



# Intro to SQL

# Hear, forget. See, remember. Do, Understand

## / Objetivos

- Aprender conceptos fundamentales de base de datos como entidades y relaciones.
- Entender dónde situar SQL en el panorama actual.
- Conocer la sintaxis básica del lenguaje, así como funciones más avanzadas.
- Familiarizarse con la forma de trabajo en proyectos reales.





# Contexto

# SQL | Contexto

- Todas las empresas quieren ser hoy en día **data-driven**.
- Y a lo largo de los años han ido creando **bases de datos** para poder operar en su día a día, así como para analizar históricos.
- Esos sistemas, muy diversos entre sí, comparten sin embargo una forma de trabajar: **Structured Query Language (SQL)**.
- Aunque tiene connotaciones negativas (antiguo, *verboso*, limitado, etc.) al final siempre termina apareciendo en los lugares más insospechados.
- Razones? (ahí va mi apuesta):
  - Facilidad de aprendizaje.
  - Ser declarativo.
  - Ubíquo.

# SQL | Modelado

- Las bases de datos se modelan a partir de **entidades** (pensadlos como nombres: personas, lugares, cosas o eventos)
  - En un dataset de libros, nos encontraremos entidades como *libros, autores, editores...*
  - En un dataset de deportes, tendremos *jugadores, posiciones, equipos, partidos...*
- Estas entidades tienen conexión con otras entidades a través de distintos tipos de **relaciones**
  - **1:1**, cuando una entidad puede tener o pertenecer a otra entidad únicamente (*1 libro es publicado por 1 editorial*)
  - **1:M**, cuando una entidad puede tener o pertenecer a múltiples entidades (*1 equipo se compone de muchos jugadores*)
  - **M:M**, cuando una entidad puede tener o pertenecer a múltiples entidades, y viceversa (*1 libro es creado por múltiples autores y 1 autor puede crear múltiples libros*)

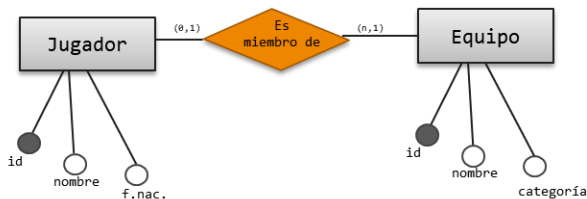
# SQL | Normalización

- La información en una base de datos relacional se organiza de una manera *normalizada*.
- Existen seis reglas (**formas normales**) que permiten evitar las anomalías e inconsistencias de un modelo, que se pueden resumir en:
  - Cada entidad debe estar representada como una **tabla** donde cada representación individual (registro) debe tener un **identificador único** (clave primaria)
  - Las relaciones entre entidades se mantienen mediante **claves foráneas**, donde un campo de una tabla hace referencia al identificador de la tabla relacionada.
- El objetivo de este diseño normalizado es evitar la **redundancia** y garantizar la **coherencia** de los datos.
- Pero... hay más formas de modelar una base de datos (*dimensional modeling, data vault, anchor modeling, etc.*) cuando nuestros objetivos son otros (**analytics**).

# SQL | Diseño

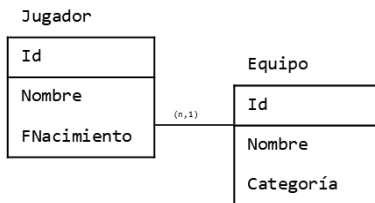
- La forma de diseñar las bases de datos es a través de **diagramas Entidad Relación (E/R)**.
- A través de ellos podemos ver las entidades de las que se compone nuestro modelo, así como las relaciones existentes entre ellas.

## Modelo Conceptual



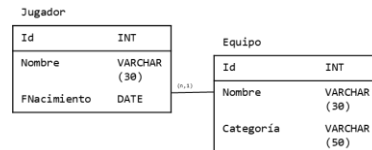
- Descripción a alto nivel para que técnico y usuario lleguen a un acuerdo.
- Solo se incluye la realidad que tenga sentido almacenar en la base de datos.

## Modelo Lógico



- Diseño teórico de las tablas, sin tener en consideración criterios específicos del motor de base de datos

## Modelo Físico



- Diseño final de las tablas considerando ya el motor donde se implementarán
- Se tienen en cuenta consideraciones de rendimiento

## SQL | Summary

**#1:** Hemos retrocedido un poco para entender el lugar que ocupa SQL hoy en día.

**#2:** Para pasar a comentar brevemente los conceptos fundamentales del diseño de base de datos.

**#3:** Así que ya estamos preparados para empezar a jugar.





# Lenguaje SQL

# SQL

- *Structured Query Language.*
- Diseñado, principalmente, para SGBDR.
- Aunque es un lenguaje estándar, existen implementaciones **particulares** (según el motor).
- Lenguaje **declarativo**: *qué*, no el *cómo*.
- El lenguaje no es sensible a mayúsculas/minúsculas, pero los objetos creados sí pueden serlo.
- No es sólo para consultas
  - DDL – Data **Definition** Language
  - DML – Data **Manipulation** Language
  - DCL – Data **Control** Language



Credit: [https://commons.wikimedia.org/wiki/File:Sql\\_data\\_base\\_with\\_logo.png](https://commons.wikimedia.org/wiki/File:Sql_data_base_with_logo.png)

# SQL | DDL

- Nos permite **crear, modificar y eliminar** objetos dentro de la base de datos
- Los nombres deben ser **únicos** dentro del nivel en el que se encuentran
  - Bases de datos dentro del servidor.
  - Tablas / vistas dentro de la base de datos.
  - Columnas dentro de la tabla / vista



## CREATE

Define un nuevo objeto (tabla, índice, vista, etc.)

```
CREATE TABLE { tabla } (  
  [ opciones creación de tabla ]  
  [ definiciones de columnas ]  
  [ restricciones a nivel de tabla ]  
  [ definiciones de índices ]  
) [ ; ]
```

```
CREATE TABLE dbo.myTable  
(  
  ID INT NOT NULL REFERENCES otherTable.ID(ID),  
  colB SMALLINT NOT NULL,  
  colC BIGINT NOT NULL,  
  colD FLOAT NULL,  
  colE DATETIME NOT NULL CONSTRAINT df DEFAULT (GETDATE()),  
  myTable_PK PRIMARY KEY CLUSTERED (ID, colB)  
);
```



## DROP

Elimina un objeto existente

```
DROP TABLE { tabla } [ ; ]
```

```
DROP TABLE dbo.myTable;
```



## ALTER

Modifica un objeto existente

```
ALTER TABLE { tabla } (  
  [ modificación de tabla ]  
  [ modificación de columna ]  
  [ modificación de restricción ]  
  [ modificación de índices ]  
) [ ; ]
```

```
ALTER TABLE dbo.myTable ADD colF INT NOT NULL;  
ALTER TABLE dbo.myTable ALTER COLUMN colF FLOAT NOT NULL;  
ALTER TABLE dbo.myTable DROP COLUMN colF;  
ALTER TABLE dbo.myTable DROP CONSTRAINT myTable_PK;
```

# SQL | DML

Tal como su nombre indica, nos permite manipular los datos.



## INSERT

Introduce registro(s) en una tabla

```
INSERT [INTO] { table } [(col1[,...n])]  
{  
  { VALUES ({DEFAULT | NULL | <expression> } [,...n] )}  
  |derived_table }  
} [;]
```

```
INSERT INTO dbo.myTable (ID, colB, colC, colD, colE)  
VALUES (1, 2, 3, 0.5, '20100101');
```



## DELETE

Elimina registro(s)

```
DELETE [FROM] { table } [WHERE <search_condition>]  
[ ; ]
```

```
DELETE dbo.myTable WHERE ID = 1;
```



## UPDATE

Modifica registro(s)

```
UPDATE { table } SET {col_name = {<expression> |  
DEFAULT | NULL}}  
[FROM { table_source} ]  
[WHERE {search_condition}] [ ; ]
```

```
UPDATE dbo.myTable SET colB = 10 WHERE ID = 1;
```

# SQL | DML

Además de las clásicas, existe una que nos permite ejecutar todas ellas en la misma instrucción.



## MERGE

Inserta, actualiza o ejecuta operaciones de borrado en la tabla destino a partir del resultado de una combinación con una tabla de origen

```
MERGE { target_table }  
USING { source_table }  
ON <merge_search_condition>  
WHEN MATCHED THEN <merge_matched>  
WHEN NOT MATCHED THEN <merge_not_matched>  
WHEN NOT MATCHED BY SOURCE <merge_not_matched>  
[;]
```

```
MERGE dbo.myTable  
USING dbo.myOtherTable  
ON myTable.ID = myOtherTable.ID  
WHEN MATCHED THEN UPDATE SET colB = 20  
WHEN NOT MATCHED THEN INSERT (colB) VALUES (myOtherTable.colZ)  
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

# Lab 01

**DDL**



# Lab 01

- Basado en [https://github.com/ih-datapt-mad/dataptmad1121\\_labs/tree/main/module-1/mysql](https://github.com/ih-datapt-mad/dataptmad1121_labs/tree/main/module-1/mysql)
- Pero usando un entorno de trabajo virtual: <https://dbfiddle.uk/> (Chrome)
- Vamos a tratar de crear las instrucciones CREATE TABLE a partir de los datos de ejemplo
- Una vez hecho, el siguiente paso es tratar de crear las instrucciones INSERT INTO
- Y por último (bonus time), tratemos de crear las instrucciones UPDATE y DELETE propuestas

# SQL | DML

Pero sin duda, el protagonista del lenguaje SQL es la posibilidad que nos ofrece para consultar y extraer los datos almacenados tal como necesitamos.



## SELECT

Devuelve registros de la base de datos y permite la selección de uno o varios registros y columnas de una o varios conjuntos de datos de origen.

```
SELECT [ TOP (top_expression) ] [ ALL | DISTINCT ]  
[ * | column_name | <expression> ] [,...n]  
[ FROM { source_table } [, ...n] ]  
[ WHERE <search_condition> ]  
[ GROUP BY <group_by_clause> ]  
[ HAVING <search_condition> ]  
[ ORDER BY <order_by_expression> ]  
[;]
```

```
SELECT * FROM dbo.myTable;
```



## SELECT | Logical order

¿Sabías, sin embargo, que el orden de ejecución de las cláusulas es distinto?

1. **FROM.** Esta fase identifica las tablas de origen y las combina según los criterios establecidos.
2. **WHERE.** A partir del conjunto de resultados anterior, le aplica los filtros establecidos.
3. **GROUP BY.** Esta fase organiza el conjunto de resultados en grupos que cumplan las condiciones establecidas.
4. **HAVING.** Filtra los grupos de la fase anterior que cumplan las condiciones establecidas.
5. **SELECT.** Esta fase procesa los elementos de esta cláusula, resolviendo las expresiones que existan y aplicando el criterio de unicidad en caso de que exista la cláusula DISTINCT.
6. **ORDER BY.** Se ordena el conjunto de datos de la fase anterior según la lista indicada.
7. **TOP.** Filtra registros del conjunto de datos anterior.

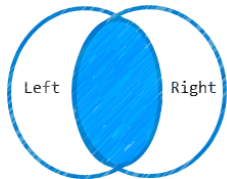
# SELECT | Joins

- Una de las capacidades más potentes de SQL es el de combinar de múltiples maneras distintos conjuntos de datos: por ejemplo, en Lab01 saber el nombre de los coches (*Cars*) que han sido vendidos (*Invoices*).
- Al combinar conjuntos de datos, el primero que se declara se le considera la parte izquierda (*left*) de la combinación y al segundo la parte derecha (*right*). Dependiendo del tipo de combinación, tener claro este dato es importante.

## INNER JOIN

Registros que existen en ambos conjuntos de datos.

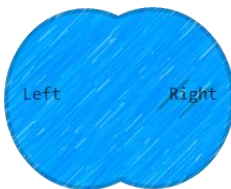
Es el predeterminado si no se especifica.



## FULL JOIN

Todos los registros de ambas tablas.

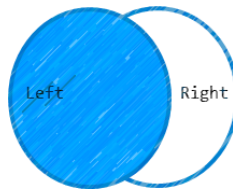
Los registros no existentes se ponen a NULL.



## LEFT JOIN

Todos los registros de la tabla izquierda de la combinación.

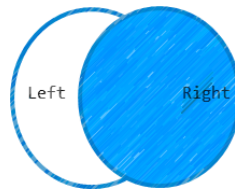
Los registros no existentes se ponen a NULL.



## RIGHT JOIN

Todos los registros de la tabla derecha de la combinación

Los registros no existentes se ponen a NULL.

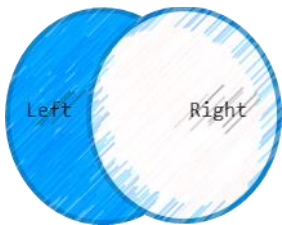


# SELECT | Joins

- Existen combinaciones más avanzadas

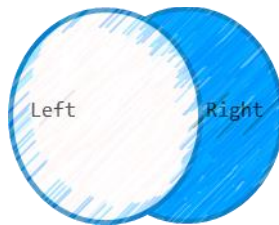
## EXCLUSIVE LEFT JOIN

Registros que solo se encuentran en la parte izquierda de la combinación.



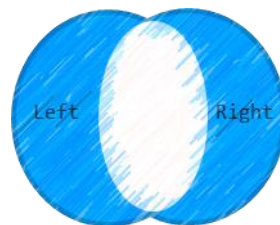
## EXCLUSIVE RIGHT JOIN

Registros que solo se encuentran en la parte derecha de la combinación.



## EXCLUSIVE FULL JOIN

Registros que solo se encuentran en la parte izquierda o que solo se encuentran en la parte derecha.  
Los registros no existentes se ponen a NULL.



## SELECT | Joins

- Es posible combinar tablas con una sintaxis basada (ANSI-89) en la cláusula WHERE

```
SELECT *  
FROM Cars, Invoices  
WHERE Cars.ID = Invoices.Car -- condición de combinación  
      AND Cars.Manufacturer = 'Ford' -- condición adicional
```

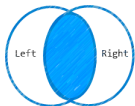
- Pero hoy en día (ANSI-92) es más normal (y recomendable) usar la cláusula [ INNER | LEFT | RIGHT | FULL ] JOIN

```
SELECT *  
FROM Cars INNER JOIN Invoices ON Cars.ID = Invoices.Car -- condición de combinación  
WHERE Cars.Manufacturer = 'Ford' -- condición adicional
```

# SELECT | Joins

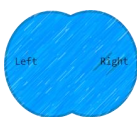
## INNER JOIN

```
SELECT *  
FROM Cars INNER JOIN Invoices  
ON Cars.ID = Invoices.Car
```



## FULL JOIN

```
SELECT *  
FROM Cars FULL JOIN Invoices  
ON Cars.ID = Invoices.Car
```



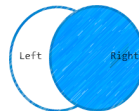
## LEFT JOIN

```
SELECT *  
FROM Cars LEFT JOIN Invoices  
ON Cars.ID = Invoices.Car
```



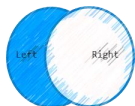
## RIGHT JOIN

```
SELECT *  
FROM Cars RIGHT JOIN Invoices  
ON Cars.ID = Invoices.Car
```



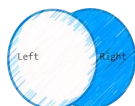
## EXCLUSIVE LEFT JOIN

```
SELECT *  
FROM Cars LEFT JOIN Invoices  
ON Cars.ID = Invoices.Car  
WHERE Invoices.Car IS NULL
```



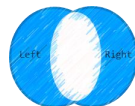
## EXCLUSIVE RIGHT JOIN

```
SELECT *  
FROM Cars RIGHT JOIN Invoices  
ON Cars.ID = Invoices.Car  
WHERE Cars.ID IS NULL
```



## EXCLUSIVE FULL JOIN

```
SELECT *  
FROM Cars FULL JOIN Invoices  
ON Cars.ID = Invoices.Car  
WHERE Cars.ID IS NULL OR  
Invoices.Car IS NULL
```



## SELECT | Joins

- Existe una combinación adicional que permite hacer **productos cartesianos**: CROSS JOIN
- Cada registro de la tabla izquierda se combina con un registro de la tabla derecha, obteniendo un conjunto que es el resultado de  **$n \times m$  filas** de los conjuntos originales.
- No existe la cláusula ON porque **no hay condición de combinación** como tal.
- Su uso es más limitado, pero es muy práctico para generar conjuntos de datos de ejemplo.

```
SELECT *  
FROM Cars CROSS JOIN Invoices
```

## SELECT | Joins

- Es posible usar **varias** instrucciones JOIN para combinar varias tablas.
- Es posible usar los **diferentes tipos** de JOIN en la misma instrucción.
- Es posible combinar una tabla **consigo misma** (por ejemplo en relaciones de jerarquía).
- En cualquier caso, es importante prestar atención a lo que indicamos en la **cláusula ON** para evitar combinaciones incorrectas que arrojen resultados inesperados.

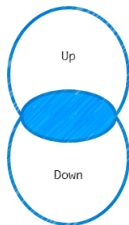
```
SELECT *  
FROM Cars INNER JOIN Invoices ON Cars.ID = Invoices.Car -- condición de combinación  
INNER JOIN Customers ON Invoices.Customer = Customers.ID -- condición de combinación
```

## SELECT | Set Operators

- Existen otras formas de operar con varias tablas además de combinaciones.
- A través de los operadores de conjuntos podemos **sumar o excluir** varios conjuntos de resultados en uno solo.
- Es cierto que al final combinamos datasets, pero en un caso (JOINS) lo ensanchamos (añadimos columnas) y en otro (Set Operators) lo **alargamos** (añadimos registros).
- La condición es que los datasets deben tener **igual forma** (número y tipo de datos de las columnas).

### INTERSECT

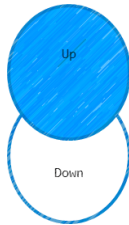
Devuelve las filas únicas que se encuentran en los dos datasets.



```
SELECT ID
FROM table1
INTERSECT
SELECT ID
FROM table2
```

### EXCEPT

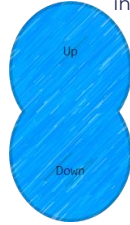
Devuelve los registros únicos que sólo están en el dataset inicial.



```
SELECT ID, c1
FROM table1
EXCEPT
SELECT ID, c2
FROM table2
```

### UNION [ALL]

Concatena los resultados de dos consultas en un único dataset. ALL incluye duplicados



```
SELECT c1, c4
FROM table1
UNION
SELECT c2, c6
FROM table2
```



# SELECT | ORDER

- El conjunto de resultados se puede **ordenar** en base a una condición específica.
- Es importante tener en cuenta que si no se especifica esta cláusula, **no hay garantía** de obtener un orden concreto en el resultado.
- En el criterio se puede usar el nombre de la columna, el alias o la posición que ocupa en la cláusula SELECT (aunque esto último no está recomendado) .
- Puede ser un criterio de ordenación compuesto por varias expresiones

## ORDER BY

Ordena el resultado en base a los criterios establecidos

```
[ ORDER BY {<order_by_expression> [ ASC | DESC ] } [,...] ]
```

```
SELECT * FROM dbo.myTable ORDER BY c1, c2 DESC;
```

# Lab 02

**JOIN**



## Lab 02

- Basado en la bbdd creada en Lab 01
- Usando el entorno de trabajo virtual: <https://dbfiddle.uk/> (Chrome)
- Probar los diferentes tipos de JOIN mostrados en la teoría combinando las tablas que consideréis (siempre que tengan sentido)
- INNER JOIN, FULL JOIN, [INCLUSIVE | EXCLUSIVE] LEFT JOIN, [INCLUSIVE | EXCLUSIVE] RIGHT JOIN, CROSS JOIN

# SELECT | WHERE

- Se usa para **filtrar** registros del conjunto de datos.
- No sólo aplica a la instrucción **SELECT**, se puede usar también en **UPDATE** e **INSERT**.
- Existen diferentes expresiones lógicas, pudiéndose **combinar** con operadores lógicos **AND** y **OR**.

=	Igualdad
>	Mayor que
<	Menor que
>=	Mayor o igual
<=	Menor o igual
<>	Distinto. En algunos casos también se puede usar !=
[NOT] BETWEEN ... AND	Entre
[NOT] LIKE	Patrón. Pueden usarse expresiones regulares como % (cualquier carácter), _ (un carácter), [] (cualquier carácter dentro del rango), [^] (cualquier carácter que no esté en el rango)
[NOT] IN	Valores posibles

# SELECT | Subconsultas

- Son consultas **dentro** de consultas.
- Se usan a menudo para filtrar datos de una forma más avanzada, y a veces por legibilidad.
- Pueden ser **escalares** (solo devuelven un único valor), **multivalor** o como **tabla** (múltiples campos, múltiples registros).
- Cuando la subconsulta tiene una **dependencia** con la consulta principal se denominan *consultas correlacionadas*.

## SUBCONSULTA ESCALAR

Devuelve un único registro

```
SELECT *  
FROM Invoices  
WHERE "Date" =  
(SELECT MAX("Date") FROM Invoices)
```

## SUBCONSULTA MULTIVALOR

Devuelve múltiples registros

```
SELECT *  
FROM SalesPerson  
WHERE ID IN  
(SELECT "Sales Person" FROM Invoices)
```

## SUBCONSULTA CORRELACIONADA

Depende de la consulta exterior.

```
SELECT *  
FROM Orders AS o1  
WHERE OrderDate =  
(SELECT MAX(OrderDate) FROM Orders AS o2  
WHERE o1.Customer = o2.customer)
```

## SELECT | Subconsultas

- Siempre van encerradas entre **paréntesis**.
- Los datos que devuelve **no se incluyen** en el conjunto de datos final.
- Pueden ir precedidas de distintas **cláusulas**
  - [NOT] ALL, devuelve TRUE cuando todos los valores retornados por la subconsulta satisfacen la comparación establecida.
  - [NOT] [ SOME | ANY ], devuelve TRUE cuando cualquier valor devuelto por la subconsulta satisface la comparación establecida.
  - [NOT] EXISTS, para comprobar si la subconsulta devuelve algún registro.

## SELECT | Alias

- Nombre **alternativo** asignado a una tabla, subconsulta o columna.
- Necesario cuando la misma entidad aparece **varias veces** en la instrucción.
- Pero también es muy útil para **legibilidad**.

# SELECT | Vistas

- Una forma de **reutilizar** una subconsulta (no correlacionada) es creando una vista.
- Una vista no es más que un **alias** dado a una consulta cuya definición (**no los datos**) se persiste.
- Tiene varias ventajas:
  - Permite **reutilizar** una consulta en varios sitios.
  - Permite **simplificar** una consulta compleja.
  - Permite ofrecer un **interfaz de acceso** a datos más amigable y seguro para consumidores finales.

## VISTA

Almacena la definición de una consulta

```
[ CREATE | ALTER ] VIEW {view_name} AS <select_statement>
```



# SELECT | CTE

- Acrónimo de **Common Table Expression**, es un nombre temporal dado a un resultset.
- Como si definiéramos una **vista temporal** con alcance dentro de la instrucción SELECT.
- Pueden crearse definiciones de **varias instrucciones**.
- Tiene varias ventajas:
  - o Permite **reutilizar** una consulta en varios sitios.
  - o Permite **simplificar** una consulta compleja.
  - o Hace más **legible** el código

## CTE

Especifica un nombre temporal a un resultset

```
WITH <cte_name> AS (<cte_query_definition>)
```

```
WITH max_values AS (SELECT MAX('Date') AS d FROM Invoices)  
SELECT *  
FROM Invoices  
WHERE 'Date' = (SELECT d FROM max_values)
```

## SQL | Summary

**#1:** Hemos visto los diferentes *dialectos* que tiene SQL.

**#2:** Hemos aprendido diferentes formas de combinar datasets.

**#3:** Y hemos empezado a filtrar conjuntos de datos.



# Las bbdd y el cloud computing

# Cloud Computing

- Aunque no específico a las bbdd, el cloud ha permitido usar (despliegue, escalado, testing, etc.) más fácilmente los SGBDR.
- Conocer sus fundamentos nos va a ofrecer otro punto de vista para trabajar con estos sistemas.



# Definición

## Qué es el cloud computing ?

*“Disponibilidad de servicios de **computación a demanda**  
y con pago por uso a través de **Internet**  
sin **mantenimiento directo** por parte del usuario”*

# Cloud Computing | Beneficios



## Ahorro

No solo evita la necesidad de comprar, poseer y mantener data centers físicos y pagar por lo que realmente se usa, sino que ese gasto es mucho menor que si fuera propio (economía de escala).



## Agilidad

Fácil acceso a muchos tipos de servicios y tecnologías.  
Agilidad tanto para desplegarlos como para probar nuevas ideas.



## Global

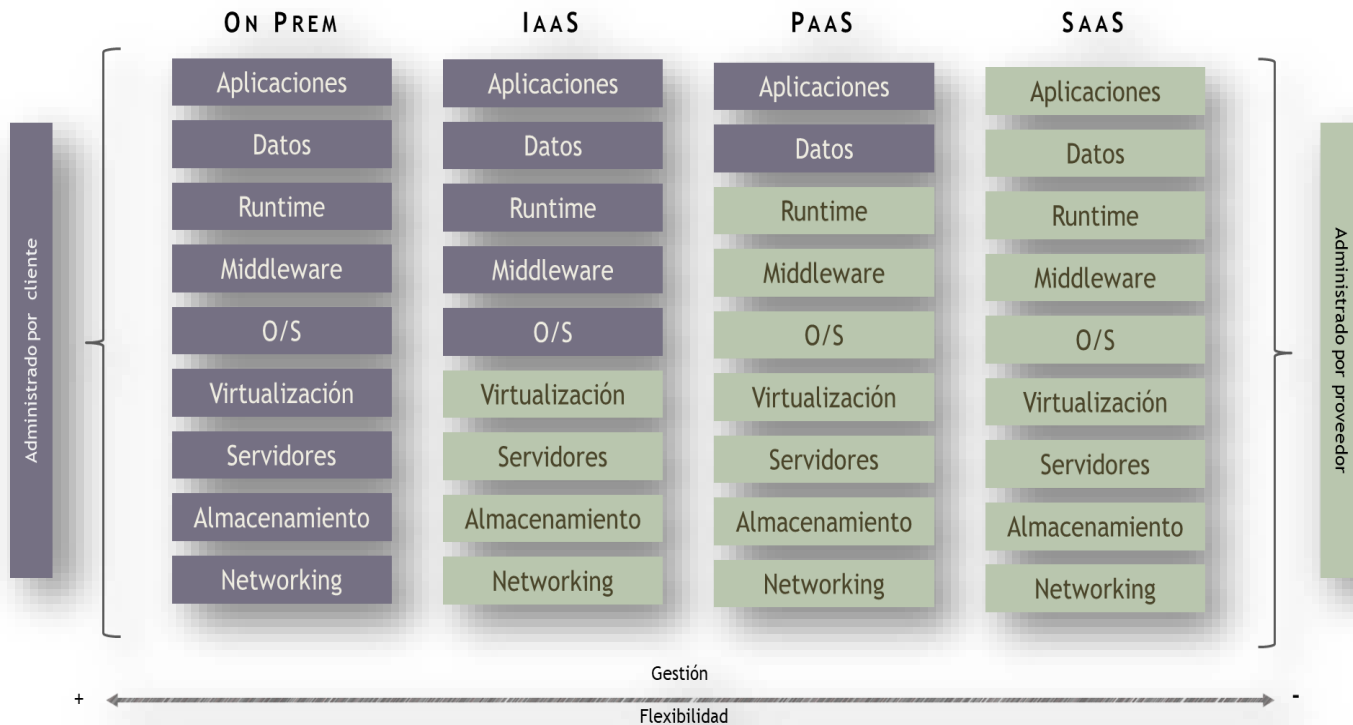
Posibilidad de expandirse a nuevas regiones a lo largo del globo a un golpe de clic, acercando las aplicaciones a los usuarios finales y de ese modo reducir latencias.



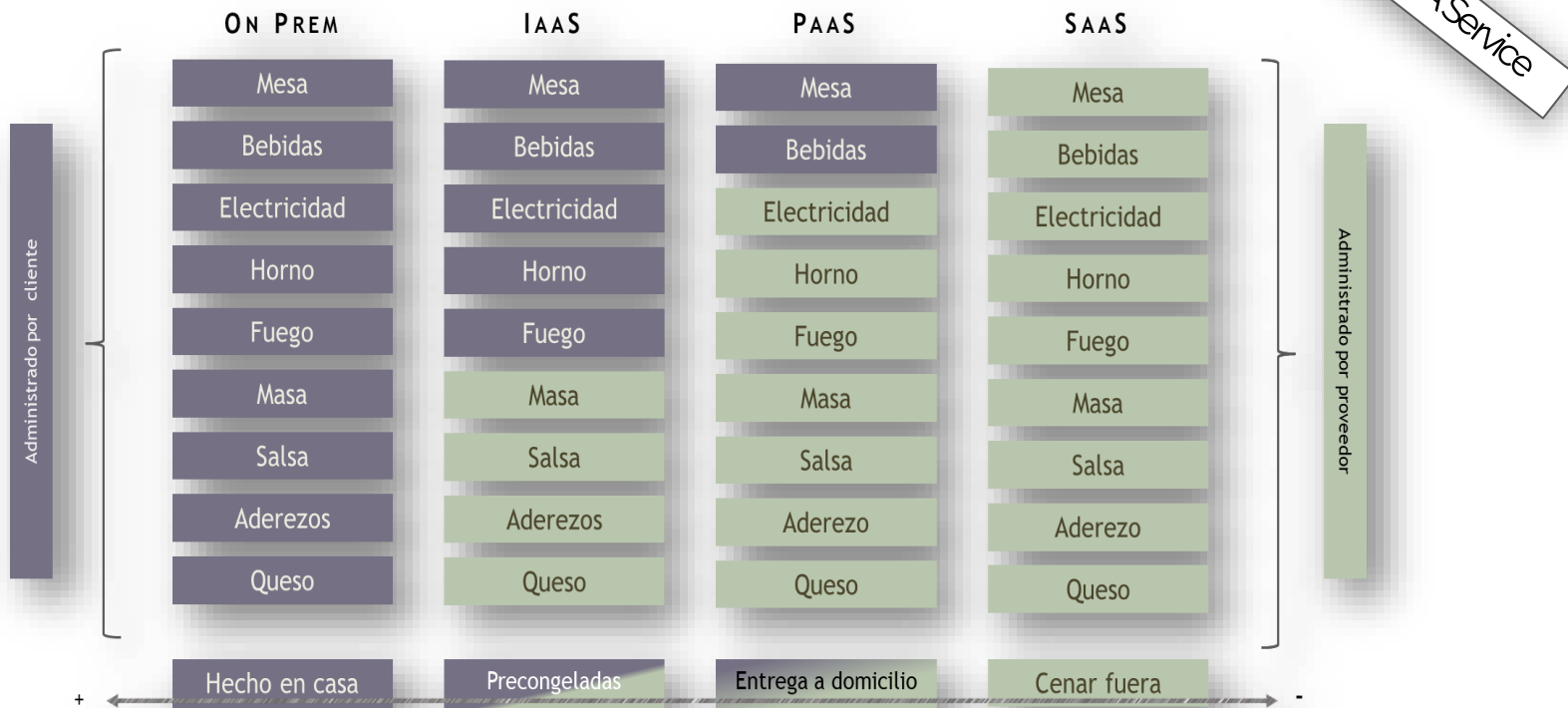
## Elasticidad

Adaptar los recursos a las necesidades de cada momento, sin necesidad de sobreestimar por crecimientos futuros o por picos puntuales.

# Cloud Computing | Tipos según despliegue

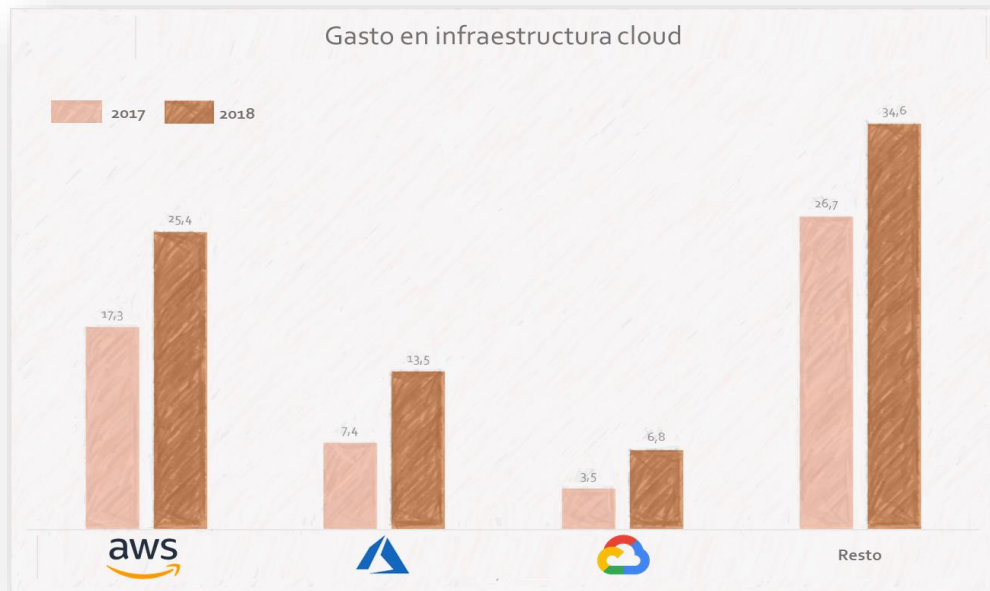


# Cloud Computing | Tipos según despliegue

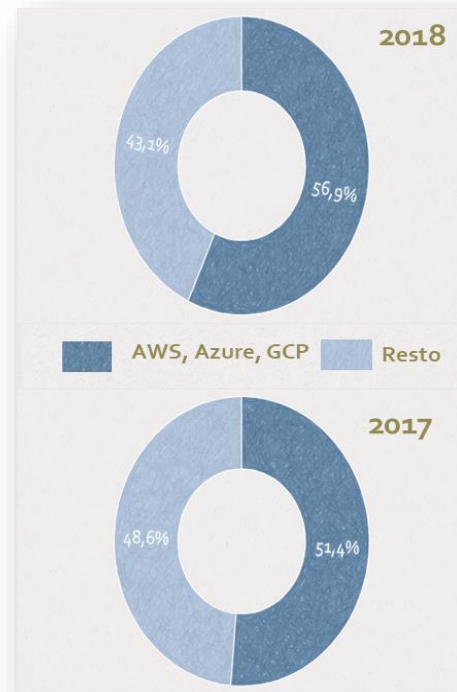




# Cloud Computing | Vendors



Fuente: <https://www.statista.com/chart/7994/cloud-market-share/>



# Lab 03

CLOUD



# Lab 03

- Registrarnos en Azure para usar su free tier: <https://azure.microsoft.com/en-us/free/>
  - Necesitaremos una cuenta Live y un número de tarjeta para registrarnos (no hacen cargo alguno)
- Seguiremos los pasos indicados en <https://docs.microsoft.com/en-us/azure/azure-sql/database/single-database-create-quickstart?tabs=azure-portal>, especificando:

<b>Grupo de recursos</b>	rg_ironhack
<b>Server name</b>	sql<iniciales del alumno>123
<b>Location</b>	(Europe) West Europe
<b>Database</b>	AdventureWorks
<b>User name</b>	azureuser
<b>Password</b>	P@\$w0rd123

- Pulsamos en “Go to resource” cuando haya terminado de desplegarse el servicio.

## Lab 03

- Nos descargamos Azure Data Studio de <https://docs.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15>
- Una vez instalado, lo lanzamos y creamos una nueva conexión hacia el servidor recién desplegado:

<b>Server</b>	sql<iniciales del alumno>123.database.windows.net
<b>Authentication type</b>	SQL Login
<b>User name</b>	azureuser
<b>Password</b>	P@\$\$w0rd123
<b>Database</b>	AdventureWorks

- Pulsamos en “New query” y escribimos la siguiente instrucción:

```
SELECT TOP 20 pc.Name as CategoryName, p.name as ProductName
FROM SalesLT.ProductCategory pc JOIN SalesLT.Product p ON pc.productcategoryid = p.productcategoryid
ORDER BY CategoryName;
```

# Lab 03

- **Challenge 1 – Qué ha comprado cada cliente?**

- Hay que combinar varias tablas para saber qué productos ha comprado cada cliente
- Hay que considerar como mínimo las columnas First Name, Last Name (SalesLT.Customer) y Name (SalesLT.Product), pero la salida debe ser como la imagen siguiente:

	Customer Fullname	Product Name
1	Andrea Thomsen	Rear Brakes
2	Anthony Chor	HL Touring Frame - Blue, 50
3	Anthony Chor	HL Touring Frame - Blue, 54
4	Anthony Chor	HL Touring Frame - Blue, 60
5	Anthony Chor	HL Touring Frame - Yellow, 60
6	Anthony Chor	HL Touring Handlebars
7	Anthony Chor	HL Touring Seat/Saddle
8	Anthony Chor	LL Touring Frame - Yellow, 44
9	Anthony Chor	LL Touring Frame - Yellow, 50

*La imagen no muestra la salida completa*

## Lab 03

- **Challenge 2 – Mostrar la descripción árabe del producto cuyo código es el 710**
  - La cultura árabe tiene la abreviatura 'ar'
  - La salida debe ser como la imagen siguiente:

	Product Model	Description
1	Mountain Bike Socks	...يجفأفها وتعمل كوسائد ملائمة

*La imagen muestra la salida completa*

## Lab 03

- **Challenge 3 (B O N U S) – Total de ventas por producto, ordenado descendientemente**
  - La salida debe ser como la imagen siguiente:

	name	Total Orders
1	Classic Vest, S	10
2	Long-Sleeve Logo Jersey, L	10
3	AWC Logo Cap	9
4	Short-Sleeve Classic Jersey,...	9
5	Short-Sleeve Classic Jersey,...	8
6	Hitch Rack - 4-Bike	8
7	Bike Wash - Dissolver	7
8	Front Brakes	7

*La imagen no muestra la salida completa*

## SQL | Summary

**#1:** Nos hemos iniciado en el cloud computing.

**#2:** Hemos desplegado una base de datos en Azure, el cloud provider de Microsoft.

**#3:** Hemos interactuado con una base de datos más completa y practicado SQL con ella.

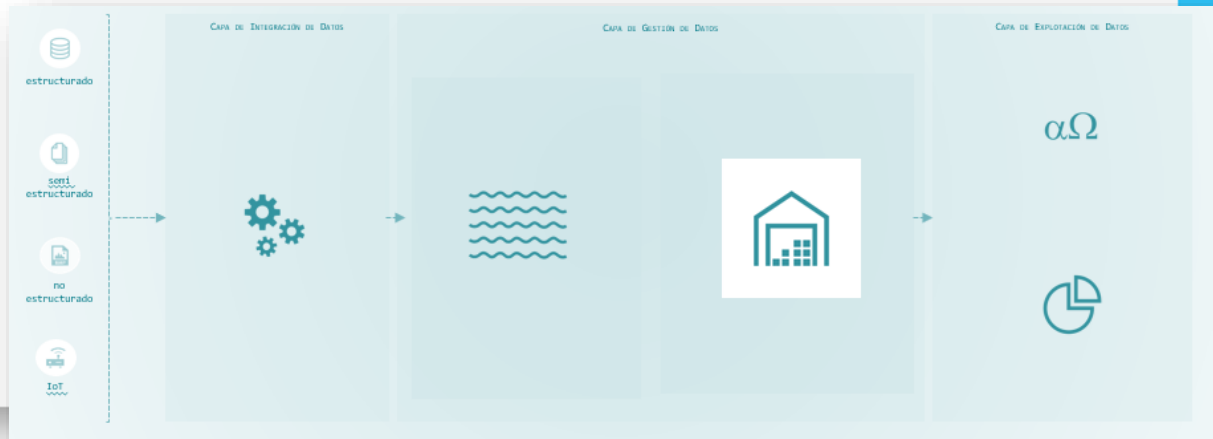




# Mundo analítico

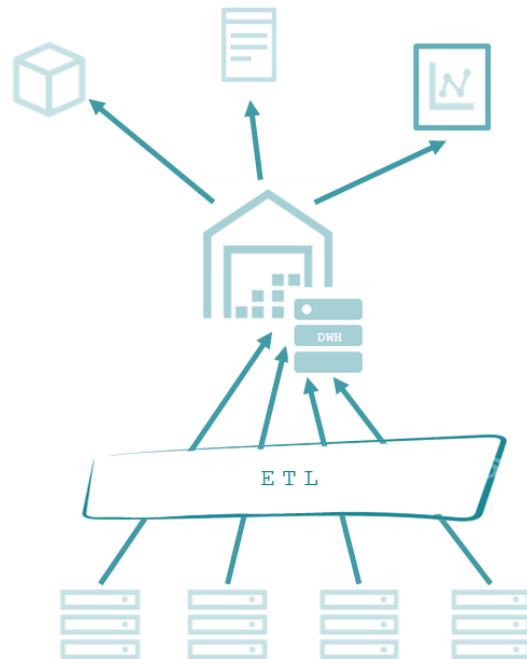
# Analítica en base de datos

- En el último ejercicio del laboratorio aparecía una pequeña función (COUNT) que no hemos visto aun, pero que abre un nuevo mundo en el uso de las bases de datos: el mundo informacional.
- Gracias (en parte) a estas funciones, es posible empezar a extraer información de los datos.
- Pero también gracias a un concepto, igualmente con cierta carga negativa: el data warehouse



# Data Warehouse

- Ese otro repositorio creado para dar soporte a los informes analíticos, destino final de información proveniente de distintas fuentes, es lo que se denomina Data Warehouse
- En otras palabras: es el sitio en donde los responsables se basarán para tomar las **decisiones** de la empresa, puesto que es donde se encuentran persistidos los datos historificados, orientados a negocio y confiables.



# Kimball

Los datos se organizan en hechos e información de referencia

(Bottom-Up) El EDW es un compendio de datamarts departamentales

Agilidad en la construcción

Fácilmente entendible por negocio

El rendimiento es muy bueno

Las herramientas de BI entienden el modelado dimensional



# Inmon

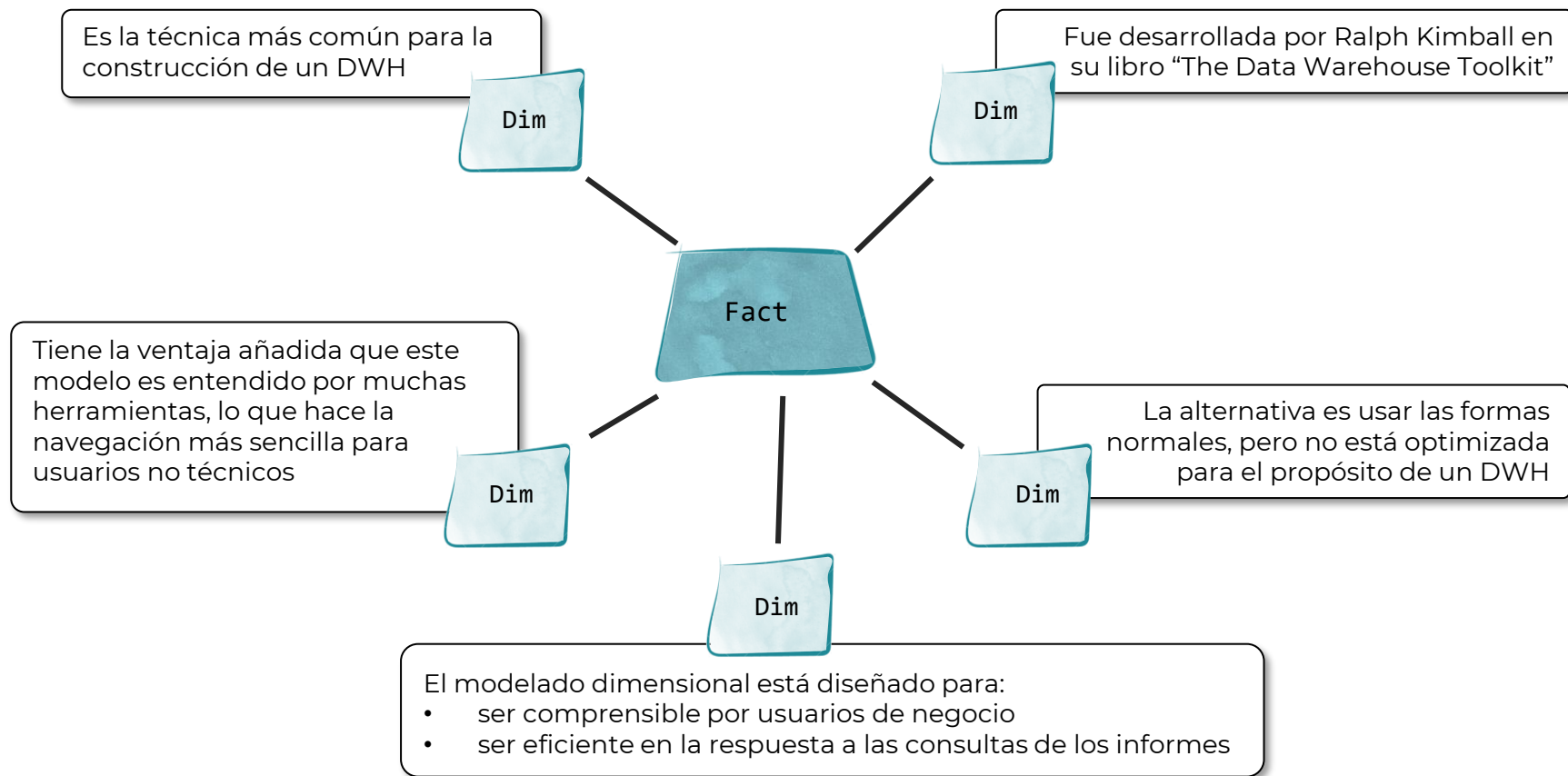
Se siguen las reglas de normalización de bbdd (E.F.Codd) en la construcción del DWH

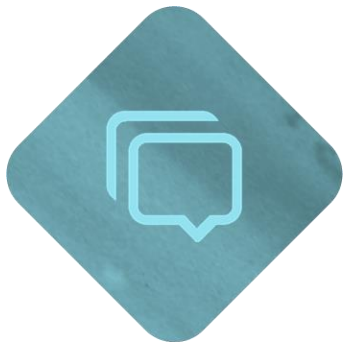
(Top-Down) Primero se crea el EDW y sobre él los datamart departamentales

El EDW es realmente el único punto de verdad del dato

Es relativamente sencillo añadir información (muchas tablas normalizadas)







El concepto más importante en el modelado dimensional es saber dividir los datos en dos categorías:

- Conjuntos de datos delimitados (datos de referencia), también llamados **dimensiones**
- Conjuntos de datos sin límites, también llamados **hechos**



### Hechos

Representan **eventos del negocio** (ventas, envíos, etc.) sobre los que se definen las medidas.

Se implementan físicamente en las **tablas de hechos**, que crecen rápidamente y no se modifican.

Estas tablas solo contienen esas **medidas y claves** que apuntan a las dimensiones



### Dimensiones

Proporcionan el contexto alrededor del evento: el **quién, qué, dónde y cuándo**

Se implementan en **tablas de dimensiones**, no contienen tanta información pero sí suelen modificarse sus datos.

Contienen propiedades que definen la entidad, muchas veces gobernadas por un responsable

## SELECT | Agregaciones

- La implementación de los conceptos que acabamos de ver se apoyan en las **consultas de agregación** para mostrar todo su potencial.
- Hasta ahora nos hemos centrado en manipular y consultar datos, así como aprender las técnicas básicas de modelado para sistemas operacionales.
- Veamos ahora cómo podemos extraer información de esos datos para poder tomar **decisiones informadas**.

## SELECT | Agregaciones

- La cláusula `GROUP BY` permite dividir el resultado en **conjuntos** de filas.
- Normalmente, esos grupos se les usa para realizar algún tipo de **cálculo sobre ellos**.
- Los grupos se determinan en base a las **columnas o expresiones** que se especifican en la cláusula.
- Las expresiones que **no forman parte** de la función de agregado **deben aparecer** en la cláusula `GROUP BY`.
- Recordad que la cláusula `WHERE` elimina las filas **antes** de que `GROUP BY` las ponga en el grupo correspondiente.



### GROUP BY

Divide el conjunto de resultados en conjuntos de filas

```
GROUP BY { <column_expression> | GROUPING SETS (<grouping_set> [,...]) }[,...]
```

```
SELECT c1, COUNT(*) AS 'total' FROM dbo.myTable GROUP BY c1;
```



## SELECT | Agregaciones

- Pero la cláusula GROUP BY necesita verse acompañada por una **función de agregación** que permita realizar **cálculos** sobre cada uno de los grupos que nos devuelve.
- Funciones de agregación hay muchas, incluso cada motor implementa las suyas propias, pero como norma general podemos encontrarnos con las siguientes

SUM ([ALL   DISTINCT] <expression>)	Devuelve la suma de todos los valores del grupo
COUNT ([ALL   DISTINCT] <expression>   *)	Devuelve el número de elementos encontrados en el grupo
MAX ([ALL   DISTINCT] <expression>)	Devuelve el valor máximo de los valores del grupo
MIN ([ALL   DISTINCT] <expression>)	Devuelve el valor mínimo de los valores del grupo
AVG ([ALL   DISTINCT] <expression>)	Devuelve la media de todos los valores del grupo
VAR ([ALL   DISTINCT] <expression>)	Devuelve la varianza de todos los valores del grupo
STDEV ([ALL   DISTINCT] <expression>)	Devuelve la desviación estándar de todos los valores del grupo

## SELECT | Agregaciones

- La cláusula GROUP BY es **flexible** para incluir varias columnas o expresiones por las que agrupar.
- Pero al final, las funciones de agregado **aplican a los grupos** que conforman la unión de esas columnas o expresiones
- Sin embargo, también tenemos disponible la opción GROUPING SETS que nos permite especificar **diferentes grupos** en la misma consulta.
- Dicho de otro modo: es como si estuviéramos ejecutando **varias instrucciones GROUP BY independientes** en las que solo cambiara las columnas o expresiones por las que agrupamos los resultados.

```
SELECT col1, col2, COUNT(*) AS total
FROM t1
GROUP BY GROUPING SETS ((col1), (col1, col2))
```

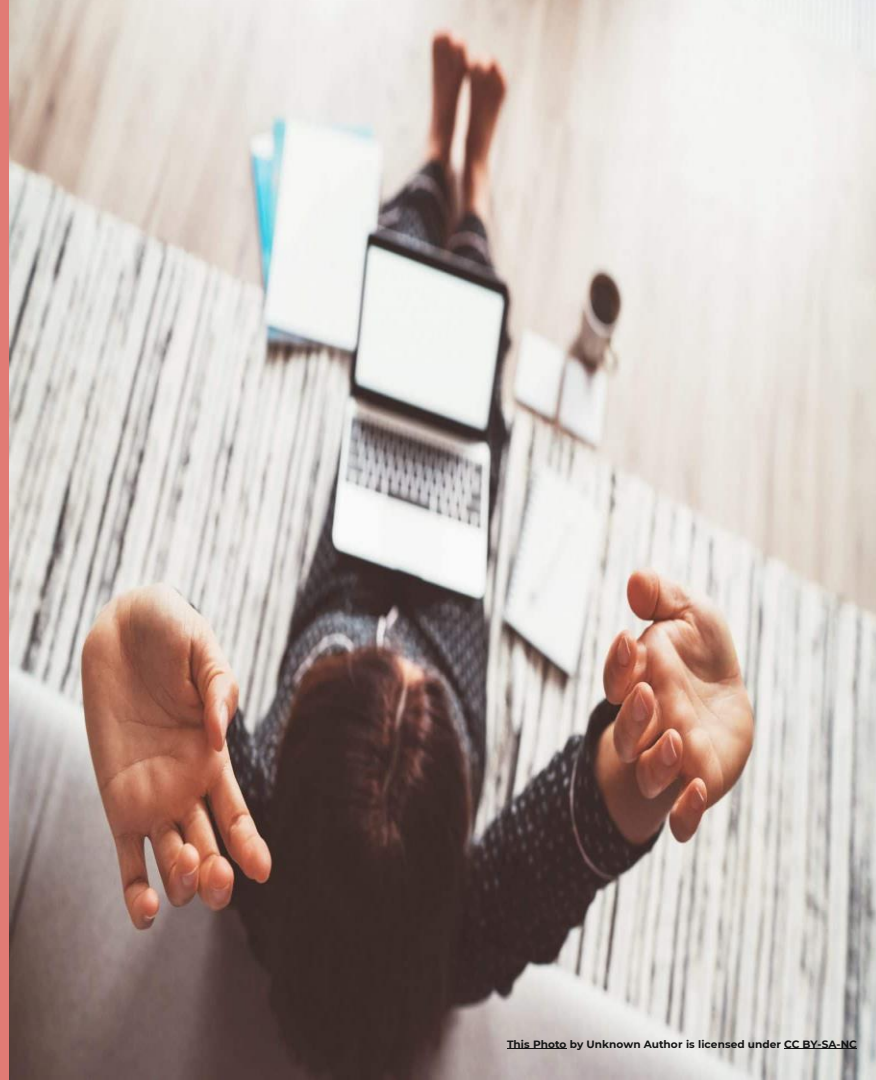
## SELECT | Agregaciones

- La cláusula HAVING nos permite **filtrar grupos** que no cumplan las condiciones establecidas.
- HAVING es a GROUP BY lo que WHERE es al FROM.
- Como no se filtran registros individuales sino grupos, siempre se usa con una **función de agregación**.

```
SELECT col1, col2, COUNT(*) AS total
FROM t1
GROUP BY col1, col2
HAVING COUNT(*) > 10
```

# Lab 04

## AGREGACIONES



# Lab 04

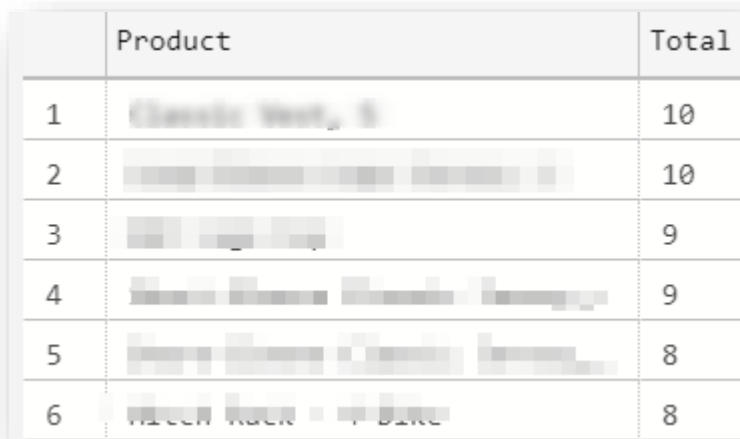
- Conectamos a nuestra base de datos con Azure Data Studio

<b>Server</b>	sql<iniciales del alumno>123.database.windows.net
<b>Authentication type</b>	SQL Login
<b>User name</b>	azureuser
<b>Password</b>	P@\$\$w0rd123
<b>Database</b>	AdventureWorks

- Vamos a hacer unos retos para practicar con las agrupaciones

# Lab 04

- **Challenge 1 – Qué producto ha sido el más vendido?**
  - Por “más vendido” nos referimos a número de tickets, no unidades vendidas
  - Para encontrarlo, habrá que ordenar por el número de ventas de forma descendente.



	Product	Total
1	Classic West, S	10
2	Long Island Sound, S	10
3	Long Island Sound, S	9
4	Long Island Sound, S	9
5	Long Island Sound, S	8
6	Long Island Sound, S	8

*La imagen no muestra la salida completa*

# Lab 04

- **Challenge 2 – Cuántas unidades han sido vendidas por categoría y producto?**
  - Ordena el resultado por Categoría y Producto.

	Category	Product	Total Qty
1	Bike Racks	Hitch Rack - 4-Bike	32
2	Bottles and Cages	Water Bottle - 30 oz.	54
3	Bottom Brackets	HL Bottom Bracket	15
4	Bottom Brackets	LL Bottom Bracket	7
5	Brakes	Front Brakes	12
6	Brakes	Rear Brakes	1
7	Cups	AWC Long Cup	52

La imagen no muestra la salida completa

# Lab 04

- **Challenge 3 – Cuántas unidades han sido vendidas por categoría, y por categoría-producto?**
  - Ordena el resultado por Categoría y Producto.
  - Hay que hacerlo en una sola instrucción.

	Category	Product	Total Qty
1	Bike Racks	NULL	32
2	Bike Racks	Hitch Rack - 4-Bike	32
3	Brakes	NULL	13
4	Caps	NULL	52
5	Caps	AWC Logo Cap	52
6	Derailleurs	NULL	21
7	Gloves	NULL	57

La imagen no muestra la salida completa



## Lab 04

- **Challenge 4 (B O N U S) – De la consulta anterior, descarta los grupos que tengan menos de 8 tickets de venta**
  - 1 ticket de venta es 1 registro de la tabla SalesLT.SalesOrderHeader
  - Ordena el resultado por Categoría y Producto.

	Category	Product	Total Qty
1	Bike Racks	NULL	32
2	Bike Racks	Hitch Rack - 4-Bike	32
3	Brakes	NULL	13
4	Caps	NULL	52
5	Caps	AWC Logo Cap	52
6	Handlebars	NULL	27
7	Helmets	NULL	124

La imagen no muestra la salida completa

## SELECT | aggregate window functions

- Hemos visto la potencia de **GROUP BY** combinado con funciones de agregado para obtener insights de los datos.
- El problema es que **reduce** el número de registros devueltos, no tenemos forma (fácil) de incluir algún dato que no esté en la cláusula **GROUP BY**.
- Pero existe también la posibilidad de usar **funciones de ventana**, las cuales también operan sobre grupos de registros, pero **no reducen** el conjunto de resultados final.



### OVER

Realiza un cálculo sobre un conjunto de datos sin limitar los resultados

```
OVER ( [ <partition_by_clause> ] [<order_by_clause>] [<range_clause>])
```

```
SELECT c1, c2, COUNT(*) OVER (PARTITION BY c1) 'total c1' FROM dbo.myTable;
```

## SELECT | aggregate window functions

Tenemos por tanto las dos opciones: tener en el resultado los totales por grupo, pero también **el detalle y un cálculo** según la agrupación que nos interese.

<code>SUM ([ALL] &lt;expression&gt;) OVER ([ &lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Calcula la suma de las particiones encontradas
<code>COUNT ([ALL] { &lt;expression&gt;   * }) OVER ([ &lt;partition_by_clause&gt;])</code>	Calcula el número de elementos encontrados en la partición
<code>MAX ([ALL] &lt;expression&gt;) OVER (&lt;partition_by_clause&gt; [ &lt;order_by_clause&gt;])</code>	Calcula el valor máximo de las particiones encontradas
<code>MIN ([ALL] &lt;expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Calcula el valor mínimo de las particiones encontradas
<code>AVG ([ALL   DISTINCT] &lt;expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Calcula la media de las particiones encontradas
<code>VAR ([ALL   DISTINCT] &lt;expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Calcula la varianza de las particiones encontradas
<code>STDEV ([ALL   DISTINCT] &lt;expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Calcula la desviación estándar de las particiones encontradas

## SELECT | ranking window functions

- La cláusula OVER nos abre un **mundo de posibilidades** nuevos dentro de SQL, porque junto con ella aparecen otras funciones no tan conocidas.
- Un tipo de estas nuevas funciones son las de **ranking**.

`ROW_NUMBER() OVER ([PARTITION BY <partition_by_clause>] ORDER BY <order_by_clause>)`

Devuelve un número secuencial para cada una de las filas dentro de la partición especificada

`RANK() OVER ([PARTITION BY <partition_by_clause>] ORDER BY <order_by_clause>)`

Es igual a ROW\_NUMBER, pero RANK devuelve el mismo número en caso de ser el mismo valor, saltando el número para el siguiente registro.

`DENSE_RANK() OVER ([PARTITION BY <partition_by_clause>] ORDER BY <order_by_clause>)`

Es igual a RANK, pero sin saltos de número en caso de empate.

`NTILE( <integer_expression> ) OVER ([PARTITION BY <partition_by_clause>] ORDER BY <order_by_clause>)`

Distribuye las filas en una partición ordenada en el número especificado de grupos

## SELECT | analytical window functions

- Otras funciones de ventana muy útiles son las **analíticas**, las cuales calculan un **valor agregado** basado en un grupo de filas.
- La diferencia con las que conocemos es que pueden devolver **varias filas** dentro de cada grupo.
- Se usan en escenarios de moving average, running totals, porcentajes... dentro de un grupo

<code>LAG(&lt;scalar_expression&gt; [, &lt;offset_expression&gt;] [, &lt;default_expression&gt;]) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Accede al valor de una fila previa del conjunto de resultados dentro de la partición especificada
<code>LEAD(&lt;scalar_expression&gt; [, &lt;offset_expression&gt;] [, &lt;default_expression&gt;]) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt;)</code>	Accede al valor de una fila subsiguiente del conjunto de resultados dentro de la partición especificada
<code>FIRST_VALUE(&lt;scalar_expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt; [rows_range_clause])</code>	Devuelve el primer valor de un conjunto ordenado de valores
<code>LAST_VALUE(&lt;scalar_expression&gt;) OVER ([&lt;partition_by_clause&gt;] &lt;order_by_clause&gt; [rows_range_clause])</code>	Devuelve el último valor de un conjunto ordenado de valores

*\* Existen más funciones, pero son para escenarios más específicos*

# Lab 05

## WINDOW FUNCTIONS



# Lab 05

- Conectamos a nuestra base de datos con Azure Data Studio

<b>Server</b>	sql<iniciales del alumno>123.database.windows.net
<b>Authentication type</b>	SQL Login
<b>User name</b>	azureuser
<b>Password</b>	P@\$\$w0rd123
<b>Database</b>	AdventureWorks

- Vamos a comprobar los resultados que nos devuelven algunas funciones de ventana

# Lab 05

- **Challenge 1 – Revisa la salida de la siguiente instrucción**
  - Fíjate en los resultados para entender mejor la potencia de estas funciones

```
SELECT p.ProductID, pc.Name AS 'Category', p.Name AS 'Product', p.[Size]
      , ROW_NUMBER() OVER (PARTITION BY p.ProductCategoryID ORDER BY p.Size) AS 'Row Number Per Category & Size'
      , RANK() OVER (PARTITION BY p.ProductCategoryID ORDER BY p.Size) AS 'Rank Per Category & Size'
      , DENSE_RANK() OVER (PARTITION BY p.ProductCategoryID ORDER BY p.Size) AS 'Dense Rank Per Category & Size'
      , NTILE(2) OVER (PARTITION BY p.ProductCategoryID ORDER BY p.Name) AS 'NTile Per Category & Name'

      , SUM(p.StandardCost) OVER() AS 'Standard Cost Grand Total'
      , SUM(p.StandardCost) OVER(PARTITION BY p.ProductCategoryID) AS 'Standard Cost Per Category'

      , LAG(p.Name, 1, '-- NOT FOUND --') OVER(PARTITION BY p.ProductCategoryID ORDER BY p.Name) AS 'Previous Product Per Category'
      , LEAD(p.Name, 1, '-- NOT FOUND --') OVER(PARTITION BY p.ProductCategoryID ORDER BY p.Name) AS 'Next Product Per Category'

      , FIRST_VALUE(p.Name) OVER(PARTITION BY p.ProductCategoryID ORDER BY p.Name) AS 'First Product Per Category'
      , LAST_VALUE(p.Name) OVER(PARTITION BY p.ProductCategoryID ORDER BY p.Name) AS 'Last Product Per Category'
FROM SalesLT.Product AS p
  INNER JOIN SalesLT.ProductCategory AS pc ON p.ProductCategoryID = pc.ProductCategoryID
ORDER BY pc.Name, p.Name
```



## SQL | Summary

**#1:** Nos hemos iniciado en el mundo informacional.

**#2:** Hemos conocido cómo calcular agregados.

**#3:** Hemos visto la versatilidad de las funciones de ventana.

# Bibliografía| Recursos

- **T-SQL Querying (Developer Reference)**, Itzik Ben-Gan, 2015
- **SQL Cookbook**, Anthony Molinaro, 2005
- **SQL: the complete reference**, Paul Weinberg, 2009
- **Learning SQL**, Alan Beaulieu, 2014
- **Data Analysis Using SQL and Excel**, Gordon S. Linoff, 2015
- **The Data Warehouse Toolkit**, Ralph Kimball, 2013



Fuente: <https://www.amazon.com/Computer-Programmer-Funny-Joke-T-shirt/dp/B078V378PP>



**THANKS !**