# **20.- SELECT**

|  |
| --- |
| SELECT <columnas/\*>  FROM <tabla>; |

**Consultas:**

USE sakila;

SELECT \*

FROM actor;

SELECT first\_name

FROM actor;

SELECT last\_name

FROM actor;

SELECT first\_name, last\_name

FROM actor;

SELECT city, country\_id

FROM city;

SELECT country\_id, country

FROM country;

# 

# **21.- OPERADORES SQL**

**Operadores básicos de SQL:**

|  |
| --- |
| < menor que  > mayor que  <= menor o igual que  >= mayor o igual que  <> diferente de  = igual |

**Consulta:**

SELECT \*

FROM actor

WHERE first\_name = ‘JOE’;

Esta consulta nos devolverá aquellos registros, de la tabla actor, cuyo primer nombre sea JOE.

# **22.- WHERE**

Se utiliza para consultas más específicas, que contengan una condición.

Su estructura básica es la siguiente:

|  |
| --- |
| SELECT <Columnas/\*>  FROM <Tabla>  WHERE <Condición>; |

**Consultas:**

SELECT first\_name, last\_name

FROM actor

WHERE first\_name = ‘JOE’;

(Nombre y apellidos de actores que se llamen JOE)

SELECT \*

FROM actor

WHERE actor\_id >= 100;

(Código de actor mayor o igual a 100)

SELECT \*

FROM actor

WHERE actor\_id<=0;

(esto serviría de test para nuestra base de datos, pues no puede haber un actor\_id=0)

SELECT \*

FROM film

WHERE rental\_duration > = 6;

(películas con un alquiler superior o igual a 6 días)

SELECT \*

FROM film

WHERE rental\_duration < 6;

(con esta consulta podríamos verificar la veracidad de los datos, comparándolos con los de la consulta anterior).

SELECT first\_name, last\_name, email

FROM customer

WHERE fisrt\_name = ‘PAMELA’

(Nos dará el nombre completo y el correo electrónico de los clientes que se llamen ‘PAMELA’.

SELECT first\_name, last\_name, email

FROM customer

WHERE first\_name <> ’PAMELA’

(Nos dará el nombre completo y el correo electrónico de los clientes que no se llamen ‘PAMELA’)

# **23.- ORDER BY**

Lo utilizamos para ordenar, por uno o varios campos. La estructura será:

|  |
| --- |
| SELECT <columnas/\*>  FROM <tabla>  ORDER BY <columna> <ASC/DESC>; |

**Consultas:**

SELECT \*

FROM inventory

ORDER BY store\_id desc;

SELECT \*

FROM film

ORDER BY title desc;

SELECT \*

FROM film

WHERE rental\_duration >= 6

ORDER BY rental\_rate desc;

SELECT \*

FROM city

WHERE country\_id = 2

ORDER BY city;

SELECT \*

FROM inventory;

SELECT \*

FROM inventory

ORDER BY store\_id;

SELECT \*

FROM inventory

ORDER BY store\_id desc;

SELECT \*

FROM film

ORDER BY title desc;

SELECT \*

FROM film

ORDER BY rental\_rate desc;

SELECT \*

FROM film

ORDER BY rental\_rate desc;

SELECT \*

FROM film

WHERE rental\_duration >= 6

ORDER BY rental\_rate desc;

SELECT \*

FROM city

WHERE country\_id = 2

ORDER BY city;

# **24.- DISTINCT**

Nos devuelve valores únicos.

|  |
| --- |
| SELECT DISTINCT (<columnas/\*>)  FROM <tabla>; |

**Consultas:**

SELECT DISTINCT country\_id

FROM city

WHERE country\_id = 2

ORDER BY city\_id;

SELECT DISTINCT first\_name

FROM actor

where first\_name ='PENELOPE';

SELECT DISTINCT first\_name

FROM actor

ORDER BY first\_name;

# **25.- FUNCIÓN MAX-MIN**

Nos devuelve los valores máximos y mínimos de una fila.

Su estructura básica es la siguiente:

|  |
| --- |
| SELECT <MAX/MIN>(<columnas/\*>)  FROM <Tabla>; |

**Consultas:**

Nos devuelve el valor máximo del campo “rental\_duration” en la tabla **film**:

SELECT MAX(rental\_duration)

FROM film;

Nos devuelve el valor máximo en el campo “replacement\_cost” de la tabla **film:**

SELECT MAX(replacement\_cost)

FROM film;

Nos devuelve el valor mínimo del campo “title” en la tabla **film:**

SELECT MIN(Title)

FROM film;

Nos devuelve el valor máximo de la columna “replacement\_cost” donde el identificador sea mayor a 500 en la tabla **film:**

SELECT MAX(replacement\_cost)

FROM film

WHERE film\_id >500;

Devuelve el valor máximo en la columna “replacement\_cost” de la tabla **film**, buscando sólo en los que tengan la condición “release\_year = 2006”:

SELECT MAX(replacement\_cost)

FROM film

WHERE release\_year = ’2006’;

Ordena el título y nos da el último, entre los 100 primeros:

SELECT MAX(title)

FROM film

WHERE film\_id <=100

ORDER BY title;

Ordena la tabla “actor” por “last\_name” y selecciona el mayor entre los que tengan un “actor\_id” menor o igual a 75:

SELECT MAX(last\_name)

FROM actor

WHERE actor\_id<=75

ORDER BY last\_name;

# 

# **26.- FUNCIÓN SUM**

A función SUM retorna a suma dunha columna. A estrutura da consulta é a seguinte:

|  |
| --- |
| SELECT SUM(<columna/\*>)  FROM <tabla>; |

**Consultas:**

SELECT SUM (first\_name)

FROM actor

WHERE actor\_id < 75

ORDER BY last\_name; **\*\*\***

SELECT SUM(first\_name)

FROM actor;

*Da como resultado 0*

SELECT SUM(language\_id)

FROM actor; **\*\*\***

SELECT SUM(rental\_rate)

FROM film; \*\*\*

*Da como resultado 2980.00*

SELECT SUM(language\_id, rental\_duration)

FROM film; **\*\*\***

\*\*\* = *Da como resultado Query with errors.*

# 

# **27.- FUNCIÓN AVG**

Nos ayuda a realizar consultas, las cuales devuelven el promedio de la columna. La estructura es la siguiente:

|  |
| --- |
| SELECT AVG(<columna/\*>)  FROM <tabla>; |

**Consultas:**

SELECT AVG(rental\_duration)

FROM film

WHERE film\_id < 500;

Esta consulta promediará los que tengan un film\_id menor a 500 lo que nos dará un resultado de 5,032.

Cuantos menos números tengamos en el promedio, lógicamente será menor el resultado.

Utilizaremos a continuación una tabla que contenga mucha información, preferiblemente números.

SELECT AVG(amount)

FROM payment;

Cuando ejecutamos nos da 4.2006667;

**La función AVERAGE, tiene como resultado darnos el promedio de una columna.**

# 

# **28.- FUNCIÓN COUNT**

Para obtener el número de registros de una tabla.

|  |
| --- |
| SELECT COUNT(<columna/\*>)  FROM <tabla>; |

**Consultas:**

Para obtener la cantidad de registros dentro de la tabla film:

SELECT COUNT(\*)

FROM film;

Da como resultado la cantidad de registros con id <= 30:

SELECT COUNT(\*)

FROM film

WHERE film\_id <=30;

Cuenta los “PENELOPE” como first\_name en la tabla actor:

SELECT COUNT(first\_name)

FROM actor

WHERE first\_name = "PENELOPE";

SELECT COUNT(campo2)

FROM nombre

WHERE campo2 =''; no es igual a WHERE campo2 is null;

**NULL** significa valor no asignado

**VACÍO** significa asignado sin contenido , es decir, string con 0 caracteres

**IMPORTANTE:**

[**https://en.wikipedia.org/wiki/Null\_%28SQL%29**](https://es.wikipedia.org/wiki/Null_(SQL))

**VERSIÓN EN ESPAÑOL** [**https://es.wikipedia.org/wiki/Null\_(SQL)**](https://es.wikipedia.org/wiki/Null_(SQL))

# **29.- GROUP BY**

Tiene la función principal de agrupar un conjunto de datos, este tiene que estar dentro del SELECT, y suele ser usado con funciones AVG, COUNT, SUM.

|  |
| --- |
| SELECT <columnas/\*>  FROM <tabla>  GROUP BY <columna>; |

**Consultas:**

SELECT \*

FROM city;

SELECT COUNT(distinct country\_id)

FROM city;

Cantidad de ciudades por país:

SELECT country\_id, COUNT(city)

FROM city

GROUP BY country\_id;

SELECT \*

FROM film;

SELECT rental\_duration, COUNT(rental\_rate), AVG(rental\_rate)

FROM film

GROUP BY rental\_duration;

# **30.- HAVING**

Es como un WHERE 2.0. Se usa para crear condiciones, pero permite usar funciones como COUNT, MAX, SUM, etc. Se suele utilizar con Group By, porque el Group By utiliza estas condiciones, y el Having trabaja sobre este grupo.

Where: opera sobre toda la tabla.

Having: opera sobre el grupo hecho por Group By.

|  |
| --- |
| SELECT <columna/\*>  FROM <tabla>  GROUP BY <columna>  HAVING <función/condición>; |

**Consultas:**

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration <= 5;

Lo mismo se puede hacer con WHERE

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

WHERE rental\_duration <= 5

GROUP BY rental\_duration;

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration <> 0;

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration = NULL;

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration = '';

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration = -1;

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration >= -125;

**No se puede poner un WHERE con una función, p.ej. count**

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

WHERE COUNT(rental\_duration) >= 200;

**Con HAVING sí se puede:**

SELECT rental\_duration, COUNT(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING count(rental\_duration) <= 200;

Muestra los países que tengan 2 o + ciudades:

SELECT country\_id, COUNT(city)

FROM city

GROUP BY country\_id

HAVING count(city) > 1;

SELECT country\_id, COUNT(city)

FROM city

GROUP BY country\_id

HAVING count(city) > 35;

SELECT last\_name, COUNT(last\_name)

FROM actor

GROUP BY last\_name

HAVING COUNT(last\_name) > 3;

## **MÁS ACERCA DE HAVING**

DESCRIBE film;

SELECT \*

FROM film;

SELECT film\_id, rental\_duration, count(rental\_duration)

FROM film

WHERE rental\_duration < 5

GROUP BY rental\_duration

SELECT film\_id, rental\_duration, count(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING rental\_duration < 5

SELECT film\_id, rental\_duration, count(rental\_duration)

FROM film

GROUP BY rental\_duration

HAVING COUNT(rental\_duration) <= 200

SELECT film\_id, rental\_duration, count(rental\_duration)

FROM film  
 GROUP BY rental\_duration

HAVING COUNT(rental\_duration) <= 1000

/\* --------------- \*/

DESCRIBE city;

SELECT \*

FROM city;

SELECT \*

FROM city

GROUP BY country\_id;

SELECT country\_id, COUNT(city)

FROM city

GROUP BY country\_id

HAVING count(city) > 1

SELECT first\_name, last\_name, COUNT(last\_name)

FROM actor

GROUP BY last\_name

HAVING last\_name = 'GUINESS' /\* ¿donde está la función?

**Enlaces útiles:**

**MariaDB**

<https://mariadb.com/kb/en/group-by/>

**3schools**

<https://www.w3schools.com/sql/sql_having.asp>

**MySQLTutorial**

<https://www.mysqltutorial.org/mysql-having.aspx/>

**Aner Barrena**

<https://www.anerbarrena.com/mysql-having-9901/>

# 

# **31.- AND**

Nos permite realizar consultas con condiciones múltiples, donde todas las condiciones tienen que ser VERDADERAS para que se cumpla.

|  |
| --- |
| SELECT <columnas/\*>  FROM <tabla>  WHERE <condición1> AND <condición2>...; |

[Tabla de verdad](https://es.wikipedia.org/wiki/Tabla_de_verdad)

**Consultas:**

SELECT \*

FROM film

WHERE rental\_duration > 5 AND replacement\_cost < 30;

SELECT \*

FROM film

WHERE rental\_duration > 5 AND replacement\_cost < 12;

SELECT \*

FROM film

WHERE rental\_duration > 5 AND replacement\_cost < 12

ORDER BY rental\_duration ASC;

SELECT \*

FROM film

WHERE rental\_duration > 5 AND replacement\_cost < 12 AND length < 100

ORDER BY rental\_duration ASC;

SELECT \*

FROM film

WHERE rental\_duration > 5 AND replacement\_cost < 12 AND length > 1000

ORDER BY rental\_duration ASC;

/\*Introducimos en la tabla actor una nueva ‘Penelope Guiness’\*/

INSERT INTO actor

(first\_name,last\_name) VALUES ("PENELOPE","GUINESS");

SELECT \*

FROM actor

WHERE first\_name='PENELOPE' AND last\_name = 'GUINESS';

SELECT \*

FROM actor

WHERE first\_name = 'PENELOPE' AND last\_name = PERALTA;

SELECT rental\_rate, SUM(rental\_rate)

FROM film

GROUP by rental\_rate

HAVING SUM(rental\_rate) < 1000;

SELECT rental\_rate, SUM(rental\_rate)

FROM film

GROUP by rental\_rate

HAVING SUM(rental\_rate) = 150;

SELECT rental\_duration,rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_rate) < 1000;

SELECT rental\_duration,rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration,rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING COUNT(rental\_duration) AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration,rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_duration) AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration,rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_duration) = 200 AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration, rental\_rate, SUM(rental\_duration), COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_duration) < 1000 AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration, rental\_rate, SUM(rental\_duration), COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_duration) < 300 AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration, rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING SUM(rental\_duration) < 300 AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

SELECT rental\_duration, rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING COUNT(rental\_duration) < 70 AND SUM(rental\_rate) < 1000

ORDER BY rental\_duration DESC;

# 

# **32.- OR (Inclusivo)**

Nos permite realizar consultas con condiciones múltiples, donde sólo una condición tiene que ser VERDADERA para que se cumpla.

|  |
| --- |
| SELECT <columnas/\*>  FROM <tabla>  WHERE <condición1> OR <condición2>...; |

**Consultas:**

SELECT rental\_duration, rental\_rate, COUNT(rental\_duration), SUM(rental\_rate)

FROM film

GROUP by rental\_duration, rental\_rate

HAVING COUNT(rental\_duration) < = 10 or SUM(rental\_rate) = 0

ORDER BY rental\_duration DESC;

SELECT \*

FROM city

WHERE country\_id > 100 or city\_id > 20;

SELECT \*

FROM city

WHERE country\_id > 300 or city\_id <= 299;

# 

# **33.- OPERADOR LIKE**

El operador LIKE es utilizado en la cláusula WHERE para buscar datos específicos, para el LIKE se utilizan dos operadores:

**%** Representa 0,1 o múltiples caracteres.

**\_** Representa un solo carácter.

|  |
| --- |
| SELECT <columna/\*>  FROM <tabla>  WHERE <columna> LIKE <buscador>; |

**Consultas:**

SELECT \*

FROM film

WHERE title LIKE 'AC%';

SELECT \*

FROM film

where title LIKE '%';

SELECT \*

FROM film

WHERE title LIKE 'ACADEMY DINOSA%';

SELECT \*

FROM country

WHERE country LIKE '%A%';

SELECT \*

FROM country

WHERE (country LIKE 'A%'

OR country LIKE 'a%')

AND (country LIKE '%A'

OR country LIKE '%a');

# 34.- OPERADOR NOT

Operador utilizado en la cláusula WHERE para mostrar todo menos la condición.

|  |
| --- |
| SELECT <columnas / \*>  FROM <tabla>  WHERE NOT <condición>; |

**Exemplos:**

SELECT \*  
 FROM actor

WHERE NOT first\_name LIKE "PEN%"

SELECT \*  
 FROM actor

WHERE first\_name NOT LIKE "PEN%"  
 *(Devolve todos os nomes de actor que NON comecen por ‘Pen’)*

*SELECT \**

*FROM actor*

*WHERE NOT first\_name LIKE "PEN%"*

*SELECT \**

*FROM actor*

*WHERE first\_name LIKE "PEN%"*

*SELECT \**

*FROM actor*

*WHERE first\_name NOT LIKE "PEN%"*

*SELECT \**

*FROM actor*

*WHERE first\_name LIKE NOT "PEN%" -- INCORRECTA*

*SELECT \**

*FROM actor*

*WHERE NOT first\_name NOT LIKE "PEN%"*

# 35.- INSERT

Utilizado para realizar insertar datos dentro de la base de datos. La estructura es la siguiente:

|  |
| --- |
| INSERT INTO nombre de tabla (campo x, campo y, …)  VALUES (dato1, dato2, ...) |

Ejemplos:

/\*NOS DA UN ERROR POR LLAVE DUPLICADA\*/

SELECT \*

FROM actor;

INSERT INTO actor (actor\_id, first\_name, last\_name)

VALUE (100, 'PEDRO', 'VARGAS');

Forma correcta:

SELECT \*

FROM actor;

INSERT INTO actor (actor\_id, first\_name, last\_name)

VALUE (1002, 'ROCIO', 'DURCAL');

 1 fila insertada. (La consulta tardó 0,0044 segundos.)

SELECT \*

FROM actor;

INSERT INTO actor (actor\_id, first\_name, last\_name)

VALUE (500, 'RAPHAEL', 'MARCOS');

 1 fila insertada. (La consulta tardó 0,0044 segundos.)

SELECT \*

FROM actor;

INSERT INTO actor (first\_name, last\_name)

VALUE ('JULIO', 'IGLESIAS');

 1 fila insertada.

La Id de la fila insertada es: 1003 (La consulta tardó 0,0043 segundos.)

# 36.- UPDATE

Para actualizar datos, lo más común es usar la cláusula WHERE(no siempre, pero si no actualizaria la columna entera).

|  |
| --- |
| UPDATE <TABLA>  SET <COLUMNA> = <NUEVO\_DATO>  WHERE<CONDICIÓN> |

Ejemplo:

Cambiar Cuzco por Cusco:

UPDATE city

SET city = 'Cusco'

WHERE city\_id = 1001

Es importante poner el IGUAL, si no podemos cambiar muchísimos datos que no queremos cambiar.

# 37.- DELETE (ELIMINAR)

La cláusula DELETE se usa para eliminar datos de una BD.

**¡¡¡ IMPORTANTÍSIMO !!!**

***NO OLVIDAR DE UTILIZAR EL WHERE, si no será eliminada toda la tabla.***

|  |
| --- |
| DELETE FROM <tabla>  WHERE <condición>; |

Ejemplos:

DELETE FROM city

WHERE city\_id >= 1001;

* Esta consulta eliminaría solamente el registro que cumple la condición que hemos puesto ( >= 1001).

DELETE FROM city

WHERE CITY\_ID >= 132;

* Esta otra consulta arrojaría ERROR, puesto que no se puede eliminar/actualizar este registro por ser city\_id>=132 una CLAVE FORÁNEA de otra tabla, es decir afecta a otros registros de diferentes tablas.

***Las operaciones sobre un registro que pertenece a una relación deben ser en cascada (en todas las tablas implicadas).***

# 38.- SUBCONSULTAS.

Una subconsulta es un método utilizado para pasar como parámetro a la cláusula WHERE una consulta SELECT. La estructura es la siguiente:

|  |
| --- |
| SELECT <columna / \*>  FROM <tabla>  WHERE <columna> <operador>(  SELECT <columna>  FROM <tabla2>  WHERE <condición>) |

Ejemplos:

SELECT \* FROM city;

SELECT \* from country;

Con estas consultas determinamos las claves comunes (country\_id, que es clave foránea de “city” y primaria de “country”. La tabla “country\_id” es la tabla padre).

SELECT \*

FROM city

WHERE country\_id = (

SELECT country\_id

FROM country

WHERE country=”Algeria”);

Hacemos una subconsulta (entre paréntesis) para poder buscar las ciudades de Algeria. Para ello indicamos con la subconsulta la condición de la consulta principal, en este caso el country\_id de Algeria, que nos permita encontrar todas las ciudades de ese país, y en la principal city.

CONSULTA

========

SELECT (campos / \*)

FROM tabla

WHERE condición;

SUBCONSULTA

===========

a)

SELECT (campos / \*)

FROM tabla

WHERE condición = (CONSULTA)

b)

SELECT (campos / \*)

FROM tabla

WHERE condición = (

SELECT (campos / \*)

FROM tabla

WHERE condición

);

# 39.- LIMIT

Se utiliza para limitar la cantidad de resultados mostrados en las consultas SELECT

|  |
| --- |
| SELECT <columna>  FROM <tabla> LIMIT <número de resultados a visualizar> |

Nos mostraría el primer registro almacenado

**select** \*

**from** actor **limit** 1;

# 40.- IN

El operador **IN** le permite especificar múltiples valores en una cláusula WHERE. El operador **IN** es una abreviatura para múltiples condiciones **OR**.

|  |
| --- |
| SELECT <columna / \*>  FROM <tabla>  WHERE <columna> IN <valor1, valor2, …> |

**Ejemplos:**

SELECT \*

FROM actor

WHERE first\_name In (“PENELOPE”,”JOHNY”)

AND last\_name IN (“GUINESS,”LOLLOBRIGIDA”);

SELECT \*

FROM city

WHERE city IN (“Abha”,”Cusco”,”Aden”);

SELECT \*

FROM city

WHERE city IN (60,80,100);

# 41.- BETWEEN

Este operador nos permite realizar consultas en un rango especificado, es una manera de abreviar el uso de mayor que y menor que.

|  |
| --- |
| SELECT <columna / \*>  FROM <tabla>  WHERE <columna> BETWEEN <valor1> AND < valor2, …>; |

**Ejemplos:**

SELECT \*

FROM city

WHERE city\_id BETWEEN 5 AND 8;

SELECT \*

FROM payment

WHERE payment\_date BETWEEN '2005-05-24' AND '2005-05-26';

SELECT \*

FROM actor

WHERE actor\_id BETWEEN 1 AND 210 AND first\_name = 'PENELOPE';

# 42.- ALIAS

Se utilizan para dar a una tabla o una columna un nombre temporal. Los alias a menudo se usan para hacer que los nombres de columna sean más legibles. **Solo existe un alias mientras dura la consulta.**

|  |
| --- |
| SELECT <columna> AS <alias>  FROM <tabla>; |

|  |
| --- |
| SELECT <columna> AS <alias>  FROM <tabla> AS <alias>; |

**Consultas ejemplos:**

SELECT actor\_id AS 'Id', first\_name AS 'Nombre',

last\_name AS 'Apellido', last\_update AS 'Ultima Actualización'

from actor;

SELECT actor\_id AS 'Id', first\_name AS 'Nombre',

last\_name AS 'Apellido', last\_update AS 'Ultima Actualización'

from actor As 'a';

# NOTACIÓN DE PUNTO

# objeto.propiedad

SELECT \*

FROM `sakila`.`actor`;

SELECT `actor`.`first\_name`

FROM `sakila`.`actor`;

SELECT a.actor\_id Id, first\_name Nombre,

last\_name Apellido, last\_update 'Ultima Actualización'

from actor a;

# 43. NOT NULL

Este operador nos va a servir para mostrar todos los registros que no se encuentren vacíos

|  |
| --- |
| SELECT <columna / \*>  FROM <tabla>  WHERE <columna> IS NOT NULL; |

Ejemplos:

SELECT \*

FROM country

WHERE country IS NOT NULL

Insertamos un valor nulo en la tabla country, pero antes debemos alterar la tabla para que acepte valores nulos:

[ALTER](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/alter-table.html) [TABLE](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/alter-table.html) `country` CHANGE `country` `country` [VARCHAR](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/string-types.html)(50) [CHARACTER](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/charset.html) [SET](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/charset.html) utf8mb4 [COLLATE](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/charset-collations.html) utf8mb4\_general\_ci NULL;

INSERT INTO `country` (`country\_id`, `country`, `last\_update`) VALUES (NULL, 'null', current\_timestamp());

SELECT \*

FROM country

WHERE country IS NULL

ORDER BY country

# 44.- CONSULTA A MÚLTIPLES TABLAS

La operación JOIN y UNION permite mostrar columnas de varias tablas como si se tratase de una sola tabla, combinando entre sí los registros relacionados.

# TODOS LOS TIPOS DE JOIN EN SQL – GUÍA DE REFERENCIA RÁPIDA

<https://ingenieriadesoftware.es/tipos-sql-join-guia-referencia/>

# **45.- LEFT JOIN**

# Esta sentencia combina los valores de la primera tabla con los valores de la segunda tabla. Siempre devolverá las filas de la primera tabla, incluso aunque no cumplan la condición.

La consulta será de la siguiente manera:

* SELECT <columna / \*>
* FROM <tabla> AS t1
* LEFT JOIN <tabla2> AS t2 ON t1.columna = t2.columna
* SELECT <columna / \*>
* FROM <tabla> AS t1
* LEFT JOIN <tabla2> AS t2 ON t1.columna = t2.columna
* WHERE t2.columna IS NULL.

**Ejemplo:**

[**SELECT**](https://mariadb.com/kb/en/library/select/) **\***

**FROM** [**film\_category**](http://localhost/adminer/?username=root&db=sakila&table=film_category) **AS fc**

**LEFT** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**category**](http://localhost/adminer/?username=root&db=sakila&table=category) **AS c ON c.category\_id = fc.category\_id**

**ORDER BY c.category\_id DESC;**

# **46.- RIGHT JOIN**

La sentencia RIGHT JOIN combina los valores de la segunda tabla con los valores de la primera tabla. Siempre devolverá las filas de la segunda tabla, incluso aunque no cumplan la condición.

* SELECT <columna / \*>
* FROM <tabla> t1
* RIGHT JOIN <tabla2> t2
* ON t1.columna = t2.columna2
* SELECT <columna / \*>
* FROM <tabla> t1
* RIGHT JOIN <tabla2> t2
* ON t1.columna = t2.columna2
* WHERE t1.column IS NULL

**Ejemplos:**

[**SELECT**](https://mariadb.com/kb/en/library/select/) **\***

**FROM** [**film\_category**](http://localhost/adminer/?username=root&db=sakila&table=film_category) **AS fc**

**RIGHT** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**category**](http://localhost/adminer/?username=root&db=sakila&table=category) **AS c ON c.category\_id = fc.category\_id**

**ORDER BY c.category\_id DESC;**

[**SELECT**](https://mariadb.com/kb/en/library/select/) **\***

**FROM** [**film\_category**](http://localhost/adminer/?username=root&db=sakila&table=film_category) **AS fc**

**RIGHT** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**film**](http://localhost/adminer/?username=root&db=sakila&table=film) **AS f ON f.film\_id = fc.film\_id**

**RIGHT** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**category**](http://localhost/adminer/?username=root&db=sakila&table=category) **AS c ON c.category\_id = fc.category\_id**

**ORDER BY c.category\_id DESC;**

# 47.- INNER JOIN

La sentencia Inner Join es la sentencia Join por defecto y consiste en combinar cada fila de una tabla con cada fila de la otra tabla, seleccionando aquellas filas que cumplan una determinada condición.

* SELECT <columna / \*>
* FROM <tabla> t1
* INNER JOIN <tabla2> t2
* ON t1.columna = t2.columna2

Ejemplos:

[**SELECT**](https://mariadb.com/kb/en/library/select/) **c.name AS Categoria, COUNT(f.title)**

**FROM** [**film\_category**](http://localhost/adminer/?username=root&db=sakila&table=film_category) **AS fc**

**INNER** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**film**](http://localhost/adminer/?username=root&db=sakila&table=film) **AS f**

**ON f.film\_id = fc.film\_id**

**INNER** [**JOIN**](https://mariadb.com/kb/en/library/join/) [**category**](http://localhost/adminer/?username=root&db=sakila&table=category) **AS c**

**ON c.category\_id = fc.category\_id**

**ORDER BY c.category\_id DESC;**

SELECT \*

FROM actor AS a

INNER JOIN film\_actor AS fa

ON a.actor\_ID = fa.actor\_id;

SELECT a.first\_name AS Actor, f.title AS Pelicula

FROM actor AS a

INNER JOIN film\_actor AS fa

ON a.actor\_ID = fa.actor\_id

INNER JOIN film AS f

ON f.film\_id = fa.film\_id

ORDER BY a.actor\_id desc;

INSERT INTO actor(actor\_id, first\_name, last\_name)

VALUE(1002, ‘Ana Paula’, ‘Chavez’)

Si insertamos un valor en la tabla actor como no hay nada relacionado con esa actriz en las otras tablas, al hacer INNER JOIN no aparecerá porque sólo pertenece a la tabla actor. Si queremos que aparezca debemos hacer un LEFT JOIN ( aparecerá sin datos)

# 48.- UNION

Se utiliza para seleccionar los datos relacionados entre dos tablas. **Las columnas tienen que ser del mismo tipo de datos.** Solo se devuelven los valores distintos.

* SELECT <columna / \*>
* FROM <table1>
* UNION
* SELECT<columna / \*>
* FROM table2;

Ejemplos:

Esta consulta funciona porque ambas tienen 4 campos:

SELECT \*

FROM actor

UNION

SELECT \*

FROM city;

Sin embargo esta otra consulta daría error porque en la primera tabla le pide un solo campo y en la segunda todo:

SELECT actor\_id

FROM actor

UNION

SELECT \*

FROM city;

Primero devuelve los nombres y a continuación las ciudades, en una única columna:

SELECT first\_name AS "NOMBRE O CIUDAD"

FROM actor

UNION

SELECT city

FROM city;

# 49.- UNION ALL

Es similar a UNION con la excepción de que se seleccionan todos los valores.

* **SELECT <columna / \*>**
* **FROM <table1>**
* **UNION ALL**
* **SELECT<columna / \*>**
* **FROM table2;**

SELECT first\_name AS "NOMBRE O CIUDAD"

FROM actor

UNION ALL

SELECT city

FROM city;

------------

CREATE TABLE nuevo1(

idnuevo1 int PRIMARY KEY,

nombre VARCHAR(20));

CREATE TABLE nuevo2(

idnuevo2 int PRIMARY KEY,

nombre VARCHAR(20));

INSERT INTO nuevo1(idnuevo1,nombre)

VALUE(1001,"Pedro"),(1002,"Jorge"),(1003,"Roberto");

INSERT INTO nuevo2(idnuevo2,nombre)

VALUE(1001,"Pedro"),(1002,"Rosmeri"),(1003,"Maria");

-----------

-- Con el UNION se detecta intersección de dos registros (son iguales) y se muestra solo 1

SELECT \*

FROM nuevo1

UNION

SELECT \*

FROM nuevo2;

-- muestra los repetidos también

SELECT \*

FROM nuevo1

UNION ALL

SELECT \*

FROM nuevo2;

# 50.- UNIQUE

La restricción UNIQUE asegura que todo los valores en una columna son diferentes

**CREATE TABLA** <tabla>(<columna><tipo de dato> [tamaño]<columna> **UNIQUE**

**);**

# 51.- NOT NULL

Por defecto, una columna puede contener valores **NULL.** La restricción **NOT NULL** impone una columna para **NO** aceptar valores **NULL**.

CREATE TABLE <tabla> (<columna><tipo de dato> [tamaño] **NOT NULL**,);

# **52.-** **CREATE**

CREATE es un statement que nos permite crear tanto una tabla como una nueva base de datos.

Sintaxis:

* CREATE DATABASE <database>;
* CREATE TABLE <tabla> (

<columna1> <tipo de dato>,

<columna2> <tipo de dato>,

<columna3> <tipo de dato>);

Ejemplo:

CREATE DATABASE bd\_test;

CREATE TABLE personas (

id\_personas INT PRIMARY KEY, nombre VARCHAR (20) NOT NULL,

identidad INT UNIQUE, edad);

INSERT INTO personas(nombre,identidad,edad) VALUE

( ‘Pedro’, 86586, 30)

( ‘Juan’, 65833, 45);

# 53.- ALTER

ALTER es un statement que nos ayudará a realizar cambios dentro de las columnas de una tabla.

* ALTER TABLE <tabla> ADD <columna><tipo de dato>;
* ALTER TABLE <tabla> rename to <nombre>;
* ALTER TABLE <tabla> rename COLUMN <columna1> to <columna2>;
* ALTER TABLE <TABLE> MODIFY COLUMN <columna><tipo de dato>;

Ejemplo:

ALTER TABLE personas

ADD FOREIGN KEY (idtipo) REFERENCES tipo(idtipo);

**54.- DROP**

DROP es un statement que nos ayudará a eliminar una tabla.

* DROP TABLE<tabla>

Si queremos eliminar una columna tenemos que hacer:

* ALTER TABLE <nombre de la tabla>
* DROP COLUMN <nombre de la columna>

Ejemplo de eliminación de columna

* ALTER TABLE personas
* DROP COLUMN eliminame

Si existe una clave foránea en la tabla no se puede eliminar dicha tabla por estar relacionada.

# 55. - TRUNCATE

TRUNCATEes un statement que nos ayudará a vaciar una tabla.

* TRUNCATE TABLE <tabla>

TRUNCATE

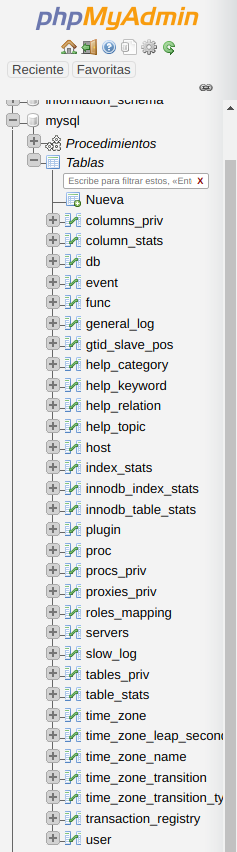
FROM tipo

**El TRUNCATE reinicia el Primary Key**

# 56. - INTRODUCCIÓN A LOS USUARIOS

* El usuario de la base de datos es necesario para identificar quién realizó un inicio de sesión a una base de datos.
* El usuario es la entidad de seguridad a la que se les puede asignar permisos sobre los objetos de la base de datos.

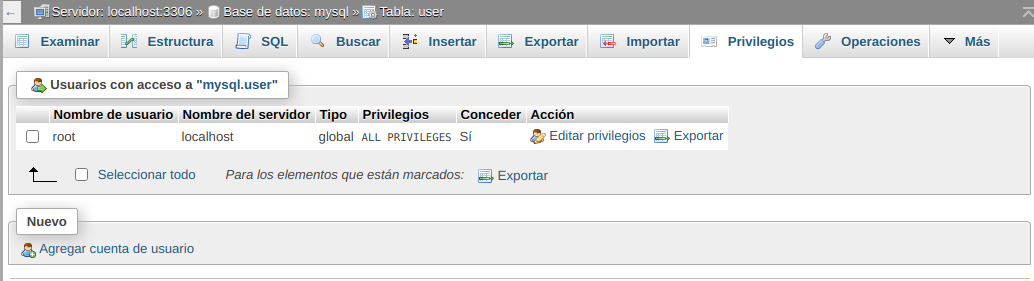
Mediante los privilegios del usuario **root** podemos consultar la base de datos **mysql** y dentro de ella la tabla **users**



En la pestaña **Cuentas de usuarios** podemos listar todos los usuarios registrados en el servidor

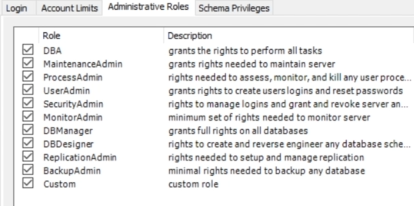


Una vez seleccionada la tabla user podemos visualizar los privilegios en la pestaña **Privilegios**



**Seleccionar** al usuario **root** y hacer clic en **Editar Privilegios**

**Roles Administrativos**

****

**Muestra las bases de datos del servidor.**

* SHOW DATABASES;

**Muestra todas las tablas de la base de datos seleccionada**

* USE mysql;
* SHOW FULL TABLES; (muestra todas las tablas de esa base de datos)
* SELECT
* FROM user;

# **57.- CREACIÓN DE USUARIOS**

Query:

CREATE USER <usuario> IDENTIFIED BY

“<password>”;

**Ejemplo:**

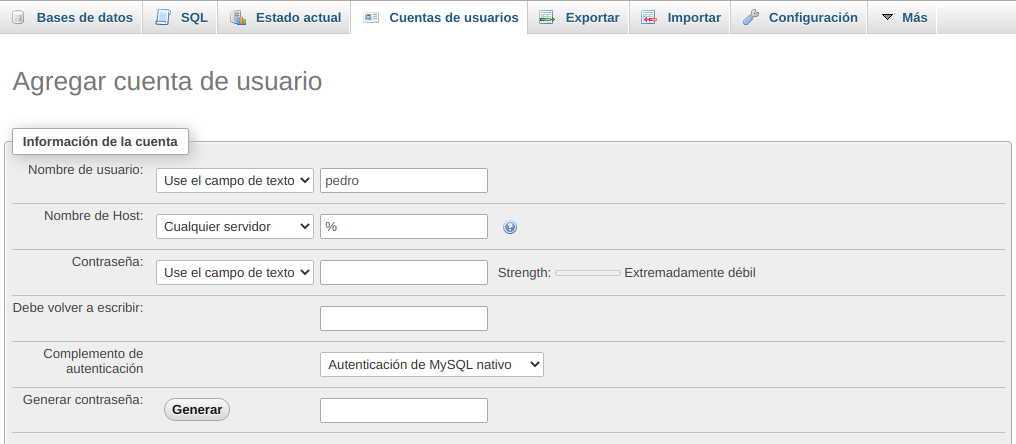
[**CREATE USER**](https://mariadb.com/kb/en/library/create-user/) pedro IDENTIFIED **BY**

"1234";

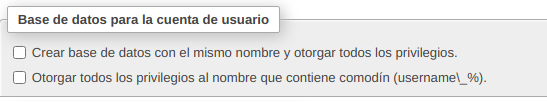
**Por interfaz gráfica:**



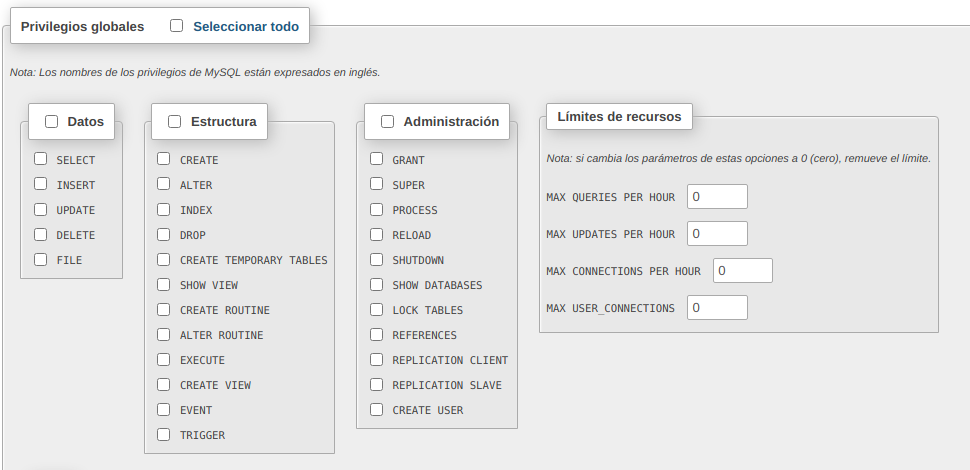
Nombre de usuario, Host, contraseña y autenticación:



En caso de querer crear una base de datos para el usuario:



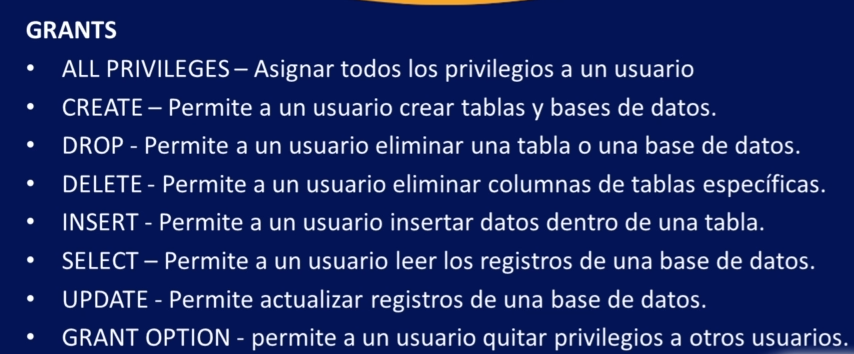
Permisos y límite de recursos:



Tipo de conexión:



# **58.- PRIVILEGIOS BÁSICOS**



# **59.- ADMINISTRACIÓN DE PRIVILEGIOS**

Existen dos statements que nos ayudarán en la administración de privilegios de un usuario:  
  
 **- GRANT (Dar permisos)**

GRANT permiso ON <base de datos> TO usuario@'nombre de host';

GRANT permiso ON <base de datos.tabla> TO usuario@'nombre de host';

GRANT permiso ON <tabla> TO usuario@'nombre de host';

-- Si quiero dar permisos de SELECT para todas las tablas de 'sakila' sakila.\*

GRANT SELECT ON sakila.\* TO pedro@'localhost';

-- Para darle permisos SELECT en todas las bases de datos del servidor

GRANT SELECT ON \*.\* TO pedro@'localhost';

**- REVOKE (Quitar permisos)**

REVOKE permiso ON <base de datos> FROM usuario@'nombre de host';

REVOKE permiso ON <base de datos.tabla> FROM usuario@'nombre de host';

REVOKE permiso ON <tabla> FROM usuario@'nombre de host';

REVOKE SELECT ON \* FROM pedro@'localhost';

REVOKE UPDATE ON \* FROM pedro@'localhost';

REVOKE ALL PRIVILEGES ON \*.\* FROM pedro@'localhost';

## 

## **## Consulta para crear/modificar el usuario y/o su host…**

**## y también eliminarlo**

CREATE USER 'jc'@'localhost' IDENTIFIED BY PASSWORD '\*D5AACBA2F825D6F79C6BEE77862009F391C1B298';

GRANT SELECT ON \*.\* TO 'jc'@'localhost';

GRANT UPDATE ON `sakila`.`actor` TO 'jc'@'localhost';

DROP USER 'jc'@'%';

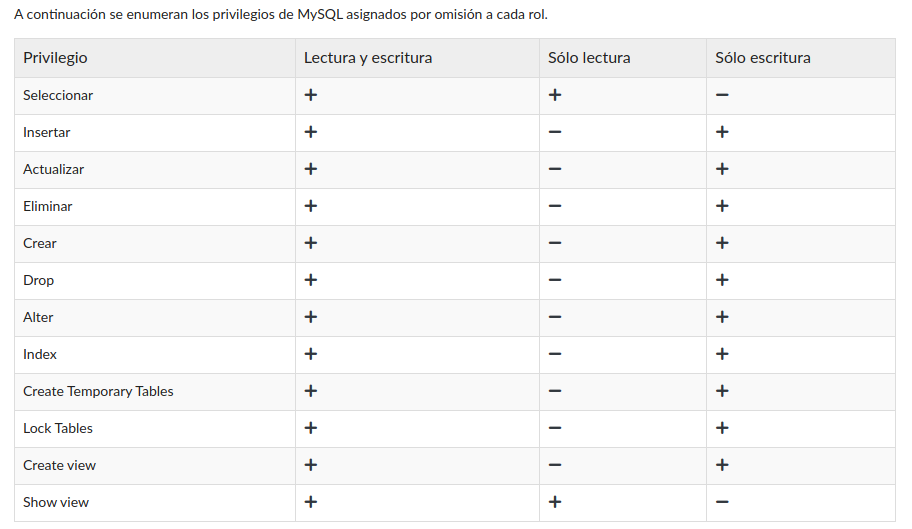
jc@localhost

jc@30.25.26.290

jc@el\_nombre\_del\_servidor\_que\_sea

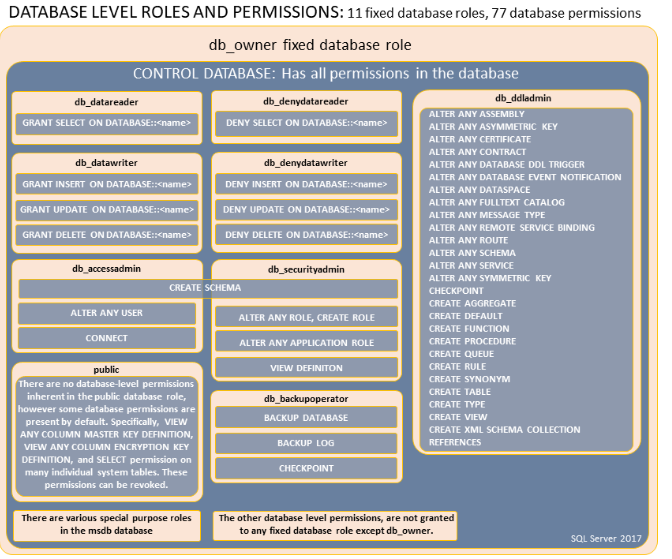
# **60.- ROLES DE LOS USUARIOS**

* Los roles son utilizados para agrupar usuarios con mismos privilegios
* Cada gestor de base de datos puede tener roles por defecto como es el caso de MySQL.



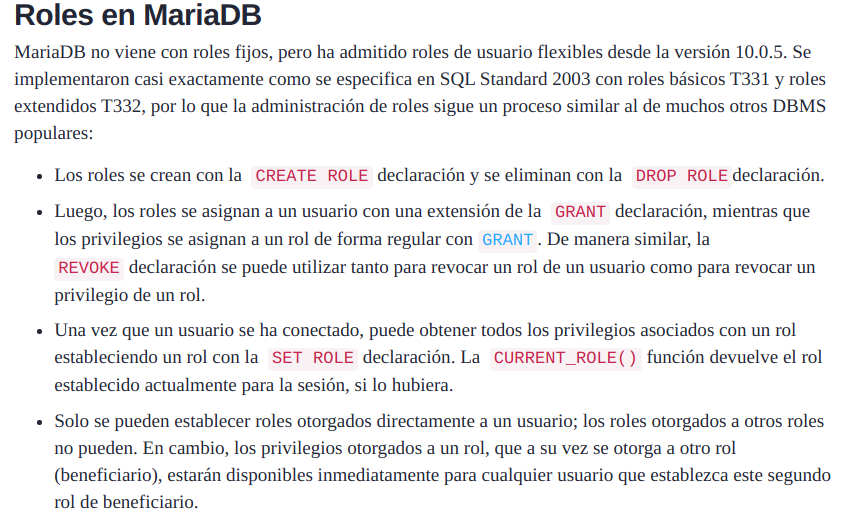
<https://docs.plesk.com/es-ES/obsidian/administrator-guide/servidores-de-bases-de-datos/permisos-y-roles-de-los-usuarios-de-bases-de-datos.74697/>

<https://mariadb.com/kb/en/roles/>

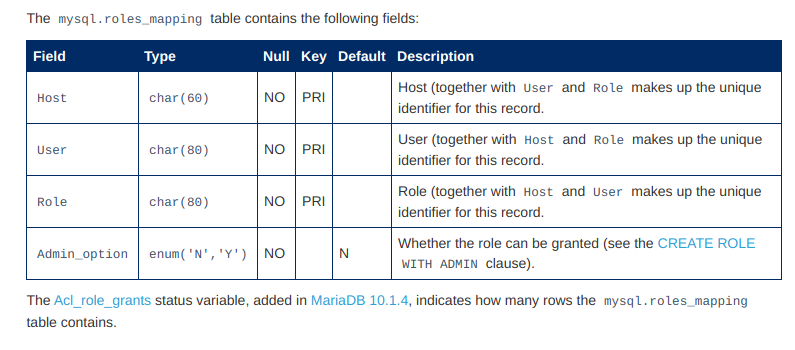


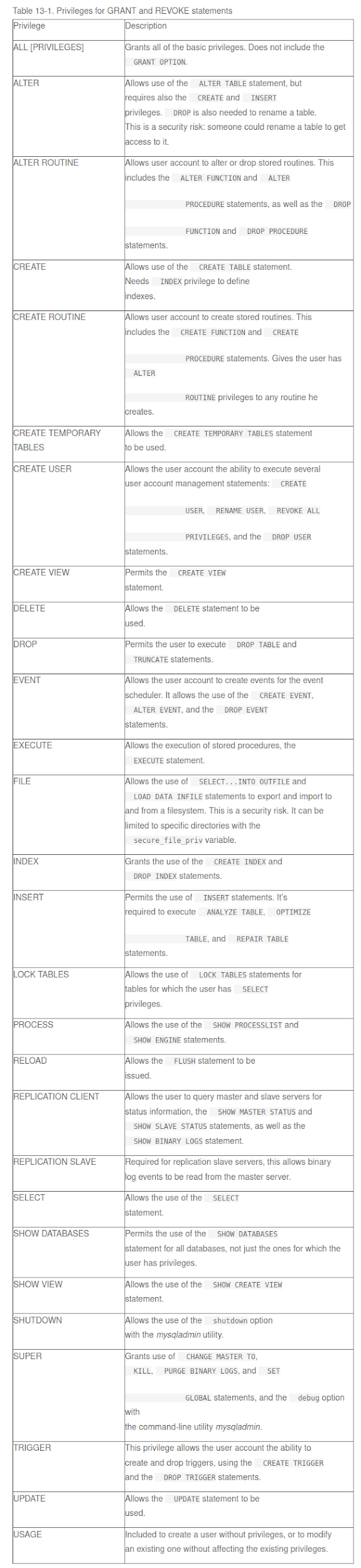
<https://docs.microsoft.com/es-es/sql/relational-databases/security/authentication-access/database-level-roles?view=sql-server-ver15> (Roles sql)

<https://mariadb.org/wp-content/uploads/2018/07/MariaDB-Roles-Tampere-Unconference-2018.pdf> (Roles mariadb)



<https://dzone.com/articles/mysql-80-vs-mariadb-comparison-of-database-roles>





# **61.- ELIMINACIÓN DE USUARIOS**

Para eliminar un usuario se debe seguir la siguiente estructura:

|  |
| --- |
| DROP USER <usuario>@<hostname> |

# **62.- INTRODUCCIÓN A TRIGGERS**

* Un desencadenador (o Trigger) es una clase especial de procedimiento almacenado que se ejecuta automáticamente cuando se produce un evento en el servidor de bases de datos.
* Recordar que un trigger es un objeto relacionado a una tabla que es ejecutado cuando sucede un evento en la tabla a la que va asociado. Son aquellas sentencias (INSERT, UPDATE, DELETE) que modifican los datos dentro de una tabla.

# **63.- OBJETOS Y EVENTOS**

* En los Triggers existen 2 objetos:
  + OLD
  + NEW
* En los Triggers existen 3 eventos:
  + INSERT
  + DELETE
  + UPDATE

Ejemplo de aplicación:

Persona

id 1

nombre Pedro

|  |
| --- |
| UPDATE persona  SET nombre = ‘Jorge’  WHERE id = 1 |

OLD.nombre = Pedro

NEW.nombre = Jorge

BEFORE / AFTER

# **64.- TRIGGER SYNTAX**

La sintaxis a utilizar es la siguiente:

|  |
| --- |
| CREATE TRIGGER <nombre>  AFTER/BEFORE <statement>  ON <tabla>  FOR EACH ROW  STATEMENT; |

**Crear nueva base de datos:**

CREATE DATABASE prueba;

USE prueba;

CREATE TABLE persona(

idpersona int PRIMARY KEY,

nombre VARCHAR(20),

dinero int

);

CREATE TABLE tienda(

idtienda int PRIMARY KEY,

item VARCHAR(20),

stock int,

precio int,

ganancias int

);

CREATE TABLE ventas(

idpersona int,

idtienda int,

FOREIGN KEY (idpersona) REFERENCES persona(idpersona),

FOREIGN KEY (idtienda) REFERENCES tienda(idtienda)

);

**Insertar datos en las tablas:**

INSERT INTO persona

(idpersona,nombre,dinero)

VALUES (1,'Pedro',100),(2,'Javier',150);

INSERT INTO tienda

(idtienda,item,stock,precio,ganancias)

VALUES (1,'Vaso',10,5,0);

**Trigger para actualizar el dinero de la tabla “persona” cada vez se haga una venta:**

CREATE TRIGGER persona\_compra

AFTER INSERT

ON ventas

FOR EACH ROW

UPDATE persona AS p

INNER JOIN ventas AS v

ON p.idpersona = v.idpersona

INNER JOIN tienda AS t

ON t.idtienda = v.idtienda

SET dinero = dinero - t.precio

WHERE p.idpersona = NEW.idpersona;

**Trigger para cambiar el inventario en “tienda” al hacer una venta:**

CREATE TRIGGER tienda\_stock

AFTER INSERT

ON ventas

FOR EACH ROW

UPDATE tienda

SET stock = stock - 1

WHERE idtienda = NEW.idtienda;

**Trigger para aumentar ganancias en “tienda” al hacer una venta:**

CREATE TRIGGER tienda\_ganancias

AFTER INSERT

ON ventas

FOR EACH ROW

UPDATE tienda

SET ganancias = ganancias + precio

WHERE idtienda = NEW.idtienda;

**Pruebas:**

INSERT INTO ventas

(idpersona,idtienda)

VALUES (1,1);

INSERT INTO ventas

(idpersona,idtienda)

VALUES (2,1);

Ejemplo encontrado en el siguiente enlace

<https://www.adictosaltrabajo.com/2008/07/11/triggersql/>,

De cómo realizar un TRIGGER, con explicaciones de cada paso que se va realizando:

A continuación y **a modo de consulta** os voy a mostrar un ejemplo de un Trigger DML que realiza las siguientes tareas:

Dada una tabla con información sobre «expedientes», se creará un Trigger que controle las modificaciones del «estado del expediente» de la siguiente manera:

1. Anotará en el campo «stateChangedDate» la fecha/hora en la que se produjo un cambio de estado.
2. A modo de histórico, insertará un registro en la tabla «expStatusHistory» con información sobre los cambios de estado de cada expediente.

DELIMITER $$

USE db\_test;

$$

# Creamos el Schema si no existe

CREATE SCHEMA IF NOT EXISTS db\_test;

$$

-- Eliminamos el procedimiento almacenado si existiese

DROP PROCEDURE IF EXISTS db\_test.procedureTemp;

$$

CREATE PROCEDURE db\_test.procedureTemp()

BEGIN

DECLARE cuenta INT DEFAULT 0;

-- Si no existe la tabla de expedientes, la creamos.

SELECT COUNT(\*) INTO cuenta FROM `information\_schema`.`tables` WHERE TABLE\_SCHEMA='db\_test' AND TABLE\_NAME='expedientes' LIMIT 1;

IF (cuenta = 0) THEN

CREATE TABLE `expedientes` (

code VARCHAR(15) NOT NULL COMMENT 'Código del expediente',

state VARCHAR(20) COMMENT 'Estado del expediente',

stateChangedDate DATETIME COMMENT 'Fecha/Hora en la que se produjo el último cambio de estado',

PRIMARY KEY `PK\_Exp` (code)

) ENGINE=InnoDB CHARSET=utf8 collate=utf8\_general\_ci;

END IF;

-- Insertamos algunos expedientes de ejemplo

DELETE FROM expedientes WHERE code IN ('exp1','exp2', 'exp3');

INSERT INTO expedientes (code) VALUES ('exp1');

INSERT INTO expedientes (code) VALUES ('exp2');

INSERT INTO expedientes (code) VALUES ('exp3');

-- Si no existe la tabla de cambios de estado la creamos

SELECT COUNT(\*) INTO cuenta FROM `information\_schema`.`tables` WHERE TABLE\_SCHEMA='db\_test' AND TABLE\_NAME='expStatusHistory' LIMIT 1;

IF (cuenta = 0) THEN

CREATE TABLE `expStatusHistory` (

`id` INT AUTO\_INCREMENT,

`code` VARCHAR(15) NOT NULL COMMENT 'Código del expediente',

`state` VARCHAR(20) NOT NULL COMMENT 'Estado del expediente',

`date` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP COMMENT 'Fecha/Hora en la que el expediente pasó a ese estado',

PRIMARY KEY `PK\_ExpHistory` (`id`)

) ENGINE=MyISAM CHARSET=utf8 collate=utf8\_general\_ci; -- No transacciones => MyISAM

END IF;

END;

$$

-- Invocamos el procedimiento almacenado

CALL db\_test.procedureTemp();

$$

-- Borramos el procedimiento almacenado

DROP PROCEDURE IF EXISTS db\_test.procedureTemp;

$$

-- Borramos el Trigger si existiese

DROP TRIGGER IF EXISTS StatusChangeDateTrigger;

$$

-- Creamos un Trigger sobre la tabla expedientes

CREATE TRIGGER StatusChangeDateTrigger

BEFORE UPDATE ON expedientes FOR EACH ROW

BEGIN

-- ¿Ha cambiado el estado?

IF NEW.state != OLD.state THEN

-- Actualizamos el campo stateChangedDate a la fecha/hora actual

SET NEW.stateChangedDate = NOW();

-- A modo de auditoría, añadimos un registro en la tabla expStatusHistory

INSERT INTO expStatusHistory (`code`, `state`) VALUES (NEW.code, NEW.state);

END IF;

END;

$$

DELIMITER;

# **65.- TRIGGER CONDICIONAL**

* La sintaxis a utilizar es la siguiente:

|  |
| --- |
| Delimiter$$ IF <condición> THEN  CREATE TRIGGER <nombre> <resultado>;  AFTER/BEFORE <statement> END IF;  ON <tabla>  FOR EACH ROW  BEGIN  Condicionales;  END; $$ |

**Ejemplo 1. No permitir stock negativo:**

DELIMITER $$

CREATE TRIGGER tienda\_stock\_validation

BEFORE UPDATE

ON data.tienda

FOR EACH ROW

BEGIN

IF NEW.stock < 0

THEN

SET NEW.stock = 0;

END IF;

END $$

INSERT INTO ventas (id\_persona, id\_tienda)

VALUES (1,1)

**Ejemplo 2. No permitir nombres vacíos:**

DELIMITER $$

CREATE TRIGGER persona\_validacion\_nombre

BEFORE UPDATE

ON persona

FOR EACH ROW

BEGIN

IF LENGTH(NEW.nombre) = 0

THEN

SET NEW.nombre = OLD.nombre;

END IF;

END $$

UPDATE persona

SET nombre = " "

WHERE id\_persona = 1;

# **66.- VER Y BORRAR TRIGGERS**

**Ver Trigger**

|  |
| --- |
| SHOW TRIGGER <nombre\_trigger> |

**Eliminar Trigger**

|  |
| --- |
| DROP TRIGGER <nombre\_trigger> |

# **67.- ERROR - NOT FOUND**

# **68.- CREACIÓN DE VISTAS**

La estructura es la siguiente:

|  |
| --- |
| CREATE VIEW <nombre\_vista>  AS <consulta> |

Ejemplos:

Utilizando Sakila, vamos a crear una primera lista de actores:

Primero recabamos toda la información con INNER JOIN.

SELECT a.first\_name, a.last\_name,

count(\*)

FROM actor a

INNER JOIN film\_actor fa

ON a.actor\_id = fa.actor\_id

INNER JOIN film f

ON f.film\_id = fa.film\_id

GROUP BY a.actor\_id;

Ahora crearemos la vista de películas hechas por un actor.

CREATE VIEW cant\_film\_by\_actor

AS SELECT a.first\_name AS Nombre,

a.last\_name AS Apellidos,

count(\*) AS Cantidad

FROM actor a

INNER JOIN film\_actor fa

ON a.actor\_id = fa.actor\_id

INNER JOIN film f

ON f.film\_id = fa.film\_id

GROUP BY a.actor\_id;

Para ejecutar la consulta recurrimos al nombre de la vista que hemos creado.

|  |
| --- |
| SELECT \* FROM cant\_film\_by\_actor; |

Encontraremos la vista entre las tablas con

|  |
| --- |
| SHOW TABLES; |

Es recomendable identificar los nombres de vistas con algún indicador propio para no confundirlas con las tablas.

Si necesitamos **eliminar la vista** utilizamos

|  |
| --- |
| DROP VIEW cant\_film\_by\_actor; |

Utilizando la base de datos Prueba hacemos lo siguiente para saber lo que ha gastado cada persona.

SELECT p.nombre AS “Nombre”,

COUNT(\*) AS “Cantidad de compras”,

COUNT(\*)\*t.precio AS “Dinero gastado”

FROM persona AS p

INNER JOIN ventas AS v

ON p.idpersona = v.idpersona

INNER JOIN tienda AS t

ON t.idtienda = v.idtienda

GROUP BY p.idpersona;

A continuación creamos la vista.

CREATE VIEW total\_gastado

AS SELECT p.nombre AS “Nombre”,

COUNT(\*) AS “Cantidad de compras”,

COUNT(\*)\*t.precio AS “Dinero gastado”

FROM persona AS p

INNER JOIN ventas AS v

ON p.idpersona = v.idpersona

INNER JOIN tienda AS t

ON t.idtienda = v.idtienda

GROUP BY p.idpersona;

Creamos otra vista con esta misma tabla:

SELECT p.nombre, p.dinero, p.dinero/5

AS ‘Cantidad de objetos que puede comprar’

FROM persona AS p

INNER JOIN ventas AS v

ON p.idpersona =v.idpersona

INNER JOIN tienda AS t

ON t.idtienda =v.idtienda

GROUP BY p.idpersona;

CREATE VIEW cantidad\_comprar

SELECT p.nombre, p.dinero, p.dinero/5

AS “Cantidad de objetos que puede comprar”

FROM persona AS p

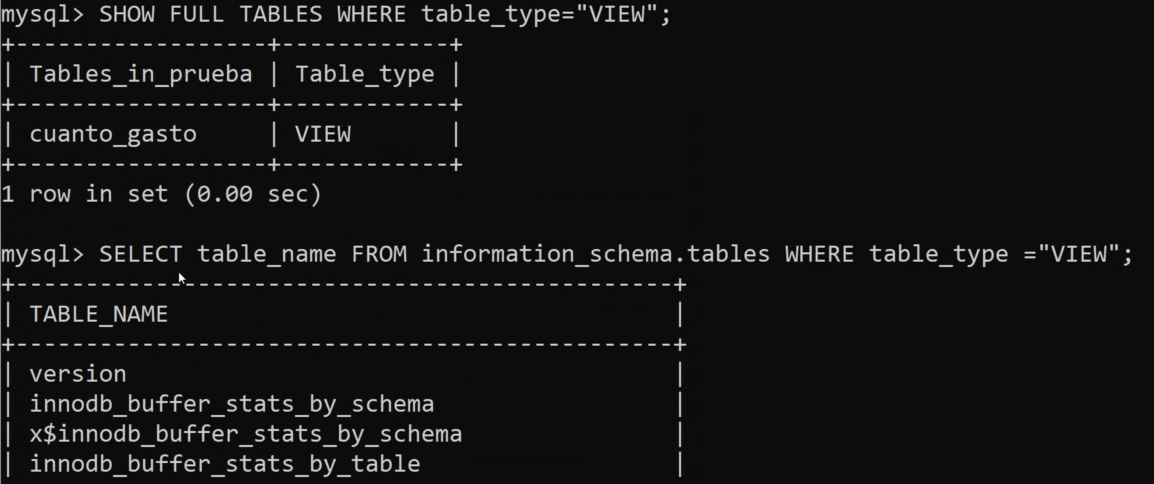
INNER JOIN ventas AS v

ON p.idpersona = v.idpersona

INNER JOIN tienda AS t

ON t.idtienda = v.idtienda

GROUP BY p.idpersona;



Podemos visualizar las vistas con estas instrucciones.

<https://mariadb.com/kb/en/user-defined-variables/>

Con Variables

SELECT nombre INTO @pexiste

from persona

where nombre = "Pedro"

--Ver lo que hay en la variable

SELECT \*

FROM persona

WHERE nombre = @pexiste

# **69 - WEB APPLICATION PENETRATION TESTING**

|  |
| --- |
| * Es la práctica de atacar diversos entornos con la intención de descubrir otros fallos de seguridad, para así poder prevenir ataques. Es una rama de estudio relativamente reciente y en auge. * Cuando el entorno de ataque es un sitio web. |



<https://owasp.org/?gclid=CjwKCAjw_NX7BRA1EiwA2dpg0igtOdJ5NvmZDAk86Elz8YG3cNsVnl3apVjpJ5EFkhqD6hCB-w47nxoC9hQQAvD_BwE>

# **70 - QUÉ ES SQLi**

|  |
| --- |
| La inyección SQL es una vulnerabilidad web que permite que un atacante interfiera con las consultas que una aplicación realiza en su BD. En general, permite que un atacante vea datos que normalmente no puede recuperar. Esto puede incluir datos que pertenecen a otros usuarios o cualquier otro dato al que la aplicación pueda acceder. En muchos casos, un atacante puede modificar o eliminar estos datos, causando cambios persistentes en el contenido o el comportamiento de la aplicación. |

# **71.- DESCARGA E INSTALACIÓN DE Metasploitable2**

**Descargar la ISO**

<https://sourceforge.net/projects/metasploitable/>

**Descarga VMWARE Workstation player**

<https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html>

**Credenciales:**

-msfadmin:msfadmin.

# **72.- LÓGICA DE SQLi**

Pensar en cómo se encuentra programada la página web.

|  |
| --- |
| SELECT \*  FROM user  WHERE iduser =’$VARIABLE’; |

Ejemplo:

Base de Datos - DVWA

Tabla - user

Columnas

id int primary key

first\_name varchar(20)

surname varchar(20)

Cual es su ID: 1

<input type =”text2 name=”id”>

SELECT id, first\_name, surname

FROM user

WHERE id =”$VARIABLE”; jeurhwer (comentario)

Cual es su ID: 1”;#

<input type =”text2 name=”id”>

SELECT id, first\_name, surname

FROM user

WHERE id =”1”;#”; jeurhwer (comentario)

El comentario se pone después del WHERE para forzar la consulta.

# **73.- BÚSQUEDA DE CAMPOS VULNERABLES A SQLi**

Tenemos que buscar parámetros

# 

# 

ÍNDICES

Los índices son un grupo de datos vinculados a una o varias columnas que almacena una relación entre el contenido y la fila en la que se encuentra. Con esto se agilizan las búsquedas en una tabla al evitar tener que recorrer toda la tabla para obtener los datos solicitados.

## VENTAJAS E INCONVENIENTES

### 

### VENTAJAS

Si una tabla tiene definido un índice sobre una columna se puede localizar mucho más rápidamente una fila que tenga un determinado valor en esa columna.

Recuperar las filas de una tabla de forma ordenada por la columna en cuestión también será mucho más rápido.

### 

### INCONVENIENTES

Al ser el índice una estructura de datos adicional a la tabla, ocupa un poco más de espacio en disco.

Cuando se añaden, modifican o se borran filas de la tabla, el sistema debe actualizar los índices afectados por esos cambios lo que supone un tiempo de proceso mayor.

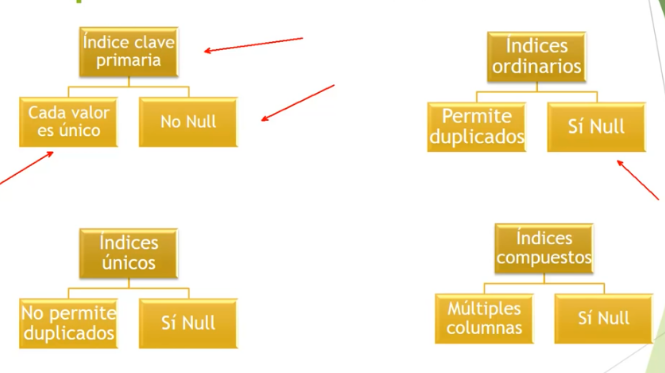
Por estas razones **no es aconsejable definir índices de forma indiscriminada**.

Fuente:<https://www.aulaclic.es/sql/b_8_4_1.htm>

# TIPOS DE ÍNDICES

Hay cuatro tipos principales de índices;

* Claves primarias (únicas y no nulas),
* Índices únicos (únicos y pueden ser nulos),
* Índices simples (no necesariamente únicos) e
* Índices de texto completo (para búsquedas de texto completo).



## PRIMARY KEY : CLAVE PRINCIPAL

Una clave principal es única y nunca puede ser nula. Siempre identificará solo un registro y cada registro debe estar representado. Cada tabla sólo puede tener una clave principal.

## UNIQUE : ÍNDICE ÚNICO

El modificador UNIQUE indica que la combinación de valores en las columnas indexadas debe ser único, pero puede ser nulo. Por tanto, cada valor de clave identifica solo un registro, pero no es necesario representar cada registro.

Los índices pueden contener más de una columna. MariaDB puede usar una o más columnas en la parte más a la izquierda del índice, si no puede usar todo el índice.

Para crear un índice único en una tabla, debe especificar la palabra clave UNIQUE al crear el índice. Nuevamente, esto se puede hacer con una instrucción CREATE TABLE o una instrucción

Por ejemplo:

|  |
| --- |
| -- El siguiente código crea un índice en una tabla existente  CREATE INDEX <index\_name>  ON <table\_name> (<column\_names>)  CREATE TABLE websites  ( website\_id INT(11) NOT NULL AUTO\_INCREMENT,  website\_name VARCHAR(25) NOT NULL,  server\_name VARCHAR(20),  creation\_date DATE,  CONSTRAINT websites\_pk PRIMARY KEY (website\_id),  UNIQUE INDEX websites\_idx (website\_name, server\_name)  );  O  CREATE TABLE websites  ( website\_id INT(11) NOT NULL AUTO\_INCREMENT,  website\_name VARCHAR(25) NOT NULL,  server\_name VARCHAR(20),  creation\_date DATE,  CONSTRAINT websites\_pk PRIMARY KEY (website\_id)  );  CREATE UNIQUE INDEX websites\_idx  ON websites (website\_name, server\_name); |

Ambos ejemplos crearían un índice único en los campos website\_name y server\_name de modo que la combinación de estos campos siempre debe contener un valor único sin duplicados. Esta es una excelente manera de hacer cumplir la integridad dentro de su base de datos si necesita valores únicos en columnas que no forman parte de su clave principal

## ÍNDICE ORDINARIO

Un índice que no es primario y permite valores duplicados (a menos que los campos hayan sido especificados como UNIQUE) .

Para crear un índice ordinario tenemos básicamente dos opciones:

Podemos crear un índice ordinario al mismo tiempo que creamos la tabla con el uso de la opción INDEX.

|  |
| --- |
| CREATE TABLE nombreTabla(campo1 tipoDato, campo2 tipoDato,..  INDEX [nombreIndice] (campo1 [,campo2...])); |

De igual manera, podemos crear el índice con el uso de la sentencia ALTER TABLE si es que la tabla ya existe.

|  |
| --- |
| ALTER TABLE nombreTabla ADD INDEX [nombreIndice] (campo1 [,campo2...]); |

También es posible usar la sentencia CREATE INDEX para crear un índice en una tabla existente.

|  |
| --- |
| CREATE INDEX nombreIndice ON nombreTabla(campo1 [,campo2,campo3...]); |

Lo que está entre corchetes es opcional

Ambas sentencias piden el nombre del índice, sin embargo con la sentencia CREATE INDEX el nombre es obligatorio.

## ÍNDICES DE TEXTO COMPLETO (FULLTEXT)

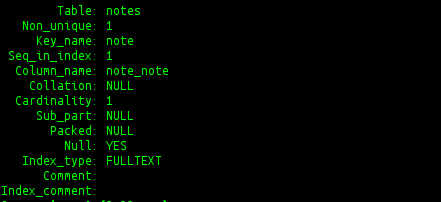
Estos índices FULLTEXT o de texto completo se usan en tablas del tipo MyISAM , InnoDB, Aria entre otros..., se debe relacionar con columnas de tipo TEXT, CHAR o VARCHAR.

Optimizan el rendimiento de las consultas con búsquedas en tablas con grandes cantidades de información en campos de texto.

No aplica ninguna restricción especial a los datos de la columna (o columnas) que componen el índice, sino que se emplea simplemente para mejorar el tiempo de ejecución de las consultas.





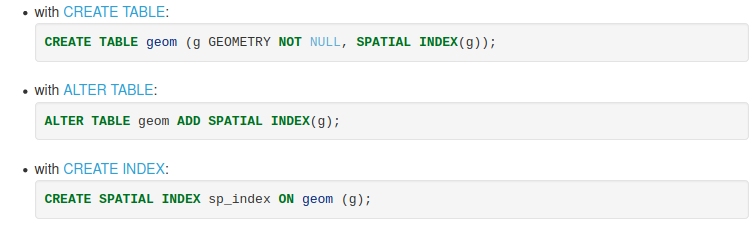


## SPATIAL INDEX

En las tablas MyISAM y Aria, así como en las tablas InnoDB de MariaDB 10.2.2, MariaDB puede crear índices espaciales (un índice de árbol R) usando una sintaxis similar a la de crear índices regulares, pero extendida con la palabra clave SPATIAL. Actualmente, las columnas de los índices espaciales deben declararse NOT NULL.

Estos índices se emplean para realizar búsquedas sobre datos que componen formas geométricas representadas en el espacio. Un índice espacial es un tipo de índice extendido que permite indizar una columna espacial. Una columna espacial es una columna de tabla que contiene datos de un tipo espacial, como **geometría** o **geografía**.

Es importante destacar que todos estos índices pueden construirse empleando una o más columnas.



<https://docs.microsoft.com/es-es/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-ver15#:~:text=Un%20%C3%ADndice%20espacial%20es%20un,espacial%2C%20como%20geometr%C3%ADa%20o%20geograf%C3%ADa>.

<https://geoinnova.org/blog-territorio/el-secreto-de-las-bases-de-datos-espaciales/>

<https://geotalleres.readthedocs.io/es/latest/postgis-indexacion-espacial/indexacion_espacial.html#:~:text=La%20indexaci%C3%B3n%20espacial%20es%20una,n%C3%BAmero%20de%20datos%20sean%20eficientes.&text=La%20indexaci%C3%B3n%20organiza%20los%20datos,la%20b%C3%BAsqueda%20de%20un%20registro>.

## ÍNDICE COMPUESTO

Los índices compuestos son simplemente aquellos que están basados en múltiples columnas. Si tenemos varias columnas que frecuentemente aparecen juntas en una cláusula WHERE, tenemos la oportunidad de acelerar estas consultas al crear un índice compuesto.

Para añadir un nuevo índice compuesto se puede hacer de la siguiente manera

|  |
| --- |
| alter table nombre\_tabla add index `nombre\_indice` (`campo1`, `campo2`[, `campo3`, `campo4`,...]) |

Esto puede dar lugar a veces a que el tamaño del índice que estamos generando sea demasiado grande.

Para ello vamos ajustando el tamaño que asignamos a cada campo para el índice.

|  |
| --- |
| alter table nombre\_tabla add index `nombre\_indice` (campo1[, campo2, campo3, …]); |

### TRES REGLAS PARA DEFINIR LOS ÍNDICES

Antes de indicar las reglas, es importante saber que las reglas son de aplicación independientemente del algoritmo a utilizar (tanto B-Tree como Fractal Trees) y de la estructura de las tablas. Además, es muy importante seleccionar la consulta o consultas más frecuentes que va a tener la tabla, para adecuar los índices a dichas consultas, además del coste de mantenimiento de los índices.

Además, no hay una regla fija, sino que son:

* **Recupera cuando menos datos puedas**: cuanto más datos a recuperar, más trabajo, más ancho de banda, más recursos, … basada en la cláusula WHERE.

Para poder aplicar esta regla, hay que estudiar la cláusula WHERE, y lo mejor es que **los campos de igualdades estén al principio del índice**, lo que reduce el número de filas a procesar. Si no hay una igualdad, el orden de los campos del índice no asegura el recuperación de las menos filas posibles

* **Evita consultas puntuales**: el acceso secuencial es mucho más rápido que el acceso puntual.

Si bien es cierto que el coste por registro de recorrer toda una tabla es bajo, el precio por los registros buscados es alto, precisamente por tener que recorrer toda la tabla. Sin embargo, es deseable que en el índice también esté incluido campos sobre los que requerimos operaciones, de forma que los datos estén ya presentes en el índice, y no tenga que volver a la tabla a buscarlos. Es por ello, que para evitar el coste de buscar otras columnas en la tabla, los índices deberían incluir todos los campos usados en la sentencia completa, y no sólo los de la cláusula WHERE.

* **Evita el ordenado**, ya que los operadores GROUP BY y ORDER son aplicados una vez recuperados los datos, y por tanto, implica más trabajo.

La mejor forma de evitar el ordenado es usando esos campos en los índices, lo que ya pre ordena estos datos, de forma que la consulta pueda ser ejecutada más rápidamente.

Aunque no se ha mencionado aún, es posible diseñar índices por varios campos, pero hay que tener en cuenta que en la consulta **debe respetarse el orden de los campos del índice** para que el funcionamiento del índice sea óptimo, y que el **orden de los campos del índice es muy importante**.

Referencia: <https://www.manejandodatos.es/2016/04/mecanismo-los-indices-mysql-mariadb/>

# LONGITUD DE ÍNDICES

# <https://use-the-index-luke.com/es/sql/agrupacian/solo-escanear-indices#:~:text=Los%20%C3%ADndices%20MyISAM%20se%20limitan,de%20clave%20a%201%2C000%20bytes>..

# OPERACIONES SOBRE ÍNDICES

## CREAR ÍNDICE

Para crear un índice, se empleará la siguiente estructura:

|  |
| --- |
| CREATE INDEX <index\_name>  ON <table\_name> (<column\_names>) |

Referencia: <https://www.mariadbtutorial.com/mariadb-index/mariadb-create-index/>

Donde:

* + **index\_name:** es el nombre del índice.
  + **table\_name:** es el nombre de la tabla donde se va a crear el índice.
  + **column\_names:** nombre de la columna (o columnas) que formarán el índice.

Veamos un ejemplo de cómo crear un índice en MariaDB usando la declaración

CREATE TABLE. Esta declaración crearía la tabla y el índice al mismo tiempo.

Por ejemplo:

|  |
| --- |
| CREATE TABLE websites  ( website\_id INT(11) NOT NULL AUTO\_INCREMENT,  website\_name VARCHAR(25) NOT NULL,  server\_name VARCHAR(20),  creation\_date DATE,  CONSTRAINT websites\_pk PRIMARY KEY (website\_id),  INDEX websites\_idx (website\_name)  ); |

En este ejemplo, hemos creado la tabla de sitios web , así como un índice llamado websites\_idx que consta de la columna website\_name .

A continuación, le mostraremos cómo crear la tabla primero y luego crear el índice usando la declaración CREATE INDEX.

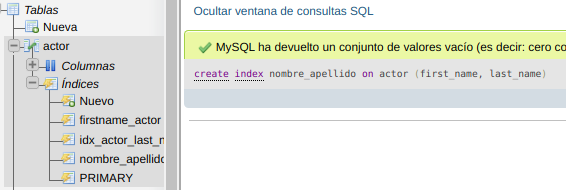
Por ejemplo:

|  |
| --- |
| CREATE TABLE websites  ( website\_id INT(11) NOT NULL AUTO\_INCREMENT,  website\_name VARCHAR(25) NOT NULL,  server\_name VARCHAR(20),  creation\_date DATE,  CONSTRAINT websites\_pk PRIMARY KEY (website\_id)  ); |

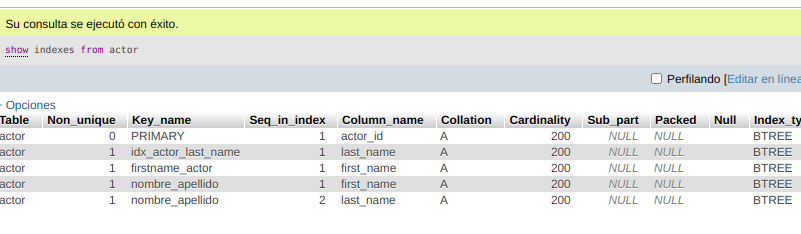
|  |
| --- |
| CREATE INDEX websites\_idx  ON websites (website\_name); |

En este ejemplo, la declaración CREATE TABLE creará la tabla de sitios web . La declaración CREATE INDEX crearía un índice llamado websites\_idx que consiste en el campo nombre\_sitio web.

Ejemplo en la base de datos Sakila:



Para comprobar que se ha creado el índice utilizamos la siguiente sentencia:



Otra opción es modificar una tabla ya existente, usando esta sintaxis:

|  |
| --- |
| **ALTER TABLE** table\_name  **ADD INDEX** index\_name (column\_name, column\_name2, ...); |

# 

## CAMBIAR EL NOMBRE DEL ÍNDICE

Puede cambiar el nombre de un índice en MariaDB usando la instrucción ALTER TABLE.

Sintaxis

La sintaxis para cambiar el nombre de un índice usando la instrucción ALTER TABLE es:

|  |
| --- |
| ALTER TABLE table\_name  DROP INDEX index\_name,  ADD INDEX new\_index\_name [ USING BTREE | HASH ]  (index\_col1 [(length)] [ASC | DESC],  index\_col2 [(length)] [ASC | DESC],  ...  index\_col\_n [(length)] [ASC | DESC]); |

Esta operación se puede hacer en dos pasos:

1º paso

|  |
| --- |
| ALTER TABLE table\_name  ADD INDEX new\_index\_name [ USING BTREE | HASH ]  (index\_col1 [(length)] [ASC | DESC],  index\_col2 [(length)] [ASC | DESC],  ...  index\_col\_n [(length)] [ASC | DESC]); |

2º paso

* Pregunto si existe el índice
* Si existe elimina el anterior

|  |
| --- |
| DROP INDEX index\_name, |

## MOSTRAR TODOS LOS ÍNDICES

La declaración show indexes le permite consultar índices de una tabla.Para mostrar los índices asociados con una tabla, utilice la siguiente declaración:

|  |
| --- |
| **show** **indexes** **from** table\_name; |

Si no está conectado a ninguna base de datos específica, puede especificar el nombre de la base de datos en la declaración show indexes como se muestra en el siguiente comando:

|  |
| --- |
| **show** **indexes** **from** table\_name  **in** database\_name; |

O:

|  |
| --- |
| **show** **indexes** **from** database\_name.table\_name; |

El siguiente statement muestra todos los índices de la tabla countries:

|  |
| --- |
| show indexes from countries; |

Mostrar algunos índices de un ejemplo de tabla

El siguiente ejemplo usa la show indexes declaración para buscar **sólo** índices únicos de la tabla countries:

|  |
| --- |
| show indexes from countries  where non\_unique = 0; |

## 

## ELIMINAR INDICE

Referencia:

<https://www.mariadbtutorial.com/mariadb-index/mariadb-drop-index/>

Sintaxis:

|  |
| --- |
| **drop** **index** [**if** **exists**] index\_name  **on** table\_name |

En esta sintaxis:

* Primero, especifique el nombre del índice que desea eliminar después de las drop index palabras clave. La if exists cláusula opcional elimina condicionalmente el índice sólo si existe.
* En segundo lugar, especifique el nombre de la tabla a la que se asocia el índice después de la palabra clave.

## 

## 

## 

## CLÁUSULAS CON **INDEX**

**USE** INDEX

**FORCE** INDEX

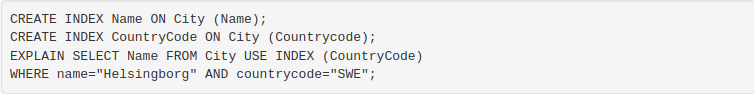
**IGNORE** INDEX

Ejemplo:

### **USE INDEX**

El USE INDEX solo afecta cuando se optimiza la cláusula WHERE.

USE INDEX se utiliza después del nombre de la tabla en la cláusula FROM.

****

EXPLAIN muestra (explica) cómo son procesadas las sentencias SELECT por mariadb, como se usan los índices, y cómo se unen las tablas. Utilizar EXPLAIN puede ayudarnos a seleccionar mejores índices y escribir nuestras consultas más óptimamente. Lo único que tenemos que hacer es agregar la palabra EXPLAIN al inicio de la consulta para que mariadb nos diga cómo la está ejecutando. En vez de ejecutar la consulta, mariadb reportará la lista de índices que se podrían usar en la consulta y lo que conoce acerca de ellos.



**IGNORE** **INDEX**: Se utiliza después del nombre de la tabla en la cláusula FROM

|  |
| --- |
| CREATE INDEX Name ON City (Name);  CREATE INDEX CountryCode ON City (Countrycode);  EXPLAIN SELECT Name FROM City IGNORE INDEX (Name)  WHERE name="Helsingborg" AND countrycode="SWE"; |

IGNORE INDEX Indica al optimizador de consultas que no se utilice el índice en la consulta.

### **FORCE INDEX**

Se usa FORCE INDEX en la tabla para decirle a mariadb que use el índice dado.

Si hemos creado un índice sobre alguna tabla y al momento de ejecutar alguna consulta notamos que en el plan de ejecución no se hace uso de este índice, podemos forzar la utilización del mismo de la siguiente manera:

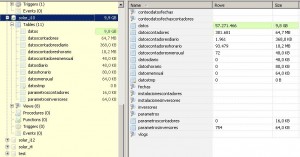


OPTIMIZADOR DE CONSULTAS

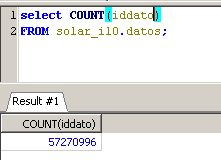
El optimizador de consultas es un componente del servidor de base de datos que hace el plan de ejecución más óptimo para una declaración SQL. El optimizador de consultas utiliza las estadísticas disponibles para elaborar el plan que tiene el costo más bajo entre todos los planes candidatos.Por ejemplo, una consulta puede solicitar productos cuyos precios estén entre 10 y 80. Si las estadísticas muestran que el 80% de los productos tienen estos rangos de precios, entonces puede decidir que un escaneo completo de la tabla es el más eficiente. Sin embargo, si las estadísticas muestran que muy pocos productos tienen estos rangos de precios, leer un índice seguido de un acceso a la tabla podría ser más rápido y eficiente que un escaneo completo de la tabla.

#### FORZANDO ÍNDICES.ANÁLISIS DE UN CASO REAL EN MYSQL

Dentro de un proyecto de investigación en la [Universidad de Córdoba](http://www.uco.es), trabajamos con una base de datos en MySQL que tiene unos cuantos registros, en torno a **9 Gb**, de los que el 95% se concentran en una sola tabla «*datos*«.



En esta tabla se guardan datos con una frecuencia de 5 minutos, pero necesitamos calcular los datos agregados horarios. Para año y medio de datos, tenemos unos cuantos millones de datos, concretamente 57 millones.



##### DATOS

Nos metemos en faena, pero comprobamos los índices disponibles:



##### FORZANDO ÍNDICES

La consulta que tenemos prevista es la siguiente:

SELECT DATE\_FORMAT( fecha,'%Y-%m-%d %H:00:00') AS fecha, dc.dia, dc.anno, dc.idparametro, dc.idinstalacioninversor,AVG(dc.valor) AS prom, SUM(dc.valor) AS suma, MAX(dc.valor) AS maximo, MIN(dc.valor) AS minimo, AVG(dc.valor) as valorFROM solar\_i10.datos dcINNER JOIN solar\_comun.parametros p using (idparametro)WHERE p.agregado < 5 AND dc.anulado = 0 AND dc.idinstalacioninversor = 417 AND dc.anno = 2011GROUP BY dc.idinstalacioninversor, dc.idparametro, dc.dia, dc.hora ORDER BY DATE\_FORMAT( fecha,'%Y-%m-%d %H:00:00');

##### ANALIZANDO LA CONSULTA CON **EXPLAIN**:



###### **EXPLAIN**

Sintaxis en mariadb:

|  |
| --- |
| EXPLAIN tbl\_name |

Fuente: https://mariadb.com/kb/en/explain/

A priori, los criterios que aplica MySQL son bastante buenos. En este caso, ha seleccionado el **índice** *fecha\_idparametro\_idinstalacioninversor* dato que la fecha es usado en la cláusula **ORDER BY** y los campos *idparametro* e *idinstalacioninversor* son usados en la cláusula **WHERE**. En total, estamos trabajando con 3.914.722 registros!

Sin embargo, la ejecución de la misma se hace eterna, alcanzando los 70 minutos (en la siguiente imagen vemos que va por los 1269 segundos, esto es unos 20 minutos …. y decidimos parar la ejecución!).



###### CONSULTA LENTA

Si observamos la consulta, la cláusula **WHERE** incluye 3 campos (*anulado, idinstalacioninversor y anno*) que constituyen un índice de la tabla, así que … **¿por qué no usarlo en lugar del que me propone MySQL?**

Modificamos la consulta SQL, pero ahora hay que indicarle que queremos utilizar un índice concreto, usando [FORCE INDEX](http://dev.mysql.com/doc/refman/5.6/en/index-hints.html):

SELECT DATE\_FORMAT( fecha,’%Y-%m-%d %H:00:00′) AS fecha, dc.dia, dc.anno, dc.idparametro, dc.idinstalacioninversor,AVG(dc.valor) AS prom, SUM(dc.valor) AS suma, MAX(dc.valor) AS maximo, MIN(dc.valor) AS minimo, AVG(dc.valor) as valor FROM solar\_comun.parametros p INNER JOIN solar\_i10.datos dc **FORCE INDEX** (anno\_idinstalacioninversor\_anulado) using (idparametro) WHERE p.agregado < 5 AND dc.anulado = 0 AND dc.idinstalacioninversor = 417 AND dc.anno = 2011 GROUP BY dc.idinstalacioninversor, dc.idparametro, dc.dia, dc.hora

ORDER BY DATE\_FORMAT( fecha,’%Y-%m-%d %H:00:00′);

La nueva consulta tiene algunas modificaciones respecto a la original, marcadas en rojo por cambios en los **INNER JOIN** (esto no afecta), pero ahora sí incluimos la cláusula **FORCE INDEX** y el índice que queremos utilizar (en este caso, *nno\_idinstalacioninversor\_anulado*).

Al igual que antes, verificamos qué nos cuenta MySQL cuando ejecutamos un EXPLAIN de la nueva consulta:



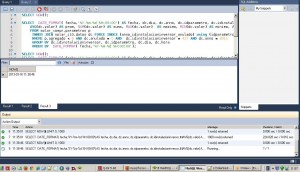
###### EXPLAIN con FORCE INDEX

La información de MySQL para la nueva consulta selecciona el índice que le hemos obligado a tomar, pero nos toma más registros a analizar (4.313.310 registros). **Hay una diferencia de más de 360.000 registros de diferencia entre uno y otro, habiendo más en esta nueva versión de la consulta**.

Sin embargo, el tiempo de ejecución de la consulta baja a solo 10-12 minutos!!

Esto supone un ahorro bastante considerable en tiempo, por lo que en este caso concreto, **MySQL no ha seleccionado el mejor índice!**

Comprobamos los tiempos ejecutando ambas consultas desde MySQL WorkBench:



Comprobamos que la consulta modificada aquí solo tarda 224 segundos (porque solo devuelve 1000 registros), mientras que la consulta original la cortamos cuando vamos por 20 minutos (empezó a las 11:38 y abajo a la derecha marca ya las 11:57)!!

Por último, comentar que **es muy importante la explicación (EXPLAIN) que nos da MySQL de las consultas**, y … si se alargan en el tiempo es porque hay algo que se puede mejorar, y he aquí un caso concreto de mejora del rendimiento de una consulta forzando a MySQL a usar un índice que es seleccionado por el usuario.

Fuente: <https://www.manejandodatos.es/2013/10/forzando-indices-en-mysql-5-6-10/>

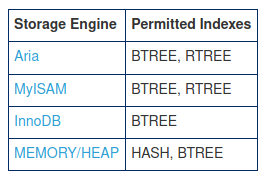
# ESTRUCTURAS PARA ALMACENAR ÍNDICES

## ESTRUCTURAS DE DATOS DE TIPO ÁRBOL

Si utilizas el tipo de tecnología de almacenamiento InnoDB, sólo soporta índices de tipo BTREE.

Si utilizas MyISAM como tecnología de almacenamiento, éste admite índices de tipo BTREE y RTREE. Según el libro [k]Pro MySQL[/k] de Michael Kruckenberg y Jay Pipes, el tipo de índice RTREE se utiliza principalmente para tipos de datos espaciales (coordenadas geográficas).

Por lo que, en resumidas cuentas, casi siempre utilizarás BTREE como tipo de índice. De hecho, si utilizas "MySQL Administrator", cuando creas una tabla MySQL verás que si dejas el tipo de índice como DEFAULT, te pondrá BTREE de forma automática.

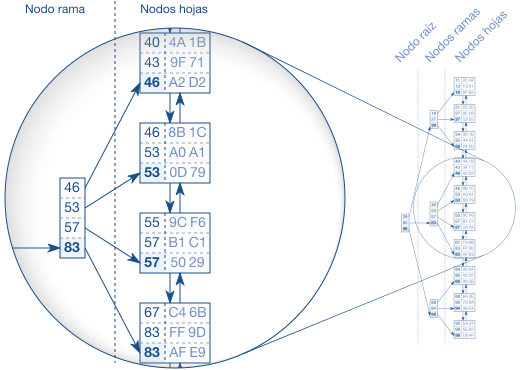


### BTREE

Los índices B-Tree se utilizan para comparaciones de columnas mediante los operadores>,> =, =,> =, < o BETWEEN, así como para comparaciones LIKE que comienzan con una constante.

Los términos 'CLAVE' e 'ÍNDICE' generalmente se usan indistintamente, y las declaraciones deben funcionar con cualquier palabra clave.

Los nodos hojas del índice están almacenados en un orden aleatorio, es decir su posición en el disco no corresponde a la posición lógica según el orden del índice. Es lo mismo que una guía telefónica con las hojas mezcladas. Si buscas “Sánchez” pero primero abres la guía telefónica por “Rodríguez”, nada nos asegura que Rodríguez siga a Sánchez. Una base de datos necesita una segunda estructura para encontrar rápidamente los datos dentro de las hojas mezcladas: un árbol de búsqueda equilibrado, o sea, un B-Tree.



Muestra un ejemplo de un índice con 30 entradas. La lista doblemente enlazada establece el orden lógico entre los nodos hojas. La raíz y sus ramas permiten hacer una búsqueda rápida entre los nodos hojas.

Una vez creado el índice, la base de datos lo mantiene automáticamente. Se aplican cada insert, delete y update al índice y se conserva el árbol equilibrado, lo que genera una sobrecarga de mantenimiento para las operaciones de escritura.



el mismo número de etapas, y en segundo lugar, al crecimienEsto es una muestra de un fragmento de un índice para ilustrar la búsqueda de la llave “57”. El recorrido del árbol empieza desde el nodo raíz del lado izquierdo. Cada entrada está procesada en el orden ascendente hasta encontrar un valor superior o igual (>=) al valor buscado (57). En esta ilustración, se trata del valor 83. La base de datos sigue la referencia sobre el nodo rama correspondiente y repite el proceso hasta que el recorrido del árbol alcanza un nodo hoja.

El recorrido del árbol es una operación muy eficiente. De hecho es tan eficiente que me refiero a ella como la primera potencia de la indexación. Funciona de manera casi instantánea incluso con grandes volúmenes de datos. Ello se debe principalmente a la característica equilibrada del árbol, lo que permite tener acceso a todos los elementos conto logarítmico de la profundidad del árbol. Eso significa que la profundidad del árbol crece lentamente en comparación al número de hojas. Hay índices reales con millones de registros que tienen una profundidad de cuatro o cinco. Es poco común encontrar una profundidad de seis. El apartado sobre la “Escalabilidad logarítmica” lo describe más en detalle

### HASH

Los índices hash, por el contrario, solo se pueden usar para comparaciones de igualdad, por lo que aquellos que usan los operadores = o <=>.

HASH es un mapa de valor-clave desordenado. Es incluso más eficiente que BTREE: O (1) en lugar de O (log n); Pero no tiene ningún concepto de orden. Eso significa que no se puede utilizar para operaciones de clasificación o para buscar rangos.

### R-TREE

RTREE es un 'índice espacial' que significa que puede identificar rápidamente valores 'cercanos' en 2 o más dimensiones. Se utiliza en bases de datos geográficas para consultas como "todos los puntos dentro de X metros desde (x, y)".

## ***CÓMO CAMBIAR MOTOR DE BASE DE DATOS PHPMYADMIN:***

<https://www.neoguias.com/cambiar-motor-base-de-datos-tabla-mysql/#:~:text=Si%20usas%20phpMyAdmin%2C%20tambi%C3%A9n%20lo,prefieras%20y%20guarda%20los%20cambios>.

**Cómo cambiar el motor MySQL por defecto:**

Cuando creas una tabla dentro de una base de datos, ésta utilizará el motor de base de datos que esté establecido por defecto en MySQL si no lo seleccionas explícitamente. Vamos a ver cómo saber cuál es el motor de base de datos por defecto de MySQL y cómo puedes cambiarlo.

### Saber cuál es el motor MySQL por defecto

Si quieres saber cuál es el motor de bases de datos de MySQL que está establecido por defecto, [conéctate a mysql a través de la consola](https://www.neoguias.com/como-conectarse-mysql-desde-linea-comandos/) *(mysql -u nombre\_usuario -p)* y ejecuta esta sentencia:

SHOW ENGINES;

Se mostrará un listado con todos los motores soportados junto con su description y características que soporta cada uno. El motor que está establecido por defecto es el que se indica como DEFAULT.

### Cambiar el motor MySQL por defecto

|  |
| --- |
| *Es posible cambiar el motor que MySQL utiliza por defecto para así no tener que configurar individualmente cada base de datos o cada tabla. Una vez cambiado, todas las tablas que crees utilizarán el motor seleccionado, a no ser que escojas otro durante la creación de las mismas.* |

Para cambiar el motor de base de datos por defecto sigue estos pasos:

1. Editar el archivo **my.cnf** de tu sistema local o de tu servidor. Este [archivo](https://www.neoguias.com/archivo/) estará en la carpeta ***/etc*** en sistemas CentOS y Fedora, o en la carpeta ***/etc/mysql*** en sistemas Ubuntu y Debian.
2. El archivo my.cnf está dividido en varias secciones. Vete a la sección [mysqld] y localiza la siguiente línea, en donde MOTOR será el motor por defecto que está establecido actualmente:  
   default-storage-engine=MOTOR
3. Debes reemplazar MOTOR por el nombre del motor que quieres utilizar, que habitualmente será MyISAM o InnoDB. *En caso de que escojas el motor InnoDB, asegúrate de que la línea skip-innodb está comentada*. Puedes comentarla con un símbolo sostenido tal que así:  
   #skip-innodb
4. Guarda el archivo **my.cnf** y sal del editor de texto.
5. Ahora debes reiniciar el servidor MySQL para que se cargue la nueva configuración y los cambios surtan efecto. Puedes hacerlo con uno de estos dos comandos:
   * El sistemas CentOS y Fedora:  
     service mysqld restart
   * En sistemas Ubuntu o Debian:  
     service mysql restart

Una vez se reinicie MySQL, [conéctate a mysql](https://www.neoguias.com/como-conectarse-mysql-desde-linea-comandos/) y ejecuta la siguiente consulta para verificar si el motor establecido por defecto se ha cambiado correctamente:  
SHOW ENGINES

INYECCIONES SQL

Inyección SQL es un método de infiltración de código intruso que se vale de una vulnerabilidad presente en una aplicación en el nivel de validación de las entradas para realizar operaciones sobre una base de datos. El origen de la vulnerabilidad radica en la incorrecta comprobación o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL. Es, de hecho, un error de una clase más general de vulnerabilidades que puede ocurrir en cualquier lenguaje de programación o script que esté embebido dentro de otro. Se conoce como Inyección SQL, indistintamente, al tipo de vulnerabilidad, al método de infiltración, al hecho de incrustar código SQL intruso y a la porción de código incrustado. Se dice que existe o se produjo una inyección SQL cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos(1)

(1()Información recogida de: h[tp://diposit.ub.edu/dspace/bitstream/2445/143419/2/memoria.pdf](http://diposit.ub.edu/dspace/bitstream/2445/143419/2/memoria.pdf)

<https://es.stackoverflow.com/questions/10518/qu%C3%A9-es-la-inyecci%C3%B3n-sql-y-c%C3%B3mo-puedo-evitarla>

<https://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL>

<https://www.php.net/manual/es/security.database.sql-injection.php>

<https://softwarelab.org/es/que-es-inyeccion-sql/>

Ejemplos de ataques SQL inyection : <https://diego.com.es/ataques-sql-injection-en-php>

<https://backtrackacademy.com/articulo/inyeccion-sql-definicion-y-ejemplos>

**SCRUM Y XP DESDE LAS TRINCHERAS**

**Cómo hacemos Scrum**

## **1. No incluido, corresponde a prólogos y comentarios.**

## **2. Cómo hacemos Pilas de Producto**

La pila del producto es el corazón del Scrum. Es una lista priorizada de requisitos o funcionalidades que el cliente quiere y para la que crearemos la terminología del cliente: a esto lo denominaremos “historia”.

Nuestras historias incluyen los siguientes campos:

* **ID**. Identificador único, simplemente un número incremental con el que reconocer cada. historia.
* **Importancia**. Ratio de importancia y prioridad, indicado por el dueño del producto. Es recomendable que sea numérico, teniendo como indicador de prioridad máxima es el número más alto.
* **Estimación inicial**. Valoración inicial del equipo acerca de cuánto trabajo es necesario para implementar la historia. La unidad son “puntos de historia”, y usualmente corresponde a “días-personas ideales”. Lo importante es que las estimaciones absolutas sean correctas (es decir, que una historia de dos puntos deba durar dos días).
* **Cómo probarlo**. La descripción a alto nivel de cómo se demostrará esta historia en la Demo al final del Sprint. Se trata, esencialmente, en la especificación de un test. La tabla que elaboramos la mantenemos en un documento Excel con “compartir” habilitado pues, aunque oficialmente el propietario es el Dueño de Producto, no queremos dejar al resto de los usuarios fuera.

**Campos de historia adicionales.**

A menudo utilizamos campos adicionales en la Pila de Producto para facilitar al dueño del producto el elegir sus prioridades.

* **Categoría.** Categorización básica de la historia; de este modo el Dueño del Producto puede filtrar fácilmente la información.
* **Componentes**. Usualmente implementado en la forma de “checkboxes” en el documento Excel. Aquí el Dueño de Producto puede identificar qué componentes técnicos estarán involucrados en la implementación de la historia. Es útil cuando hay varios equipos Scrum y queremos que sea fácil para cada equipo saber a qué historias deben dedicarse.
* **Solicitante**. El Dueño de Producto puede querer mantener un historial acerca de qué cliente o persona interesada pidió originalmente la historia.
* **Bug tracking ID**. Si se tiene aparte un “bug tracking” (seguimiento de errores) es útil mantener un historial de correspondencia directa entre una historia y uno o más errores reportados.

## **3. Cómo nos preparamos para la planificación del Sprint.**

**Asegurarse de que la Pila de Producto está perfectamente elaborada antes de la reunión de planificación de Sprint:**

* La pila de producto debe existir.
* Debería haber una Pila de Producto y un Dueño de Producto.
* Todos los elementos importantes deberían tener ratios de importancia asignados, cualquier historia sobre la que el Dueño de Producto piense que tiene la más remota posibilidad de incluirse en el Sprint debería tener un nivel de importancia único definido. También otras personas además del Dueño de Producto pueden añadir sus historias a la Pila del Producto, pero no pueden asignarles niveles de importancia (eso es cometido exclusivo del Dueño de Producto) ni tampoco establecer estimaciones (es un cometido exclusivo del equipo).
* Otras aproximaciones.
  + Usar un sistema de seguimiento de errores para almacenar la Pila del Producto. Excel resulta muy útil.
  + Utilizar una herramienta de soporte a procesos ágiles.

## **4. Cómo hacemos la planificación de Sprint.**

La planificación del Sprint es una reunión crítica, probablemente la más importante del Scrum. Su propósito es proporcionar al equipo información suficiente como para que pueda trabajar en paz y sin interrupciones durante unas semanas.

Una planificación de Sprint produce:

* *Una meta de Sprint.*
* *Una lista de miembros y su nivel de dedicación.*
* *Una Pila de Sprint (lista de historias incluidas en el Sprint).*
* *Una fecha concreta para la Demo del Sprint.*
* *Un lugar y momento definidos para el Scrum Diario.*

## **Por qué debe asistir el Dueño del Producto:**

A veces los Dueños de Producto se resisten a pasar horas con el equipo preparando la preparación del Sprint. La razón por la que deberían asistir es que cada historia contiene tres variables que son muy dependientes unas de otras:

* *Alcance.*
* *Estimación.*
* *Importancia.*

El alcance y la importancia los fija el Dueño del Producto, la estimación la proporciona el equipo; en algunos casos la estimación no será la que el Dueño de Equipo esperaba, lo que puede forzarle a cambiar la importancia de la historia o su alcance, obligando al equipo a re-estimarla, etc, etc…

¿Pero qué sucede si, durante todo el Desarrollo Ágil de software, el Dueño de Producto es remiso a intervenir en las reuniones de planificación del Sprint? Sería recomendable utilizar alguna de estas estrategias, en el siguiente orden:

* Tratar de convencerlo de que su presencia es crucial.
* Intentar que alguien del equipo se presente como delegado del Dueño del Producto durante la reunión.
* Tratar de convencer a la gerencia de que presente a un nuevo Dueño del Producto.
* Posponer el lanzamiento del Sprint hasta que el Dueño de Producto disponga de tiempo para asistir a la reunión. Mientras tanto no se ha de comprometer ninguna entrega.

## **Por qué la calidad no es negociable.**

Hemos dejado intencionadamente fuera de las anteriores variables la de la calidad. Distingamos ahora entre calidad interna y externa.

* **Calidad externa***.* La que perciben los usuarios.
* **Calidad interna***.* Aspectos que habitualmente no son visibles por los usuarios, pero que tienen un profundo efecto sobre la mantenibilidad del sistema.

¿Y cómo distinguimos la diferencia entre aspectos de calidad externa y otros de calidad interna?

Supongamos que el Dueño de Producto pretende que hagamos un “apaño” rápido para hacerlo en la mitad de tiempo; en este caso está intentando utilizar la calidad interna como una variable.

Eso no lo podemos permitir; sacrificar la calidad interna supone un daño inasumible pues, si permitimos que una base de código comience a deteriorarse desde su inicio, es muy difícil volver a conseguir su calidad más adelante.

*Cuando el Dueño de Producto aprende que la calidad interna no es negociable suele hacerse muy bueno en manipular las otras variables.*

## **Reuniones de planificación de Sprint que duran, y duran…**

Todo es Scrum tiene, en teoría, una duración determinada (time-boxed); entonces, ¿qué podemos hacer cuando la reunión de planificación de Sprint está llegando a su final y no hay atisbos de una meta de Sprint o de Pila de Sprint?

Si no se ha producido un plan de Sprint tras 2-8 horas de reunión probablemente no se logrará en una hora más, pero siempre se debe proponer la solución de alargarla y, si no se acepta, continuar al día siguiente a una hora determinada. El Sprint sufrirá… pero también será una lección más en el desarrollo.

*Aprende a mantener tus duraciones determinadas, aprende a establecer duraciones realistas. Esto se debe de aplicar tanto a las reuniones como a los Sprints.*

## **Agenda de la reunión de planificación de Sprint.**

Es esencial tener algún tipo de agenda u orden del día de la reunión de planificación de Sprint. Este ejemplo nos puede ser de utilidad:

Reunión de planificación de Sprint: 13:00 - 17:00, con diez minutos de descanso entre cada hora.

* *13:00 - 13:30*. El Dueño del Producto comenta la meta del Sprint y resume la Pila del Producto. Se establecen fecha, lugar y hora para la Demo.
* *13:30 - 15:00*. El equipo da estimaciones de tiempo y divide los elementos tanto como sea necesario. El Dueño de Producto analiza los ratios de importancia. Se clarifican los elementos, y para todos los elementos de alta importancia se establece “cómo probarlo”.
* *15:00 - 16:00*. El equipo selecciona las historias que se incluirán en el Sprint. Se realizan cálculos de velocidad para chequear si es factible.
* *16:00 - 17:00*. Se seleccionan lugar y hora para el Scrum diario. Se continúa dividiendo las historias en tareas.

## **Definiendo la duración del Sprint.**

Generalmente los Dueños de Producto prefieren los Sprints cortos y los desarrolladores los sprints largos,, así que su duración ha de ser una solución de compromiso. Es recomendable que el equipo experimente desde un principio con los tiempos, eso lo llevará a ir mejorando y decidirse finalmente por un período estándar: si, tras varios proyectos, se llega a un óptimo de “x” semanas, se recomienda mantenerlo para los próximos trabajos, mantenerlo como norma.

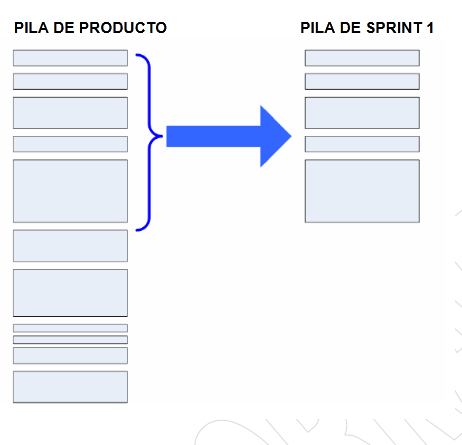
**Definiendo la meta del Sprint.**

Por alguna razón es difícil conseguir una meta del Sprint, pero es bueno definirla, describiéndola en términos de negocio. Debería ser algo que no se haya logrado aún, incluso marcarla a mitad del Sprint, cuando la gente comienza a sentirse confusa acerca de lo que deberían de estar haciendo.

**Decidiendo qué historias incluir en el Sprint**.

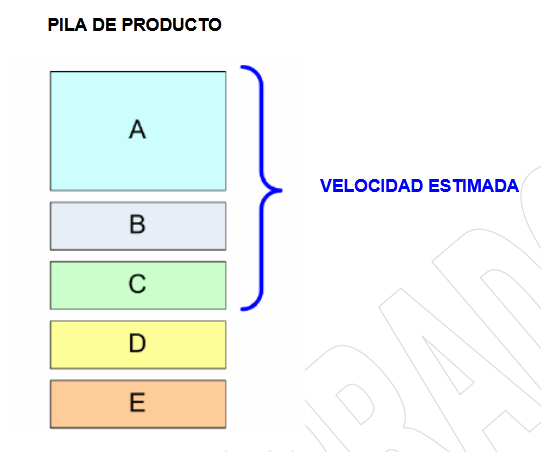
Una de las principales actividades durante la planificación de Sprint es decidir qué historias se incluyen, más específicamente, qué historias de la Pila de Producto se han de copiar en la Pila de Sprint.

En el siguiente gráfico cada rectángulo representa una historia, y se han ordenado por importancia. La más importante está al principio de la lista. El tamaño de cada rectángulo representa el tamaño de la historia (es decir, el tiempo estimado en puntos de historia). La altura del corchete azul representa la velocidad estimada del equipo, es decir, cuántos puntos de historia cree el equipo que puede completar durante el próximo Sprint. La Pila de Sprint de la derecha es una instantánea de las historias la Pila de Producto. Representa las historias a las que el equipo se compromete durante este Sprint. El equipo decide cuántas historias incluirá en el Sprint.

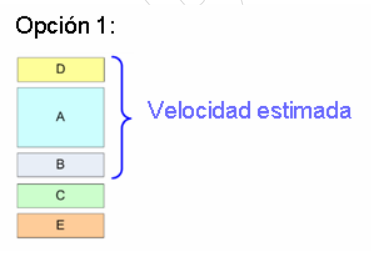


**¿Cómo puede el Dueño de Producto alterar las historias que se incluyen en el Sprint?**

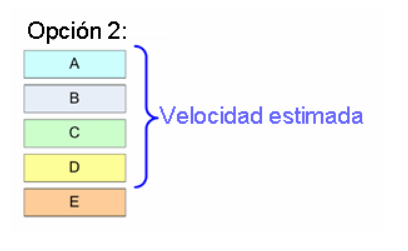
Al Dueño de Producto no le gusta que la historia D no se vaya a incluir en el Sprint. ¿Cuáles son sus opciones durante la reunión de planificación de Sprint?



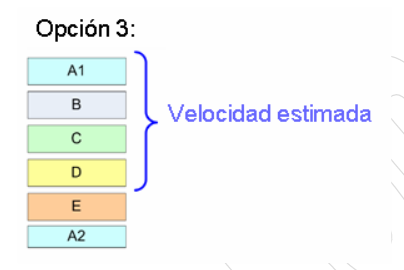
Una es re-priorizar; si le da al elemento D la mayor importancia, el equipo se verá obligado a añadirlo al Sprint en primer lugar (descartando la historia C)



La segunda opción es cambiar o reducir el alcance de la historia A hasta que el equipo crea que la historia D puede caber en el Script.



La tercera sería dividir una historia; el Dueño de Producto podría decidir que hay algunos aspectos de la historia A que no son tan importantes, así que se dividiría la historia A en dos historias A1 y A2 con diferentes niveles de importancia.



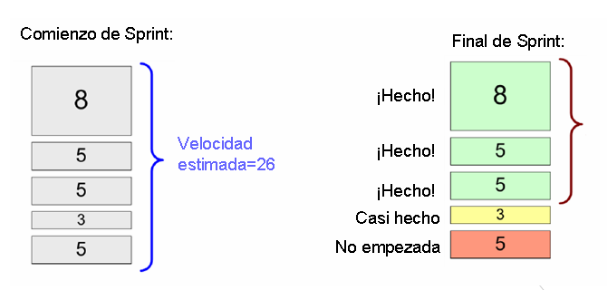
**¿Cómo decide el equipo qué historias incluir en el Sprint?**

Utilizamos dos técnicas:

* **“A ojo de buen cubero”**. Funciona bastante bien en grupos pequeños y en Sprints cortos.
* **Estimando cálculos de velocidad.** Esta técnica consta de dos pasos:

1. *Decidir la velocidad estimada.*
2. *Calcular cuántas historias se pueden añadir sin superar la velocidad estimada.*

La velocidad es una medida de “cantidad de trabajo realizado”, donde cada elemento se evalúa en función de su estimación inicial. El siguiente gráfico muestra un ejemplo de velocidad estimada al principio de un Sprint y la velocidad real al final de dicho Sprint. Cada rectángulo es una historia y el número interior es su estimación inicial.



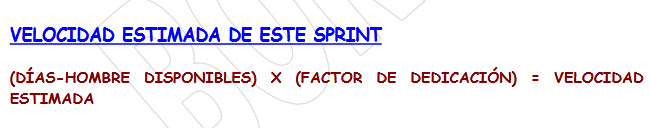
Nota que la velocidad real está basada en las estimaciones iniciales de cada historia, y que se ignora cualquier actualización a la estimación de la historia realizada durante el Sprint. Se trata de un número bastante aproximado, pero es bastante útil; sean cuales sean las razones, aquí está la diferencia entre cuánto se creía poder hacer y cuánto se hizo en realidad.

Un modo muy fácil de estimar la velocidad es revisar la historia del equipo, cuál fue su velocidad durante los últimos Sprints, y asumir que así será en el próximo. Esta técnica es factible para equipos que ya han hecho algunos Sprints (de forma que haya estadísticas disponibles) y que harán el próximo más o menos de la misma manera.

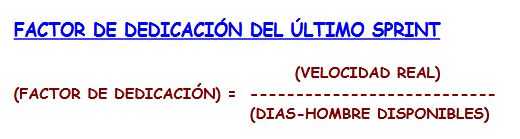
Una forma más sofisticada de hacerlo es realizar un cálculo de recursos. Digamos que estamos planificando un Sprint de 3 semanas (15 días laborables) con un equipo de 4 personas. Lisa estará de vacaciones 2 días. Dave sólo estará disponible al 50% y estará un día de vacaciones. Poniéndolo todo junto...



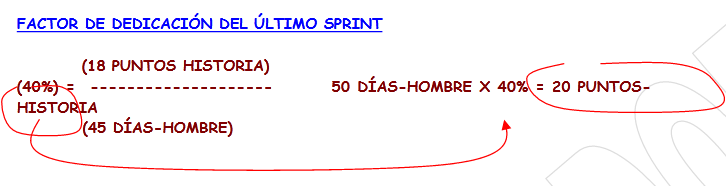
Pero esta no es la velocidad estimada; la unidad de estimación son puntos de historia, lo que corresponde más o menos a “días-hombre ideales”. Un día-hombre ideal es un día efectivo, sin distracciones, lo cual es raro. Además, se han de tener en cuenta cosas no planificadas que se añaden al Sprint, gente que se pone enferma, etc. Así que la velocidad estimada será sin duda menor de 50. ¿Pero cuánto menor? Para eso añadimos el “factor de dedicación”, que es una estimación de cómo de centrado va a estar el equipo.



La mejor manera de determinar un factor de dedicación razonable es estudiar el último Sprint (o incluso mejor, la media de los últimos Sprints).



La velocidad real es la suma de las estimaciones iniciales que se completaron en el último Sprint. Digamos que en el último Sprint se completaron 18 puntos de historia utilizando un equipo de 3 personas formado por Tom, Lisa y Sam trabajando 3 semanas hasta un total de 45 días-hombre. Y ahora estamos intentando calcular la velocidad del próximo Sprint. Para complicar las cosas, un nuevo tipo, Dave, se une al equipo para este Sprint. Teniendo en cuenta las vacaciones y demás asuntos tenemos 50 días-hombre ideales.

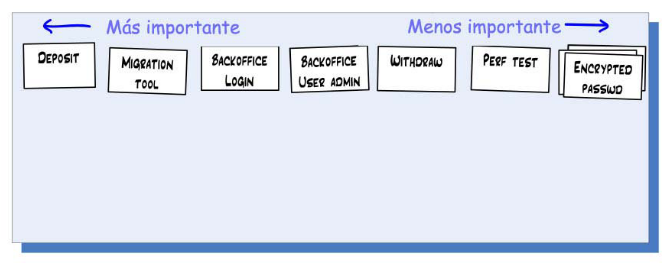


Así que nuestra velocidad estimada para el próximo Sprint es de 20 puntos de historia. Eso significa que el equipo debe añadir historias al Sprint hasta que sume aproximadamente 20. En este caso, el equipo puede escoger las 4 historias más importantes hasta un total de 19 puntos de historia, o las 5 historias más importantes hasta 24 puntos de historia. Digamos que escogen 4 historias, ya que se aproximan más a los 20 puntos de historia; ante la duda escoge añadir menos historias.

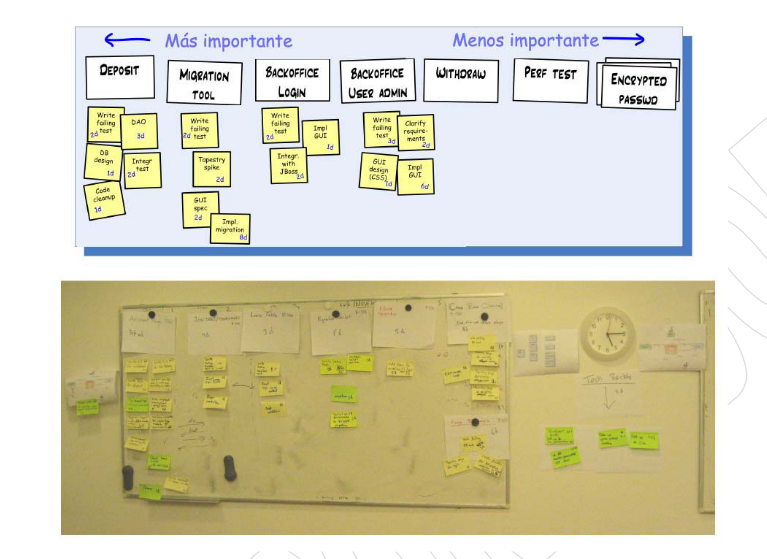
El factor de dedicación “por defecto” que usamos para equipos nuevos es habitualmente el 70%.

**Por qué usamos tarjetas.**

La mayor parte de la reunión de planificación de Sprint se dedica a las historias de la Pila de Producto, estimándolas, repriorizándolas, clarificándolas, dividiéndolas… ¿Cómo hacer en la práctica? Uno de los métodos que funciona es crear tarjetas y ponerlas en la pared (o en una mesa).



**Importante:** Tras la reunión de planificación de Sprint, nuestro ScrumMaster actualiza manualmente la Pila de Producto en Excel respecto a cualquier cambio que se haya realizado sobre las tarjetas de historia físicas.

****

No actualizamos la Pila de Producto en Excel respecto a nuestra división en tareas por dos razones: La división en tareas suele ser bastante volátil, es decir, se cambia y se refina con frecuencia durante el Sprint, así que es bastante molesto mantener la Pila de Producto en Excel sincronizada.

Al igual que con las tarjetas de historia, los post-it de tareas pueden reutilizarse directamente en la Pila de Sprint.

**Definición de “terminado”.**

Es importante que el Dueño de Producto y el equipo estén de acuerdo en una definición clara de “terminado”.

Hemos llegado a la conclusión de que todas las historias no se pueden tratar igual. Una historia llamada “formulario de consulta de usuarios” se tratará de una forma muy diferente a otra llamada “manual de operaciones.**”;** en este caso la definición de terminado puede significar simplemente “aceptada por el equipo de operaciones”. Es por eso que el sentido común es, a menudo, mejor que las listas de comprobación formales.

**Estimación de tiempos usando Planning Poker**

La estimación es una labor de equipo: todos los miembros del equipo deben involucrarse en estimar cada historia. ¿Por qué?

- *A la hora de planificar, normalmente no sabemos exactamente quién implementará qué partes de cada historia.*

*- Las historias normalmente involucran a bastantes personas y de diferentes áreas de experiencia (diseño de interfaz de usuario, programación, pruebas, etc.).*

*- Para poder proporcionar una estimación, un miembro del equipo necesita comprender de alguna forma de qué trata la historia. Pidiendo a todo el mundo que estime la historia nos aseguramos de que cada miembro del equipo comprende de qué trata cada elemento.*

*- Cuando pedimos a todo el mundo que de estimaciones muchas veces encontramos discrepancias. Es mejor descubrirlas al principio.*

Si le pides al equipo que proporcione una estimación, normalmente la persona que entiende mejor la historia será el primero en soltar una. Desafortunadamente, esto afectará severamente a las estimaciones de los demás. Hay una técnica excelente para evitar esto: se llama planning poker:

- *Cada miembro del equipo cuenta con una baraja de 13 cartas, como las que se muestran en la imagen. Cada vez que hay que estimar una historia cada miembro del equipo selecciona una carta que representa su estimación de tiempo (en puntos de historia) y la coloca boca abajo en la mesa.*

- *Cuando todos los miembros del equipo han preparado sus cartas, se les da la vuelta al mismo tiempo. Así obligamos a cada miembro del equipo a pensar por sí mismo, en lugar de seguir la estimación de otro.*

*- Si hay mucha discrepancia entre dos estimaciones, el equipo discute las diferencias y trata de construir una imagen común del trabajo necesario para la historia. Después, el equipo estima de nuevo. Este bucle se repite hasta que la estimación de tiempo converge.*

**Dividiendo historias en historias más pequeñas.**

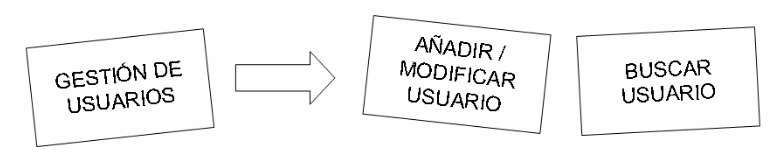
Las historias no deberían ser ni demasiado pequeñas ni demasiado grandes (en términos de estimación). Si tenemos un montón de historias de 0.5 puntos seremos víctimas de la microgestión; por otra parte, una historia de 40 puntos corre un importante riesgo de acabar parcialmente completa.

Casi siempre es posible dividir una historia grande en historias más pequeñas. Simplemente debemos de que las historias pequeñas siguen representando entregables con valor de negocio.

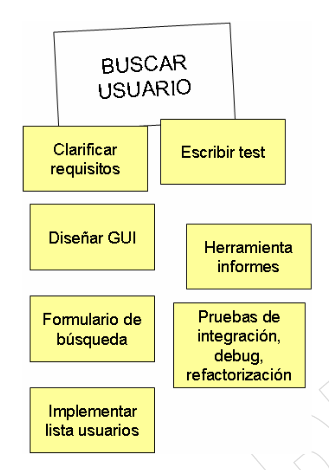
**Dividiendo las historias en tareas.**

¿Cuál es la diferencia entre historias y tareas? Las historias son entregables de los que el Dueño de Producto se preocupa, las tareas son no-entregables o aspectos de los que el Dueño de Producto no se preocupa.

*Ejemplo de división de una historia en historias más pequeñas:*



*Ejemplo de división de una historia en tareas:*



Este tipo de división frecuentemente revela trabajo adicional que hace que las estimaciones suban, con lo que se consigue un plan de Sprint más realista, hace que los Scrum diarios sean notablemente más eficientes; incluso si la división es inexacta y cambia una vez que empezamos, todas las ventajas anteriormente mencionadas siguen siendo válidas.

**Dónde trazar la línea.**

- Prioridad 1: Una meta de Sprint y una fecha para la demo son el mínimo necesario para comenzar un Sprint. El equipo tiene una meta y una fecha de finalización, y pueden trabajar directamente con la Pila de Producto.

- Prioridad 2: Lista de qué historias ha aceptado terminar el equipo en este Sprint.

- Prioridad 3: Una estimación para cada historia de Sprint.

- Prioridad 4: “Como probarlo”, relleno para cada historia del Sprint.

- Prioridad 5: Cálculos de velocidad y recursos, como chequeo de la planificación del Sprint. Incluyendo una lista de los miembros del equipo y sus compromisos.

- Prioridad 6: Un sitio y hora específicos para la realización del Scrum diario.

- Prioridad 7: Historias divididas en tareas. Esta división puede sin embargo hacerse diariamente durante los Scrum diarios, pero interferirá levemente en el flujo del Sprint.

**Historias técnicas (elementos no funcionales).**

Son cosas que deben hacerse pero que no son un entregable ni están directamente relacionadas con ninguna historia específica, y no son de valor inmediato para el Dueño de Producto. Por ejemplo:

* *Instalar un servidor de compilación continua.*
* *Escribir una descripción general del diseño.*
* *Refactorizar la capa de acceso a datos.*
* *Actualizar seguimiento de errores.*

Las historias técnicas suelen obtener mínima prioridad del Dueño de Producto. Por ello es recomendable:

* *Evitar las historias técnicas, transformándolas en historias normales con valor de negocio mensurable, así el Dueño de Producto tendrá mejores oportunidades para realizar decisiones correctas entre los pros y los contras.*
* *Si no podemos transformar una historia técnica en una historia normal tratemos de hacerla como una tarea dentro de otra historia.*
* *Si lo anterior falla, definirla como historia técnica y mantener una lista separada con dichas historias.*

**Sistema de seguimiento de errores vs. Pila de Producto.**

Excel es un formato estupendo para la Pila de Producto, pero aun así necesitas un sistema de seguimiento de errores, y Excel probablemente no funcione bien.

Para incluir los elementos del sistema de seguimiento de errores en una reunión de planificación de Sprint se pueden utilizar diversos métodos:

- El Dueño de Producto imprime los elementos del sistema de seguimiento de errores más importantes, los trae a la reunión de planificación de Sprint y los coloca en la pared junto al resto de historias.

- El Dueño de Producto crea historias que se refieren a los elementos del sistema de seguimiento de errores.

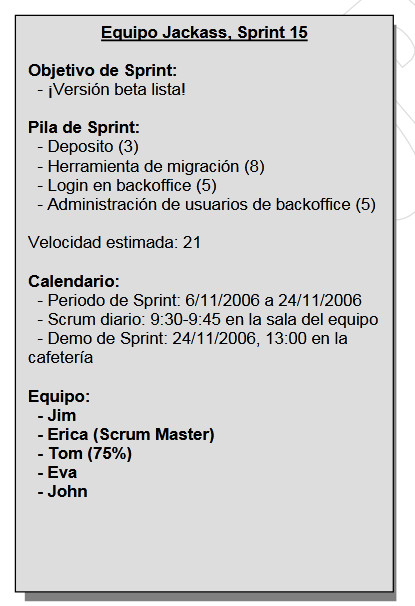
- La corrección de errores se considera algo fuera del Sprint, se asume que el equipo pasará parte de su tiempo arreglándolos.

**¡Por fin acabó la reunión de planificación de Sprint!**

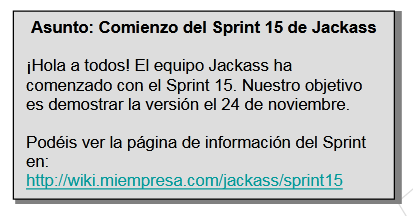
La reunión de planificación de Sprint es lo más importante que se hace en Scrum. Emplear una buena cantidad de esfuerzo en hacerla facilita el que todo lo demás resulte más fácil.

**5. Cómo comunicamos los Sprints**

Es importante mantener a toda la compañía informada sobre lo que está ocurriendo. Para esto utilizamos una “página de información de Sprint”.

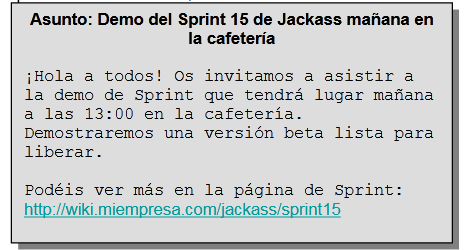


Tan pronto como sea posible tras la reunión de planificación de Sprint, el Scrum Master crea esta página y manda un spam a toda la compañía.



Adicionalmente, el Scrum Master imprime la página de información de Sprint y la pega en la pared fuera de la sala del equipo.

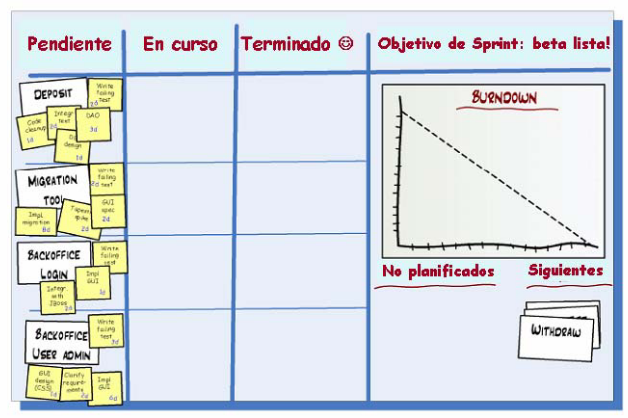
Cuando el Sprint se acerca a su final, el Scrum Master recuerda a todo el mundo que se acerca la Demo.



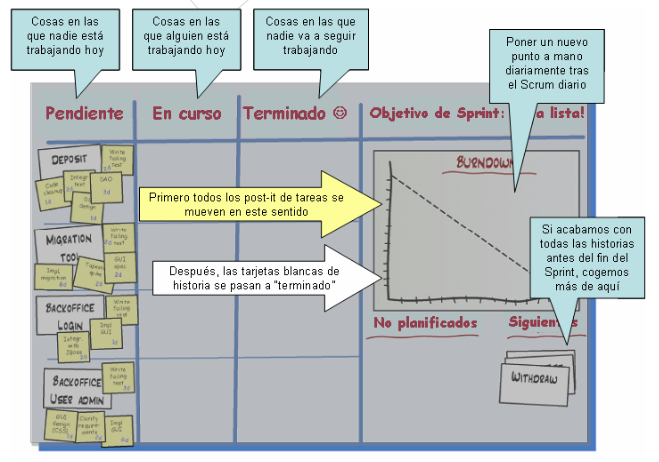
**6. Cómo hacemos las pilas de Sprint**

**Formato de la Pila de Sprint:**

Busca una pared que no esté usada o contenga cosas inútiles. Vacíala y pega una gran hoja de papel. Y entonces haz esto:



**Cómo funciona el tablón de tareas:**



**Ejemplo 1 – tras el primer Scrum diario:**

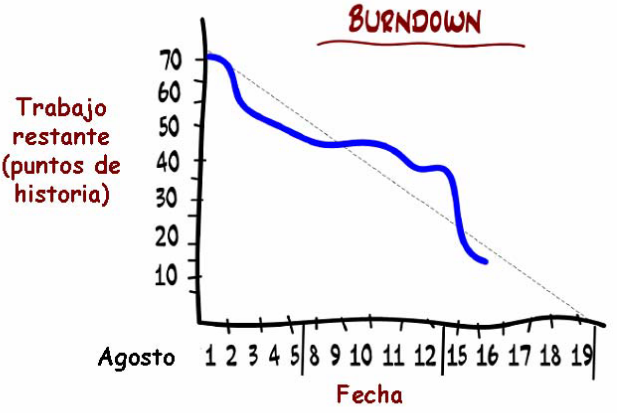
****

**Ejemplo 2 – tras unos cuantos días:**

****

Aquí tenemos un ejemplo de una Pila de Sprint real casi al final de un Sprint. Se vuelve bastante liosa conforme el Sprint progresa, pero no pasa nada, ya que tiene una vida muy corta. En cada nuevo Sprint, creamos una limpia y nueva Pila de Producto.

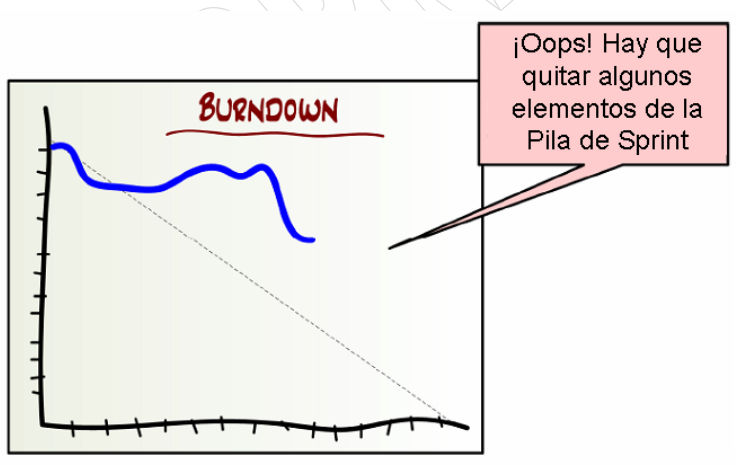
**Cómo funciona el diagrama burn-down:**

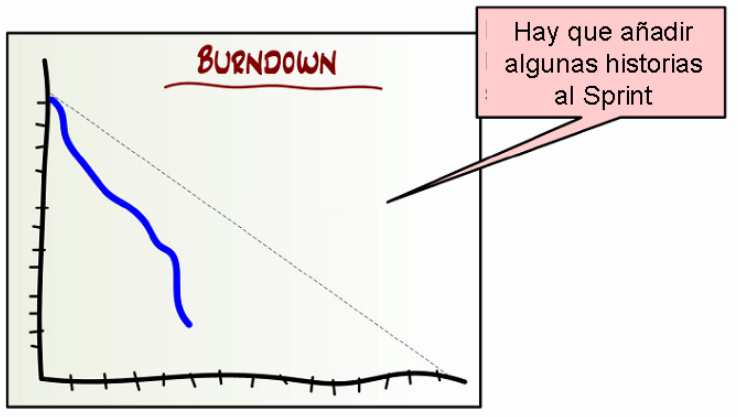
****

Este diagrama muestra que:

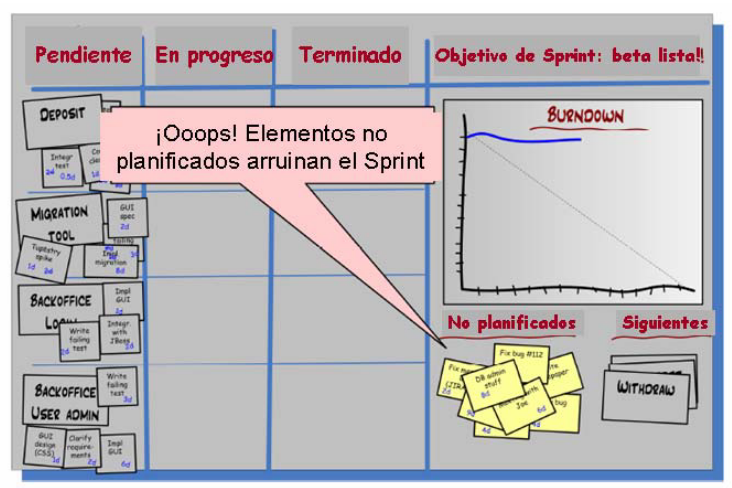
* En el primer día del Sprint, 1 de Agosto, el equipo estimó que habían aproximadamente 70 puntos de historia en los que trabajar. Ésta era la velocidad estimada para todo el Sprint.
* El 16 de Agosto el equipo estima que quedan aproximadamente 15 puntos de historia por hacer. La línea de puntos muestra incluso más avance respecto a la planificación.

**Señales de alarma en el burn-down:**

****

****

****

****

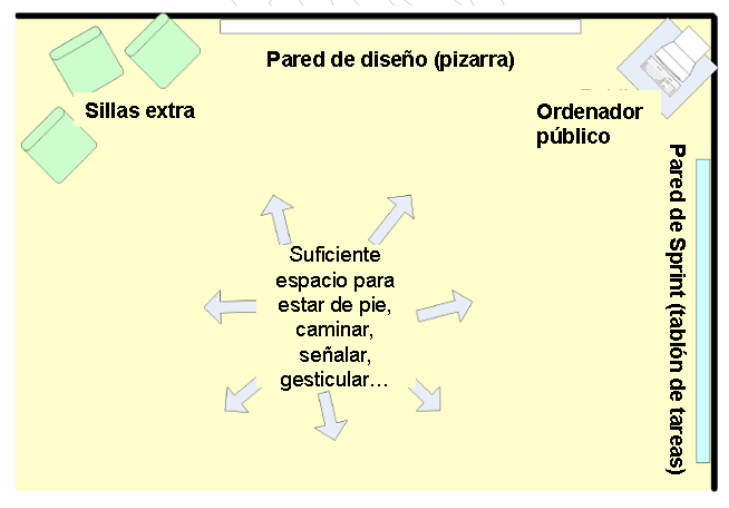
**Estimando en días vs horas:**

En la mayoría de los libros y artículos sobre Scrum encontrarás que las tareas se estiman en horas, no en días. Ahora hemos dejado de hacerlo por las siguientes razones:

* *Las estimaciones en horas-hombre eran demasiado granulares. Esto provocaba una tendencia a estimar muchas tareas en 1-2 horas, y con ello a la microgestión.*
* *Dos unidades diferentes causan confusión.*
* *Ahora usamos días-hombre como base para todas nuestras estimaciones.*

**7. Cómo distribuimos la sala del equipo:**

**La esquina de diseño:**

****

No hay mejor manera para obtener una visión general del sistema que estar de pie en la esquina de diseño y observar ambas paredes, entonces echar un vistazo en el ordenador y probar la última compilación del sistema.

La “pared de diseño” es una pizarra grande que contiene los garabateos más importantes sobre el diseño y la documentación impresa más importante.

**¡Sienta al equipo junto!**

A la gente le cuesta cambiar de sitio, no quieren tener que coger sus cosas, mover sus trastos a una nueva mesa. Pero cuando se trata de construir equipos Scrum efectivos, no obstante, no hay alternativa. Una vez ell equipo esté junto, los resultados serán inmediatos.

“Juntos” significa:

- *Audibilidad:* cualquier miembro del equipo puede hablar con cualquier otro sin tener que levantarse de su sitio.

- *Visibilidad*: todo el mundo puede ver a los demás. Todo el mundo puede ver el tablón de tareas. No necesariamente tan de cerca como para poder leerlo, pero si por lo menos verlo.

- *Aislamiento*: si todo el equipo necesita levantarse y agruparse para una discusión sobre el diseño, nadie fuera del equipo está tan cerca como para ser molestado. Y viceversa.

**Mantén al Dueño de Producto a mano.**

El Dueño de Producto debería estar suficientemente cerca como para que el equipo pueda caminar hasta donde está y preguntarle algo, y para que él pueda acercarse al tablón de tareas. Pero no debería sentarse junto al equipo, pues es probable que interrumpiese con sus observaciones la labor.

**Mantén a los gerentes y coachs a mano**

Si eres un coach en Scrum (y quizás también un gerente), involúcrate lo máximo posible. Pero sólo por un periodo de tiempo limitado. Entonces quítate de en medio y deja que sea el equipo el que se autogestione. Chequea el equipo de vez en cuando atendiendo a las demos de Sprint, mirando el tablón de tareas y escuchando en los Scrum diarios.

Si ves un área de mejora, reúnete con el Scrum Master y ayúdale, pero no enfrente del equipo.

**7. Cómo hacemos Scrum diarios**

**Cómo actualizamos el tablón:**

Actualizamos el tablón de tareas durante los Scrum diarios:

* *Conforme cada persona describe lo que hizo el día anterior y lo que hará hoy, mueve los post-it en el tablón.*
* *Conforme describe elementos no planificados, pone un pos-it nuevo para cada uno de ellos.*
* *Conforme actualiza sus estimaciones, escribe una nueva estimación en el post-it correspondiente y tacha la anterior estimación.*
* *A veces el Scrum Master hace todo esto mientras los demás hablan.*
* *Algunos equipos tienen la política de que cada persona debe hacer la actualización del tablón que le corresponda antes de cada reunión.*
* *Inmediatamente tras el Scrum diario, alguien suma todas las estimaciones de tiempo (ignorando los de la columna “terminado”, por supuesto) y dibuja un nuevo punto en el burn-down de Sprint.*

**9. Cómo hacemos la demo de Sprint.**

**Por qué insistimos en que todos los Sprints acaben con una demo.**

Una demo bien ejecutada tiene un efecto profundo:

* *El equipo obtiene reconocimiento por sus logros.*
* *Otras personas se enteran de lo que está haciendo el equipo.*
* *La demo consigue feedback vital de los interesados.*
* *Las demos son un evento social donde diferentes equipos pueden interactuar unos con otros y debatir sus trabajos.*
* *Hacer una demo fuerza al equipo a acabar realmente las cosas y entregarlas.*

**Lista de comprobación para demos de Sprint**

* *Asegúrate de presentar claramente el objetivo del Sprint.*
* *No pierdas mucho tiempo preparando la demo, especialmente en llamativas presentaciones.*
* *Concentra tu preparación en que la demo sea rápida en lugar de bonita.*
* *Mantén la demo a nivel de negocio.*
* *No muestres un montón de pequeños errores solucionados y funcionalidades triviales. Menciónalos, pero no los muestres.*

**10. Cómo hacemos retrospectivas de Sprint:**

**Por qué insistimos en qué todos los equipos hagan retrospectivas.**

Los equipos no siempre parecen inclinados a hacer retrospectivas. Sin embargo todo el mundo coincide en que las retrospectivas son extremadamente útiles. De hecho, la retrospectiva es el segundo evento más importante de Scrum ya que es la mejor oportunidad para mejorar.

**Cómo organizamos las retrospectivas.**

* *Reservamos 1-3 horas, dependiendo de cuánta discusión esperemos.*
* *Participantes: el Dueño de Producto, el equipo y yo mismo.*
* *Nos vamos a una reunión cerrada, un rincón cómodo con sofás, el patio del tejado o algún sitio similar.*
* *Normalmente no hacemos retrospectivas en la sala del equipo, ya que la atención de la gente suele diluirse.*
* *Alguien es designado secretario.*
* *El Scrum Master muestra la Pila de Sprint y, con ayuda del equipo, resume el Sprint.*
* *Hacemos “la ronda”. Cada persona tiene una oportunidad de decir, sin ser interrumpida, qué piensa.*
* *Observamos la velocidad estimada frente a la real.*
* *Cuando el tiempo casi se ha acabado, el Scrum Master trata de resumir las sugerencias concretas.*