



SQL

Parte I

Presentación



Los motores de bases de datos relacionales, utilizan SQL (Standard Query Language) como lenguaje para poder indicarle las sentencias a ejecutar.

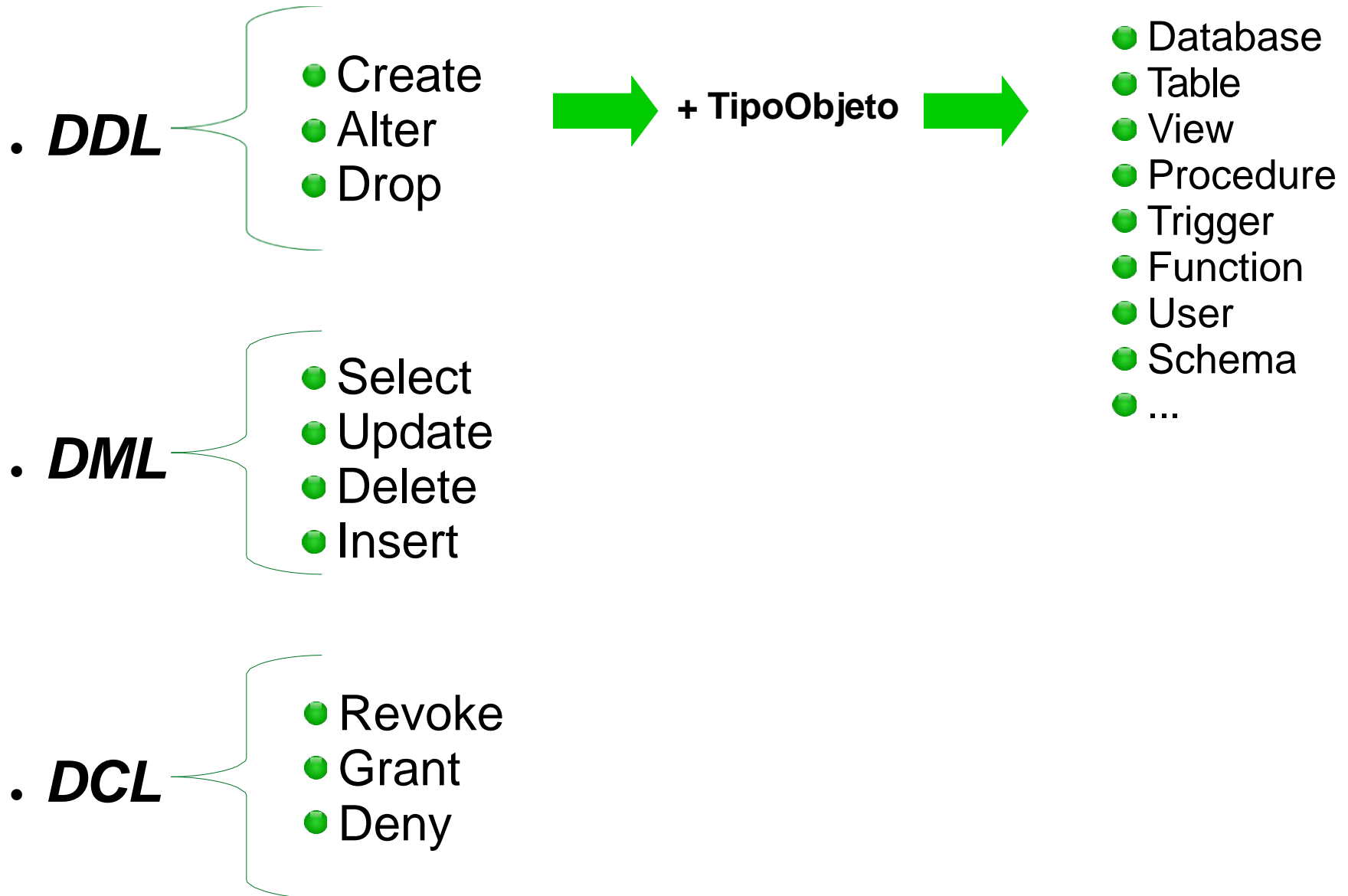
El lenguaje SQL está estandarizado por normas ISO. Cada uno de los modelos de SGBD incorporan un determinado standard.



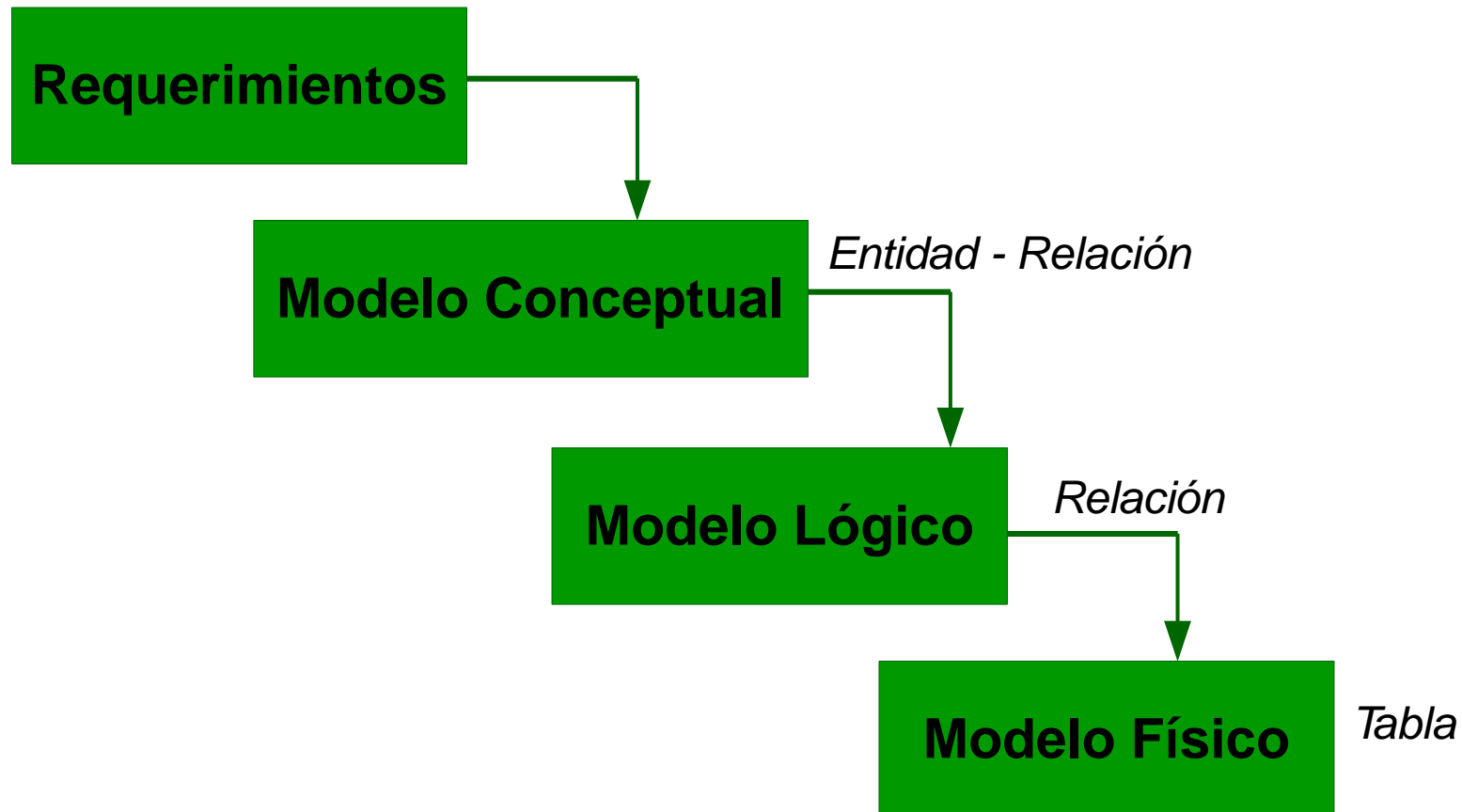
Clasificación

- ***DDL (Data Definition Language):*** Son todas aquellas instrucciones que permiten la definición de los distintos objetos de la base de datos.
- ***DML (Data Manipulation Language):*** Son aquellas instrucciones que permite la manipulación de los datos almacenados en la base de datos.
- ***DCL (Data Control Language):*** Son las aquellas instrucciones que permite el control de acceso a los datos.

Clasificación



Etapas del diseño



DDL: Table



Field / Column

DNI	Apellido	Nombre	FechaIngreso
35888333	Perez	Juan	01/02/2015
31999000	Sanchez	Ana	03/05/2019
25777111	Gomez	Lola	06/10/2006

Row / Record

Data / Value



DDL: CREATE TABLE

● Sintaxis Simple:

```
CREATE TABLE NombreTabla
(NombreCampo1 tipodedatos1 [NULL|NOT NULL] [PRIMARY KEY],
 NombreCampo2 tipodedatos2 [NULL|NOT NULL] [PRIMARY KEY],
 ...
 NombreCampoN tipodedatosN [NULL|NOT NULL] [PRIMARY KEY],
[CONSTRAINT NombreC PRIMARY KEY (NombreCampo1[,NombreCampoN])],
[CONSTRAINT NombreC FOREIGN KEY (NombreCampo1[,NombreCampoN])
REFERENCES NombreTablaC (NombreCampo1[,NombreCampoN])]
)
```

● Ejemplo Simple:

Alumno(Legajo, NyA, FechaIng, FechaNac, Mail)

Profesor (Legajo, NyA)

Curso(LegajoAlumno, LegProfesor)

```
CREATE TABLE Alumno
(Legajo int NOT NULL PRIMARY KEY,
 NyA varchar(100) NULL,
 FechaIng date,
 FechaNac date,
 Mail varchar(200)
)
```

```
CREATE TABLE Curso
(LegajoAlumno int NOT NULL,
 LegajoProfesor int NOT NULL,
 CONSTRAINT PKCURSO PRIMARY KEY
 (LegajoAlumno, LegajoProfesor),
 CONSTRAINT FKALUMNO FOREIGN KEY
 (LegajoAlumno) REFERENCES Alumno(Legajo),
 CONSTRAINT FKPROFESOR FOREIGN KEY
 (LegProfesor) REFERENCES Profesor(Legajo))
```

DDL: CREATE TABLE



● Tipos de Datos:

Clasificación	Tipo de datos	Tamaños
Numérico Entero	bit	0-1 (1 byte)
	Tinyint	0-255 (1 byte)
	Smallint	-2 ¹⁵ a 2 ¹⁵ (2 bytes)
	Int	-2 ³¹ a 2 ³¹ (4 bytes)
	bigint	-2 ⁶³ a 2 ⁶³ (8 bytes)
Numérico con Decimales	Numeric -Decimal	-2 ³⁸ a 2 ³⁸ (5-17bytes)
	Float – Real (n)	1-27 (7 digitos) (4 bytes)
		23-53 (15 digitos) (8 bytes)
Texto	char (n)	n=1 a 8000
	varchar(n)	
	text	2 ³¹
Fecha/Hora	Date	3 bytes
	Time	4 bytes
	Smalldatetime	4 bytes
	Datetime	8 bytes
	Timestamp /rowversion	8-16 bytes
Otros	Binary	2 ³¹
	XML	hasta 2 gb

* solo se citan los más relevantes



DDL: CREATE TABLE

● Sintaxis Completa:

CREATE TABLE

{ database_name.schema_name.table_name | schema_name.table_name | table_name }

[AS FileTable]

({ <column_definition>

| <computed_column_definition>

| <column_set_definition>

| [<table_constraint>] [,... n]

| [<table_index>] }

[,...n]

[PERIOD FOR SYSTEM_TIME (system_start_time_column_name
 , system_end_time_column_name)]

)

[ON { partition_scheme_name (partition_column_name)

| filegroup

| "default" }]

[TEXTIMAGE_ON { filegroup | "default" }]

[FILESTREAM_ON { partition_scheme_name

| filegroup

| "default" }]

[WITH (<table_option> [,...n])]

[;]

<column_definition> ::=

column_name <data_type>

[FILESTREAM]

[COLLATE collation_name]

[SPARSE]

[MASKED WITH (FUNCTION = ' mask_function ')

[CONSTRAINT constraint_name [DEFAULT consta

<data_type> ::=

[type_schema_name .] type_name

[(precision [, scale] | max |

[{ CONTENT | DOCUMENT }] xml_schema_c



DDL: CREATE TABLE

● Ejemplo:

Proyecto (ID, Nombre, fechainicio, HsTotales)

Cargo (ID, Descripcion)

Empleado (TipoDoc, NroDoc, NyA, Fingreso, Sueldo, IDCargo)

Trabaja (IDProyecto, TipoDoc, NroDoc, fechaInicio)

CREATE TABLE PROYECTO

(ID int not null primary key,
Nombre varchar(60),
Fechainicio date,
HsTotales smallint)

CREATE TABLE EMPLEADO

(TipoDoc tinyint not null,
Nrodoc bigint not null,
Nya varchar(150),
Fingreso date,
Sueldo numeric(12,2),
IDCargo smallint not null,
constraint PKEmpleado primary key (Nrodoc, tipodoc),
constraint FK1Empleado foreign key (IDCargo)
references Cargo (ID)

CREATE TABLE CARGO

(ID smallint not null primary key,
Descripcion varchar(60))

CREATE TABLE TRABAJA

(IDProyecto int not null,
TipoDoc tinyint not null,
NroDoc bigint not null,
FechaInicio date,
constraint PKTrabaja
primary key (idproyecto,tipodoc,nrodoc),
constraint FK1Trabaja foreign key (idproyecto)
references Proyecto (ID),
constraint FK2Trabaja foreign key (nrodoc,tipodoc)
references Empleado (nrodoc,tipodoc))

)



DDL: DROP TABLE

- **Sintaxis Simple:**

```
DROP TABLE NombreTabla
```

- **Ejemplo Simple:**

```
DROP TABLE Curso
```

```
DROP TABLE Alumno
```



DDL: DROP TABLE

- Sintaxis Completa:

```
DROP TABLE [ IF EXISTS ] { database_name.schema_name.table_name | schema_name.table_name |  
[ ; ]
```



DDL: ALTER TABLE

● Sintaxis Simple:

```
ALTER TABLE Nombretabla  
    ADD NombreCampo tipodatos [modificadores]
```

```
ALTER TABLE Nombretabla  
    DROP COLUMN NombreCampo [,NombreCampoN]
```

```
ALTER TABLE Nombretabla  
    ALTER COLUMN NombreCampo tipodatos [modificadores]
```

```
ALTER TABLE Nombretabla  
    ADD CONSTRAINT nombreC [PRIMARY KEY|FOREIGN KEY] ...
```

```
ALTER TABLE Nombretabla  
    DROP CONSTRAINT nombreC
```

● Ejemplo Simple:

```
ALTER TABLE Empleado ADD Mail varchar(200)  
ALTER TABLE Empleado DROP COLUMN Mail
```

```
ALTER TABLE Empleado ADD CONSTRAINT PKEmpleado PRIMARY KEY (NroDoc, TipoDoc)  
ALTER TABLE Empleado DROP CONSTRAINT FKCargo
```

DDL: ALTER TABLE



● Sintaxis Completa:

```
ALTER TABLE { database_name.schema_name.table_name | schema_name.table_name | table_name }
{
    ALTER COLUMN column_name
    {
        [ type_schema_name. ] type_name
        [ (
            {
                precision [ , scale ]
                | max
                | xml_schema_collection
            }
        ) ]
        [ COLLATE collation_name ]
        [ NULL | NOT NULL ] [ SPARSE ]
        | { ADD | DROP }
        { ROWGUIDCOL | PERSISTED | NOT FOR REPLICATION | SPARSE | HIDDEN }
        | { ADD | DROP } MASKED [ WITH ( FUNCTION = ' mask_function ' ) ]
    }
    [ WITH ( ONLINE = ON | OFF ) ]
    | [ WITH { CHECK | NOCHECK } ]

    | ADD
    {
        <column_definition>
        | <computed_column_definition>
        | <table_constraint>
        | <column_set_definition>
    } [ ,...n ]
    | [ system_start_time_column_name datetime2 GENERATED ALWAYS AS ROW START
        [ HIDDEN ] [ NOT NULL ] [ CONSTRAINT constraint_name ]
        DEFAULT constant_expression [WITH VALUES] ,
        system_end_time_column_name datetime2 GENERATED ALWAYS AS ROW END
        [ HIDDEN ] [ NOT NULL ] [ CONSTRAINT constraint_name ]
        DEFAULT constant_expression [WITH VALUES] ,
    ]
    PERIOD FOR SYSTEM_TIME ( system_start_time_column_name, system_end_time_column_name )
    | DROP
    [ {
        [ CONSTRAINT ] [ IF EXISTS ]
```

DDL: TABLE



● Resumen de comandos:

- `CREATE TABLE Nombretabla ...`
- `ALTER TABLE Nombretabla ...`
- `DROP TABLE Nombretabla ...`

DML



- Select
- Update
- Delete
- Insert

DML: Select



● Sintaxis Simple:

```
SELECT  [top n] [distinct] [* | <lista de campos>]  
FROM    tabla1 [,tabla2, ... tablan]  
[WHERE  <condicion> [AND|OR <condicion>]]  
[ORDER BY campo1 [asc|desc] [,campo2 [asc|desc] ...]]
```

● Ejemplo Simple:

```
SELECT  * FROM empleado
```

```
SELECT  * FROM empleado WHERE legajo=1
```

```
SELECT  mail, nya FROM empleado WHERE legajo>6 and nac='AR'
```

```
SELECT  nya FROM empleado WHERE legajo>=100 order by nya
```

```
SELECT  nya FROM empleado order by legajo desc
```

```
SELECT  nya FROM empleado order by nya desc, legajo asc
```

```
SELECT  TOP 10 legajo FROM empleado
```

```
SELECT  distinct nya FROM empleado
```



DML: Select. Condiciones

Las condiciones armadas en un where, podrían utilizar cualquiera de los siguientes Operadores:

Operador	Ejemplo
=	legajo = 1
<>	nombre <> 'Juan'
>	fechaNac > '2015-01-01'
<	legajo < 100
>=	fechaing >='2020-01-01'
<=	sueldo <= 20000
Like	nya like 'Perez%' / nya like '%perez%' / nya like '_perez'
between	Legajo between 20 and 50
IN (<lista de valores>)	legajo IN (10,20,30) / nombre IN ('Juan','Ana','Lola')
Is / is not	mail is null / mail is not null



DML: Like - IN

- **Like:** Es un operador que permite utilizar comodines.

Por ejemplo:

```
SELECT * FROM EMPLEADO WHERE APELLIDO LIKE 'PEREZ%'
```

```
SELECT * FROM EMPLEADO WHERE APELLIDO LIKE 'PEREZ_'
```

- **IN:** Es un operador que permite realizar comparaciones de OR con uno o más valores de la lista.

Por ejemplo:

```
SELECT * FROM EMPLEADO WHERE APELLIDO IN ('Perez', 'Lopez')
```

```
SELECT * FROM EMPLEADO WHERE IDCargo IN (1,2,10,55)
```

```
SELECT * FROM EMPLEADO WHERE IDCargo NOT IN (1,2,10,55)
```



DML: Alias

Los nombres de los campos y las tablas se pueden llamar a través de un alias. Para las tablas, esto permite que podamos referenciarla por el alias en lugar de usar el nombre de la tabla. En el caso de los campos, también podemos colocarle un alias y en el caso que se encuentren en el Select, mostrará con ese valor la salida.

```
SELECT campo [as] aliascampo  
  
FROM tabla [as] aliastabla
```

● Ejemplo Simple:

```
SELECT e.nombreapellido as nya, e.legajo as leg  
FROM empleado e  
Where e.legajo between 10 and 20
```

```
SELECT e.* FROM EMPLEADO e
```



DML: Any. Some. All

Estas sentencias permiten comparar un campo, utilizando cualquier operador, con un conjunto de valores devueltos a través de una consulta dinámica.

Any y **Some** actúan de igual modo y sólo existen ambos por compatibilidad.

```
campo <operador> ANY (<sentencia select>)
```

```
campo <operador> SOME (<sentencia select>)
```

```
campo <operador> ALL (<sentencia select>)
```

● Ejemplo Simple:

```
SELECT * FROM empleado WHERE legajo = ANY (select legajo from hist)
```

```
SELECT * FROM empleado WHERE legajo = SOME (select legajo from hist)
```

```
SELECT * FROM empleado WHERE legajo >= ANY (select legajo from hist)
```

```
SELECT * FROM empleado WHERE legajo >= ALL (select legajo from hist)
```



campo IN (<sentencia select>|<lista de valores>)

exists (<sentencia select>)

● Ejemplo Simple:

```
SELECT * FROM empleado WHERE legajo IN (select legajo from hist)
```

[illegible]



DML: Union. Union all

Estos operadores permiten unir el resultado de dos o más consultas distintas. Las condiciones que deben tener estas consultas, es que deben ser de igual grado (misma cantidad de campos) e igual dominio de los campos de igual orden. La diferencia entre Union y Union all, es que Union permite anular los duplicados, realizando un distinct luego de la union.

```
SELECT campo1, campo2 FROM tabla1
```

```
[UNION ALL|UNION]
```

```
SELECT campo1, campo2 FROM tabla2
```

● Ejemplo Simple:

```
SELECT legajo, fechaingreso from empleado1
```

```
UNION
```

```
SELECT leg, fing FROM empleado2
```



DML: Intersect

Este operador devuelve todas las filas en común de las tablas involucradas en la operación. También deben ser compatibles, es decir, igual grado e igual dominio.

```
SELECT campo1, campo2 FROM tabla1  
  
INTERSECT  
  
SELECT campo1, campo2 FROM tabla2
```

● Ejemplo Simple:

```
SELECT legajo, fechaingreso from empleado1  
  
INTERSECT  
  
SELECT leg, fing FROM empleado2
```


DML: Except



Este operador devuelve todas las filas que no están en el resto de las tablas de la operación. Simula ser una resta de las operaciones algebraicas. También deben ser compatibles, es decir, igual grado e igual dominio.

```
SELECT campo1, campo2 FROM tabla1
```

```
EXCEPT
```

```
SELECT campo1, campo2 FROM tabla2
```

● Ejemplo Simple:

```
SELECT legajo, fechaingreso from empleado1
```

```
EXCEPT
```

```
SELECT leg, fing FROM empleado2
```

DML: Join



Las juntas permiten unir tuplas de distintas relaciones para generar una nueva tupla.

- **Tipos de Joins:**

- Inner Join
- Left Join
- Right Join
- Cross Join
- Full Join

- **Sintaxis:**

```
TABLA1 T1 [INNER|LEFT|RIGHT|CROSS|FULL] JOIN TABLA2 T2  
ON T1.CAMPO1=T2.CAMPO2
```



DML: Inner Join

Este tipo de junta, solo devuelve una tupla cuando encuentra exactamente una coincidencia en la otra relación.

Empleado		Trabaja		Proyecto	
Legajo	NyA	Legajo	IDP	IDP	Desc
L1	Juan	L1	PR1	PR1	Migración
L2	Ana	L2	PR2	PR2	Analisis
L3	Lola	L3	PR1	PR3	Patch
L4	Pedro	L3	PR2		
L5	Martín				

```
SELECT e.legajo, e.NyA
from empleado e
      inner join trabaja t on e.legajo=t.legajo
```



Legajo	NyA
L1	Juan
L2	Ana
L3	Lola
L3	Lola



DML: Left Join

Este tipo de junta, solo devuelve una tupla cuando encuentra en la primer tabla (left table) del Join, independientemente que no exista en la segunda tabla.

Empleado		Trabaja		Proyecto	
Legajo	NyA	Legajo	IDP	IDP	Desc
L1	Juan	L1	PR1	PR1	Migración
L2	Ana	L2	PR2	PR2	Analisis
L3	Lola	L3	PR1	PR3	Patch
L4	Pedro	L3	PR2		
L5	Martín				

```
SELECT e.legajo, t.idp
from empleado e
      left join trabaja t on e.legajo=t.legajo
```



Legajo	IDP
L1	PR1
L2	PR2
L3	PR1
L3	PR2
L4	null
L5	null



DML: Right Join

Este tipo de junta, solo devuelve una tupla cuando encuentra en la segunda tabla (right table) del Join, independientemente que no exista en la primer tabla.

Empleado		Trabaja		Proyecto	
Legajo	NyA	Legajo	IDP	IDP	Desc
L1	Juan	L1	PR1	PR1	Migración
L2	Ana	L2	PR2	PR2	Analisis
L3	Lola	L3	PR1	PR3	Patch
L4	Pedro	L3	PR2		
L5	Martín				

```
SELECT e.legajo, t.idp
from trabaja t
      right join empleado e on e.legajo=t.legajo
```



Legajo	IDP
L1	PR1
L2	PR2
L3	PR1
L3	PR2
L4	null
L5	null



DML: Full Join

Este tipo de junta, solo devuelve una tupla cuando encuentra en la segunda tabla (right table) del Join o en la primer tabla (left table). Aquellos datos que no pueda completar porque no exista coincidencia en la tupla, se visualizará con null.

Empleado		Trabaja		Proyecto	
Legajo	NyA	Legajo	IDP	IDP	Desc
L1	Juan	L1	PR1	PR1	Migración
L2	Ana	L2	PR2	PR2	Analisis
L3	Lola	L3	PR1	PR3	Patch
L4	Pedro	L3	PR2		
L5	Martín				

```
SELECT e.nya, p.desc
from trabaja t
    full join empleado e on e.legajo=t.legajo
    full join proyecto p on t.IDP=p.IDP
```



NyA	Desc
Juan	Migracion
Ana	Analisis
Lola	Migracion
Lola	Analisis
Pedro	null
Martin	null
null	Patch



DML: Cross Join

Realiza un producto cartesiado con ambas tablas. En este caso el cross join no tiene condición de junta, ya que junta todas las tuplas.

Empleado

Legajo	NyA
L1	Juan
L2	Ana
L3	Lola
L4	Pedro
L5	Martín

Trabaja

Legajo	IDP
L1	PR1
L2	PR2
L3	PR1
L3	PR2

Proyecto

IDP	Desc
PR1	Migración
PR2	Análisis
PR3	Patch

```
SELECT e.legajo, p.idp
from empleado e
      cross join proyecto p
Where e.legajo in ('L1','L2')
```



Legajo	IDP
L1	PR1
L1	PR2
L1	PR3
L2	PR1
L2	PR2
L2	PR3



Revisión SQL I

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- SELECT ... FROM
- SELECT ... FROM ... WHERE ...
- ORDER BY
- DISTINCT
- TOP
- ALIAS
- ANY. SOME. ALL
- IN. EXISTS
- UNION. UNION ALL
- INTERSECT
- EXCEPT
- JOIN. INNER. LEFT. RIGHT. FULL. CROSS

Distintos tipos de Juntas (JOINS) y sus resultados.

Conjuntos Iniciales

Tabla 1 (t1)

num	name
1	a
2	b
3	c

Tabla 2 (t2)

num	value
1	xxx
3	yyy
5	zzz

Producto Cartesiano

```
SELECT * FROM t1 CROSS JOIN t2;
```

num	name	num	value
1	a	1	xxx
1	a	3	yyy
1	a	5	zzz
2	b	1	xxx
2	b	3	yyy
2	b	5	zzz
3	c	1	xxx
3	c	3	yyy
3	c	5	zzz

(9 rows)

Junta por campos relacionados

```
SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
```

o

```
SELECT * FROM t1 JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
3	c	3	yyy

(2 rows)

Junta por campos relacionados II (USING)

```
SELECT * FROM t1 INNER JOIN t2 USING (num);
```

o

```
SELECT * FROM t1 JOIN t2 USING (num);
```

num	name	value
1	a	xxx
3	c	yyy

(2 rows)

Junta por campos relacionados III (NATURAL)

```
SELECT * FROM t1 NATURAL INNER JOIN t2;
```

o

```
SELECT * FROM t1 NATURAL JOIN t2;
```

num	name	value
1	a	xxx
3	c	yyy

(2 rows)

Junta manteniendo todos los registros de la tabla izquierda

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
2	b		
3	c	3	yyy

(3 rows)

<- Tiene valores para t1 y no para t2

Junta manteniendo todos los registros de la tabla izquierda y usando el USING

```
SELECT * FROM t1 LEFT JOIN t2 USING (num);
```

num	name	value	
1	a	xxx	
2	b		<- Tiene valores para t1 y no para t2
3	c	yyy	

Junta manteniendo todos los registros de la tabla derecha

```
SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

num	name	num	value	
1	a	1	xxx	
3	c	3	yyy	<- Tiene valores para t2 y no para t1
		5	zzz	<- Tiene valores para t2 y no para t1

Junta completa. Todos los registros de la izquierda presentes. También los de la derecha

```
SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
```

num	name	num	value	
1	a	1	xxx	<- Hay correspondencia entre ambas tablas
2	b			<- Tiene valores para t1 y no para t2
3	c	3	yyy	<- Hay correspondencia entre ambas tablas
		5	zzz	<- Tiene valores para t2 y no para t1

(4 rows)

Diferencias en poner una condición en el WHERE o en el JOIN

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num
AND t2.value = 'xxx';
```

num	name	num	value	
1	a	1	xxx	
2	b			<- Como es un LEFT JOIN, solo aplicará el '='xxx' en las tuplas que ese campo tenga datos.
3	c			

(3 rows)

Ahora, si la condición está en el where

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num
```

```
WHERE t2.value = 'xxx';
```

```
num | name | num | value  
-----+-----+-----+-----
```

```
1 | a | 1 | xxx
```

**<- Aplica el where luego de hacer la junta
entonces elimina las tuplas sin valores**

```
(1 row)
```



Universidad Nacional de La Matanza
Catedra de Base de Datos

Clase Teórica de SQL – Parte 2

Lunes 26/10/2020



Funciones de agregación:

Se aplican sobre un conjunto de filas y devuelven un único valor para todas ellas.

Hay muchas, las que mas se utilizan son:

- SUM()
- MAX()
- MIN()
- AVG()
- COUNT()



SUM()

Calcula la suma de valores de una columna.

Ejemplo:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Suma de salarios de todos los empleados

```
SELECT SUM(salario)
FROM Empleado
```



SUM()

Calcula la suma de valores de una columna.

Ejemplo2:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Suma de salarios de todos los empleados de Sistemas

```
SELECT SUM(salario)
FROM Empleado e, Departamento d
WHERE e.cod_depto = d.cod_depto
AND d.descripcion = 'Sistemas'
```




MAX() Devuelve el mayor valor de una columna

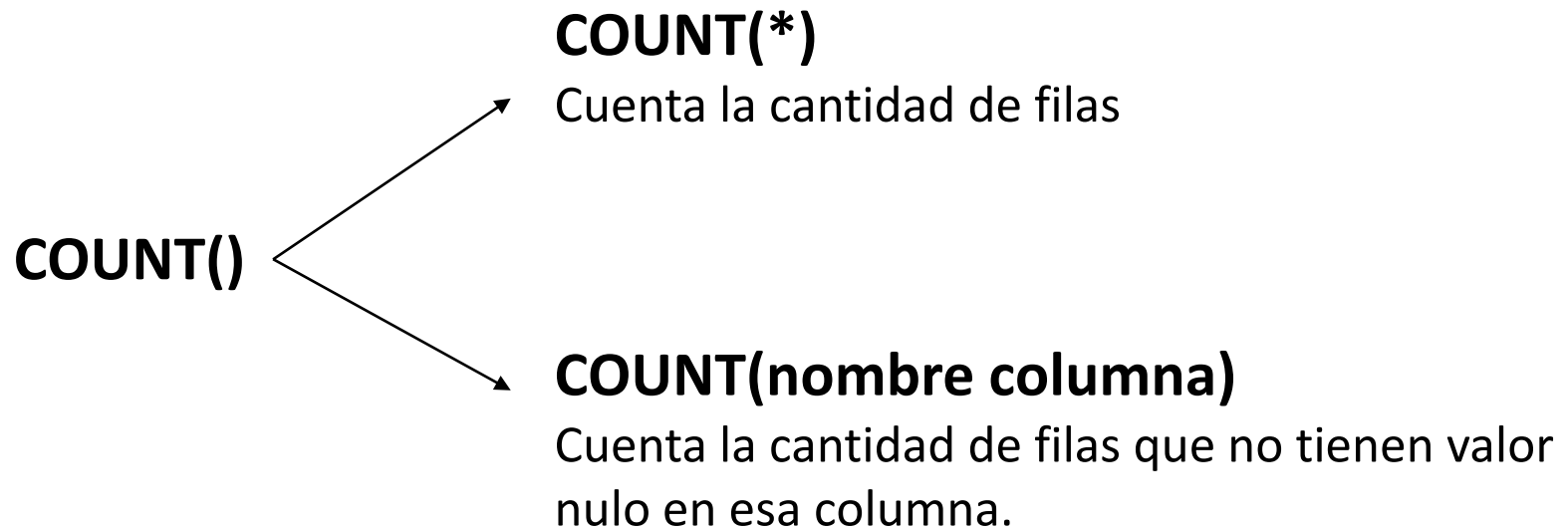
MIN () Devuelve el menor valor de una columna

Ejemplo:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Salario Mínimo y Máximo de los empleados de categoría 3

```
SELECT MIN(salario), MAX(salario)
FROM Empleado
WHERE categoria = 3
```





COUNT()

Ejemplo:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Cantidad de Departamentos

```
SELECT COUNT(*)  
FROM Departamento
```



COUNT()

Ejemplo2:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Cantidad total de Empleados y cuantos de ellos tienen teléfono

```
SELECT COUNT(*) cant_empleados,  
        COUNT(tel) cant_con_tel  
FROM Empleado
```



GROUP BY

Permite dividir el resultado de una consulta en grupos, según el valor de uno o mas atributos.

Ejemplo:

EMPLEADO (legajo, nom, ape, salario, categoría, tel, cod_depto)
DEPARTAMENTO (cod_depto, descripcion)

Contar la cantidad de empleados de cada categoría

```
SELECT categoria, COUNT(*)  
FROM Empleado  
GROUP BY categoria
```



GROUP BY

```
SELECT categoria, COUNT(*)  
FROM Empleado  
GROUP BY categoria
```

Las columnas normales (no agrupadas)
siempre **DEBEN** estar en el GROUP BY



GROUP BY

```
SELECT COUNT(*)  
FROM Empleado  
GROUP BY categoria
```

Pero en el GROUP BY es posible colocar
columnas que no estén en el SELECT



HAVING

Permite especificar condiciones que se aplicarán luego del GROUP BY.

Es como el WHERE pero para colocar condiciones luego de haber agrupado.

Las condiciones del WHERE se aplican antes de hacer el agrupamiento.



HAVING

Ejemplo:

Listar los Departamentos que tengan 5 o más empleados de categoría 2.

```
SELECT cod_depto, count(*) cant
FROM Empleado
WHERE categoria = 2
GROUP BY cod_depto
HAVING count(*) >=5
```

Esta condición se
resuelve **antes** del
GROUP BY

Esta condición se
resuelve **después** del
GROUP BY

NO se puede usar el Alias de la columna
HAVING cant >= 5



ORDER BY

Permite ordenar el resultado de una consulta por una o mas columnas.

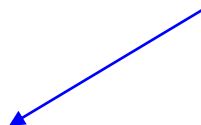
Se debe colocar al **final** de la consulta.

SELECT

FROM

.....

Por defecto ordena en
forma Ascendente



ORDER BY columna1, columna2 **DESC**,, columnaN **ASC**



ORDER BY

- Se puede ordenar por una columna que no se encuentre en el SELECT (siempre y cuando la consulta no esté agrupada)

R (a, b, c)

```
SELECT a, b  
FROM R  
ORDER BY c
```



ORDER BY

- Se puede usar el alias de una columna en el ORDER BY

R (a, b, c)

```
SELECT a, b as e  
FROM R  
ORDER BY e
```



ORDER BY

- Se puede indicar el número de columna en el ORDER BY

R (a, b, c)

```
SELECT a, c  
FROM R  
ORDER BY 2, 1
```



ORDER BY

- Se puede ordenar por una columna derivada (proveniente de un cálculo)

R (a, b, c)

```
SELECT a, b-c  
FROM R  
ORDER BY b-c
```



COCIENTE

En SQL no hay un operador que permita resolver directamente el cociente, tal como existe en A.R.

Existen varias formas de resolver el Cociente en SQL utilizando otros operadores. La forma mas recomendada es usando tres consultas anidadas vinculadas entre si mediante NOT EXISTS.



COCIENTE

Ejemplo:

ALUMNO (legajo, nom, ape, email, telefono)

MATERIA (cod_mat, nombre, año_de_la_carrera)

CURSA (legajo, cod_mat)

Listar los legajos y apellidos de los Alumnos que cursan **todas** las materias de cuarto año.



COCIENTE

Solución:

Listar los legajos y apellidos de los Alumnos que cursan **todas** las materias de cuarto año.

```
SELECT a.legajo, a.apellido
FROM Alumno a
WHERE NOT EXISTS ( SELECT *
                    FROM Materia m
                    WHERE año_de_la_carrera = 4
                    AND NOT EXISTS ( SELECT *
                                    FROM Cursa c
                                    WHERE c.legajo = a.legajo
                                    AND c.cod_mat = m.cod_mat ))
```



COCIENTE

Solución:

Listar los legajos y apellidos de los Alumnos que cursan **todas** las materias de cuarto año.

```
SELECT  a.legajo, a.apellido
FROM    Alumno a
WHERE   NOT EXISTS ( SELECT *
                     FROM  Materia m
                     WHERE  año_de_la_carrera = 4
                     AND NOT EXISTS ( SELECT *
                                     FROM  Cursa c
                                     WHERE c.legajo = a.legajo
                                     AND c.cod_mat = m.cod_mat ))
```

Alumnos tales que
NO EXISTE una Materia de 4to año
que NO cursen



COCIENTE

Hay otras dos formas de resolverlo:

1) Contar cuantas materias hay en 4to año y luego encontrar a los alumnos que cursan esa misma cantidad de materias de 4to año.

2) Resolverlo el cociente con operadores básico de AR y luego traducir esa expresión a SQL.

Todas las operaciones básicas que vimos en AR se pueden traducir al SQL.

$\text{Mat4to} \leftarrow \sigma_{\text{año_de_la_carrera}=4} (\text{Materia})$

$\text{CursanTodas} \leftarrow \pi_{\text{legajo}}(\text{Cursa}) - \pi_{\text{legajo}} ((\pi_{\text{legajo}}(\text{Cursa}) \times \text{Mat4to}) - \text{Cursa})$

$\text{Resultado} \leftarrow \pi_{\text{legajo}, \text{apellido}} (\text{CursanTodas} \mid \times \mid \text{Alumno})$



COCIENTE

Hay otras dos formas de resolverlo:

1) Contar cuantas materias hay en 4to año y luego encontrar a los alumnos que cursan esa misma cantidad de materias de 4to año.

2) Resolverlo el cociente con operadores básico de AR y luego traducir esa expresión a SQL.

Todas las operaciones básicas que vimos en AR se pueden traducir al SQL.

$\text{Mat4to} \leftarrow \sigma_{\text{año_de_la_carrera}=4} (\text{Materia})$

$\text{CursanTodas} \leftarrow \pi_{\text{legajo}(\text{Alumno})} - \pi_{\text{legajo} ((\pi_{\text{legajo}(\text{Alumno}) \times \text{Mat4to}) - \text{Cursa})$

$\text{Resultado} \leftarrow \pi_{\text{legajo, apellido} (\text{CursanTodas} \mid \times \mid \text{Alumno})$



Vistas

Las vistas son Consultas SQL almacenadas en la base de datos con un nombre.

A diferencia de las Tablas, las vistas no contienen información, sino que simplemente devuelven el resultado de la consulta la cual se ejecuta cada vez que la vista es invocada.

Las vistas se utilizan principalmente para:

1. Almacenar consultas que utilizamos con frecuencia
2. Para restringir permisos sobre ciertos datos (por ejemplo: tabla de empleados sin la columna salario)



Creación de una vista

CREATE VIEW nombre_vista [(nombre de columnas)] AS

Consulta SQL



Creación de una vista

Ejemplo:

EMPLEADO (legajo, nom, ape, fecha_ingreso, salario, cod_depto)
DEPARTAMENTO (cod_depto, descripcion, legajo_gerente)

```
CREATE VIEW Depto AS  
  SELECT descripción, count(*) as cant_empleados, g.ape as gerente  
  FROM Empleado e, Departamento d, Empleado g  
  WHERE e.cod_depto=d.cod_depto  
  AND d.legajo_gerente=g.legajo  
  GROUP BY descripción, g.ape
```



Creación de una vista

Luego podemos consultar la vista como si fuese una tabla:

```
SELECT *  
FROM Depto  
WHERE cant_empleados > 10
```




Eliminación de una vista

`DROP VIEW nombre_vista`

Ejemplo:

`DROP VIEW Depto`

Si hay otras vistas que la usan, deja eliminarla igual, pero luego las otras vistas darán error cuando se quieran utilizar.



Actualización de los datos de una vista

Si un usuario quiere modificar, insertar o eliminar un dato de una tabla, lo mejor es que lo haga sobre la misma tabla.

Sin embargo, en algunos casos es posible hacerlo sobre la vista y el cambio se aplica automáticamente sobre la tabla a la que apuntan.

No todas las vistas son actualizables, es decir, permiten que se modifiquen los datos. Eso depende de cada motor de base de datos y versión del mismo.

En términos generales, podemos decir que si la vista es sobre una sola tabla y contiene su clave, casi seguro que permitirá que se modifiquen los datos.

Por el contrario, si la vista es una consulta agrupada, entonces casi seguro que no permitirá modificar sus datos.



Sentencias DML (Data Manipulation Language)

Permiten modificar los datos de las tablas.
Se pueden deshacer (admiten Roll Back)

- INSERT
- UPDATE
- DELETE

Sentencias DDL (Data Definition Language)

Permiten modificar la estructura de las tablas.
NO se pueden deshacer (NO admiten Roll Back)

- TRUNCATE
- DROP
- ALTER
- CREATE



INSERT

Permite insertar una o varias filas en una tabla.

Se puede utilizar de dos formas:

Forma 1 (inserta de a una fila):

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

Ejemplo:

```
INSERT INTO Empleado (legajo, nombre, apellido)  
VALUES (10,'Jorge', 'Varela')
```



INSERT

Permite insertar una o varias filas en una tabla.

Se puede utilizar de dos formas:


Forma 1 (inserta de a una fila):

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

Si se omite, se supone que son todas las columnas de la tabla

Ejemplo:

```
INSERT INTO Empleado (legajo, nombre, apellido)  
VALUES (10,'Jorge', 'Varela')
```





INSERT

Forma 2 (inserta multiples filas):
Inserta en una tabla el resultado de una consulta.

```
INSERT [INTO] nombre_tabla [(lista de columnas)]  
  SELECT lista de columnas  
  FROM nombre_tabla
```

Ejemplo:
INSERT INTO Empleado_backup
SELECT *
FROM Empleado



DELETE


Permite eliminar una o varias filas en una tabla.

```
DELETE [FROM] nombre_tabla  
[WHERE condición]
```

Ejemplo:

```
DELETE Empleado  
WHERE cod_depto IN (5,4)
```

Puede dar error si se intenta
eliminar una fila referenciada por
una Foreign Key de otra tabla.





UPDATE

Permite modificar los valores de una o varias columnas de una o varias filas de una tabla.

```
UPDATE nombre_tabla  
SET columna1 = <nuevo valor>,  
    columna2 = <nuevo valor>,  
    ...  
    columnaN = <nuevo valor>  
[WHERE condición]
```

Ejemplo:

```
UPDATE Empleado  
SET    telefono = '555-1234', salario = 50000  
WHERE  legajo = 23
```




TRUNCATE

Permite eliminar TODAS las filas de una tabla.

```
TRUNCATE TABLE nombre_table
```

Ejemplo:

```
TRUNCATE TABLE Empleado
```



TRUNCATE

Es similar al DELETE, ya que ambas sentencias eliminan filas, pero tiene 3 diferencias:

1. El TRUNCATE elimina TODAS las filas de la tabla, no puede eliminar solo algunas
2. No tiene posibilidad de deshacerse, ya que no permite Roll Back.
Igualmente esto está cambiando y algunas versiones nuevas de algunos motores están comenzando a permitirlo.
3. Libera el espacio físico que ocupan las filas. El delete hace un borrado lógico y no libera el espacio.
Si vemos cuanto espacio ocupa una tabla, luego hacemos un DELETE y eliminamos la mitad de sus filas y por ultimo volvemos a ver cuanto espacio ocupa la tabla, veremos que sigue ocupando lo mismo. Ya que no libera el espacio de las filas eliminadas. El TRUNCATE si lo hace.



SQL

Parte III

Objetos



- *Database*
- *Table*
- *View*
- *Stored Procedure*
- *Function*
- *Trigger*



Stored Procedure

Es un conjunto de sentencias SQL que pueden ejecutarse, con tan solo invocar su nombre. Los Stored procedures son similares a los procedimientos que se generan en otros lenguajes de programación. En ellos se podrá:

- *Incluir 1 ó n sentencias SQL, ya sea de DML ó DDL.*
- *Aceptar parámetros de entrada y podrá emitir parámetros de salida.*
- *Se podrá llamar a otro procedure.*
- *Podrá devolver el estado de ejecución del procedure en su nombre.*
- *Utilizar estrategias de programación, tales como variables, ciclos, condicionales.*



Stored Procedure: Crear

- **Sintaxis Simple:**

```
CREATE [OR ALTER] { PROC | PROCEDURE } NombreProcedure
    ([@parametro tipodedatos [ = default ] [ OUT | OUTPUT ]] [,...n ])
[ WITH [RECOMPILE|ENCRYPTION] ]
AS
[BEGIN]
    sql_statement
[END]
```

- **Ejemplo Simple:**

```
CREATE PROCEDURE p_borrarclientes
AS
    DELETE FROM CLIENTE
```

```
CREATE PROCEDURE p_borrarclientesID (@IDDESDE int, @IDHASTA int)
AS
BEGIN
    DELETE FROM CLIENTE where idcliente between @IDDESDE and @IDHASTA
END
```



Stored Procedure: Crear

● Ejemplo Simple 2:

```
CREATE PROCEDURE p_borrarclientesID (@IDDESDE int, @IDHASTA int)
AS
BEGIN

    DELETE FROM VENTA where idcliente between @IDDESDE and @IDHASTA

    DELETE FROM CLIENTE where idcliente between @IDDESDE and @IDHASTA

END

CREATE PROCEDURE p_listarclientesID (@IDDESDE int, @IDHASTA int)
AS
BEGIN

    SELECT idcliente, nombre + ' ' + apellido, fechaalta, cuit
    FROM CLIENTE
    where idcliente between @IDDESDE and @IDHASTA
    Order by idcliente

END
```



Stored Procedure: Borrar

- **Sintaxis Simple:**

```
DROP {PROC|PROCEDURE} [IF EXISTS] NombreProcedure
```

- **Ejemplo Simple:**

```
DROP PROCEDURE p_borrarclientes
```

```
DROP PROCEDURE IF EXISTS p_borrarclientesID
```


Stored Procedure: Cambiar



- **Sintaxis Simple:**

```
ALTER { PROC | PROCEDURE } NombreProcedure
    ([@parameter tipodedatos [ = default ] [ OUT | OUTPUT ]] [,...n ])
[ WITH [RECOMPILE|ENCRYPTION] ]
AS
[BEGIN]
    sql_statement
[END]
```

- **Ejemplo Simple:**

```
ALTER PROCEDURE p_borrarclientes
AS
    DELETE FROM CLIENTE
```

```
ALTER PROCEDURE p_borrarclientesID (@IDDESDE int, @IDHASTA int)
AS
BEGIN
    DELETE FROM CLIENTE where idcliente between @IDDESDE and @IDHASTA
END
```



Stored Procedure: Ejecutar

- **Sintaxis Simple:**

```
{EXEC | EXECUTE} {PROC | PROCEDURE} NombreProcedure [@Param1 [OUTPUT], ...n]
```

- **Ejemplo Simple:**

```
EXEC PROCEDURE p_borrarclientes
```

```
EXECUTE PROCEDURE p_borrarclientesID @pcliid1,@pcliid2
```

Variables



- **Declaración: Sintaxis**

```
DECLARE @nombrevariable [AS] tipodedatos [=valordefecto] [,...n]
```

- **Declaración: Ejemplo**

```
DECLARE @vapellido varchar(100)
```

```
DECLARE @id int, @vnombre varchar(100)
```

- **Asignación: Sintaxis**

```
SET @nombrevariable = <valor>
```

- **Asignación: Ejemplo**

```
SET @id = 1
```

```
SET @vapellido='Perez'
```



Estructuras: while

- **While: Sintaxis**

```
WHILE <condicion>
[BEGIN]
    <SeteneciasSQL>

    [BREAK | CONTINUE]
[END]
```

- **While: Ejemplo**

```
DECLARE @cant int
SET @cant = (select count(*) from producto)

While @cant>0
Begin

    UPDATE Producto set precio=precio*1.10 where id=@cant

    Set @cant=@cant-1

End
```



Estructuras: Condicionales

- **IF: Sintaxis**

```
IF <condicion>
[BEGIN]
    <SeteneciasSQL>
[END]
[ELSE]
[BEGIN]
    <SeteneciasSQL>
[END]
```

- **IF: Ejemplo**

```
DECLARE @sueldo numeric(10,2)
SET @Sueldo = (Select sueldo from empleado where legajo=@legajo)

If @Sueldo > 20000
    UPDATE empleado SET sueldo=sueldo*1.20 where legajo=@legajo
Else
    UPDATE empleado SET sueldo=sueldo*1.30 where legajo=@legajo
```

Stored Procedure: Ejemplos



● **Ejemplo:**

```
CREATE PROCEDURE p_nuevocliente (@razonsocial varchar(100), @cuit bigint,  
@domicilio varchar(200), @mail varchar(200))  
AS  
BEGIN  
  
    Declare @id int  
    Set @id = (Select max(idcliente) from cliente)  
  
    If @razonsocial is null or @razonsocial='' or @cuit=0  
        Select 'No se podrá insertar el cliente. Verificar datos'  
    Else  
    Begin  
        If len(@mail)=0  
            Set @mail='NO ASIGNADO'  
  
        INSERT INTO Cliente (id,razonsocial,cuit,domicilio,mail)  
        VALUES(@id+1, @razonsocial, @cuit, @domilicio, @mail)  
  
    end  
  
END
```

Stored Procedure: Ejemplos



● **Ejemplo:**

```
CREATE PROCEDURE p_nuevocliente (@razonsocial varchar(100), @cuit bigint,  
@domicilio varchar(200), @mail varchar(200), @newid INT output)  
AS  
BEGIN  
    Set @newid = (Select max(idcliente) from cliente)+1  
  
    INSERT INTO Cliente (id,razonsocial,cuit,domicilio,mail)  
    VALUES(@newid, @razonsocial, @cuit, @domilicio, @mail)  
END
```

● **Ejecución:**

```
declare @rz varchar(200) = 'Perez S.A.'  
declare @cuit bigint = 30258761234  
declare @dom varchar(200) = 'Salta 123'  
declare @new int  
  
EXEC p_nuevocliente @rz,@cuit,@dom,@new OUTPUT  
  
print @new
```

Objetos



- *Database*
- *Table*
- *View*
- *Stored Procedure*
- *Function*
- *Trigger*

Function



Es un conjunto de sentencias SQL que pueden ejecutarse, con tan solo invocar su nombre y en su nombre devolverá una respuesta. Las funciones son similares a las funciones que se generan en otros lenguajes de programación. En ellas se podrá:

- *Incluir 1 ó n sentencias SQL, ya sea de DML ó DDL.*
- *Aceptar parámetros de entrada y podrá emitir parámetros de salida.*
- *Se podrá invocar a otra función.*
- *Utilizar estrategias de programación, tales como variables, ciclos, condicionales.*
- *Devolver un valor en su nombre. El valor que devuelve será un valor escalar o también podrá devolver un valor de tipo table.*



Scalar Function: Crear

- **Sintaxis Simple:**

```
CREATE [ OR ALTER ] FUNCTION NombreFuncion
([@parametro tipodedatos [ = default ]] [ ,...n ])
RETURNS return_tipodedatos
[AS]
BEGIN
    Sentencias SQL

    RETURN valor
END
```

- **Ejemplo Simple:**

```
CREATE FUNCTION f_ProximoCliente ()
RETURNS int
AS
BEGIN
    declare @ult int
    Set @ult=(select coalesce(max(idcliente),1) from Cliente)

    return @ult + 1
END
```



Table Function: Crear

● Sintaxis Simple:

```
CREATE [ OR ALTER ] FUNCTION NombreFuncion
([@parametro tipodedatos [ = default ]] [ ,...n ])
RETURNS table
[AS]
    RETURN (sentencia_Select)
```

```
CREATE [ OR ALTER ] FUNCTION NombreFuncion
([@parametro tipodedatos [ = default ]] [ ,...n ])
RETURNS @varable table (<definición>)
[AS]
BEGIN
    <sentencia_SQL>
    INSERT INTO @varable ...
    RETURN
END
```

● Ejemplo Simple:

```
CREATE FUNCTION f_Clientes ()
RETURNS table
AS
    return (select idcliente,razonsocial from Cliente)
```



Table Function: Invocar

- **Ejemplo Simple:**

Select * from f_clientes()

```
CREATE FUNCTION f_Clientes ()  
RETURNS table  
AS  
    return (select idcliente,razonsocial from Cliente)
```

Select f_proximocliente()

Insert into cliente values (f_proximocliente(),'Juan')

```
CREATE FUNCTION f_ProximoCliente ()  
  
RETURNS int  
AS  
BEGIN  
    declare @ult int  
    Set @ult=(select coalesce(max(idcliente),1) from Cliente)  
  
    return @ult + 1  
END
```



Table Function: Borrar

- **Sintaxis Simple:**

```
DROP FUNCTION [ IF EXISTS ] NombreFunction
```

- **Ejemplo Simple:**

```
DROP FUNCTION f_Clientes
```

```
DROP FUNCTION IF EXISTS f_Clientes
```



Function: Cambiar

● Sintaxis Simple:

```
ALTER FUNCTION NombreFuncion
([@parametro tipodedatos [= default ]] [ ,...n ])
RETURNS table
[AS]
    RETURN (sentencia_Select)
```

```
ALTER FUNCTION NombreFuncion
([@parametro tipodedatos [= default ]] [ ,...n ])
RETURNS @varable table (<definición>)
[AS]
BEGIN
    <sentencia_SQL>
    INSERT INTO @varable ...
    RETURN
END
```

● Ejemplo Simple:

```
ALTER FUNCTION f_Clientes ()
RETURNS table
AS
    return (select idcliente as idX,razonsocial from Cliente)
...
INVOCAR:  SELECT * FROM F_CLIENTES() WHERE idX=10
```

Objetos



- *Database*
- *Table*
- *View*
- *Stored Procedure*
- *Function*
- *Trigger*

Trigger



Es un conjunto de sentencias SQL que sólo se disparan cuando se produce un evento. Los eventos de DML pueden ser de Update, Delete ó Insert. En ellos se podrá:

- *Incluir 1 ó n sentencias SQL, ya sea de DML ó DDL.*
- *Utilizar estrategias de programación, tales como variables, ciclos, condicionales.*
- *Utilizar el mismo trigger para distintos eventos de DML.*
- *Utilizar sólo para una única tabla o vista.*
- *Utilizar las tablas temporales deleted, inserted para verificar lo que está tratando de cambiarse en el trigger.*



Trigger: Crear

- **Sintaxis Simple:**

```
CREATE [ OR ALTER ] TRIGGER NombreTrigger
ON { table|view }
{ FOR | AFTER | INSTEAD OF}
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
    Sentencias_SQL
```

- **Ejemplo Simple:**

```
CREATE TRIGGER tg_borrarclientes ON cliente INSTEAD OF DELETE
AS
    If not exists(Select 1 from factura f
                  Where f.nrocliente in (select nro from deleted))
        delete from cliente
        where nro in(select nro from deleted)

CREATE TRIGGER tg_clientes ON cliente AFTER INSERT,UPDATE
AS
    UPDATE cliente
    Set fechamodificacion=getdate()
    Where nro in (Select nro from inserted)
```



Trigger: Borrar

- **Sintaxis Simple:**

```
DROP TRIGGER [IF EXISTS] NombreTrigger
```

- **Ejemplo Simple:**

```
DROP TRIGGER tg_clientes
```

```
DROP TRIGGER IF EXISTS tg_clientes
```



Trigger: Cambiar

● Sintaxis Simple:

```
ALTER TRIGGER NombreTrigger
ON { table|view }
{ FOR | AFTER | INSTEAD OF } {[INSERT] [,] [UPDATE] [,] [DELETE]}
AS
BEGIN
    Sentencias_SQL
END
```

● Ejemplo Simple:

```
ALTER TRIGGER tg_borrarclientes ON cliente INSTEAD OF DELETE
AS
BEGIN
    If not exists(Select 1 from factura f
                  Where f.nrocliente in (select nro from deleted))
        delete from cliente
        where nro in(select nro from deleted)
END
```

```
ALTER TRIGGER tg_clientes ON cliente AFTER INSERT,UPDATE
AS
BEGIN
    UPDATE cliente
    Set fechamodificacion=getdate()
    Where nro in (Select nro from inserted)
END
```

Trigger: Habilitar / Deshabilitar



- **Sintaxis Simple:**

```
ENABLE TRIGGER [NombreTrigger|ALL] ON [table|view]
```

```
DISABLE TRIGGER [NombreTrigger|ALL] ON [table|view]
```

- **Ejemplo Simple:**

```
ENABLE TRIGGER tg_clientes on Cliente
```

```
DISABLE TRIGGER ALL on Cliente
```



Revisión SQL III

- CREATE PROCEDURE
- DROP PROCEDURE
- ALTER PROCEDURE

- CREATE FUNCTION
- DROP FUNCTION
- ALTER FUNCTION

- CREATE TRIGGER
- DROP TRIGGER
- ALTER TRIGGER
- ENABLE TRIGGER
- DISABLE TRIGGER

Stored Procedures, Triggers y Funciones

Stored Procedures (Procedimientos Almacenados)

Un procedimiento almacenado es un programa (o procedimiento) el cual es almacenado físicamente en una base de datos.

La **ventaja** de los procedimientos almacenados es que se ejecutan directamente en el motor de bases de datos, directo en el mismo servidor donde se encuentra ubicada la base de datos. Como tal, poseen acceso directo a los datos que necesitan manipular y sólo necesitan enviar sus resultados de regreso al usuario, deshaciéndose de la sobrecarga resultante de comunicar grandes cantidades de datos salientes y entrantes.

Generalmente se utilizan para procesos complejos o grandes que podrían requerir la 'ejecución' de varias consultas SQL, tales como la manipulación de un 'dataset' enorme para producir un resultado resumido.

Los SP tienen un nombre y se ejecutan a petición del usuario/cliente.

Creación de un SP en SQL Server

```
CREATE PROCEDURE procedure_name [ @parametro1 tipo_de_dato,  
                                  @parametro2 tipo_de_dato,  
                                  ...  
                                  @parametroN tipo_de_dato ]
```

AS

```
[  
  DECLARE @variable1 tipo_de_dato;  
  DECLARE @variableN tipo_de_dato;  
]
```

[BEGIN]

Sentencias de Transact SQL

[END;]

Ejecución de un SP

```
EXECUTE procedure_name [parametros]
```

Desventajas:

- La lógica de la aplicación termina distribuida parte en la base de datos y parte en el código de la aplicación.
- Aumenta la dependencia del repositorio de datos (El lenguaje de los SP suele ser bastante diferente entre los distintos motores de BD, por ej., Transact-SQL para SQL Server y PL/SQL para Oracle.) Si en algún momento, se decide cambiar de Base de Datos, esa migración será más compleja.

Ventajas:

- Reutilización de código (distintos sistemas pueden utilizar los mismos SP). Por ejemplo, un mismo SP se puede disparar desde una página web, desde una aplicación Android o desde una clásica aplicación cliente-servidor.
- Mayor rendimiento (los datos no viajan desde la BD hasta la Aplicación para luego hacer cálculos, sino que todo se calcula dentro de la base y solo se entrega el resultado).
- Tráfico de Red: Pueden reducir el tráfico de la red, debido a que se trabaja sobre el motor (en el servidor), y si una operación incluye hacer un trabajo de lectura primero y en base a eso realizar algunas operaciones, esos datos que se obtienen no viajan por la red.

Ejemplo:

Dada la siguiente tabla:

Tabla DIRECTORIO

ID_DIR	NOMBRE_DIR	FECHA_CREACION	ID_DIR_PADRE
1	\	01/01/2015	
10	Programas\	05/01/2015	1
11	Programa1\	05/01/2015	10
12	Config\	05/01/2015	11
13	Data\	05/01/2015	11
14	Programa2\	06/01/2015	10
15	Log\	07/01/2015	14
20	Fotos\	02/01/2016	1
21	Viaje1\	15/01/2016	20
22	Viaje2\	10/05/2016	20
23	Viaje3\	03/06/2016	20
30	Documentos\	08/01/2015	1
31	UNLaM\	10/04/2016	30
32	Base_de_Datos\	12/04/2016	31
33	Fisica_1\	20/04/2015	31

El siguiente SP recibe como parámetro el ID de un directorio y devuelve el Path completo de su ubicación. Como la cantidad de niveles que puede tener ese directorio es variable, esto no se puede resolver con una simple consulta SQL (salvo que se fije una cantidad máxima de niveles).

Este SP posee una iteración que recorre todos los niveles y arma el Path Completo concatenando los nombres de los directorios donde se encuentra ubicado.

```
CREATE PROCEDURE sp_ListaDirectorios (@pDir int) AS
```

```
DECLARE @Padre int;
```

```
DECLARE @pPathCompleto varchar(200);
```

```
BEGIN
```

```
--Primero toma el nombre del Directorio recibido por parámetro
```

```
SET @pPathCompleto = (SELECT nombre_dir
                      FROM Directorio
                      WHERE id_dir = @pDir);
```

```
--Se fija si tiene algún directorio Padre
```

```
SET @Padre = (SELECT isnull(id_dir_padre,0)
              FROM Directorio
              WHERE id_dir = @pDir);
```

```
--Mientras exista algún Padre, va concatenando los nombres y armando el Path.
```

```
WHILE (@Padre <> 0) begin
```

```
--Agrega el nombre del Padre
```

```
SET @pPathCompleto = (SELECT nombre_dir + @pPathCompleto
                      FROM Directorio
                      WHERE id_dir = @Padre);
```

```
--Se fija si tiene otro directorio Padre
```

```
SET @Padre = (SELECT isnull(id_dir_padre,0)
              FROM Directorio
              WHERE id_dir = @Padre);
```

```
END;
```

```
--Finalmente, devuelve el valor
```

```
SELECT @pPathCompleto as RESULTADO;
END;
```

Una vez creado, el Stored Procedure puede ejecutarse de la siguiente manera:

```
EXECUTE sp_ListaDirectorios 33
```

Resultado:

\Documentos\UNLaM\Fisica_1\

Triggers (Disparadores o Desencadenadores)

Es el mismo concepto de los SP pero la diferencia es que los **Triggers se ejecutan por algún evento de la base de datos** (por ejemplo, cuando se inserta una fila en una tabla).

Se utilizan comúnmente para hacer validaciones de datos.

Creación de un SP en SQL Server

```
CREATE TRIGGER trigger_name ON objeto FOR/AFTER/INSTEAD OF tipo_evento
AS
[
DECLARE @variable1 tipo_de_dato;
DECLARE @variableN tipo_de_dato;
]
[BEGIN]
```

Sentencias de Transact SQL

[END;]

Objeto: Tabla / Vista / Database / ALL SERVER

Tipo de Evento: INSERT / UPDATE / DELETE / CREATE / DROP / ALTER / etc.

FOR / AFTER: **El trigger se ejecuta después de que el evento se ejecutó correctamente** en forma completa.

Se pueden definir varios Triggers de este tipo para un mismo objeto y un mismo evento.

INSTEAD OF: **El trigger se ejecuta en vez del evento disparador (antes), por lo que se puede suplantar el evento original por otra acción.**

Se puede definir un único Trigger del tipo Instead Of un mismo objeto y un mismo evento.

Tablas INSERTED y DELETED

Las tablas INSERTED y DELETED se pueden utilizar dentro del Trigger y contienen los registros insertados o eliminados. Son tablas virtuales y tienen la misma estructura (mismas columnas) que la tabla sobre la cual se define el Trigger.

No existe una tabla UPDATED, cuando se hace un UPDATE, la fila con el valor viejo queda en la tabla DELETED y la fila con el valor nuevo queda en la tabla INSERTED.

Ejemplo 1:

El siguiente Trigger incrementa en 1 la cantidad de ventas del vendedor que corresponda, cada vez que se hace una Venta y se inserta un registro en la tabla VENTAS.

```
CREATE TRIGGER ActualizaVentasVendedores
ON Ventas FOR INSERT
AS
UPDATE Vendedores
SET cant_ventas = cant_ventas + 1
WHERE id_vendedor IN (SELECT i.id_vendedor FROM inserted i)
```

Ejemplo 2:

Cada vez que se elimina un Producto, se guarda una copia de la fila eliminada en la tabla Producto_Eliminado que tiene exactamente la misma estructura que la tabla Producto.

```
CREATE TRIGGER t_prod_eliminado
ON Producto FOR delete
AS
INSERT INTO Producto_Eliminado
SELECT * FROM deleted
```

Ejemplo 3:

Cada vez que se intenta eliminar un Cliente, se acciona un Trigger antes que cancela la eliminación de la fila y simplemente hace una marca de borrado lógico en la fila y guarda también la fecha del borrado.

```
CREATE TRIGGER t_cli_eliminado
ON Cliente INSTEAD OF delete
AS
UPDATE Cliente
SET eliminado=1, fecha_eliminado=getdate()
WHERE cod_cli IN (SELECT cod_cli FROM deleted)
```


Ejemplo 4:

Cada vez que se modifica la tabla Cliente, se guarda la fila Vieja (sin el cambio) y la fila Nueva (con el cambio) en una tabla de Log.

```
CREATE TRIGGER Cliente_Update_Log
ON Cliente AFTER UPDATE
AS
BEGIN
    INSERT INTO Cliente_Log
    SELECT getdate(), 'Viejo', d.*
    FROM DELETED d;

    INSERT INTO Cliente_Log
    SELECT getdate(), 'Nuevo', i.*
    FROM INSERTED i;
END;
```

Ejemplo 5:

Dadas las siguientes tablas:

Empleado (legajo, nombre, apellido)

Asignacion (idCargo, legajo, fecha_ini, fecha_fin)

Realice un trigger que solo permita borrar un empleado si no tiene ninguna Asignacion vigente (fecha_fin nula o mayor a la fecha actual), caso contrario ignore la instrucción.

```
CREATE TRIGGER Ejercicio ON Empleado INSTEAD OF DELETE AS
DELETE Empleado
WHERE Empleado.legajo IN (SELECT legajo FROM deleted)
AND NOT EXISTS
(
    SELECT *
    FROM Asignacion a
    WHERE a.legajo = Empleado.legajo
    AND (a.fecha_fin IS NULL OR fecha_fin >= getdate() )
);
```

Funciones

Es el mismo concepto de los SP pero con la diferencia que las funciones siempre deben retornar un valor como resultado.

Luego, se utilizan comúnmente en las sentencias SELECT.

En SQL Server existen 2 tipos de funciones:

- Funciones escalares (devuelven como resultado un valor único)
- Funciones con valores de tabla (devuelven como resultado una tabla)

Las que más se usan son las funciones escalares.

Creación de una Función escalar en SQL Server

```
CREATE FUNCTION function_name [ @parametro1 tipo_de_dato,
                                @parametro2 tipo_de_dato,
                                ...
                                @parametroN tipo_de_dato ]
RETURNS tipo_de_dato
AS
[
    DECLARE @variable1 tipo_de_dato;
    DECLARE @variableN tipo_de_dato;
]
[BEGIN]

    Sentencias de Transact SQL

    RETURN valor_a_retornar

[END;]
```

Ejemplo1: Función escalar

```

CREATE FUNCTION NombreMes (@nro_mes int)
RETURNS varchar(20)
AS
BEGIN
    DECLARE @nombre_mes varchar(20)

    SET @nombre_mes = (
        CASE @nro_mes
            WHEN 1 THEN 'Enero'
            WHEN 2 THEN 'Febrero'
            ...
            WHEN 12 THEN 'Diciembre'
            ELSE 'Desconocido'
        END
    );

    RETURN (@nombre_mes);
END

```

Ejecución de la función creada
 SELECT legajo, NombreMes(month(fecha_nacimiento)) Mes_Cumpleaños
 FROM Empleado

Funciones con valor de Tabla

Dentro de este tipo de funciones existen 2 subtipos:

- Funciones con valores de tabla de varias instrucciones
- Funciones con valores de tabla en línea

Ambas retornan una tabla como resultado, pero la diferencia es que en el primer tipo se define una tabla y se carga con datos para finalmente retornarla como resultado. Mientras que el segundo caso es más sencillo y solo devuelve el resultado de una consulta.

Veamos un ejemplo de cada una:

Ejemplo 2: Función con valores de tabla de varias instrucciones
 Recibe como parámetro un país y devuelve todos los clientes de ese país.

```

CREATE FUNCTION Clientes_x_Pais (@pais varchar(64))
RETURNS @clientes TABLE
(customer_id integer
,razon_social varchar(50)
,pais varchar(64)
)
AS
BEGIN

    INSERT @clientes
    SELECT customer_id, razon_social, pais
    FROM Cliente
    WHERE pais = @pais;

    RETURN

END

```

Ejecución de la función creada
 SELECT * FROM Clientes_x_Pais('Chile')

Ejemplo 3: Función con valores de tabla en línea
 La misma función que el ejemplo anterior, recibe un país como parámetro y devuelve todos los clientes de ese país.

```

CREATE FUNCTION Clientes_x_Pais2 (@pais varchar(64))
RETURNS TABLE
AS
RETURN
(
    SELECT customer_id, razon_social, pais
    FROM Cliente
    WHERE pais = @pais
)

```

Ejecución de la función creada
 SELECT * FROM Clientes_x_Pais2('Chile')