivaluados:** son atributos que pueden tomar más de un valor (direcciones de mail de un empleado, subordinados de un jefe, entre otros)



DER – Diagrama de Entidad Relación

- **Instancia de una entidad:** Ocurrencia particular de una entidad.
- **Relaciones entre entidades**
 - Representan asociaciones del mundo real entre entidades
 - Se puede representar con un verbo o preposición que conecta dos entidades.

Características de las Relaciones

- **Grado**
 - Unarias o Recursivas: Son relaciones que asocian a una misma entidad.
 - Binarias: Relaciones que asocian a dos entidades.
 - N-arias: Relaciones que asocian a N entidades
- **Cardinalidad**
 - Describe la cantidad de instancias de entidad permitidas en un relación entre dos entidades
 - 1 a 1: un cliente tiene un domicilio y cada domicilio pertenece a un cliente.
 - 1 a n: una empresa tiene muchas sucursales, cada sucursal pertenece a una empresa.
 - n a n: un profesor da clases a muchos alumnos, cada alumno tiene muchos profesores
- **Modalidad**
 - Relación Mandatoria (Obligatoria): Si la instancia de una entidad debe existir en la relación. Ej.: Una Factura debe contener a lo sumo un Ítem.
 - Relación Opcional: Si la instancia de una entidad no necesita existir en la relación. Ej.: Un cliente puede tener facturas asociadas.
- **Especialización o Generalización**
 - Entidad Supertipo:
 - Entidad padre.
 - Entidad Subtipo:
 - Son las entidades hijas.
 - Son mutuamente excluyentes
 - Contienen distintos atributos



Algunas Notaciones

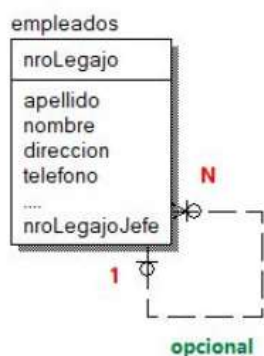
Notación	Uno a Uno	Uno a Muchos	Muchos a Muchos
Ross			
Bachman			
Martin (*)			
Chen			
IDEFX1			

Notación	Mandatoria	Opcional
Ross		
Bachman		
Martin (*)		
Chen		N/A
IDEFX1		

Algunos ejemplos.

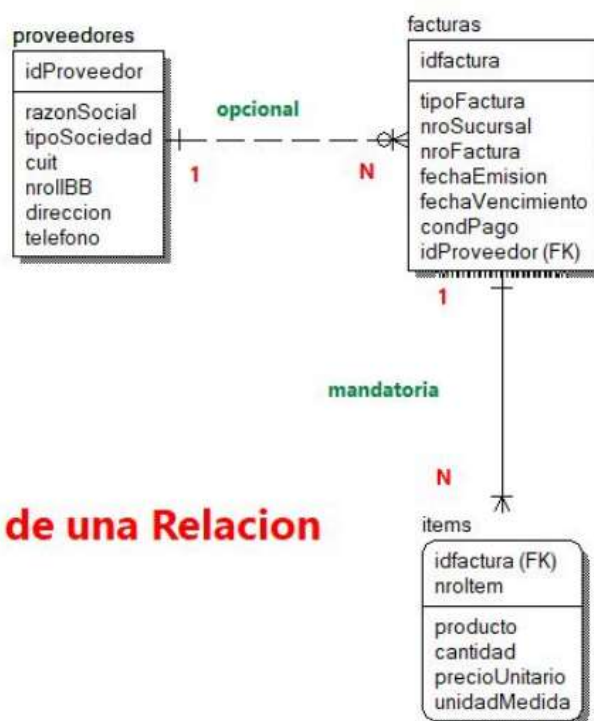
Grado de una Relacion

Unaria



Modalidad de una Relacion

Binaria



Cardinalidad de una Relacion

Grado de una Relación

N-Aria

Modalidad de una Relación



Ejemplo Relación N-Aria

Cardinalidad de una Relación

Modelo Lógico

Modalidad N a N

Relación N a N

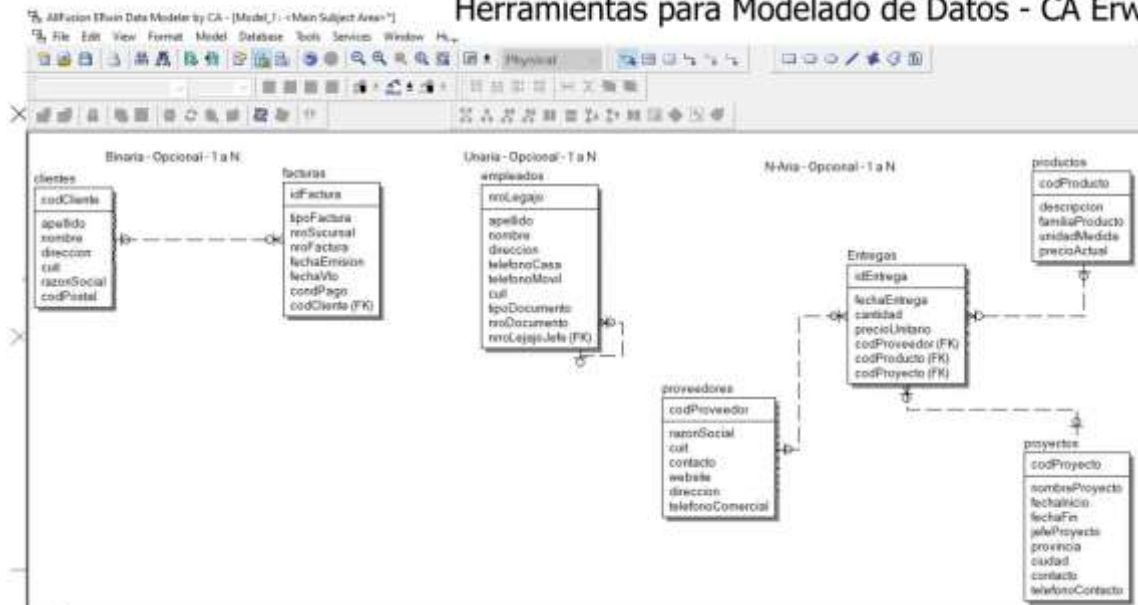
Modelo Físico

Modalidad N a N



Cómo se Implementa?

Herramientas para Modelado de Datos - CA Erwin



Normalización / Desnormalización

Normalizar una base de datos significa transformar un conjunto de datos que tienen una cierta complejidad en su entendimiento y que su distribución en el modelo provoca problemas de lógica en las acciones de manipulación de datos, en una estructura de datos que posea un diseño claro, donde estos datos guarden coherencia y no pierdan su estado de asociación.

Objetivos de la normalización

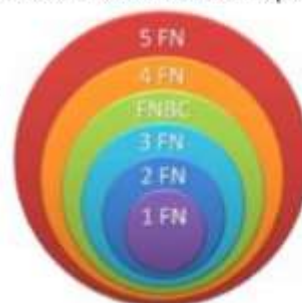
- Reducir la redundancia de datos, y por lo tanto, las inconsistencias.
- Facilitar el mantenimiento de los datos.
- Evitar anomalías en la manipulación de datos.
- Reducir el impacto de los cambios en los datos.

Normalización / Desnormalización

Las formas normales son heurísticas o criterios que permiten resolver esas redundancias. En algunas bibliografías se habla de errores o inconsistencias, es verdad que las redundancias pueden surgir por errores en el diseño de los datos, pero veremos a continuación que esa redundancia puede ser buscada.

Cada forma normal introduce restricciones nuevas, donde la primera restricción que aplica es cumplir la forma normal anterior.

Nosotros para implementar nos vamos a quedar con sólo las primeras 3 Formas Normales.



Primera Forma Normal

Normalización / Desnormalización

Una entidad que está en primera forma normal no puede tener campos repetitivos (arrays, mismo campo repetido n veces), o campos multivaluados.

Pedidos

idPedido
idCliente
fechaPedido
...
idGusto[1..n]
Cantidad[1..n]
DescripcionGusto[1..n]



Pedidos

idPedido
idCliente
fechaPedido
...

Gustos_Pedidos

idPedido (FK)
idGusto
Cantidad
descripcionGusto

N

Segunda Forma Normal

Una estructura de datos está en 2FN si y sólo si no hay dependencias funcionales parciales entre un atributo no clave y parte de la clave compuesta, y se satisface 1FN.

Gustos_Pedidos

idPedido (FK)
idGusto
Cantidad
descripcionGusto



Gustos_Pedidos

idPedido (FK)
idGusto (FK)
Cantidad

N

Gustos

idGusto
descripcionGusto

1

El atributo DescripcionGusto depende funcionalmente solo de idGusto, no tiene dependencia con el idPedido. Por lo que tiene una dependencia funcional parcial de la Clave Primaria.



Tercera Forma Normal

Una estructura de datos está en 3FN si y sólo si no hay dependencias funcionales entre los atributos no claves (y se satisface 2FN).



SQL SQL - Structured Query Language

Edgar Codd en los años 70 creó el modelo relacional y el álgebra relacional.

IBM, se creó el nuevo software de base de datos System R basado en el modelo teórico de E. Codd.

Allá por 1974 para gestionar los datos almacenados en System R, se creó el lenguaje SQL.

En un principio se llamó SEQUEL, un nombre que todavía se utiliza como una pronunciación alternativa para SQL, pero más tarde fue renombrado a sólo SQL.



Sublenguajes del SQL

- DDL - Data Definition Language
- DML - Data Manipulation Language
- TCL - Transaction Control language
- DCL - Data Control Language

- **DDL - Data Definition Language**

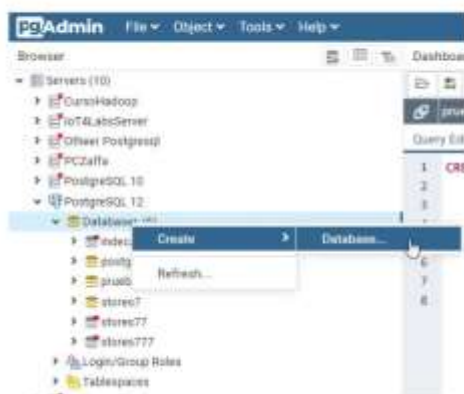
Es el sublenguaje que se encarga de la modificación de la estructura de los objetos de la base de datos, o sea de definir su metadata. Posee instrucciones para modificar, borrar o definir tablas, vistas, base de datos, entre otros. Estos comandos son:

- **CREATE.**
- **ALTER.**
- **DROP.**
- **TRUNCATE.**

Que es una Base de Datos?

Una BD es un conjunto de **datos persistentes e interrelacionados** que es **utilizado por los sistemas de aplicación** de una empresa, los mismos se encuentran **almacenados en un conjunto independiente y sin redundancias** o con redundancias mínimas.

CREATE DATABASE comercial;



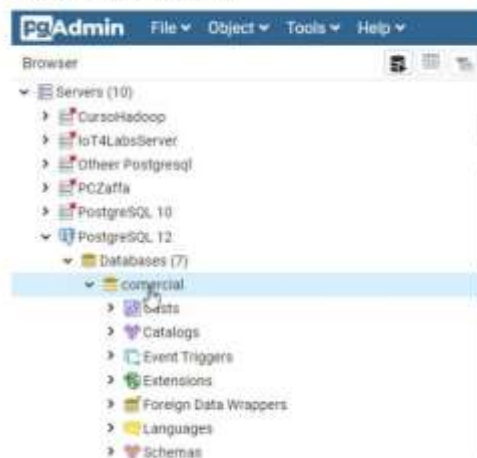
Este comando permite crear una base de datos con todas sus estructuras y componentes internas.

En general y dependiendo del motor de base de datos tiene un conjunto de parámetros asociados a estructuras físicas y espacios de discos de aloados.

Dentro del PGAdmin, boton derecho sobre databases y luego Create.



USE comercial;

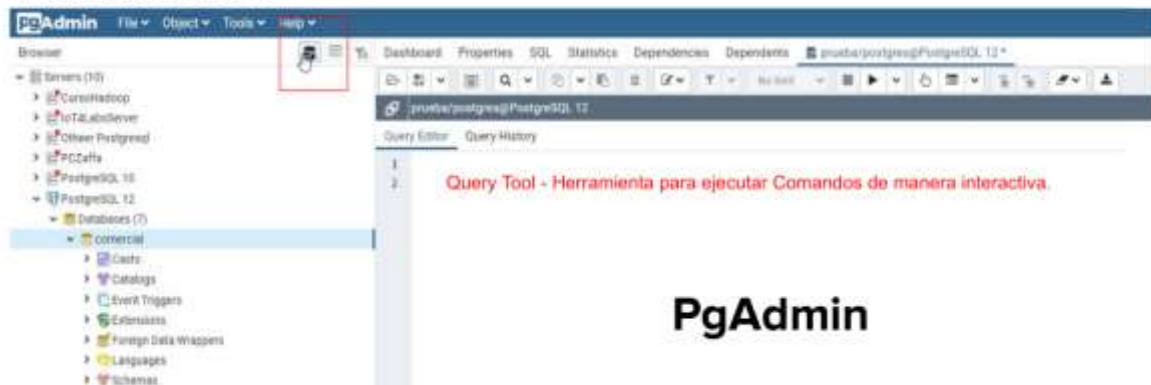


Este comando permite acceder a la Base de Datos, conectándonos a la misma.

Dentro del PgAdmin, doble click sobre la base de datos a acceder.

GUI - Graphic User Interface

Desarrollada por la gente de PostgreSQL para acceder a las Bases de datos de una manera más amigable al usuario.



PgAdmin



Tablas

Es la unidad básica de almacenamiento de datos. Los datos están almacenados en filas y columnas. Son de existencia permanente y poseen un nombre identificador único por esquema o por base de datos (dependiendo del motor de base de datos).

Cada columna tiene entre otros datos un nombre, un tipo de datos y un ancho (este puede estar predeterminado por el tipo de dato).

```
CREATE TABLE ordenes (  
    N_orden INT NULL ,  
    N_cliente INT NULL ,  
    F_orden DATE NULL ,  
    C_estado SMALLINT NULL ,  
    F_alta_audit TIMESTAMP NULL ,  
    D_usuario VARCHAR(20) NULL )
```

Constraints

Integridad de Entidad

La integridad de entidades es usada para asegurar que los datos pertenecientes a una misma tabla tienen una única manera de identificarse, es decir que cada fila de cada tabla tenga una primary key capaz de identificar unívocamente una fila y esa no puede ser nula

PRIMARY KEY CONSTRAINT: Puede estar compuesta por una o más columnas, y deberá representar unívocamente a cada fila de la tabla. No debe permitir valores nulos.



Integridad Referencial

La integridad referencial es usada para asegurar la coherencia entre datos de dos tablas.

FOREIGN KEY CONSTRAINT: Puede estar compuesta por una o más columnas, y estará referenciando a la PRIMARY KEY de otra tabla.

Los constraints referenciales permiten a los usuarios especificar claves primarias y foráneas para asegurar una relación PADRE-HIJO (MAESTRO-DETALLE).

Tipos de Constraints Referenciales

Ciclic Referential Constraint.

Asegura una relación de PADRE-HIJO entre tablas. Es el más común.

Ej. CLIENTE □ FACTURAS

Self Referencing Constraint.

Asegura una relación de PADRE-HIJO entre la misma tabla.

Ej. EMPLEADOS □ EMPLEADOS

Multiple Path Constraint.

Se refiere a una PRIMARY KEY que tiene múltiples FOREIGN KEYS. Este caso también es muy común.

Ej. CLIENTES □ FACTURAS
CLIENTES □ RECLAMOS

Integridad Semántica

La integridad semántica es la que nos asegura que los datos que vamos a almacenar tengan una apropiada configuración y que respeten las restricciones definidas sobre los dominios o sobre los atributos.

- DATA TYPE
- DEFAULT
- UNIQUE
- NOT NULL
- CHECK



DATA TYPE: Este define el tipo de valor que se puede almacenar en una columna.

DEFAULT CONSTRAINT: Es el valor insertado en una columna cuando al insertar un registro ningún valor fue especificado para dicha columna. El valor default por default es el NULL.

Se aplica a columnas no listadas en una sentencia INSERT.

El valor por default puede ser un valor literal o una función SQL (USER, TODAY, etc.)

Aplicado sólo durante un INSERT (NO UPDATE).

UNIQUE CONSTRAINT: Especifica sobre una o más columnas que la inserción o actualización de una fila contiene un valor único en esa columna o conjunto de columnas.

NOT NULL CONSTRAINT: Asegura que una columna contenga un valor durante una operación de INSERT o UPDATE. Se considera el NULL como la ausencia de valor.

CHECK CONSTRAINT: Especifica condiciones para la inserción o modificación en una columna. Cada fila insertada en una tabla debe cumplir con dichas condiciones. Actúa tanto en el INSERT, como en el UPDATE.

Es una expresión que devuelve un valor booleano de TRUE o FALSE.

Son aplicados para cada fila que es INSERTADA o MODIFICADA.

Todas las columnas a las que referencia deben ser de la misma tabla (la corriente).

No puede contener subconsultas, secuencias, funciones (de fecha, usuario) ni pseudocolumnas.

Todas las filas existentes en una tabla deben pasar un nuevo constraint creado para dicha tabla. En el caso de que alguna de las filas no cumpla, no se podrá crear dicho constraint o se creará en estado deshabilitado.

Tipos de Constraints

- Existen dos métodos para definir constraints.

Restricciones a nivel de Columna

Ej.

```
CREATE TABLE ordenes (
  N_orden      INT PRIMARY KEY,
  N_cliente    INT,
  F_orden      DATE,
  C_estado     SMALLINT,
  F_alta_audit DATE,
  D_usuario    VARCHAR(20) );
```

Restricciones a nivel de Tabla

Ej.

```
CREATE TABLE items_ordenes (
  N_orden      INT,
  N_item        SMALLINT,
  C_producto    INT,
  Q_cantidad    INT,
  I_precunit    NUMERIC(9,3),
  PRIMARY KEY (n_orden, n_item) );
```

Cuando la restricción es sobre un grupo de columnas se debe utilizar una restricción a nivel de tabla, cuando es sobre sólo una columna puede utilizarse cualquiera de los dos modos.



Ejemplos de PRIMARY KEY

Restricciones a nivel de Columna

Ej.

```
CREATE TABLE ordenes (
    N_orden      INT PRIMARY KEY,
    N_cliente    INT,
    F_orden      DATE,
    C_estado     SMALLINT,
    F_alta_audit DATE,
    D_usuario    VARCHAR(20) );
```

Restricciones a nivel de Tabla

Ej.

```
CREATE TABLE items_ordenes (
    N_orden      INT,
    N_item        SMALLINT,
    C_producto    INT,
    Q_cantidad    INT,
    I_precunit    NUMERIC(9,3),
    PRIMARY KEY (n_orden, n_item) );
```

Ejemplos de FOREIGN KEY

Restricciones a nivel de Columna

```
CREATE TABLE ordenes (
    N_orden INTEGER PRIMARY KEY,
    N_cliente INTEGER,
    F_orden DATE,
    C_estado SMALLINT,
    F_alta_audit DATE,
    D_usuario VARCHAR(20)
```

```
CREATE TABLE items_ordenes (
    N_orden INT REFERENCES ordenes,
    N_item    SMALLINT,
    C_producto INT,
    Q_cantidad INT,
    I_precunit NUMERIC(9,3),
    PRIMARY KEY (n_orden, n_item));
```

Restricciones a nivel de Tabla

```
CREATE TABLE items_ordenes (
    N_orden INT REFERENCES ordenes,
    N_item    SMALLINT,
    C_producto INTEGER,
    Q_cantidad INTEGER,
    I_precunit DECIMAL ( 9,3),
    PRIMARY KEY (n_orden, n_item) );
```

```
CREATE TABLE mov_stock (
    N_stock      INT,
    C_movimiento SMALLINT,
    N_orden      INT,
    N_item        SMALLINT,
    C_producto    INT,
    Q_cantidad    INT,
    FOREIGN KEY (n_orden, n_item)
    REFERENCES items_ordenes);
```



Ejemplos de SELF REFERENCING CONSTRAINT

Restricciones a nivel de Columna

```
CREATE TABLE empleados (
  N_empleado INTEGER PRIMARY KEY,
  D_Apellido VARCHAR(60),
  D_nombres VARCHAR(60),
  N_cuil NUMERIC(11,0),
  N_jefe INTEGER,
  FOREIGN KEY (n_jefe) REFERENCES empleados (n_empleado) );
```

El motor no permitirá ingresar un empleado cuyo nro. de jefe no exista como número de empleado.

Lo que si permitirá es ingresar un nro. de jefe NULO.

Ejemplos de DEFAULT

```
CREATE TABLE ordenes
(
  N_orden INT NOT NULL,
  N_cliente INT,
  F_orden DATE,
  C_estado SMALLINT DEFAULT 1,
  F_alta_audit DATE DEFAULT CURRENT_DATE,
  D_usuario VARCHAR(20) DEFAULT USER
);
```

Al de ejecutar el siguiente comando

```
INSERT INTO ordenes (n_orden, n_cliente, f_orden) VALUES (117, 10, '2020-10-05');
```

El motor de base de datos insertará el siguiente registro en la tabla **ordenes**

n_orden	n_cliente	f_orden	c_estado	f_alta_audit	d_usuario
integer	integer	date	smallint	date	character varying (20)
117	10	2020-10-05	1	2020-10-04	postgres

Ejemplos de NOT NULL

```
CREATE TABLE ordenes
(
  N_orden INT NOT NULL,
  N_cliente INT,
  F_orden DATE,
  C_estado SMALLINT NOT NULL,
  F_alta_audit DATE,
  D_usuario VARCHAR(20)
);
```



Ejemplos de UNIQUE

Restricciones a nivel de Columna

```
CREATE TABLE empleados (
    N_empleado INT PRIMARY KEY,
    D_Apellido VARCHAR(60),
    D_nombres VARCHAR(60),
    N_cuil BIGINT UNIQUE,
    F_nacimiento DATE,
    F_ingreso DATE,
    N_jefe INT);
```

Restricciones a nivel de Tabla

```
CREATE TABLE empleados (
    N_empleado INT PRIMARY KEY,
    D_Apellido VARCHAR(60),
    D_nombres VARCHAR(60),
    T_docum SMALLINT,
    N_docum NUMERIC(11,0),
    F_nacimiento DATE,
    F_ingreso DATE,
    N_jefe INT,
    UNIQUE (t_docum, n_docum));
```

La tabla empleados tiene como clave primaria al atributo **n_empleado**.
Con una restricción de UNIQUE podemos representar claves alternas.

En el primer ejemplo, el **n_cuil** es un atributo que posee valores únicos para cada fila de la tabla **empleados**.

En el segundo ejemplo, la clave compuesta por los atributos **t_docum** y **n_docum** (tipo y número de documento) posee valores únicos para cada fila de la tabla **empleados**.

Ejemplos de CHECK

Restricciones a nivel de Columna

```
CREATE TABLE ordenes
(
    N_orden NUMBER NOT NULL,
    N_cliente NUMBER,
    F_orden DATE,
    C_estado NUMBER CHECK (c_estado IN (1,2,3)),
    F_alta_audit DATE,
    D_usuario VARCHAR2(20)
);
```

Restricciones a nivel de Tabla

```
CREATE TABLE empleados
(
    N_empleado NUMBER,
    D_Apellido VARCHAR2(60),
    D_nombres VARCHAR2(60),
    N_cuil NUMBER(11) UNIQUE,
    F_nacimiento DATE,
    F_ingreso DATE,
    N_jefe NUMBER,
    CHECK (F_nacimiento < F_ingreso)
);
```



Ejemplos de ALTER TABLE

Este comando permite modificar la estructura y metadata de una tabla dada.

Renombrar una tabla

```
ALTER TABLE ordenes RENAME TO ordenesCompra;
```

Renombrar una columna

```
ALTER TABLE ordenes RENAME COLUMN n_orden TO nroOrden;
```

Agregar una nueva columna

```
ALTER TABLE ordenes ADD COLUMN fechaEmbarque DATE;
```

Eliminar una columna existente

```
ALTER TABLE ordenes DROP COLUMN address RESTRICT;  
(Restrict no permite eliminarla si tiene dependencias)
```

Agregar un Constraint a la tabla

```
ALTER TABLE ordenes ADD CONSTRAINT Fembchk CHECK (fechaEmbarque >= fechaEmision);
```

Eliminar un Constraint a la tabla

```
ALTER TABLE ONLY ordenes DROP CONSTRAINT Fembchk;
```

Ejemplos de DROP

DROP TABLE

```
DROP TABLE ordenes;
```

Elimina la tabla ordenes y su contenido.
El motor de base de datos chequeará que no existan dependencias de otras tablas hacia la tabla a borrar.

DROP DATABASE

```
DROP DATABASE comercial;
```

Elimina la base de datos con todos sus objetos creados y su contenido.
El motor de base de datos chequeará que no existan conexiones vigentes en la Base a borrar.

TRUNCATE TABLE

```
TRUNCATE TABLE ordenes;
```

Elimina el contenido de la tabla ordenes, sin eliminar su Definición.
El motor de base de datos chequeará que no existan dependencias de otras tablas hacia los datos de la tabla a borrar.

Es muy utilizado cuando uno quiere eliminar todos los datos de una tabla, ya que es mucho más eficiente que el comando del DML - `DELETE FROM ordenes`.



Lenguaje SQL.



Sub Lenguajes

- DDL – Data Definition Language
- DML – Data Manipulation Language
 - SELECT
 - INSERT
 - UPDATE
 - DELETE

SQL – Operador SELECT

SELECT * | lista de columnas

FROM nom_tabla | lista de tablas

WHERE condiciones ó filtros

GROUP BY columnas clave de agrupamiento

HAVING condiciones sobre lo agrupado

ORDER BY columnas clave de ordenamiento

LIMIT limita la cantidad de filas a mostrar



Ejemplos de uso.

```
SELECT *  
FROM customer
```

Consulta todas las columnas de la tabla customer.

```
SELECT lname, fname, state, city  
FROM customer
```

Consulta de distintas columnas pertenecientes a la tabla customer.

```
SELECT stock_num,manu_code,unit_price, unit_price*1.15  
FROM products
```

Consulta de distintas columnas con expresiones aritméticas.

```
SELECT stock_num,manu_code,unit_price, unit_price*1.15 NewPrice  
FROM products
```

Alias de Columnas o Etiquetas

```
SELECT stock_num codTipoProducto,manu_code codFabricante,  
unit_Price precioUnitario  
FROM products
```

Alias de Columnas o Etiquetas

```
SELECT lname || ', ' || fname apellidoYNombre, company, city  
FROM customer
```

Concatenar columnas en una sola nueva columna de salida.

Además en el ejemplo le ponemos el alias apellidoYNombre



```
SELECT order_num, order_date, DATE_PART('year',order_date) anio,
      DATE_PART('month',order_date) mes,
      DATE_PART('day',order_date) dia,
      CURRENT_USER
```

Utilización de
funciones especiales.

```
FROM orders
```

```
stores7/postgres@PostgreSQL 12
Query Editor  Query History
1 SELECT order_num, order_date, DATE_PART('year',order_date) anio,
2       DATE_PART('month',order_date) mes, DATE_PART('day',order_date) dia,
3       current_user
4 FROM orders
5
```

ARITMÉTICAS
TRIGONOMÉTRICAS
FINANCIERAS
DE FECHA
DE STRINGS
Entre otras..

Criterios de selección.

```
SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE condiciones
```

Condiciones

AND, OR, NOT

=	igualdad
<> !=	distinto
>, >=	mayor, mayor igual
<, <=	menor, menor igual
[NOT] LIKE	validar substrings
[NOT] BETWEEN	entre rango
[NOT] IN	en lista de valores
IS [NOT] NULL	es o no es nulo

```
SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE customer_num=104
```

```
stores7/postgres@PostgreSQL 12
Query Editor  Query History
1 SELECT order_num, order_date, customer_num, paid_date
2 FROM orders
3 WHERE customer_num=104
```

Condiciones por
igualdad.




```
SELECT order_num, order_date, customer_num, paid_date  
FROM orders  
WHERE customer_num=104  
AND order_num >1010
```

Varias Condiciones
con AND.



The screenshot shows a PostgreSQL Query Editor window titled 'stores7/postgres@PostgreSQL 12'. It has two tabs: 'Query Editor' and 'Query History'. The 'Query Editor' tab is active and contains the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date  
2 FROM orders  
3 WHERE customer_num=104  
4 AND order_num >1010  
5
```

```
SELECT order_num, order_date, customer_num, paid_date, ship_date  
FROM orders  
WHERE ship_date IS NULL
```

Condición por
columnas con Nulos.



The screenshot shows a PostgreSQL Query Editor window titled 'stores7/postgres@PostgreSQL 12'. It has two tabs: 'Query Editor' and 'Query History'. The 'Query Editor' tab is active and contains the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date, ship_date  
2 FROM orders  
3 WHERE ship_date IS NULL  
4
```

```
SELECT order_num, order_date, customer_num, paid_date  
FROM orders  
WHERE order_num BETWEEN 1004 AND 1020
```

Condición por un
rango de valores para
una columna.

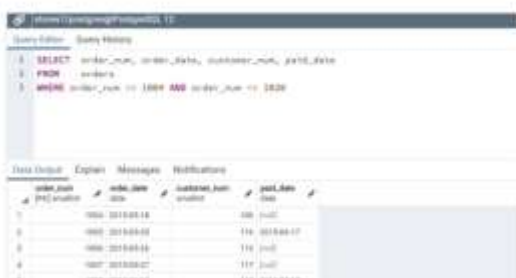


The screenshot shows a PostgreSQL Query Editor window titled 'stores7/postgres@PostgreSQL 12'. It has two tabs: 'Query Editor' and 'Query History'. The 'Query Editor' tab is active and contains the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date  
2 FROM orders  
3 WHERE order_num BETWEEN 1004 AND 1020  
4
```



SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE order_num >= 1004 AND order_num <= 1020



The screenshot shows a PostgreSQL query editor with the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date
2 FROM orders
3 WHERE order_num >= 1004 AND order_num <= 1020
```

The results table has the following columns: order_num, order_date, customer_num, and paid_date. The data is as follows:

order_num	order_date	customer_num	paid_date
1004	2019-05-18	100	paid
1005	2019-05-18	110	2019-05-17
1006	2019-05-18	110	paid
1007	2019-05-17	111	paid
1008	2019-05-18	110	2019-05-17

Condición por un rango de valores para una columna.
Devuelve lo mismo que el Between.

SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE customer_num IN (104,110,127)



The screenshot shows a PostgreSQL query editor with the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date
2 FROM orders
3 WHERE customer_num IN (104,110,127)
4
```

Condición de una columna por una lista de valores

SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE customer_num =104 OR customer_num=110
OR customer_num=127



The screenshot shows a PostgreSQL query editor with the following SQL query:

```
1 SELECT order_num, order_date, customer_num, paid_date
2 FROM orders
3 WHERE customer_num =104 OR customer_num=110
4 OR customer_num=127
5
```

Condición de una columna por una lista de valores. Devuelve igual resultado que el IN.



```
SELECT order_num, order_date, customer_num, paid_date
FROM orders
WHERE (order_num >= 1010 AND customer_num = 104)
      OR (customer_num = 101)
```

Condiciones con AND
y OR.



```
stores7/postgres@PostgreSQL 12
Query Editor  Query History
1 SELECT order_num, order_date, customer_num, paid_date
2 FROM orders
3 WHERE (order_num >= 1010 AND customer_num = 104)
4      OR (customer_num = 101)
```

```
SELECT customer_num, lname, fname, company, city
FROM customer
WHERE condiciones
```

Condiciones con operador LIKE

lname LIKE 'A%' apellidos que comiencen con 'A'.

lname LIKE '%th%' apellidos que contenga 'th' en cualquier parte.

lname LIKE 'A_ _ _' apellidos que comiencen con 'A' y tengan 4 letras

```
SELECT customer_num, lname, fname, company, city
FROM customer
WHERE lname LIKE 'A%'
```

% reemplaza a 0 o más
caracteres.



```
stores7/postgres@PostgreSQL 12
Query Editor  Query History
1 SELECT customer_num, lname, fname, company, city
2 FROM customer
3 WHERE lname LIKE 'A%'
4
```



SELECT customer_num, lname, fname, company, city
FROM customer
WHERE lname LIKE '%i%'

% reemplaza a 0 o más caracteres.



The screenshot shows a PostgreSQL Query Editor window with the following query:

```
1 SELECT customer_num, lname, fname, company, city
2 FROM customer
3 WHERE lname LIKE 'i%'
4
5
```

The results are displayed in a table with the following columns: customer_num, lname, fname, company, and city. The first row of results is:

customer_num	lname	fname	company	city
101	Park	Lodging	All Sports Supplies	Stuyvesant

En este caso muestra todos los clientes cuyo apellido contenga una i, en cualquier lado.

SELECT customer_num, lname, fname, company, city
FROM customer
WHERE lname LIKE 'P_____'

_ Reemplaza a 1 sólo carácter.



The screenshot shows a PostgreSQL Query Editor window with the following query:

```
1 SELECT customer_num, lname, fname, company, city
2 FROM customer
3 WHERE lname LIKE 'P_____'
```

En este caso muestra todos los clientes cuyo apellido comience con P y tenga 5 letras.

SELECT customer_num, lname, fname, company, city
FROM customer
WHERE lname LIKE 'P%'

Diferencia entre _ y %



The screenshot shows a PostgreSQL Query Editor window with the following query:

```
1 SELECT customer_num, lname, fname, company, city
2 FROM customer
3 WHERE lname LIKE 'P%'
```

En este caso muestra todos los clientes cuyo apellido comience con P, sin importar la cantidad de letras.



SELECT customer_num, fname, lname, city
FROM customer
ORDER BY city, customer_num

Ordenamiento del resultado de la consulta por una clave o múltiples claves.



customer_num	fname	lname	city
124	Olivia	Puryear	Barksville
127	Kim	Suther	Blue Island
122	James	Henry	Brington
119	Bob	Shaner	Cherry Hill
126	Steven	Nease	Danvers

Observamos que las filas están ordenadas por ciudad ascendente y a igual ciudad ordena por customer_num también ascendente.

SELECT customer_num, fname, lname, city
FROM customer
ORDER BY city, customer_num **DESC**

Ordenamiento del resultado de la consulta por una clave o múltiples claves.



customer_num	fname	lname	city
124	Olivia	Puryear	Barksville
127	Kim	Suther	Blue Island
122	James	Henry	Brington
119	Bob	Shaner	Cherry Hill
126	Steven	Nease	Danvers

Observamos que las filas están ordenadas por ciudad ASCENDENTE (default) y a igual ciudad ordena por customer_num DESCENDENTE.

SELECT customer_num, fname, lname, city
FROM customer
ORDER BY 4, 1 **DESC**

Ordenamiento del resultado de la consulta por una clave o múltiples claves.



customer_num	fname	lname	city
124	Olivia	Puryear	Barksville
127	Kim	Suther	Blue Island
122	James	Henry	Brington
119	Bob	Shaner	Cherry Hill
126	Steven	Nease	Danvers

Se puede observar en este ejemplo que en el **ORDER BY** se pueden poner números que indican la posición de la columna en la consulta, en lugar del nombre.

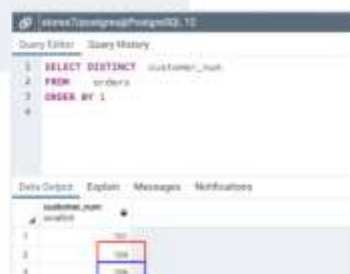
SELECT customer_num
FROM orders
ORDER BY 1



Listar valores únicos para una columna, ante una repetición de valores de esa columna.

vs.

SELECT DISTINCT customer_num
FROM orders
ORDER BY 1

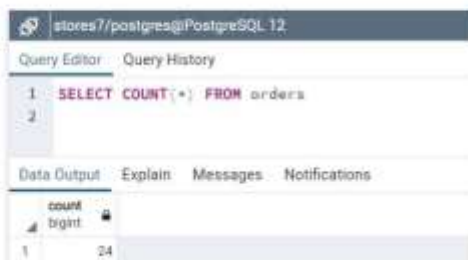


Funciones Agregadas

SUM (columna)	Suma los valores de una columna.
COUNT (*)	Cuenta todas las filas de la tabla.
COUNT (columna)	Cuenta filas con dicha columna No Nula.
COUNT (DISTINCT columna)	Cuenta solo una vez cada valor.
MIN (columna)	Devuelve el valor mínimo de una columna.
MAX (columna)	Devuelve el valor máximo de una columna.
AVG (columna)	Promedia la suma de los valores de una columna con la cantidad de filas.

Son funciones que dado un conjunto de datos realizan operaciones agregadas devolviendo un único valor como resultado.

Función COUNT (*) – Mostrar cantidad de Órdenes de Compra.



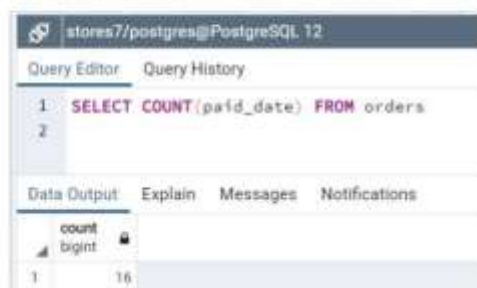
The screenshot shows a PostgreSQL query editor with the following query:

```
1 SELECT COUNT(*) FROM orders
```

The results are displayed in a table with the following structure:

count	bigint
1	24

Función COUNT (paid_date) – Mostrar cantidad de Órdenes de Compra con fecha de pago No Nula.



The screenshot shows a PostgreSQL query editor with the following query:

```
1 SELECT COUNT(paid_date) FROM orders
```

The results are displayed in a table with the following structure:

count	bigint
1	16

Función COUNT (DISTINCT customer_num) – Mostrar cuántos clientes nos pusieron Órdenes de Compra.



The screenshot shows a PostgreSQL query editor with the following query:

```
1 SELECT COUNT(DISTINCT customer_num) cantidadClientes FROM orders
```

Función SUM (unit_price*quantity) – Función Agregada de fórmulas aritméticas. Mostrar cuánto dinero nos ingreso por los ítems de todas las Órdenes de compra.



The screenshot shows a PostgreSQL query editor with the following query:

```
1 SELECT SUM(unit_price*quantity) totalVendido FROM items
```



Varias funciones combinadas. Informar primer fecha de Orden de Compra, última fecha de orden de compra y cantidad de órdenes de compra.



The screenshot shows a PostgreSQL query editor interface. The query editor tab is active, displaying a SQL query. Below the query editor, the 'Data Output' tab is active, showing the results of the query in a table format. The table has three columns: 'primera compra date', 'ultima compra date', and 'cantidadcompras bigint'. The first row of data shows the first purchase date as '2015-05-16', the last purchase date as '2020-01-10', and the total number of purchases as '24'.

```

1 SELECT MIN(order_date) primeraCompra, MAX(order_date) ultimaCompra,
2        COUNT(*) cantidadCompras
3 FROM orders

```

	primera compra date	ultima compra date	cantidadcompras bigint
1	2015-05-16	2020-01-10	24

Cláusulas GROUP BY y HAVING

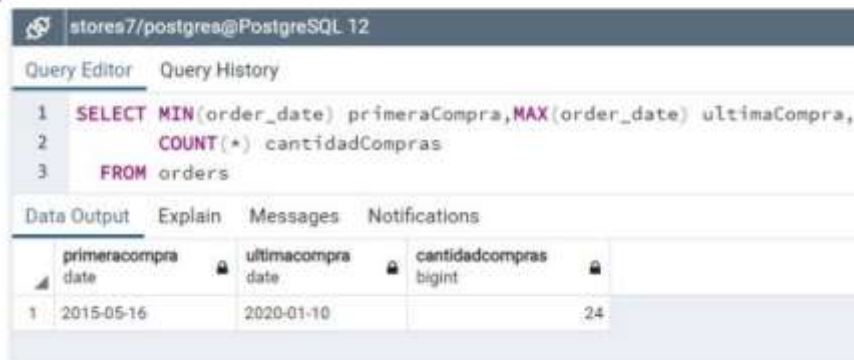
La cláusula **GROUP BY** sirve para agrupar filas a partir de una o varias columnas tomadas como clave.

Su potencial máximo se logra cuando lo combinamos con funciones agregadas, ya que rápidamente y con pequeños cambios en la consulta podríamos tener resultados sumarios o agregados por distintas dimensiones.

Obviamente que dependiendo del volumen de datos, esto implicaría implementar algún mecanismo para agilizar la consulta, como por ejemplo índices.



Volviendo a esta consulta, vemos que nos muestra, la primer orden de compra, la última orden y la cantidad de órdenes de compra.



The screenshot shows a PostgreSQL query editor with the following query:

```
1 SELECT MIN(order_date) primeraCompra, MAX(order_date) ultimaCompra,
2        COUNT(*) cantidadCompras
3 FROM orders
```

The results are displayed in a table with the following columns: primeraCompra (date), ultimaCompra (date), and cantidadCompras (bigint).

	primeraCompra date	ultimaCompra date	cantidadCompras bigint
1	2015-05-16	2020-01-10	24

Vamos a modificar esta consulta, obteniendo la misma información, pero agrupada por cliente.

Vamos a modificar esta consulta, obteniendo la misma información, pero agrupada por cliente.

```
SELECT customer_num, MIN(order_date) primeraCompra,
       MAX(order_date) ultimaCompra,
       COUNT(*) cantidadCompras
FROM orders
```



The screenshot shows the same query as before, but it has failed with an error. The error message is:

```
ERROR: column "orders.customer_num" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT customer_num, MIN(order_date) primeraCompra, MAX(order...
```

Below the error message, the text "Por qué da error?" is written in red.

Cuando queremos combinar en un SELECT atributos desagrupados con funciones agregadas, es necesario que dichos atributos sean incluidos en la cláusula GROUP BY.

No se pueden combinar atributos desagrupados con funciones agregadas, es necesario agruparlos en la cláusula GROUP BY.

Vamos a modificar esta consulta, obteniendo la misma información, pero agrupada por cliente.

```
SELECT customer_num, MIN(order_date) primeraCompra,  
        MAX(order_date) ultimaCompra,  
        COUNT(*) cantidadCompras  
FROM orders  
GROUP BY customer_num
```

En la cláusula GROUP BY ponemos los atributos no agregados que están en la cláusula SELECT.

```
SELECT customer_num, MIN(order_date) primeraCompra,  
        MAX(order_date) ultimaCompra, COUNT(*) cantidadCompras  
FROM orders  
GROUP BY customer_num
```



Observamos de los clientes que compraron, tenemos la cantidad de Órdenes de compra, la fecha de la primer y ultima compra.



Observamos que haciendo un pequeño cambio a la consulta, tenemos la cantidad de Órdenes de compra, la fecha de la primer y ultima compra, agrupados por año y mes.

```
SELECT DATE_PART('year',order_date) anio,
       DATE_PART('month',order_date) mes, MIN(order_date)
primeraCompra, MAX(order_date) ultimaCompra,
COUNT(*) cantidadCompras
FROM orders
GROUP BY DATE_PART('year',order_date),
         DATE_PART('month',order_date)
```

```
SELECT DATE_PART('year',order_date) anio,
DATE_PART('month',order_date) mes, MIN(order_date) primeraCompra,
MAX(order_date) ultimaCompra, COUNT(*) cantidadCompras
FROM orders
GROUP BY DATE_PART('year',order_date),
         DATE_PART('month',order_date)
ORDER BY 1,2
```

Ordenamos la
salida por año y
mes.

```
stores7/postgres@PostgreSQL 12
Query Editor  Query History
1  SELECT DATE_PART('year',order_date) anio,
2  DATE_PART('month',order_date) mes, MIN(order_date) primeraCompra,
3  MAX(order_date) ultimaCompra, COUNT(*) cantidadCompras
4  FROM orders
5  GROUP BY DATE_PART('year',order_date),DATE_PART('month',order_date)
6  ORDER BY 1,2
7
```



Aplicamos la cláusula HAVING para filtrar los filas de nuestro conjunto de datos de salida cuya cantidad de compras sea menor a 9.

```
SELECT DATE_PART('year',order_date) anio,
DATE_PART('month',order_date) mes, MIN(order_date)
primeraCompra, MAX(order_date) ultimaCompra, COUNT(*)
cantidadCompras
FROM orders
GROUP BY DATE_PART('year',order_date),
DATE_PART('month',order_date)
HAVING COUNT(*) < 9
ORDER BY cantidadCompras DESC
```

Cláusula HAVING con error ya que no se pueden atributos que no estén en la cláusula GROUP BY.



```
1 SELECT DATE_PART('year',order_date) anio,
2 DATE_PART('month',order_date) mes, MIN(order_date) primeraCompra, MAX(order_date) ultimaCompra, COUNT(*) cantidadCompras
3 FROM orders
4 GROUP BY DATE_PART('year',order_date), DATE_PART('month',order_date)
5 HAVING order_num > 1010
6 ORDER BY cantidadCompras DESC
7
```

ERROR: column "orders.order_num" must appear in the GROUP BY clause or be used in an aggregate function
LINE 5: HAVING order_num > 1010
SQL state: 42883
Character: 255

Poner la condición en la cláusula WHERE

Forma correcta de realizarlo:



```
1 SELECT DATE_PART('year',order_date) anio,
2 DATE_PART('month',order_date) mes, MIN(order_date) primeraCompra, MAX(order_date) ultimaCompra, COUNT(*) cantidadCompras
3 FROM orders
4 WHERE order_num > 1010
5 GROUP BY DATE_PART('year',order_date), DATE_PART('month',order_date)
6 ORDER BY cantidadCompras DESC
7
```



Cláusula LIMIT

Otro caso para el LIMIT, obtener los tres clientes con mayor cantidad de órdenes de compra.

```
SELECT customer_num, MIN(order_date) primeraCompra, MAX(order_date) ultimaCompra,  
COUNT(*) cantidadCompras  
FROM orders  
GROUP BY customer_num  
ORDER BY cantidadCompras DESC  
LIMIT 3
```

SQL – DML

SQL – Operador INSERT

Se inserta una sola fila en la tabla

```
INSERT INTO nom_tabla [(lista de columnas)] (*) (**)  
VALUES (lista de valores) (**)
```

Se insertan multiples filas en la tabla

```
INSERT INTO nom_tabla [(lista de columnas)] (*) (**)  
VALUES (lista de valores),  
      (lista de valores),  
      (lista de valores)
```

```
INSERT INTO nom_tabla [(lista de columnas)]  
SELECT ... (***)
```

(*) La lista de columnas entre corchetes significa que es opcional.

(**) la lista de columnas y valores tienen como separador una coma.

(***) El select debe devolver una lista de columnas similar a la que espera recibir el INSERT.



```
INSERT INTO product_types
VALUES (375, 'Short Baño')
```

Inserta en la tabla tipo de productos un nuevo tipo de producto.

```
Query Editor Query History
1 INSERT INTO product_types VALUES (375, 'Short Baño')
```

```
INSERT INTO product_types
VALUES ('Short Baño', 375)
```

El problema de no poner la lista de columnas a insertar es que mi lista de valores debe respetar 100 el orden de las columnas en la tabla, acoplando mi consulta al modelo.

```
Query Editor Query History
1 INSERT INTO product_types VALUES ('Short Baño', 375)
2

Data Output Explain Messages Notifications
ERROR: invalid input syntax for type smallint: "Short Baño"
LINE 1: INSERT INTO product_types VALUES ('Short Baño', 375)
SQL state: 22P02
Character: 35
```

En este ejemplo, vemos que al querer insertar los valores fuera de orden, el comando falla debido a que espera un valor smallint en la primer posición.

Si la tabla hubiese tenido dos campos varchar, no nos habríamos dado cuenta del error, ya que el Motor de BD habría aceptado los datos erróneos.

```
INSERT INTO customer VALUES
(266, 'Godoy', 'Estela', 'Pintos
3325', 'Ituza SA', null, 'Buenos Aires',
null, null, null, null, null)
```

El problema de no poner la lista de columnas a insertar es que mi lista de valores debe respetar el orden de las columnas en la tabla, acoplando mi consulta al modelo.

```
Query Editor Query History
1 INSERT INTO customer VALUES (266, 'Godoy', 'Estela', 'Pintos 3325', 'Ituza SA', null, 'Buenos Aires', null, null, null, null)
2
```

En este caso vemos que el motor de BD acepto los datos erróneos, porque son del mismo tipo.

El apellido y nombre están invertidos y la direccion1 y la compañía también.




```
INSERT INTO customer VALUES
(267, 'Godoy', 'Estela', 'Pintos
3325',
'Ituza SA', null, 'Buenos Aires')
```

En este caso vemos que el motor de BD ingreso los datos en la Base de forma errónea y los aceptó debido a que eran del mismo tipo.

Query Editor Query History

```
1 INSERT INTO customer VALUES (267, 'Godoy', 'Estela',
2 'Pintos 3325', 'Ituza SA', null, 'Buenos Aires');
3 SELECT * FROM customer WHERE customer_num=267;
```

Data Output Explain Messages Notifications

customer_num	first_name	last_name	company	address1	address2
PK integer	character varying (10)	character varying (10)	character varying (20)	character varying (255)	character
1	267	Godoy	Estela	Pintos 3325	Ituza SA

```
INSERT INTO product_types
(stock_num, description)
VALUES (376, 'Short Rugby')
```

Inserta en la tabla tipo de productos un nuevo tipo de producto.

Agregando la lista de atributos podemos desacoplar la instrucción de la definición de la tabla.

```
INSERT INTO product_types
(stock_num, description)
VALUES (377, 'Camiseta Rugby'),
(378, 'Botines Rugby'),
(379, 'Short Rugby')
```

Inserta varias filas en la tabla tipo de productos de nuevos tipos de productos.

Agregando la lista de atributos podemos desacoplar la instrucción de la definición de la tabla.

Query Editor Query History

```
1 INSERT INTO product_types
2 (stock_num, description)
3 VALUES (377, 'Camiseta Rugby'),
4 (378, 'Botines Rugby'),
5 (379, 'Short Rugby');
```



```
INSERT INTO customer
(stock_num, lname, fname, address1,
Company, address2, city)
VALUES
(266, 'Godoy', 'Estela', 'Pintos 3325',
'Ituza SA', null, 'Buenos Aires')
```

Al agregar la lista de columnas, podemos insertar filas con una cantidad menor de valores a los que tiene la definición de la tabla y sin necesitar respetar el orden de los mismos en la tabla real.

Siempre y cuando la lista de columnas y la lista de valores coincidan

```
INSERT INTO closed_orders
SELECT * FROM orders
WHERE paid_date IS NOT null
```

En una tabla creada previamente con la misma estructura que la tabla ordenes, insertamos las filas que devuelve el SELECT

Query Editor QueryHistory

```
1 CREATE TABLE closed_orders
2
3 order_num smallint NOT NULL,
4 order_date date,
5 customer_num smallint NOT NULL,
6 ship_instruct character varying(40),
7 backlog character(1),
8 po_num varchar(18),
9 ship_date date,
10 ship_weight numeric(8,2),
11 ship_charge numeric(8,2),
12 paid_date date,
13 CONSTRAINT closed_orders_pk PRIMARY KEY (order_num),
14 CONSTRAINT closed_orders_customer_num_fk FOREIGN KEY (customer_num)
15 REFERENCES customer (customer_num) MATCH SIMPLE
16 ON UPDATE NO ACTION
17 ON DELETE NO ACTION
18
19
20 INSERT INTO closed_orders SELECT * FROM orders WHERE paid_date IS NOT NULL
```

Es fundamental que el select devuelva las mismas filas y en el mismo orden las columnas que la tabla destino.

En este ejemplo el insert está acoplado a la definición de la tabla closed_orders y el SELECT está acoplado a la definición de la tabla orders.

```
INSERT INTO closed_orders
(order_num, order_date, customer_num,
ship_instruct, backlog, po_num, ship_date,
ship_weight, ship_charge, paid_date)
SELECT order_num, order_date, customer_num,
ship_instruct, backlog, po_num, ship_date,
ship_weight, ship_charge, paid_date
FROM orders
WHERE paid_date IS NOT null
```

En una tabla creada previamente con la misma estructura que la tabla ordenes, insertamos las filas que devuelve el SELECT.

Con la lista de columnas en el INSERT y en el SELECT los desacoplamos de la definición de las tablas y viendo las instrucciones sabemos claramente que se está insertando y en donde.



Ejemplo con ALTER TABLE

```
INSERT INTO manufact
(manu_code, manu_name, lead_time, state)
VALUES ('DBL', 'DBLANDIT', 1, 'CA')
```

Query Editor Query History

```
1 ALTER TABLE manufact ALTER COLUMN f_alta_audit SET DEFAULT CURRENT_DATE;
2 ALTER TABLE manufact ALTER COLUMN d_usualta_audit SET DEFAULT USER;
3
4 INSERT INTO manufact (manu_code, manu_name, lead_time, state)
5 VALUES ('DBL', 'DBLANDIT', 1, 'CA');
6
7 SELECT * FROM manufact WHERE manu_code='DBL';
```

Data Output Explain Messages Notifications

manu_code	manu_name	lead_time	state	f_alta_audit	d_usualta_audit
(PK) character (3)	character varying (10)	smallint	character (2)	date	character varying (20)
1 DBL	DBLANDIT		1 CA	2020-10-17	postgres

Alteramos la tabla manufact agregando dos valores DEFAULT.
Se observa que en los atributos f_alta_audit y d_usualta_audit tienen los datos definidos como DEFAULT en la definición de la tabla.

DEFAULT CONSTRAINT

Cuando insertamos valores en una tabla y obviamos determinadas columnas, por DEFAULT tendrán valor NULL, salvo que con el constraint de DEFAULT les asignemos otro valor.

```
INSERT INTO empleados (nombre, apellido, cuit)
VALUES ('Lisandro', 'Ayestaran', 20235582487)
```

Query Editor Query History

```
1 CREATE TABLE empleados
2 (empleadoid SERIAL,
3 nombre VARCHAR(60),
4 apellido VARCHAR(60),
5 cuit BIGINT,
6 );
7
8 INSERT INTO empleados (nombre, apellido, cuit)
9 VALUES ('Lisandro', 'Ayestaran', 20235582487);
10
11 SELECT * FROM empleados;
```

Data Output Explain Messages Notifications

empleadoid	nombre	apellido	cuit
integer	character varying (60)	character varying (60)	bigint
1	Lisandro	Ayestaran	20235582487

SERIAL / SECUENCIA

Al insertar una fila en una tabla con un atributo con un tipo de dato SERIAL, NO se debe incluir dicha columna en la lista de columnas, ni poner valor en la lista de valores.

En el ejemplo vemos que la columna empleadoid no es incluida en la lista de columnas y en la lista de valores.

SQL – Operador UPDATE

Se modifica una o varias columnas de las filas que cumplan con la condición.

```
UPDATE nom_tabla  
SET columna=valor[, columna=valor...],  
[WHERE condiciones] (*)
```

(*) La cláusula **WHERE** es opcional, pero **CUIDADO** si no se pone condición el UPDATE se realizará sobre la **TOTALIDAD DE LAS FILAS DE LA TABLA**.

```
UPDATE customer  
SET company = 'ITBA', phone = '5555-5555'  
WHERE customer_num = 112
```

```
UPDATE empleados  
SET apellido='García'
```

WARNING

Este UPDATE no tiene la cláusula WHERE, por lo que se modificarán todas las filas de la tabla asignándole al apellido 'García'.

```
UPDATE products  
SET unit_price= unit_price*1,20  
WHERE manu_code='ANZ'
```

Modificar ciertas filas cambiando el valor de una columna por el resultado de una fórmula.



SQL – Operador DELETE

Se eliminan las filas que cumplan con la condición del DELETE.

```
DELETE FROM nom_tabla  
[WHERE condiciones] (*)
```

Se eliminan las filas que cumplan con la condición del DELETE, o sea cuyo customer_num sea 266.

```
DELETE FROM customer  
WHERE customer_num=266
```

Se eliminan las filas que cumplan con la condición del DELETE, o sea cuyo customer_num sea 104.

```
DELETE FROM customer  
WHERE customer_num=104
```

WARNING

Cuidado con la Integridad Referencial resguardada por las PK y las FK.



SQL – Transacciones

Mecanismos para garantizar la consistencia de datos

El DBMS cuenta con distintos mecanismos para poder asegurar la consistencia de los datos que existen en la/s Base/s de Datos.

Conceptos relacionados con la consistencia de datos

• TRANSACCIONES

- Es un conjunto de sentencias SQL que se ejecutan atómicamente en una unidad lógica de trabajo. Partiendo de que una transacción lleva la base de datos de un estado correcto a otro estado correcto, el motor posee mecanismos de manera de garantizar que la operación completa se ejecute o falle, no permitiendo que queden datos inconsistentes.
- Cada sentencia de alteración de datos (insert, update o delete) es una transacción en sí misma (singleton transaction).

• **LOGS TRANSACCIONALES:** Es un registro donde el motor almacena la información de cada operación llevada a cabo con los datos

• **RECOVERY:** Método de recuperación ante caídas.

Transacciones

Para definir una transacción debemos definir un conjunto de instrucciones precedidas por la sentencia **BEGIN WORK**, de esta manera las sentencias a continuación se ejecutarán de forma atómica. Pero para lo que es el concepto de transacción, una transacción puede finalizar correctamente o puede fallar; en el caso de finalizar correctamente, todos los datos actualizados se confirmarían en la base de datos y en caso de fallar o por una regla de negocio definida, se deberían deshacer todos los cambios hasta el comienzo de la transacción.

Para manejar estas acciones contamos con la sentencia **COMMIT WORK** para actualizar los datos en la base de datos, y con la sentencia **ROLLBACK WORK** para deshacer nuestra transacción.

BEGIN WORK;

```
INSERT INTO customer (customer_num, fname, lname)
VALUES (168, 'Rodrigo', 'Yanez');
INSERT INTO orders (order_num, customer_num, order_date)
VALUES (1545, 168, '2020-09-16');
INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
VALUES (1, 1545, 1, 'HRO', 10, 275);
INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
VALUES (2, 1545, 1, 'SMT', 25, 445);
```

En el ejemplo observamos que se inició una transacción y se realizaron varios inserts que representan una única unidad, o sea si algo fallase se deshacería toda la transacción. Ahora bien, observamos que la transacción no está finalizada, ni por **OK Commit**, ni por **Error Rollback**.



Transacciones (Cont.)

Caso 1 - Queremos abandonar la sesión de PGAdmin cerrando la ventana del Query Tool.

The screenshot shows the PGAdmin Query Editor with the following SQL code:

```

1 BEGIN WORK;
2
3 INSERT INTO customer (customer_num, first_name, last_name) VALUES (100, 'Rodrigo', 'Yanez');
4 INSERT INTO orders (order_num, customer_num, order_date) VALUES (1045, 100, '2020-09-10');
5 INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
6 VALUES (1, 1045, 1, '980', 10, 275);
7 INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
8 VALUES (2, 1045, 1, '981', 20, 445);
9
10 select * from items where stock_num=1;
  
```

A modal dialog titled "Commit transaction?" is displayed, asking: "The current transaction is not committed to the database. Do you want to commit or rollback the transaction?". The dialog has three buttons: "Cancel", "Rollback", and "Commit".

Transacciones (Cont.)

Caso 2 - Realizamos un Commit a la Transacción

The screenshot shows the PGAdmin Query Editor with the following SQL code:

```

1 BEGIN WORK;
2
3 INSERT INTO customer (customer_num, first_name, last_name) VALUES (100, 'Rodrigo', 'Yanez');
4 INSERT INTO orders (order_num, customer_num, order_date) VALUES (1045, 100, '2020-09-10');
5 INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
6 VALUES (1, 1045, 1, '980', 10, 275);
7 INSERT INTO items (item_num, order_num, stock_num, manu_code, quantity, unit_price)
8 VALUES (2, 1045, 1, '981', 20, 445);
9
10 COMMIT WORK;
  
```

The "Data Output" tab shows the results of the queries:

customer_num	first_name	last_name	company
100	Rodrigo	Yanez	

The "Messages" tab shows the following message:

```

13 SELECT * FROM customer WHERE customer_num=100;
14 SELECT * FROM orders WHERE order_num=1045;
  
```

Transacciones (Cont.)

Caso 3 - Realizamos un Rollback a la Transacción

Consultas Durante la Transacción

The screenshot shows the PGAdmin Query Editor with the following SQL code:

```

8 VALUES (2, 1045, 1, '981', 20, 445);
9
10 ROLLBACK WORK;
11
  
```

The "Data Output" tab shows the results of the queries:

customer_num	first_name	last_name	company
100	Rodrigo	Yanez	

The "Messages" tab shows the following message:

```

13 SELECT * FROM customer WHERE customer_num=100;
14 SELECT * FROM orders WHERE order_num=1045;
  
```

The "Data Output" tab shows the results of the queries after the rollback:

customer_num	first_name	last_name	company
100	Rodrigo	Yanez	

Secuencias

Los generadores de secuencias proveen una serie de números secuenciales, especialmente usados en entornos multiusuarios para generar una números secuenciales y únicos sin el overhead de I/O a disco ó lockeo transaccional.

Los motores de base de datos proveen diferentes formas de implementar secuencias a través de:

- Tipo de dato de una columna (Informix, PostgreSQL)
 - Propiedades de una columna (SqlServer, Mysql, DB2)
 - Objeto Sequence (Oracle, Informix, PostgreSQL, DB2, SqlServer)
-
- Tipo de Dato de una columna
 - Motor BD Informix/PostgreSQL **SERIAL, BIGSERIAL.**
 - Propiedades de una columna
 - Motor SqlServer, DB2 **IDENTITY**
 - Motor Mysql **AUTO_INCREMENT**
 - Objeto Sequence
Motores Oracle, Informix, PostgreSQL, DB2, SqlServer.
 - **CREATE SEQUENCE**

Tipo de Dato de una columna

El motor PostgreSQL posee varios tipo de datos SERIAL, BIGSERIAL que permiten realizar lo mismo que un objeto secuencia. Al insertar una fila en dicha tabla y asignarle un valor cero, el motor va a buscar el próximo nro. del más alto existente en la tabla.




```
CREATE TABLE tickets (
    ticketid BIGSERIAL,
    customer_num INTEGER REFERENCES customer,
    order_num INTEGER REFERENCES orders,
    ticket_date DATE,
    description VARCHAR(2000),
    d_alta_audit DATE DEFAULT CURRENT_DATE,
    d_usuario VARCHAR(20) DEFAULT USER);

INSERT INTO tickets (customer_num, order_num, ticket_date, description)
VALUES (168,1545,'16/10/2020',
        'El cliente reclama la entrega de productos')
```

Propiedades de una columna

Existen motores que poseen propiedades de columna que permite realizar lo mismo que una secuencia. Al insertar una fila en dicha tabla, el motor va a buscar el próximo nro. del más alto existente en la tabla.

Ej. SQLServer IDENTITY

```
CREATE TABLE ordenes (
    N_orden int IDENTITY (1, 1),
    N_cliente int NULL ,
    F_orden datetime NULL);

INSERT INTO ordenes
(n_cliente, f_orden)

VALUES (114,'2020-03-03')
```

Ej. MySQL AUTO_INCREMENT

```
CREATE TABLE animales
(animal_id INT AUTO_INCREMENT,
nombre CHAR(30) NOT NULL);

INSERT INTO animales
('perro'),
('gato');
```

Objeto SEQUENCE

Una secuencia debe tener un nombre, debe ser ascendente o descendente, debe tener definido el intervalo entre números, tiene definidos métodos para obtener el próximo número ó el actual (entre otros).

Ej. SEQUENCE PostGreSQL

```
CREATE SEQUENCE sq_numerador
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
START WITH 3000
```

```
INSERT INTO orders (order_num, customer_num, order_date)
VALUES (NEXT('sq_numerador'), 168, '2020-10-15');
```

```
INSERT INTO items (order_num, item_num, stock_num, manu_code,
quantity, unit_price)
VALUES (CURRVAL('sq_numerador'),1,1,'SMT',30, 250);
```

Ejemplo de actualización de secuencia con valor específico

```
SELECT SETVAL('sq_numerador', 4000) FROM orders LIMIT 1
```

Métodos asociados a una secuencia:

NEXTVAL('nombreSecuencia') - Devuelve el proximo valor de la secuencia actualizandola.

CURRVAL('nombreSecuencia') - Devuelve el valor actual de la secuencia.

SETVAL('nombreSecuencia', 4000) Actualiza la secuencia con un número específico.

Views

Una view es un conjunto de columnas, ya sea reales o virtuales, de una misma tabla o no, que puede contar con algún filtro para condicionar el resultado.

De esta forma, es una presentación adaptada de los datos contenidos en una o más tablas, o en otras vistas. Una vista toma la salida resultante de una consulta y la trata como una tabla.

Se pueden usar vistas en la mayoría de las situaciones en las que se pueden usar tablas.

- Tiene un nombre específico
- No contiene datos almacenados y no aloca espacio de almacenamiento.
- Está definida por una consulta sobre una o varias tablas.
- Solo se almacena la metadata de su definición.



Las vistas se pueden utilizar para:

- Suministrar un **nivel adicional de seguridad** restringiendo el acceso a un conjunto predeterminado de filas o columnas de una tabla.
- **Ocultar la complejidad** de los datos.
- **Simplificar sentencias** al usuario.
- Presentar los datos desde una **perspectiva diferente**.
- **Aislar a las aplicaciones** de los cambios en la tabla base.

RESTRICCIONES

- Tener en cuenta ciertas restricciones para el caso de Actualizaciones:
 - Si en la tabla existieran campos que no permiten nulos y en la view no aparecen, los inserts fallarían.
 - Si en la view no aparece la primary key los inserts podrían fallar.
 - Se puede borrar filas desde una view que tenga una columna virtual.
 - Con la opción WITH CHECK OPTION, se puede actualizar siempre y cuando el chequeo de la opción en el where sea verdadero.

```
CREATE VIEW V_clientes_california  
(codigo, apellido, nombre)  
AS
```

```
SELECT customer_num, lname, fname  
FROM customer  
WHERE state='CA'
```



```
CREATE VIEW V_clientes_california_WCK  
(codigo, apellido, nombre, estado)  
AS
```

```
SELECT customer_num, lname, fname, state  
FROM customer  
WHERE state='CA'
```

WITH CHECK OPTION

WITH CHECK OPTION realiza un chequeo de integridad de los datos a insertar o modificar, los cuales deben cumplir con las condiciones del WHERE de la vista.

Tablas Temporales

Son tablas creadas cuyos datos son de existencia temporal.

No son registradas en las tablas del diccionario de datos.

No es posible alterar tablas temporarias. Si eliminarlas y crear los índices temporales que necesite una aplicación.

Las actualizaciones a una tabla temporal podrían no generar ningún log transaccional si así se configurara.

Tipos de Tablas

- De Sesión (locales)
- Globales

Tipos de Creación

- Explícita
- Implícita

De Sesión (locales)

Son visibles sólo para sus creadores durante la misma sesión (conexión) a una instancia del motor de BD.

Las tablas temporales locales se eliminan cuando el usuario se desconecta o cuando decide eliminar la tabla durante la sesión.

Globales

Las tablas temporales globales están visibles para cualquier usuario y sesión una vez creadas. Su eliminación depende del motor de base de datos que se utilice.

(En postgresql, la clausula Global está deprecada en las tablas temporales.)



Tipos de Creación

Creación Explícita.

Este tipo de creación se realiza mediante la instrucción CREATE. De manera explícita se deberá crear la tabla indicando el nombre, sus campos, tipos de datos y restricciones.

Creación Implícita

Se pueden crear tablas temporales a partir del resultado de una consulta SELECT.

Por qué utilizarlas?

Como almacenamiento intermedio de Consultas Muy Grandes:

Por ejemplo, se tiene una consulta SELECT que realiza **"JOINS"** con ocho tablas. Muchas veces las consultas con varios **"JOINS"** pueden funcionar de manera poco performante.

Una técnica para intentar es la de dividir una consulta grande en consultas más pequeñas. Si usamos tablas temporales, podemos crear tablas con resultados intermedios basados en consultas de menor tamaño, en lugar de intentar ejecutar una consulta única que sea demasiado grande y múltiples **"JOINS"**.

Para optimizar accesos a una consulta varias veces en una aplicación:

Por ejemplo, usted está utilizando una consulta que tarda varios segundos en ejecutarse, pero sólo muestra un conjunto acotado de resultados, el cual desea utilizar en varias áreas de su procedimiento almacenado, pero cada vez que se llama se debe volver a ejecutar la consulta general.

Para resolver esto, puede ejecutar la consulta una sola vez en el procedimiento, llenando una tabla temporal, de esta manera se puede hacer referencia a la tabla temporal en varios lugares en su código, sin incurrir en una sobrecarga de resultados adicional.

Para almacenar resultados intermedios en una aplicación:

Por ejemplo, usted necesita en un determinado proceso generar información que se irá actualizando y/o transformando en distintos momentos de la ejecución, sin querer actualizar o impactar a tablas reales de la BD hasta el final del procedimiento.

Para resolver esto, puede crear una tabla temporal de sesión durante la ejecución del procedimiento, realizando en ella inserciones, modificaciones, borrado y/o transformación de datos. Al llegar al final del procedimiento, con los datos existentes en la tabla temporal se actualizará la o las tablas físicas de la BD que corresponda.



Tablas Temporales de Sesión

×

Creación Explícita

```
CREATE TEMPORARY TABLE pending_orders(
  order_num smallint NOT NULL,
  order_date date,
  customer_num smallint NOT NULL,
  ship_instruct character varying(40),
  backlog character(1),
  po_num character varying(10),
  ship_date date,
  ship_weight numeric(8,2),
  ship_charge numeric(6,2),
  paid_date date );

INSERT INTO pending_orders
SELECT * FROM orders WHERE paid_date IS NOT NULL
```

Creación Implícita

```
CREATE TEMPORARY TABLE pending_orders
AS
SELECT * FROM orders
WHERE paid_date IS NOT NULL
```

Tanto el ejemplo de creación explícita y el de implícita generarán la misma tabla temporal con los mismos datos.

Tablas Temporales de Sesión

Sesión 1

Query Editor Query History

```
1 CREATE TEMPORARY TABLE pending_orders;
2   order_num smallint NOT NULL,
3   order_date date,
4   customer_num smallint NOT NULL,
5   ship_instruct character varying(40),
6   backlog character(1),
7   po_num character varying(10),
8   ship_date date,
9   ship_weight numeric(8,2),
10  ship_charge numeric(6,2),
11  paid_date date );
12
13
14 INSERT INTO pending_orders
15 SELECT * FROM orders WHERE paid_date IS NOT NULL;
16
17 SELECT * FROM pending_orders
```

Data Output Explain Messages Notifications

order_num	order_date	customer_num	ship_instruct	backlog	po_num
1	1991-10-10-10	100	Express	Y	077000
2	1991-10-10-11	101	NO surface delivery	N	0100
3	1991-10-10-10	100	Express	Y	077000
4	1991-10-10-10	100	NO surface delivery	Y	0000

Sesión 2

Query Editor Query History

```
1 SELECT * FROM pending_orders
```

Data Output Explain Messages Notifications

ERROR: relation "pending_orders" does not exist
LINE 1: SELECT * FROM pending_orders

SQL STATE: 42P01
CHARACTER: 10

En este ejemplo creamos una tabla temporal de sesión en la sesión 1, le insertamos datos y luego consultamos la tabla temporal.

En una sesión 2 se quiere consultar la tabla temporal creada y la misma no existe, esto ocurre porque las tablas de sesión solo viven dentro de la sesión en la que fueron creadas.

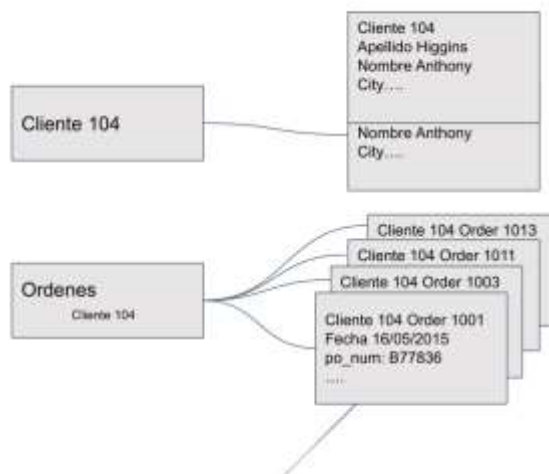
SQL – DML

•DML – Data Manipulation Language

- SELECT
 - INNER JOIN
 - OUTER JOIN

Si queremos hacer coincidir filas de dos o más tablas a partir de un atributo con valores comunes, por ej.

A partir del customer_num de la tabla **orders** obtener el lname y fname de cada cliente de la tabla customer.



INNER JOIN (clave simple)

Se realiza un matcheo de las filas de una tabla que coincidan a través de un atributo o combinación de atributos con filas de otras tabla.

El INNER JOIN solo devolverá las filas que coincidan.

```
SELECT c.customer_num, fname, lname, order_num, order_date
FROM customer c INNER JOIN orders o
ON (c.customer_num = o.customer_num)
```

INNER JOIN (clave compuesta)

Queremos obtener el código de producto, total vendido y el código de unidad de medida. Para esto deberemos tomar la unit_code de la tabla products.

SQL Query:

```
SELECT f.stock_num, f.manu_code, SUM(f.unit_price*quantity) tot_producto, unit_code
FROM items f INNER JOIN products p
ON (f.stock_num = p.stock_num and f.manu_code=p.manu_code)
GROUP BY f.stock_num, f.manu_code, unit_code
```

Data Output:

stock_num	manu_code	tot_producto	unit_code
1	110 SHM	228.00	16
2	6 SMT	184.00	12
3	1 SMT	16325.00	6
4	111 SHM	4499.81	16
5	2 HRD	252.00	13
6	305 ANZ	1248.00	13
7	5 ANZ	5484.00	19
8	114 PRC	130.00	7
9	4 HRD	940.00	13

Diagram illustrating the join operation between items and products tables. The items table has columns stock_num, manu_code, and unit_code. The products table has columns stock_num, manu_code, and unit_code. The join is performed on stock_num and manu_code.

Por qué utilizamos alias de tabla en algunas columnas?

INNER JOIN (mas de dos tablas)

Obtener el código de producto, total vendido y la descripción de la unidad de medida. Para esto deberemos tomar la unit_code de la tabla products y el unit_descr de la tabla units.

```
Query Editor Query History
1 SELECT f.stock_num, f.manu_code, SUM(f.unit_price*quantity) tot_products, unit_descr
2
3 FROM items
4 INNER JOIN products p ON (f.stock_num = p.stock_num AND f.manu_code = p.manu_code)
5 INNER JOIN units u ON (p.unit_code = u.unit_code)
6
7 GROUP BY f.stock_num, f.manu_code, unit_descr
```

INNER JOIN (mas de dos tablas)

Obtener el cod_cliente, apellido y nombre de cliente, orden de compra, fecha emisión, nro. de stock, código fabricante, nombre_fabricante, descripción tipo producto, nro de ítem, cantidad y precio unitario.

```
1 SELECT c.customer_num, fname,lname, o.order_num, order_date, item_num, f.stock_num, f.manu_code, manu_name, p.description,
2 quantity, unit_price, (quantity*unit_price) tot_item
3
4 FROM customer c
5 INNER JOIN orders o ON (c.customer_num = o.customer_num)
6 INNER JOIN items i ON (o.order_num = i.order_num)
7 INNER JOIN product_types p ON (f.stock_num = p.stock_num)
8 INNER JOIN manufact m ON (f.manu_code = m.manu_code)
```

INNER JOIN (mas de dos tablas con condiciones adicionales en WHERE)

Mismo ejemplo anterior pero con Condiciones, fabricante de nombre Hero, órdenes del mes de mayo

```
1 SELECT c.customer_num, fname,lname, o.order_num, order_date, item_num, f.stock_num, f.manu_code, manu_name, p.description,
2 quantity, unit_price, (quantity*unit_price) tot_item
3
4 FROM customer c
5 INNER JOIN orders o ON (c.customer_num = o.customer_num)
6 INNER JOIN items i ON (o.order_num = i.order_num)
7 INNER JOIN product_types p ON (f.stock_num = p.stock_num)
8 INNER JOIN manufact m ON (f.manu_code = m.manu_code)
9 WHERE manu_name='Hero' AND DATE_PART('month', order_date)=5 AND DATE_PART('year', order_date)=2015
```

INNER JOIN (mas de dos tablas con condiciones adicionales en WHERE)

Mismo ejemplo anterior pero con Condiciones, cliente de nombre Husky, órdenes del mes de mayo del 2015 y precio unitario mayor que 126.

```
1 SELECT c.customer_num, fname,lname, o.order_num, order_date, item_num, f.stock_num, f.manu_code, manu_name, p.description,
2 quantity, unit_price, (quantity*unit_price) tot_item
3
4 FROM customer c
5 INNER JOIN orders o ON (c.customer_num = o.customer_num)
6 INNER JOIN items i ON (o.order_num = i.order_num)
7 INNER JOIN product_types p ON (f.stock_num = p.stock_num)
8 INNER JOIN manufact m ON (f.manu_code = m.manu_code)
9 WHERE manu_name='Husky' AND DATE_PART('month', order_date)=5 AND DATE_PART('year', order_date)=2015 AND unit_price > 126
```



OUTER JOIN

El Outer join, mostrará todas las filas de la tabla dominante machine o no con la otra tabla.
El Outer puede ser LEFT. (tabla izquierda dominante), RIGHT (tabla derecha dominante) o FULL (ambas tablas dominantes)

La tabla customer tienen clientes que no han puesto ordenes de compra y clientes que no.

Customer INNER JOIN orders

Customer LEFT JOIN orders

```
SELECT c.customer_num, fname, lname, COUNT(order_num) cantOrdenes
FROM customer c INNER JOIN orders o
ON (c.customer_num = o.customer_num)
GROUP BY c.customer_num, fname, lname
```

```
SELECT c.customer_num, fname, lname, COUNT(order_num)
cantOrdenes
FROM customer c LEFT JOIN orders o
ON (c.customer_num = o.customer_num)
GROUP BY c.customer_num, fname, lname
```

OUTER JOIN

Comparemos resultados.

Inner Join 17 clientes que compraron.

```
11 SELECT c.customer_num, fname, lname, COUNT(order_num) cant_ordenes
12 FROM customer c JOIN orders o
13 ON (c.customer_num = o.customer_num)
14 GROUP BY c.customer_num, fname, lname
15 ORDER BY 4 DESC
16
```

customer_num [PK] smallint	fname character varying (15)	lname character varying (15)	cant_ordenes bigint
104	Anthony	Higgins	4
117	Arnold	Spies	2
106	George	Volken	2
110	Roy	Jakger	2
127	Eve	Sattler	1
124	Chris	Pittman	1
121	Jason	Wallace	1
116	Jean	Parnowski	1
130	Fred	Jewell	1
111	Francis	Kovak	1
123	Martin	Horton	1
126	Ellen	Reese	1
122	Cathy	O'Brien	1
119	Bob	Shuster	1
101	Ludwig	Paul	1
118	Alfred	Grant	1
112	Margaret	Lemon	1

Left Join 28 clientes,
17 con cantOrdenes mayor que 0 y
11 clientes con cantOrdenes=0

```
11 SELECT c.customer_num, fname, lname, COUNT(order_num) cant_ordenes
12 FROM customer c LEFT JOIN orders o
13 ON (c.customer_num = o.customer_num)
14 GROUP BY c.customer_num, fname, lname
15 ORDER BY 4 DESC
16
```

customer_num [PK] smallint	fname character varying (15)	lname character varying (15)	cant_ordenes bigint
127	Sam	Sattler	1
124	Chris	Pittman	1
121	Jason	Wallace	1
128	Fred	Jewell	1
101	Ludwig	Paul	1
115	Alfred	Grant	1
112	Margaret	Lemon	1
119	Bob	Shuster	1
116	Jean	Parnowski	1
111	Francis	Kovak	1
125	Martin	Horton	1
126	Ellen	Reese	1
122	Cathy	O'Brien	1
107	Charles	Ryan	0
102	Carole	Sedler	0
109	Jane	Miller	0
118	Dick	Baumer	0
108	Donald	Gunn	0
113	Lana	Beatty	0
125	James	Henry	0
114	Frank	Abernathy	0
128	Frank	Lemon	0
103	Philip	Cune	0
105	Raymond	Vector	0

OUTER JOIN

Comparemos resultados , pero con un RIGHT JOIN.

Inner Join 17 clientes que compraron.

```
21 SELECT c.customer_num, fname, lname, count(order_num) cant_ordenes
22 FROM customer c JOIN orders o
23 ON c.customer_num = o.customer_num
24 GROUP BY c.customer_num, fname, lname
25 ORDER BY c.custid
```

customer_num	fname	lname	cant_ordenes
104	Anthony	Higgins	1
105	Arnold	Sipes	1
106	George	Watson	1
107	Ray	Walter	1
108	John	Walter	1
109	John	Walter	1
110	John	Walter	1
111	John	Walter	1
112	John	Walter	1
113	John	Walter	1
114	John	Walter	1
115	John	Walter	1
116	John	Walter	1
117	John	Walter	1
118	John	Walter	1
119	John	Walter	1
120	John	Walter	1
121	John	Walter	1
122	John	Walter	1
123	John	Walter	1
124	John	Walter	1
125	John	Walter	1
126	John	Walter	1
127	John	Walter	1
128	John	Walter	1
129	John	Walter	1
130	John	Walter	1
131	John	Walter	1
132	John	Walter	1
133	John	Walter	1
134	John	Walter	1
135	John	Walter	1
136	John	Walter	1
137	John	Walter	1
138	John	Walter	1
139	John	Walter	1
140	John	Walter	1
141	John	Walter	1
142	John	Walter	1
143	John	Walter	1
144	John	Walter	1
145	John	Walter	1
146	John	Walter	1
147	John	Walter	1
148	John	Walter	1
149	John	Walter	1
150	John	Walter	1
151	John	Walter	1
152	John	Walter	1
153	John	Walter	1
154	John	Walter	1
155	John	Walter	1
156	John	Walter	1
157	John	Walter	1
158	John	Walter	1
159	John	Walter	1
160	John	Walter	1
161	John	Walter	1
162	John	Walter	1
163	John	Walter	1
164	John	Walter	1
165	John	Walter	1
166	John	Walter	1
167	John	Walter	1
168	John	Walter	1
169	John	Walter	1
170	John	Walter	1
171	John	Walter	1
172	John	Walter	1
173	John	Walter	1
174	John	Walter	1
175	John	Walter	1
176	John	Walter	1
177	John	Walter	1
178	John	Walter	1
179	John	Walter	1
180	John	Walter	1
181	John	Walter	1
182	John	Walter	1
183	John	Walter	1
184	John	Walter	1
185	John	Walter	1
186	John	Walter	1
187	John	Walter	1
188	John	Walter	1
189	John	Walter	1
190	John	Walter	1
191	John	Walter	1
192	John	Walter	1
193	John	Walter	1
194	John	Walter	1
195	John	Walter	1
196	John	Walter	1
197	John	Walter	1
198	John	Walter	1
199	John	Walter	1
200	John	Walter	1

Left Join 28 clientes,
17 con cantOrdenes mayor que 0 y
11 clientes con cantOrdenes=0

```
21 SELECT c.customer_num, fname, lname, count(order_num) cant_ordenes
22 FROM customer c LEFT JOIN orders o
23 ON c.customer_num = o.customer_num
24 GROUP BY c.customer_num, fname, lname
25 ORDER BY c.custid
```

customer_num	fname	lname	cant_ordenes
104	Anthony	Higgins	1
105	Arnold	Sipes	1
106	George	Watson	1
107	Ray	Walter	1
108	John	Walter	1
109	John	Walter	1
110	John	Walter	1
111	John	Walter	1
112	John	Walter	1
113	John	Walter	1
114	John	Walter	1
115	John	Walter	1
116	John	Walter	1
117	John	Walter	1
118	John	Walter	1
119	John	Walter	1
120	John	Walter	1
121	John	Walter	1
122	John	Walter	1
123	John	Walter	1
124	John	Walter	1
125	John	Walter	1
126	John	Walter	1
127	John	Walter	1
128	John	Walter	1
129	John	Walter	1
130	John	Walter	1
131	John	Walter	1
132	John	Walter	1
133	John	Walter	1
134	John	Walter	1
135	John	Walter	1
136	John	Walter	1
137	John	Walter	1
138	John	Walter	1
139	John	Walter	1
140	John	Walter	1
141	John	Walter	1
142	John	Walter	1
143	John	Walter	1
144	John	Walter	1
145	John	Walter	1
146	John	Walter	1
147	John	Walter	1
148	John	Walter	1
149	John	Walter	1
150	John	Walter	1
151	John	Walter	1
152	John	Walter	1
153	John	Walter	1
154	John	Walter	1
155	John	Walter	1
156	John	Walter	1
157	John	Walter	1
158	John	Walter	1
159	John	Walter	1
160	John	Walter	1
161	John	Walter	1
162	John	Walter	1
163	John	Walter	1
164	John	Walter	1
165	John	Walter	1
166	John	Walter	1
167	John	Walter	1
168	John	Walter	1
169	John	Walter	1
170	John	Walter	1
171	John	Walter	1
172	John	Walter	1
173	John	Walter	1
174	John	Walter	1
175	John	Walter	1
176	John	Walter	1
177	John	Walter	1
178	John	Walter	1
179	John	Walter	1
180	John	Walter	1
181	John	Walter	1
182	John	Walter	1
183	John	Walter	1
184	John	Walter	1
185	John	Walter	1
186	John	Walter	1
187	John	Walter	1
188	John	Walter	1
189	John	Walter	1
190	John	Walter	1
191	John	Walter	1
192	John	Walter	1
193	John	Walter	1
194	John	Walter	1
195	John	Walter	1
196	John	Walter	1
197	John	Walter	1
198	John	Walter	1
199	John	Walter	1
200	John	Walter	1

Observamos que para utilizar un
RIGHT JOIN y obtener el mismo
resultado, lo único que hicimos fue
invertir el orden de las tablas.

JOIN AUTO REFERENCIADO (SELF REFERENCING JOIN)

La tabla customer tiene un atributo customer_num_referredBy, que nos indica quién fue el cliente que lo referencio.

Podríamos armar una consulta que nos diga nombre y apellido del referido y de quien lo referencio.

```
27 SELECT c1.fname nombreReferido, c1.lname apellidoReferido, c2.fname nombreReferente, c2.lname apellidoReferente
28 FROM customer c1 JOIN customer c2 ON (c1.customer_num_referredBy=c2.customer_num)
```

nombreReferido	apellidoReferido	nombreReferente	apellidoReferente
1	Carle	Seller	Ludwig
2	Philip	Carle	Ludwig
3	Anthony	Higgins	Philip
4	Raymond	Walter	Philip
5	George	Watson	Philip
6	Donat	Quinn	Charles
7	John	Miller	Charles
8	Lane	Beatty	Frances
9	Frank	Albertson	Frances
10	John	Partridge	Alfred
11	Arnold	Sipes	Alfred
12	Dick	Baer	Alfred

JOIN (COLUMNAS AMBIGUAS)

Cuando un atributo existe en más de una tabla del SELECT, es obligatorio identificar de qué tabla lo estamos tomando, utilizando el punto como separador (dot notation) tabla.columna o alias.columna.

```
23 SELECT customer_num, fname, lname, order_num
24 FROM customer c LEFT JOIN orders o
25 ON (c.customer_num = o.customer_num)
```

```
23 SELECT c.customer_num, fname, lname, order_num
24 FROM customer c LEFT JOIN orders o
25 ON (c.customer_num = o.customer_num)
```

```
23 SELECT c.customer_num, fname, lname, order_num
24 FROM customer c LEFT JOIN orders o
25 ON (c.customer_num = o.customer_num)
```

PRODUCTO CARTESIANO

El producto cartesiano es muy costoso para el motor de base de datos. En el caso que deban realizar alguno, deben tratar de achicar el working set lo máximo que puedan, proyectando solo las columnas que necesiten y condicionando con WHERE lo que pudiesen. **RECOMENDACIÓN NO LO UTILICEN.**

<pre> 1 SELECT * 2 FROM customer, orders </pre>											
<p>Cientes (12 cols)+Ordenes(10 cols)= 22 cols</p>											
<p>28 clientes * 23 ordenes =644 filas</p>											
<p>Que pasaría si fuesen 1000 clientes con 100000 ordenes?</p>											

SQL – Subquery en INSERT

INSERT INTO closed_orders
SELECT * FROM orders
WHERE paid_date **IS NOT** null

En una tabla creada previamente con la misma estructura que la tabla ordenes, insertamos las filas que devuelve el SELECT.

Es fundamental que el select devuelva las mismas filas y en el mismo orden las columnas que la tabla destino.

```

40 CREATE TABLE closed_orders
41 (
42     order_num smallint NOT NULL,
43     order_date date,
44     customer_num smallint NOT NULL,
45     ship_instructions character varying(40),
46     backlog character(1),
47     ps_num character varying(30),
48     ship_date date,
49     ship_weight numeric(8,2),
50     ship_charge numeric(8,2),
51     paid_date date,
52     CONSTRAINT orders_pk PRIMARY KEY (order_num),
53     CONSTRAINT orders_customer_num_fk FOREIGN KEY (customer_num)
54         REFERENCES customer (customer_num)
55 );
56
57 INSERT INTO closed_orders
58 SELECT * FROM orders
59 WHERE paid_date IS NOT null;
60
61 SELECT * FROM closed_orders;

```

SQL – SubQuery en DELETE

DELETE FROM customer

WHERE customer_num **NOT IN**

(**SELECT DISTINCT** customer_num **FROM** cust_calls)

1

AND customer_num **NOT IN**

(**SELECT DISTINCT** customer_num **FROM** orders)

2

AND customer_num **NOT IN**

(**SELECT DISTINCT** customer_num_referredBy **FROM** customer c2

3

WHERE customer_num_referredBy **IS NOT NULL**)

Utilizando tres subqueries como condiciones del DELETE, podríamos borrar todos los clientes de la Tabla customer, que no posean órdenes de compra asociadas y que no hayan referenciado a otro cliente y que no tengan llamados telefónicos.

SQL – SubQuery en DELETE

```
64 CREATE TEMP TABLE clientesParaBorrar AS SELECT * FROM customer;
65
66 DELETE FROM clientesParaBorrar
67     WHERE customer_num NOT IN
68         (SELECT DISTINCT customer_num FROM cust_calls)
69     AND customer_num NOT IN
70         (SELECT DISTINCT customer_num FROM orders)
71     AND customer_num NOT IN
72         (SELECT DISTINCT customer_num_referredBy FROM customer c2
73         WHERE customer_num_referredBy IS NOT NULL)
74
```

Para hacer la prueba y no borrar datos de nuestra tabla real corrimos el ejemplo en la tabla temporal creada clientesParaBorrar.



SQL – SubQuery en UPDATE

Subquery en CLAUSULA SET

```
1 ALTER TABLE state RENAME COLUMN code TO state; -- Corregido para que quede igual que el modelo pasado.
2
3 CREATE TEMP TABLE clientesParaBorrar AS SELECT * FROM customer;
4
5 UPDATE clientesParaBorrar
6 SET state=(SELECT state from state WHERE sname= 'Florida'); Subquery
7 WHERE customer_num=101;
8
9 SELECT cpb.lname, cpb.fname, cpb.state NuevoState, c.state OrigState
10 FROM clientesParaBorrar cpb JOIN customer c ON (cpb.customer_num = c.customer_num)
11 WHERE cpb.state <> c.state
```

El subquery puede retornar solo una única columna y fila. O sea un valor escalar, debido a que está combinado con un operador de igualdad (=).

Subquery en CLAUSULA WHERE con Subquery que devuelve valor escalar

```
13 UPDATE clientesParaBorrar
14 SET state='CA'
15 WHERE customer_num = (SELECT customer_num from customer WHERE lname= 'Pauli'); Subquery
16
17 SELECT lname, fname, state
18 FROM customer c
19 WHERE customer_num = 101
20
```

Data Output	Explain	Messages	Notifications									
<table><tr><th>lname</th><th>fname</th><th>state</th></tr><tr><td>character varying (15)</td><td>character varying (15)</td><td>character (2)</td></tr><tr><td>1 Pauli</td><td>Ludwig</td><td>CA</td></tr></table>	lname	fname	state	character varying (15)	character varying (15)	character (2)	1 Pauli	Ludwig	CA			
lname	fname	state										
character varying (15)	character varying (15)	character (2)										
1 Pauli	Ludwig	CA										

El subquery puede retornar solo una única columna y fila. O sea un valor escalar.



Subquery en CLAUSULA WHERE con Subquery que devuelve una columna con multiples filas.

```

22 UPDATE manufact SET lead_time=15
23 WHERE manu_code IN (SELECT DISTINCT manu_code FROM items);
24
25
26

```

Data Output Explain Messages Notifications

UPDATE 0

Query returned successfully in 668 msec.

Subquery

```

26 SELECT DISTINCT i.manu_code, lead_time
27 FROM items i JOIN manufact m ON (i.manu_code=m.manu_code)

```

Data Output Explain Messages Notifications

manu_code character(3)	lead_time smallint
1 ANZ	15
2 KAR	15
3 SMT	15
4 MSK	15
5 PPC	15
6 HRO	15
7 SHM	15
8 NKL	15
9 NRG	15

Comprobación

Modificamos el lead_time (tiempo de entrega) a 15 días para todos los proveedores de los cuales ya hayamos entregado productos.

Subquery en Lista de columnas.

```

110 SELECT customer_num, COUNT(order_num) cantOCcliente,
111      (SELECT count(*) FROM orders) CantTotalOC
112 FROM orders
113 GROUP BY customer_num
114

```

Data Output Explain Messages Notifications

customer_num smallint	cantoccliente bigint	canttotaloc bigint
1	101	23
2	116	91

El subquery devuelve la cantidad total de órdenes que existen y como si fuese un valor constante lo muestra en cada fila



Subquery en el FROM.

```

115 SELECT lname apellido, fname nombre, cliente, cantidad
116 FROM customer c1 JOIN
117     (SELECT customer_num cliente, count(order_num) cantidad
118      FROM orders GROUP BY customer_num) c2
119 ON (c1.customer_num = c2.cliente)
120 WHERE cantidad > 3

```

	apellido character varying (15)	nombre character varying (15)	cliente smallint	cantidad bigint
1	Higgins	Anthony	104	4

El subquery devuelve un conjunto de filas se asocian al alias c2 como si fuese el result set de una tabla y luego estos datos son JOINEADOS con la tabla customer c1.

Subquery en el WHERE.

```

101 SELECT CONCAT (lname, ', ', fname) customer_num
102 FROM customer
103 WHERE customer_num IN (SELECT customer_num FROM cust_calls
104                       GROUP BY customer_num HAVING COUNT(*) > 1)
105

```

	customer_num text
1	Parmelee, Jean

La consulta muestra los clientes que posean más de una llamada. Muchas veces este tipo de subqueries se pueden resolver mediante joins.

Subquery en el WHERE con un valor escalar.

```

106 SELECT COUNT(*) FROM customer
107 WHERE city = (SELECT city FROM customer WHERE lname='Higgins')
108

```

Data Output		Explain	Messages	Notifications
count	bigint			
1	5			

La consulta devuelve la cantidad de clientes que viven en la misma ciudad en la que vive Higgins, incluso el mismo.
Se puede resolver con joins.

```

80 SELECT customer_id, name
81 FROM customer c
82 WHERE NOT EXISTS (SELECT order_num FROM orders o
83                  WHERE o.customer_num = c.customer_num)
84

```

Data Output		Explain	Messages	Notifications
customer_id	name			
102	James			
103	Philip			
109	Reginald			
107	Charles			
108	Shirley			
109	Jane			
110	Lane			
114	Frank			
116	Dick			
120	Jessie			
120	Frank			

Subquery CORRELACIONADO.

Existe una correlación entre la ejecución del subquery y la fila en la que se está en el Query original.
En general no es muy performante!!

La consulta devuelve los clientes que para los que no existen órdenes de compra.
Muchas veces es necesario utilizarlo en subqueries en el WHERE de comandos UPDATE o DELETE.

SQL – Operador UNION

```

SELECT stock_num,manu_code
FROM products
WHERE unit_price < 25
UNION
SELECT stock_num,manu_code
FROM items
WHERE stock_num=5
ORDER BY 1,2

```

La cantidad de campos en cada consulta debe corresponderse.

Los campos en igual posición deben tener el mismo tipo de datos.

La cláusula ORDER BY va al final del último SELECT.

Solo se puede Ordenar referenciando por posición.

En el caso de filas con idénticos valores solo deja una de las dos.




```
SELECT stock_num,manu_code
FROM products
WHERE unit_price < 25
```

stock_num	manu_code
5	ANZ
5	ANZ
103	PRC
106	PRC
302	HRO
302	KAR

```
68 SELECT stock_num,manu_code
69 FROM products WHERE unit_price < 25
70 UNION
71 SELECT stock_num,manu_code
72 FROM items WHERE stock_num=5
73 ORDER BY 1,2
```

stock_num	manu_code
5	ANZ
5	NRG
5	SMT

```
SELECT stock_num,manu_code
FROM items
WHERE stock_num=5
```

stock_num	manu_code
5	NRG
5	SMT
5	ANZ
5	ANZ
5	NRG
5	ANZ
5	ANZ
5	SMT

```
SELECT stock_num PROD,manu_code FAB FROM products
WHERE unit_price < 25
```

UNION

```
SELECT stock_num,manu_code FROM items WHERE stock_num=5
ORDER BY 1,2
```

```
SELECT stock_num PROD,manu_code FAB FROM products
WHERE unit_price < 25
UNION
SELECT stock_num,manu_code FROM items WHERE stock_num=5
ORDER BY 1,2
```

PROD	FAB
5	ANZ
5	NRG
5	SMT
5	ANZ
103	PRC
106	PRC

La tabla de salida toma los nombres de columnas del primer SELECT.

```
SELECT 1 orden,customer_num,order_num, order_date
FROM orders WHERE customer_num=127
UNION
SELECT 3 orden,customer_num,order_num, order_date
FROM orders WHERE customer_num BETWEEN 111 AND 126
UNION
SELECT 2 orden,customer_num,order_num, order_date
FROM orders WHERE customer_num BETWEEN 1 AND 110
ORDER BY 1 ASC, 2 DESC
```

En este caso particular vemos que utilizamos un ordenamiento de filas determinado por una constante en cada select que dice que filas irán primero en la salida.

```
57 SELECT 1 orden,customer_num,order_num, order_date
58 FROM orders WHERE customer_num=127
59 UNION
60 SELECT 3 orden,customer_num,order_num, order_date
61 FROM orders WHERE customer_num BETWEEN 111 AND 126
62 UNION
63 SELECT 2 orden,customer_num,order_num, order_date
64 FROM orders WHERE customer_num BETWEEN 1 AND 110
65 ORDER BY 1 ASC, 2 DESC
```

orden	customer_num	order_num	order_date
1	1	127	1607-2016-01-20
3	3	110	1608-2016-06-03

Ordenamiento particular, primero el 127, luego del 110 al 1, y finalmente del 128 al 111

SQL – Operador UNION ALL

```
SELECT stock_num,manu_code FROM products WHERE unit_price < 25
UNION ALL
SELECT stock_num,manu_code FROM items WHERE stock_num=5
ORDER BY 1,2
```

```
51 SELECT stock_num,manu_code FROM products WHERE unit_price < 25
52 UNION ALL
53 SELECT stock_num,manu_code FROM items WHERE stock_num=5
54 ORDER BY 1,2
55
```

stock_num	manu_code
1	9 ANZ
2	9 ANZ
3	9 ANZ

El Operador UNION con la cláusula ALL, repite en la estructura de salida todas las filas de las tablas involucradas, sean o no iguales.



SQL – Operador INTERSECT

```
SELECT stock_num,manu_code
FROM products
WHERE unit_price < 25
INTERSECT
SELECT stock_num,manu_code
FROM items
WHERE stock_num=5
ORDER BY 1,2
```

```
41 SELECT stock_num,manu_code
42 FROM products
43 WHERE unit_price < 25
44 INTERSECT
45 SELECT stock_num,manu_code
46 FROM items
47 WHERE stock_num=5
48 ORDER BY 1,2
```

Mismas restricciones del UNION.

Devuelve las filas que están en ambas consultas.

SQL – Operador EXCEPT

```
SELECT stock_num,manu_code
FROM products
WHERE unit_price < 25
EXCEPT
SELECT stock_num,manu_code
FROM items
WHERE stock_num=5
ORDER BY 1,2
```

```
31 SELECT stock_num,manu_code
32 FROM products
33 WHERE unit_price < 25
34 EXCEPT
35 SELECT stock_num,manu_code
36 FROM items
37 WHERE stock_num=5
38 ORDER BY 1,2
```

Mismas restricciones del UNION.

Devuelve las filas del Primer SELECT que no están en la intersección con el Segundo SELECT.

SQL – Comparación U - Ua - I - E

stock_num	manu_code
1	5
2	5
3	103
4	106
5	302
6	302

SELECT
stock_num,manu_code
FROM products
WHERE unit_price < 25

SELECT stock_num,manu_code
FROM items
WHERE stock_num=5

stock_num	manu_code
1	5
2	5
3	5
4	5
5	5
6	5
7	5
8	5
9	5

UNION

stock_num	manu_code
5	ANZ
5	NRG
5	SMT
5	ANZ
103	PRC

UNION ALL

stock_num	manu_code
5	ANZ
5	ANZ
5	ANZ
5	ANZ
5	ANZ
5	ANZ

INTERSECT

stock_num	manu_code
5	ANZ

EXCEPT

stock_num	manu_code
5	ANZ
103	PRC
106	PRC
302	NRG
302	KAR