



**UNIVERSIDAD NACIONAL EXPERIMENTAL DE GUAYANA**  
**VICERRECTORADO ACADÉMICO**  
**COORDINACIÓN GENERAL DE PREGRADO**  
**PROYECTO DE CARRERA: INGENIERÍA INFORMÁTICA**  
**SISTEMAS DE BASE DE DATOS II**

**PROYECTO MONGODB**  
**GLOBALMARKET ANALYTICS & SEARCH ENGINE**

Docente:

Clinia Cordero

Integrantes:

Centeno Fernando

Longart Juan

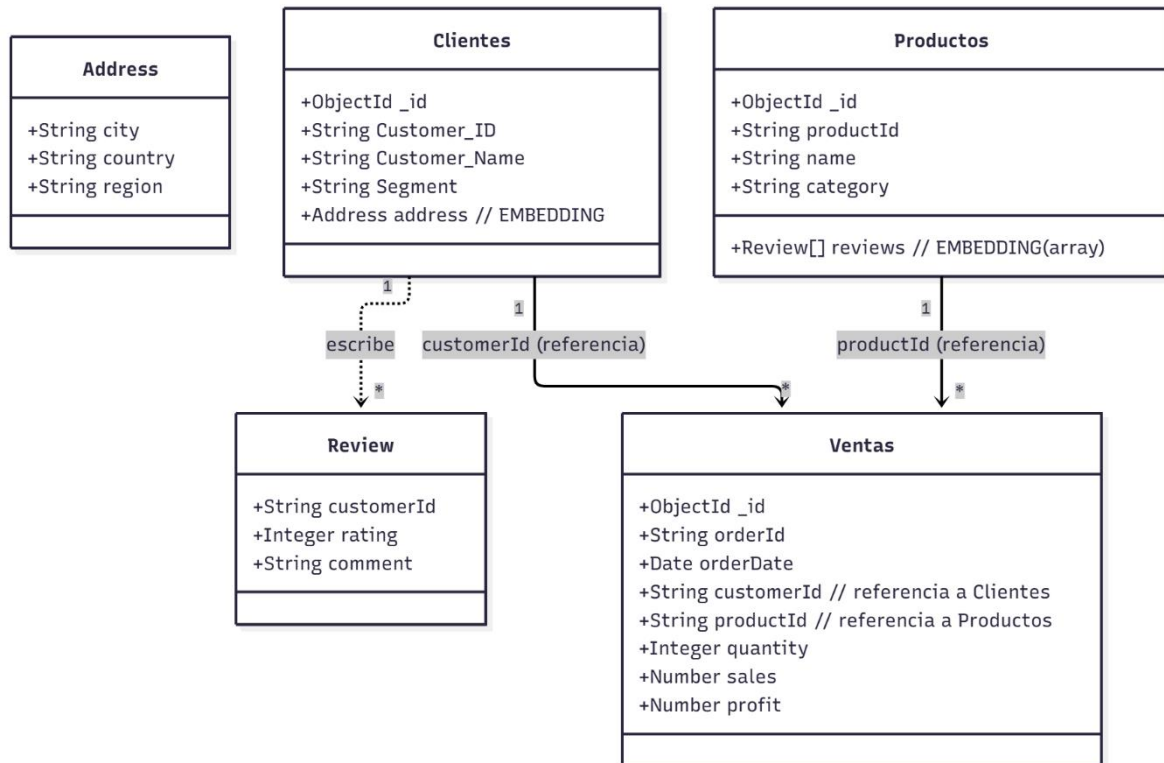
Rodriguez Rafael

Ciudad Guayana, diciembre de 2025

## A. Modelado y Validación (Schema Design)

No se trata solo de importar el JSON/CSV. Los alumnos deben:

### 1. Diseñar el Esquema



**Customers (colección):** La colección customers representa a los clientes del sistema: cada documento almacena la identificación y segmentación del cliente.

- Customer ID: identificador único de negocio del cliente, usado por orders.customerId para referenciarlo.
- Customer Name: nombre del cliente, utilizado en reportes y listados.
- Segment: categoría de cliente (por ejemplo, consumidor, corporativo), útil para análisis de ventas por tipo de cliente.

**Address (embedding):** address se modeló como un subdocumento embebido dentro de customers, agrupando la información geográfica del cliente.

- address.city, address.state, address.country: localización del cliente a nivel de ciudad, estado/provincia y país.
- address.market, address.region, address.postalCode: campos para análisis por mercado/región y para cubrir detalles logísticos.

Se eligió embebido porque la dirección solo pertenece a ese cliente, se consulta junto con sus datos básicos y no se comparte, lo que evita otra colección y consultas adicionales.

**Products (colección):** La colección products almacena la información maestra de cada producto, sin depender de las órdenes.

- productId: identificador único de producto, referenciado desde orders.productId.
- name: nombre comercial del producto.
- category, subCategory: clasificación del producto para análisis por tipo de artículo.
- unitPrice: precio base por unidad del producto, que permite relacionar descuentos y ventas con el valor original.

**reviews (embedding):** En products se decidió embeber las reseñas en el campo reviews, un array de subdocumentos. Cada elemento de reviews incluye:

- customerId: cliente que realizó la reseña.
- rating: calificación numérica del producto (por ejemplo, de 1 a 5).
- comment: texto asociado a la reseña.
- createdAt: fecha en la que se generó la reseña.

Este diseño permite calcular directamente, sobre el mismo documento, métricas como cantidad de reseñas y calificación promedio sin otra colección ni joins adicionales.

**Orders (colección):** La colección orders representa las transacciones de venta; cada documento es una línea de pedido que conecta un cliente con un producto en una fecha concreta.

- orderId: identificador de la orden, permite agrupar varias líneas de un mismo pedido.
- orderDate, shipDate: fecha en que se realiza la orden y fecha de envío, usadas para análisis temporales y de logística.
- Ship Mode: modo de envío (por ejemplo, estándar, urgente).
- orderPriority: prioridad asignada a la orden (baja, media, alta).

Relaciones con otras colecciones:

- customerId: referencia lógica al cliente en customers (Customer ID).
- productId: referencia lógica al producto en products (productId).

Datos económicos de la línea:

- quantity: número de unidades del producto vendidas en esa línea.
- sales: importe total facturado por la línea, resultado de precio unitario con descuentos por la cantidad.
- discount: proporción de descuento aplicada (por ejemplo, 0.4 equivale a 40% de descuento sobre el precio base).
- profit: beneficio o pérdida generada por esa venta después de costes.
- shippingCost: coste de envío imputado a la orden.

2. Schema Validation: Implementar reglas de validación JSON en Atlas para asegurar la integridad de los datos (ej. que el precio sea siempre positivo, que el email del usuario tenga formato correcto).

## validation.json

### Validación Orders

```
1 // 1) VALIDACIÓN PARA ORDERS
2
3 db.runCommand({
4   collMod: "orders",
5   validator: {
6     $jsonSchema: {
7       bsonType: "object",
8       required: ["orderId", "orderDate", "customerId", "productId", "sales", "quantity"],
9       properties: {
10         orderId: { bsonType: "string" },
11         orderDate: { bsonType: "date" },
12         shipDate: { bsonType: "date" },
13         customerId: { bsonType: "string" },
14         productId: { bsonType: "string" },
15         sales: {
16           bsonType: ["double", "int", "decimal"],
17           minimum: 0
18         },
19         quantity: {
20           bsonType: ["int", "long"],
21           minimum: 1
22         },
23         discount: {
24           bsonType: ["double", "int", "decimal"],
25           minimum: 0
26         },
27         profit: {
28           bsonType: ["double", "int", "decimal"]
29         },
30         shippingCost: {
31           bsonType: ["double", "int", "decimal"],
32           minimum: 0
33         },
34         orderPriority: { bsonType: "string" },
35         shipMode: { bsonType: "string" }
36       }
37     }
38   },
39   validationLevel: "moderate"
40 })
```

Ese comando define la **validación de esquema** de la **colección orders** usando \$jsonSchema.

### Qué hace esta validación

- Obliga a que cada documento de orders sea un objeto con campos requeridos: orderId, orderDate, customerId, productId, sales y quantity.
- Define tipos y restricciones:
  - orderId, customerId, productId, orderPriority, shipMode: deben ser string.
  - orderDate, shipDate: deben ser fechas (date).

- sales, discount, profit, shippingCost: numéricos (double, int o decimal), con mínimo 0 en ventas, descuento y coste de envío.
- quantity: numérico entero (int o long) y al menos 1.

### Efecto en la colección orders

- Con validationLevel: "moderate", los documentos antiguos se mantienen, pero cualquier **insert o update nuevo** que no cumpla estas reglas será rechazado por MongoDB.
- Esto garantiza que todas las órdenes futuras tengan datos coherentes (fechas bien tipadas, cantidades positivas, descuentos no negativos, etc.).

### Validación Customers

```

42 // 2) VALIDACIÓN PARA CUSTOMERS
43 db.runCommand({
44   collMod: "customers",
45   validator: {
46     $jsonSchema: {
47       bsonType: "object",
48       required: ["Customer ID", "Customer Name"],
49       properties: {
50         "Customer ID": { bsonType: "string" },
51         "Customer Name": { bsonType: "string" },
52         Segment: { bsonType: "string" },
53
54         // Dirección embebida
55         address: {
56           bsonType: "object",
57           properties: {
58             city: { bsonType: "string" },
59             state: { bsonType: "string" },
60             country: { bsonType: "string" },
61             postalCode: { bsonType: ["string", "int"] },
62             market: { bsonType: "string" },
63             region: { bsonType: "string" }
64           }
65         }
66       }
67     },
68   },
69   validationLevel: "moderate"
70 })

```

Ese comando configura la **validación de esquema para la colección customers**, controlando datos básicos y la dirección embebida.

### **Qué exige el esquema de customers**

- El documento debe ser un objeto con campos obligatorios:
  - "Customer ID": tipo string.
  - "Customer Name": tipo string.
- Segment también debe ser string, para garantizar que la clasificación del cliente tenga el tipo correcto.

### **Validación del subdocumento address**

- address debe ser un objeto que agrupa los datos de ubicación del cliente:
  - city, state, country, market, region: todos de tipo string.
  - postalCode: puede ser string o int, para soportar códigos postales numéricos o con letras.
- Con validationLevel: "moderate", los documentos existentes se respetan, pero cualquier nuevo insert/update debe cumplir esta estructura, asegurando consistencia en los datos de clientes y su dirección embebida.

## Validación Products

```
72 // 3) VALIDACIÓN PARA PRODUCTS
73 db.runCommand({
74   collMod: "products",
75   validator: {
76     $jsonSchema: {
77       bsonType: "object",
78       required: ["productId", "name"],
79       properties: {
80         productId: { bsonType: "string" },
81         name: { bsonType: "string" },
82         category: { bsonType: "string" },
83         subCategory: { bsonType: "string" },
84
85         // Reseñas embebidas (embedding)
86         reviews: {
87           bsonType: "array",
88           items: {
89             bsonType: "object",
90             required: ["customerId", "rating"],
91             properties: {
92               customerId: { bsonType: "string" },
93               rating: { bsonType: "int", minimum: 1, maximum: 5 },
94               comment: { bsonType: "string" },
95               createdAt: { bsonType: "date" }
96             }
97           }
98         }
99       }
100     }
101   },
102   validationLevel: "moderate"
103 })
104
```

Ese comando define la **validación de esquema** para la colección **products**, incluyendo las reseñas embebidas.

### Qué valida en products

- Requiere que cada documento tenga al menos:
  - productId: tipo string, identificador único del producto.
  - name: tipo string, nombre del producto.



- También define como string los campos de clasificación:
  - category, subCategory.

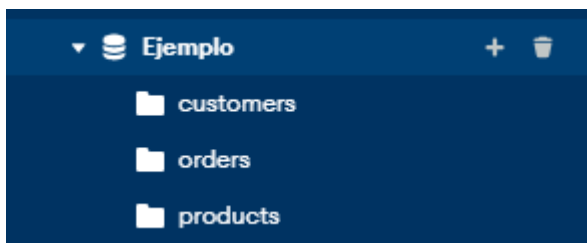
### Validación del array reviews (embedding)

- reviews debe ser un **array de objetos**; cada elemento del array es una reseña.
- Cada reseña (items) debe cumplir:
  - Tipo object y campos requeridos customerId y rating.
  - customerId: string, identifica al cliente que reseña.
  - rating: int entre 1 y 5 (minimum: 1, maximum: 5).
  - comment: string, texto de la reseña (opcional).
  - createdAt: date, fecha de creación de la reseña.

Con validationLevel: "moderate", los documentos existentes se mantienen, pero cualquier nueva inserción o actualización en products debe respetar tanto los campos básicos como la estructura y rango de las reseñas embebidas.

3. Ingesta: Cargar los datos a un clúster M0 en Atlas (usando mongoimport o MongoDB Compass).

### Exportación



mongo@ejemplo.7j7h... customers orders Ejemplo products +

ejemplo.7j7h2.mongodb.net > Ejemplo > customers Open MongoDB shell

Documents **19K** Aggregations Schema Indexes 1 Validation

Type a query: { field: 'value' } or [Generate query](#) Explain Reset Find Options

ADD DATA EXPORT DATA UPDATE DELETE 100 1 - 100 of 1917

```
{
  "_id": ObjectId("692c6a2801cba78d23971c23"),
  "Customer ID": "SC-20725",
  "Customer Name": "Steven Cartwright",
  "Segment": "Consumer",
  "address": {
    "city": "Tegucigalpa",
    "state": "Francisco Morazán",
    "country": "Honduras",
    "market": "LATAM",
    "region": "Central"
  }
}
```

```
{
  "_id": ObjectId("692c6a2801cba78d23971c24"),
  "Customer ID": "RR-19315",
  "Customer Name": "Ralph Ritter",
  "Segment": "Consumer",
  "address": {
    "city": "Aurora",
    "state": "Illinois",
    "country": "United States",
    "postalCode": 60505,
    "market": "US",
    "region": "Central"
  }
}
```

```
{
  "_id": ObjectId("692c6a2801cba78d23971c25"),
  "Customer ID": "TH-21235",
  "Customer Name": "Tiffany House",
  "Segment": "Corporate",
  "address": {}
}
```

```
{
  "_id": ObjectId("692c6a2801cba78d23971c26"),
  "Customer ID": "RA-19915",
  "Customer Name": "Russell Applegate",
  "Segment": "Consumer",
  "address": {}
}
```

mongo@ejemplo.7j7h... customers orders Ejemplo products +

ejemplo.7j7h2.mongodb.net > Ejemplo > products Open MongoDB shell

Documents **18K** Aggregations Schema Indexes 1 Validation

Type a query: { field: 'value' } or [Generate query](#) Explain Reset Find Options

ADD DATA EXPORT DATA UPDATE DELETE 100 1 - 100 of 1771

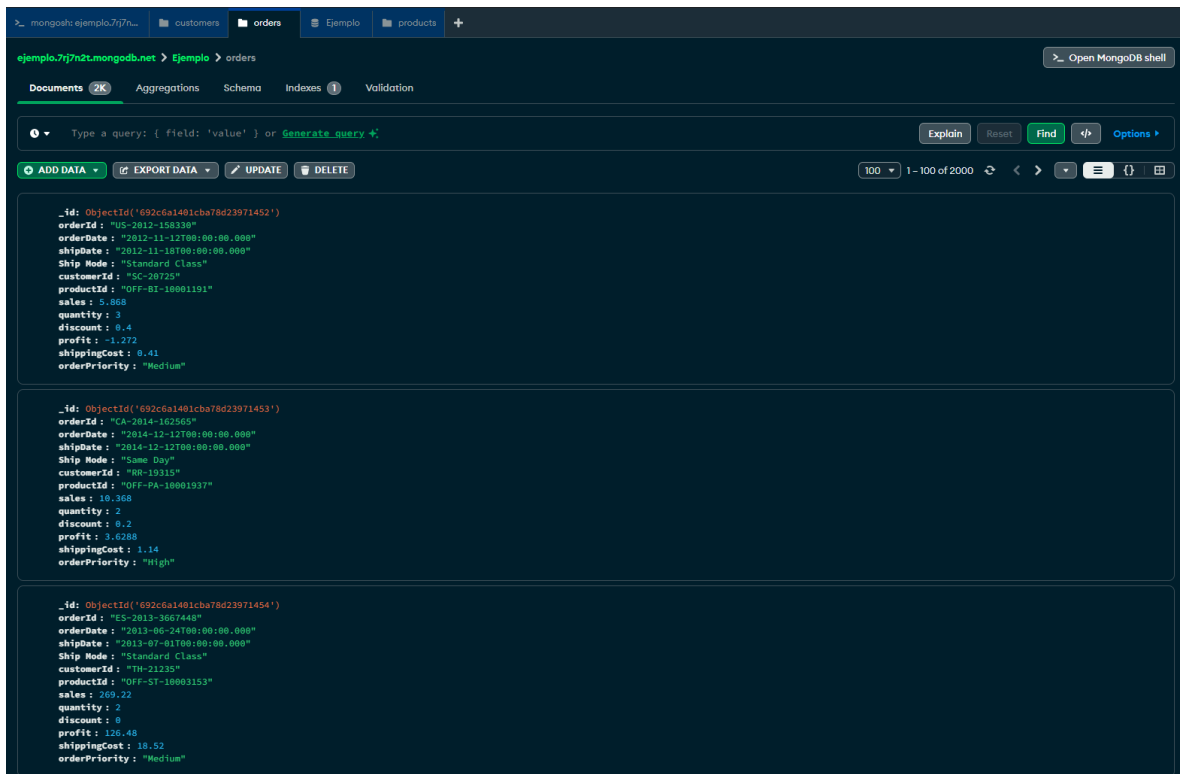
```
{
  "_id": ObjectId("692cdc7501cba78d23972a96"),
  "productid": "OFF-BI-10001191",
  "category": "Office Supplies",
  "subCategory": "Binders",
  "name": "Yellow Jones Hole Reinforcements, Recycled",
  "reviews": Array(50),
  "unitPrice": 3.26
}
```

```
{
  "_id": ObjectId("692cdc7501cba78d23972a97"),
  "productid": "OFF-PA-10001937",
  "category": "Office Supplies",
  "subCategory": "Paper",
  "name": "Xerox 21",
  "reviews": Array(50),
  "unitPrice": 6.48
}
```

```
{
  "_id": ObjectId("692cdc7501cba78d23972a98"),
  "productid": "OFF-ST-10003153",
  "category": "Office Supplies",
  "subCategory": "Storage",
  "name": "Toner File Cart, Single Width",
  "reviews": Array(50),
  "unitPrice": 134.61
}
```

```
{
  "_id": ObjectId("692cdc7501cba78d23972a99"),
  "productid": "OFF-AR-10001079",
  "category": "Office Supplies",
  "subCategory": "Art",
  "name": "Model L Table or Wall-Mount Pencil Sharpener",
  "reviews": Array(50),
  "unitPrice": 17.99
}
```

```
{
  "_id": ObjectId("692cdc7501cba78d23972a9a"),
  "productid": "OFF-AR-10003551",
  "category": "Office Supplies",
  "subCategory": "Art",
  "name": "Sanford Pens, Water Color",
  "reviews": Array(50),
  "unitPrice": 9.52
}
```



## B. Consultas Avanzadas (Aggregation Framework)

Deben construir un "Pipeline de Agregación" para responder preguntas de negocio.

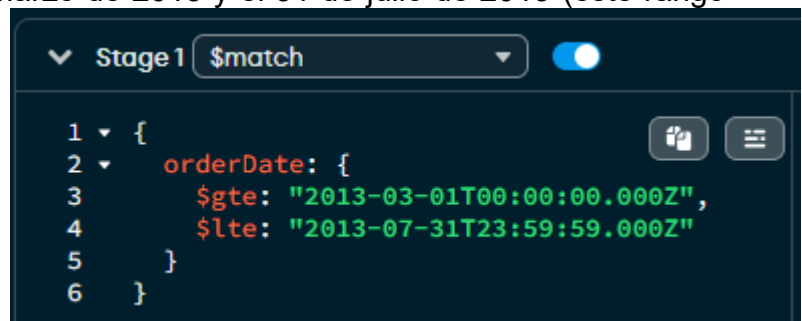
Se requieren al menos 3 stages complejos:

**Reporte de Ventas: Calcular el total de ventas por categoría y por mes.**

(Uso de `$match`, `$group`, `$project`).

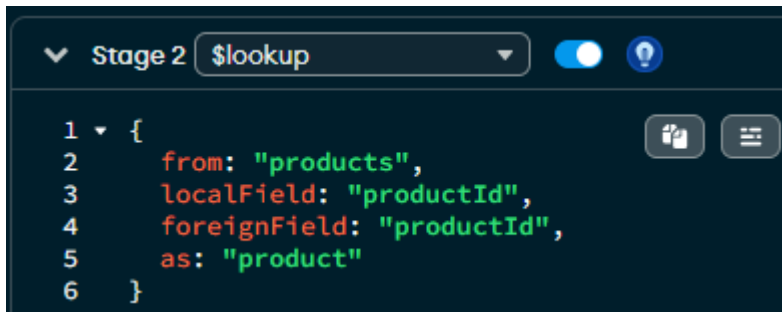
Ese pipeline obtiene, para un rango de fechas, cuánto se vendió por categoría y por mes, y además qué productos se vendieron en cada categoría/mes.

**Stage 1 – `$match`:** rango de fechas: Filtra las órdenes y deja solo las que tienen `orderDate` entre el 1 de marzo de 2013 y el 31 de julio de 2013 (este rango de tiempo se puede modificar)



## Stage 2 – \$lookup: traer datos de producto

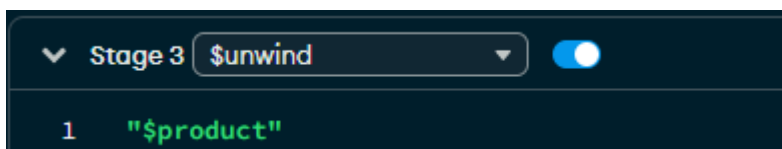
Hace un “join” lógico: por cada orden busca en products el documento cuyo productId coincide, y lo mete en un array product.



```
1 {  
2   from: "products",  
3   localField: "productId",  
4   foreignField: "productId",  
5   as: "product"  
6 }
```

## Stage 3 – \$unwind: pasar de array a objeto

Como product es un array, \$unwind lo “explota”: cada orden pasa a tener product como un solo objeto con los datos del producto, listo para usar "\$product.category", "\$product.name", etc.



```
1 "$product"
```

## Stage 4 – \$group: agrupar y calcular

- `_id.category`: agrupa por categoría del producto.
- `_id.year` y `_id.month`: sacan año y mes desde el string `orderDate` con `$substr` y los convierten a entero con `$toInt`.
- `totalVentas`: suma el campo `sales` de todas las órdenes que caen en ese grupo (esa categoría, ese año, ese mes).
- `productos`: usa `$addToSet` para guardar un set de productos distintos (id y nombre) vendidos en esa categoría/mes, sin duplicados.

```
Stage 4 $group ☒

1 {
2   _id: {
3     category: "$product.category",
4     year: {
5       $toInt: {
6         $substr: ["$orderDate", 0, 4]
7       }
8     },
9     month: {
10      $toInt: {
11        $substr: ["$orderDate", 5, 2]
12      }
13    }
14  },
15  totalVentas: {
16    $sum: "$sales"
17  },
18  productos: {
19    $addToSet: {
20      productId: "$product.productId",
21      name: "$product.name"
22    }
23  }
24 }
```

### Stage 5 – \$sort: ordenar el reporte

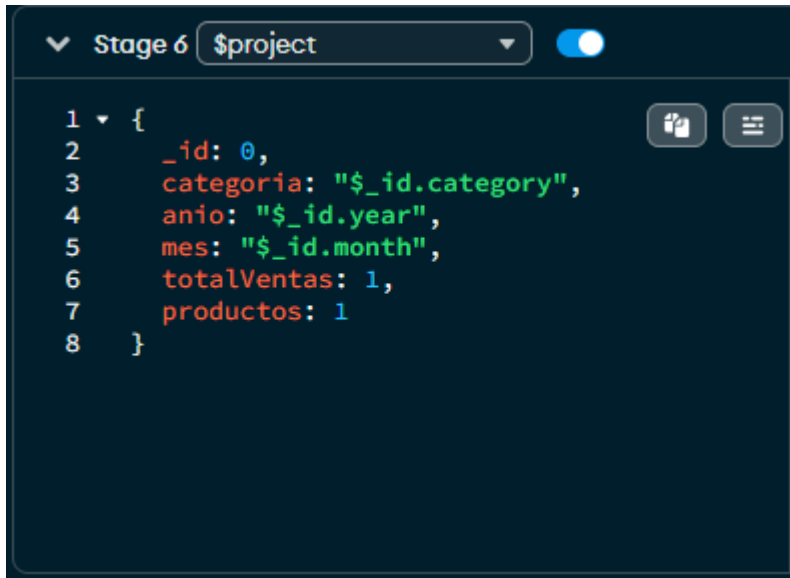
Ordena los grupos primero por año ascendente, luego por mes, y dentro de cada mes por categoría alfabéticamente.

```
Stage 5 $sort ☒

1 {
2   "_id.year": 1,
3   "_id.month": 1,
4   "_id.category": 1
5 }
```

## Stage 6 – \$project: salida limpia

Renombra y deja solo los campos que interesan en el resultado: categoría, año, mes, total de ventas y lista de productos, ocultando \_id.



```
1 {
2   _id: 0,
3   categoria: "$_id.category",
4   anio: "$_id.year",
5   mes: "$_id.month",
6   totalVentas: 1,
7   productos: 1
8 }
```

## Conclusión:

Este pipeline genera un reporte de ventas por categoría y mes en un periodo dado, mostrando:

- Para cada combinación de categoría-año-mes, el total vendido (totalVentas).
- El conjunto de productos distintos que se vendieron en esa categoría y mes (productos).

Sirve para contestar preguntas de negocio tipo: “Entre marzo y julio de 2013, ¿cuánto vendí por categoría cada mes y qué productos se movieron en cada categoría?”.

## Top Clientes/Productos: Identificar los productos con mejor calificación promedio que tengan más de 50 reseñas.

Ese pipeline saca el top de productos mejor valorados, solo considerando los que tienen muchas reseñas (50 o más).

### Stage 1 – \$addFields: calcular métricas de reseñas

Añade dos campos calculados a cada producto:

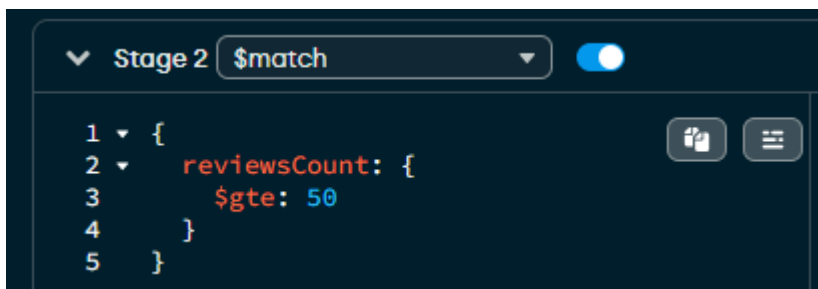
- reviewsCount: número de reseñas en el array reviews.
- avgRating: promedio de las calificaciones rating dentro de reviews.



```
1 {
2   reviewsCount: {
3     $size: "$reviews"
4   },
5   avgRating: {
6     $avg: "$reviews.rating"
7   }
8 }
```

### Stage 2 – \$match: filtrar productos con muchas reseñas

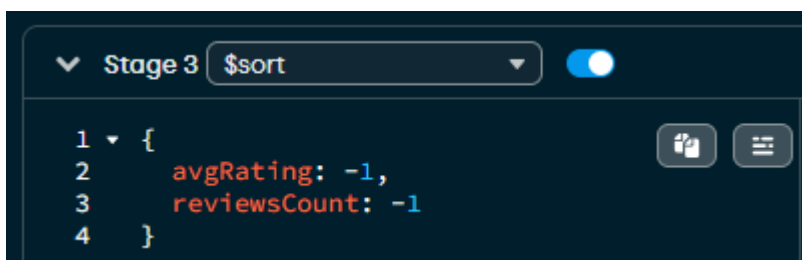
Conserva solo los productos cuyo reviewsCount es mayor o igual a 50, es decir, con suficiente volumen de reseñas.



```
1 {
2   reviewsCount: {
3     $gte: 50
4   }
5 }
```

### Stage 3 – \$sort: ordenar por calidad y volumen

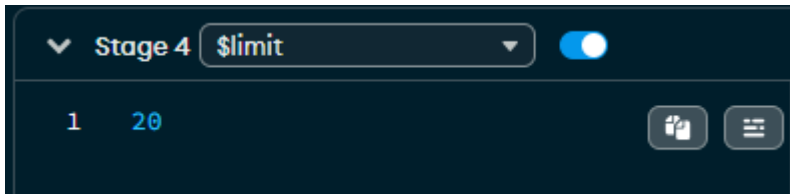
Ordena los productos primero por mejor promedio de rating (de mayor a menor) y, en caso de empate, por cantidad de reseñas (más reseñas primero).



```
1 {
2   avgRating: -1,
3   reviewsCount: -1
4 }
```

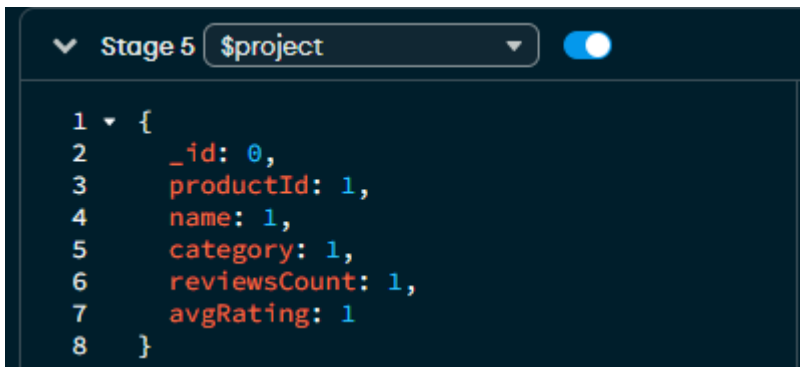
#### Stage 4 – \$limit: quedarse con el top N

Limita el resultado a los primeros 20 productos, quedándote con el top 20 según el orden anterior.



#### Stage 5 – \$project: salida limpia para reporte

Muestra: solo productId, name, category, reviewsCount y avgRating, ocultando \_id, para que el resultado sea legible en el informe.



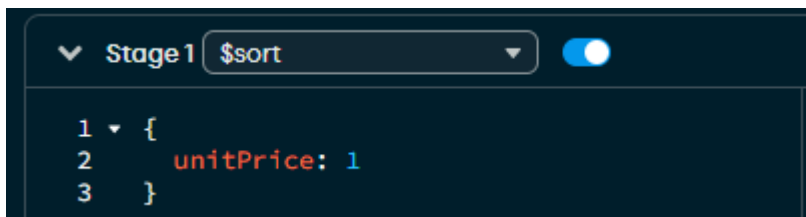
#### Conclusión:

Este pipeline responde a la pregunta de negocio: “¿Cuáles son los productos mejor calificados por los clientes, considerando solo aquellos con suficiente volumen de reseñas ( $\geq 50$ ) para que el promedio sea confiable?”.



**Bucket Pattern: Agrupar productos por rangos de precios (Bajo, Medio, Alto) usando \$bucket o \$bucketAuto.**

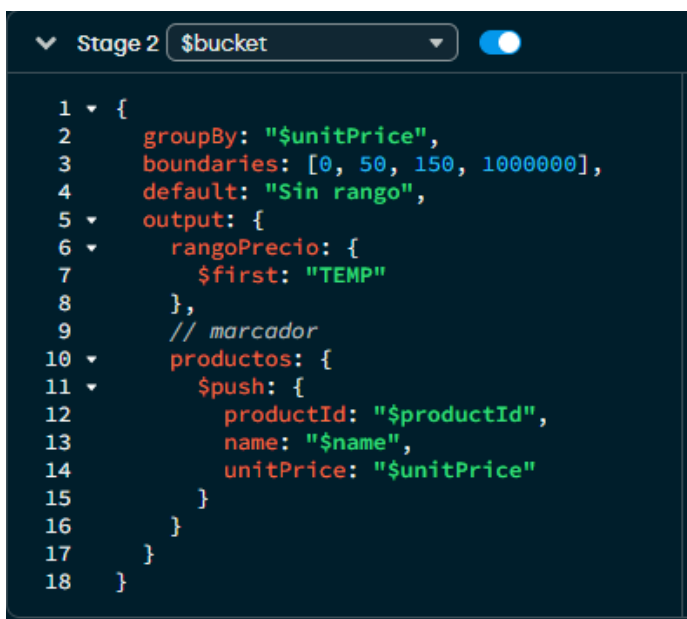
**Stage 1 – \$sort:** ordenar por precio:  
Ordena todos los documentos de products por unitPrice ascendente para que, al agrupar, los productos dentro de cada rango queden ya de menor a mayor precio.



```
1 {  
2   unitPrice: 1  
3 }
```

**Stage 2 – \$bucket: agrupar en rangos de precio**

Crea buckets según unitPrice con límites [0, 50, 150, 1000000], acumulando en productos los productos que caen en cada intervalo (con productId, name y unitPrice).



```
1 {  
2   groupBy: "$unitPrice",  
3   boundaries: [0, 50, 150, 1000000],  
4   default: "Sin rango",  
5   output: {  
6     rangoPrecio: {  
7       $first: "TEMP"  
8     },  
9     // marcador  
10    productos: {  
11      $push: {  
12        productId: "$productId",  
13        name: "$name",  
14        unitPrice: "$unitPrice"  
15      }  
16    }  
17  }  
18 }
```

### Stage 3 – \$sort: ordenar los buckets

Ordena los buckets por `_id` (el límite inferior del rango) de forma ascendente, garantizando que los rangos queden en el orden correcto:  $0 \rightarrow 50 \rightarrow 150$ .

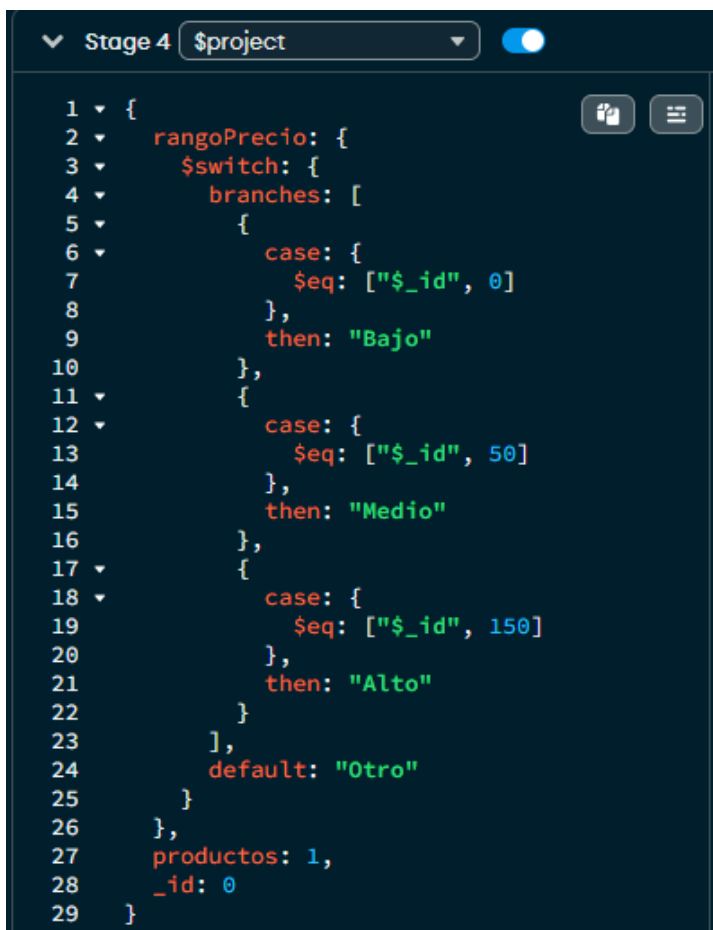


The screenshot shows the MongoDB Atlas interface for Stage 3. The stage is named '\$sort' and is enabled. The pipeline stage is defined as follows:

```
1 {
2   _id: 1
3 }
```

### Stage 4 – \$project: etiquetar rangos y limpiar salida

Reemplaza `_id` con una etiqueta de texto usando `$switch` (Bajo, Medio, Alto según el límite del bucket), mantiene la lista de productos y oculta `_id`, dejando un resultado legible por rango.



The screenshot shows the MongoDB Atlas interface for Stage 4. The stage is named '\$project' and is enabled. The pipeline stage is defined as follows:

```
1 {
2   rangoPrecio: {
3     $switch: {
4       branches: [
5         {
6           case: {
7             $eq: ["$_id", 0]
8           },
9           then: "Bajo"
10        },
11        {
12          case: {
13            $eq: ["$_id", 50]
14          },
15          then: "Medio"
16        },
17        {
18          case: {
19            $eq: ["$_id", 150]
20          },
21          then: "Alto"
22        }
23      ],
24      default: "Otro"
25    }
26  },
27  productos: 1,
28  _id: 0
29 }
```

## Conclusión:

Este pipeline permite analizar productos por rangos de precio, viendo para cada rango (“Bajo”, “Medio”, “Alto”) qué productos pertenecen a ese intervalo y en qué orden de precio, útil para estudiar oferta por segmentos de precio o armar reportes de catálogo segmentado.

## C. Búsqueda y Rendimiento (Atlas Search & Indexing)

**1. Atlas Search: Implementar un índice de búsqueda (Lucene) para permitir búsqueda difusa (fuzzy search) por nombre de producto o descripción. Esto simula la barra de búsqueda de la tienda.**

Search index que implementa Lucene para permitir la búsqueda difusa:

```
1 {
2   "mappings": {
3     "dynamic": false,
4     "fields": {
5       "name": {
6         "type": "string",
7         "analyzer": "lucene.standard",
8         "searchAnalyzer": "lucene.standard"
9       },
10      "description": {
11        "type": "string",
12        "analyzer": "lucene.standard",
13        "searchAnalyzer": "lucene.standard"
14      }
15    }
16  }
17 }
```


Resultado del \$search con fuzzy search implementado, muestra que al ingresar “okia” se devuelven los resultados aproximados

```
▼ Stage 1 $search
1 {
2   index: "default",
3   text: {
4     query: "okia",
5     path: "name",
6     fuzzy: {}
7   }
8 }
```

## STAGE OUTPUT

OPTIONS ▾

Sample of 10 documents

▶ **\_id:** ObjectId('692c6c7501cba78d23972e0a') 

**productId:** "TEC-PH-10002102"  
**category:** "Technology"  
**subCategory:** "Phones"  
**name:** "Nokia Headset, Cordless"  
▶ **reviews:** Array (50)  
**unitPrice:** 50.64

**\_id:** ObjectId('692c6cbf01cba78d23972fb2')  
**productId:** "TEC-PH-10001535"  
**category:** "Technology"  
**subCategory:** "Phones"  
**name:** "Nokia Speaker Phone, Cordless"  
▶ **reviews:** Array (50)  
**unitPrice:** 126

**\_id:** ObjectId('692c6cbf01cba78d23973170')  
**productId:** "TEC-NOK-10001172"  
**category:** "Technology"  
**subCategory:** "Phones"  
**name:** "Nokia Speaker Phone, VoIP"  
▶ **reviews:** Array (50)  
**unitPrice:** 123.24

**\_id:** ObjectId('692c6c7501cba78d23972c53')  
**productId:** "TEC-PH-10003945"  
**category:** "Technology"  
**subCategory:** "Phones"  
**name:** "Nokia Audio Dock, VoIP"  
▶ **reviews:** Array (50)  
**unitPrice:** 111.22

**\_id:** ObjectId('692c6cbf01cba78d239730c2')  
**productId:** "TEC-PH-10000026"  
**category:** "Technology"  
**subCategory:** "Phones"  
**name:** "Nokia Office Telephone, Cordless"  
▶ **reviews:** Array (50)  
**unitPrice:** 67.98

2. Optimización: Crear índices tradicionales (Simples y Compuestos) para acelerar las consultas frecuentes.

Name & Definition	Type	Size	Usage
> _id_	REGULAR ⓘ	196.6 kB	164 (since Sun Nov 30 2025)
> productId_1	REGULAR ⓘ	45.1 kB	1061 (since Tue Dec 02 2025)
> category_1	REGULAR ⓘ	24.6 kB	0 (since Thu Dec 04 2025)

Índices creados para acelerar las consultas

3. Análisis: Usar la herramienta "Explain Plan" para demostrar la diferencia de tiempo de respuesta antes y después de crear los índices.

### Query Performance Summary

- 359 documents returned
- 1771 documents examined
- 1 ms execution time
- Is not sorted in memory
- 0 index keys examined
- No index available for this query.**

Análisis del rendimiento

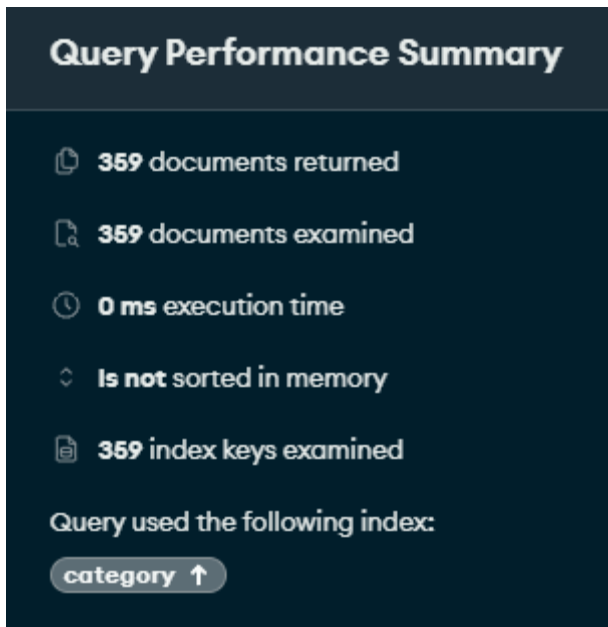
El plan explicativo muestra una etapa COLLSCAN, lo que significa que MongoDB realizó un escaneo de la colección.

Se examinaron los 1771 documentos de la colección, pero solo se devolvieron 359 documentos coincidentes.

No se utilizaron índices (totalKeysExamined es 0 y usedIndexes está vacío).

Aunque la consulta se ejecutó muy rápidamente (1 ms), es probable que esto se deba a que el conjunto de datos era pequeño y puede que no se adapte bien a medida que la colección crece.

Un escaneo completo de la colección es menos eficiente, especialmente a medida que aumenta el tamaño del conjunto de datos, y puede afectar negativamente al rendimiento en colecciones más grandes.



### Análisis del rendimiento

La consulta utiliza un escaneo de índice (IXSCAN) en el índice `category_1`, que encuentra de manera eficiente todos los documentos cuya categoría es «Tecnología».

El escaneo de índice va seguido de una etapa FETCH, que recupera los documentos reales del disco, ya que pueden ser necesarios campos adicionales que no están cubiertos por el índice.

Tanto `totalKeysExamined` como `totalDocsExamined` son iguales al recuento de documentos devueltos (359), lo que implica que no hay claves o documentos escaneados superfluos.

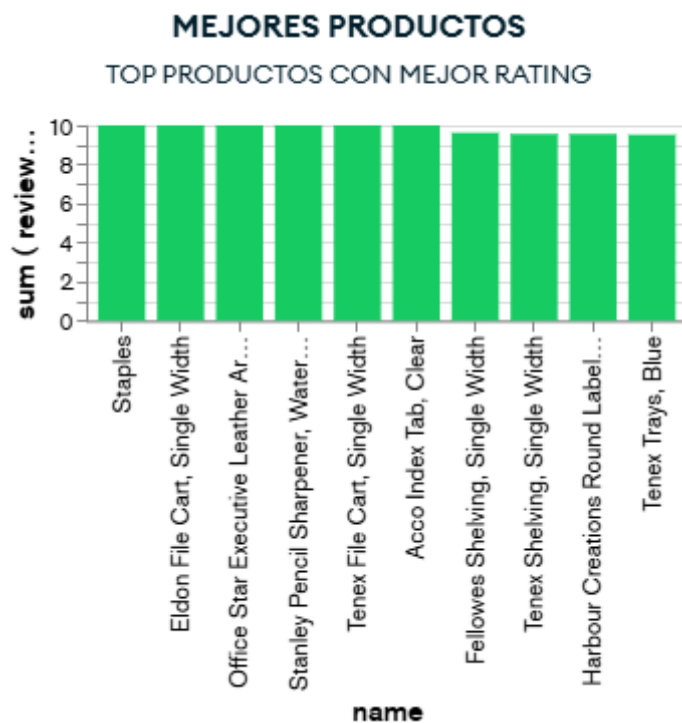
En conclusión, implementar un índice hace que la búsqueda sea mas eficiente, siendo mas notable cuando hay datos grandes, lo que implica un mejor rendimiento.

## MongoDB Charts

### 1. Gráfico de Barras: Top Productos Mejor Calificados

Este gráfico visualiza directamente la consulta de "Top Clientes/Productos". Muestra qué productos son las estrellas de la tienda.

- Colección: products
- Tipo de Gráfico: Column (Columnas)



## 2. Gráfico de Líneas: Histórico de Ventas (Tendencia)

Esencial para cualquier Dashboard de E-commerce. Responde a la lógica de "Reporte de Ventas"

Muestra las ganancias obtenidas por fecha

