

Memoria Proyecto EDGE

Alumnos:	2
Datos	3
Modelo Relacional	3
Tablas.....	3
Creación de datos.....	5
Consultas.....	6
Modelo Relacional extendido	11
Esquema de Tipos Compuestos (Agregación Nativa de PostgreSQL).....	12
Implementación Técnica:.....	12
Consultas.....	14
Análisis de impacto:.....	16
Esquema JSONB (Agregación Documental).....	16
Implementación Técnica:.....	16
Consultas.....	19
Análisis de impacto:.....	21
Comparativa de rendimiento.....	21
Base de Datos Relacional distribuida	21
● Importación de los datos en formato relacional.....	23
● Cambios realizados:.....	23
● Distribución.....	24
-Tablas distribuidas:.....	24
● Rendimiento de consultas en función de workers operativos.....	25
● Uso formato columnar.....	26
● Conclusiones.....	28
Base de datos NoSQL documental: MongoDB	29
-CREACIÓN DEL CLUSTER:.....	29
-IMPORTACIÓN DE LOS DATOS:.....	33
-AGREGACIÓN DE LOS DATOS:.....	33
-INDEXACIÓN Y SHARDING:.....	36
-EXPLICACIÓN DE LAS CONSULTAS:.....	38
-CONSULTAS BENEFICIADAS Y PERJUDICADAS:.....	41
-PRUEBAS DETENIENDO MÁQUINAS:.....	41
-GUIA DE EJECUCIÓN:.....	43
-CONCLUSIONES:.....	43
Base de datos NoSQL de grafos: Neo4J	44
-CONFIGURACIÓN INICIAL:.....	44
-IMPORTACIÓN DE LOS DATOS:.....	44
-REALIZACIÓN DE LAS CONSULTAS:.....	47
-GUIA DE EJECUCIÓN:.....	49
-CONCLUSIONES:.....	50

*El cómo ejecutar los scripts está detallado en el archivo **README .md**

Alumnos:

Mateo Fraguas Abal mateo.fraguas@rai.usc.es
Álvaro Garnelo Luaces alvaro.garnelo@rai.usc.es
Rodrigo López Rego rodrigo.lopez.rego@rai.usc.es

Datos

Viendo los datasets disponibles a través de kaggle decidimos seleccionar dos de ellos, uno de [luchas](#) y otro de [luchadores](#). Para utilizar estos en nuestra base de datos primero definimos el modelo relacional a utilizar.

Modelo Relacional

Tablas

Definimos 5 tablas, luchadores; eventos; peleas; estilos y estilos_luvhadores. Luchadores guarda la información de cada uno de los luchadores, su nombre; mote; fecha de nacimiento; edad; país de origen; altura; peso; asociación (gimnasio donde entran o equipo de entrenamiento); categoría de peso y sus resultados en pérdidas y victorias.

```
CREATE TABLE luchadores (
    url TEXT PRIMARY KEY,
    fighter_name VARCHAR(100) NOT NULL,
    nickname VARCHAR(100),
    birth_date DATE,
    age INT,
    country VARCHAR(100),
    height_cm NUMERIC(5,2),
    weight_kg NUMERIC(5,2),
    association VARCHAR(100),
    weight_class VARCHAR(50),
    wins INT,
    losses INT);
```

La tabla de eventos guarda el título de cada evento, de que organización fue, en qué fecha y la localización en nombre y en coordenadas de latitud y longitud; cada una de las peleas son definidas por el evento en el que ocurrieron, el orden en el que ocurrió (1: primera pelea, n: pelea titular), los luchadores que participaron, el resultado (sólo empate, sin resultado o victorias en las que se define como ganador a fighter1 sobre fighter2), el método y detalles de victoria (sí se aplica), el nombre del árbitro, el número de rondas y el tiempo de duró.

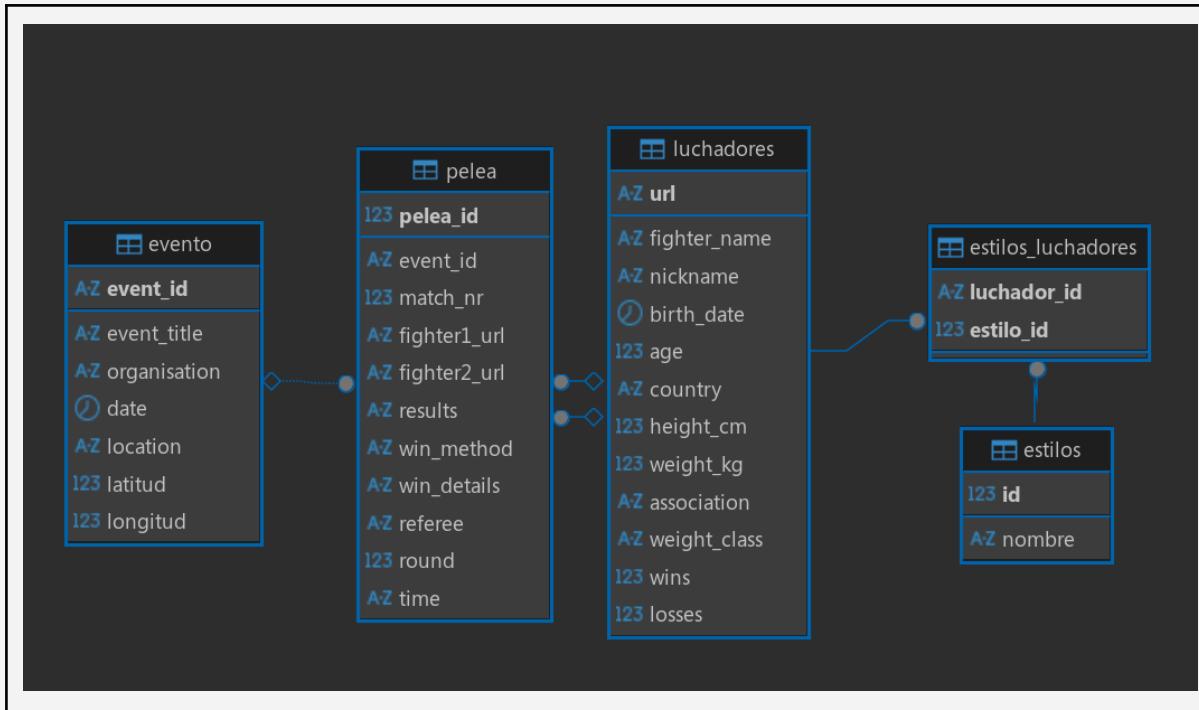
```
CREATE TABLE evento (
    event_id TEXT PRIMARY KEY,
    event_title VARCHAR(100) NOT NULL,
    organisation VARCHAR(100),
    date DATE,
    location VARCHAR(255),
    latitud NUMERIC(10, 7),
    longitud NUMERIC(11, 8)
);

CREATE TABLE pelea (
    pelea_id SERIAL PRIMARY KEY,
    event_id TEXT REFERENCES evento(event_id) ON DELETE CASCADE,
    match_nr INT,
    fighter1_url TEXT REFERENCES luchadores(url) ON DELETE CASCADE,
    fighter2_url TEXT REFERENCES luchadores(url) ON DELETE CASCADE,
    results VARCHAR(50),
    win_method VARCHAR(100),
    win_details VARCHAR(255),
    referee VARCHAR(100),
    round INT,
    time VARCHAR(10)
);
```

La tabla de estilos guarda los 8 posibles estilos de los luchadores (jiu-jitsu, wrestling, kickboxing, boxing, judo, karate, sambo y taekwondo) y la tabla estilos_luchadores que luchadores tienen asignados que estilos.

```
CREATE table estilos (
    id SERIAL primary key,
    nombre VARCHAR(50));

create table estilos_luchadores (
    luchador_id TEXT REFERENCES luchadores(url) ON DELETE CASCADE,
    estilo_id INT REFERENCES estilos(id) ON DELETE CASCADE,
    PRIMARY KEY (luchador_id, estilo_id));
```



Creación de datos

Para llenar las tablas de datos tuvimos que adaptar los dos archivos existentes. Con los datos de las luchas de (`pro_mma_fights.csv`) obtenemos dos archivos CSV normalizados (`events.csv` y `fights.csv`). Para el archivo de `eventos` tomamos las columnas (`url`, `event_title`, `organisation`, `date`, `location`) y eliminamos los duplicados, cada evento tiene una combinación única de todos los valores de esas columnas y al eliminar los duplicados pasamos de tener una fila por pelea a una por evento. Utilizamos la `url` como clave principal. Para el archivo de `peleas` seleccionamos las columnas (`url`, `match_nr`, `fighter1_url`, `fighter2_url`, `fighter1_result`, `win_method`, `win_details`, `referee`, `round`, `time`), tomamos sólo `fighter1_result` y lo renombramos como `results` para simplificar y renombramos `url` a `event_id`, la clave foránea.

Además decidimos incluir las coordenadas de los estadios donde ocurren los eventos, para buscar e incluir estos datos automáticamente creamos un script de python con la librería [geopy](#). Recorriendo la lista de localizaciones llamamos al agente geolocalizador [Nominatim](#) declarado (definimos `user_agent` para respetar la política de uso de Nominatim) y la pasamos el nombre de cada uno de los estadios, si en algún caso no encuentra la localización quitamos el nombre del estadio y buscamos otra vez.

```
Guardando los resultados en data/luchas/events_con_coordenadas.csv
¡Proceso completado con éxito!
Los datos con las coordenadas se han guardado en: data/luchas/events_con_coordenadas.csv
Resumen:
Se procesaron 1006 eventos.
Coordenadas encontradas para 973 eventos.
No se encontraron coordenadas para 33 eventos.
```

Ejecutando el script obtenemos la mayoría de las latitudes y longitudes del conjunto de datos, como la definición de la tabla no nos obliga a que todas las líneas tengan los valores de coordenadas no causa ningún error la ausencia de unas 33 localizaciones. Para los datos de luchadores necesitamos definir dos funciones que conviertan las medidas de altura y peso a centímetros y kilogramos. La medida de altura inicial está definida en pies y pulgadas así que primero sepáramos ojos dos valores, los sumamos como pulgadas y los convertimos a centímetros multiplicando el valor por 2.54. La medida de peso original incluye 'lbs' así que tenemos que limpiar el valor antes de convertirlo en kilogramos multiplicando por 0.4536 (viendo los datos hay luchadores con un peso de 0 lbs, esto es un error de la página de la que provienen los datos, no del dataset seleccionado. Decidimos no substituir estos valores). Guardamos las columnas que queremos conservar y renombramos **height** y **weight** a **height_cm** y **weight_kg** además de corregir **lossess** por **losses**. Posteriormente, al inyectar los datos en postgresql, nos dimos cuenta de que hay unos pocos luchadores sin nombre que nos estaban causando fallo ya que definimos la tabla para que todo luchador tenga el valor de nombre no nulo. Incluimos en el script una función que extraiga el nombre a partir de la url aunque al principio simplemente los añadimos a mano.

```
Número de luchadores sin nombre: 3
Extraído nombre del URL '/fighter/Hunter-Tucker-54564': 'Hunter Tucker'
Extraído nombre del URL '/fighter/Noad-Lahat-35732': 'Noad Lahat'
Extraído nombre del URL '/fighter/AJ-Matthews-21922': 'Aj Matthews'
```

Para los archivos de 'estilos' y 'estilos-luchadores' utilizamos un script que guarda los valores de estilos.txt en un csv y que genera asociaciones al azar (de 0 a 3 estilos por luchador).

Consultas

1. Consulta de victorias por estilo: Esta sentencia permite identificar la efectividad de cada disciplina marcial mediante el requerimiento 1 (JOIN de más de dos tablas), conectando las tablas de estilos, luchadores y peleas. Al utilizar la función COUNT(*) y agrupar los resultados por el nombre del estilo, cumple satisfactoriamente con el requerimiento 2 (funciones de agregado) y el requerimiento 3 (cláusula GROUP BY), ofreciendo un ranking de victorias totales de forma estructurada.

	AZ estilo	123 total_victorias
1	judo	2.056
2	sambo	2.048
3	jiu-jitsu	2.040
4	karate	1.986
5	taekwondo	1.952
6	kickboxing	1.916
7	boxing	1.890
8	wrestling	1.831

2. Consulta de total de peleas por luchador: Diseñada para medir la veterana en el octágono, esta consulta satisface el requerimiento 1 (uso de JOIN) al vincular la tabla de luchadores con la de peleas mediante una unión externa. Cumple además con el requerimiento 2 (funciones de agregado) y el requerimiento 3 (cláusula GROUP BY) al contabilizar el número total de apariciones de cada atleta y agruparlas por su nombre, permitiendo visualizar la actividad competitiva de cada perfil en el dataset.

	AZ fighter_name	123 total_peleas
1	Donald Cerrone	37
2	Jim Miller	37
3	Andrei Arlovski	36
4	Jeremy Stephens	34
5	Cheick Kongo	33
6	Demian Maia	33
7	Diego Sanchez	32
8	Tito Ortiz	31
9	Frank Mir	30
10	Clay Guida	30
11	Rafael dos Anjos	30
12	Lyoto Machida	29
13	Michael Bisping	29
14	Ben Saunders	29
15	Gleison Tibau	28
16	Frankie Edgar	28
17	Charles Oliveira	28
18	Matt Brown	28
19	Ryan Bader	28
20	Thiago Alves	27

3. Consulta de más de 5 victorias por sumisión: Esta búsqueda avanzada filtra especialistas técnicos utilizando el operador LIKE para localizar el término "Submission", cumpliendo con el requerimiento 7 (búsqueda por palabras clave

en texto). Al agrupar por luchador y aplicar un filtro posterior al cálculo, satisface los requerimientos 2 (agregado) y 3 (GROUP BY), destacando especialmente por cumplir el requerimiento 4 (uso de la cláusula HAVING) para discriminar grupos que superan el umbral de cinco victorias.

	AZ fighter_name	123 victorias_por_sumision
1	Charles Oliveira	14
2	Demian Maia	11
3	Royce Gracie	10
4	Jim Miller	10
5	Shinya Aoki	9
6	Frank Mir	8
7	Goiti Yamauchi	8
8	Nate Diaz	8
9	Neiman Gracie	7
10	A.J. McKee	7
11	Angela Lee	7
12	Kenny Florian	7
13	Cole Miller	7
14	Joe Lauzon	7
15	Rani Yahya	7
16	Alex Silva	7
17	Marat Gafurov	7
18	Gunnar Nelson	7
19	Tony Ferguson	6
20	Alexandre Bezerra	6

4. Consulta de origen brasileño o estilo Jiu-Jitsu: Esta sentencia combina dos criterios de búsqueda independientes mediante el uso de la cláusula UNION, cumpliendo estrictamente con el requerimiento 5 (uso de UNION). Al integrar información geográfica de los luchadores y técnica de los estilos mediante tablas intermedias, valida también el requerimiento 1 (JOIN), garantizando un conjunto de resultados único y sin duplicados que representa la esencia técnica y regional de esta disciplina.

O	AZ fighter_name	AZ country	AZ nombre
1	Scott Writz	United States	jiu-jitsu
2	Adriano Moraes	Brazil	karate
3	Larissa Pacheco	Brazil	taekwondo
4	Jimmy Lugo	United States	jiu-jitsu
5	Mayra Bueno Silva	Brazil	taekwondo
6	Valdir Araujo	Brazil	boxing
7	Almíro Barros	Brazil	wrestling
8	Dane Bonnigson	United States	jiu-jitsu
9	Adriano Martins	Brazil	judo
10	Leandro Silva	Brazil	judo
11	Roger Gracie	Brazil	karate
12	Larissa Pacheco	Brazil	kickboxing
13	Yuri Simoes	Brazil	jiu-jitsu
14	William Macario	Brazil	taekwondo
15	Shahbulat Shamhalaev	Russia	jiu-jitsu
16	Rory MacDonald	Canada	jiu-jitsu
17	Andre Muniz	Brazil	judo
18	Daniel Compton	United States	jiu-jitsu
19	Joe Stevenson	United States	jiu-jitsu
20	Frank Shamrock	United States	jiu-jitsu

5. Consulta de búsqueda por patrón de texto 'mar': Centrada en la recuperación eficiente de información, esta consulta utiliza el operador ILIKE sobre el campo URL y nombre del luchador, cumpliendo con el requerimiento 7 (procura basada en palabras clave sobre campos de texto). Es una consulta esencial para la usabilidad del sistema que demuestra cómo filtrar registros basándose en fragmentos de texto parciales sin importar su posición o el uso de mayúsculas.

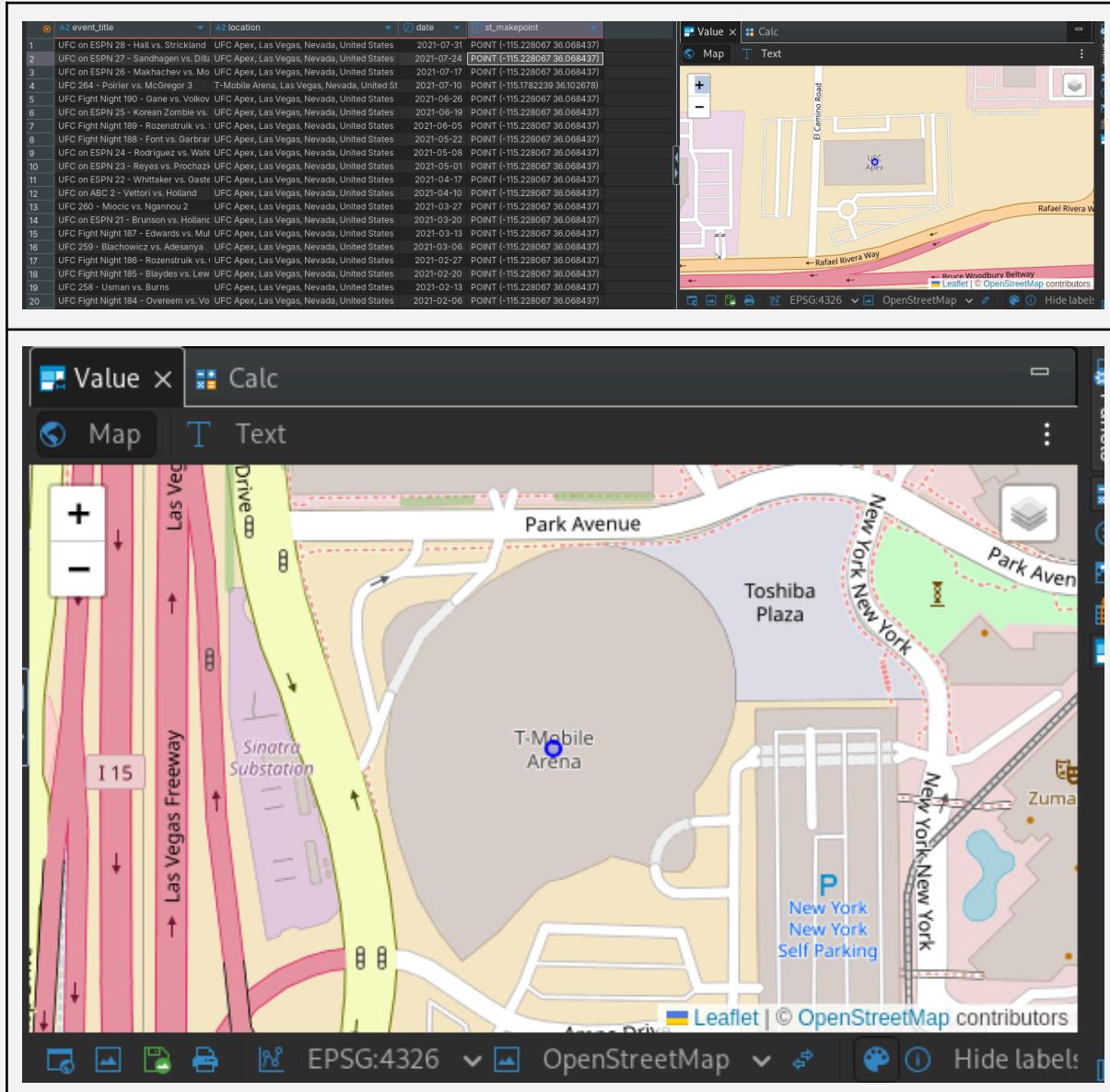
	AZ fighter_name	AZ nickname	AZ ↠ url
1	Sidemar Honorio	Sideco Honor	/fighter/Sidemar-Honorio-20757
2	Sultan Umar	[NULL]	/fighter/Sultan-Umar-222829
3	Randa Markos	Quiet Storm	/fighter/Randa-Markos-75417
4	Marvin Maldonado	The Bulldog	/fighter/Marvin-Maldonado-150275
5	Mariya Agapova	[NULL]	/fighter/Mariya-Agapova-206861
6	Marcin Zywica	The Mauler	/fighter/Marcin-Zywica-72846
7	Jesus Martinez	Chavo	/fighter/Jesus-Martinez-55554
8	Martin Nguyen	The Situ-Asian	/fighter/Martin-Nguyen-96155
9	Lucas Martins	Mineiro	/fighter/Lucas-Martins-100031
10	Ryan Martinez	[NULL]	/fighter/Ryan-Martinez-59376
11	John Marsh	The Bull	/fighter/John-Marsh-9
12	Marcos Vinicius Borges	Vina	/fighter/Marcos-Vinicio-Borges-51728
13	Dodi Mardian	The Maung	/fighter/Dodi-Mardian-269889
14	DeMarcus Simmons	[NULL]	/fighter/DeMarcus-Simmons-264305
15	Marcelo Mello	[NULL]	/fighter/Marcelo-Mello-219
16	Omar Johnson	[NULL]	/fighter/Omar-Johnson-123407
17	Marcel Fortuna	Maozinha	/fighter/Marcel-Fortuna-79995
18	Marco Polo Reyes	El Toro	/fighter/Marco-Polo-Reyes-114125
19	Marcus Brimage	The Bama Beast	/fighter/Marcus-Brimage-21618
20	Marius Enache	[NULL]	/fighter/Marius-Enache-68204

6. Consulta de tres grados de separación: Esta operación rastrea una cadena lógica de tres enfrentamientos sucesivos (Luchador A -> B -> C -> D), lo que cumple con el requerimiento 8 (navegación más de dos veces por relación reflexiva o ciclo). Al requerir múltiples auto-uniones de las tablas de peleas y luchadores para validar la secuencia cronológica y la competitividad entre diferentes atletas, cumple también de forma extensiva con el requerimiento 1 (múltiples JOINs).

	AZ luchador_inicial	AZ evento_1	AZ oponente_nivel1	AZ evento_2	AZ oponente_nivel2	AZ evento_3	AZ oponente_nivel3
1	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 195 - Lawler vs. Condit	Joseph Duffy	UFC 217 - Bisping vs. St. Pierre	James Vick
2	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 195 - Lawler vs. Condit	Joseph Duffy	UFC Fight Night 107 - Manuwa vs. Anderson	Reza Madadi
3	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 195 - Lawler vs. Condit	Joseph Duffy	UFC Fight Night 147 - Till vs. Masvidal	Marc Diakiese
4	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 195 - Lawler vs. Condit	Joseph Duffy	UFC Fight Night 172 - Figueiredo vs. Benavidez 2	Joel Alvarez
5	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 195 - Lawler vs. Condit	Joseph Duffy	UFC Fight Night 90 - Dos Anjos vs. Alvarez	Mitch Clarke
6	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC 216 - Ferguson vs. Lee	Lando Vannata
7	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC 265 - Lewis vs. Gane	Rafael Fiziev
8	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC Fight Night 164 - Blachowicz vs. Jacare	Francisco Trinaldo
9	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC Fight Night 173 - Brunson vs. Shahbazyan	Lando Vannata
10	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC Fight Night 177 - Waterson vs. Hill	Alan Patrick
11	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC Fight Night 181 - Hall vs. Silva	Thiago Moises
12	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC on ESPN 11 - Blaydes vs. Volkov	Clay Guida
13	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC on Fox 24 - Johnson vs. Reis	Rashid Magomedov
14	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC on Fox 27 - Jacare vs. Brunson 2	Erik Koch
15	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 199 - Rockhold vs. Bisping 2	Bobby Green	UFC on Fox 31 - Iaquinta vs. Lee 2	Drakkar Klose
16	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 208 - Holm vs. De Randamie	Jim Miller	UFC 213 - Romero vs. Whittaker	Anthony Pettis
17	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 208 - Holm vs. De Randamie	Jim Miller	UFC 228 - Woodley vs. Till	Alex White
18	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 208 - Holm vs. De Randamie	Jim Miller	UFC 252 - Miocic vs. Cormier 3	Vinc Pichel
19	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 208 - Holm vs. De Randamie	Jim Miller	UFC Fight Night 119 - Brunson vs. Machida	Francisco Trinaldo
20	Conor McGregor	UFC 178 - Johnson vs. Cariaso	Dustin Poirier	UFC 208 - Holm vs. De Randamie	Jim Miller	UFC Fight Night 128 - Barboza vs. Lee	Dan Hooker

7. Consulta de proximidad espacial (PostGIS): Haciendo uso de la extensión PostGIS, esta sentencia resuelve el requerimiento 9 (consulta espacial sobre representaciones geométricas). Utiliza funciones como ST_MakePoint y

ST_DWithin para proyectar coordenadas de longitud y latitud en un plano geográfico y calcular qué eventos ocurrieron en un radio de 10 km respecto a un punto central, demostrando la capacidad del modelo relacional para gestionar datos de ubicación física.



Modelo Relacional extendido

Para el desarrollo del Modelo Relacional Extendido, hemos rediseñado la base de datos original utilizando estrategias de agregación de datos. El objetivo es reducir el número de tablas y mejorar la eficiencia en la recuperación de objetos completos (perfils de luchadores o carteleras de eventos) a costa de una mayor complejidad en las consultas analíticas transversales.

Esquema de Tipos Compuestos (Agregación Nativa de PostgreSQL)

En este primer enfoque, hemos optado por una dirección de agregación centrada en el luchador. La idea es que cada fila de la tabla represente a un luchador con toda su información histórica "incrustada".

Implementación Técnica:

Para lograr esto, primero definimos un tipo de dato estructurado que represente un combate y luego lo integramos en la tabla principal como un array:

```
create type relacional_extendido.tipo_combate as (
    evento_titulo varchar(100),
    data_combate date,
    oponente_nome varchar(100),
    oponente_url text,
    resultado varchar(50),
    metodo_vitoria varchar(100)
);

create table relacional_extendido.luchadores_agregado (
    url text primary key,
    fighter_name varchar(100) not null,
    nickname varchar(100),
    birth_date date,
    country varchar(100),
    estilos_de_loita text[],
    historial_combates relacional_extendido.tipo_combate[]
);
```

La carga de datos es la parte más compleja, ya que requiere el uso de **CTEs (Common Table Expressions)** para agrupar los estilos y pivotar las peleas desde la perspectiva de cada luchador (usando un UNION ALL para captar tanto si fue el primer o el segundo contendiente).

```
-- insertar los datos transformados en la nueva tabla
insert into relacional_extendido.luchadores_agregado (
    url,
    fighter_name,
    nickname,
    birth_date,
    country,
    estilos_de_loita,
```

Proyecto final EDGE: Mateo, Álvaro y Rodrigo

```
historial_combates
)
with

-- 1. agregamos los estilos de cada luchador
estilos_agregados as (
    select
        el.luchador_id,
        array_agg(es.nombre) as lista_estilos
    from public.estilos_luchadores el
    join public.estilos es on el.estilo_id = es.id
    group by el.luchador_id
),

-- 2. preparamos el historial de combates
-- desde la perspectiva de un luchador
todos_los_combates as (
    -- parte a: combates donde el luchador es fighter1
    select
        p.fighter1_url as luchador_url,
        e.event_title,
        e.date,
        oponente.fighter_name as oponente_nome,
        p.fighter2_url as oponente_url,
        p.results,
        p.win_method
    from public.pelea p
    join public.evento e on p.event_id = e.event_id
    join public.luchadores oponente on p.fighter2_url = oponente.url

    union all
    -- parte b: combates donde el luchador es fighter2
    select
        p.fighter2_url as luchador_url,
        e.event_title,
        e.date,
        oponente.fighter_name as oponente_nome,
        p.fighter1_url as oponente_url,
        case p.results
            when 'win' then 'loss'
            when 'loss' then 'win'
            else p.results
        end as results,
```

```
p.win_method
from public.pelea p
join public.evento e on p.event_id = e.event_id
join public.luchadores oponente on p.fighter1_url = oponente.url
),
-- 3. agregamos el historial de combates en un array del tipo que
definimos
historiales_agregados as (
    select
        luchador_url,
        array_agg(
            row(event_title,
                date,
                oponente_nombre,
                oponente_url,
                results,
                win_method)::relacional_extendido.tipo_combate
            order by date desc
        ) as lista_combates
    from todos_los_combates
    group by luchador_url
)
-- consulta final que une toda la información
select
    l.url,
    l.fighter_name,
    l.nickname,
    l.birth_date,
    l.country,
    coalesce(ea.lista_estilos, '{})::text[]),
    coalesce(ha.lista_combates,
    '{}'::relacional_extendido.tipo_combate[])
from public.luchadores l
left join estilos_agregados ea on l.url = ea.luchador_id
left join historiales_agregados ha on l.url = ha.luchador_url;
```

Consultas

-- 1. Consulta de victorias por estilo

```

select
    estilo,
    count(*) as total_victorias
from
    relational_extendido.luchadores_agregado,
    unnest(estilos_de_loita) as estilo,
    unnest(historial_combates) as h
where
    h.resultado = 'win'
group by estilo
order by total_victorias desc;

```

-- 2. Consulta de total de peleas por luchador.

```

select
    fighter_name,
    array_length(historial_combates, 1) as total_peleas
from
    relational_extendido.luchadores_agregado
order by total_peleas desc;

```

-- 3. Consulta de más de 5 victorias por sumisión

```

select
    fighter_name,
    count(*) as vitorias_por_submision
from
    relational_extendido.luchadores_agregado,
    unnest(historial_combates) as h
where
    h.resultado = 'win'
    and h.metodo_vitoria like '%Submission%'
group by fighter_name
having count(*) > 5
order by vitorias_por_submision desc;

```

-- 4. Consulta de origen brasileño o estilo Jiu-Jitsu

```

select fighter_name, country, estilos_de_loita
from relational_extendido.luchadores_agregado
where
    country = 'brazil'
    or 'jiu-jitsu' = any(estilos_de_loita);

```

-- 5. Consulta de búsqueda por patrón de texto 'mar'

```

select fighter_name, nickname, url
from relational_extendido.luchadores_agregado
where url ilike '%mar%';

```

-- 6. Consulta de tres grados de separación

```

select distinct
    l1.fighter_name as loitador_inicial,
    h1.evento_titulo as evento_1,

```

```
12.fighter_name as oponente_nivel_1,
h2.evento_titulo as evento_2,
13.fighter_name as oponente_nivel_2,
h3.evento_titulo as evento_3,
14.fighter_name as oponente_nivel_3
from
relacional_extendido.luchadores_agregado 11
cross join unnest(11.historial_combates) h1
join relational_extendido.luchadores_agregado 12 on
h1.oponente_url = 12.url
cross join unnest(12.historial_combates) h2
join relational_extendido.luchadores_agregado 13 on
h2.oponente_url = 13.url
cross join unnest(13.historial_combates) h3
join relational_extendido.luchadores_agregado 14 on
h3.oponente_url = 14.url
where
11.fighter_name = 'Conor McGregor'
and 11.url != 13.url
and 12.url != 14.url
and h1.data_combate < h2.data_combate
and h2.data_combate < h3.data_combate;
```

La consulta 7 no varía, ya que apunta a la tabla de eventos original y no se ve afectada por el nuevo esquema de luchadores extendido compuesto.

Análisis de impacto:

- Consultas Beneficiadas: Aquellas que acceden a estadísticas individuales. Por ejemplo, la Consulta 2 (Total de peleas) pasa de requerir un JOIN y un GROUP BY masivo a un simple ARRAY_LENGTH(historial_combates, 1). Esto reduce drásticamente el tiempo de ejecución al evitar el cruce de tablas en tiempo real.
- Consultas Perjudicadas: La Consulta 6 (Cadena de oponentes). En un modelo relacional, navegar por claves foráneas indexadas es muy rápido. Aquí, el motor de base de datos debe ejecutar un UNNEST (desempaquetar el array) para cada nivel de la búsqueda, lo cual consume más CPU y memoria.

Esquema JSONB (Agregación Documental)

En el segundo enfoque, hemos cambiado la dirección de agregación hacia el evento. Este modelo sigue una filosofía de "Base de Datos de Documentos", donde el objetivo es recuperar toda la información de un evento (su cartelera completa) en un solo acceso a disco.

Implementación Técnica:

Proyecto final EDGE: Mateo, Álvaro y Rodrigo

Aquí utilizamos el tipo de dato JSONB, que es una representación binaria y eficiente de JSON en PostgreSQL. A diferencia del modelo anterior, los datos de los luchadores se duplican dentro de cada combate para que el documento del evento sea autosuficiente.

```
-- Crear la tabla de eventos con datos agregados en jsonb
create table eventos_schema.eventos_json (
    event_id text primary key,
    event_title varchar(100) not null,
    organisation varchar(100),
    date date,
    location varchar(255),
    cartelera_combates jsonb
);

-- 1: Insertar los datos transformados en la tabla eventos_json
insert into eventos_schema.eventos_json (event_id, event_title,
organisation, date, location, cartelera_combates)

with

-- CTE que agrupa los estilos de lucha por luchador
estilos_por_luchador as (
    select
        el.luchador_id,
        jsonb_agg(es.nombre) as estilos_json
    from
        estilos_luchadores el
    join
        estilos es on el.estilo_id = es.id
    group by
        el.luchador_id
),
combates_json as (
    select
        p.event_id,
        jsonb_build_object(
            'match_nr', p.match_nr,
            'result', p.results,
            'win_method', p.win_method,
            'fighter1', jsonb_build_object(
                'url', l1.url,
```

```

        'name', l1.fighter_name,
        'country', l1.country,
        'styles', coalesce(s1.estilos_json, '[]'::jsonb)
    ) ,
    'fighter2', jsonb_build_object(
        'url', l2.url,
        'name', l2.fighter_name,
        'country', l2.country,
        'styles', coalesce(s2.estilos_json, '[]'::jsonb)
    )
) as combate_obj
from
    pelea p
join luchadores l1 on p.fighter1_url = l1.url
join luchadores l2 on p.fighter2_url = l2.url
left join estilos_por_luchador s1 on l1.url = s1.luchador_id
left join estilos_por_luchador s2 on l2.url = s2.luchador_id
),
-- CTE que agrega todos los combates en un array ordenado por número
de combate
cartelera_final as (
    select
        event_id,
        jsonb_agg(combate_obj order by
            combate_obj->'match_nr') as cartelera
    from combates_json
    group by event_id
)
-- SELECT final que une eventos con sus carteles completas en formato
JSONB
select
    e.event_id,
    e.event_title,
    e.organisation,
    e.date,
    e.location,
    coalesce(cf.cartelera, '[]'::jsonb)
from evento e
join cartelera_final cf on e.event_id = cf.event_id;

```

Consultas

-- 1. Consulta de victorias por estilo

```
select
    estilo,
    count(*) as total_victorias
from
    eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate,
    -- El ganador siempre es fighter1 según el DDL original si
    el resultado es 'win'
    jsonb_array_elements(combate->'fighter1'->'styles') as
estilo
where
    combate->>'result' = 'win'
group by estilo
order by total_victorias desc;
```

-- 2. Consulta de total de peleas por luchador.

```
select
    luchador_nombre,
    count(*) as total_peleas
from (
    select combate->'fighter1'->>'name' as luchador_nombre
    from eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate
    union all
    select combate->'fighter2'->>'name' as luchador_nombre
    from eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate
) as todas_las_peleas
group by luchador_nombre
order by total_peleas desc;
```

-- 3. Consulta de más de 5 victorias por sumisión

```
select
    combate->'fighter1'->>'name' as fighter_name,
    count(*) as vitorias_por_submision
from
    eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate
where
    combate->>'result' = 'win'
    and combate->>'win_method' ilike '%Submission%'
group by fighter_name
having count(*) > 5
order by vitorias_por_submision desc;
```

-- 4. Consulta de origen brasileño o estilo Jiu-Jitsu

```

select distinct
    luchador->>'name' as fighter_name,
    luchador->>'country' as country
from
    eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate,
    lateral (select (combate->'fighter1') union all select
    (combate->'fighter2')) as l(luchador)
where
    luchador->>'country' = 'Brazil'
    or luchador->'styles' @> '['"jiu-jitsu"]' ::jsonb;

```

-- 5. Consulta de búsqueda por patrón de texto 'mar'

```

select
    luchador->>'name' as fighter_name
from
    eventos_schema.eventos_json,
    jsonb_array_elements(cartelera_combates) as combate,
    lateral (select (combate->'fighter1') union all select
    (combate->'fighter2')) as l(luchador)
where
    luchador->>'name' ilike '%mar%';

```

-- 6. Consulta de tres grados de separación

```

select distinct
    e1.cartelera_combates->0->'fighter1'->>'name' as
luchador_inicial,
    e1.event_title as evento_1,
    c1->'fighter2'->>'name' as oponente_nivel_1,
    e2.event_title as evento_2,
    c2->'fighter2'->>'name' as oponente_nivel_2
from
    eventos_schema.eventos_json e1
    cross join jsonb_array_elements(e1.cartelera_combates) c1
    join eventos_schema.eventos_json e2 on
e2.cartelera_combates @>
    jsonb_build_array(jsonb_build_object('fighter1',
    jsonb_build_object('url', c1->'fighter2'->>'url'))))
    cross join jsonb_array_elements(e2.cartelera_combates) c2
where
    e1.cartelera_combates @> '[{"fighter1": {"name": "Conor
McGregor"}}]'
    and e1.date < e2.date;

```

En cuanto a la consulta 7, la extensión PostGIS está optimizada para trabajar sobre columnas numéricas o tipos GEOGRAPHY/GEOMETRY nativos. Si incluyéramos las coordenadas dentro de un campo JSONB, perderíamos la capacidad de utilizar índices espaciales GIST, obligando al motor a realizar un escaneo completo de la tabla (Full Table Scan) y parsear cada JSON para calcular la distancia, lo cual sería ineficiente.

Análisis de impacto:

- **Consultas Beneficiadas:** Las de visualización de cartelera. Si una aplicación necesita mostrar todos los combates de "UFC 264", este modelo responde instantáneamente con un SELECT *. En el modelo relacional, esto requeriría unir 5 tablas (evento, pelea, luchador 1, luchador 2 y sus respectivos estilos).
- **Consultas Perjudicadas:** Las de agregación analítica (Consultas 1, 2 y 3). Para contar victorias por estilo, el motor debe parsear el JSON, navegar por las claves internas y filtrar. Aunque JSONB es rápido, nunca superará la velocidad de una columna relacional indexada. Además, la integridad de datos sufre: si un luchador cambia de apodo, habría que actualizarlo en cada evento donde haya participado.

Comparativa de rendimiento

El modelo de Tipos Compuestos es ideal para sistemas de consulta de perfiles atléticos, mientras que el modelo JSONB es perfecto para sistemas de distribución de contenidos (carteleras), donde la velocidad de lectura del "documento completo" es la prioridad absoluta.

Base de Datos Relacional distribuida

En esta sección crearemos un cluster de 3 contenedores donde 1 hará el rol de coordinador y los otros 2 trabajarán como workers.

- **Creación Cluster**

Para crear los contenedores utilizaremos el siguiente Dockerfile que hemos usado en las prácticas 3:

```
FROM ubuntu:jammy
EXPOSE 5432
ARG DEBIAN_FRONTEND=noninteractive
```

```
RUN apt update && \
    apt install -y curl && \
    curl https://install.citusdata.com/community/deb.sh | \
bash && \
    apt update && \
    apt install -y postgresql-16 && \
    apt install -y postgresql-16-citus-12.1 && \
    apt install -y postgresql-16-postgis-3

COPY pg_hba.conf /etc/postgresql/16/main/

RUN pg_conftool 16 main set shared_preload_libraries citus
RUN pg_conftool 16 main set listen_addresses '*'
RUN pg_conftool 16 main set wal_level 'logical'
CMD service postgresql start;su postgres -c "psql -c 'CREATE
EXTENSION citus'";su postgres
```

Posteriormente crearemos la imagen para los contenedores con el comando:

```
docker buildx build --platform=linux/amd64 -t citus-server:edge-gria "ruta a
carpeta con dockerfile"
```

A continuación se debe crear una red para que los contenedores se puedan comunicar entre sí:

```
docker network create network-edge-gria
```

Ahora sí podemos crear los contenedores e integrarlos en la red utilizando la imagen anteriormente mencionada. Primeramente crearemos el coordinador y especificaremos un puerto disponible, en mi caso:

```
docker run --name=edge-gria-citus-coordinator -dti -p 35432:5432
--platform=linux/amd64 -v c:\temp:/home/alumnobd/host-temp --network
network-edge-gria citus-server:edge-gria
```

Después de tener nuestro coordinador creado ahora lanzamos los workers sin especificar el puerto ya que no accederemos a ellos directamente, eso sí debemos integrarlos en la misma red para que el coordinador se pueda comunicar con ellos:

```
docker run --name=edge-gria-citus-worker1 -dti --platform=linux/amd64
--network network-edge-gria citus-server:edge-gria

docker run --name=edge-gria-citus-worker2 -dti --platform=linux/amd64
--network network-edge-gria citus-server:edge-gria
```

Una vez tenemos nuestra estructura creada, abrimos una nueva conexión en Dbeaver y añadiremos nuestros nodos workers al cluster al coordinador, especificando el puerto del coordinador de esta forma:

```
SELECT * from citus_add_node('edge-gria-citus-worker1', 5432);
SELECT * from citus_add_node('edge-gria-citus-worker2', 5432);
```

- **Importación de los datos en formato relacional**

Modificamos brevemente la estructura respecto al modelo relacional para adaptarla a la ejecución en clúster y a la distribución que haremos posteriormente. La estructura es la siguiente:

```
DROP TABLE IF EXISTS estilos_luchadores;
DROP TABLE IF EXISTS estilos;
DROP TABLE IF EXISTS pelea;
DROP TABLE IF EXISTS evento;
DROP TABLE IF EXISTS luchadores;

CREATE TABLE luchadores (
    url TEXT PRIMARY KEY,
    fighter_name VARCHAR(100) NOT NULL,
    nickname VARCHAR(100),
    birth_date DATE,
    age INT,
    country VARCHAR(100),
    height_cm NUMERIC(5,2),
    weight_kg NUMERIC(5,2),
    association VARCHAR(100),
    weight_class VARCHAR(50),
    wins INT,
    losses INT);

CREATE TABLE estilos (
    id SERIAL primary key,
    nombre VARCHAR(50));

CREATE TABLE estilos_luchadores (
    luchador_id TEXT REFERENCES luchadores(url) ON DELETE CASCADE,
    estilo_id INT REFERENCES estilos(id) ON DELETE CASCADE,
    PRIMARY KEY (luchador_id, estilo_id));

CREATE TABLE evento (
    event_id TEXT PRIMARY KEY,
    event_title VARCHAR(100) NOT NULL,
    organisation VARCHAR(100),
    date DATE,
    location VARCHAR(255),
    latitud NUMERIC(10, 7),
    longitud NUMERIC(11, 8)
);

CREATE TABLE pelea (
    -- Eliminamos SERIAL PRIMARY KEY en pelea_id
    pelea_id INT,
    event_id TEXT REFERENCES evento(event_id) ON DELETE CASCADE,
    match_nr INT,
    fighter1_url TEXT REFERENCES luchadores(url) ON DELETE CASCADE,
    fighter2_url TEXT,
    results VARCHAR(50),
    win_method VARCHAR(100),
    win_details VARCHAR(255),
    referee VARCHAR(100),
    round INT,
    time VARCHAR(10),
    -- NUEVA CLAVE PRIMARIA COMPUUESTA que incluye la columna de distribución (fighter1_url)
    PRIMARY KEY (fighter1_url, event_id, match_nr)
);
```

- **Cambios realizados:**

- Cambiamos la clave primaria de pelea, esto es debido a que en un Cluster la clave primaria debe contener la columna de distribución, que en nuestro caso será fighter1_url. Por eso utilizamos una clave compuesta en lugar de pelea_id

- Eliminamos Serial de la tabla de peleas para utilizar INT. El motivo de este cambio es debido a que Serial depende de secuencias de bases de datos, si dos nodos intentan obtener el siguiente número simultáneamente, se genera latencia o conflictos. Como solución utilizamos INT y simplemente añadimos una columna id al Csv de peleas, otorgando un número distinto a cada una de ellas.

- Se eliminan las cláusulas REFERENCES en columnas que no son claves de distribución, Citus solo puede garantizar una Clave Foránea si los datos de ambas tablas están colocados.

● Distribución

```
• --Tablas de referencia
SELECT create_reference_table('estilos');
SELECT create_reference_table('evento');

• --Tablas distribuidas
SELECT create_distributed_table('luchadores', 'url');

SELECT create_distributed_table('estilos_luchadores', 'luchador_id', colocate_with => 'luchadores');
SELECT create_distributed_table('pelea', 'fighter1_url', colocate_with => 'luchadores');
```

- Tablas de referencia: Elegimos las tablas estilos y evento para ser copiadas íntegramente en todo los nodos worker del clúster. Lo hacemos debido a que contienen información que se une constantemente con las tablas principales. Además estilos es una tabla muy corta y eventos no tiene tantos datos en comparación a pelea y luchadores.

-Tablas distribuidas:

- Luchadores: Esta es la tabla central. Se fragmenta en shards a través de los workers utilizando el hash de la columna url. Utilizamos url debido a que es el identificador único y el campo que vincula a un luchador con sus peleas y sus estilos. Así, nos aseguramos de que la carga de datos esté equilibrada entre todos los servidores.
- Estilos_luchadores: Se distribuye por **luchador_id**. Al colocarla con **luchadores**, Citus garantiza que los estilos de "Luchador A" residan físicamente en el mismo disco duro que los datos personales de "Luchador A".

- Pelea: Se distribuye por **luchador_id**, vinculando la pelea al primer luchador. Al colocarla con **luchadores**, todas las peleas donde un luchador aparece como **fighter1** están en el mismo nodo que su perfil

● Rendimiento de consultas en función de workers operativos

Analizaremos el rendimiento de las consultas en función de los workers en los que se distribuyan los shards. Para ello hemos ejecutado la consulta que cuenta el número de peleas por luchador y los ordena de manera descendentes en función del número de peleas. La hemos modificado de la siguiente manera:

AZ fighter_name	total_peleas
Jim Miller	37
Donald Cerrone	37
Andrei Arlovski	36
Jeremy Stephens	34
Demian Maia	33
Chéick Kongo	33
Diego Sanchez	32
Tito Ortiz	31
Frank Mir	30
Rafael dos Anjos	30
Clay Guida	30
Michael Bisping	29
Lyoto Machida	29
Ben Saunders	29
Charles Oliveira	28
Ryan Bader	28
Matt Brown	28

La consulta tarda aproximadamente 130ms. Esto lo hemos calculado después de 10 ejecuciones y hacer su media. Este rendimiento se produce con 2 workers activos y sus shard distribuidos. Ahora eliminaremos un worker para eso, primeramente debemos desactivar la opción `shouldhaveshards` para que podamos vaciar los shards contenidos en ese worker, ya que sino sera imposible su borrado. El código sería el siguientes:

```
SELECT * FROM citus_set_node_property('edge-gria-citus-worker2', 5432, 'shouldhaveshards', false);
```

Posteriormente ejecutaremos:

```
SELECT * FROM citus_rebalance_start(drain_only := true);
```

```
Scheduled 16 moves as job 1
```

Éste balancea todo los shards del worker 2 al worker 1, permitiéndonos la eliminación del worker2:

```
SELECT * from citus_remove_node('edge-gria-citus-worker2', 5432);
```

	nodeid	groupid	nodename	nodeport	noderack	hasmetadata	isactive	nodeid
1	1	1	edge-gria-citus-worker1	5432	default	[v]	[v]	primary

Volvemos a ejecutar la consulta:

```

SELECT
    l.fighter_name,
    COALESCE(p1.c1, 0) + COALESCE(p2.c2, 0) AS total_peleas
FROM luchadores l
LEFT JOIN (
    SELECT fighter1_url AS url, COUNT(*) AS c1
    FROM pelea
    GROUP BY fighter1_url
) p1 ON l.url = p1.url
LEFT JOIN (
    SELECT fighter2_url AS url, COUNT(*) AS c2
    FROM pelea
    GROUP BY fighter2_url
) p2 ON l.url = p2.url
ORDER BY total_peleas DESC;

SELECT
    fighter_name COALESCE(c1, 0) + COALESCE(c2, 0) AS total_peleas
FROM (
    SELECT
        fighter_name,
        COALESCE(p1.c1, 0) + COALESCE(p2.c2, 0) AS total_peleas
    FROM luchadores l
    LEFT JOIN (
        SELECT fighter1_url AS url, COUNT(*) AS c1
        FROM pelea
        GROUP BY fighter1_url
    ) p1 ON l.url = p1.url
    LEFT JOIN (
        SELECT fighter2_url AS url, COUNT(*) AS c2
        FROM pelea
        GROUP BY fighter2_url
    ) p2 ON l.url = p2.url
    ORDER BY total_peleas DESC
) subquery
ORDER BY total_peleas DESC;

```

AZ fighter_name	l23 total_peleas
Donald Cerrone	37
Jim Miller	37
Andrei Arlovski	36
Jeremy Stephens	34
Cheick Kongo	33
Demian Maia	33
Diego Sanchez	32
Tito Ortiz	31
Clay Guida	30
Rafael dos Anjos	30
Frank Mir	30
Ben Saunders	29
Lyoto Machida	29
Michael Bisping	29
Frankie Edgar	28
Charles Oliveira	28
Gleison Tibau	28

Al ser una consulta bastante simple no visualizamos gran diferencia entre tener uno y 2 workers funcionando. En este caso, el factor limitante no es la potencia de cálculo de los workers, sino la latencia de red y la gestión del coordinador. En un caso donde tuviéramos una grandísima cantidad de datos podríamos ver que el tiempo se duplica al estar trabajando solo uno en vez de 2.

- Uso formato columnar

Ejecutamos el código:

```
SELECT undistribute_table('evento', cascade_via_foreign_keys=>true);
```

Proyecto final EDGE: Mateo, Álvaro y Rodrigo

Borrando la distribución actual para poder modificar la estructura y agregar el formato columnar a las tablas con más datos y que se verán más beneficiadas y eliminamos las referencias a otras tablas:

```
DROP TABLE IF EXISTS estilos_luchadores;
DROP TABLE IF EXISTS estilos;
DROP TABLE IF EXISTS pelea;
DROP TABLE IF EXISTS evento;
DROP TABLE IF EXISTS luchadores;

CREATE TABLE luchadores (
    url TEXT PRIMARY KEY,
    fighter_name VARCHAR(100) NOT NULL,
    nickname VARCHAR(100),
    birth_date DATE,
    age INT,
    country VARCHAR(100),
    height_cm NUMERIC(5,2),
    weight_kg NUMERIC(5,2),
    association VARCHAR(100),
    weight_class VARCHAR(50),
    wins INT,
    losses INT
) USING columnar;

-- PLEA (COLUMNAS)
CREATE TABLE pelea (
    pelea_id INT,
    event_id TEXT, -- NO REFERENCES
    match_nr INT,
    fighter1_url TEXT, -- NO REFERENCES
    fighter2_url TEXT,
    results VARCHAR(50),
    win_method VARCHAR(100),
    win_details VARCHAR(255),
    referee VARCHAR(100),
    round INT,
    time VARCHAR(10),
    PRIMARY KEY (fighter1_url, event_id, match_nr)
) USING columnar;

-- ESTILOS (FILA)
CREATE table estilos (
    id SERIAL primary key,
    nombre VARCHAR(50));

-- EVENTO (FILA)
CREATE TABLE evento (
    event_id TEXT PRIMARY KEY,
    event_title VARCHAR(100) NOT NULL,
    organisation VARCHAR(100),
    date DATE,
    location VARCHAR(255),
    latitud NUMERIC(10, 7),
    longitud NUMERIC(11, 8)
);

-- ESTILOS_LUCHADORES (FILA)
create table estilos_luchadores (
    luchador_id TEXT, -- NO REFERENCES
    estilo_id INT, -- NO REFERENCES
    PRIMARY KEY (luchador_id, estilo_id));
```

Después de crear la nueva estructura volvemos a importar los datos y volvemos a ejecutar la misma consulta anterior y analizamos si utilizar el formato columnar en las tablas pelea y luchadores a tenido impacto o su rendimiento es similar:

	AZ fighter_name	123 total_peleas
1	Jim Miller	37
2	Donald Cerrone	37
3	Andrei Arlovski	36
4	Jeremy Stephens	34
5	Cheick Kongo	33
6	Demian Maia	33
7	Diego Sanchez	32
8	Tito Ortiz	31
9	Clay Guida	30
10	Frank Mir	30
11	Rafael dos Anjos	30
12	Ben Saunders	29
13	Lyoto Machida	29
14	Michael Bisping	29
15	Charles Oliveira	28
16	Gleison Tibau	28
17	Frankie Edgar	28

Esta vez vemos como el tiempo que ha durado la consulta se ha reducido desde 130ms a 70ms lo cuál es un impacto considerable. Podemos sentenciar que el uso del formato columnar en tablas que tienen una cantidad considerable de datos, es una práctica muy recomendada ya que mejora el rendimiento de las consultas, en nuestro caso prácticamente a la mitad.

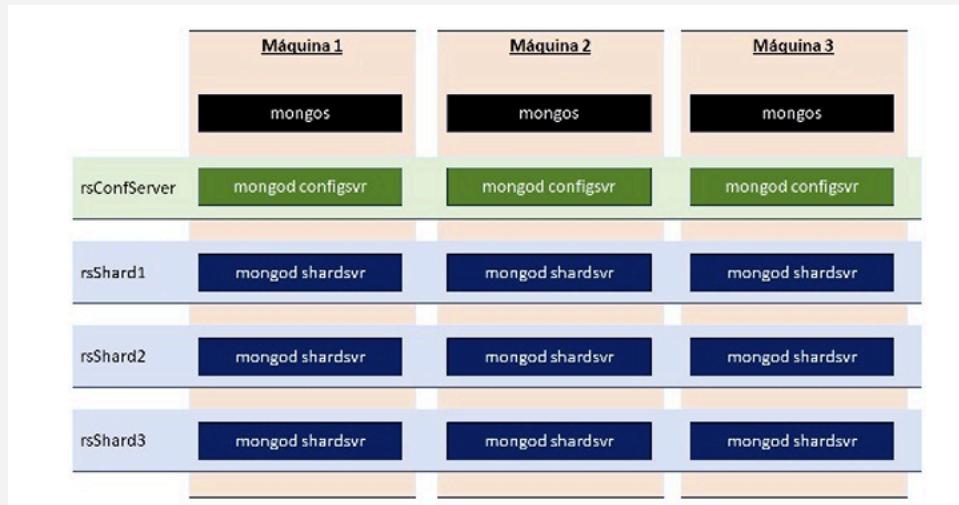
- **Conclusiones**

Nuestro ejemplo no se ve ampliamente beneficiado por el uso del clúster debido a que no tenemos una cantidad de datos lo suficientemente grande como para observar un gran impacto en las consultas, ya que la mayoría del tiempo de ejecución de estas se debe a un overhead de comunicación de la red. Sin embargo, sí que podemos observar una mejoría notable al utilizar el formato columnar en tablas específicas. Esto es debido a que en la ejecución de ciertas consultas el sistema no está obligado a leer toda la fila sino la columna que necesita para la realización de esta, impactando positivamente en el tiempo esperado.

Base de datos NoSQL documental: MongoDB

-CREACIÓN DEL CLUSTER:

Para la creación del cluster de mongoDB, empleamos la arquitectura propuesta en la siguiente imagen:



El clúster de MongoDB que se está creando combina [sharding](#) y [replica sets](#) para lograr escalabilidad y alta disponibilidad. El sharding consiste en dividir los datos en fragmentos (chunks) distribuidos entre varios shards, lo que permite manejar grandes volúmenes de información y mejorar el rendimiento de las consultas, ya que cada shard almacena solo una parte de los datos según la clave de shard definida.

Cada shard y los config servers se implementan como replica sets, es decir, conjuntos de nodos que contienen copias idénticas de los datos o metadatos. Esto garantiza que, si un nodo falla, los demás continúan operando sin pérdida de información, proporcionando tolerancia a fallos y consistencia dentro del clúster. Los procesos [mongos](#) actúan como routers que dirigen las consultas al shard correspondiente usando la información de los config servers.

Creamos un script “cluster.sh” que crea y configura el cluster para facilitar la implementación de este en otras máquinas. A continuación se explican las partes más relevantes de este script en orden de ejecución:

1. Inicialización de los contenedores

```
recreate_container() {
    local name=$1
    docker rm -f $name 2>/dev/null || true
    docker run --name $name -d \
        --network $NETWORK \
        -v $DATA_DIR:$CSV_DIR \
        --entrypoint bash mongo:latest -c "sleep infinity"
}

for m in "${MACHINES[@]}"; do
    recreate_container "$m-$CONFIGSVR"
    for s in "${SHARDS[@]}"; do
        recreate_container "$m-$s"
    done
    recreate_container "$m-$MONGOS"
done
```

Definimos la función `recreate_container` que nos permite crear el contenedor con su configuración (volumen, red, modo dormido...) al invocarla.

Luego creamos un broker, tres shards y un mongo por máquina. En total creamos 15 contenedores.

2. Configuración de los shards y brokers

```
for m in "${MACHINES[@]}"; do
    docker exec -d "$m-$CONFIGSVR" mongod --configsvr --replicaSet rsConfig \
        --port 27017 --bind_ip_all --dbpath /data/configdb
    for s in "${SHARDS[@]}"; do
        rsName="rs${s:5}"
        docker exec -d "$m-$s" mongod --shardsvr --replicaSet "$rsName" \
            --port 27017 --bind_ip_all --dbpath /data/db
    done
done
```

Configuramos los config server y los shards.

- El config server es el encargado de almacenar los metadatos de cada máquina, como por ejemplo la información que almacena cada shard.
- Los shards son los nodos que almacenan los datos reales de la base de datos, distribuyendo la información en fragmentos según la clave de shard definida. Cada shard funciona como un replica set para garantizar alta disponibilidad y replicación de los datos, de manera que si un nodo falla, los demás pueden continuar atendiendo las solicitudes.

3. Iniciación de los replica sets

```
for i in {1..3}; do
    echo "Inicializando replica set rs$i..."
    docker exec "m1-shard$i" mongosh --eval "
rs.initiate({
    _id: 'rs$i',
    members: [
        {_id:0, host:'m1-shard$i:27017'},
        {_id:1, host:'m2-shard$i:27017'},
        {_id:2, host:'m3-shard$i:27017'}
    ]
})
"
done
```

```
docker exec m1-$CONFIGSVR mongosh --eval "
rs.initiate({
    _id: 'rsConfig',
    configsvr: true,
    members: [
        {_id:0, host:'m1-configsvr:27017'},
        {_id:1, host:'m2-configsvr:27017'},
        {_id:2, host:'m3-configsvr:27017'}
    ]
})
"
```

Aquí se configura cada shard como un replica set con tres nodos, garantizando replicación interna y alta disponibilidad de los datos de cada shard.

Se configura el replica set del config server con tres nodos, indicando **configsvr: true**, de modo que almacene los metadatos del clúster y permita que los mongos enruten correctamente las consultas.

4. Inicio de mongos en cada máquina

```
for m in "${MACHINES[@]}"; do
    docker exec -d "$m-$MONGOS" mongos \
        --configdb rsConfig/m1-configsvr:27017,m2-configsvr:27017,m3-configsvr:27017 \
        --bind_ip_all --port $MONGOS_PORT
    wait_for_mongo "$m-$MONGOS" $MONGOS_PORT
done
```

En esta sección se inician los procesos **mongos** en cada máquina del clúster. Los mongos actúan como routers del clúster, utilizando la información de los config servers para enrutar las consultas de los clientes hacia el shard correspondiente. Cada mongos se conecta a todos los config servers y queda listo para gestionar las operaciones de lectura y escritura de forma transparente

5. Agregar shards

```
docker exec m1-$MONGOS mongosh --port $MONGOS_PORT --eval "
sh.addShard('rs1/m1-shard1:27017,m2-shard1:27017,m3-shard1:27017');
sh.addShard('rs2/m1-shard2:27017,m2-shard2:27017,m3-shard2:27017');
sh.addShard('rs3/m1-shard3:27017,m2-shard3:27017,m3-shard3:27017');
sh.status();
"
```

En esta sección se agregan los shards al clúster utilizando el primer mongos. Cada shard, representado por su replica set, se registra en el clúster para que mongo pueda enrutar las consultas correctamente. La función `sh.addShard()` permite que el router conozca todos los shards disponibles.

Podemos ver en la salida del `sh.status()` como se ha agregado correctamente los shards, confirmando que cada replica set está activo y listo para enrutar consultas dentro del clúster..

```
shards
[
  {
    _id: 'rs1',
    host: 'rs1/m1-shard1:27017,m2-shard1:27017,m3-shard1:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1765883870, i: 10 }),
    replSetConfigVersion: Long('-1')
  },
  {
    _id: 'rs2',
    host: 'rs2/m1-shard2:27017,m2-shard2:27017,m3-shard2:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1765883870, i: 28 }),
    replSetConfigVersion: Long('-1')
  },
  {
    _id: 'rs3',
    host: 'rs3/m1-shard3:27017,m2-shard3:27017,m3-shard3:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1765883871, i: 17 }),
    replSetConfigVersion: Long('-1')
  }
]
```

-IMPORTACIÓN DE LOS DATOS:

Creamos otro script llamado “`import.sh`” que importe los csv a colecciones de mongo, ejecute el script de agregación y aplique índices y sharding. Este script emplea mongoimport para importar los csv necesarios a nuestro cluster. Los archivos csv se encuentran ya en el volumen que definimos anteriormente, los importamos usando el siguiente bloque de código.

```
docker exec "$MONGOS" bash -c "
set -e
for csvfile in $CSV_DIR/data/*.csv; do
    [ -e \"\$ csvfile\" ] || continue
    filename=\$(basename \"\$ csvfile\")
    collection=\"\${filename%.*}\"
    echo \"Importando \$collection desde \$filename...\"
    mongoimport --uri mongodb://localhost:$MONGOS_PORT/$DBNAME \
        --collection \"\$collection\" \
        --type csv \
        --headerline \
        --file \"\$ csvfile\"
done
"
```

-AGREGACIÓN DE LOS DATOS:

Empleamos la misma agregación usada en el modelo relacional extendido de JSON, pero modificando el código para que cumpla con la sintaxis que mongodb requiere. Contamos con un script en JavaScript dentro del volumen que realiza la agregación, este script es lo siguiente en ejecutarse en nuestro script de importación principal.

En este pipeline, la dirección de agregación es unidireccional y [centrada en cada luchador](#): se parte de los documentos de la colección luchadores y se van enriqueciendo progresivamente con información relacionada en otras colecciones, como estilos y combates, mediante `$lookup` y `$addFields`. Cada etapa toma los resultados de la anterior y los transforma o combina, siguiendo un flujo secuencial de arriba hacia abajo, hasta construir un documento final completo por luchador que luego se guarda en `luchadores_agregados`.



Partiendo de nuestro archivo csv de luchadores inicial realizamos un [aggregate](#).

```
{
  $lookup: {
    from: "estilos_luchadores",
    localField: "url",
    foreignField: "luchador_id",
    as: "estilos_rel"
  },
  {
    $lookup: {
      from: "estilos",
      localField: "estilos_rel.estilo_id",
      foreignField: "id",
      as: "estilos_nombres" } },
  {
    $addFields: {
      estilos_de_loita: {
        $map: {
          input: "$estilos_nombres",
          as: "e",
          in: "$$e.nombre" } } } },
}
```

Este fragmento de agregación toma cada documento de la colección luchadores y le añade la información de sus estilos. Primero, mediante \$lookup, obtiene los registros de estilos_luchadores que coinciden con el url del luchador y los guarda en estilos_rel. Luego, realiza un segundo \$lookup para convertir los estilo_id de esos registros en los nombres reales de los estilos desde la colección estilos, almacenándolos en estilos_nombres. Por último añadimos un campo llamado estilos_de_lucha al documento de luchador, que contiene un arreglo con los nombres de los estilos obtenidos, usando \$map para extraer únicamente el campo nombre.

Esta parte del pipeline realiza dos \$lookup sobre la colección peleas para obtener los combates de cada luchador. El primer \$lookup busca todos los combates donde el luchador aparece como fighter1 y los almacena en combates_f1. El segundo \$lookup busca los combates donde el luchador aparece como fighter2 y los guarda en combates_f2, permitiendo luego combinar toda la información de los enfrentamientos de cada luchador.

```
{
  $lookup: {
    from: "peleas",
    localField: "url",
    foreignField: "fighter1_url",
    as: "combates_f1"
  },
  {
    $lookup: {
      from: "peleas",
      localField: "url",
      foreignField: "fighter2_url",
      as: "combates_f2"
    } },
}
```

```
{  
  $addFields: {  
    historial_combates: {  
      $concatArrays: [  
        {  
          $map: {  
            input: "$combates_f1",  
            as: "c",  
            in: {  
              evento_titulo: "$$c.event_id",  
              oponente_url: "$$c.fighter2_url",  
              resultado: "$$c.results",  
              metodo_vitoria: "$$c.win_method"  
            }  
          }  
        },  
        {  
          $map: {  
            input: "$combates_f2",  
            as: "c",  
            in: [  
              evento_titulo: "$$c.event_id",  
              oponente_url: "$$c.fighter1_url",  
              resultado: {  
                $switch: {  
                  branches: [  
                    { case: { $eq: ["$$c.results", "win"] }, then: "loss" },  
                    { case: { $eq: ["$$c.results", "loss"] }, then: "win" }  
                  ],  
                  default: "$$c.results"  
                }  
              },  
              metodo_vitoria: "$$c.win_method"  
            ]  
          }  
        }  
      ]  
    }  
  }  
}
```

Este fragmento de agregación de MongoDB crea un campo llamado historial_combates que unifica todos los combates de un luchador en un solo array. Para ello, recorre los combates donde aparece como fighter1 y fighter2, extrayendo la información relevante del evento, fecha, oponente, resultado y método de victoria. En los combates donde el luchador es fighter2, el resultado se invierte para mantener siempre la perspectiva del luchador analizado. El resultado final es un historial coherente y homogéneo de todos sus combates, independientemente de su rol original en cada pelea.

Eliminamos las bases de datos que importamos y usamos para hacer la agregación usando `drop`. Después de realizar la agregación tenemos un tipo de dato más compacto y documental, abajo se muestra un ejemplo de este:

```
{  
    _id: ObjectId('69467516c69e3f62025945b4'),  
    url: '/fighter/Hiroshi-Nakamura-12076',  
    fighter_name: 'Hiroshi Nakamura',  
    nickname: 'Iron',  
    birth_date: '14/02/1981',  
    country: 'Japan',  
    estilos_de_loita: [],  
    historial_combates: [  
        {  
            evento_titulo:  
                '/events/BFC-Bellator-Fighting-Championships-64-20395',  
                oponente_url: '/fighter/Rodrigo-Lima-38280',  
                resultado: 'win',  
                metodo_vitoria: 'Decision'  
        },  
        {  
            evento_titulo:  
                '/events/BFC-Bellator-Fighting-Championships-70-21571',  
                oponente_url: '/fighter/Luis-Alberto-Nogueira-29850',  
                resultado: 'loss',  
                metodo_vitoria: 'KO'  
        } ...  
    ]  
}
```

-INDEXACIÓN Y SHARDING:

El siguiente paso de nuestro script de importación es ejecutar un script ubicado en el volumen del contenedor que realiza la indexación y aplica el sharding.

Nuestros datos no tienen un tamaño superior a 64MB que es el tamaño por defecto de `chunksize`, por lo tanto no se van a shardear. Para forzar que se dividan en chunks nos conectamos a la base de datos `config` y cambiamos el chunksize por 1MB, el tamaño mínimo permitido.

```
const configDB = db.getSiblingDB("config");
configDB.settings.updateOne(
  { _id: "chunksizer" },
  { $set: { value: 1 } },
  { upsert: true };
```

A continuación realizamos el particionamiento. Elegimos emplear como estrategia de particionamiento **Hashed Sharding**. No usamos **Ranged Sharding** porque este último distribuye los datos por rangos de valores, lo que podría generar hotspots: shards desbalanceados si muchos documentos tienen valores similares o consecutivos en la clave de shard. Con Hashed Sharding, cada valor se transforma en un hash, lo que asegura una distribución uniforme de los documentos entre los shards y evita que un nodo reciba mucha carga mientras otros permanecen casi vacíos. Como **shard key** usamos la url de cada luchador, ya que es un valor único por luchador, lo que garantiza que cada documento se ubique de forma determinística en un shard y permite búsquedas rápidas por ese campo.

Para habilitar el sharding y aplicarlo en la colección, nuestro script usa los siguientes comandos:

```
const dbLuchas = db.getSiblingDB("luchasDB");
dbLuchas.luchadores_agregados.createIndex({ url: "hashed" });
sh.enableSharding("luchasDB");
sh.shardCollection("luchasDB.luchadores_agregados", { url: "hashed" });
```

A continuación creamos los índices secundarios cuya función es la de acelerar las consultas que no usan la shard key y mejorar la eficiencia de búsquedas, filtrados y agregaciones sobre campos específicos. Estos son los índices secundarios que escogimos. La elección de estos índices en específico se debe a que son los más usados en las consultas propuestas.

```
dbLuchas.luchadores_agregados.createIndex({ fighter_name: 1 });
db.luchadores_agregados.createIndex({ "historial_combates.resultado": 1, "historial_combates.metodo_vitoria": 1 })
db.luchadores_agregados.createIndex({ estilos_de_loita: 1 })
```

Esto deja la colección más eficiente para las consultas más frecuentes, sin afectar la distribución de los datos en los shards.

-EXPLICACIÓN DE LAS CONSULTAS:

Los resultados de las consultas son idénticos a los obtenidos con las consultas en el apartado del modelo relacional. Resolvemos las consultas con mongodb:

1. Número de victorias por estilo en orden descendente.

```
db.luchadores_agregados.aggregate([
  { $unwind: "$historial_combates" },
  { $unwind: "$estilos_de_loita" },
  { $match: { "historial_combates.resultado": "win" } },
  { $group: {
    _id: "$estilos_de_loita",
    vitorias_por_estilo: { $sum: 1 }
  }},
  { $sort: { vitorias_por_estilo: -1 } }
])
```

Usamos **unwind** para desplegar cada elemento del array que contiene todos los combates de cada luchador. Volvemos a usar unwind para desplegar los estilos de lucha de cada luchador. Filtramos por los combates que han sido ganados usando **match**. Agrupamos por estilo de lucha usando **group** y contamos cuántas victorias tiene cada estilo con sum. Por último ordenamos de mayor a menor usando **sort**.

2. Número de peleas de cada luchador en orden descendente.

```
db.luchadores_agregados.aggregate([
  { $unwind: "$historial_combates" },
  { $group: {
    _id: "$fighter_name",
    total_peleas: { $sum: 1 }
  }},
  { $sort: { total_peleas: -1 } },
  { $project: { _id: 0, luchador: "$_id", total_peleas: 1 } }
])
```

Primero, **unwind** despliega cada elemento del array **historial_combates** para que cada combate sea un documento individual. Luego, **group** agrupa los documentos por el nombre del luchador (**fighter_name**) y cuenta cuántos combates tiene cada uno con **\$sum: 1**. Después, **sort** ordena los luchadores de mayor a menor según el total de peleas, y finalmente **project** muestra solo los campos deseados, renombrando **_id** a **luchador** y manteniendo **total_peleas**.

3. Luchadores con más de 5 victorias por sumisión.

```
db.luchadores_agregados.aggregate([
  { $unwind: "$historial_combates" },
  { $match: { "historial_combates.resultado": "win",
  "historial_combates.metodo_vitoria": /Submission/i } },
  { $group: { _id: "$fighter_name", vitorias_por_submision: { $sum: 1
} } },
  { $match: { vitorias_por_submision: { $gt: 5 } } },
  { $sort: { vitorias_por_submision: -1 } },
  { $project: { _id: 0, fighter_name: "$_id", vitorias_por_submision: 1
} } ]);
```

Filtramos los combates ganados por sumisión usando `match`. Después, agrupamos por luchador y contamos cuántas victorias por sumisión tiene cada uno. A continuación, otro `match` filtra solo a los luchadores con más de 5 victorias por sumisión. Finalmente, `sort` ordena de mayor a menor número de victorias y `project` muestra los campos finales renombrando `_id` a `fighter_name` y manteniendo `vitorias_por_submision`.

4. Luchadores de Brasil o especialistas en jiu-jitsu

```
db.luchadores_agregados.aggregate([
  { $match: { $or: [ { country: "Brazil" }, { estilos_de_loita:
"jiu-jitsu" } ] } },
  { $project: { fighter_name: 1, country: 1, estilos_de_loita: 1, _id:
0 } },
  { $limit: 5
])
```

Primero, `match` filtra los documentos donde el luchador sea de Brasil (`country: "Brazil"`) o tenga "jiu-jitsu" en su array de estilos . Luego, `project` selecciona solo los campos deseados (`fighter_name`, `country`, `estilos_de_loita`) y elimina `_id` para una presentación más limpia..

5. Luchadores con “mar” en su nombre

```
db.luchadores_agregados.find(
  { url: /mar/i },
  { _id: 0, fighter_name: 1, nickname: 1, url: 1 }
).limit(5)
```

Usamos una expresión regular insensible a mayúsculas para encontrar los luchadores que contengan “mar” en su url.

6. Consulta del ciclo

```
db.luchadores_agregados.aggregate([
  { $match: { fighter_name: "Conor McGregor" } },
  // ===== NIVEL 1 =====
  { $unwind: "$historial_combates" },
  {
    $lookup: {
      from: "luchadores_agregados",
      localField: "historial_combates.oponente_url",
      foreignField: "url",
      as: "l2"
    }
  },
  { $unwind: "$l2" },
  // ===== NIVEL 2 =====
  { $unwind: "$l2.historial_combates" }, ...
```

La consulta parte del luchador inicial, Conor McGregor, y descompone su historial de combates mediante `$unwind` para obtener todos sus oponentes directos, constituyendo el primer nivel de separación. A continuación, mediante `$lookup` sobre la misma colección, se recupera el documento completo de cada oponente y se repite el proceso sobre su propio historial de combates para construir el segundo nivel. Este encadenamiento de descomposición y unión permite modelar una cadena de relaciones entre luchadores, avanzando progresivamente por los distintos grados de separación definidos en el pipeline.

```
... $match: {
  $expr: {
    $and: [
      { $ne: ["$url", "$l3.url"] },    // 11 != 13
      { $ne: ["$l2.url", "$l4.url"] } // 12 != 14 ...
```

Luego se aplican los filtros `$match` y `$expr` que se utilizan para evitar inconsistencias lógicas y ciclos dentro de la cadena de relaciones. En concreto, se impide que el luchador inicial reaparezca en niveles posteriores y que un luchador de un nivel intermedio coincida con uno de niveles más avanzados.

-CONSULTAS BENEFICIADAS Y PERJUDICADAS:

La gran mayoría de consultas han sido beneficiadas al haber eliminado las relaciones y centralizado la información en la colección luchadores_agregados, lo que permite acceder directamente a los datos sin necesidad de múltiples joins o lookups. Consultas de agregación, filtrado por luchador, estilo, país o resultado de combates se ejecutan de manera más rápida gracias a los índices secundarios y al sharding que distribuye uniformemente los documentos.

Algunas consultas como la del ciclo de luchadores (3 grados de separación) se han visto perjudicadas porque, al centralizar los combates dentro de un solo documento (historial_combates), recorrer múltiples niveles requiere pipelines largos con \$unwind y \$lookup encadenados. Esto hace que la consulta sea más compleja, difícil de mantener y con un rendimiento menor, especialmente si la colección contiene muchos documentos o si se necesitan más grados de separación.

-PRUEBAS DETENIENDO MÁQUINAS:

Para parar las máquinas ejecutamos un docker stop sobre todos los contenedores de esta. En nuestro caso sería:

```
docker stop m3-mongos m3-configsvr m3-shard1 m3-shard2 m3-shard3
```

Si paramos una de las máquinas no pasa nada. Las consultas tardan un poco más en ejecutarse cuando paramos m1, la primaria, ya que mongodb tiene que elegir una nueva primaria.

Si paramos dos máquinas, incluida la primaria:

```
[direct: mongos] luchasDB> load("home/alumnobd/host-temp/scripts/consultas.js")
MongoServerError[FailedToSatisfyReadPreference]: Could not find host matching read
preference { mode: "primary" } for set rs2
```

Esto ocurre porque cada replica set necesita un **quorum** (mayoría) de nodos activos para elegir un primario. Si no hay quorum, el replica set no tiene primaria, y cualquier operación con **readPreference: primary** falla. La opción de preferencia de lectura por defecto es siempre primary. MongoDB define el quorum como más de la mitad de los nodos del replica set. Por ejemplo, en un replica set de 3 nodos, se necesitan al menos 2 nodos activos para que exista un primario. Por eso al parar dos máquinas obtenemos ese error.

Podemos probar a cambiar las opciones de preferencia para ver el impacto que tiene en el comportamiento de las consultas cuando tenemos paradas dos de las tres máquinas. Para cambiar las opciones de preferencia usamos el comando:

```
db.getMongo().setReadPref("readPreference")
```

A continuación se definen las opciones de preferencia que vamos a probar y sus resultados en las consultas:

- **primaryPreferred** : Intenta leer del nodo primario, pero si este no está disponible, lee de un nodo secundario. Con esta opción, las consultas siguen funcionando incluso si el primario del replica set ha caído, ya que se redirigen automáticamente a un secundario disponible.
- **secondary** : La lectura se realiza exclusivamente desde un nodo secundario. Con esta opción, las consultas pueden ejecutarse mientras haya al menos un nodo secundario activo, independientemente del estado del primario.
- **secondaryPreferred** : Intenta leer de un nodo secundario, y si no hay ninguno disponible, lee del primario. Esta opción también permite que las consultas continúen funcionando si los secundarios están disponibles.
- **nearest**: La lectura se realiza desde cualquier nodo del replica set (primario o secundario), seleccionando el nodo que minimice la latencia de respuesta. Con esta opción, las consultas siguen ejecutándose mientras haya al menos un nodo activo, optimizando el tiempo de respuesta.

Los compromisos de lectura nos permiten controlar las propiedades de consistencia y aislamiento de los datos que se leen desde los replica sets y clusters shardeados.

Ahora probamos con distintos compromisos de lectura para ver para ver el impacto que tiene en el comportamiento de las consultas cuando tenemos paradas dos de las tres máquinas. Para especificar los compromisos de lectura en una operación de lectura usamos el comando:

```
db.collection.find().readConcern('name')
```

La opción de preferencia empleada para probar los compromisos de lectura es **secondary**. A continuación se definen los compromisos de lectura que vamos a probar y sus resultados en las consultas:

- **local** : El compromiso de lectura local no comprueba que los datos hayan sido confirmados por la mayoría de los nodos. Con la preferencia **secondary**, las consultas funcionan correctamente, ya que el secundario devuelve la información disponible en su estado local sin necesidad de mayoría ni primario.

- **available** : El compromiso available prioriza la disponibilidad y no requiere confirmación por mayoría. En un entorno sharded puede reducir la latencia frente a local. Las consultas se ejecutan correctamente al leer directamente desde el nodo secundario activo.
- **majority** : El compromiso majority garantiza que los datos leídos han sido confirmados por la mayoría de las réplicas. En las pruebas realizadas, las consultas funcionan, pero presentan una latencia elevada debido a los intentos de MongoDB por satisfacer el requisito de mayoría.
- **linearizable** : El compromiso linearizable solo puede utilizarse en lecturas sobre el primario y garantiza la lectura del valor más reciente. En este escenario no funciona, produciendo un error de conexión, ya que no existe un nodo primario activo.
- **snapshot** : El compromiso snapshot solo está disponible dentro de transacciones y requiere confirmación por mayoría. En las pruebas realizadas no funciona y las consultas quedan bloqueadas debido a la falta de mayoría en el replica set.

-GUIA DE EJECUCIÓN:

Dentro de la carpeta nosql/mongodb ejecutamos el script cluster.sh, cambiando la variable DATA_DIR por vuestra ruta a la carpeta import_data.

Luego ejecutamos el script import.sh y ya tenemos el cluster funcionando.

Podemos entrar a el con:

```
docker exec -it m1-mongos mongosh --port 27019
```

Nos conectamos a nuestra base de datos:

```
use luchasDB
```

Ahora podemos ejecutar todas las consultas con :

```
load("home/alumnobd/host-temp/scripts/consultas.js")
```

-CONCLUSIONES:

Como conclusión, la migración de la base de datos de SQL a MongoDB permitió centralizar y desnormalizar la información, eliminando la necesidad de múltiples joins y mejorando significativamente la eficiencia de las consultas. La estructura basada en documentos facilitó el acceso a datos complejos, como historiales de combates y relaciones entre luchadores y eventos, de forma más directa y rápida. Además, MongoDB ofrece mayor flexibilidad para escalar y adaptar el modelo a futuros requerimientos sin alterar la estructura de tablas rígidas, lo que simplifica el mantenimiento y la evolución de la base de datos.

Base de datos NoSQL de grafos: Neo4J

-CONFIGURACIÓN INICIAL:

Comenzamos creando un script “`build.sh`” que crea el contenedor e instala los plugins necesarios para hacer la conexión con PostgreSQL.

```
# Ejecutar contenedor Neo4j
docker run --name $CONTAINER \
-p 7474:7474 -p 7687:7687 \
-d --platform=linux/amd64 \
-e NEO4J_AUTH=neo4j/pwalumnobd \
-v $RUTA_AL_VOLUMEN:/home/alumnobd/host-temp \
--network network-edge-gria \
neo4j:5.26.0

# Esperar unos segundos a que el contenedor arranque
echo "Esperando a que Neo4j arranque..."
sleep 5

# Copiar plugins y cambiar permisos
docker exec -u root $CONTAINER bash -c "
cd /
cp /home/alumnobd/host-temp/config_data/*.jar /var/lib/neo4j/plugins/ || true
chown neo4j:neo4j /var/lib/neo4j/plugins/*.jar || true
"
```

En esta captura de una parte de nuestro script se puede ver como lanzamos el contenedor y le instalamos los plugins necesarios para realizar la conexión JDBC. Tenemos que tener nuestro contenedor de PostgreSQL con la base de datos relacional activo y conectado a la misma red para que se puedan conectar entre sí.

-IMPORTACIÓN DE LOS DATOS:

En el volumen asignado a nuestro contenedor de neo4j tenemos un script de importación “`import.cql`”. Para ejecutarlo, entramos en nuestro contenedor en modo bash interactivo, nos logueamos en cypher-shell y lo ejecutamos con :source ruta/import.cql. Este script nos permite migrar nuestra base de datos relacional a un modelo de grafos gracias a la conexión JDBC.

En esta captura se puede ver como migramos la tabla de los luchadores:

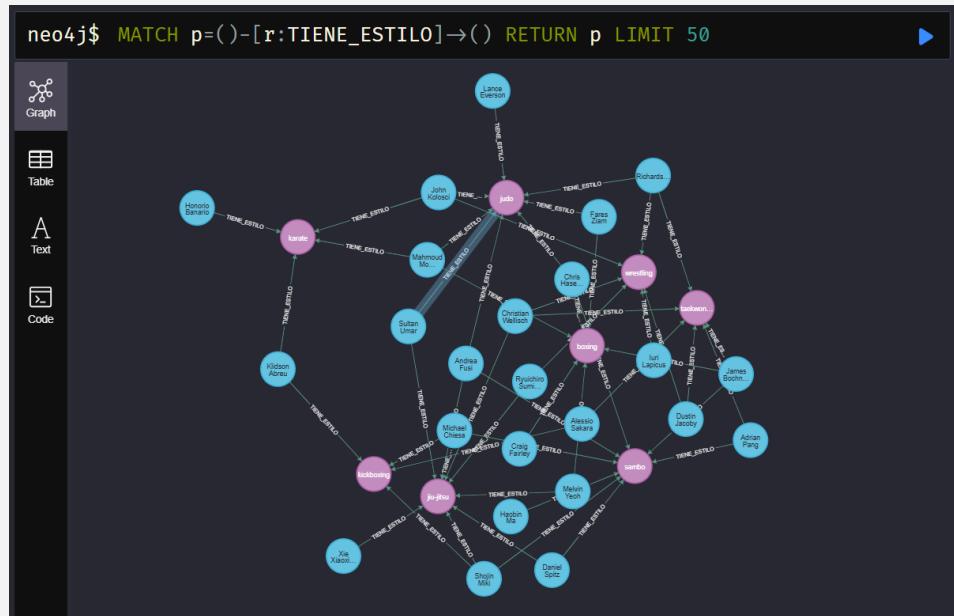
Proyecto final EDGE: Mateo, Álvaro y Rodrigo

```
CALL apoc.load.jdbc(
  "jdbc:postgresql://edge-gria-pgsql/alumnobd?user=alumnobd&password=pwalumnobd",
  "SELECT url, fighter_name, nickname, birth_date, age, country, height_cm, weight_kg,
  | association, weight_class, wins, losses FROM luchadores"
) YIELD row
CREATE (1:Luchador {
  url: row.url, nombre: row.fighter_name, nickname: row.nickname,
  birthDate: row.birth_date, age: row.age, country: row.country,
  heightCm: row.height_cm, weightKg: row.weight_kg, association: row.association,
  weightClass: row.weight_class, wins: row.wins, losses: row.losses
});
```

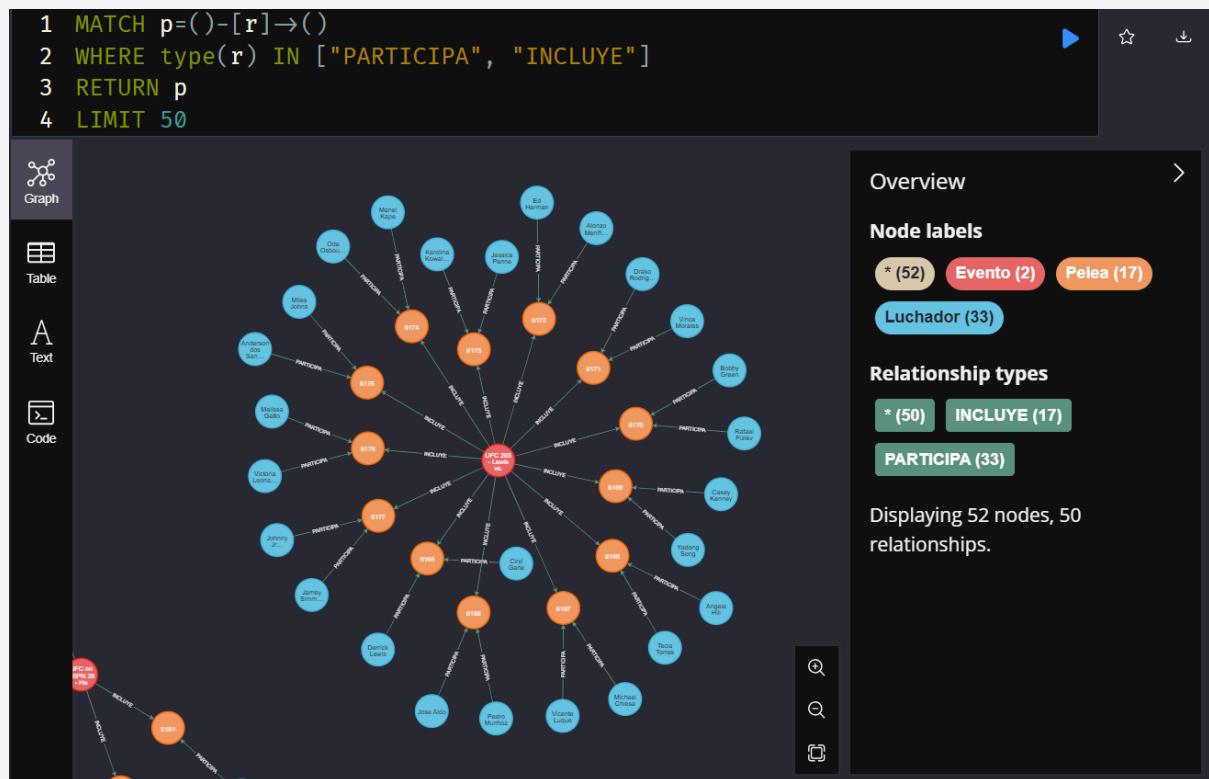
Neo4j no realiza distinciones explícitas entre cardinalidades de las relaciones, ya que todas las relaciones se definen y gestionan de la misma forma a nivel del motor. Dado que los datos se importan desde una base de datos relacional, se asume que las restricciones de cardinalidad ya se cumplen en el origen y se preservan durante el proceso de carga. En la siguiente captura se puede ver cómo se crea la relación entre los luchadores y los estilos.

```
CALL apoc.load.jdbc(
  "jdbc:postgresql://edge-gria-pgsql/alumnobd?user=alumnobd&password=pwalumnobd",
  "SELECT luchador_id, estilo_id FROM estilos_luchadores"
) YIELD row
MATCH (l:Luchador {url: row.luchador_id})
MATCH (e:Estilo {id: row.estilo_id})
CREATE (l)-[:TIENE_ESTILO]->(e);
```

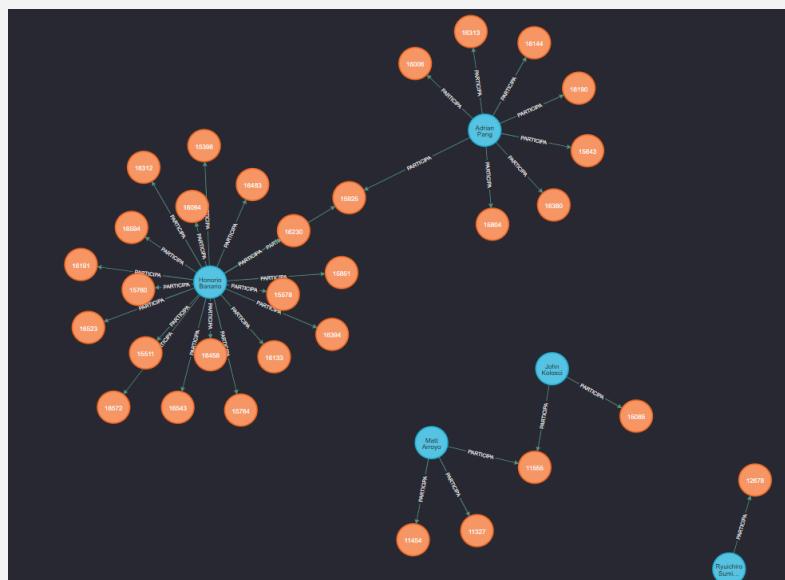
Después de ejecutar el script completo, podemos ver en el cliente web los nodos y relaciones que hemos creado. La siguiente captura corresponde a la relación creada en la captura anterior (**TIENE_ESTILO**) que relaciona a los luchadores con sus estilos:



Contamos con más relaciones que podemos visualizar en el cliente web. Con esta consulta podemos ver dos de ellas. Los nodos en azul son luchadores, los naranjas son peleas y los rojos son eventos. La relación **PARTICIPA** relaciona a dos luchadores con un combate mientras que la relación **INCLUYE** relaciona varios combates con un evento. Se puede apreciar cómo se forman agrupaciones a partir de un evento:



En esta otra imagen se pueden ver como los luchadores están relacionados entre sí mediante las peleas que han realizado.



-REALIZACIÓN DE LAS CONSULTAS:

A continuación resolveremos las consultas de la sección 3 en nuestra base de datos de grafos.

1. Número de victorias por estilo en orden descendente.

```
MATCH (l:Luchador)-[:TIENE_ESTILO]->(e:Estilo),
      (l)-[r:PARTICIPA]->(p:Pelea)
WHERE r.rol = "fighter1" AND NOT p.results IN ['draw', 'NC']
RETURN e.nombre AS estilo,
       COUNT(*) AS total_victorias
ORDER BY total_victorias DESC;
```

Se utilizan patrones de nodos y relaciones para navegar entre luchadores, estilos y peleas, y el **WHERE** filtra por rol y resultados. La agregación se hace con **COUNT(*)** y **ORDER BY** organiza los resultados de mayor a menor.

2. Número de peleas de cada luchador en orden descendente.

```
MATCH (l:Luchador)
OPTIONAL MATCH (l)-[:PARTICIPA]->(p:Pelea)
RETURN l.nombre AS luchador,
       COUNT(p) AS total_peleas
ORDER BY total_peleas DESC;
```

Aquí se usa **OPTIONAL MATCH** para incluir luchadores aunque no tengan peleas registradas, y **COUNT(p)** cuenta cuántas relaciones de participación existen por luchador. **ORDER BY** permite ordenar los resultados en forma descendente según el total de peleas.

3. Luchadores con más de 5 victorias por sumisión.

```
MATCH (l:Luchador)-[:PARTICIPA]->(p:Pelea)
WHERE p.win_method CONTAINS "Submission"
WITH l, COUNT(p) AS vitorias_por_submision
WHERE vitorias_por_submision > 5
RETURN l.fighter_name, vitorias_por_submision
ORDER BY vitorias_por_submision DESC;
```

La cláusula **WITH** permite agregar resultados intermedios, contando las peleas por sumisión para cada luchador antes de filtrarlas con un segundo **WHERE**. Esto muestra

cómo combinar agregación y filtrado en múltiples pasos dentro del mismo pipeline de consulta.

4. Luchadores de Brasil o especialistas en jiu-jitsu

```
MATCH (l:Luchador)-[:TIENE_ESTILO]->(e:Estilo)
WHERE l.country = "Brazil"
RETURN l.fighter_name AS fighter_name, l.country AS country, e.nombre AS estilo
UNION
MATCH (l:Luchador)-[:TIENE_ESTILO]->(e:Estilo)
WHERE e.nombre = "jiu-jitsu"
RETURN l.fighter_name AS fighter_name, l.country AS country, e.nombre AS estilo
```

Se utiliza **UNION** para combinar dos patrones distintos de búsqueda: por país o por estilo, y cada **MATCH** puede retornar los mismos campos. Esto permite unificar resultados de consultas separadas en un solo conjunto de salida.

5. Luchadores con “mar” en su nombre o apellido

```
MATCH (l:Luchador)
WHERE toLower(l.url) CONTAINS "mar"
RETURN l.fighter_name AS fighter_name, l.nickname AS nickname, l.url AS url;
```

Se aplica **toLower()** junto con **CONTAINS** para realizar una búsqueda insensible a mayúsculas dentro de un campo de texto. Esta sintaxis de filtrado permite emular el comportamiento de un LIKE '%mar%' en SQL directamente en Cypher.

6. Consulta del ciclo

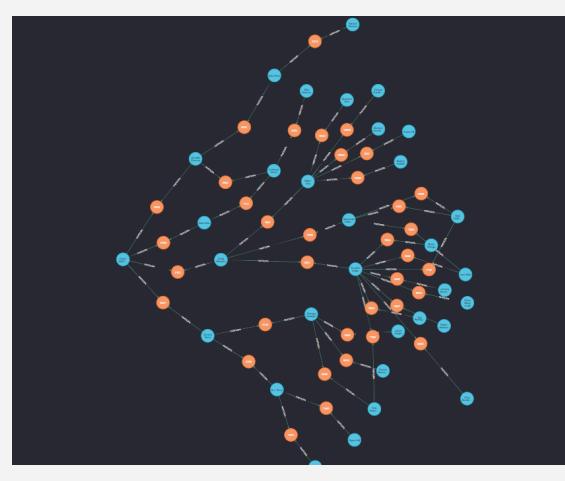
```
MATCH (l1:Luchador {nombre: "Conor McGregor"})-[:PARTICIPA]->(p1:Pelea)<-[::PARTICIPA]-(l2:Luchador)
MATCH (l2)-[:PARTICIPA]->(p2:Pelea)<-[::PARTICIPA]-(l3:Luchador)
MATCH (l3)-[:PARTICIPA]->(p3:Pelea)<-[::PARTICIPA]-(l4:Luchador)
WITH l1, l2, l3, l4, p1, p2, p3
MATCH (ev1:Evento)-[:INCLUYE]->(p1),
      (ev2:Evento)-[:INCLUYE]->(p2),
      (ev3:Evento)-[:INCLUYE]->(p3)
WHERE l1 <> l3 AND l2 <> l4 AND
      p1 <> p2 AND p2 <> p3 AND
      ev1.date < ev2.date AND ev2.date < ev3.date
RETURN DISTINCT
      l1.nombre AS luchador_inicial,
      ev1.title AS evento_1,
      l2.nombre AS oponente_nivel_1,
      ev2.title AS evento_2,
      l3.nombre AS oponente_nivel_2,
```

```
ev3.title AS evento_3,  
l4.nombre AS oponente_nivel_3  
LIMIT 50;
```

La consulta utiliza patrones de nodos y relaciones encadenadas para recorrer tres niveles de oponentes. Se aplican filtros con **WHERE** para evitar ciclos y combinaciones redundantes, y se conectan las peleas con sus eventos mediante relaciones **INCLUYE**. Finalmente, **RETURN DISTINCT** proyecta los nombres de los luchadores y los títulos de los eventos, y **LIMIT** controla la cantidad de resultados devueltos.

Esto es especialmente ventajoso en Neo4j, ya que el motor de grafos navega directamente por las relaciones existentes sin necesidad de unir múltiples tablas, lo que simplifica la consulta y mejora el rendimiento en recorridos de varios niveles.

En Neo4j, las relaciones están explícitamente modeladas como aristas, lo que permite recorrer múltiples niveles de oponentes de manera directa sin múltiples JOIN complejos. Esto reduce la cantidad de combinaciones intermedias y mejora el rendimiento en consultas de varios grados de separación. La sintaxis de patrones de nodos hace que la consulta sea más legible y fácil de mantener que una cadena larga de joins. Además, el motor de grafos está optimizado para recorridos de relaciones, mientras que en SQL cada nivel adicional incrementa exponencialmente el costo de las uniones. Por último, Neo4j permite explorar visualmente el grafo en el cliente web, facilitando el análisis de conexiones entre luchadores y peleas.



Si visualizamos la salida de esta consulta (limitada a 25), podemos ver cómo tiene forma de árbol, donde el primer luchador es Conor McGregor y se expande con los luchadores que han peleado con él, de manera consecutiva, hasta alcanzar una altura de cuatro niveles.

-GUIA DE EJECUCIÓN:

Para poder ejecutar nuestra base de datos necesitamos tener el contenedor de PostgreSQL activo, conectado a la red [network-edge-gria](#), y con nuestra base de datos relacional cargada.

Dentro de la carpeta del proyecto nosql/neo4j hay que ejecutar el [build.sh](#) cambiando la ruta absoluta al volumen por la de su sistema operativo.

A continuación hay que entrar en el contenedor neo4j en modo interactivo con la consola ([docker exec -it edge-gria-neo4j bash](#)).

Dentro de la terminal del contenedor ponemos [cypher-shell](#) para llamar a la consola de neo4j. Nos logueamos con nuestro usuario : [neo4j](#) y nuestra contraseña : [pwalumnobd](#).

Ahora podemos ejecutar el script de importación de los datos con:

```
neo4j@neo4j> :source /home/alumnobd/host-temp/scripts/import.cql
```

Para ejecutar todas las consultas:

```
neo4j@neo4j> :source /home/alumnobd/host-temp/scripts/consultas.cql
```

-CONCLUSIONES:

Como conclusión, la migración de la base de datos a Neo4j permitió modelar explícitamente las relaciones entre luchadores, peleas y eventos, facilitando la exploración de conexiones complejas y recorridos de múltiples niveles de manera natural. Las consultas que antes requerían múltiples JOIN en SQL se simplifican usando patrones de nodos y relaciones, mejorando la legibilidad y el rendimiento en análisis de grafos. Además, Neo4j ofrece una visualización intuitiva del grafo, lo que permite comprender rápidamente las interacciones entre los datos y realizar análisis de redes que serían complicados de ejecutar en un modelo relacional.

FIN :)