Índice Temático

Modelo Relacional de Base de datos en la actualidad y su relación con las bases de datos NoSQL **2**
Joins en MySQL - explicados de forma simple por https://www.vichaunter.org/ 2
Diagrama de la relación: 3
Código de creación de base de datos y usuario: 3
Código de creación de tablas e inserción de datos en MySQL: (agregar unique!) 4
Tipos de Joins: 5
INNER JOIN o simplemente JOIN: 5
LEFT JOIN: 6
RIGHT JOIN: 8
OUTER JOIN ó FULL OUTER JOIN: 9
Funciones **10**
MySQL/MariaDB: 10
PostgreSQL: 11
Procedimientos Almacenados **11**
SQLite: 11
MySQL/MariaDB: 12
PostgreSQL: 12
Triggers **13**
SQLite: 13
MySQL/MariaDB: 13
PostgreSQL: 13
Eventos **14**
SQLite: 14
MySQL/MariaDB: 14
PostgreSQL: 14
Referencias **15**

Modelo Relacional de Base de datos en la actualidad y su relación con las bases de datos NoSQL

Las bases de datos SQL o que aplican un modelo relacional no han sido reemplazadas como se pretendió en algún momento por las bases de datos no-sql. En informática, NoSQL (a veces llamado "no solo SQL") es una amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico de SGBDR (Sistema de Gestión de Bases de Datos Relacionales) en aspectos importantes, siendo el más destacado que no usan SQL como lenguaje principal de consultas. Los datos almacenados no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN, ni garantizan completamente ACID (atomicidad, consistencia, aislamiento y durabilidad) y habitualmente escalan bien horizontalmente. Los sistemas NoSQL se denominan a veces "no solo SQL" para subrayar el hecho de que también pueden soportar lenguajes de consulta de tipo SQL.NoSQL. (2024, 8 de febrero)

Cómo concluyen Bas Abad, V. J. (2015) “Los sistemas NoSQL se han mostrado mucho más eficientes a la hora de gestionar grandes cantidades de datos. Mientras más grande era el número de variables y el periodo de selección de datos, mejor rendimiento ofrecían respecto a sus homólogos SQL…Por los motivos citados y por la alta tasa de transmisión de datos, en este proyecto se ha considerado a los sistemas NoSQL y en concreto a Couchbase como mejor posicionados para este tipo de aplicaciones donde se manejen un alto número de datos con pocas relaciones.” En caso contrario donde el problema sea netamente relacional se debe considerar el uso de un RDBMS.

Esto demuestra que debemos ser agnósticos respecto a las tecnologías no hay una mejor, sólo se comporta mejor según el contexto en el que debemos aplicarla.

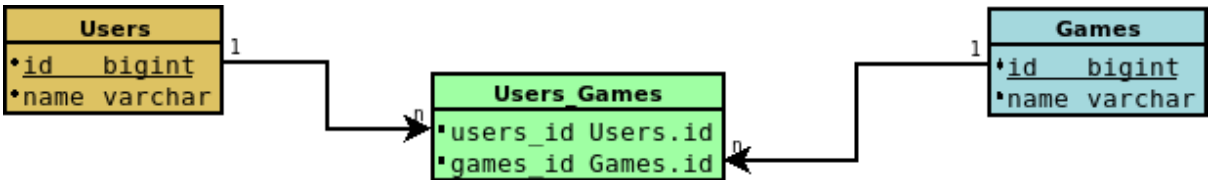
Joins en MySQL - explicados de forma simple por https://www.vichaunter.org/

Los JOIN son usados en una sentencia SQL para recuperar datos de varias tablas al mismo tiempo. Estas tablas tienen que estar relacionadas de alguna forma, por ejemplo, en una relación con cardinalidad “muchos a muchos” se usa una tabla intermedia que a menudo se denomina: “tabla de unión”. El ejemplo tiene 3 tablas:

- Users: Tabla que representa a los usuarios. Un mismo usuario puede tener varios juegos, o varios usuarios tener un juego en concreto.
- Games: Tabla que representa a los juegos. Un mismo juego puede pertenecer a muchos usuarios.

- Users_Games: Tabla unión en una relación de muchos a muchos entre juegos y usuarios.

Diagrama de la relación:



Código de creación de base de datos y usuario:

Aclaración IMPORTANTE: Si bien se insertan los ids con valor numérico, no es estrictamente necesario hacerlo ya que son campos autoincrementales, solo se hace con fines didácticos.

(ejecutar como root de la base de datos):

```
---------------------------------
-- Create Database games
---------------------------------
CREATE DATABASE `games` COLLATE 'utf8mb4_unicode_ci';

-- ---------------------------------------------------------
-- Create Database user "games_db_user" with password 1234 in hash
-- and grant all privileges on database "games"
-- ---------------------------------------------------------
CREATE USER 'games_db_user'@'localhost' IDENTIFIED BY PASSWORD
'*A4B6157319038724E3560894F7F932C8886EBFCF';
GRANT ALL PRIVILEGES ON `games`.* TO 'games_db_user'@'localhost';
```

Código de creación de tablas e inserción de datos en MySQL: (agregar unique!)

```sql
-- ----------------------------
-- FOREIGN_KEY SAFE EDITION
-- ----------------------------
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for Users
-- ----------------------------
DROP TABLE IF EXISTS `Users`;
CREATE TABLE `Users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` varchar(255) COLLATE 'utf8mb4_unicode_ci' NOT NULL
) ENGINE='InnoDB' COLLATE 'utf8mb4_unicode_ci';

-- ----------------------------
-- Records of Users
-- ----------------------------
INSERT INTO `Users` VALUES ('1', 'vichaunter');
INSERT INTO `Users` VALUES ('2', 'pepito');
INSERT INTO `Users` VALUES ('3', 'jaimito');
INSERT INTO `Users` VALUES ('4', 'ataulfo');

-- ----------------------------
-- Table structure for Games
-- ----------------------------
DROP TABLE IF EXISTS `Games`;
CREATE TABLE `Games` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` varchar(255) COLLATE 'utf8mb4_unicode_ci' NOT NULL
) ENGINE='InnoDB' COLLATE 'utf8mb4_unicode_ci';

-- ----------------------------
-- Records of Games
-- ----------------------------
INSERT INTO `Games` VALUES ('1', 'Final Fantasy VII');
INSERT INTO `Games` VALUES ('2', 'Zelda: A link to the past');
INSERT INTO `Games` VALUES ('3', 'Crazy Taxy');
INSERT INTO `Games` VALUES ('4', 'Donkey Kong Country');
INSERT INTO `Games` VALUES ('5', 'Fallout 4');
INSERT INTO `Games` VALUES ('6', 'Saints Row');
INSERT INTO `Games` VALUES ('7', 'La taba');

-- ----------------------------
-- Table structure for Users_Games
-- ----------------------------
DROP TABLE IF EXISTS `Users_Games`;
CREATE TABLE `Users_Games` (
  `users_id` int(11) unsigned NOT NULL,
  `games_id` int(11) unsigned NOT NULL,
  FOREIGN KEY (`users_id`) REFERENCES `Users` (`id`),
  FOREIGN KEY (`games_id`) REFERENCES `Games` (`id`)
) ENGINE='InnoDB' COLLATE 'utf8mb4_unicode_ci';
```

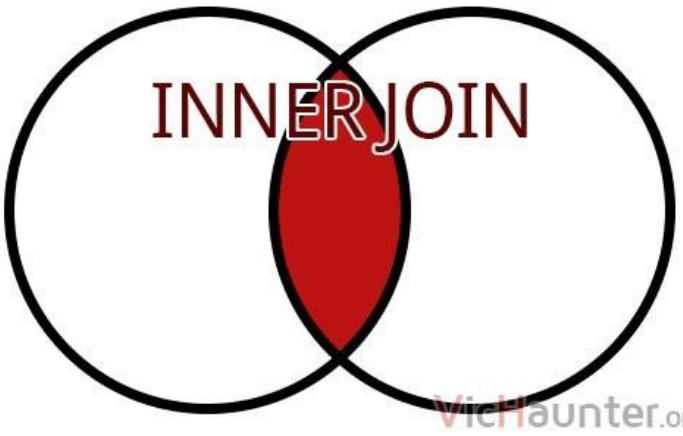
```
-- ----------------------------
-- Records of Users_Games
-- ----------------------------
INSERT INTO Users_Games VALUES ('1', '1');
INSERT INTO Users_Games VALUES ('1', '2');
INSERT INTO Users_Games VALUES ('1', '3');
INSERT INTO Users_Games VALUES ('1', '4');
INSERT INTO Users_Games VALUES ('1', '6');
INSERT INTO Users_Games VALUES ('1', '7');
INSERT INTO Users_Games VALUES ('2', '3');
INSERT INTO Users_Games VALUES ('2', '7');
INSERT INTO Users_Games VALUES ('4', '1');
INSERT INTO Users_Games VALUES ('4', '2');
INSERT INTO Users_Games VALUES ('4', '4');
INSERT INTO Users_Games VALUES ('4', '7');

-- ----------------------------
-- ACTIVATE FOREIGN_KEY CHECKS
-- ----------------------------
SET FOREIGN_KEY_CHECKS=1;
```

Ahora que tenemos toda la base de datos creada con registros cargados vamos a ver que tipos de sentencias JOIN existen:

Tipos de Joins:

INNER JOIN o simplemente JOIN:

Este tipo de unión te ayuda a combinar varias tablas, y te devuelve únicamente los datos que estén disponibles en todas las tablas a la vez.



Esto significa, que si por ejemplo haces un INNER JOIN para ver los juegos que tiene cada usuario, solo devolverá datos siempre que un juego pertenezca a un usuario. Si un juego no tiene ningún propietario, pero existe en la tabla, no aparecerá, y si un usuario no tiene ningún juego asociado tampoco verás a ese usuario. La consulta sería algo así:

```
SELECT Users.name, Games.name FROM Users
INNER JOIN Users_Games ON Users.id = Users_Games.users_id
INNER JOIN Games ON Users_Games.games_id = Games.id
```

Al utilizar una tabla relacional tienes que hacer dos joins. El primer JOIN nos une la tabla usuarios, con la tabla juegousuario. Ahora que ya la tenemos unida, podemos utilizar los datos de la tabla juegousuario, y lanzar la consulta con la tabla juegos en otro JOIN.

Como este join devuelve solo los datos que hay en las dos tablas, esperamos un resultado como el siguiente:

name	name
vichaunter	Final Fantasy VII
vichaunter	Zelda: A link to the past
vichaunter	Crazy Taxy
vichaunter	Donkey Kong Country
vichaunter	Saints Row
vichaunter	La taba
pepito	Crazy Taxy
pepito	La taba
ataulfo	Final Fantasy VII
ataulfo	Zelda: A link to the past
ataulfo	Donkey Kong Country
ataulfo	La taba

Como puedes ver, no hay ni rastro del usuario jaimito, ni del juego Fallout 4, pues ni el uno tiene juegos, ni el juego es de nadie. Por supuesto esta consulta la podemos hacer simplemente con dos tablas, evitando una tabla de unión. En una tienda de alquiler, en la que solo hay un juego de cada, bastaría con poner un campo id_usuario a cada juego como en la primera tabla, y hacer solo un join.

LEFT JOIN:

En este caso, el left join devuelve todos los resultados que coincidan en la primera tabla, con los datos que tenga de la segunda. En el caso de que falte algún dato, devolverá un valor null en lugar del dato, pero seguiremos teniendo el valor de la primera tabla.

Por ejemplo, si quieres saber todos los juegos de los usuarios, con un left join tendremos una lista completa de todos los usuarios, incluso si no tienen ningún juego.

```
SELECT Users.name, Games.name FROM Users
LEFT JOIN Users_Games ON Users.id = Users_Games.users_id
LEFT JOIN Games ON Users_Games.games_id = Games.id
```

El resultado sería el siguiente:

name	name
vichaunter	Final Fantasy VII
vichaunter	Zelda: A link to the past
vichaunter	Crazy Taxy
vichaunter	Donkey Kong Country
vichaunter	Saints Row
vichaunter	La taba
pepito	Crazy Taxy
pepito	La taba
jaimito	*NULL*
ataulfo	Final Fantasy VII
ataulfo	Zelda: A link to the past
ataulfo	Donkey Kong Country
ataulfo	La taba

Si te fijas, ahora sí que tenemos a jaimito, pero sin embargo nos muestra que no hay juego. Esto puede ser muy útil, por ejemplo para localizar todos los usuarios que no tienen juegos (con un WHERE juegoname IS NULL, por ejemplo).

RIGHT JOIN:

En el caso del RIGHT JOIN, pasa exactamente lo mismo que con el anterior, pero con la diferencia de que devuelve todos los datos de la tabla con la que se relaciona la anterior. Si estamos ejecutando un SELECT en la tabla usuarios, las demás serán tablas con las que se relaciona.

Por ejemplo, supón que quieres saber todos los juegos que tienes, y a qué usuarios pertenecen (o simplemente si le pertenece a algún usuario). Lo harías de esta forma:

```
SELECT Users.name, Games.name FROM Users
RIGHT JOIN Users_Games ON Users.id = Users_Games.users_id
RIGHT JOIN Games ON Users_Games.games_id = Games.id
```

Con estos resultados:

name	name
vichaunter	Final Fantasy VII
ataulfo	Final Fantasy VII
vichaunter	Zelda: A link to the past
ataulfo	Zelda: A link to the past
vichaunter	Crazy Taxy
pepito	Crazy Taxy
vichaunter	Donkey Kong Country
ataulfo	Donkey Kong Country
NULL	Fallout 4
vichaunter	Saints Row
vichaunter	La taba
pepito	La taba
ataulfo	La taba

Ahora se ha invertido. Como se puede observar, ya no hay ni rastro de jaimito, pero sin embargo aparece Fallout 4 sin estar asociado a ningún usuario. Esto realmente tiene muchas aplicaciones interesantes, solo hay que echarle imaginación.

OUTER JOIN ó FULL OUTER JOIN:

El OUTER JOIN consiste en recuperar TODOS los datos que haya en ambas tablas, tanto los que tienen contenido en ambos extremos, como los que no. Es la oveja negra en MySQL, ya que no es directamente compatible, pero sí se puede conseguir un efecto similar.

Para poder hacer este tipo de consultas tendrás que echar mano de UNION, que sirve precisamente para combinar los resultados de varios SELECT en una sola consulta (¿nunca lo has usado? es como hacer trampa).

```
SELECT Users.name, Games.name FROM Users
LEFT JOIN Users_Games ON Users.id = Users_Games.users_id
LEFT JOIN Games ON Users_Games.games_id = Games.id

UNION

SELECT Users.name, Games.name FROM Users
RIGHT JOIN Users_Games ON Users.id = Users_Games.users_id
RIGHT JOIN Games ON Users_Games.games_id = Games.id
```

Como seguro que ya te habrás fijado, la norma para poder utilizar UNION, es que en todos los SELECT debe haber los mismos campos seleccionados. La salida será:

name	name
vichaunter	Final Fantasy VII
vichaunter	Zelda: A link to the past
vichaunter	Crazy Taxy
vichaunter	Donkey Kong Country
vichaunter	Saints Row
vichaunter	La taba
pepito	Crazy Taxy
pepito	La taba
jaimito	*NULL*
ataulfo	Final Fantasy VII
ataulfo	Zelda: A link to the past
ataulfo	Donkey Kong Country
ataulfo	La taba
NULL	Fallout 4

Como ves salen todos los datos, incluso los que no tienen nada en uno de los lados. De esta forma puedes crear varios filtros distintos después de extraerlos por ejemplo. Así con una única consulta a la base de datos tendrías todo lo que necesitas.

Funciones

SQLite:

SQLite no soporta la creación de funciones definidas por el usuario de forma nativa como MySQL o PostgreSQL. Sin embargo, puedes crear funciones personalizadas mediante lenguajes como Python, C o SQLite3 en un entorno específico, pero aquí nos enfocaremos en ejemplos simples para MySQL y PostgreSQL.

MySQL/MariaDB:

En MySQL, puedes crear una función usando `CREATE FUNCTION`. Un ejemplo que devuelve el nombre completo de un usuario sería:

```
DELIMITER //

CREATE FUNCTION get_full_name(first_name VARCHAR(50), last_name
VARCHAR(50))
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
    RETURN CONCAT(first_name, ' ', last_name);
END //

DELIMITER ;
```

Uso de la función:

```
SELECT get_full_name('John', 'Doe');
```

PostgreSQL:

En PostgreSQL, puedes crear una función usando `CREATE FUNCTION` también. Aquí un ejemplo similar al anterior:

```
CREATE OR REPLACE FUNCTION get_full_name(first_name VARCHAR,
last_name VARCHAR)
RETURNS VARCHAR AS $$
BEGIN
    RETURN first_name || ' ' || last_name;
END;
$$ LANGUAGE plpgsql;
```

Uso de la función:

```
SELECT get_full_name('John', 'Doe');
```

Procedimientos Almacenados

SQLite:

SQLite no tiene soporte nativo para procedimientos almacenados como lo hacen MySQL y PostgreSQL.

MySQL/MariaDB:

Un procedimiento almacenado puede ser creado con CREATE PROCEDURE. Aquí un ejemplo que inserta un nuevo usuario en la tabla users:

```
DELIMITER //

CREATE PROCEDURE insert_user(IN first_name VARCHAR(50), IN last_name
VARCHAR(50))
BEGIN
    INSERT INTO users (first_name, last_name) VALUES (first_name,
last_name);
END //

DELIMITER ;
```

Para ejecutar el procedimiento:

```
CALL insert_user('John', 'Doe');
```

PostgreSQL:

En PostgreSQL, puedes crear un procedimiento de manera similar con CREATE PROCEDURE. Un ejemplo que inserta un usuario en la tabla users:

```
CREATE OR REPLACE PROCEDURE insert_user(first_name VARCHAR,
last_name VARCHAR)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO users (first_name, last_name) VALUES (first_name,
last_name);
END;
$$;
```

Para ejecutarlo:

```
CALL insert_user('John', 'Doe');
```

Triggers

SQLite:

SQLite soporta triggers. Aquí un ejemplo que registra la fecha y hora cada vez que se inserta un nuevo usuario en la tabla users:

```
CREATE TRIGGER insert_user_log
AFTER INSERT ON users
BEGIN
    INSERT INTO user_log (user_id, log_time) VALUES (new.id,
DATETIME('now'));
END;
```

MySQL/MariaDB:

En MySQL, puedes crear un trigger de manera similar. Aquí un ejemplo para registrar el tiempo de inserción de un nuevo usuario:

```
DELIMITER //

CREATE TRIGGER insert_user_log
AFTER INSERT ON users
FOR EACH ROW
BEGIN
    INSERT INTO user_log (user_id, log_time) VALUES (NEW.id, NOW());
END //

DELIMITER ;
```

PostgreSQL:

En PostgreSQL, los triggers se crean con CREATE TRIGGER. Un ejemplo que registra el tiempo de inserción de un nuevo usuario:

```
CREATE OR REPLACE FUNCTION log_user_insertion()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO user_log (user_id, log_time) VALUES (NEW.id, NOW());
    RETURN NEW;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER insert_user_log
AFTER INSERT ON users
FOR EACH ROW
EXECUTE FUNCTION log_user_insertion();
```

Eventos

SQLite:

SQLite no soporta eventos de forma nativa como MySQL.

MySQL/MariaDB:

MySQL permite la creación de eventos, que son tareas programadas que se ejecutan en momentos específicos. Aquí un ejemplo que borra usuarios inactivos cada día:

```
CREATE EVENT delete_inactive_users
ON SCHEDULE EVERY 1 DAY
DO
BEGIN
    DELETE FROM users WHERE last_login < NOW() - INTERVAL 1 YEAR;
END;
```

PostgreSQL:

PostgreSQL no tiene un sistema de eventos incorporado como MySQL. Sin embargo, puedes usar el cron o pg_cron (una extensión para PostgreSQL) para ejecutar tareas programadas.

```
SELECT cron.schedule('delete_inactive_users', '0 0 * * *', $$
    DELETE FROM users WHERE last_login < NOW() - INTERVAL '1
year';
$$);
```

Referencias

NoSQL. (2024, 8 de febrero). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 18:40, febrero 8, 2024 desde https://es.wikipedia.org/w/index.php?title=NoSQL&oldid=158068698.

Bas Abad, V. J. (2015). Estudio comparativo de BBDD relacionales y NoSQL en un entorno industrial (Doctoral dissertation, Universitat Politècnica de València). Recuperado de https://riunet.upv.es/bitstream/handle/10251/55530/Memoria.pdf?sequence=1&isAllowed=y

VicHaunter (2017) Joins en MySQL bien explicado: Todo lo que necesitas saber Consultado de: https://www.vichaunter.org/desarrollo-web/joins-mysql-bien-explicado-lo-necesitas-saber

Elmasri R. Navathe S. (2016). Sistemas de Bases de Datos.4°ed. España: Addison Wesley Iberoamericana.

Marqués M. (2011). Bases de Datos. España: Universidad Jaumet I.

Silberschatz A. Korth H. Sudarshan S. (2002) Fundamentosde Bases de Datos. 4° ed. Buenos Aires: Mc Graw Hill.