



# Conceptos y Aplicaciones de Big Data

Operaciones clásicas con MapReduce

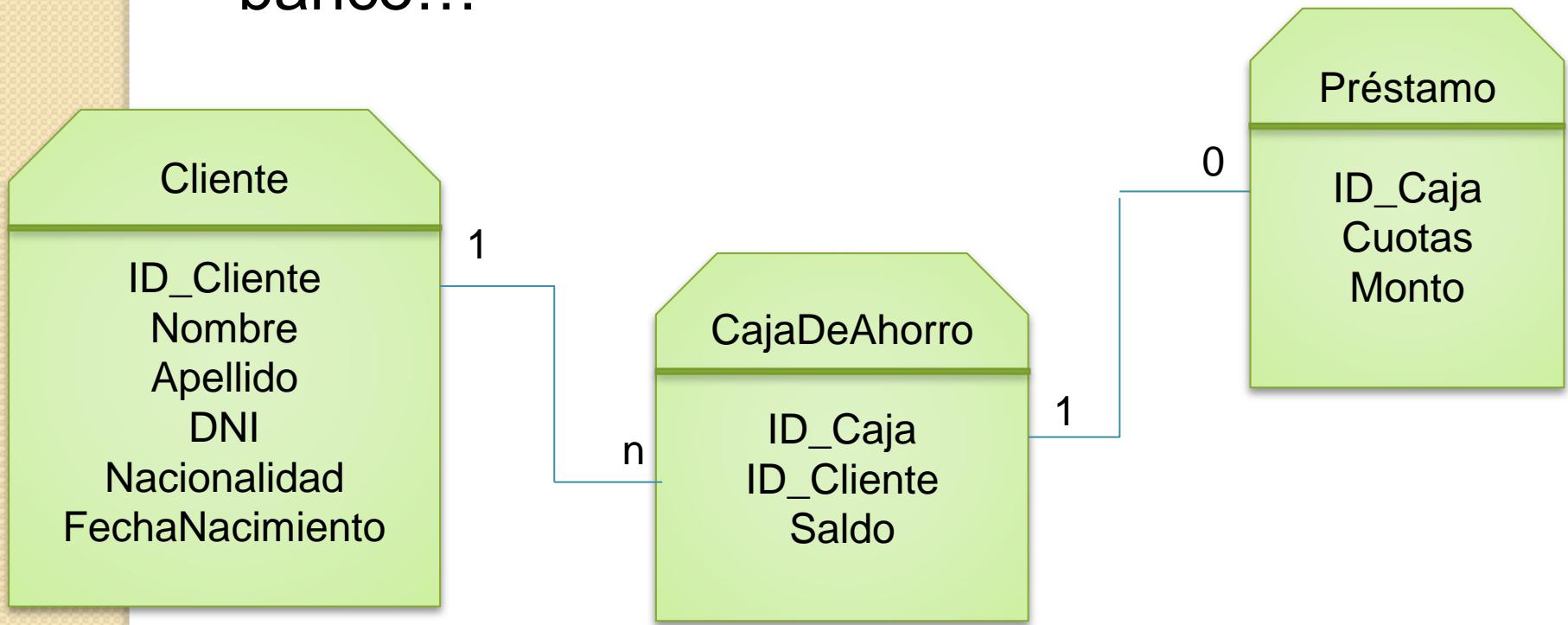
Prof. Waldo Hasperué  
[\(wasperue@lidi.info.unlp.edu.ar\)](mailto:wasperue@lidi.info.unlp.edu.ar)

# Temario

- Operaciones clásicas con MapReduce
  - Filtros
  - Resúmenes
  - Transformaciones
  - Joins

# Problema

Dada la siguiente base de datos de un banco...



... si la pensamos en términos de Big Data ...

# Problema

¿Cómo sería un job que permita hacer esta consulta?

```
SELECT Sum(Saldo)  
FROM CajaDeAhorro  
GROUP BY ID_Cliente
```

# Solución – Job group by

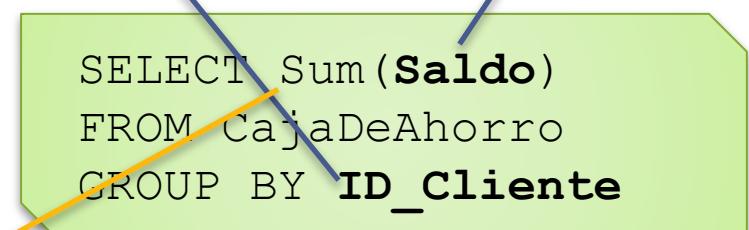
```
map(key, value):  
    write(value["ID_Cliente"], value["Saldo"] )  
  
reduce(key, values):  
    acum = 0  
    for v in values  
        acum = acum + v  
    write(key, acum)
```

```
SELECT Sum(Saldo)  
FROM CajaDeAhorro  
GROUP BY ID_Cliente
```

# Solución – Job group by

```
map(key, value):  
    write(value["ID_Cliente"], value["Saldo"])  
  
reduce(key, values):  
    acum = 0  
    for v in values  
        acum = acum + v  
    write(key, acum)
```

```
SELECT Sum(Saldo)  
FROM CajaDeAhorro  
GROUP BY ID_Cliente
```



# Problema

¿Cómo sería un job que permita hacer esta consulta?

```
SELECT *
FROM Prestamo
WHERE Cuotas = 36
```

# Solución – Job filter

```
map(key, value) :  
    if (value["Cuotas"] == 36) :  
        write(value["ID_Caja"],  
              value["Cuotas"] + value["Monto"])
```

```
SELECT *  
FROM Prestamo  
WHERE Cuotas = 36
```

```
reduce(key, values) :  
    write(key, values[0])
```

# Solución – Job filter

```
map(key, value) :  
    if (value["Cuotas"] == 36) :  
        write(value["ID_Caja"],  
              value["Cuotas"] + value["Monto"])
```

```
SELECT *  
FROM Prestamo  
WHERE Cuotas = 36
```

```
reduce(key, values) :  
    write(key, values[0])
```

# Problema

¿Cómo sería un job que permita hacer esta consulta?

```
SELECT Apellido, Nombre  
FROM Cliente
```

# Solución – Job projection

```
map(key, value):  
    write(value["Apellido"] + value["Nombre"],  
          None)  
  
reduce(key, values):  
    for v in values:  
        write(key, None)
```

```
SELECT Apellido, Nombre  
FROM Cliente
```



# Solución – Job projection

```
map(key, value) :
```

```
    write(value["Apellido"] + value["Nombre"],  
          None)
```

```
reduce(key, values) :
```

```
    for v in values
```

```
        write(key, None)
```

SELECT **Apellido, Nombre**  
FROM Cliente

Puede haber más de una misma  
tupla "Apellido y nombre"

# Problema

¿Cómo sería un job que permita hacer esta consulta?

```
SELECT DISTINCT Apellido,  
    Nombre  
FROM Cliente
```

# Solución – Job distinct

```
map(key, value) :
```

```
    write(value["Apellido"] + value["Nombre"],  
          None)
```

```
reduce(key, values) :
```

```
    write(key, None)
```

```
SELECT DISTINCT  
    Apellido, Nombre  
FROM Cliente
```

Solo debemos escribir una tupla

# Problema

¿Cómo sería un job que permita hacer esta consulta?

```
SELECT *
FROM CajaDeAhorro AS CA
    INNER JOIN Prestamo AS P
        ON CA.ID_CAJA = P.ID_Caja
```

# Problema

Se deben contemplar varias situaciones

- Fuente de los datos
  - 1) Usando un único map
  - 2) Usando más de un map
- Tipos de join:
  - 1) El join es 1 a 1
  - 2) El join es 1 a n

# Solución – Job join

```
map(key, value) :
```

```
    write(value["ID_Caja"], value)
```

```
SELECT *
FROM CajaDeAhorro AS CA
    INNER JOIN Prestamo AS P
        ON CA.ID_CAJA = P.ID_Caja
```

```
reduce(key, values) :
```

```
    write(key, values[0] + values[1])
```

# Solución – Job

Usamos como clave el campo del join

```
map(key, value) :  
    write(value["ID_Caja"], value)
```

```
SELECT *  
FROM CajaDeAhorro AS CA  
    INNER JOIN Prestamo AS P  
    ON CA.ID_CAJA = P.ID_Caja
```

```
reduce(key, values) :  
    write(key, values[0] + values[1])
```

Concatenamos las dos tablas

# Solución – Job

¿Cómo sabemos si value es un préstamo o una caja de ahorros?

```
map(key, value) :
```

```
    write(value["ID_Caja"], value)
```

```
reduce(key, values) :
```

```
    write(key, values[0] + values[1])
```

# Solución – Job

Si existe forma de saberlo entonces con un map se puede resolver

```
map(key, value) :  
    if (value is CajaDeAhorro) :  
        write(value["ID_Caja"], value)  
    else:  
        write(value["ID_Caja"], value)
```

```
reduce(key, values) :  
    write(key, values[0] + values[1])
```

# Solución – Job

De lo contrario hay que tener tantos maps como "tablas" existan y a cada map se le asigna como entrada una tabla diferente (directorio en el HDFS)

```
mapCA(key, value):
```

```
    write(value["ID_Caja"], value)
```

```
mapP(key, value):
```

```
    write(value["ID_Caja"], value)
```

```
reduce(key, values):
```

```
    write(key, values[0] + values[1])
```

# Solución – Job join

```
mapCA(key, value):
```

```
    write(value["ID_Caja"], value)
```

```
mapP(key, value):
```

```
    write(value["ID_Caja"], value)
```

```
reduce(key, values):
```

```
    write(key, values[0] + values[1])
```

¿MapReduce da garantía que  
value[0] sea el préstamo y  
value[1] la caja de ahorro, o  
viceversa?

# Solución – Job

ID_C	Clien	Sal	Cuot	Mont
ID_C	Clien	Sal	Cuot	Mont
ID_C	Cuot	Mont	Clien	Sal
ID_C	Clien	Sal	Cuot	Mont
ID_C	Cuot	Mont	Clien	Sal

```
mapCA(key, value) :
```

```
    write(value["ID_Caja"], value)
```

Possible salida  
del reduce

```
mapP(key, value) :
```

```
    write(value["ID_Caja"], value)
```

```
reduce(key, values) :
```

```
    write(key, values[0] + values[1])
```

¿MapReduce da garantía que  
value[0] sea el préstamo y  
value[1] la caja de ahorro, o  
viceversa?

# Solución – Job

Agregamos un identificador al valor, para poder distinguir que representa

```
mapCA(key, value):  
    write(value["ID_Caja"], ("CA", value))
```

```
mapP(key, value):  
    write(value["ID_Caja"], ("P", value))
```

```
reduce(key, values):  
    for v in values:  
        if (v[0] = "CA"):  
            caja_de_ahorro = v[1]  
        else:  
            prestamo = v[1]  
    write(key, caja_de_ahorro + prestamo)
```

# Solución – Job join

```
mapCA(key, value):  
    write(value["ID_Caja"], ("CA", value))
```

```
mapP(key, value):  
    write(value["ID_Caja"], ("P", value))
```

```
reduce(key, values):  
    for v in values:  
        if (v[0] = "CA"):  
            caja_de_ahorro = v[1]  
        else:  
            prestamo = v[1]  
    write(key, caja_de_ahorro + prestamo)
```

Esto funciona en una relación 1 a 1.  
¿Qué sucede en nuestro ejemplo  
que es 0 a 1?

# Solución – Job join

```
mapCA(key, value):  
    write(value["ID_Caja"], "CA" + value)
```

```
mapP(key, value):  
    write(value["ID_Caja"], "P" + value)
```

```
reduce(key, values):  
    prestamo = None  
    for v in values:  
        if (v[0] = "CA")  
            caja_de_ahorro = v  
        else:  
            prestamo = v  
    if (prestamo != None):  
        write(key, caja_de_ahorro + prestamo)
```

Escribimos en la salida solo si la caja de ahorro tiene asociado un préstamo

# Problema

¿Y si el problema fuera 1 a n?

```
SELECT *
FROM CajaDeAhorro AS CA
    INNER JOIN Cliente AS C
        ON CA.ID_Cliente = C.ID_Cliente
```

# Solución – Job join

```
mapCA(key, value):  
    write(value["ID_Cliente"], ("CA", value))  
  
mapC(key, value):  
    write(value["ID_Cliente"], ("C", value))  
  
reduce(key, values):  
    for v in values:  
        if (v[0] = "C"):  
            cliente = v[1]  
        else:  
            caja_de_ahorro = v[1]  
            write(key, cliente +  
                  caja_de_ahorro)
```

Tenemos que escribir más de una "tupla" de salida, pero ¿sabemos en qué momento vamos a recibir al cliente?

# Solución – JCo

CA	ID	Clien	Sald		
CA	ID	Clien	Sald		
CA	ID	Clien	Sald		
C	ID	Nom	Ape	Naci	Fech
CA	ID	Clien	Sald		

```
mapCA(key, value):
```

```
    write(value["ID_Cliente"], CA + value)
```

```
mapC(key, value):
```

```
    write(value["ID_Cliente"], "C" + value)
```

```
reduce(key, values):
```

```
    for v in values:
```

```
        if (v[0] = "C"):
```

```
            cliente = v
```

```
    else:
```

```
        caja_de_ahorro = v
```

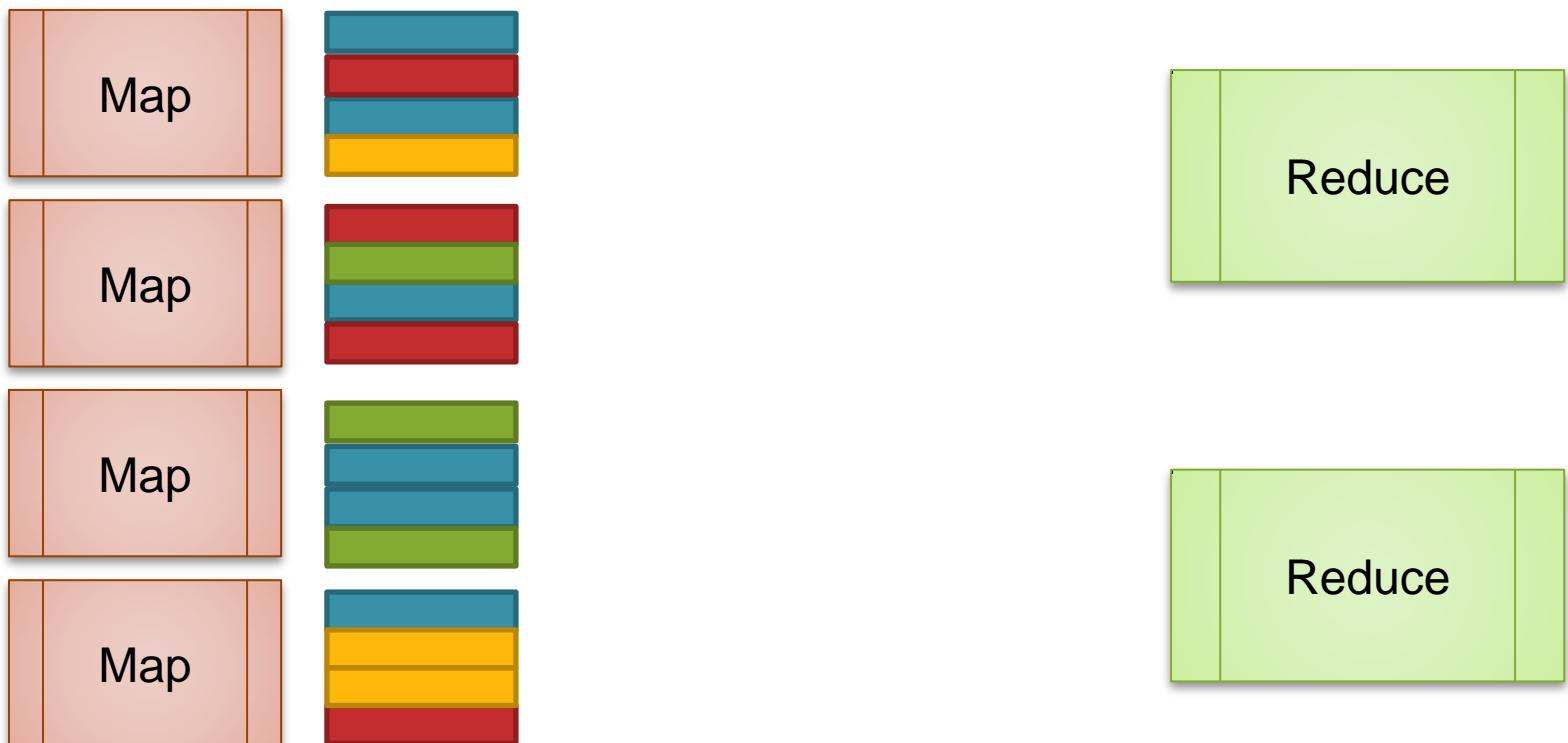
```
    write(key, cliente +  
          caja_de_ahorro)
```

Tenemos que escribir más de una "tupla" de salida, pero ¿sabemos en qué momento vamos a recibir al cliente?

Possible entrada como *values*

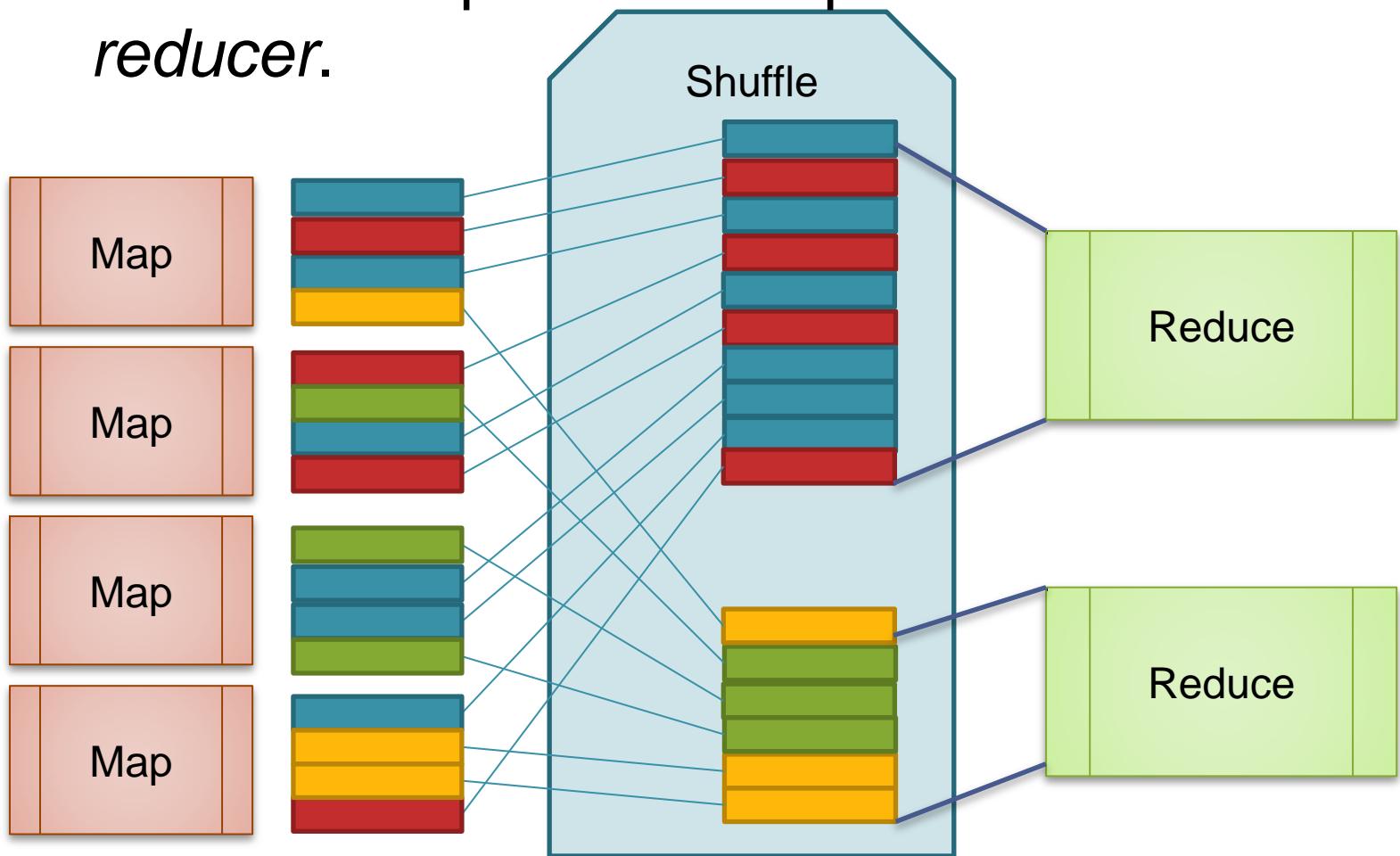
# Solución – Job join

"Meter mano" en las etapas intermedias de *shuffle* y *sort*.



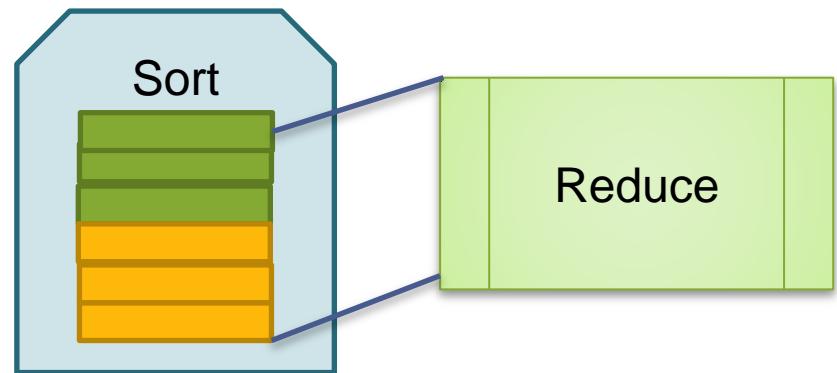
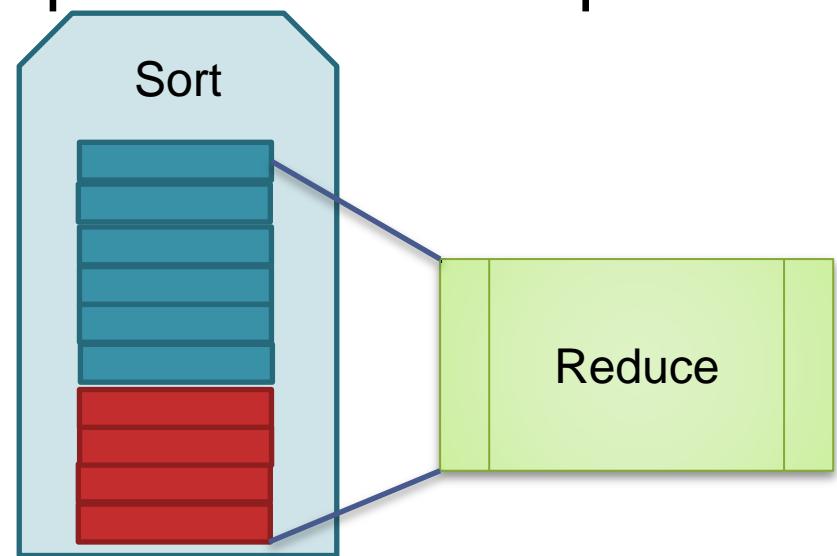
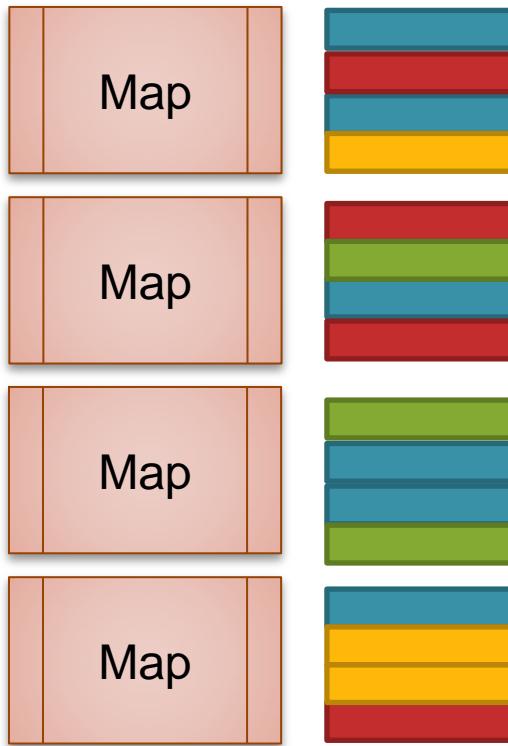
# Etapa shuffle

La etapa *shuffle* se encarga de que todas las claves sean procesadas por el mismo *reducer*.



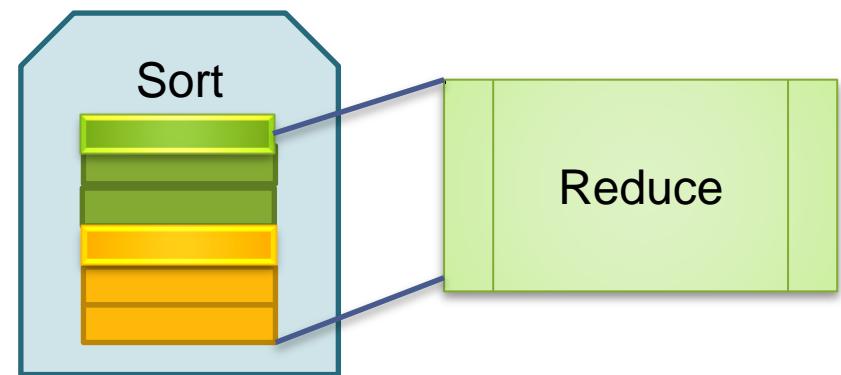
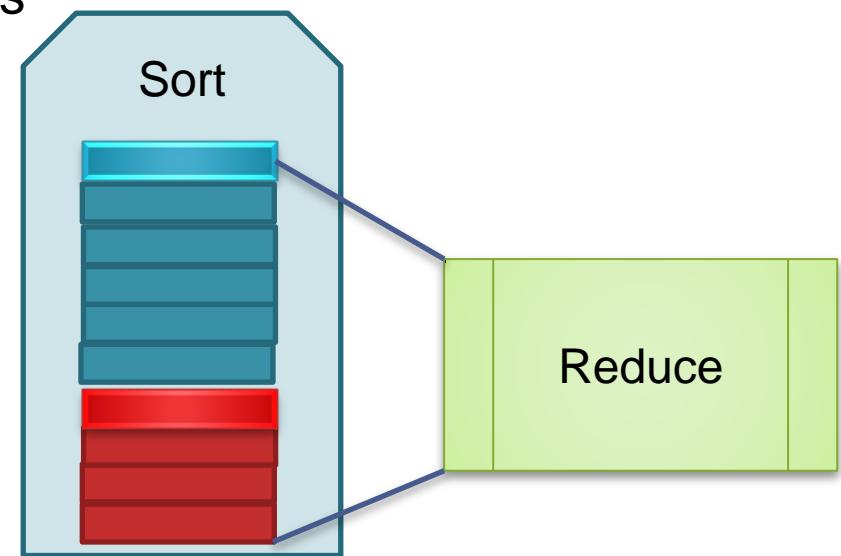
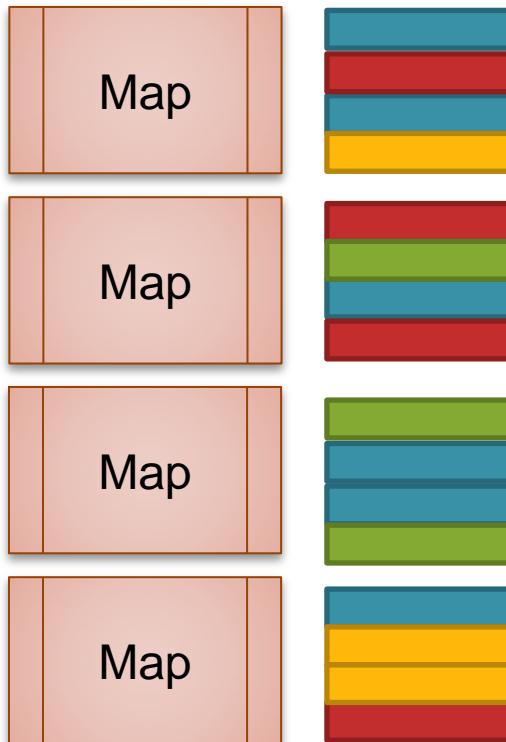
# Etapa sort

La etapa *sort* se encarga de que un mismo *reducer* reciba las tuplas ordenadas por clave.



# Personalización de comparadores de claves

Lo que buscamos es un mecanismo que nos permita agrupar por el mismo ID de cliente, pero al ordenar, que el cliente sea la primera tupla de los valores asociados.



# Comparadores de claves

- Necesitamos implementar:
  - Un comparador de claves para ser usado durante la etapa *shuffle*
  - Un comparador de claves para ser usado durante la etapa *sort*

# Clave personalizada

```
mapCA(key, value):
```

```
    write(("CA", value["ID_Cliente"]), value)
```

Generamos una clave que tenga el ID del cliente y la distinción entre caja de ahorro y cliente

```
mapC(key, value):
```

```
    write(("C", value["ID_Cliente"]), value)
```

```
reduce(key, values):
```

```
    cliente = values[0]
```

```
    for v in values[1..]:
```

```
        caja_de_ahorro = v
```

```
    write(key, cliente + caja_de_ahorro)
```

Con ello nos aseguramos de recibir primero al cliente y luego la lista de cajas de ahorro

# Comparador shuffle

```
def cmpShuffle(key1, key2):  
    if (key1[1] == key2[1]):  
        return 0  
    elif (key1[1] < key2[1]):  
        return -1  
    else:  
        return 1
```

```
compare(a, b)  
    0 → a==b  
    -1 → a<b  
    1 → a>b
```

El comparador, usado por la tarea de shuffle, es una función que compara dos claves. Para asegurarnos que los datos de un mismo ID de cliente (cliente o caja de ahorro) sea destinado al mismo reducer solo comparamos el ID del cliente.

# Comparador sort

```
def cmpSort(key1, key2):  
    if(key1[0] == key2[0]):  
        return 0  
    elif(key1[0] == "C"):  
        return -1  
    else:  
        return 1
```

```
compare(a, b)  
0 → a==b  
-1 → a<b  
1 → a>b
```

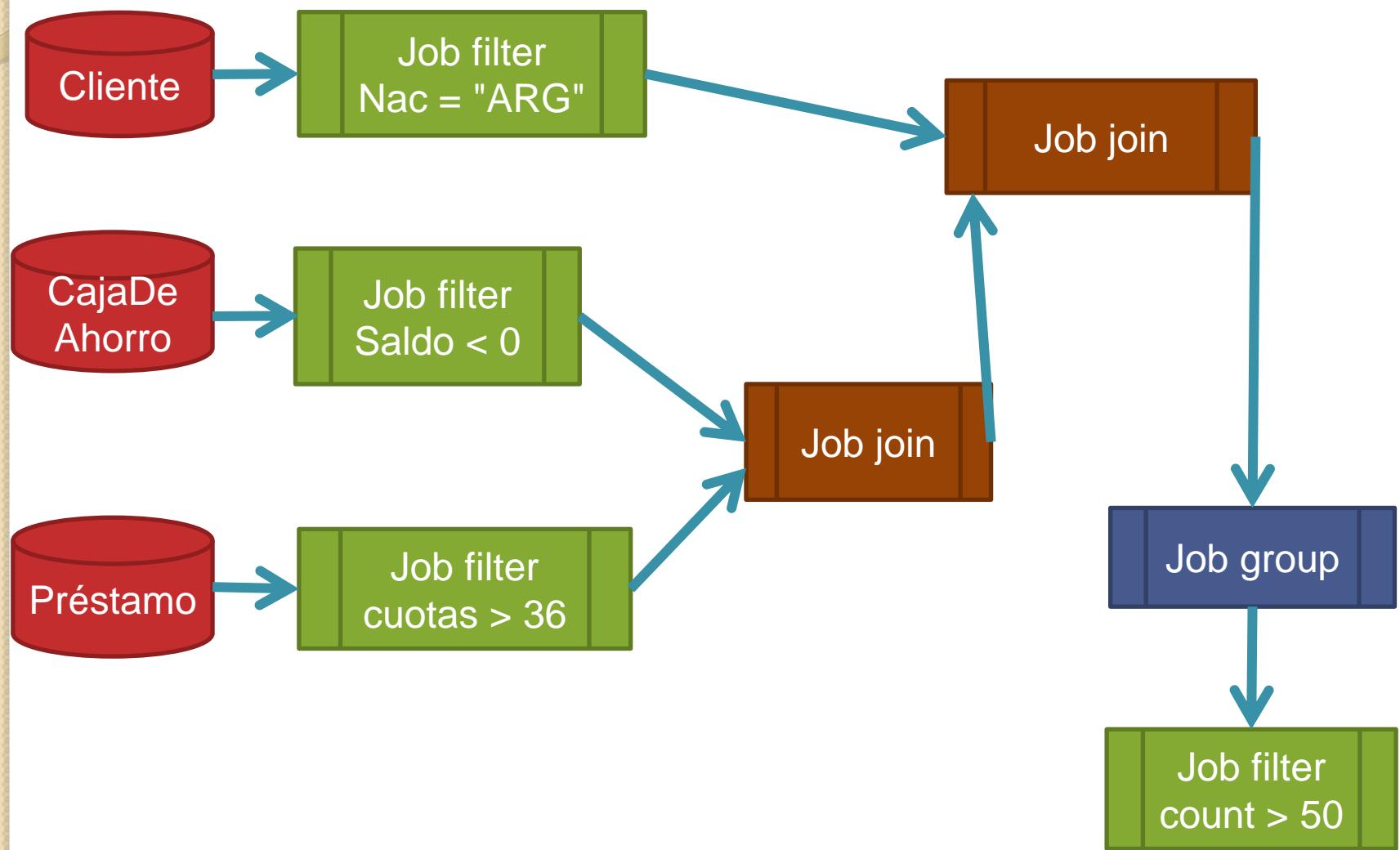
El comparador, usado por la tarea de sort, es una función que compara dos claves. Para asegurarnos que la tupla de un cliente sea el primer valor en recibir el *reducer* comparamos por el identificador personalizado de nuestra clave ("C" o "CA").

# Problema

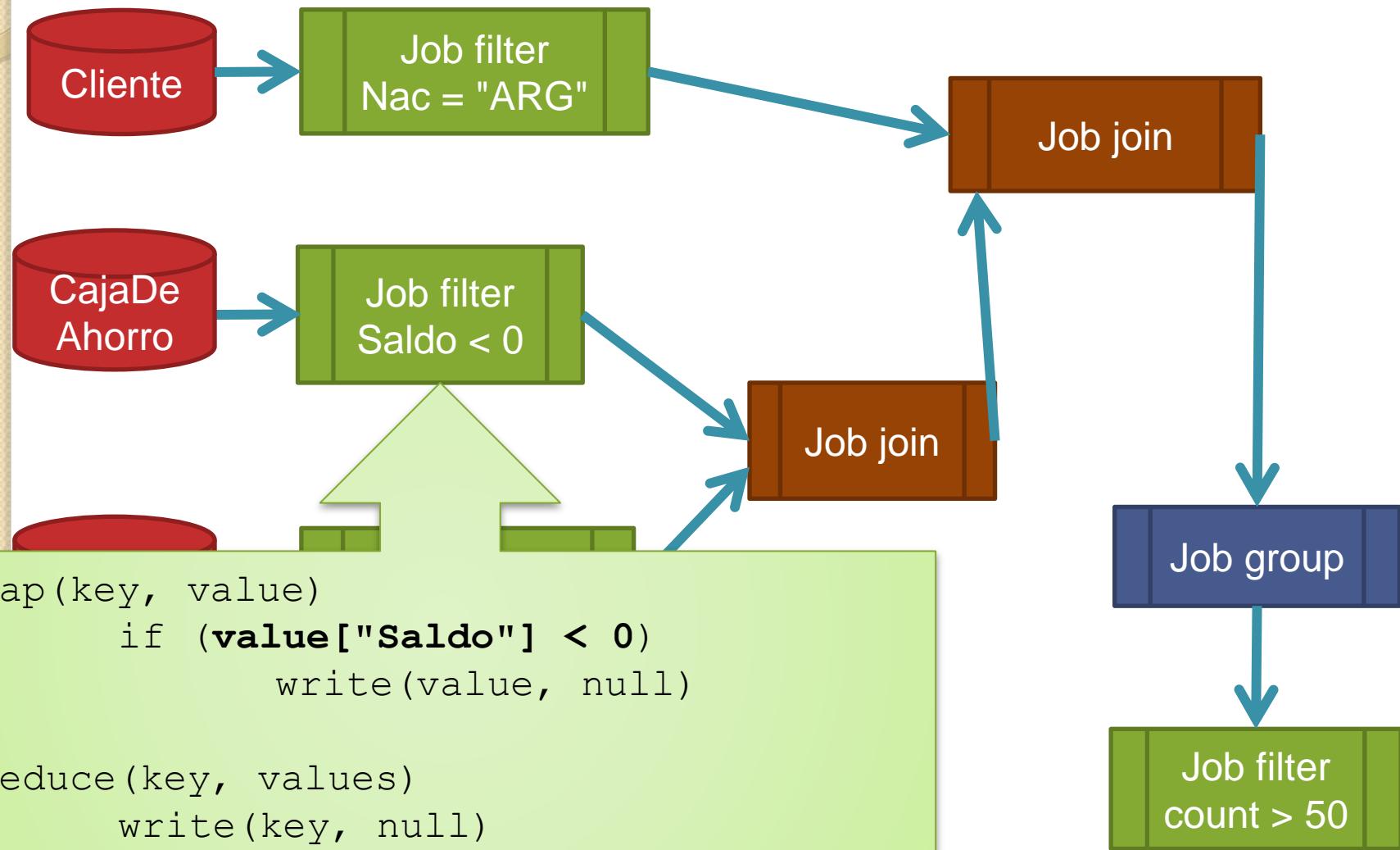
¿Cómo sería una solución MapReduce que permita hacer esta consulta?

```
SELECT count(Año(Nacimiento))  
FROM Ciente AS C INNER JOIN  
      CajaDeAhorro AS CA  
      ON C.ID_Cliente = CA.ID_Cliente  
WHERE Nacionalidad = "ARG"  
AND Saldo < 0  
AND CA.ID_Caja IN  
  ( SELECT ID_Caja  
    FROM Prestamo AS P INNER JOIN  
          CajaDeAhorro AS CA  
          ON P.ID_Caja = CA.ID_Caja  
    WHERE Cuotas > 36 )  
GROUP BY Año(Nacimiento)  
HAVING Count(Año(Nacimiento)) > 50
```

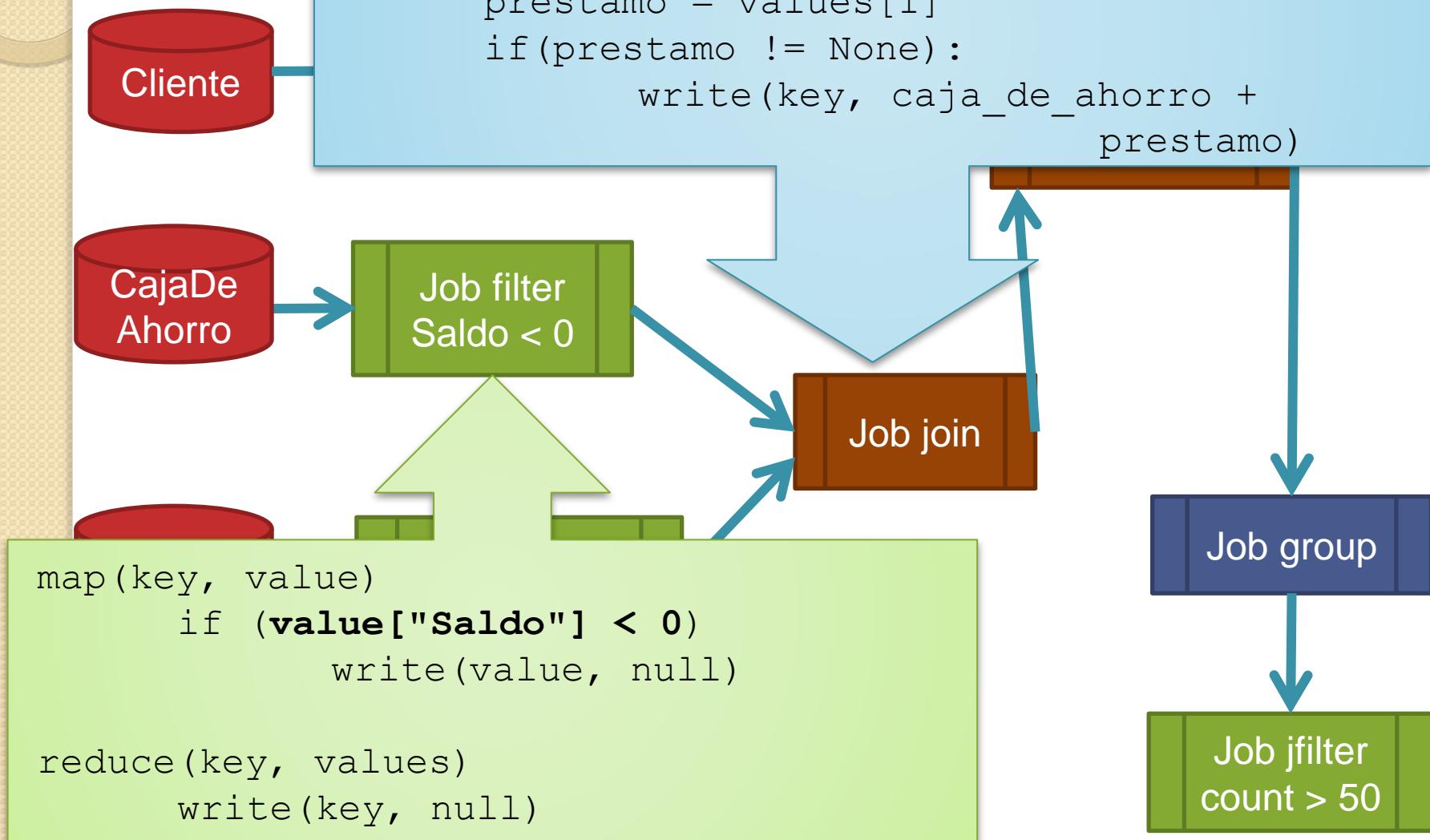
# Planteo de la solución



# Planteo de la solución



# Plano de trabajo

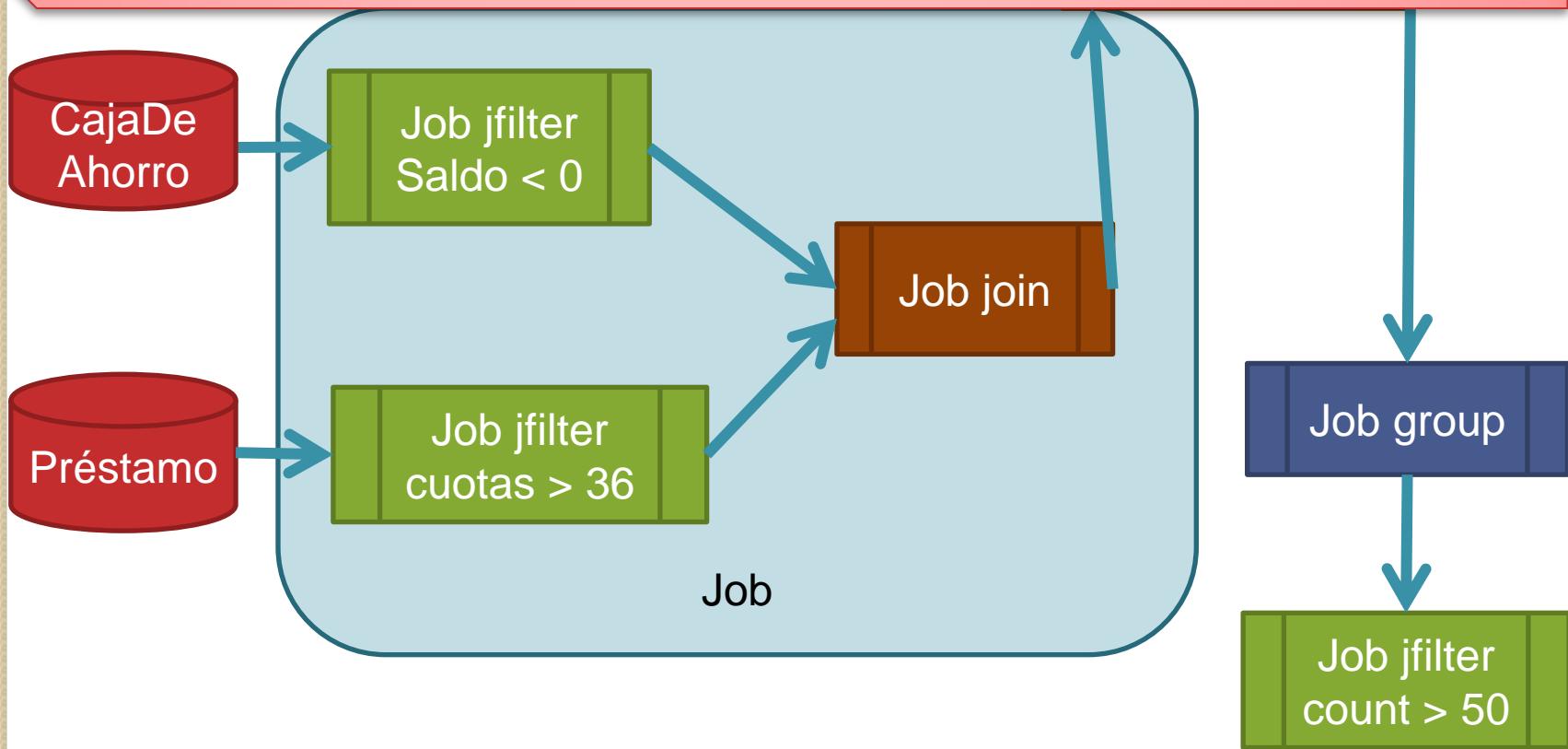


```

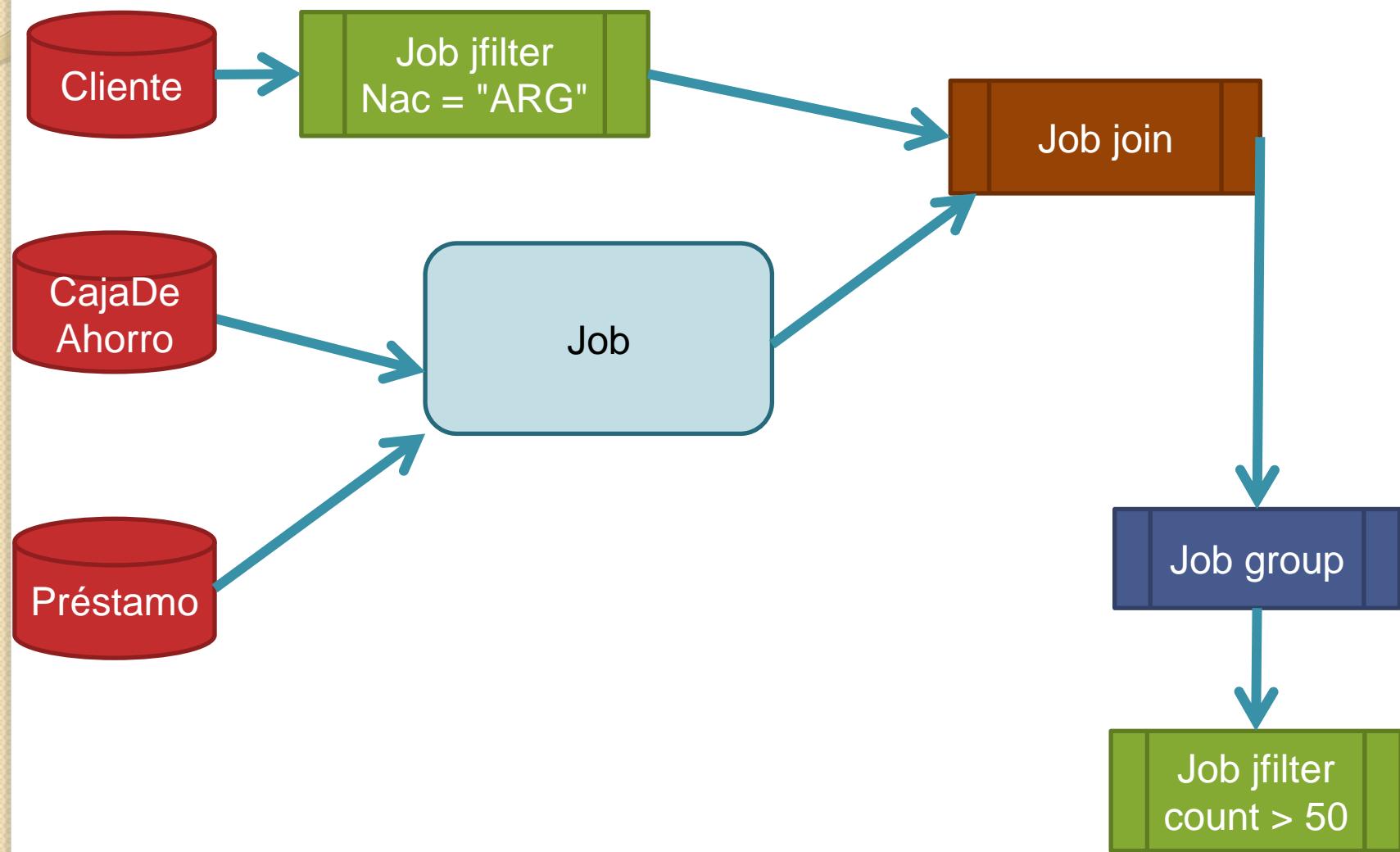
mapCA(key, value)
    if (value["Saldo"] < 0)
        write("CA" + value["ID_Caja"], value)

reduce(key, values)
    caja_de_ahorro = values[0]
    prestamo = null
    for v in values
        prestamo = v
    write(key, caja_de_ahorro + prestamo)

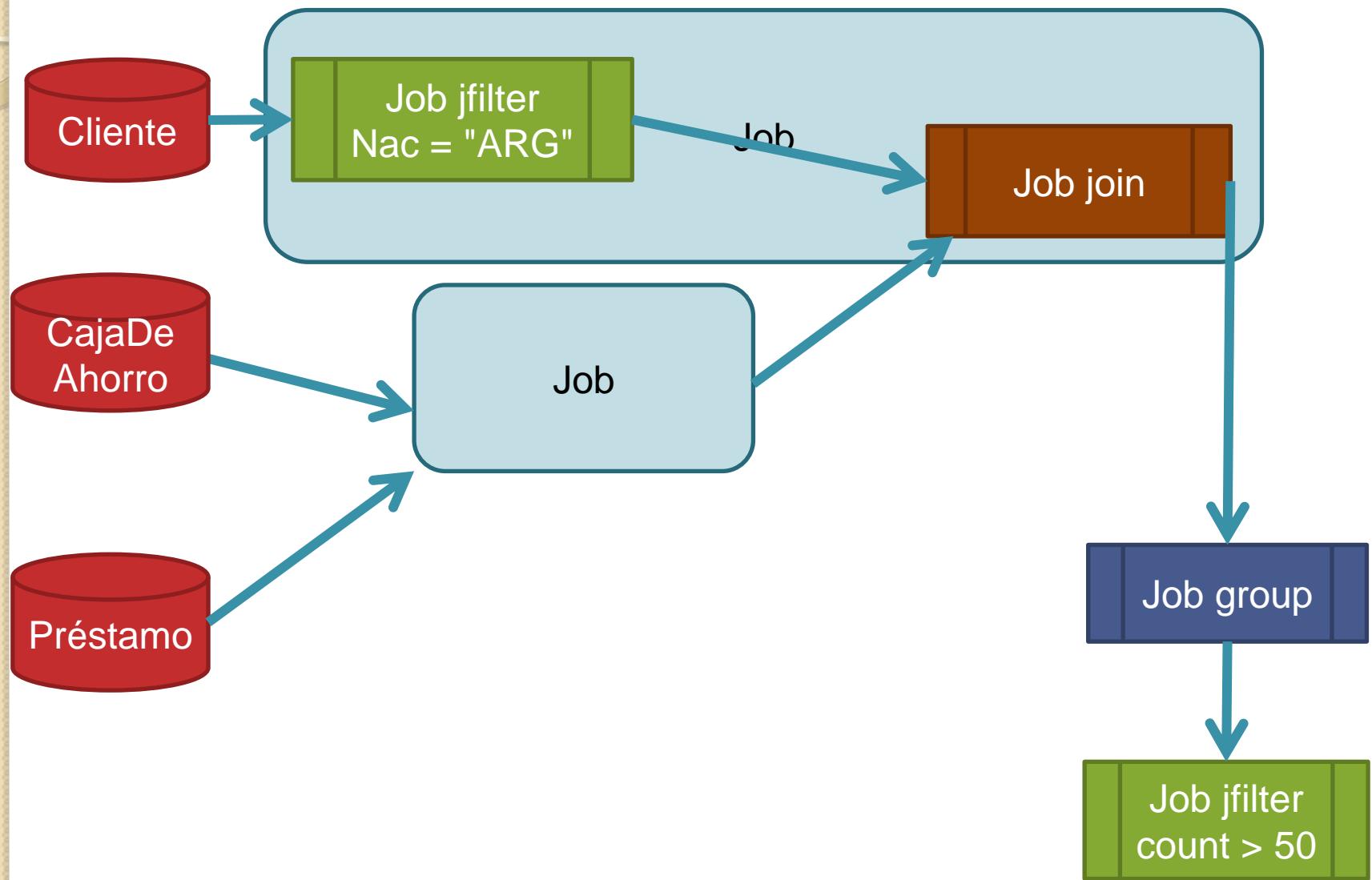
```



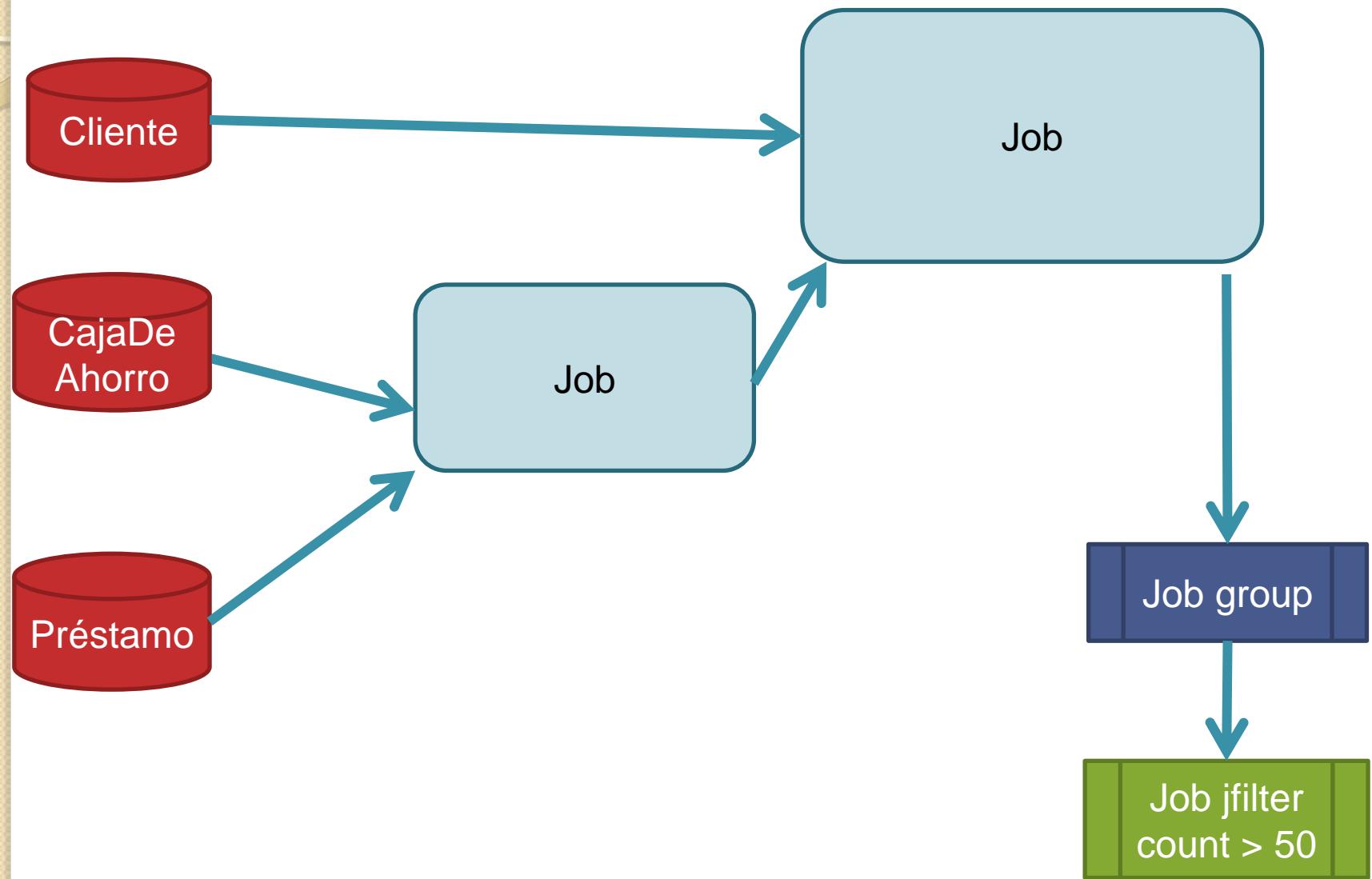
# Planteo de la solución



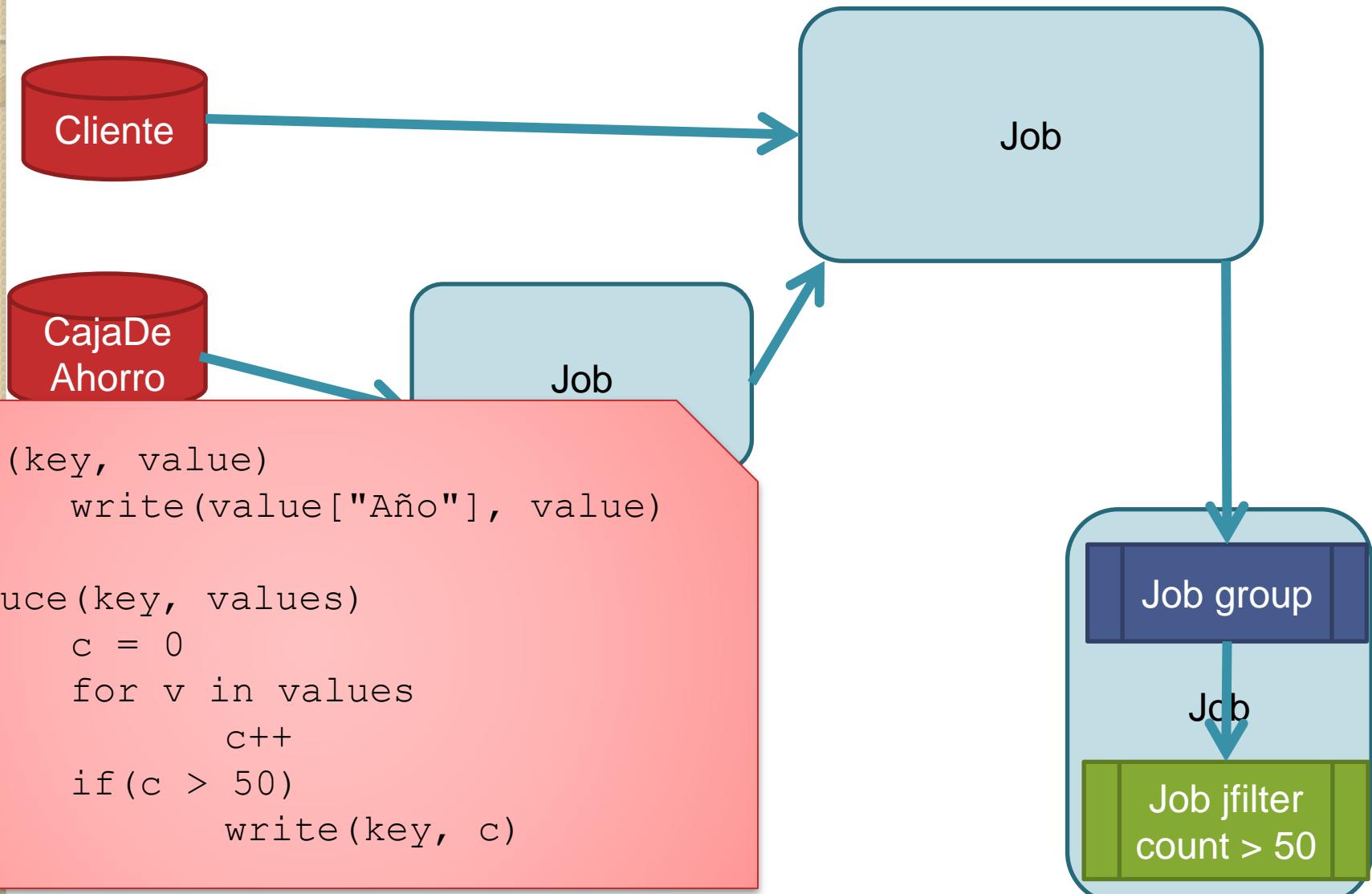
# Planteo de la solución



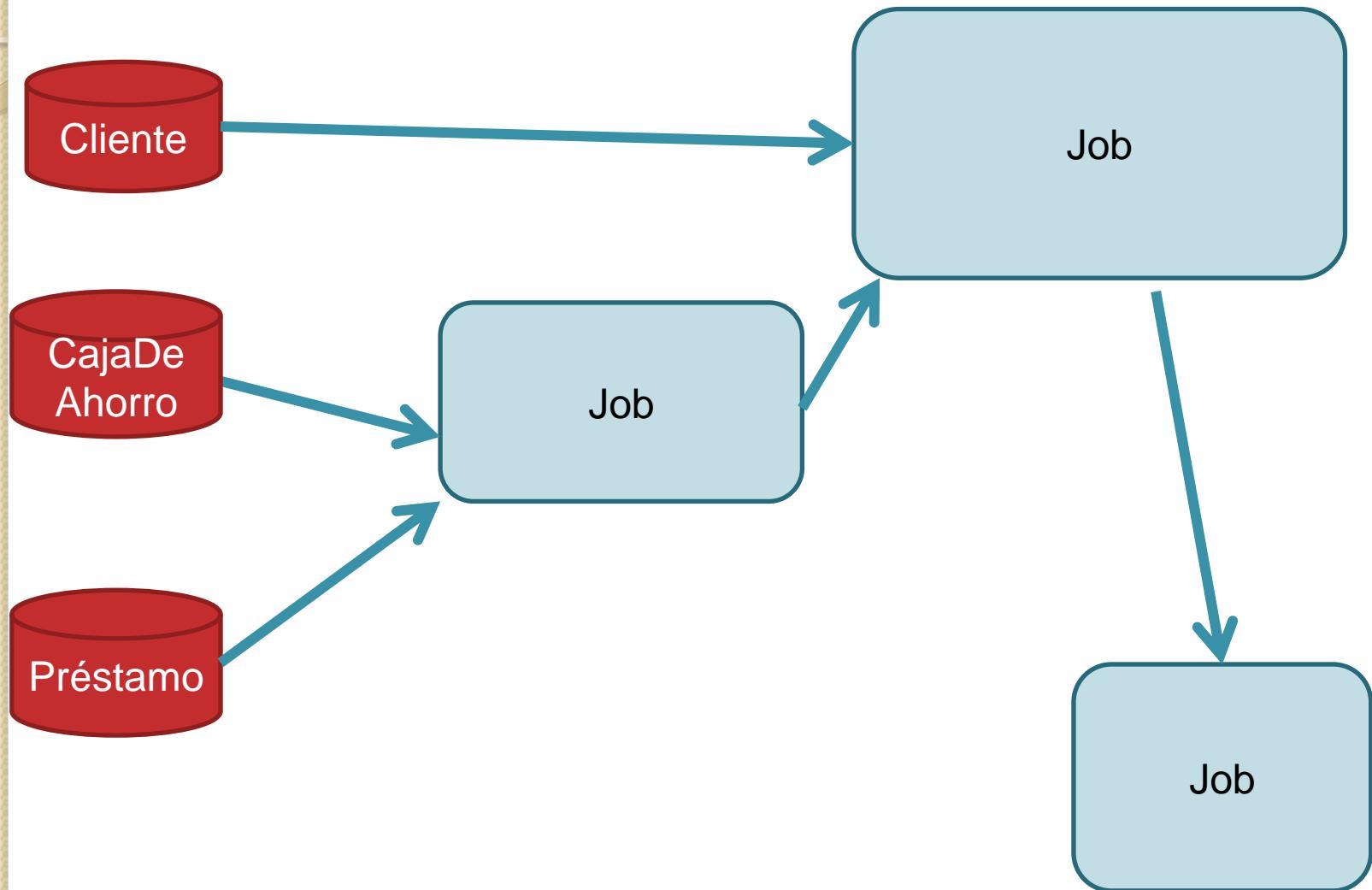
# Planteo de la solución



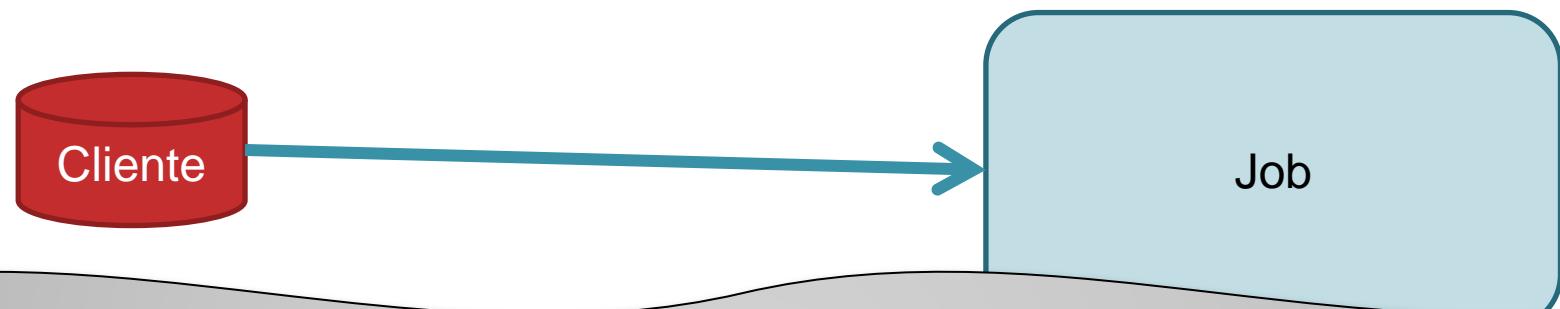
# Planteo de la solución



# Planteo de la solución



# Planteo de la solución



¿Es posible realizar todas las tareas en un único Job?

