SQL

En el contexto de las bases de datos, los lenguajes se dividen en varios tipos:

- Lenguajes de definición de datos: nos permiten expresar la estructura y las restricciones de nuestro modelo de datos
- Lenguajes de manipulación de datos: nos permiten ingresar, modificar, eliminar y consultar datos en nuestro modelo
- Lenguajes de control de datos: manejan cuestiones vinculadas con los permisos de acceso a los datos

SQL es un lenguaje de definición de datos (DDL) y de manipulación de datos (DML)

- Tiene gramática libre de contexto (CFG). Su sintaxis puede ser descripta a través de reglas de producción
- La notación más común que usa es la notación de Backus-Naur Form
- En un modelo relacional, una tupla no puede estar repetida en una relación. En SQL se permite que una fila esté repetida muchas veces en una tabla, y eso se conoce como **MULTISET** o **BAG OF TUPLES**

Ejemplo de especificación

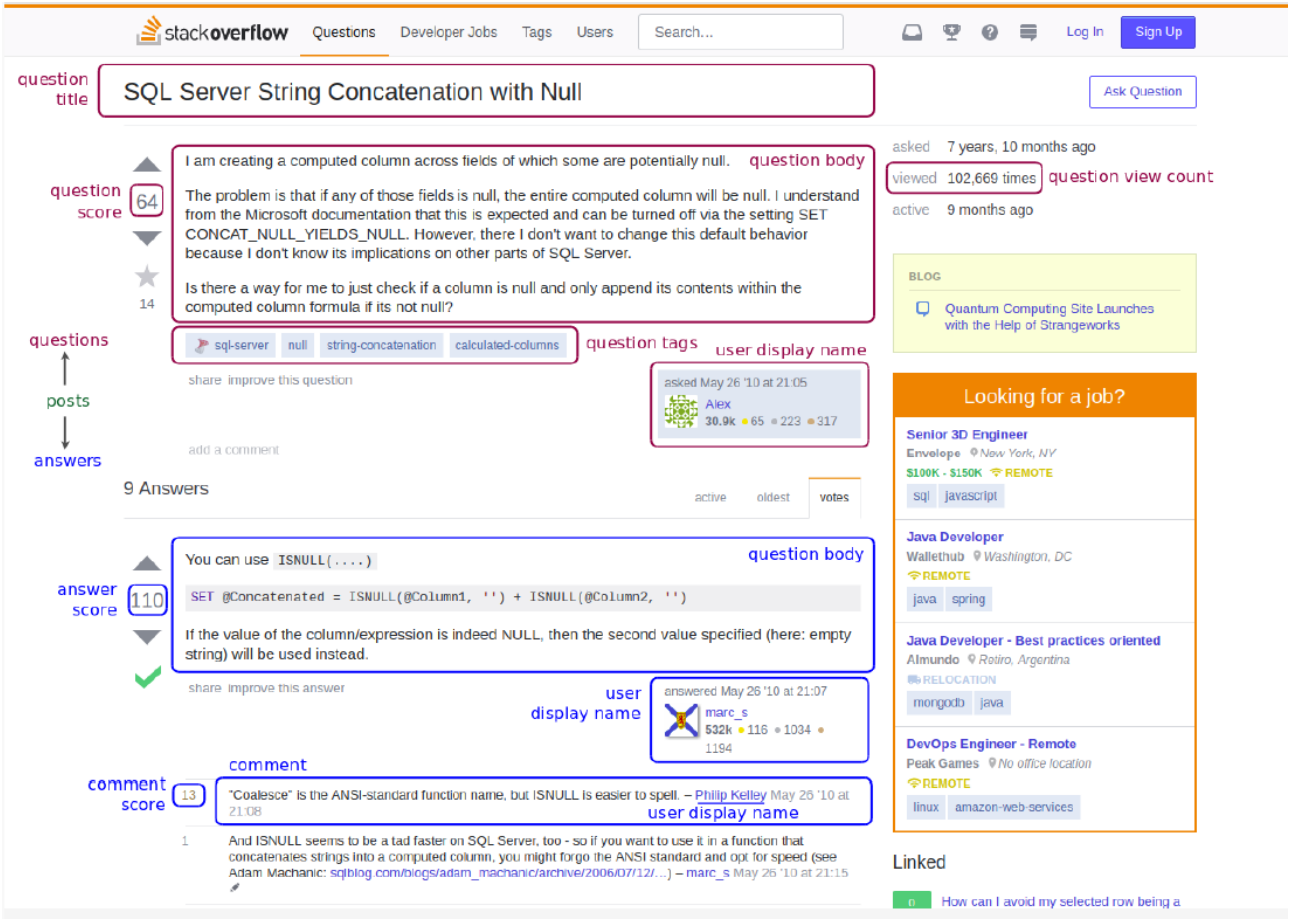
```
<query specification> ::=
  SELECT [ <set quantifier> ] <select list> <table expression>

<set quantifier> ::=
    DISTINCT
  | ALL

 <select list> ::=
    <asterisk>
  | <select sublist> [ { <comma> <select sublist> }... ]

 <table expression> ::=
  <from clause>
      [ <where clause> ]
      [ <group by clause> ]
      [ <having clause> ]
      [ <window clause> ]
```

Esta clase vamos a usar: https://data.stackexchange.com/stackoverflow/query/new



Modelo conceptual simplificado

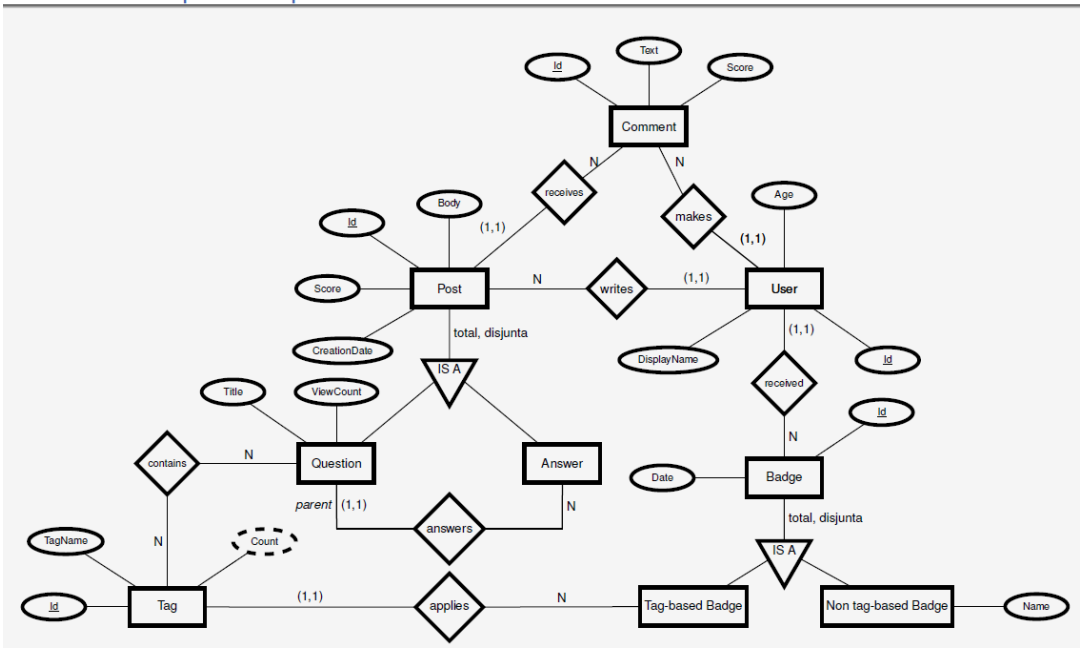
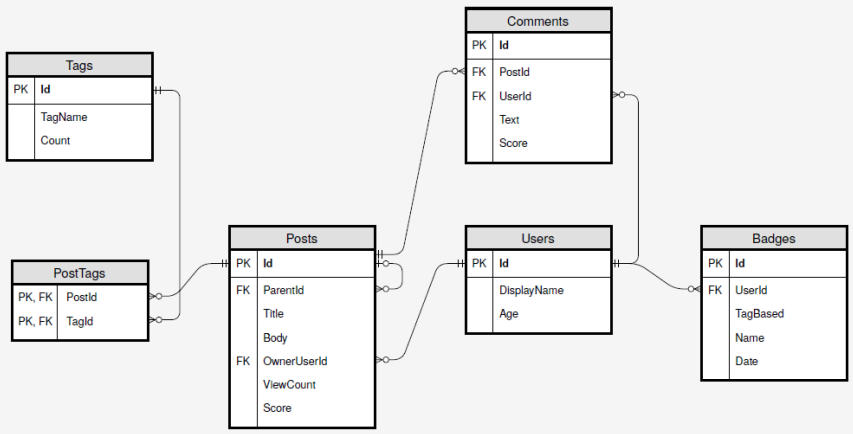


Diagrama de tablas simplificado



Esta es una visualización comunmente utilizada del modelo lógico relacional. No estudiaremos estos diagramas en el curso, pero aquí utilizamos uno para ilustrar la estructura de la base de datos de Stack Exchange. Encontrarán un diagrama más completo en https://meta.stackexchange.com/questions/250396/database-diagram-of-stack-exchange-model.

También, en el panel de la derecha del Data Explorer se muestra la descripción completa de cada tabla.

Definición de datos en SQL

CREATE SCHEMA

Nos permite crear una nueva base de datos dentro de nuestro SGBD

Su sintaxis es:

```
CREATE SCHEMA nombre_esquema [ AUTHORIZATION AuthId ];
```

Ejemplo:

```
CREATE SCHEMA empresa [ AUTHORIZATION mbeiro ];
```

- La opción AUTHORIZATION identifica quién será el dueño del esquema.
- Puedo tener varios esquemas de BDD en un mismo entorno de SQL. Los esquemas se agrupan en colecciones que se llaman ***catálogos***
- Todo catálogo contiene un esquema llamado INFORMATION_SCHEMA, que describe a todos los demás esquemas contenidos en él.

Tipos de variables en SQL

- Tipos numéricos estándar:
  - **INTEGER**: Tipo entero. Abreviado **INT**.
  - **SMALLINT**: Tipo entero pequeño.
  - **FLOAT(n)**: Tipo numérico aproximado. *n* indica la precisión en bits.
  - **DOUBLE PRECISION**: Tipo numérico aproximado de alta precisión. →En Postgres, double precision f-p, IEEE 754 (n=53, e=11)
  - **NUMERIC(i,j)**: Tipo numérico exacto. Permite especificar la *precisión* (i) y la *escala* (j) en dígitos.
    (*precisión*: dígitos totales; *escala*: dígitos decimales)
- Strings: Se delimitan con comillas *simples* (').
  - **CHARACTER(n)**: De longitud fija. Abreviado **CHAR(n)**. →default, n=1
  - **CHARACTER VARYING(n)**: De longitud variable. Abrev. **VARCHAR(n)**.
- Fecha y hora:
  - **DATE**: Precisión de días. Se ingresa como string con formato YYYY-MM-DD. →(ISO 8601)
  - **TIME(i)**: Precisión de hasta microsegundos. Se ingresa como string con formato HH:MM:SS.[0-9]^{i} (ISO 8601). Tantos dígitos decimales como **i**.
  - **TIMESTAMP(i)**: Combina un **DATE** y un **TIME(i)**.

Numeric es para un tipo de dato numérico al que le quiera definir una precisión y/o escala en particular. Ej: quiero una precisión de 3 decimales.

- Booleanos *(opcional)*:
  - **BOOLEAN**: **TRUE**, **FALSE** o **UNKNOWN**. Se emplea *lógica de tres valores*.
- Otros tipos:
  - **CLOB**: **(Character Large Object)** Para documentos de texto de gran extensión.
  - **BLOB**: **(Binary Large Object)** Para archivos binarios de gran extensión.
- Tipos definidos por el usuario:

```
CREATE DOMAIN NOMBRE_DOMINIO AS TIPO_BASICO;
```

  - Ejemplo:

```
CREATE DOMAIN CODIGO_PAIS AS CHAR(2);
```

  - Facilita la realización de cambios futuros en el diseño.

Los tipos propios no suelen usarse mucho.

CREATE TABLE

Nos permite definir la estructura de una tabla

```
CREATE TABLE Persona (
dni_persona INT PRIMARY KEY,
nombre_persona VARCHAR(255),
fecha_nacimiento DATE);
```

```
CREATE TABLE HijoDe (
dni_hijo INT,
dni_padre INT,
PRIMARY KEY (dni_hijo, dni_padre),
FOREIGN KEY (dni_hijo) REFERENCES Persona(dni_persona),
FOREIGN KEY (dni_padre) REFERENCES Persona(dni_persona));
```

Creación de una tabla: estructura general

CREATE TABLE T_1 (
A_1 $type_1$ **[NOT NULL] [CHECK** $condition_1$**] [PRIMARY KEY]**,
A_2 $type_2$ **[NOT NULL] [CHECK** $condition_2$**] [PRIMARY KEY]**,
...,
A_n $type_n$ **[NOT NULL] [CHECK** $condition_n$**] [PRIMARY KEY]**,
[PRIMARY KEY $(A_{p_1}, A_{p_2}, ..., A_{p_k})$**]**
{**UNIQUE** $(A_{u_1}, A_{u_2}, ..., A_{u'_k})$} ...
{**FOREIGN KEY** $(A_{h_1}, A_{h_2}, ..., A_{h'_k})$ **REFERENCES** $T_2(B_{f_1}, B_{f_2}, ..., B_{f'_k})$
[ON DELETE SET NULL | RESTRICT | CASCADE | SET DEFAULT **]**
[ON UPDATE SET NULL | RESTRICT | CASCADE | SET DEFAULT **]**}...);

- Las columnas también pueden ser configuradas con valores por defecto (DEFAULT) o autoincrementales (AUTO_INCREMENT)
- Se puede definir el tipo de una columna
- Se puede restringir la posibilidad que una columna tome valores nulos (**NOT NULL**)
- Se puede restringir aun mas el conjunto de valores posibles a través de un “chequeo en forma dinámica”
- UNIQUE: cierta columna/combinaciones de columnas deben ser únicas; como una especie de clave candidata: no se pueden repetir en != filas el valor de ciertas columnas.

```
CUIT_tipo INT CHECK (CUIT_tipo=20) OR (CUIT_tipo=23) OR...
```

Trabajando con restricciones en SQL

Restricciones de unicidad

- Se indica la clave primaria con **PRIMARY KEY**. Si esta compuesta de una única columna, puede indicarse a continuación del tipo.
- Con la palabra clave **UNIQUE** se indica que una columna o conjunto de columnas no puede estar repetido en dos filas distintas (es una manera de identificar claves candidatas)
- **Por más que SQL NO obligue a definir una PK, siempre debemos hacerlo.**

Restricciones de integridad

- **Integridad de entidad:** la clave primaria de una tabla nunca debe ser **NULL**
- **Integridad referencial:** las claves foráneas se especifican con **FOREIGN KEY… REFERENCES**

Manipulación de datos en SQL

SELECT FROM WHERE

- El esquema básico de una consulta en SQL es:

SELECT $A_1, A_2, ..., A_n$
FROM $T_1, T_2, ..., T_m$
[**WHERE** *condition*];

- En donde $A_1, A_2, ..., A_n$ es una lista de nombres de columnas, $T_1, T_2, ..., T_m$ es una lista de nombres de tablas, y *condition* es una condición.
- Es el análogo a la siguiente expresión del álgebra relacional:

$$\pi_{A_1,A_2,...,A_n}(\sigma_{condition}(T_1 \times T_2 \times ... \times T_m))$$

- Con la diferencia de que la proyección en SQL no elimina filas repetidas.

La condición WHERE es optativa.

WHERE

- Las condiciones admitidas dentro de la cláusula ***WHERE*** son:
  - $A_i \odot A_j$
  - $A_i \odot c$, con $c \in dom(A_i)$
  - A_i [NOT] LIKE p, en donde A_i es un *string* y p es un *patrón*
  - $(A_i, A_{i+1}, ...)$ [NOT] IN m, en donde m es un *set* o un *multiset*
  - A_i [NOT] BETWEEN a AND b, con $a, b \in dom(A_i)$
  - A_i IS [NOT] NULL
  - EXISTS t, en donde t es una tabla
  - $A_i \odot [ANY|ALL]$ t, en donde t es una tabla
  - En donde $\odot$ debe ser un operador de comparación:
    - $=, <>$
    - $>, >=, <, <=$ (para columnas cuyos dominios están ordenados)
- Varias condiciones atómicas pueden unirse con operadores lógicos **AND, OR, NOT**

Pattern matching con WHERE

- La cláusula **WHERE** también permite condiciones de reconocimiento de patrones para columnas que son *strings*.

```
...WHERE attrib LIKE pattern;
```

- Se acepta como patrón una secuencia de caracteres delimitada por comillas (’), combinada con los siguientes caracteres especiales en su interior:
  - _ (representa un caracter arbitrario)
  - % (representa cero o más caracteres arbitrarios)
  - Si se necesita un _ ó un % literal en el patrón, se debe *escapear*

FROM

- En la cláusula ***FROM*** se pueden usar alias para las tablas

```
...FROM Persona p...
...FROM Persona AS p...
```

- Cuando se selecciona una columna, si la misma es ambigua, **se debe** indicar el nombre de la tabla/su alias

```
SELECT business.id
FROM business...

SELECT b.id
FROM business b...
```

- Si una tabla se usa dos veces en una misma clausula From, se necesita usar un alias si o si para distinguir las columnas. También se puede renombrar las columnas

```
..FROM Persona AS p1(dni1, nombre1), Personas AS p2(dni2, nombre2)..
```

SELECT

Se usa para decir cuales son las columnas que quiero que me devuelva. En realidad, se ponen **expresiones.** Una expresión puede ser un nombre de columna, o tmb una expresión matemática. Como por ej: SELECT Id * 100, o SELECT UPPER(p1.nombre)

También es posible cambiar el nombre de las columnas en el resultado:

```
SELECT p1.nombre AS NPadre, p2.nombre AS NHijo...
```

Y realizar operaciones entre las columnas en el resultado:

```
SELECT Producto.precio * 0.90 AS precioDescontado...
```

- Las operaciones permitidas son:
  - +, -, *, / (columnas numéricas)
  - || (concatenación de *strings*)
  - +, - (sumar a una fecha, hora o timestamp un intervalo)
  - LN, EXP, POWER, LOG, SQRT, FLOOR, CEIL, ABS, .. (no son Core-SQL)

Funciones de agregación

- Se puede aplicar una función de agregación a cada una de las columnas del resultado. Las más habituales son:

  - **SUM(A)**
    - Suma los valores de la columna *A* de todas las filas
  - **COUNT([DISTINCT] A | *)**
    - **COUNT(A)** cuenta la cantidad de filas con valor no nulo de *A*.
    - **COUNT(DISTINCT A)** cuenta la cantidad de valores distintos de *A*, sin contar el valor nulo.
    - **COUNT(*)** cuenta la cantidad de filas en el resultado.
  - **AVG(A)**
    - Calcula el promedio de los valores de *A*, descartando los valores nulos.
  - **MAX(A)**
    - Sólo para dominios ordenados.
  - **MIN(A)**
    - Sólo para dominios ordenados.

En general el count* tiene mejor performande que el count A

En estos casos, el resultado colapsa a una única fila.

Ejemplos:

1. Cuente la cantidad de usuarios que existen en la base de datos

```
SELECT COUNT(*)
FROM Users;
```

2. Cuente la cantidad de *posts* que son preguntas

```
SELECT COUNT(*)
FROM Posts p
WHERE p.ParentId IS NULL;
```

Omisión de la selección y de la proyección, duplicados

- La condición de selección puede omitirse si no se escribe la cláusula WHERE
- La proyección sobre un subconjunto de columnas puede omitirse escribiendo **SELECT ***
- La palabra clave **DISTINCT** después de la clausula **SELECT** elimina los duplicados en el resultado.

Ejemplos:

3. Liste los nombres de *badges* que otorga StackOverflow y que no están basados en tags

```
SELECT DISTINCT Name
FROM Badges
WHERE TagBased=0;
```

4. Liste los *tags* que utilizó el usuario ‘Jon Skeet’

```sql
SELECT DISTINCT t.TagName
FROM Users u, Posts p, PostTags pt, Tags t
WHERE pt.PostId=p.Id AND pt.TagId=t.Id
AND p.OwnerUserId=u.Id AND u.DisplayName='Jon Skeet'
```

JOIN

- Es el operador que usa SQL para implementar operaciones de junta

- Junta theta ($\bowtie$)

```
...FROM R INNER JOIN S ON condition...
...FROM R INNER JOIN S USING(attribute)...
```

- Junta natural ($*$)

```
...FROM R NATURAL JOIN S...
```

  - Al igual que en el álgebra relacional, los nombres de las columnas de junta deben coincidir en ambas tablas.

- Junta externa ($\overset{\bowtie}{}$, $\overset{\bowtie}{}$, $\overset{\bowtie}{}$)

```
...FROM R LEFT OUTER JOIN S ON condition...
...FROM R RIGHT OUTER JOIN S ON condition...
...FROM R FULL OUTER JOIN S ON condition...
```

Se hace un join de a pares, y la condición se coloca después del ON.

Natural join: se fija si las tablas tienen cols con el mismo nombre.

Ejemplo:

5. Utilizando la junta theta, encuentre los posts que utilizan conjuntamente los tags ‘relational’ y ‘entity-relationship’, indicando el id del post, el titulo y la cantidad de vistas.

```sql
SELECT DISTINCT p.Id, p.Title, p.ViewCount
FROM (((Tags t1 INNER JOIN PostTags pt1 ON t1.Id=TagId)
     INNER JOIN Posts p ON pt1.PostId=p.Id)
     INNER JOIN PostTags pt2 ON p.Id=pt2.PostId)
     INNER JOIN Tags t2 ON pt2.TagId=t2.Id
WHERE t1.TagName='relational'
AND t2.TagName='entity-relationship';
```

OPERACIONES DE CONJUNTOS

Unión ($\cup$)

```
...R UNION [ALL] S...
```

Intersección ($\cap$)

```
...R INTERSECT [ALL] S...
```

Diferencia ($-$)

```
...R EXCEPT [ALL] S...
```

- Las tablas R y S pueden provenir a su vez de una subconsulta.
- R y S deben ser ***uniones compatibles***
- Si no se agrega la palabra clave **ALL** el resultado será un set en vez de un *multiset* y entonces no habrá filas repetidas.

ORDENAMIENTO Y PAGINACION

- Extendemos el esquema básico con la cláusula **ORDER BY**:

  SELECT $A_1, A_2, ..., A_n$
  FROM $T_1, T_2, ..., T_m$
  [**WHERE** *condition*]
  [**ORDER BY** A_{k_1} [ASC | DESC], A_{k_2} [ASC | DESC], ...];

- Las columnas de ordenamiento $A_{k_1}, A_{k_2}, ...$ deben pertenecer a dominios ordenados, y deben estar incluídas dentro de las columnas de la proyección en la cláusula **SELECT**.
- Si no se especifica, cada columna es ordenada en forma ascendente.
- La paginación es la posibilidad de escoger un rango $[t_{inicio}, t_{fin}]$ del listado de filas del resultado.
  - La forma estándar es: **[OFFSET..ROWS] FETCH FIRST..ROWS ONLY**.
  - Algunos SGBD's implementan otras cláusulas como **LIMIT**.

Ejemplo:

De entre las preguntas hechas desde el 01/01/2017 en adelante, muestre las 10 que recibieron más visitas, indicando su título, la fecha en que se hicieron y la cantidad de visitas que recibieron.

```
SELECT Title, CreationDate, ViewCount
FROM Posts
WHERE CreationDate>='2017-01-01'
AND ParentId IS NULL
ORDER BY ViewCount DESC
OFFSET 0 ROWS FETCH FIRST 10 ROWS ONLY;
```

Limit 5: dame las 1ras 5 una vez que las ordene,

Limit 5 offset 5: `empezando de la 5 dame la 7, 8, 9, 10`

AGREGACION

- Analicemos la siguiente tabla Campeones(nombre_tenista, nombre_torneo, premio) indicando los torneos ganados por distintos tenistas en 2016:

CAMPEONES

nombre_tenista	nombre_torneo	premio
Novak Djokovic	Abierto de Australia	8.000.000
Rafael Nadal	Abierto de Barcelona	1.500.000
Novak Djokovic	Abierto de Madrid	2.500.000
Novak Djokovic	Roland Garros	5.000.000
Andy Murray	Abierto de China	2.500.000
Andy Murray	Master de Shangai	4.000.000
Juan Martín del Potro	Abierto de Estocolmo	300.000
Andy Murray	Master BNP Paribas	2.000.000
Andy Murray	ATP Tour Finals de Londres	4.000.000

- Nos gustaría resumir en una tabla la cantidad de títulos ganados por cada tenista y su premio total anual.

Para hacer esto necesitamos *agrupar* los datos de cada tenista,

CAMPEONES

nombre_tenista	nombre_torneo	premio
Novak Djokovic	Abierto de Australia	8.000.000
Novak Djokovic	Abierto de Madrid	2.500.000
Novak Djokovic	Roland Garros	5.000.000
Rafael Nadal	Abierto de Barcelona	1.500.000
Juan Martín del Potro	Abierto de Estocolmo	300.000
Andy Murray	Abierto de China	2.500.000
Andy Murray	Master de Shangai	4.000.000
Andy Murray	Master BNP Paribas	2.000.000
Andy Murray	ATP Tour Finals de Londres	4.000.000

y luego dejar una única tupla por cada uno, que muestre su nombre pero resuma los torneos mostrando su cantidad, y los premios mostrando el total:

resultado:

nombre_tenista	nombre_torneo	premio
Novak Djokovic	3	15.500.000
Rafael Nadal	1	1.500.000
Juan Martín del Potro	1	300.000
Andy Murray	4	12.500.000

La agregación colapsa las tuplas que coinciden en una serie de atributos (acá es nombre_tenista), **en una única tupla que las representa a todas.**

En SQL hacemos esto con la cláusula **GROUP_BY,** que implementa la operación de agregación.

```
SELECT nombre_tenista, COUNT(nombre_torneo), SUM(premio)
FROM Campeones
GROUP BY nombre_tenista;
```

Esquema de una consulta con agregación:

SELECT A_{k_1}, A_{k_2}, ..., $f_1(B_1)$, $f_2(B_2)$, ..., $f_p(B_p)$
FROM T_1, T_2, ..., T_m
[**WHERE** $condition_1$]
GROUP BY A_1, A_2, ..., A_n
[**HAVING** $condition_2$]
[**ORDER BY** A_{k_1} [ASC | DESC], A_{k_2} [ASC | DESC], ...];

- $A_1, A_2, ..., A_n$ son las *columnas de agrupamiento*, y algunas de ellas participan de la selección final. $B_1, B_2, ..., B_p$ no son columnas de agrupamiento, pero participan de la selección final a través de las funciones de agregación anteriormente mencionadas.

Algunas posibles funciones de agregación: SUM, AVG, COUNT, MAX, MIN.

Si en una agregación no se especifican atributos de agregación (n=0), el resultado tendrá una única tupla.

GROUP BY... HAVING

- La cláusula **HAVING** es opcional, y nos permite seleccionar sólo algunos de los *grupos* del resultado.
- $condition_2$ es por lo tanto una condición que involucra *funciones de agregación sobre las columnas que no son de agrupamiento en el* **GROUP BY**.

Ejemplo

Liste los tags cuyo primer uso ocurrió después del 01/01/2018.

```
SELECT t.TagName
FROM Tags t, PostTags pt, Posts p
WHERE t.Id = pt.TagId
AND pt.PostId = p.Id
GROUP BY t.TagName
HAVING MIN(p.CreationDate)>='2018-01-01';
```

Ejemplo

Muestre los nombres de los 10 usuarios cuyas respuestas a preguntas taggeadas con 'c#' acumulan mayor puntaje.

```
SELECT u.DisplayName, SUM(resp.Score)
FROM Users u, Posts resp, Posts preg, PostTags pt, Tags t
WHERE u.Id = resp.OwnerUserId
AND pt.PostId = preg.Id
AND pt.TagId=t.Id
AND t.TagName='c#'
AND resp.ParentId = preg.Id
GROUP BY u.Id, u.DisplayName
ORDER BY SUM(resp.Score) DESC
OFFSET 0 ROWS FETCH FIRST 10 ROWS ONLY;
```

(ejemplo de parcial)

CONSULTAD ANIDAD

Subqueries en la cláusula WHERE

Yo puedo usar el resultado de un select para ejecutar otra consulta. Como cuando en algebra relacional guardábamos una selección en una variable que después usábamos en otra consulta.

- Es posible incluir una subconsulta dentro de la cláusula **WHERE**.
  - Recordemos: el resultado de una consulta es siempre una tabla.
  - Tip: ¡Y cuando el resultado sólo contiene una fila con una única columna, puede ser pensado y utilizado como un valor constante!

```
SELECT ... FROM ...
WHERE A IN (SELECT X FROM ...); -- Debe devolver una única columna

SELECT ... FROM ...
WHERE A = (SELECT X FROM ...); -- Debe devolver sólo 1 fila!

SELECT ... FROM ...                    -- (feature opcional)
WHERE (A, B) IN (SELECT X, Y FROM ...); -- Debe devolver 2 columnas

SELECT ... FROM ...
WHERE A < [ SOME | ALL ] (SELECT X FROM ...);

SELECT ... FROM ...                          -- Devuelve FALSE/TRUE según
WHERE [ NOT ] EXISTS (SELECT ... FROM ...); -- la tabla esté vacía o no
```

Cuantificadores ALL y SOME

Encuentre para cada *tag* la/s pregunta/s que haya/n tenido la mayor cantidad de vistas, indicando el nombre del *tag*, el título de la/s pregunta/s y la cantidad de vistas. Sólo muestre aquellos *tags* para los que la cantidad de vistas es de al menos 2 millones.

```
SELECT t.TagName, preg.Title, preg.ViewCount
FROM Tags t, PostTags pt, Posts preg
WHERE t.Id = pt.TagId
AND pt.PostId = preg.Id
AND preg.ParentId IS NULL
AND preg.ViewCount > 2000000
AND ViewCount >= ALL (SELECT ViewCount
                      FROM PostTags pt2, Posts p
                      WHERE pt2.TagId = t.Id
                      AND pt2.PostId = p.Id);
```

INSERCIONES

- Las inserciones se realizan con el comando **`INSERT INTO`**. Dada una tabla T con columnas $A_1, A_2, ..., A_n$, se admiten las siguientes posibilidades:
  - Insertar un listado de n-filas:

    INSERT INTO T
    VALUES $(a_{11}, a_{12}, ..., a_{1n}), (a_{21}, a_{22}, ..., a_{2n}), ..., (a_{p1}, a_{p2}, ..., a_{pn})$;

    - El orden de carga de los valores debe ser el mismo que el de las columnas en la tabla.
  - Insertar un listado de k-filas, con $k < n$

    INSERT INTO $T(A_{i_1}, A_{i_2}, ..., A_{i_k})$
    VALUES $(a_{1i_1}, a_{1i_2}, ..., a_{1i_k}), (a_{2i_1}, a_{2i_2}, ..., a_{2i_k}), ..., (a_{pi_1}, a_{pi_2}, ..., a_{pi_k})$;

  - Insertar el resultado de una consulta:

    INSERT INTO $T(A_{i_1}, A_{i_2}, ..., A_{i_k})$
    SELECT ...
    FROM ...;

    - La consulta debe devolver una tabla con k columnas, y las columnas deben ser unión compatibles.
  - En cualquiera de los casos, si...
    - Se asigna a una columna un valor fuera de su dominio
    - Se omitió una columna que no podía ser **`NULL`**
    - Se puso en **`NULL`** una columna que no podía ser **`NULL`**
    - La clave primaria asignada ya existe en la tabla
    - Una clave foránea hace referencia a una clave no existente
  - ... ✗ no se inserta.

ELIMINACIONES

- La sintaxis para las eliminaciones es:

```
DELETE FROM T
WHERE condition;
```

  - *condition* puede ser una combinación de condiciones atómicas.
  - Si no se especifican condiciones, se eliminan todas las filas.
- Para cada fila *t* que se intente eliminar y que es referenciada por una clave foránea desde otra tabla:
  - Si dicha clave foránea se configuró en **`ON DELETE CASCADE`**
    - ✓ Se eliminan todas las filas que referencian a ésta, y luego se elimina *t*.
  - Si en cambio se configuró en **`ON DELETE SET NULL`**
    - ✓ Se ponen en **`NULL`** todas las claves foráneas de las filas que referencian a ésta, y luego se elimina *t*.
  - Si se configuró en **`ON DELETE RESTRICT`**
    - ✗ No se elimina *t*.

MODIFICACIONES

- Las modificaciones se realizan con el comando **`UPDATE`**.

```
UPDATE T
SET A1 = c1, A2 = c2, ..., Ak = ck
WHERE condition;
```

  - *condition* puede ser una combinación de condiciones atómicas.
  - Un único **`UPDATE`** puede modificar muchas filas.
- Para cada fila *t* que cumpla la condición, si...
  - Se modifica una columna asignándole un valor fuera de su dominio
  - Se pasó a **`NULL`** una columna que no podía tomar ese valor
  - Se asignó a la clave primaria un valor que ya existe en la tabla
  - Se modificó una clave foránea para hacer referencia a una clave no existente
- ... ✗ entonces *t* no se actualiza.

- Atención! Si en la fila *t* se modifica una clave primaria que es referenciada por una clave foránea desde otra tabla:
  - Si dicha clave foránea se configuró en **`ON UPDATE CASCADE`**
    - ✓ Se modifican todas las filas que referencian a ésta en forma acorde, y luego se modifica *t*.
  - Si en cambio se configuró en **`ON UPDATE SET NULL`**
    - ✓ Se ponen en **`NULL`** todas las claves foráneas de las filas que referencian a ésta, y luego se modifica *t*.
  - Si se configuró en **`ON UPDATE RESTRICT`**
    - ✗ No se modifica *t*.

TABLAS

- Una tabla se elimina con **DROP TABLE**
- Un esquema se elimina con **DROP SCHEMA**

Funciones y estructuras auxiliares

Manejo de Strings

- El estándar SQL cuenta con varias funciones para manipular strings. Entre ellas:
  - **SUBSTRING(string FROM start FOR length)**: Selecciona un substring desde la posición *start* y de largo *length*.
  - **UPPER(string)/LOWER(string)**: Convierte el string a mayúsculas/minúsculas.
  - **CHAR_LENGTH(string)**: Devuelve la longitud del string.

Conversion de Tipos

- **CAST(attr AS TYPE)** permite realizar conversiones entre tipos.
- **EXTRACT(campo FROM attr)** permite extraer información de una columna de fecha/hora (*feature* opcional).

```
SELECT (EXTRACT(DAY FROM fecha)) AS dia, COUNT(nro_factura)
FROM Facturas f;
GROUP BY dia;
```

  - Valores posibles para el campo: **YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, ...**
- **COALESCE(expr1, expr2, ..., exprN)** devuelve para cada tupla la primera expresión no nula de izquierda a derecha.

```
--Reemplaza los valores nulos en la columna domicilio
--por el string '<desconocido>'.
SELECT apellido, nombre, COALESCE(domicilio, '<desconocido>')
FROM Clientes;
```

Estructura CASE WHEN...THEN...ELSE...END

- La estructura **CASE** nos permite agregar cierta lógica de la programación estructurada a una sentencia SQL.
- Permite seleccionar entre distintos valores posibles de salida en función de distintas condiciones.

padrón	apellido	nombre	primera_op	primer_rec	segundo_rec
102111	Bertrán	Verónica	9		
103553	Salamanca	Ernesto	2	7	
104617	Guzmán	Claudia			8
105928	Sanz	Rubén		2	

```
SELECT padrón, apellido, nombre, CASE
       WHEN primera_op>=4 OR primer_rec>=4 OR segundo_rec>=4
       THEN 'APROBO_PARCIAL'
       ELSE 'DESAPROBO_PARCIAL'
       END AS situacion_parcial
FROM Notas_Parcial;
```

→

padrón	apellido	nombre	situacion_parcial
102111	Bertrán	Verónica	APROBO_PARCIAL
103553	Salamanca	Ernesto	APROBO_PARCIAL
104617	Guzmán	Claudia	APROBO_PARCIAL
105928	Sanz	Rubén	DESAPROBO_PARCIAL

Clausula WITH

- La cláusula **WITH** permite construir una tabla auxiliar temporal previa a una consulta. No es Core-SQL.

WITH $T[(A_1, A_2, ..., A_n)]$
AS (<*subquery*>)
<*query*>;

WITH 1: Pseudocódigo
T = *subquery*$(R_1, R_2, ...)$;
return *query*$(T, ...)$;

Listar los nombres de badges que tiene el usuario Jon Skeet.

```
WITH Jon
AS (SELECT Id
    FROM Users
    WHERE DisplayName='Jon Skeet')
SELECT DISTINCT Name
FROM Badges b, Jon j
WHERE b.UserId = j.Id;
```