



# Bases de Datos 2 2021 -TP3

Bases de Datos NoSQL / Práctica con MongoDB

**entrega: 31/5**

## Parte 1: Bases de Datos NoSQL y Relacionales

► Si bien las BBDD NoSQL tienen diferencias fundamentales con los sistemas de BBDD Relacionales o RDBMS, algunos conceptos comunes se pueden relacionar. Responda las siguientes preguntas, considerando MongoDB en particular como Base de Datos NoSQL.

1. ¿Cuáles de los siguientes conceptos de RDBMS existen en MongoDB? En caso de no existir, ¿hay alguna alternativa? ¿Cuál es?

- Base de Datos
- Tabla / Relación
- Fila / Tupla
- Columna

**Rta:**

- **Base de Datos:** Compartido por ambos tipos de BBDD, tanto RDBMS como MongoDB.
- **Tabla / Relación:** El análogo en el mundo de MongoDB serían las “Colecciones”, que actúan como contenedores de una serie de documentos de similar especie.
- **Fila / Tupla:** El análogo en el mundo de MongoDB sería el concepto de “Documento”, que tiene que adherirse al estándar JSON como formato requerido. Por lo tanto, en MongoDB se entiende a un documento como una entidad que agrupa una serie de campos, con sus correspondientes valores.
- **Columna:** Como en MongoDB se manejan colecciones de documentos con formato JSON, el análogo al concepto de “Columna” vinculado al paradigma relacional sería un campo(o “Name” según el estándar [JSON](#)) vinculado a un documento. Cada campo o “Name” tiene asociado un valor, es por eso que un documento se suele indicar como una serie de pares “clave/valor” que no siguen ningún orden particular.

2. MongoDB tiene soporte para transacciones, pero no es igual que el de los RDBMS. ¿Cuál es el alcance de una transacción en MongoDB?

**Rta:**

MongoDB por defecto garantiza atomicidad en cada operación (lectura/escritura) que se haga contra un documento. Como generalmente cada documento suele tener asociado datos que en un modelo relacional se distribuiría en varias tablas, en general se considera suficiente para la generalidad de casos de uso la propuesta de garantizar atomicidad a nivel de documento. Evidentemente siguen existiendo casos en donde estaría bueno garantizar atomicidad en operaciones complejas que abarcan varios documentos, incluso de distintas colecciones. Desde la versión 4.0, MongoDB provee transacciones multi-documento y que trabajan no solo sobre el nodo

principal, si no que se reflejan en sus replica-sets (nodos “backup” del principal).

3. Para acelerar las consultas, MongoDB tiene soporte para índices. ¿Qué tipos de índices soporta?

**Rta:**

MongoDB soporta los siguientes tipos de índices:

- **Single Field Index:** Este tipo de índice se aplica sobre un campo de un documento. Permite ordenar una colección en base a dicho campo y su sort-criteria (Ascendente o Descendente), con lo que las búsquedas por igualdad o rangos se vuelve muy eficiente.
- **Compound Index:** Este tipo de índices permite extender la idea del “Single Field Index” a más de un campo, es decir que ahora podemos hacer índices compuestos por más de una clave. Por cada campo del índice compuesto se indica el sort-criteria, y para efectuar la construcción del mismo MongoDB respeta el orden de campos indicado por el programador. Es entonces que el orden de los campos a indexar se vuelve importante, ya que con otro orden la estructura resultante podría no ser de utilidad para el caso de uso en desarrollo.
- **Multikey Index:** En ciertos escenarios, puede ocurrir que un campo tenga como valor asociado un Array de valores, y sea necesario buscar todos los documentos que contengan ciertos elementos dentro de dicho campo de tipo Array. MongoDB introduce los índices multi-clave, que permiten indexar la información de campos de tipo Array, para su posterior búsqueda eficiente.
- **Geospatial Index:** Para admitir consultas eficientes de datos de coordenadas geoespaciales, MongoDB proporciona dos índices especiales: índices 2d que usan geometría plana al devolver resultados e índices 2dsphere que usan geometría esférica para devolver resultados.
- **Text Indexes:** Este tipo de índices permite la búsqueda de strings dentro de una colección, dotando al motor de la posibilidad de realizar búsquedas de texto eficientes, ignorando en su proceso de indexación a palabras vacías, o realizando “Stemming”, un proceso por el cual se toman las palabras de un texto y se las lleva a su forma “Madre” o raíz, evitando conjugaciones y demás agregados sobre dicha palabra. Esto permite que varias palabras expresadas de formas distintas pero que comparten el mismo “Stem” puedan ser consideradas sinónimos.
- **Hashed Indexes:** Este tipo de índices se utiliza para dar soporte al sharding basado en Hashing. Lo que hacen estos índices es indexar el hash del valor del campo usado como clave. Como beneficio, permite hacer una especie de “load-balancing” de cada shard, evitando que claves muy próximas o similares siempre residan en el mismo shard, y agregando un nivel de dispersión mayor a las claves. Como contrapartida, hashing no permite búsquedas por rango, solo de acceso directo (por igualdad).

4. ¿Existen claves foráneas en MongoDB?

**Rta:**

En MongoDB existe el concepto de “Referencia”, que sería el análogo a las “Claves Foráneas” de cualquier RDBMS. Las referencias permiten construir modelos de datos físicos mas “normalizados” reduciendo la redundancia de información, aceptando que los joins decrementen la performance.

## Parte 2: Primeros pasos con MongoDB

► Descargue la última versión de MongoDB desde el [sitio oficial](#). Ingrese al cliente de línea de comando para realizar los siguientes ejercicios.

5. Cree una nueva base de datos llamada **ecommerce**, y una colección llamada **products**. En esa colección inserte un nuevo documento (un producto) con los siguientes atributos:

```
{name:'Caldera Caldaia Duo', price:140000}
```

recupere la información del producto usando el comando `db.products.find()` (puede agregar la función `.pretty()` al final de la expresión para ver los datos indentados). Notará que no se encuentran exactamente los atributos que insertó. ¿Cuál es la diferencia?

Primero, para crear la base de datos usaremos el comando **use <nombre>**.

```
---
> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
> use bd2
switched to db bd2
> db.dropDatabase()
{ "ok" : 1 }
> use ecommerce
switched to db ecommerce
> |
```

Luego, creamos la colección llamada “products” con el comando **db.createCollection(<nombre>)**

```
switched to db ecommerce
> db.createCollection("products")
{ "ok" : 1 }
```

Con el siguiente comando, podemos ver los nombres de las colecciones creadas.

```
> db.getCollectionNames()
[ "products" ]
```

Ahora guardamos el producto con los atributos mencionados usando el comando **insertOne(...)**, y luego lo recuperamos con el comando **find()**, como figura en la siguiente captura.

```

> db.products.insertOne({name:"Caldera Caldaia Duo", price:140000})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("60aaafc06a52ebb72034ec0f2")
}
> db.products.find().pretty()
{
  "_id" : ObjectId("60aaafc06a52ebb72034ec0f2"),
  "name" : "Caldera Caldaia Duo",
  "price" : 140000
}

```

La diferencia entre estos dos últimos, es que al utilizar el segundo comando, podemos ver que el objeto que habíamos creado posee además un id.

En Mongo, los id tienen 12 bytes (4 bytes que representan los segundos desde el epoch de unix, 3 bytes de identificador de máquina, 2 bytes de id del proceso y 3 bytes del contador).

#### 6. Agregue los siguientes documentos a la colección de productos:

```

{name:'Caldera Orbis 230', price:77000, tags: ['gas', 'digital']}
{name:'Caldera Ariston Clas', price:127000, tags: ['gas envasado', 'termostato']}
{name:'Caldera Caldaia S30', price:133000}
{name:'Caldera Mural Orbis 225cto', price:100000, tags: ['gas', 'digital', 'termostato']}

```

Y busque los productos:

- de \$100.000 o menos
- que tengan la etiqueta (*tag*) "digital"
- que no tengan etiquetas (es decir, que el atributo esté ausente)
- que incluyan la palabra 'Orbis' en su nombre
- con la palabra 'Orbis' en su nombre y menores de \$100.000

vuelva a realizar la última consulta pero proyecte sólo el nombre del producto en los resultados, omitiendo incluso el atributo **\_id** de la proyección.

En primer lugar, en este caso como vamos a insertar varios objetos a la vez, optamos por utilizar el comando 'insertMany', de la siguiente forma:

```

> db.products.insertMany([
  {name:"Caldera Orbis 230", price:77000, tags: ["gas", "digital"]},
  {name:"Caldera Ariston Clas", price:127000, tags: ["gas envasado", "termostato"]},
  {name:"Caldera Caldaia S30", price:133000},
  {name:"Caldera Mural Orbis 225cto", price:100000, tags: ["gas", "digital", "termostato"]}
])
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("60aaaff36a52ebb72034ec0f3"),
    ObjectId("60aaaff36a52ebb72034ec0f4"),
    ObjectId("60aaaff36a52ebb72034ec0f5"),
    ObjectId("60aaaff36a52ebb72034ec0f6")
  ]
}

```

A continuación adjuntamos capturas con los comandos ejecutados para resolver las consultas mencionadas, y los resultados dados en cada una:

- Productos de 100.000\$ o menos:

```
> db.products.find({price: {$lte: 100000}})
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f3"), "name" : "Caldera Orbis 230", "price" : 77000, "tags" : [ "gas", "digital" ] }
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f6"), "name" : "Caldera Mural Orbis 225cto", "price" : 100000, "tags" : [ "gas", "digital", "termostato" ] }
```

- Productos que tengan la etiqueta (tag) “digital”:

```
> db.products.find({tags: "digital"})
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f3"), "name" : "Caldera Orbis 230", "price" : 77000, "tags" : [ "gas", "digital" ] }
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f6"), "name" : "Caldera Mural Orbis 225cto", "price" : 100000, "tags" : [ "gas", "digital", "termostato" ] }
```

- Productos que no tengan etiquetas (es decir, que el atributo esté ausente):

```
> db.products.find({tags: {$exists: false}})
{ "_id" : ObjectId("60aafc06a52ebb72034ec0f2"), "name" : "Caldera Caldaia Duo", "price" : 140000 }
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f5"), "name" : "Caldera Caldaia S30", "price" : 133000 }
```

- Productos que incluyan la palabra ‘Orbis’ en su nombre:

```
> db.products.find({"name": {$regex: /Orbis/}})
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f3"), "name" : "Caldera Orbis 230", "price" : 77000, "tags" : [ "gas", "digital" ] }
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f6"), "name" : "Caldera Mural Orbis 225cto", "price" : 100000, "tags" : [ "gas", "digital", "termostato" ] }
```

- Productos con la palabra ‘Orbis’ en su nombre y menores de \$100.000:

```
> db.products.find({name: {$regex: /Orbis/}, price: {$lt: 100000}})
{ "_id" : ObjectId("60aaff36a52ebb72034ec0f3"), "name" : "Caldera Orbis 230", "price" : 77000, "tags" : [ "gas", "digital" ] }
```

- Misma consulta anterior, pero proyectando solo el nombre:

```
> db.products.find({name: {$regex: /Orbis/}, price: {$lt: 100000}}, {name: true, _id: false})
{ "name" : "Caldera Orbis 230" }
```

## 7. Actualice la “Caldera Caldaia S30” cambiándole el precio a \$150.000.

En este caso, primero ejecutamos el comando **updateOne()** para la actualización, y luego el comando **find()** para asegurarnos de que los cambios se habían efectuado correctamente.

```
> db.products.updateOne({name: "Caldera Caldaia S30"}, {$set: {price: 150000}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.products.find({name: "Caldera Caldaia S30"})
{ "_id" : ObjectId("60ab0492a52ebb72034ec0f7"), "name" : "Caldera Caldaia S30", "price" : 150000 }
```

## 8. Cree el array de etiquetas (tags) para la “Caldera Caldaia S30”.

Primero creamos el array vacío de la siguiente forma:

```
> db.products.updateOne({name: "Caldera Caldaia S30"}, {$set: {tags: []}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

## 9. Agregue “digital” a las etiquetas de la “Caldera Caldaia S30”

Luego utilizamos el comando \$push para agregar el tag mencionado.

```
> db.products.updateOne({name: "Caldera Caldaia S30"}, {$push: {tags: "digital"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.products.find({name: "Caldera Caldaia S30"})
{ "_id" : ObjectId("60ab0492a52ebb72034ec0f7"), "name" : "Caldera Caldaia S30", "price" : 150000, "tags" : [ "digital" ] }
```

## 10. Incremente en un 10% los precios de todas las calderas digitales.

```
> db.products.updateMany({tags: "digital"}, {$mul: {price: 1.1}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

Resultados tras la operación:

```
> db.products.find()
{ "_id" : ObjectId("60aaafc06a52ebb72034ec0f2"), "name" : "Caldera Caldaia Duo", "price" : 140000 }
{ "_id" : ObjectId("60aaaff36a52ebb72034ec0f3"), "name" : "Caldera Orbis 230", "price" : 84700, "tags" : [ "gas", "digital" ] }
{ "_id" : ObjectId("60aaaff36a52ebb72034ec0f4"), "name" : "Caldera Ariston Clas", "price" : 127000, "tags" : [ "gas envasado", "termostato" ] }
{ "_id" : ObjectId("60aaaff36a52ebb72034ec0f5"), "price" : 150000 }
{ "_id" : ObjectId("60aaaff36a52ebb72034ec0f6"), "name" : "Caldera Mural Orbis 225cto", "price" : 110000.000000000001, "tags" : [ "gas", "digital", "termostato" ] }
{ "_id" : ObjectId("60ab0492a52ebb72034ec0f7"), "name" : "Caldera Caldaia S30", "price" : 165000, "tags" : [ "digital" ] }
```

## Parte 3: Índices

- Elimine todos los productos de la colección. Guarde en un archivo llamado ‘generador.js’ el siguiente código JavaScript y ejecútelo con: `load(<ruta del archivo ‘generador.js’>)`. Si utiliza un cliente que lo permita (ej. Robo3T), se puede ejecutar directamente en el espacio de consultas.

```
for (var i = 1; i <= 50000; i++) {
  var randomTags = ['envio express', 'oferta', 'cuotas', 'verificado'].sort( function()
  { return 0.5 - Math.random() } ).slice(1, Math.floor(Math.random() * 5)); var
  randomPrice = Math.ceil(110000+(Math.random() * 300000 - 100000));
  db.products.insert({
    name:'Producto '+i,
    price:randomPrice,
    tags: randomTags
  });
}
for (var i = 1; i <= 50000; i++) {
  if (Math.random() > 0.7) {
    var randomPurchases = Math.ceil(Math.random() * 5);
    for (var r = 1; r <= randomPurchases; r++){
      var randomLong = -34.56 - (Math.random() * .23);
```

```

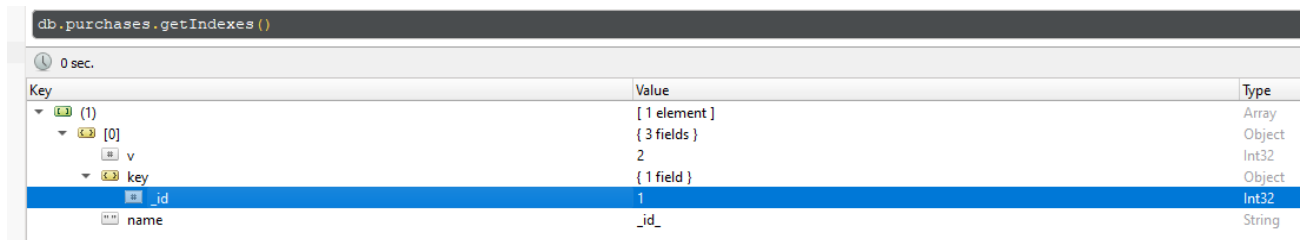
var randomLat = -58.4 - (Math.random() * .22);
var shippingCost = 200+Math.ceil(Math.random()*20) * 100;
db.purchases.insert({
  productName:'Producto '+i,
  shippingCost: shippingCost,
  location: {type: "Point",coordinates: [randomLat, randomLong]} });
}
}
}

```

11. Busque en la colección de compras (purchases) si existe algún índice definido.

**Rta:**

En la colección de compras se encuentra un índice definido para la clave primaria (\_id) que por defecto asigna MongoDB a todo documento.



Key	Value	Type
(1)	[ 1 element ]	Array
[0]	{ 3 fields }	Object
v	2	Int32
key	{ 1 field }	Object
_id	1	Int32
name	_id_	String

12. Cree un índice para el campo productName. Busque los las compras que tengan en el nombre del producto el string "11" y utilice el método explain("executionStats") al final de la consulta, para comparar la **cantidad de documentos examinados** y el **tiempo en milisegundos** de la consulta con y sin índice.



## Rta:

- La consulta **SIN** el índice aplicado al campo `productName` muestra las siguientes estadísticas:

```
db.purchases.find({productName: "Producto 11"}).explain("executionStats")
```

Key	Value
(1)	{ 4 fields }
queryPlanner	{ 6 fields }
executionStats	{ 6 fields }
executionSuccess	true
nReturned	2
executionTimeMillis	29
totalKeysExamined	0
totalDocsExamined	44641
executionStages	{ 13 fields }
serverInfo	{ 4 fields }
ok	1.0

Podemos observar que la cantidad de documentos escaneados corresponde con la totalidad de la colección “purchases”, y el tiempo para esta consulta fue de 29 milisegundos. Estas estadísticas fueron recolectadas usando como soporte de datos un disco SSD.

- La consulta **CON** el índice aplicado al campo `productName` muestra las siguientes estadísticas:

```
db.purchases.find({productName: "Producto 11"}).explain("executionStats")
```

Key	Value
(1)	{ 4 fields }
queryPlanner	{ 6 fields }
executionStats	{ 6 fields }
executionSuccess	true
nReturned	2
executionTimeMillis	0
totalKeysExamined	2
totalDocsExamined	2
executionStages	{ 13 fields }
serverInfo	{ 4 fields }
ok	1.0

Podemos observar que la cantidad de documentos escaneados de la colección “purchases” es 2, y el tiempo para esta consulta fue de 0 milisegundos. Estas estadísticas fueron recolectadas usando como soporte de datos un disco SSD.

- Busque las compras enviadas dentro de la ciudad de Buenos Aires. Para esto, puede definir una variable en la terminal y asignarle como valor el polígono del archivo provisto `caba.geojson` (copiando y pegando directamente). Cree un índice geoespacial de tipo `2dsphere` para el campo `location` de la colección `purchases` y, de la misma forma que en el punto 12, compare la performance de la consulta con y sin dicho índice.



- La consulta **SIN** el índice aplicado al campo location muestra las siguientes estadísticas:

```
db.purchases.find({location: {$geoWithin: {$geometry: cabaLocation}}}).explain("executionStats")
```

0.116 sec.

Key	Value
(1)	{ 4 fields }
queryPlanner	{ 6 fields }
executionStats	{ 6 fields }
executionSuccess	true
nReturned	9944
executionTimeMillis	115
totalKeysExamined	0
totalDocsExamined	45019
executionStages	{ 13 fields }
serverInfo	{ 4 fields }
ok	1.0

Podemos observar que la cantidad de documentos escaneados corresponde con la totalidad de la colección “purchases”, y el tiempo para esta consulta fue de 115 milisegundos. Estas estadísticas fueron recolectadas usando como soporte de datos un disco SSD.

- La consulta **CON** el índice aplicado al campo location muestra las siguientes estadísticas:

```
db.purchases.find({location: {$geoWithin: {$geometry: cabaLocation}}}).explain("executionStats")
```

0.08 sec.

Key	Value
(1)	{ 4 fields }
queryPlanner	{ 6 fields }
executionStats	{ 6 fields }
executionSuccess	true
nReturned	9944
executionTimeMillis	79
totalKeysExamined	12586
totalDocsExamined	12568
executionStages	{ 14 fields }
serverInfo	{ 4 fields }
ok	1.0

Podemos observar que la cantidad de documentos escaneados de la colección “purchases” es de 12568, y el tiempo para esta consulta fue de 79 milisegundos. Estas estadísticas fueron recolectadas usando como soporte de datos un disco SSD.

## Parte 4: Aggregation Framework

- MongoDB cuenta con un Aggregation Framework que brinda la posibilidad de hacer analítica en tiempo real del estilo OLAP (Online Analytical Processing), de forma similar a otros productos específicos como Hadoop o MapReduce. En los siguientes ejercicios se verán algunos ejemplos de su aplicabilidad.

14. Obtenga 5 productos aleatorios de la colección.

Esto puede ser logrado utilizando la función `$sample` del framework de aggregation (cuyo punto de entrada es la función `aggregate`). En nuestro caso, para seleccionar 5 es:

→ `db.mycoll.aggregate([{$sample: {size: 5}}])`

```
db.purchases.aggregate([{$sample: {size: 5}}])
```

purchases 0.084 sec.

Key	Value
(1) ObjectId("60b4025b6c60f9e0579ab691") <ul style="list-style-type: none"> <li>_id</li> <li>productName</li> <li>shippingCost</li> <li>location</li> </ul>	{ 4 fields } <ul style="list-style-type: none"> <li>ObjectId("60b4025b6c60f9e0579ab691")</li> <li>Producto 7577</li> <li>600.0</li> <li>{ 2 fields }</li> </ul>
(2) ObjectId("60b402596c60f9e0579aa137") <ul style="list-style-type: none"> <li>_id</li> <li>productName</li> <li>shippingCost</li> <li>location</li> </ul>	{ 4 fields } <ul style="list-style-type: none"> <li>ObjectId("60b402596c60f9e0579aa137")</li> <li>Producto 1342</li> <li>1100.0</li> <li>{ 2 fields }</li> </ul>
(3) ObjectId("60b402646c60f9e0579b08a1") <ul style="list-style-type: none"> <li>_id</li> <li>productName</li> <li>shippingCost</li> <li>location</li> </ul>	{ 4 fields } <ul style="list-style-type: none"> <li>ObjectId("60b402646c60f9e0579b08a1")</li> <li>Producto 30682</li> <li>1300.0</li> <li>{ 2 fields }</li> </ul>
(4) ObjectId("60b402616c60f9e0579aeca1") <ul style="list-style-type: none"> <li>_id</li> <li>productName</li> <li>shippingCost</li> <li>location</li> </ul>	{ 4 fields } <ul style="list-style-type: none"> <li>ObjectId("60b402616c60f9e0579aeca1")</li> <li>Producto 22807</li> <li>900.0</li> <li>{ 2 fields }</li> </ul>
(5) ObjectId("60b402636c60f9e0579affec") <ul style="list-style-type: none"> <li>_id</li> <li>productName</li> <li>shippingCost</li> <li>location</li> </ul>	{ 4 fields } <ul style="list-style-type: none"> <li>ObjectId("60b402636c60f9e0579affec")</li> <li>Producto 28171</li> <li>1200.0</li> <li>{ 2 fields }</li> </ul>

15. Usando el framework de agregación, obtenga las compras que se hayan enviado a 1km (o menos) del centro de la ciudad de Buenos Aires ([ -58.4586, -34.5968 ]) y guárdelas en una nueva colección.

En este caso, utilizando el operador [\\$geoNear](#), podremos obtener los purchases que se encuentran a menos de un km de distancia de las coordenadas expresadas en *\$near*. La distancia se anota en metros, en el operador *maxDistance*. En la captura siguiente, agregamos *\$count*: "location" para saber la cantidad de purchases que coinciden con nuestra búsqueda:

```
db.purchases.aggregate([
  {$geoNear: {
    includeLocs: "location",
    distanceField: "location",
    near: {type: 'Point', coordinates: [-58.4586,-34.5968]},
    maxDistance: 1000,
    spherical: true
  }},
  {$count: "location"}
])
```

purchases 0.005 sec.

Key	Value
(1) <ul style="list-style-type: none"> <li>location</li> </ul>	{ 1 field } <ul style="list-style-type: none"> <li>225</li> </ul>

En la siguiente captura, mostramos como guardamos los resultados anteriores en una colección llamada `purchasesInCaba`, utilizando el operador [\\$out](#):

```
use test

db.purchases.aggregate([
  {$geoNear: {
    includeLocs: "location",
    distanceField: "location",
    near: {type: 'Point', coordinates: [-58.4586,-34.5968]},
    maxDistance: 1000,
    spherical: true
  }},
  {$out: "purchasesInCaba"}
])
```

16. Obtenga una colección de los productos que fueron enviados en las compras del punto anterior. Note que sólo es posible ligarlas por el nombre del producto.

► Si la consulta se empieza a tornar difícil de leer, se pueden ir guardando los agregadores en variables, que no son más que objetos en formato JSON.

En la siguiente captura obtenemos la colección de productos mencionada (utilizando la función [\\$project](#) para seleccionar solo el `productName`), y la llamamos en este caso “`productsSoldInCaba`”:

```
use test

db.purchasesInCaba.aggregate([
  {$project: {_id: 0, productName: 1}},
  {$out: "productsSoldInCaba"}
])
```

17. Usando la colección del punto anterior, obtenga una nueva en la que agrega a cada producto un atributo *purchases* que consista en un array con todas las compras de cada producto.

En este inciso, para que el [\\$lookup](#) de nuestra consulta sea mas eficiente, creamos un índice en `productName`.

```
use test

db.purchases.createIndex({ productName : 1 })

db.productsSoldInCaba.aggregate([
  {$lookup:
    {
      from: "purchases",
      localField: "productName",
      foreignField: "productName",
      as: "purchases"
    }
  },
])
```

18. Obtenga el promedio de costo de envío pagado para cada producto del punto anterior.

En la siguiente captura mostramos, partiendo de la query del punto anterior, de qué forma podemos obtener el promedio de costo de envío para cada producto, utilizando primero el metodo [\\$unwind](#) para “desempaquetar” los arrays de compras, y luego el operador [\\$avg](#) para calcular el promedio:

```
use test

db.purchases.createIndex({ productName : 1 })

db.productsSoldInCaba.aggregate([
  {$lookup:
    {
      from: "purchases",
      localField: "productName",
      foreignField: "productName",
      as: "purchases"
    }
  },
  {$unwind: "$purchases"},
  {$group: {_id: "$productName", avg: {$avg: "$purchases.shippingCost"}}}
])
```