



**UNIVERSIDAD DE BUENOS AIRES**

**Departamento de Computación**

**Facultad de Ciencias Exactas y Naturales**

## **Bases de Datos**

### **Trabajo Práctico 2**

**10 de noviembre de 2015**

<b>Integrante</b>	<b>LU</b>	<b>Correo electrónico</b>
Maurizio, Miguel Sebastián	635/11	miguelmaurizio.92@gmail.com
Prillo, Sebastián	616/11	sebastianprillo@gmail.com
Tagliavini Ponce, Guido	783/11	guido.tag@gmail.com

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Ejercicio 1</b>	<b>1</b>
2.1. Desnormalización	1
2.1.1. Documento <b>empleados</b>	1
2.1.2. Documento <b>articulos</b>	2
2.1.3. Documento <b>sectores</b>	2
2.1.4. Documento <b>clientes</b>	3
2.1.5. Documento <b>tareas</b>	4
2.2. Consultas	4
2.2.1. Los empleados que atendieron clientes mayores de edad	4
2.2.2. Los artículos más vendidos	4
2.2.3. Los sectores donde trabajan exactamente 3 empleados	5
2.2.4. El empleado que trabaja en más sectores	5
2.2.5. Ranking de los clientes con mayor cantidad de compras	5
2.2.6. Cantidad de compras realizadas por clientes de misma edad	6
<b>3. Ejercicio 2</b>	<b>6</b>
3.1. Consultas	6
3.1.1. Cantidad de disposiciones tipo <i>resoluciones</i> que se hayan realizado en Abril de 2013	6
3.1.2. Cantidad de disposiciones por cada tipo definido	6
3.1.3. Fecha más citada para todos los informes	7
3.1.4. Máxima cantidad de páginas utilizadas por cada tipo de disposición	8
<b>4. Ejercicio 3</b>	<b>9</b>
4.1. Ejemplo de sharding en Mongo	9
4.2. Características de un buen atributo para sharding	10
4.3. Ejemplos de sharding	11
<b>5. Ejercicio 4</b>	<b>11</b>
5.1. Consultas	11
5.1.1. Los empleados que atendieron clientes mayores de edad	11
5.1.2. Los artículos más vendidos	12
5.1.3. Los sectores donde trabajan exactamente 3 empleados	12
5.1.4. El empleado que trabaja en más sectores	12
5.1.5. Ranking de los clientes con mayor cantidad de compras	12
5.1.6. Cantidad de compras realizadas por clientes de misma edad	13
5.2. Consultas MapReduce	13
5.3. Sharding	13
<b>6. Conclusión</b>	<b>14</b>

# 1. Introducción

## 2. Ejercicio 1

En este ejercicio diseñamos una base de datos NoSQL de tipo documentos con la capacidad de responder rápidamente a varias consultas. Para ello, empleamos desnormalización, diseñando documentos adecuados para las consultas.

### 2.1. Desnormalización

Desnormalizamos el esquema, de la forma tradicional. Cada entidad tiene un documento asociado. Las relaciones 1:N se incluyen en el documento asociado al lado 1. Las relaciones N:M se incluyen a ambos lados de la relación.

#### 2.1.1. Documento empleados

```
empleados {  
  nro_legajo : INTEGER  
  nombre : STRING  
  clientes_atendidos : [{dni : STRING, edad : INTEGER, fecha : DATETIME}]  
  sectores_donde_trabaja : [{cod_sector : INTEGER, id_tarea : INTEGER}]  
}
```

Observar que en `clientes_atendidos` nos guardamos, además del atributo identificador `dni` y el atributo de relación `fecha`, la `edad` de los clientes atendidos. Este atributo será necesario para responder una de las consultas, posteriormente.

Insertamos algunos documentos:

```
db.empleados.insert({  
  nro_legajo : 1,  
  nombre : "empleado1",  
  clientes_atendidos : [ {dni : "11111111", edad : 18, fecha : "01/01/2015"},  
                        {dni : "22222222", edad : 17, fecha : "01/02/2015"} ],  
  sectores_donde_trabaja : [{cod_sector : 1, id_tarea : 1} , {cod_sector : 2, cod_tarea : 1}]  
})
```

```
db.empleados.insert({  
  nro_legajo : 2,  
  nombre : "empleado2",  
  clientes_atendidos : [ {dni : "22222222", edad : 17, fecha : "02/02/2015"} ],  
  sectores_donde_trabaja : [{cod_sector : 1, id_tarea : 1}, {cod_sector : 2, cod_tarea : 1}]  
})
```

```
db.empleados.insert({  
  nro_legajo : 3,  
  nombre : "empleado3",  
  clientes_atendidos : [{dni : "33333333", edad : 19, fecha : "01/03/2015"}],  
  sectores_donde_trabaja : [{cod_sector : 1, id_tarea : 1}, {cod_sector : 2, cod_tarea : 1}]  
})
```

```

db.empleados.insert({
  nro_legajo : 4,
  nombre : "empleado4",
  clientes_atendidos : [],
  sectores_donde_trabaja : [{cod_sector : 1, id_tarea : 1}]
})

```

### 2.1.2. Documento articulos

```

articulos {
  codigo : STRING
  nombre : STRING
  ventas : [{dni : STRING}]
}

```

Insertamos algunos documentos:

```

db.articulos.insert({
  codigo : "1",
  nombre : "articulo1",
  ventas : [{dni : "11111111"}, {dni : "22222222"}]
})

```

```

db.articulos.insert({
  codigo : "2",
  nombre : "articulo2",
  ventas : [{dni : "11111111"}, {dni : "33333333"}]
})

```

```

db.articulos.insert({
  codigo : "3",
  nombre : "articulo3",
  ventas : [{dni : "33333333"}]
})

```

### 2.1.3. Documento sectores

```

sectores{
  cod_sector : INTEGER
  articulos : [{codigo : STRING}]
  trabaja : [{nro_legajo : INTEGER, id_tarea : INTEGER}]
}

```

Insertamos algunos documentos:

```

db.sectores.insert({
  cod_sector : 1,
  articulos : [{codigo : "1"}],
  trabaja : [ {nro_legajo : 1, id_tarea : 1},
               {nro_legajo : 2, id_tarea : 1},

```

```

        {nro_legajo : 3, id_tarea : 1},
        {nro_legajo : 4, id_tarea : 1}]
    })

db.sectores.insert({
    cod_sector : 2,
    articulos : [{codigo : "2"}],
    trabaja : [ {nro_legajo : 1, id_tarea : 1},
                 {nro_legajo : 2, id_tarea : 1},
                 {nro_legajo : 3, id_tarea : 1}]
})

```

#### 2.1.4. Documento clientes

```

clientes{
    dni : STRING
    nombre : STRING,
    edad : INTEGER,
    atendido_por : [{nro_legajo : INTEGER, fecha : DATETIME}],
    compro_articulos : [{codigo : INTEGER}]
}

```

Insertamos algunos documentos:

```

db.clientes.insert({
    dni : "11111111",
    nombre : "cliente1",
    edad : 18,
    atendido_por : [{nro_legajo : 1, fecha : "01/01/2015"}],
    compro_articulos : [{codigo : "1"} , {codigo : "2"}]
})

db.clientes.insert({
    dni : "22222222",
    nombre : "cliente2",
    edad : 17,
    atendido_por : [{nro_legajo : 1, fecha : "01/02/2015"} , {nro_legajo : 2,
                                                                    fecha : "02/02/2015"}],
    compro_articulos : [{codigo : "1"}]
})

db.clientes.insert({
    dni : "33333333",
    nombre : "cliente3",
    edad : 19,
    atendido_por : [{nro_legajo : 1, fecha : "01/03/2015"}],
    compro_articulos : [{codigo : "2"} , {codigo : "3"}]
})

```

### 2.1.5. Documento tareas

Si bien este documento no es estrictamente necesario para poder responder a las consultas, incluimos su definición por completitud.

```
tareas{
  id_tarea : INT,
  descripcion : STRING,
  empleados_realizandola : [{nro_legajo : INTEGER, cod_sector: INTEGER}]
}
```

## 2.2. Consultas

### 2.2.1. Los empleados que atendieron clientes mayores de edad

```
db.empleados.find(
  {"clientes_atendidos.edad" : {$gte : 18}},
  {nro_legajo : 1, nombre : 1}
)
```

Equivalenemente, podriamos pedir:

```
db.empleados.find(
  {"clientes_atendidos" : {$elemMatch : {"edad" : {$gte : 18}}}},
  {nro_legajo : 1, nombre : 1}
)
```

La query nos retorna, como deseamos:

```
{ "_id" : ObjectId("56420cfd92d300ad159887da"), "nro_legajo" : 1, "nombre" : "empleado1" }
{ "_id" : ObjectId("56420cfd92d300ad159887dc"), "nro_legajo" : 3, "nombre" : "empleado3" }
```

### 2.2.2. Los artículos más vendidos

```
db.articulos.aggregate([
  {$project : {"cant_ventas" : {$size : "$ventas"} , nombre : 1, codigo : 1}},
  {$sort : {"cant_ventas" : -1}},
])
```

La respuesta a esta consulta es:

```
{ "_id" : ObjectId("56420d1e92d300ad159887de"), "codigo" : "1", "nombre" : "articulo1",
  "cant_ventas" : 2 }
{ "_id" : ObjectId("56420d1e92d300ad159887df"), "codigo" : "2", "nombre" : "articulo2",
  "cant_ventas" : 2 }
{ "_id" : ObjectId("56420d1e92d300ad159887e0"), "codigo" : "3", "nombre" : "articulo3",
  "cant_ventas" : 1 }
```

Así, vemos que los articulos mas vendidos son los de codigo 1 y 2, cada uno con 2 ventas.

### 2.2.3. Los sectores donde trabajan exactamente 3 empleados

```
db.sectores.find(  
  {trabaja : {$size : 3}},  
  {cod_sector : 1}  
)
```

La respuesta a esta consulta es:

```
{ "_id" : ObjectId("56420d9e92d300ad159887e2"), "cod_sector" : 2 }
```

Efectivamente, el sector 2 contiene exactamente 3 trabajadores.

### 2.2.4. El empleado que trabaja en más sectores

```
db.empleados.aggregate([  
  {$project : {cant_sectores : {$size : "$sectores_donde_trabaja"} , nombre : 1, nro_legajo : 1}},  
  {$sort : {cant_sectores : -1}},  
  {$limit : 1}  
)
```

La respuesta a esta consulta es:

```
{ "_id" : ObjectId("56420cfd92d300ad159887da"), "nro_legajo" : 1, "nombre" : "empleado1",  
  "cant_sectores" : 2 }
```

Si bien los empleados 2 y 3 también trabajan en dos sectores, solo se nos pide dar uno solo, así que desempataremos arbitrariamente.

### 2.2.5. Ranking de los clientes con mayor cantidad de compras

```
db.clientes.aggregate([  
  {$project : {cant_compras : {$size : "$compro_articulos"} , nombre : 1, dni : 1}},  
  {$sort : {cant_compras : -1}}  
)
```

La respuesta a esta consulta es:

```
{ "_id" : ObjectId("56420df292d300ad159887e3"), "dni" : "11111111", "nombre" : "cliente1",  
  "cant_compras" : 2 }  
{ "_id" : ObjectId("56420df392d300ad159887e5"), "dni" : "33333333", "nombre" : "cliente3",  
  "cant_compras" : 2 }  
{ "_id" : ObjectId("56420df292d300ad159887e4"), "dni" : "22222222", "nombre" : "cliente2",  
  "cant_compras" : 1 }
```

Como vemos, los clientes aparecen ordenados de mayor a menor cantidad de compras.

### 2.2.6. Cantidad de compras realizadas por clientes de misma edad

```
db.clientes.aggregate([
  {$project : {cant_compras : {$size : "$compro_articulos"} , nombre : 1, dni : 1, edad : 1}},
  {$group :{
    _id : "$edad",
    total_compras : {$sum : "$cant_compras"}
  }}
])
```

El resultado de la query es:

```
{ "_id" : 19, "total_compras" : 2 }
{ "_id" : 17, "total_compras" : 1 }
{ "_id" : 18, "total_compras" : 2 }
```

En definitiva, este es un diseño de documentos que, mediante documentos adecuados y redundancia, permite responder rapidamente a todas las queries pedidas.

## 3. Ejercicio 2

### 3.1. Consultas

#### 3.1.1. Cantidad de disposiciones tipo *resoluciones* que se hayan realizado en Abril de 2013

```
var map1 = function(){
  emit(this["Tipo"],1)
}

var reduce1 = function(key,values){
  return Array.sum(values)
}

db.disposiciones.mapReduce(map1,reduce1,{query : {FechaDisposicion : {$regex : /^2013-04-/},
  Tipo : "Resoluciones"}, out : "map_res"})
db.map_res.find()
```

El resultado de esta consulta es:

```
{ "_id" : "Resoluciones", "value" : 642 }
```

#### 3.1.2. Cantidad de disposiciones por cada tipo definido

```
var map2 = function(){
  emit(this["Tipo"],1)
}

var reduce2 = function(key,values){
  return Array.sum(values)
}
```



```
db.disposiciones.mapReduce(map2,reduce2,{query : {}, out : "map_res"})
db.map_res.find()
```

El resultado de esta consulta es:

```
{ "_id" : "", "value" : 3 }
{ "_id" : "Acuerdos", "value" : 2790 }
{ "_id" : "Acuerdos del Consejo de Gobierno", "value" : 160 }
{ "_id" : "Anuncios", "value" : 17226 }
{ "_id" : "Candidaturas", "value" : 1 }
{ "_id" : "Certificaciones", "value" : 72 }
{ "_id" : "Circular", "value" : 1 }
{ "_id" : "Conflictos Positivos", "value" : 6 }
{ "_id" : "Correcciones de Erratas", "value" : 59 }
{ "_id" : "Correcciones de Errores", "value" : 321 }
{ "_id" : "Correcciones de erratas", "value" : 8 }
{ "_id" : "Correcciones de errores", "value" : 44 }
{ "_id" : "Corrección de errores", "value" : 1 }
{ "_id" : "Corrección de errores", "value" : 1 }
{ "_id" : "Cuestiones de Inconstitucionalidad", "value" : 1 }
{ "_id" : "Decretos", "value" : 828 }
{ "_id" : "Decretos Legislativos", "value" : 5 }
{ "_id" : "Decretos del Presidente", "value" : 13 }
{ "_id" : "Decretos-leyes", "value" : 15 }
{ "_id" : "Edictos", "value" : 3223 }
{ "_id" : "Instrucciones", "value" : 2 }
{ "_id" : "Leyes", "value" : 12 }
{ "_id" : "Notificaciones", "value" : 768 }
{ "_id" : "Orden", "value" : 4 }
{ "_id" : "Otros", "value" : 45 }
{ "_id" : "Reales Decretos", "value" : 5 }
{ "_id" : "Recursos de Inconstitucionalidad", "value" : 7 }
{ "_id" : "Requisitorias", "value" : 2 }
{ "_id" : "Resoluciones", "value" : 13956 }
{ "_id" : "Órdenes", "value" : 2061 }
{ "_id" : "Órdenes de Comision Delegada", "value" : 2 }
```

### 3.1.3. Fecha más citada para todos los informes

```
var map3 = function(){
    emit(this["FechaBOJA"],1)
}

var reduce3 = function(key,values){
    return Array.sum(values)
}

db.disposiciones.mapReduce(map3,reduce3,{query : {}, out : "map_res"})
db.map_res.find().sort({value : -1}).limit(1)
```

El resultado de esta consulta es:

```
{ "_id" : "01/08/2012", "value" : 242 }
```

### 3.1.4. Máxima cantidad de páginas utilizadas por cada tipo de disposición

```
var map4 = function(){
    emit(this["Tipo"],this["PaginaFinal"] - this["PaginaInicial"] + 1)
}

var reduce4 = function(key,values){
    res = values[0]
    for (var i=1; i < values.length; i++){
        if(res < values[i]){
            res = values[i]
        }
    }
    return res
}

db.disposiciones.mapReduce(map4,reduce4,{query : {}, out : "map_res"})
db.map_res.find()
```

El resultado de esta consulta es:

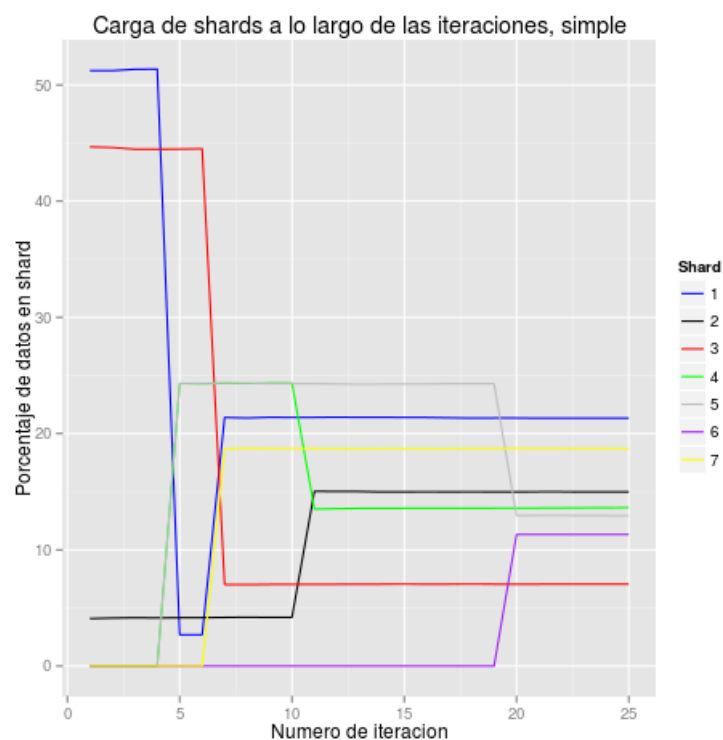
```
{ "_id" : "", "value" : 16 }
{ "_id" : "Acuerdos", "value" : 12 }
{ "_id" : "Acuerdos del Consejo de Gobierno", "value" : 83 }
{ "_id" : "Anuncios", "value" : 393 }
{ "_id" : "Candidaturas", "value" : 68 }
{ "_id" : "Certificaciones", "value" : 47 }
{ "_id" : "Circular", "value" : 3 }
{ "_id" : "Conflictos Positivos", "value" : 1 }
{ "_id" : "Correcciones de Erratas", "value" : 174 }
{ "_id" : "Correcciones de Errores", "value" : 99 }
{ "_id" : "Correcciones de erratas", "value" : 10 }
{ "_id" : "Correcciones de errores", "value" : 5 }
{ "_id" : "Corrección de errores", "value" : 1 }
{ "_id" : "Corrección de errores", "value" : 1 }
{ "_id" : "Cuestiones de Inconstitucionalidad", "value" : 1 }
{ "_id" : "Decretos", "value" : 492 }
{ "_id" : "Decretos Legislativos", "value" : 37 }
{ "_id" : "Decretos del Presidente", "value" : 3 }
{ "_id" : "Decretos-leyes", "value" : 139 }
{ "_id" : "Edictos", "value" : 48 }
{ "_id" : "Instrucciones", "value" : 3 }
{ "_id" : "Leyes", "value" : 194 }
{ "_id" : "Notificaciones", "value" : 11 }
{ "_id" : "Orden", "value" : 7 }
{ "_id" : "Otros", "value" : 65 }
{ "_id" : "Reales Decretos", "value" : 1 }
{ "_id" : "Recursos de Inconstitucionalidad", "value" : 1 }
{ "_id" : "Requisitorias", "value" : 1 }
```

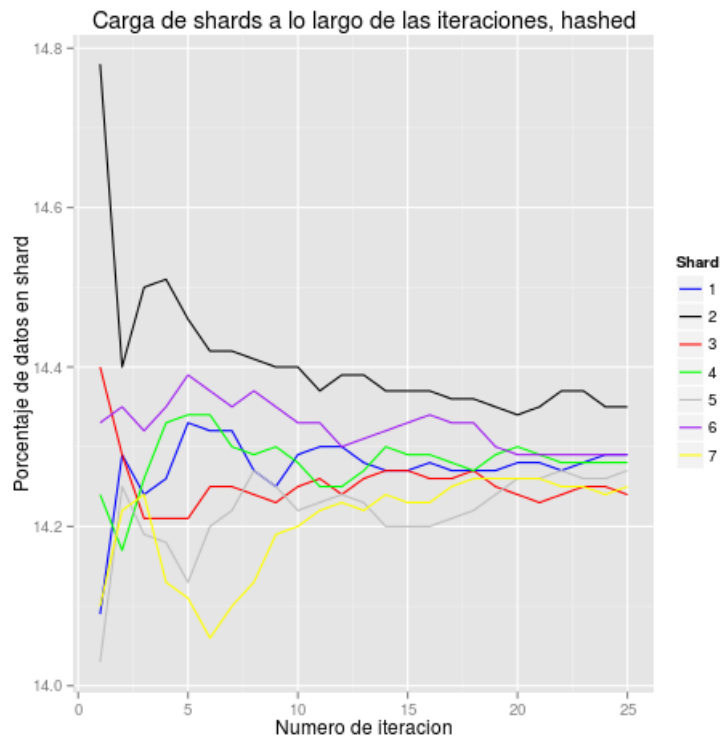
```
{ "_id" : "Resoluciones", "value" : 454 }
{ "_id" : "Órdenes", "value" : 634 }
{ "_id" : "Órdenes de Comision Delegada", "value" : 2 }
```

## 4. Ejercicio 3

### 4.1. Ejemplo de sharding en Mongo

Seguimos las sugerencias del enunciado y analizamos los efectos de usar sharding simple o hasheado en una base de datos en la que insertamos iterativamente tandas de a 20000 registros, para un total de 25 tandas. A continuación mostramos los gráficos de carga de los shards a lo largo de las iteraciones:





Recordemos que el atributo usado de clave para los shards es el código postal, que es un entero de hasta 6 dígitos. Como podemos apreciar, cuando hacemos sharding hasheando el código postal, la distribución de los datos a lo largo de los shards tiende a estar bien balanceada. Por el contrario, si no se hashea el código postal, mongo empieza cargando fuertemente dos shards, y luego empiezan a tener más participación otros. Esto se debe a que la forma de determinar qué documentos van a qué shards cuando la clave no se hashea está determinado por rangos del código postal, que mongo no conoce a priori y debe ir estimando. Una vez que la cantidad de datos crece, mongo puede inferir que la distribución es uniforme entre cero y un millón y la carga se empieza a equilibrar al darle rangos equitativos de responsabilidad a cada shard. Pero al principio, la distribución es extremadamente dispar.

## 4.2. Características de un buen atributo para sharding

Para que un atributo sea un buen candidato para aplicar la técnica de sharding, debe ser tal que al particionar los datos por ese atributo, las clases que se formen sean de tamaños similares. Esto permite que la carga sobre los shards esté bien distribuida, no sólo en términos del volumen de cada shard, sino también en el sentido de que los accesos y escrituras sean uniformes. Bajo estas condiciones obtendremos una mejor performance de I/O. Específicamente, la latencia y el tiempo de respuesta de los mismos serán bajos, gracias a que evitamos cuellos de botella.

Para ilustrar esto, pensemos en un mal ejemplo de sharding. Consideremos una base de datos que contenga los datos de los alumnos del Departamento de Computación de la facultad. Si hacemos sharding sobre el atributo *género*, tendremos dos shards completamente asimétricos en su tamaño. Claramente, la enorme mayoría de los accesos serán sobre el shard de alumnos de sexo masculino, lo cual tiene un obvio impacto en la performance.

Vale la pena mencionar que el sharding puede ser de utilidad no sólo para asegurar balanceo de carga. En ciertos sistemas, son habituales las consultas que necesitan revisar únicamente un subconjunto de todo el universo de elementos. En estos casos, es útil hacer sharding de modo tal de que uno de estos subconjuntos que nos interesan estén separados en shards. Tomemos como ejemplo una base de datos de una red social, en la cual se hace sharding por el país de residencia de las personas. En este caso, podremos optimizar la consulta *amigos*

de una persona, puesto que, en general, la mayor parte de los amigos de una persona están geográficamente cerca.

### 4.3. Ejemplos de sharding

Algunos ejemplos de una buena elección de un atributo para hacer sharding son los siguientes:

1. Sharding por atributo *género* en una base de datos del padrón electoral de un país.
2. Sharding por atributo *país de residencia* en una base de datos de personas de una red social.
3. Sharding por atributo *año de nacimiento* en una base de datos de personas nacidas durante cierto período de tiempo, fijo. Por ejemplo, durante la dictadura de 1976.

Se puede ver que en los tres casos, la proporción de los grupos en los que clasificamos es aproximadamente la misma, y que, a priori, no debería haber ninguna tendencia de acceso a un shard en particular.

Una alternativa siempre útil es hacer sharding utilizando hashing. Esto significa tomar un atributo cuya distribución sea más o menos uniforme (por ejemplo, un atributo identificador o un número de teléfono), y shardear utilizando el hash del atributo.

## 5. Ejercicio 4

Las bases de datos NoSQL del tipo *column family* almacenan la información, no por filas como las bases de datos SQL tradicionales, sino por columnas. Esto permite realizar queries que requieren acceder a grandes cantidades de datos eficientemente. Esta eficiencia se basa en dos hipótesis. La primera es que, en general, las consultas sólo requieren un subconjunto de la totalidad de las columnas de una tabla, con lo cual resulta conveniente poder restringirse sólo a las columnas necesarias. La segunda es que, cuando la cantidad de datos de la respuesta es grande, tendremos que leer casi todos los valores de una columna, por lo que resulta útil que estén almacenados consecutivamente.

Una base de datos de este tipo se compone de *familias de columnas*. Cada familia de columnas se compone de varias *claves de fila* (simbolizadas con K), y asociada a cada una de estas claves hay *columnas* (simbolizadas con C) con sus respectivos valores. Finalmente, asociado a cada clave de fila también suele almacenarse un timestamp, que es utilizado para determinar el tiempo de expiración de los datos.

Para diseñar bases de datos de este tipo, creamos, para cada query, una o más familias de columnas que sean capaces de responder a la consulta. Esto implica que los datos almacenados son estrictamente los requeridos para responder a las consultas.

### 5.1. Consultas

#### 5.1.1. Los empleados que atendieron clientes mayores de edad

idRow	K
legajo_empleado_que_atendio_mayor	C ↑

En este caso, tenemos una clave de fila *dummy*, *idRow*, en la cual se almacenarán una secuencia de columnas *legajo\_empleado\_que\_atendio\_mayor*. Cada valor de una columna *legajo\_empleado\_que\_atendio\_mayor* es el identificador de un empleado que atendió a una persona mayor de edad.

La actualización de esta familia es sencilla. Cada vez que se realiza una venta, debemos verificar si el cliente es mayor de edad, en cuyo caso insertamos en la familia al empleado que lo atendió, siempre que no estuviera en la familia de antemano.

### 5.1.2. Los artículos más vendidos

codigo_articulo	K
cantidad_vendida	++

A cada artículo le asociamos un contador `cantidad_vendida`. Con esta información es fácil extraer aquellos artículos cuyo valor del contador sea máximo.

La actualización consiste en incrementar el contador de un artículo, cada vez que es vendido.

### 5.1.3. Los sectores donde trabajan exactamente 3 empleados

codigo_sector	K
legajo_empleado_del_sector	C ↑

Para cada sector, almacenamos todos los empleados que trabajan allí. Determinar aquellos sectores donde trabajan exactamente 3 empleados consiste en determinar cuáles tienen exactamente 3 columnas.

### 5.1.4. El empleado que trabaja en más sectores

Aquí podemos saber para un empleado todos los sectores donde trabajó, así que también sabemos la cantidad.

legajo_empleado	K
codigo_sector_donde_trabaja	C ↑

La desventaja de esta implementación es que debemos revisar todos los empleados de la base de datos. En la siguiente solución alternativa remendamos eso agregando una tabla.

row_id	K
cantidad_sectores_trabaja	C ↑
empleados	

Entonces al agregar para un empleado un nuevo sector donde trabajo, primero nos fijamos si en la primera tabla aparece dicho sector. Si es así no actualizamos nada. Si no, sea  $e$  el id el número de legajo del empleado,  $s$  el nuevo sector y  $n$  la cantidad de anterior donde trabajaba. Debemos remover a  $e$  de aquellos que trabajaban en  $n$  secciones y agregarlo en  $n + 1$ . Además, hay que actualizar la primera tabla.

Manteniendo esta relación solo tenemos que ver la primera columna de `cantidad_sectores_trabaja` y tomar algún empleado.

### 5.1.5. Ranking de los clientes con mayor cantidad de compras

Este es realmente muy similar al caso anterior. Podríamos decir, tal vez, que puede haber muchas cantidades de compras por usuario, pero esto no debería ser un problema. Proponemos:

dni_usuario	K
ids_compras_realizadas	C ↑

row_id	K
cantidad_compras_realizadas	C ↑
empleados	

Notar que en las primeras columnas tenemos, justamente, los clientes que más compraron.

En cambio, si no nos interesa cuales fueron exactamente las compras y podemos recorrer toda la base de datos, la siguiente alternativa resulta razonable:

dni_cliente	K
cantidad_compras	++

Es decir, un contador de cantidad de compras por cliente.

#### 5.1.6. Cantidad de compras realizadas por clientes de misma edad

Asumiendo que no necesitamos los clientes, sino solo la cantidad, podemos tener la siguiente implementación, que es básicamente un contador por edad posible de cliente.

edad_cliente	K
cantidad_compras	++

### 5.2. Consultas MapReduce

El motor de bases de datos column-family Cassandra soporta MapReduce desde abril de 2010. Además provee la posibilidad de integrarse con Apache Hadoop, herramienta que permite realizar procesamientos distribuidos de datos, y que incorpora un motor de consultas MapReduce. Esto hace que la forma de pensar las consultas MapReduce depende de la interfaz que provee Hadoop, y no de la clase de bases de datos que vamos.

Hadoop tiene ciertas características que impactan en la ejecución de consultas MapReduce. Algunas de ellas son:

- **Escalabilidad.** Es capaz de majear volúmenes de datos importantes, al nivel de miles de nodos, cada uno con una carga de varios terabytes de información.
- **Usabilidad.** Es muy simple de usar, el programador solo debe escribir los procedimientos realcionados con map-reduce.
- **Resistencia a fallos.** Ante la caída de un nodo el sistema puede seguir operando.

Para realizar las consultas MapReduce del Ejercicio 2 para column oriented, primero deberíamos generar una o varias tablas, que contengan toda la información que queremos consultar. En este caso, deberíamos generar un diseño orientado a columnas, para los archivos JSON. Esto es lo único que cambia, puesto, las consultas MapReduce son análogas. La clave de una emisión de MapReduce es una clave de fila (K) columna (C) cualquiera y el valor asociado a esta clave es alguna función del resto de los valores de la fila.

### 5.3. Sharding

Cassandra provee un mecanismo de sharding sobre un cluster de nodos, mediante hashing. Cada registro insertado se almacena en una serie de nodos dado por el hashing de una clave del registro. Para determinar a qué nodo corresponde cierto valor del hash, se disponen los nodos formando un anillo, de modo tal que cada porción del anillo le corresponde a un nodo y a cierto rango de valores de hash.

Al utilizar sharding por hash, tenemos asegurada la uniformidad en la distribución de los datos, siempre y cuando sepamos que el atributo que se está hasheando también aparece con una distribución uniforme. En el caso del Ejercicio 3, el atributo `codigo_postal` se asume que tiene distribución uniforme (se generan en forma aleatoria), con lo cual la carga de los shards estará balanceada. Del mismo modo, si se hace sharding por un atributo `_id`, generado aleatoriamente, obtendremos una uniforme distribución de la carga.

## 6. Conclusión

En este trabajo hemos estudiado dos modelos de bases de datos no relacionales.

Por un lado, hemos visto, en profundidad, cómo realizar un modelado de una *document oriented*, responder consultas sobre la base de datos, realizar consultas MapReduce, y aplicar técnicas de sharding para que los accesos a la base de datos en un hipotético escenario de clustering, sea eficiente. Se pudo ver que este el modelado en esta clase de bases de datos es sencillo, y permite que las consultas sean más sencillas que en SQL tradicional, puesto que este diseño se orienta específicamente a responder las consultas de nuestro interés. La capacidad de realizar consultas MapReduce y de utilizar sharding ponen en evidencia lo preparado de esta clase de base de datos para un escenario de cómputo distribuido, en el cual los datos se distribuyen sobre más de una computadora. Bajo ciertas hipótesis de uniformidad sobre la información ingresada en la base de datos, mostramos que el sharding permite balancear la carga de datos sobre todos los shards, lo cual, en el mundo real, tiene un impacto no sólo en la optimización de los recursos de hardware (ningún nodo requiere demasiado disco), sino también desde el punto de vista del tiempo de respuesta de las queries (el ancho de banda necesario para mantener un delay aceptable es bajo, pues no hay nodos que sean cuellos de botella). Además, en muchos casos donde se trabaja con cantidades enormes de datos, SQL es imposible de utilizar. La gran ventaja de las bases de NOSQL es que permite que escalen en forma horizontal, además de vertical. Es decir, hablando en forma simplificada, para procesar un mayor flujo en SQL tradicional necesitamos una *mejor* computadora, en cambio en NOSQL podemos utilizar una *mayor cantidad* de computadoras. Por otro lado, estudiamos superficialmente el modelo *column oriented*. Como se vió, este modelo provee prestaciones similares, al permitir tanto consultas MapReduce como sharding. La disyuntiva en la elección entre document oriented y column oriented dependerá fuertemente del contexto en el que estemos trabajando. De hecho, incluso dentro de una misma familia, dada la gran cantidad de oferta de bases de datos NOSQL, existen diferentes implementaciones, con ventajas y desventajas que el desarrollador deberá evaluar a la hora de su elección.