

# Unidad 1 – parte 2

## Repaso++ de SQL

Bases de Datos Aplicada

v1.0.2 – Abril 2024



Universidad Nacional  
de La Matanza

**DIIT**  
Departamento de Ingeniería e  
Investigaciones Tecnológicas

# Contenido

- Definiciones básicas: Modelos de datos, abstracción, relaciones.
- Tipos de datos, esquemas, catálogos.
- Restricciones e integridad referencial.
- Objetos de una DDBB (tabla, vista, stored procedure, trigger, etc).
- SQL básico: SELECTs, juntas, uniones, subconsultas.
- Window Functions.
- Common Table Expressions.
- SQL dinámico.
- Pivot.

# Al finalizar deberías ser capaz de...

- Crear una DDBB y sus objetos básicos.
- Asegurar la consistencia de los datos en cuanto a su naturaleza y relación.
- Realizar consultas básicas y avanzadas.

# Modelo de datos: abstracción

La **abstracción de datos** se refiere a la eliminación de los detalles de la organización y almacenamiento en favor de las características esenciales para una comprensión mejorada de los datos.

Un **modelo de datos** –*colección de conceptos que pueden usarse para describir la estructura de una DB*– provee los medios para lograr dicha abstracción.

**Estructura de la base de datos** refiere a los tipos de datos, relaciones y restricciones que aplican a los datos.

Fuente: Elmasri capítulo 2.

# El modelo relacional: constraints

Las bases de datos relacionales constan de muchos tipos de **relaciones**.

Las restricciones (*constraints*) dependen del estado de la DB.

Las restricciones se pueden clasificar en:

- Inherentes al modelo de datos o implícitas.
- Expresadas en el esquema o explícitas.
- Semánticas o reglas de negocio (se manejan en la aplicación).

Fuente: Elmasri 3.2

# El modelo relacional: constraints

## Ejemplos

- Inherentes al modelo de datos o implícitas.
  - Las impuestas por tipos de datos. “Solo números”
- Expresadas en el esquema o explícitas.
  - Generadas por relaciones (PK-FK), constraints, check. “El cliente ya debe existir”.
- Semánticas o reglas de negocio (se manejan en la aplicación).
  - 2da unidad al 70% no válido en Mendoza ni Chandon.

Fuente: Elmasri 3.2

# Definición y tipos de datos

Tabla, fila y columna corresponden a relación, tupla y atributo del modelo relacional. CREATE es la declaración para generar esquemas, tablas, y otros objetos de la DB.

Se utiliza la declaración CREATE TABLE para generar relaciones nuevas, definir sus atributos y restricciones iniciales.

```
CREATE TABLE nombreEsquema.nombreTabla (  
    Campo1 tipoDato [restricciones]  
    ...  
);
```

Fuente: Elmasri cap. 4

# Esquemas (schema)

Equivalen a un namespace: jerarquía de objetos de la DB

Las primeras versiones de SQL no incluían el concepto de esquema.

Un esquema se identifica por nombre.

```
CREATE SCHEMA nombreEsquema;
```

*Postgres admite borrado en cascada de un esquema y todo lo que contiene. SQL Server no admite duplicidad en objetos entre esquemas.*



# Catálogos

**Catálogo:** un conjunto de esquemas en un entorno.

Un **entorno** es típicamente una instalación del RDBMS.

El catálogo contendrá información de todos los **esquemas** de un entorno (o instancia).

Pueden definirse restricciones entre esquemas del mismo catálogo.

En un sistema de computación pueden instalarse varias **instancias**, tal que cada una *escuche* en un puerto distinto.

Un RDBMS puede permitirnos trabajar con varios catálogos a la vez.

Fuente: Elmasri 4, Silverschatz 4.14

# Catálogo.Esquema.Objeto

Las conexiones al RDBS se realizan en un contexto de seguridad (usuario).

Cada usuario puede tener un **catálogo y esquema predeterminado**.

Cada objeto tiene un nombre completo compuesto por  
`catálogo.esquema.nombreObjeto`.

Se puede trabajar con catálogo y esquema predeterminados.

Fuente: Elmasri 4, Silverschatz 4.14

# DDL: Data Definition Language

Define y modifica el esquema de la base de datos.

## **CREATE**

- Crea una base de datos o algún objeto (tabla, vista, índice, SP, trigger, etc.)

## **DROP**

- Elimina una base de datos o algún objeto.

## **ALTER**

- Modifica un objeto de la base de datos.

## **TRUNCATE**

- Elimina el contenido de una tabla pero mantiene la estructura.

# DML: Data Manipulation Language

Permite manipular los datos: agregar, eliminar, modificar.

**INSERT :** Crea registros en una tabla.

**UPDATE :** Modifica uno o varios registros en una tabla.

**DELETE :** Elimina uno o más registros de una tabla.

**MERGE :** Realiza inserción, actualización o borrado según el resultado de una junta con una tabla de origen.

*Los motores modernos no hacen distinción entre DDL, DML (ni DQL –consultas-, DCL –control-, TCL –transacciones-).*

# Tabla, fila y columna

Se utiliza la declaración CREATE TABLE para generar relaciones nuevas, definir sus atributos y restricciones iniciales.

```
CREATE TABLE nombreEsquema.nombreTabla (  
    Campo1 tipoDato [restricciones]  
    ...  
);
```

*Ejemplo:*

```
CREATE TABLE ddbba.alumno (  
    Apellido char(50)  
);
```

# Tipos de datos

Existen tipos básicos definidos en ANSI y otros provistos por el RDBMS.

- Enteros (1 byte: `tinyint`; 2 bytes: `smallint`; 4 bytes: `int`; 8 bytes: `bigint`).
- Booleano: `bit`.
- Punto fijo decimal: `decimal` / `numeric`. Se define total de dígitos y los reservados para decimales (`total,decimales`).
- Fecha y hora: `datetime`, `smalldatetime`, `date`, `time`
- Punto flotante: simple y doble precisión.
- Cadenas de carácter: `char`. Si admite Unicode: `nchar`.  
Si son de longitud variable: `varchar/nvarchar`.

# Integridad referencial, claves primarias y foráneas, restricciones.

**Integridad referencial:** mecanismo por el cual el RDBMS asegura que un valor que aparece en una relación para un conjunto de atributos determinado también aparezca en otra relación para un cierto conjunto de atributos.

Cada relación tendrá **restricciones** de integridad referencial.

Las operaciones de inserción, borrado y actualización se **comprueban para asegurar que no violen** las reglas de integridad referencial.

# Integridad referencial, claves primarias y foráneas, restricciones.

**Clave primaria:** conjunto mínimo de campos que identifican de forma unívoca un registro. No pueden ser nulos (restricción implícita).

Es recomendable definir una clave primaria (simple o compuesta) para toda tabla, incluso las tablas temporales y variables.

En cada inserción se validará que no esté repetido (restricción implícita).

Será el(los) campo(s) del índice clúster, dictando el orden físico de los registros.

Es buena idea que sea un valor inherentemente creciente.

Es mala idea utilizar un campo que se pueda modificar.

Fuente: Silverschatz capítulo 6.



# Integridad referencial, claves primarias y foráneas, restricciones.

```
CREATE TABLE ddbba.alumno (  
    DNI      int primary key,  
    Apellido char(50)  
);
```

```
CREATE TABLE ddbba.alumno (  
    DNI      int,  
    Apellido char(50),  
    constraint pk_alumno primary key clustered (DNI)  
  
); -- dos formas de lograr lo mismo
```

*¿Qué rango de valores de DNI admitirá el sistema?  
¿Le parece óptima la selección del campo PK?*

# Integridad referencial, claves primarias y foráneas, restricciones.

**Clave foránea:** ocurrencia en una tabla dada de la clave primaria de otra tabla dada que registran una relación unívoca entre ambas. Referencia los atributos que forman la clave primaria de la tabla referenciada.

Se asume que no admite valores nulos.

Generan una restricción implícita.

Puede indicarse una operación de actualización o borrado en cascada.

No se generan índices automáticamente ante su creación.

Es **buena idea** usar un criterio definido para identificar los campos.

Es **importante** asignarles el mismo tipo de dato.

Fuente: Silverschatz capítulo 6.

# Integridad referencial, claves primarias y foráneas, restricciones.

```
CREATE TABLE ddbba.curso (  
    ID      int identity(1,1) primary key,  
    DNI     int REFERENCES ddbba.alumno (DNI)  
);  
  
CREATE TABLE ddbba.examen (  
    DNI      int,  
    IDCurso  smallint,  
    CONSTRAINT FK_InscripcionCurso (DNI, IDCurso)  
    REFERENCES ddbba.curso (DNI, id)  
); -- ejemplo de FK de más de un campo
```

Fuente: learn.microsoft.com

# Integridad referencial, claves primarias y foráneas, restricciones.

**Restricciones:** Reglas que el RDBMS aplicará a los valores de la tabla. Además de las que se generan por claves primarias (valor único) y claves foráneas (valor existente en la tabla referenciada), pueden definirse otras restricciones.

**UNIQUE:** impide la carga de duplicados.

**CHECK:** puede utilizarse para restringir valores (por ejemplo solo positivos), o que el valor se ajuste a un formato específico.

**NOT NULL:** impide que el valor quede en nulo.

**DEFAULT:** determina un valor por defecto.

Fuente: [learn.microsoft.com](https://learn.microsoft.com)

# Integridad referencial, claves primarias y foráneas, restricciones.

```
Create Table ddbba.alumno (  
    DNI      int CHECK (DNI > 0) ,  
    nombre char(20) UNIQUE,  
    patente  char(7) ,  
    CONSTRAINT CK_patente CHECK (  
        patente LIKE ' [A-Z] [A-Z] [0-9] [0-9] [0-9] [A-Z] [A-Z] '  
        OR patente LIKE ' [A-Z] [A-Z] [A-Z] [0-9] [0-9] [0-9] '  
    ) - solo autos!  
);
```

Observe que las restricciones CONSTRAINT requieren un nombre.

Fuente: learn.microsoft.com

# NULL

Puede interpretarse como dato ausente, desconocido o que no aplica.

No existe una teoría relacional satisfactoria que lo incluya.

Cada RDBMS puede hacer un manejo distinto.

La comparación `NULL == NULL` siempre será FALSE.

Existen condiciones del tipo **`valor is [not] null`** para ello, o reemplazos del tipo **`isnull(valor, 0)`**

En un JOIN no se muestran los registros con NULL en un campo de la junta.

*¿Se cuenta un NULL en un `count()`?*

*¿Se contabiliza en un `AVG()`?*

Concatenar cadenas de texto y NULL... generará un NULL.

# Consultas básicas

```
SELECT    campo1, campo2, ..., campoN  
FROM      esquema.tabla  
WHERE     condición
```

SELECT: corresponde a la proyección en álgebra relacional.

FROM: corresponde al producto cartesiano del álgebra relacional.

WHERE: corresponde al predicado selección del álgebra relacional.

Primero se resuelve el FROM, luego el WHERE y finalmente el SELECT.... *¿O no?*

# Juntas y uniones

```
SELECT    t1.campo1, t2.campo1
FROM      esquema.tabla t1 INNER JOIN
          esquema.tabla2 t2 ON t1.campoX = t2.campoY
```

## **UNION**

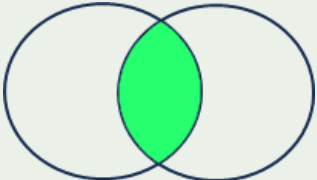
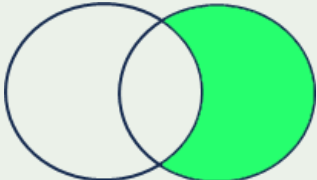
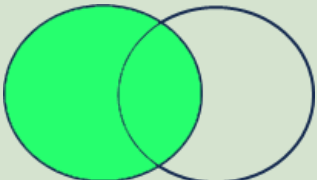
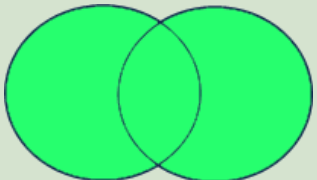
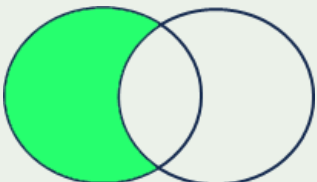
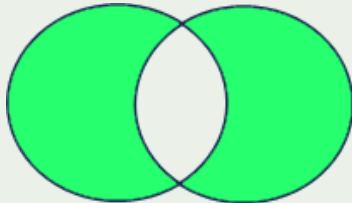
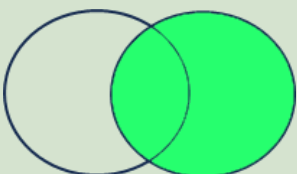
```
SELECT    t3.campo1, t4.campo1
FROM      esquema.tabla3 t3 LEFT JOIN
          esquema.tabla4 t4 ON t3.campoX = t4.campo
```

**UNION:** Combina resultados. Suprime duplicados salvo que se indique ALL.

**JOIN:** Cruza resultados (variantes LEFT, RIGHT, OUTER).



# Juntas y uniones

<b>INNER JOIN</b> 	<pre>SELECT * FROM A INNER JOIN B ON A.key = B.key</pre>	<b>RIGHT JOIN (sin intersección de A)</b> 	<pre>SELECT * FROM A RIGHT JOIN B ON A.key = B.key WHERE A.key IS NULL</pre>
<b>LEFT JOIN</b> 	<pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key</pre>	<b>FULL JOIN</b> 	<pre>SELECT * FROM A FULL OUTER JOIN B ON A.key = B.key</pre>
<b>LEFT JOIN (sin intersección de B)</b> 	<pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key WHERE B.key IS NULL</pre>	<b>FULL JOIN (sin intersección)</b> Admite también FULL OUTER JOIN 	<pre>SELECT * FROM A FULL OUTER JOIN B ON A.key = B.key WHERE A.key IS NULL OR B.key IS NULL</pre>
<b>RIGHT JOIN</b> 	<pre>SELECT * FROM A RIGHT JOIN B ON A.key = B.key</pre>		

# SQL es un lenguaje declarativo

Se indica qué queremos obtener... no la forma de hacerlo.

El ***optimizador de consultas*** analiza la consulta, genera un **plan de ejecución**, lo guarda en caché y lo pasa al ***execution engine***.

Analizar el plan de ejecución nos permite ver de qué forma se ejecuta la consulta.

En base a eso podemos optimizarla.

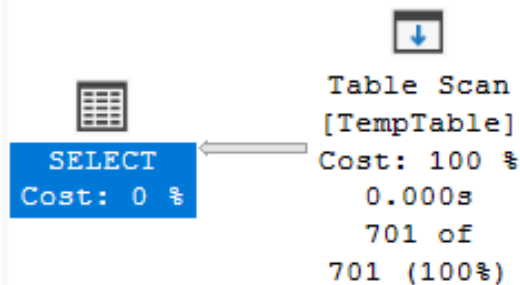
Podemos pensar en el plan de ejecución como un encadenamiento de funciones o procesos donde cada uno accede a los datos y/o realiza una tarea de preparación.

# SQL es un lenguaje declarativo

```
SELECT *  
FROM TempTable;
```

Esto le pedimos

Query 1: Query cost (relative)  
SELECT \* FROM TempTable



Esto ejecuta

Query 1: Query cost (relative)  
SELECT \* FROM TempTable

Table Scan

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	701
Actual Number of Rows for All Executions	701
Actual Number of Batches	0
Estimated I/O Cost	0,0061665
Estimated Operator Cost	0,0070161 (100%)
Estimated CPU Cost	0,0008496
Estimated Subtree Cost	0,0070161
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	701
Estimated Number of Rows Per Execution	701
Estimated Number of Rows to be Read	701
Estimated Row Size	65 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object  
[AdventureWorks2017].[dbo].[TempTable]

Output List  
[AdventureWorks2017].[dbo].[TempTable].Id; [AdventureWorks2017].[dbo].[TempTable].Dsc

# Subconsultas

Expresiones del tipo `SELECT...FROM...WHERE` que se anidan en otras expresiones `SELECT...FROM...WHERE`.

Pueden usarse subconsultas en el *where*, en el *from* y en el *select*.

También pueden utilizarse en otras sentencias DML: update, delete, etc).

```
SELECT campo1, campo2
FROM esquema.tabla1
WHERE campo3 in ( SELECT campoX
                  FROM esquema.tabla2
                  WHERE campoY IS NULL)
```

# Vistas

Tipo de dato derivado: es una **tabla derivada** (de tabla(s) o vista(s))

- Solo se almacena *la consulta*.
- Se puede consultar igual que una tabla.
- Se puede utilizar para Insert, Update, Delete con limitaciones.
- Permite mantener retrocompatibilidad con estructuras modificadas.
- Pueden usarse para limitar el acceso de usuarios a campos o registros
- La opción SCHEMABINDING bloquea cambios en las tablas utilizadas.

# Vistas

```
CREATE OR ALTER VIEW ddbba.cursosCopados
WITH SCHEMABINDING
AS
    SELECT      nombreCatedra, diaCursada,
               turno, sum(inscriptos)
    FROM        ddbba.cursos
    WHERE       puntuacion > 3
    GROUP BY    nombreCatedra, diaCursada, turno
```

*¿Qué nombre tendrá en el resultado el campo  
sum(inscriptos)?*

# Funciones de agregado

Realizan operaciones de resumen sobre grupos de filas.

- Contar, sumar, mínimo, máximo, promedio.
- Debemos indicar el criterio de agrupado (`GROUP BY`).
- No distinguen registros duplicados (uso de `DISTINCT`).
- `COUNT` genera un tipo entero, las demás funciones respetan el tipo de dato del campo resumido.
- Se puede usar `HAVING` para agregar condiciones sobre el resultado de las funciones de agregado.

Fuente: Elmasri 5.1.7

# Funciones de agregado

Ejemplo:

```
SELECT      nombreCatedra,  
            sum(inscriptos) I  
  
FROM        ddbba.curso  
  
GROUP BY    nombreCatedra  
  
HAVING      sum(inscriptos) > 90
```



# Recapitulemos...

Conceptualmente una consulta se evalúa:

1. Preparando las tuplas a partir de la cláusula `FROM` (desde tablas, vistas, subconsultas, etc).
  2. Filtrando según el criterio del `JOIN` y la cláusula `WHERE`.
  3. Aplicando el agrupamiento indicado por `GROUP BY`
  4. Filtrando según el criterio de `HAVING`.
  5. Ordenando tal como se haya indicado por `ORDER BY`
  6. Presentando las columnas solicitadas en el `SELECT`.
- Si una cláusula no está presente, ese paso no se realiza.

*Realmente... el analizador y el optimizador modifican el orden y paralelizan todo lo posible.*

Fuente: Elmasri 5.1.9

# Procedimientos almacenados

Módulos de programa (o rutinas compiladas) almacenados en la DB.

- Se ejecuta en el mismo RDBMS.
- Son útiles para normalizar el acceso a datos desde distintas aplicaciones.
- Reducen el tráfico de datos entre cliente-servidor.
- Permite parámetros de entrada y salida, de los mismos tipos de datos que los campos de las tablas.
- Cada RDBMS implementa una variante de lenguaje de programación (PL/SQL, T-SQL, PL-pgSQL, y más...).
- En PostgreSQL admiten sobrecarga.

Fuente: Elmasri 13.4

# Procedimientos almacenados

```
CREATE OR ALTER PROCEDURE esquema.NombreSP
    @parametro1 int,
    @parametro2 char(20),
    @parametro3 int=0,
    @parámetro4 datetime OUTPUT
AS
BEGIN
    -- comentario
END
```

# Procedimientos almacenados

```
/* Ejemplo de declaración de variable en T-SQL,  
   Invocación de SP y vista en consola.  
   También comentario multi línea */
```

```
declare @variable datetime  
EXEC esquema.NombreSP 1, 'Sergio Q. Lopez',,@variable OUTPUT  
print @variable
```

*Cada variante de lenguaje de programación de SP de  
cada RDBMS implementa características similares.*

# Triggers

Tipo especial de stored procedure que se ejecuta en forma automática. Permiten especificar acciones que realizará el RDBMS frente a ciertos eventos y condiciones.

- Los ejecuta el RDBMS antes o después del evento interceptable (DML, DDL).
- Pueden configurarse sobre la DB misma.
- En su ámbito de ejecución se dispone de las tablas inserted y deleted.
- Oracle admite comportamiento FOR EACH ROW (SQL Server no; PostgreSQL SI)
  - No permite modificar otro registro excepto el “*each row*”
- Oracle admite tiempo como disparador (SQL Server usa Jobs;)

Fuente: Elmasri 5.2.2

# Triggers

```
CREATE OR ALTER TRIGGER esquema.NombreTG ON  
esquema.Tabla  
AFTER INSERT, UPDATE AS -- no tienen parámetros  
BEGIN  
    declare @cantidad int  
    set @cantidad = (Select count(1) from inserted)  
    print @cantidad  
END
```

<https://learn.microsoft.com/es-es/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16>

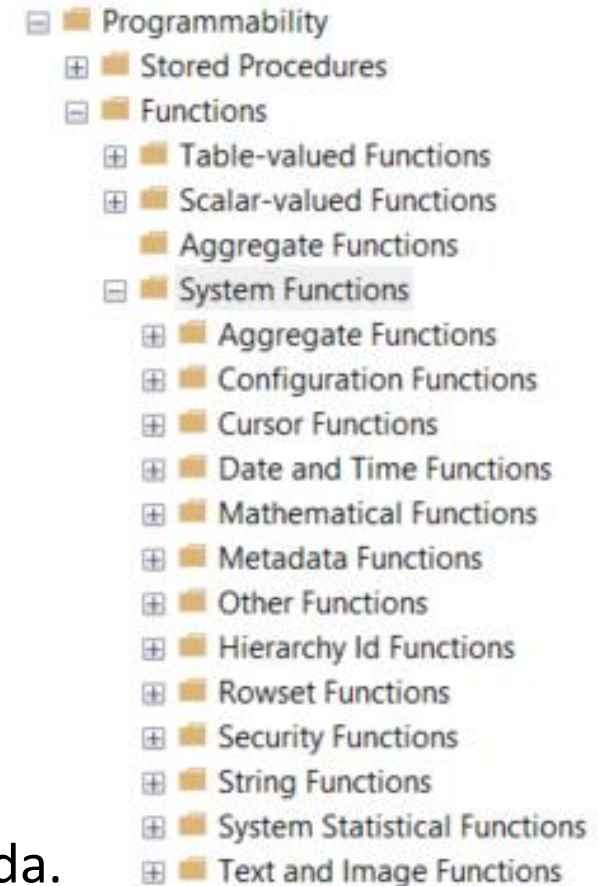
# Funciones

Se pueden clasificar en varios tipos.

- Agregado: Consideradas previamente.
- Incluidas en el sistema (de biblioteca).
- Definidas por el usuario.

Se programan de forma similar a los stored procedures:

- Solo tienen parámetros de entrada (no de salida)
- Admiten un valor de retorno escalar o vectorial.
- Pueden usarse en el SELECT, FROM o WHERE.
- Pueden utilizarse en expresiones.
- Según el RDBMS se invocan asignando su valor a una variable o salida.



# Funciones (ejemplo)

```
CREATE OR ALTER Function [dbo].[ValidaCUIT] (@cuit varchar(13))
returns int
Begin
```

```
    declare @cuitMIO  varchar(11)
    SET @cuitMIO = ltrim(rtrim(replace(@cuit, '-', '')))
    SET @cuitMIO = ltrim(rtrim(replace(@cuitMio, '_', '')))
    SET @cuitMio = ltrim(rtrim(@cuitmio))
    declare @suma  int
           ,@contador  int
           ,@retorno  int
    SET @retorno= -1
    if @cuitMio <> ''
    begin
        select @contador= 1
        select @suma= 0
```



```

while @contador < 12
begin
    SET @suma = @suma + ((cast(SUBSTRING(@cuitMIO,@contador,1)
as Int) + 48) *
        case @contador
            when '1' then 5
            when '2' then 4
            when '3' then 3
            when '4' then 2
            when '5' then 7
            when '6' then 6
            when '7' then 5
            when '8' then 4
            when '9' then 3
            when '10' then 2
            when '11' then 1
        End )
    set @contador = @contador + 1
End --fin del while
set @suma = abs(@suma % 11)
if @suma = 3
    SET @retorno = 1

End -- fin del IF
return @retorno
end

```

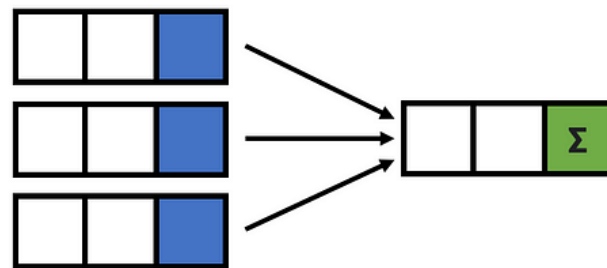
# Window Functions

Estándar a partir de SQL:1999. De la documentación de PostgreSQL:

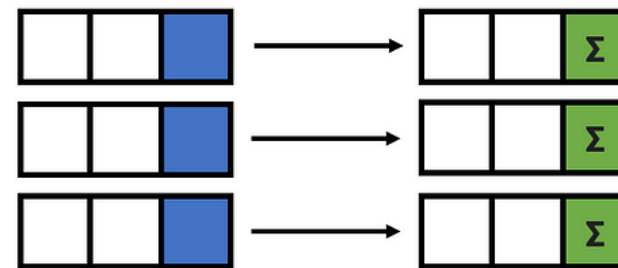
”Realizan cálculos en un conjunto de filas de una tabla que están relacionadas con la fila actual. Detrás de escena la *window function* puede acceder a más filas que solo la actual.”

- Permiten trabajar con columnas agregadas y no agregadas, producen queries más simples.
- Se clasifican en agregado, ranking y valor.
- Devuelven un valor por cada fila de la consulta (en contraste con las de agregado).

Aggregate Functions (SUM, AVG, etc.)



Window Functions



Fuente: Silverschatz 22.2.5

# Window Functions

- La ventana se define con la cláusula OVER(), pudiendo indicarse una columna específica.
- No se pueden usar en el FROM, WHERE, GROUP BY o HAVING. Solo en SELECT y ORDER BY.

*Ejemplo: Dada una tabla conteniendo registros de ventas por ciudad, se desea ver además el promedio del monto de ventas para esa ciudad y ese día.*

```
select id, fecha, ciudad, monto,  
       AVG(monto) OVER (PARTITION BY fecha,ciudad) as PromedioDiario  
From   ddbba.venta  
order by ciudad,fecha
```

*Ver guía de “Window functions” para más ejemplos.*

# Window functions

Ejemplos de aplicación:

- Ver el sueldo promedio de distintos puestos y la diferencia entre el sueldo de cada empleado y el promedio.
- Ver el saldo acumulado de una cuenta.
- Determinar los N productos más vendidos (cualquier cantidad N de ítems rankeables).

Compare la simpleza de la sintaxis respecto a la misma consulta sin usar *window functions*.

<https://towardsdatascience.com/a-guide-to-advanced-sql-window-functions-f63f2642cbf9>

# Common Table Expressions

Resultados temporales con nombre.

- Facilitan la comprensión de consultas complejas (especialmente si incluyen subconsultas).
- Su ámbito está limitado a una única consulta SELECT, INSERT, UPDATE, DELETE o MERGE.
- Admiten recursividad. Solo pueden referenciar otra CTE definida antes (no después, por eso no hay problema de recursividad cruzada).
- Se define entre paréntesis con la palabra `WITH` y se designa un alias.
- Pueden definirse varios CTE para luego usar en una consulta.
- Pueden usarse en vistas.

# Common Table Expressions

Ejemplo: Ver cantidad de medallas de oro.

```
WITH oro_maraton AS
(
    SELECT    city,    year,    country
    FROM olympic_games
    WHERE medal_type = 'Gold' AND sport like 'Marathon%' )
SELECT country,
    count(*) AS Medallas_doradas_maraton
FROM oro_maraton
GROUP BY country
ORDER BY gold_medals_in_marathon DESC;
```

<https://www.sqlservercentral.com/articles/what-exactly-is-a-cte-in-t-sql-a-comprehensive-guide-with-7-examples>

# Common Table Expressions

Ejemplo: Obtener la serie de Fibonacci

```
WITH Fibonacci (PrevN, N) AS  
(  
    SELECT 0, 1  
    UNION ALL  
    SELECT N, PrevN + N  
    FROM Fibonacci  
    WHERE N < 100  
)  
SELECT PrevN as Fibo  
FROM Fibonacci  
OPTION (MAXRECURSION 0);
```

Recomendación: Hágalo con un bucle. *Recursividad = mala idea.*

Ver guía de “Common Table Expresions” para más ejemplos.

# SQL Dinámico

Se utiliza en escenarios donde no es posible contar con la consulta requerida hasta el momento de ser ejecutada.

- Se debe generar una cadena de texto con la sentencia SQL a ejecutar.
- La sentencia se prepara en tiempo de ejecución y se ejecuta en un ámbito separado.
  - El código dinámico accederá a la DB pero no a variables del llamador.
- Es importante tomar en consideración el riesgo de ataques de SQL Injection.



# SQL Dinámico

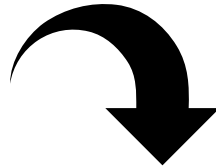
```
DECLARE @CadenaSQL NVARCHAR(MAX) ;
SELECT @CadenaSQL = COALESCE(@CadenaSQL + ' UNION ALL ', '')
                    + 'SELECT '
                    + '''' +
QUOTENAME(SCHEMA_NAME(sOBJ.schema_id))
          + '.' + QUOTENAME(sOBJ.name) + '''' + '
AS [Tabla]
          , COUNT(1) AS [CuentaDeFilas] FROM '
          + QUOTENAME(SCHEMA_NAME(sOBJ.schema_id))
          + '.' + QUOTENAME(sOBJ.name)
FROM sys.objects AS sOBJ
WHERE
    sOBJ.type = 'U'
ORDER BY SCHEMA_NAME(sOBJ.schema_id), sOBJ.name ;
EXEC sp_executesql @CadenaSQL
```

*Ver guía de “SQL dinámico” para más ejemplos.*

# PIVOT

Útil para presentar resultados más legibles.  
También conocido como *transposición*.

Results Messages			
	Total	Ciudad	Mes
1	36799.00	Buenos Aires	1-2023
2	14656.00	Carlos Paz	1-2023
3	15571.00	Claromeco	1-2023
4	43007.00	Iguazu	1-2023
5	31380.00	Rosario	1-2023
6	34226.00	Buenos Aires	2-2023
7	23095.00	Carlos Paz	2-2023
8	17943.00	Claromeco	2-2023
9	46113.00	Iguazu	2-2023
10	26425.00	Rosario	2-2023
11	40616.00	Buenos Aires	3-2023
12	16909.00	Carlos Paz	3-2023
13	11369.00	Claromeco	3-2023
14	46673.00	Iguazu	3-2023
15	21720.00	Rosario	3-2023



Results Messages								
	Ciudad	1-2023	2-2023	3-2023	4-2023	5-2023	6-2023	7-2023
1	Buenos Aires	36799.00	34226.00	40616.00	34988.00	42648.00	26889.00	40552.00
2	Carlos Paz	14656.00	23095.00	16909.00	23808.00	18328.00	18285.00	23159.00
3	Claromeco	15571.00	17943.00	11369.00	15589.00	11578.00	17338.00	15113.00
4	Iguazu	43007.00	46113.00	46673.00	49320.00	54410.00	41578.00	46646.00
5	Rosario	31380.00	26425.00	21720.00	34882.00	19345.00	35252.00	32703.00

# PIVOT

Observe que se realiza el pivot por mes y ciudad.

Requiere siempre una función de agregado y un nombre.

```
with VentasResumidas (Total, Ciudad, Mes) as (  
    select monto,  
           Ciudad,  
           cast(month(fecha) as varchar) + '-' + cast(year(fecha) as varchar)  
           Mes  
           from ddbba.Venta) -- aquí cierra el CTE  
select *  
from VentasResumidas  
    pivot(sum(Total)  
          for Mes in ([1-2023],[2-2023],[3-2023],[4-2023],[5-2023],[6-2023],[7-  
2023])) ) Cruzado
```

*Ver guía de “Pivot” para más ejemplos.*

# ¿Dudas?



Universidad Nacional  
de La Matanza

# DIIT



Departamento de Ingeniería e  
Investigaciones Tecnológicas