UNIVERSIDAD DE BUENOS AIRES

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Bases de Datos

Trabajo Práctico 2

10 de noviembre de 2015

Integrante	LU	Correo electrónico
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

1. Introducción

2. Ejercicio 1

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

2.1. Query 1

Debemos poder responder rapidamente cuales son los empleados que atendieron clientes mayores de edad. Para ello, tendremos una tabla **empleados**, con los siguientes campos:

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

Insertemos algunos empleado a la tabla:

```
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}]
})
```

De esta manera, la query se responde rapidamente con la siguiente consulta:

```
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. Query 2

Debemos poder conocer los articulos mas vendidos. Para ello, definimos una tabla articulo con los siguientes campos:

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

De esta manera, la longitus de la lista ventas nos dice cuantas veces fue vendido un articulo. Populemos la tabla de articulos con los siguientes ejemplo:

```
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"}]
})
```

De esta manera, la siguiente query responde nuestra consulta:

```
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_vent
{ "_id" : ObjectId("56420d1e92d300ad159887df"), "codigo" : "2", "nombre" : "articulo2", "cant_vent
{ "_id" : ObjectId("56420d1e92d300ad159887e0"), "codigo" : "3", "nombre" : "articulo3", "cant_vent
```

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

2.3. Query 3

Debemos poder conocer los sectores donde trabaja exactamente 3 empleados. Para ello, definimos el documento **sectores** que tiene la informacion sobre cada sector, incluyendo los empleados que trabajan en el.

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

Populemos la tabla de empleados:

```
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}]
})
```

De esta manera, la query queda:

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

Si la corremos, obtenemos:

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

Efectivamente, el sector 2 contiene exactamente 3 trabajadores.

2.4. Query 4

Debemos conocer el empleado que trabaja en mas sectores. Esto es facil, usando los documentos ya definidos:

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

La respuesta es:

```
{ "_id" : ObjectId("56420cfd92d300ad159887da"), "nro_legajo" : 1, "nombre" : "empleado1", "cant_se
```

Si bien los empleados 2 y 3 tambien trabajan en dos sectores, solo se nos pide dar uno solo, asi que desempatamos arbitrariamente.

2.5. Query 5

Esta query pide el ranking de los clientes con mayor cantidad de compras. Para ello, definimos los documentos **cliente**, que tienen los siguientes atributos:

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

Insertamos algunos clientes:

\begin{verbatim}
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"}]
})
```

La query queda:

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

El resultado es:

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

Como vemos, los clientes aparecen ordenamos de mayor a menos cantidad de compras.

2.6. Query 6

Nos piden la cantidad de compras realizadas por clientes de misma edad. Esta query puede ser respondida con los documentos ya creados anteriormente facilmente:

```
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 }
```

De esta manera, creamos un diseño de documentos que, mediante documentos adecuados y redundancia, permite responder rapidamente a las queries pedidas.

3. Ejercicio 2

4. Ejercicio 3

4.1. 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 distribuída, no sólo en términos del volúmen 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 un 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.

4.2. 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 identificatorio o un número de teléfono), y shardear utilizando el hasheo 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 subconjuntos de la totalidad de las columnas de una tabla, con lo cual resulta conveniente poder restrigirse 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 trabaja 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

legajo_empleado	K
codigo_sector_donde_trabaja	C ↑

5.1.5. Ranking de los clientes con mayor cantidad de compras

dni_cliente	K
cantidad_compras	++

5.1.6. Cantidad de compras realizadas por clientes de misma edad

edad_cliente	K
cantidad_compras	++

6. Conclusión

El ejercicio del diseño una base de datos de una escala considerable, nos dejó algunas enseñanzas. Por un lado, nos permitió adquirir mayor confianza en el trabajo con el modelado de problemas vía bases de datos relacionales, y la implementación en SQL. Pero más importante, nos permitió sentir en carne propia las dificultades del diseño, que hacen que de esto un proceso iterativo, en el cual sólo se llega a la solución final a través de refinamientos. En nuestro caso particular, la confección del DER la desarrollamos a lo largo de varias semanas, realizando sucesivas veces el antedicho refinamiento. Gracias a esto, la implementación fue fluida, sin mayores inconvenientes, más allá de los del aprendizaje del lenguaje de consultas.