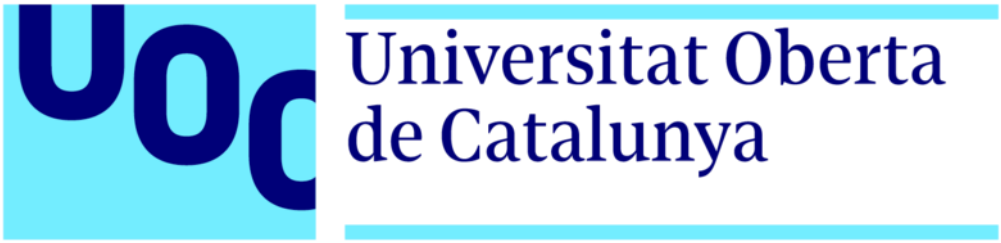

**Máster universitario de Ciencia de Datos**

# Prueba de Evaluación Continua – PEC2

## Arquitecturas de bases de datos no tradicionales – Distribución de datos y *Map Reduce*.


Autor:

Mario Ubierna San Mamés

# Índice de Contenido

# 1. Enunciado

## 1.1. Ejercicio 1

*En una organización se almacenan y gestionan series temporales (datos indexados en orden temporal) del tipo {timestamp, métrica, valor, origen, tags}. Se generan miles de datos por minuto, al final del día hay millones de registros. El tipo de datos generado, es decir los atributos generados y su formato es uniforme y no se esperan cambios a corto plazo. Se supone que los datos son consultados de forma concurrente por multitud de usuarios. Los datos no requieren un almacenamiento indefinido, ya que se utilizan para estudiar el estado de los sistemas.*

### 1.1.1. Modelo relacional o modelo NoSQL.

Vemos que el problema busca almacenar millones de registros diarios y hay un gran número de usuarios/operaciones de lectura de forma concurrente, a priori son características de un modelo NoSQL, pero la estructura de los datos está definida y no va a variar a corto plazo, característica más adecuada para el modelo relacional. En resumen, considero ambos modelos pueden servir pero creo que lo mejor es un modelo relacional, ya que éste también puede almacenar millones de registros y permitir la concurrencia de usuarios/operaciones de forma óptima manteniendo una estructura fija de los datos.

En cuanto a la arquitectura de distribución elegida es *peer-to-peer*, ya que con el enunciado no queda claro cuántos clientes/servidores vamos a tener y dónde se encuentran éstos, por lo que no podemos definir un modelo cliente/servidor o tres capas. En *peer-to-peer* todos los nodos son cliente/servidor por lo que nos evitamos el problema mencionado antes, también se generan millones de registros diarios y hay muchos usuarios, por lo que este tipo de arquitectura se comporta mejor, ya que hay muchísimos nodos en la red y las operaciones se ejecutan a lo largo de toda la red no en un servidor o conjuntos de servidores limitados.

### 1.1.2. Ventajas/inconvenientes respecto a la fragmentación

Puesto que no hay un gran número de atributos (5) considero que lo mejor es la fragmentación horizontal, teniendo como ventaja que las consultas de los propios

usuarios nos indiquen qué fragmentos se necesitan crear, también tiene como ventaja respecto a la fragmentación vertical que no necesita crear una relación entre registros de un mismo fragmento, mejorando así inserción de datos, ni hacer un estudio de la afinidad entre atributos para el diseño de un fragmento. El principal inconveniente existente es que hay que definir muy bien el fragmento en sí, ya que si se realiza un consulta y dicha fragmentación no contiene toda la información necesaria hay que seguir buscando por la red para encontrar la información, esto supone coste computacional y temporal.

### 1.1.3. Ventajas/inconvenientes respecto a la replicación

La estrategia elegida para la replicación de datos es replicación P2P con quórums, esta estrategia se comporta mejor que *master-slave* ya que no genera cuellos de botella en la copia maestra (*master-slave* es ideal cuando hay más lecturas que escrituras, pero en nuestro caso es al contrario, es verdad que la escritura a priori se va a hacer solamente una vez ya que es una serie temporal, pero según el enunciado se busca "gestionar" las series temporales, por lo que supongo que se pueden hacer *updates* sobre los registros). Dentro de la replicación P2P hay diferentes políticas, se ha elegido quórums porque tiene como ventaja un mejor rendimiento que la política síncrona (no tiene que escribir de forma atómica todo) y una mejor consistencia que la política asíncrona. Sin embargo, la principal desventaja de esta estrategia es que tenemos que definir el número de réplicas escritas de forma atómica y el número de réplicas recuperadas para la lectura, identificar los parámetros óptimos para esta estrategia no es fácil.

### 1.1.4. Modelo transaccional adecuado

Puesto que en la primera parte del ejercicio se ha definido que el modelo que mejor se ajusta es un modelo relacional, entonces el modelo transaccional adecuado sería el modelo ACID. Éste nos garantiza la consistencia de los datos, que a partir de un sistema distribuido también obtenemos la alta disponibilidad de los mismos. El modelo ACID incluye protocolos para la gestión de las réplicas y de la concurrencia que hemos visto que pueden existir según los requisitos de la aplicación descrita.

## 1.2. Ejercicio 2

*A partir de la lectura del artículo 'Consistency Models of NoSQL Databases' y de los apuntes indica si te parecen ciertas o falsas las siguientes afirmaciones.*

*Para cada una de las afirmaciones indica si es cierta o falsa, justificando la respuesta mediante lo que has leído en el artículo. En cada justificación deberá indicar el párrafo del artículo en la que se sustenta tu argumentación.*

*No serán válidas las respuestas que no se justifiquen.*

### 1.2.1. Afirmación 1

*Según el teorema CAP se puede afirmar que si una base de datos es CA implica que los datos son consistentes entre todos los nodos (mientras los nodos estén en línea) y que se puede leer/escribir de forma consistente en cualquier nodo puesto que los datos serán los mismos.*

Esta afirmación es verdadera, según el PDF "*B3_T7_BDD_BASE.pdf*" en la página 10 se hace la explicación del teorema CAP, el cual dice que en un sistema distribuido es imposible garantizar tanto la consistencia, la disponibilidad como la tolerancia a particiones al mismo tiempo, por lo tanto, si la base de datos es CA (consistencia y disponibilidad) los datos son consistentes entre todos los nodos siempre y cuando no haya fallos en el sistema (todos los nodos estén en línea y funcionando), es decir, si hubiera algún fallo (los nodos ya no están en línea) esta afirmación sería falsa.

### 1.2.2. Afirmación 2

*Las bases de datos orientadas a documentos sólo admiten replicación, resultando imposible las técnicas de distribución como sharding.*

Esta afirmación es falsa, según el PDF "*B3_T5_3_BDD_Disenyo.pdf*" en la página 13 párrafo segundo, indica que las bases de datos basadas en modelos de agregación (clave-valor, documentos…) permiten la replicación y promueven la fragmentación horizontal, la cual es comúnmente conocida como *sharding*.

### 1.2.3. Afirmación 3

*La consistencia final en el tiempo establece que todas las réplicas llegarán a ser gradualmente consistentes si no hay actualizaciones.*

Esta afirmación es verdadera, según el PDF "*B3_T7_BDD_BASE.pdf*" en la página 19 segundo párrafo, indica que si durante un período de tiempo no hay cambios en los datos todas las réplicas convergen al mismo valor, pero si hay conflictos este tipo de consistencia permite que se puedan perder algunos datos.

### 1.2.4. Afirmación 4

*La consistencia fuerte garantiza que una operación de lectura para un dato obtendrá el valor de la última escritura, aunque alguna de estas réplicas pueda tener valores inconsistentes.*

Esta afirmación es verdadera, según el PDF "*B3_T7_BDD_BASE.pdf*" en la página 16 tercer párrafo, indica que puede darse el caso de que alguna réplica sea inconsistente pero para el usuario en la base de datos todas las réplicas contienen la misma información, ya que todas ellas acaban convergiendo a un mismo valor a lo largo del tiempo, es decir, garantiza que en la lectura siempre se recuperará como mínimo una réplica que contenga el valor de la última escritura aunque haya réplicas inconsistentes.

## 1.3. Ejercicio 3

*Ejecuta el archivo IntroMapReduce.ipynb que se entrega junto al enunciado de la PEC siguiendo las instrucciones descritas en el anexo I. A continuación, resuelve los dos supuestos que se proponen. No es necesario que codifiques en Python la solución, basta que expliques tu solución utilizando pseudocódigo, explicaciones textuales así como los datos proporcionados para los resultados.*

```
red_social = [('Alicia', 'Benito'), ('Benito', 'Alicia'),
              ('Carlos', 'Benito'), ('Benito', 'Carlos'),
              ('Daniela', 'Enrique'), ('Enrique', 'Francisco'),
              ('Francisco', 'Enrique'), ('Daniela', 'Benito')]
```

### 1.3.1. Supuesto 1

*Suponer los datos de una red social que consisten en un conjunto de pares del tipo (personaA, personaB) que representan una relación "sigue a" ('following') de forma que la personaA sigue a la personaB. Dado el siguiente conjunto de datos, describe el algoritmo MapReduce que calcula el número de seguidores que tiene cada persona.*

**MAP**

Esta operación tiene que procesar la lista "red social" que recibe como parámetro de entrada, es decir, le pasamos por parámetro una lista y devuelve una tupla del tipo clave valor, donde la clave va a ser la persona que es seguida mientras que el valor va a ser la persona que le sigue (al contrario de cómo está definida la tupla de entrada), si la entrada es (Alicia, Benito) la salida va a ser ((Benito), (Alicia)).

```
# PRIMER PASO: MAP
def map(lista):
    # Recorremos cada elemento de la lista y lo mapeamos
    for relation in lista:
        return(((relation[1]), (relation[0])))
```

Tanto en este supuesto como en el siguiente el return no significa que directamente devuelva el valor, siendo Python se podría hacer uso de un yield, pero para que quedara más claro el pseudocódigo se hace uso del return como significado de emitir la tupla correspondiente.

El resultado de la anterior operación es:

```
(('Benito'), ('Alicia')),
(('Alicia'), ('Benito')),
(('Benito'), ('Carlos')),
(('Carlos'), ('Benito')),
(('Enrique'), ('Daniela')),
(('Francisco'), ('Enrique')),
(('Enrique'), ('Francisco')),
(('Benito'), ('Daniela')),
(('Benito'), ('Alicia')),
```

**SHUFFLE**

Antes de ejecutar el reduce tenemos que agrupar los valores del resultado del mapeo según la clave, éste es el segundo paso el cual nos dejaría el siguiente resultado:

```
(('Benito'), [('Alicia'), ('Carlos'), ('Daniela')]),
(('Alicia'), [('Benito')]),
(('Carlos'), [('Benito')]),
(('Enrique'), [('Daniela'), ('Francisco')]),
(('Francisco'), [('Enrique')]),
(('Daniela'), []),
```

Para conseguir este resultado shuffle tiene que recorrer el mapeo del paso anterior y agrupar los valores creando un tupla del tipo clave valores, donde clave es la clave del mapeo y valores es una lista con las personas que le siguen a cada clave.

**REDUCE**

Ésta sería la última operación, y se encarga de traducir la clave y la lista de valores a una clave valor, donde la clave sería la persona en cuestión y el valor el número de seguidores:

```
# TERCER PASO: REDUCE
def reduce(k, listV):
    # Calculamos el número de seguidores que tiene cada persona
    return((k, len(listV)))
```

El resultado que nos devolvería sería:

```
(('Benito'), 3),
(('Alicia'), 1),
(('Carlos'), 1),
(('Enrique'), 2),
(('Francisco'), 1),
(('Daniela'), 0)
```

### 1.3.2. Supuesto 2

*La relación "sigue a" no es simétrica, ya que una persona no tiene porqué seguir a sus seguidores. No obstante, a veces ocurre que una persona sigue a la persona que lo sigue. Es decir, si PersonaA sigue a PersonaB, entonces PersonaB sigue a PersonaA. Con este ejercicio queremos identificar los pares (PersonaA, PersonaB) que no tienen una relación (PersonaB, PersonaA) definida. Con los mismos datos anteriores, describe el algoritmo MapReduce que permite obtener la lista de las relaciones que cumplen dicha condición.*

**MAP**

Esta función tiene que procesar la lista que recibe como parámetro de entrada, y para cada tupla de la lista nos devuelve una tupla del tipo clave valor, donde la clave es la relación y el valor es 1, es decir, ((Alicia, Benito), 1), el uno representa que hay una relación unidireccional entre Alicia y Benito:

```
# PRIMER PASO: MAP
def map(lista):
    # Recorremos cada elemento de la lista y lo mapeamos
    for relation in lista:
        return(((relation[0], relation[1]), 1))
```

Al igual que sucedía antes el return hay que tomárselo como si se emite la tupla correspondiente.

El resultado de esta operación es el siguiente:

```
(('Alicia', 'Benito'), 1),
(('Benito', 'Alicia'), 1),
(('Carlos', 'Benito'), 1),
(('Benito', 'Carlos'), 1),
(('Daniela', 'Enrique'), 1),
(('Enrique', 'Francisco'), 1),
(('Francisco', 'Enrique'), 1),
(('Daniela', 'Benito'), 1)
```

**SHUFFLE**

Al igual que sucedía en el ejercicio anterior hay que agrupar las claves, con la pequeña diferencia de que en este caso las claves no son iguales a priori, es decir, hay que agrupar Alicia y Benito pero hay dos claves que significan lo mismo, (Alicia, Benito) y (Benito, Alicia). Por lo tanto, en este paso hay que agrupar los valores pero para cada clave hay que buscar la inversa y si la encuentra entonces se agrupa la información, es decir, en el supuesto anterior se agrupaba si k[0] == k_actual[0] y k[1] == k_actual[1], sin

embargo en este caso k[0] == k_actual[1] y k[1] == k_actual[0]. De esta forma esta operación tan básica agruparía por la misma clave, dando lugar al siguiente resultado:

```
(('Alicia', 'Benito'), [1, 1]),
(('Carlos', 'Benito'), [1, 1]),
(('Daniela', 'Enrique'), [1]),
(('Enrique', 'Francisco'), [1, 1]),
(('Daniela', 'Benito'), [1])
```

**REDUCE**

Finalmente, tenemos que ejecutar la operación reduce, para cada clave hay que comprobar si el tamaño de la lista de los valores es menor o igual a 1, de ser así estamos en un caso en el que la relación es unidireccional o directamente no hay relación.

El pseudocódigo sería el siguiente:

```
# TERCER PASO: REDUCE
def reduce(k, listV):
    # Calculamos el número de la lista de valores, si es menor o igual a 1 devolvemos la clave
    num_listaV = len(listV)

    if num_listaV <= 1:
        return(k)
```

Como resultado nos proporcionaría:

```
('Daniela', 'Enrique'),
('Daniela', 'Benito')
```

## 1.4. Ejercicio 4

*Suponiendo que debemos configurar el sistema de replicación de una base de datos para que cumpla la condición de consistencia fuerte y se nos plantean distintas alternativas. Sabemos que se reciben muchas solicitudes de lectura y relativamente pocas de escritura. ¿Cuál de las siguientes alternativas de quórum elegirías y por qué?*

- *N=7, R=3, W=1*
- *N=7, R=1, W=7*
- *N=7, R=5, W=4*
- *N=7, R=4, W=5*

Lo primero de todo es comprobar que se cumplen las inecuaciones de la consistencia fuerte, es decir:

- W > N/2
- W + R > N

Dadas las siguientes inecuaciones vemos que todas las opciones las cumplen menos la primera, por lo que esta opción queda descartada.

La alternativa quórum que elegiría sería la segunda opción (*N=7, R=1, W=7*), cuando N es igual a W estamos en un caso de política síncrona, es decir, tenemos 7 réplicas y cada vez que hay una operación de escritura se escribe en las 7 réplicas de forma atómica (hasta que no haya escrito en todas las réplicas no continúa la ejecución), esto lo que significa es que cada vez que se escribe un dato al ser igual N y W todas las réplicas van a contener la misma información en todo momento, por lo tanto solo es necesaria una R igual a uno (solo se necesita recuperar una réplica para resolver la lectura). En resumen, si estamos en una situación que se reciben muchas lecturas y muy pocas escrituras, cuanto menor sea R (número de réplicas recuperadas para resolver la lectura) mucho más eficiente va a ser nuestro sistema de replicación.

Por otro lado, tanto la opción tercera como la cuarta también sirven, pero son menos eficientes ya que tienen que recuperar más réplicas para resolver las operaciones de lectura.