Trabajo Práctico Integrador – MongoDB

Parte I – Preguntas Teóricas

1. **CRUD básicos**

    - ¿Qué diferencias hay entre `insertOne()` e `insertMany()`?

- insertOne(doc): inserta un documento.
    - Devuelve { acknowledged, insertedId }.
- insertMany([docs], { ordered }): inserta varios documentos en una sola operación
    - ordered: true (por defecto): se detiene en el primer error.
    - ordered: false: intenta seguir con los demás aunque alguno falle.
    - Devuelve { acknowledged, insertedIds }.

    - ¿Cuál es la diferencia entre `deleteOne()`, `deleteMany()` y `drop()`?

        - deleteOne(filtro): elimina un documento que coincida (el primero según el orden natural).
            - Devuelve { acknowledged, deletedCount } (0 o 1).

        - deleteMany(filtro): elimina todos los documentos que coincidan.
            - Devuelve { acknowledged, deletedCount }.

        - drop() (sobre la colección): borra la colección entera, sus índices y metadatos.
            - Devuelve true/false.

    - Explica la diferencia entre `updateOne()` y `updateMany()`.
    -
        - updateOne(filtro, actualización, opciones): actualiza un documento que coincida.
        - updateMany(filtro, actualización, opciones): actualiza todos los que coincidan.

- ¿Qué devuelve un `updateOne()` cuando no encuentra ningún documento que cumpla la condición?

  - Sin upsert (por defecto):
    { acknowledged: true, matchedCount: 0, modifiedCount: 0, upsertedId: null } (no error, solo que no coincidió nada).

  - Con upsert: true: inserta un nuevo documento y retorna, por ejemplo:
    { acknowledged: true, matchedCount: 0, modifiedCount: 0, upsertedId: ObjectId("...") }.

2. **Consultas y find()**

   - ¿Cuál es la diferencia entre `find()` y `find().pretty()`?

     - find(): devuelve un cursor con los documentos en formato JSON compacto (sin formato legible).
     - find().pretty(): aplica un formateo visual (sangrías, saltos de línea) para que sea más legible.

     No cambia el resultado, solo la presentación en consola.

   - ¿Qué significa la proyección en un `find()`? Ejemplo: `{nombre: 1, edad: 0}`
     La proyección define qué campos incluir o excluir en los documentos devueltos.
     db.coleccion.find(filtro, proyeccion)

     db.usuarios.find({}, { nombre: 1, edad: 0 })

     { nombre: 1 } incluye el campo nombre.
     { edad: 0 } excluye el campo edad.

   - Explica la diferencia entre `limit()` `sort()` en una consulta.
     db.usuarios.find().sort({ edad: -1 }).limit(3)

3. **Operadores de consulta**

   - ¿Qué diferencias hay entre `$eq`, `$ne`, `$gt`, `$lt`, `$in` y `$nin`?
     $eq: igual a
     { edad: { $eq: 25 } } (
     $ne: distinto de
     { edad: { $ne: 25 } }.

$gt: mayor que
{ edad: { $gt: 25 } }.

$lt: menor que
{ edad: { $lt: 25 } }.

$in: dentro de un conjunto
{ ciudad: { $in: ["Lima", "Bogotá"] } }.

$nin: no está en el conjunto
{ ciudad: { $nin: ["Lima", "Bogotá"] } }.

- Explica para qué sirven los operadores lógicos `$and`, `$or`, `$not`, `$nor`.
$and: todos deben cumplirse.
{ $and: [{ edad: { $gt: 18 } }, { ciudad: "Lima" }] }

$or: al menos uno debe cumplirse.
{ $or: [{ edad: { $lt: 18 } }, { ciudad: "Quito" }] }

$not: niega la condición.
{ edad: { $not: { $gt: 30 } } } // edad <= 30

$nor: ninguno debe cumplirse.
{ $nor: [{ ciudad: "Lima" }, { edad: { $lt: 18 } }] }

- ¿Qué diferencia hay entre `$exists` y `$type`?

$exists: verifica si un campo está presente o no, sin importar su valor.
{ telefono: { $exists: true } }
$type: filtra según el tipo de dato de un campo.
{ edad: { $type: "int" } }

4. **Agregaciones**

    - ¿Qué es el **pipeline de agregación** y cómo se diferencia de `find()`?

El pipeline de agregación (aggregate()) es un flujo de etapas (stages) que procesan documentos de manera secuencial, parecido a una "línea de producción":

- Cada etapa toma los documentos de la anterior.
- Puedes filtrar, transformar, agrupar, ordenar, calcular campos, unir colecciones, etc.
- Se usa mucho para reportes, estadísticas y transformaciones complejas.

find():

- Solo recupera documentos de una colección.
- Permite filtros, proyección, orden y límite, pero no operaciones complejas como agrupaciones o transformaciones avanzadas.

    - ¿Cuál es la función de `$match`, `$project`, `$group`, `$sort`, `$limit` y `$unwind`?

$match: Filtra documentos (igual que un find() pero dentro del pipeline).
{ $match: { edad: { $gt: 25 } } }

$project: Selecciona campos o crea nuevos campos.
{ $project: { nombre: 1, edad: 1, añoNacimiento: { $subtract: [2025, "$edad"] } } }

$group: Agrupa documentos por una clave y aplica funciones de agregación ($sum, $avg, $max, $min, $push, etc.).
{ $group: { _id: "$ciudad", totalUsuarios: { $sum: 1 }, edadProm: { $avg: "$edad" } } }

$sort: Ordena documentos
{ $sort: { edad: -1 } }

$limit: Limita el número de documentos que pasan al siguiente stage.
{ $limit: 5 }

$unwind: Descompone un array en múltiples documentos (uno por elemento).
{ $unwind: "$hobbies" }

- ¿Cuál es la diferencia entre `$lookup` y `$unwind`?

  $lookup: Sirve para hacer un join entre colecciones (similar a SQL).
  ```
  {
    $lookup: {
      from: "pedidos",
      localField: "_id",
      foreignField: "clienteId",
      as: "pedidosCliente"
    }
  }
  ```
  cada cliente tendrá un array pedidosCliente con sus pedidos.

  $unwind: Explota ese array en documentos individuales.
  Ejemplo: después del $lookup, si un cliente tiene 3 pedidos en el array, $unwind genera 3 documentos, cada uno con un pedido.

  $lookup une colecciones y produce arrays.
  $unwind rompe esos arrays en documentos independientes.

5. **Índices y performance**

   - ¿Qué es un índice en MongoDB y para qué sirve?

     Un índice es una estructura de datos (tipo árbol B-Tree) que guarda referencias ordenadas de los valores de un campo.

     Sirve para que MongoDB encuentre documentos más rápido sin recorrer toda la colección (lo que sería un collection scan).

     Beneficios:

     - Mejora el rendimiento de las consultas (find, sort, aggregate).
     - Optimiza búsquedas en campos con filtros frecuentes.
     - Se usan también en restricciones de unicidad (unique: true)

   - Diferencia entre índices simples y compuestos.

     Índice simple: se crea sobre un solo campo.
     db.usuarios.createIndex({ nombre: 1 }) // ascendente
     → acelera consultas filtrando u ordenando por nombre.

Índice compuesto: se crea sobre dos o más campos en orden específico.
db.usuarios.createIndex({ ciudad: 1, edad: -1 })
→ acelera consultas que usen:
ciudad solo,
ciudad + edad juntas,
pero no si solo usas edad sin ciudad (regla del prefijo).

- ¿Qué efecto tiene crear un índice sobre un campo `string` muy repetido (ej: ciudad)?

Si el campo tiene poca cardinalidad (pocos valores repetidos, como "Buenos Aires", "Córdoba"), el índice no ayuda mucho porque:

- MongoDB igualmente encuentra muchísimos documentos para cada valor.
- El overhead de mantener el índice puede ser mayor que el beneficio.

Puede incluso afectar negativamente la performance en inserciones/updates (porque debe actualizar el índice).

Los índices son más útiles en campos con alta selectividad (muchos valores distintos, como email o dni).

- ¿Cómo se usa `explain("executionStats")` para analizar el rendimiento de una query?

Devuelve:

- winningPlan → describe la estrategia usada:
  - COLLSCAN → recorrió toda la colección (lento).
  - IXSCAN → usó un índice (rápido).
- nReturned → cantidad de documentos devueltos.
- executionTimeMillis → tiempo total de ejecución.
- totalDocsExamined → cuántos documentos reales se leyeron.
- totalKeysExamined → cuántas entradas de índice se revisaron.

Claves para detectar problemas:

- COLLSCAN la consulta no usa índice → posible optimización.
- Si totalDocsExamined ≫ nReturned, significa que trae muchos documentos irrelevantes → índice mal diseñado.

6. **Modelado de documentos**

   - ¿Qué diferencia hay entre **documentos embebidos** y **referencias**?

     Documentos embebidos: Se guardan datos relacionados dentro del mismo documento.

     ```
     {
      "_id": 1,
      "nombre": "Juan",
      "direccion": {
       "calle": "Av. Siempre Viva",
       "ciudad": "Springfield"
      }
     }
     ```

     Ventajas:
     - Lectura rápida (todo está en un solo documento).
     - Ideal para datos que se consultan siempre juntos.

     Desventajas:
     - Puede crecer mucho el documento si hay listas grandes.
     - Límite de 16 MB por documento.

     Referencias: Se relacionan documentos usando un campo con el id de otro documento.

     coleccion usuario

     ```
     {
      "_id": 1,
      "nombre": "Juan",
      "direccionId": ObjectId("64f...")
     }
     ```

     Colección direccion

     ```
     {
      "_id": ObjectId("64f..."),
      "calle": "Av. Siempre Viva",
      "ciudad": "Springfield"
     }
     ```

     Ventajas:
     - Evita duplicación de datos.
     - Maneja mejor listas muy grandes (colecciones separadas).

     Desventajas:
     - Hace falta un lookup (join) o varias consultas → más lento que embebido.

- ¿Cuándo conviene usar cada enfoque?

Embebidos:

- Relaciones uno a pocos.
- Datos que siempre se consultan juntos (ej: dirección dentro de usuario).
- Cuando la cantidad de subdocumentos no va a crecer demasiado.

Referencias:

- Relaciones muchos a muchos o uno a muchísimos (ej: un usuario con miles de facturas).
- Cuando necesitas reutilizar los datos en diferentes documentos.
- Cuando los subdocumentos cambian a ritmos distintos y conviene tenerlos separados.

Regla práctica:

Si accedés a los datos casi siempre juntos → embebidos.
Si accedés a los datos por separado o son enormes → referencia

- ¿Qué significa diseñar basado en **query patterns**?

Significa modelar la base de datos en función de cómo la vas a consultar, no solo en la lógica de normalización como en SQL.

Parte II – Práctica con Datos

Imagina que tienes una base de datos llamada `tiendaOnline` con las siguientes colecciones:

Colección `clientes`

```
{
  "_id": ObjectId(),
  "nombre": "Ana Pérez",
  "edad": 29,
  "email": "ana@example.com",
  "premium": true,
  "direccion": {
    "ciudad": "Buenos Aires",
    "codigoPostal": "1000"
  },
  "suscripciones": ["newsletter", "ofertas"]
}
```

Colección `productos`

```
{
  "_id": ObjectId(),
  "nombre": "Zapatillas Running",
  "categoria": "calzado",
  "precio": 45000,
  "stock": 12
}
```

Colección `ordenes`

```
{
  "_id": ObjectId(),
  "clienteId": ObjectId("..."),
  "fecha": ISODate("2025-09-10"),
  "items": [
    {"productoId": ObjectId("..."), "cantidad": 2, "precioUnitario": 45000}
  ],
  "total": 90000
}
```

◇ Ejercicios de CRUD y Find

1. Insertar un cliente nuevo con nombre "Carlos Gómez", edad 35, ciudad "Córdoba", sin email y con suscripción a "newsletter".

```
db.clientes.insertOne({
  nombre: "Carlos Gómez",
  edad: 35,
  premium: false,                // opcional; si no lo querés, omitilo
  direccion: { ciudad: "Córdoba" },
  suscripciones: ["newsletter"]
})
```

2. Insertar 3 productos nuevos:
    - Camiseta Deportiva ($15000, stock 20)
    - Mochila ($30000, stock 5)
    - Botella de Agua ($8000, stock 50)

```
db.productos.insertMany([
  { nombre: "Camiseta Deportiva", precio: 15000, stock: 20 },
  { nombre: "Mochila", precio: 30000, stock: 5  },
  { nombre: "Botella de Agua", precio: 8000,  stock: 50 }
])
```

3. Consultar todos los clientes mayores de 30 años que no tengan email.

```
db.clientes.find(
  {
    edad: { $gt: 30 },
    $or: [{ email: { $exists: false } }, { email: null }]
  }
)
```

4. Buscar todos los productos que estén entre $10.000 y $40.000.

```
db.productos.find({ precio: { $gte: 10000, $lte: 40000 } })
```

5. Mostrar solo el nombre y la categoría de los productos, ocultando el `_id`.

```
db.productos.find(
  {},
  { _id: 0, nombre: 1, categoria: 1 }
)
```

6. Listar los clientes que tengan "premium" en `true` o que vivan en "Buenos Aires".

```
db.clientes.find({
  $or: [
    { premium: true },
    { "direccion.ciudad": "Buenos Aires" }
  ]
})
```

◇ Ejercicios de Operadores

7. Encontrar todos los clientes que tengan más de 25 años **y** estén suscritos a "ofertas".

```
db.clientes.find({
  edad: { $gt: 25 },
  suscripciones: "ofertas"
})
```

```
db.clientes.find({
edad: { $gt: 25 },
suscripciones: { $in: ["ofertas"] }
})
```

8. Buscar productos que **no** estén en la categoría "calzado".

```
db.productos.find({
  categoria: { $ne: "calzado" }
})
```

9. Listar clientes donde el campo `email` **no exista**.

```
db.clientes.find({
  email: { $exists: false }
})
```

10. Buscar productos cuyo stock sea de tipo `number` (usando `$type`).

```
db.productos.find({
stock: { $type: "number" }
})
```

◇ Ejercicios de Agregaciones

11. Obtener la cantidad de clientes por ciudad (`$group`).

```
db.clientes.aggregate([
  { $group: { _id: "$direccion.ciudad", cantidadClientes: { $sum: 1 } } },
  { $sort: { cantidadClientes: -1 } }
])
```

12. Calcular el gasto total de cada cliente sumando los `total` de sus órdenes.

```
db.ordenes.aggregate([
  { $group: { _id: "$clienteId", gastoTotal: { $sum: "$total" } } },
  { $sort: { gastoTotal: -1 } }
])
```

13. Listar los productos más vendidos (sumando las cantidades en `items`).

```
db.ordenes.aggregate([
  { $unwind: "$items" },
  { $group: {
      _id: "$items.productoId",
      cantidadVendida: { $sum: "$items.cantidad" }
  }},
  { $lookup: {
      from: "productos",
      localField: "_id",
      foreignField: "_id",
      as: "producto"
  }},
  { $unwind: "$producto" },
  { $project: { _id: 0, productoId: "$_id", nombre: "$producto.nombre", cantidadVendida: 1
} },
  { $sort: { cantidadVendida: -1 } }
])
```

14. Hacer un `$lookup` para que cada orden muestre los datos del cliente.

```
db.ordenes.aggregate([
  { $lookup: {
      from: "clientes",
      localField: "clienteId",
      foreignField: "_id",
      as: "cliente"
  }},
  { $unwind: "$cliente" },
  { $project: {
      _id: 1,
      fecha: 1,
      total: 1,
      "cliente._id": 1,
      "cliente.nombre": 1,
      "cliente.email": 1,
      "cliente.premium": 1,
      "cliente.direccion": 1
  }}
])
```

15. Hacer un `$lookup` para que cada orden muestre los nombres de los productos en `items`.

```
db.ordenes.aggregate([
  { $unwind: "$items" },
  { $lookup: {
      from: "productos",
      localField: "items.productoId",
      foreignField: "_id",
      as: "prod"
  }},
  { $unwind: "$prod" },
  { $group: {
      _id: "$_id",
      clienteId: { $first: "$clienteId" },
      fecha: { $first: "$fecha" },
      total: { $first: "$total" },
      items: { $push: {
        productoId: "$items.productoId",
        nombre: "$prod.nombre",
        cantidad: "$items.cantidad",
        precioUnitario: "$items.precioUnitario"
      }}
  }},
```

```
  { $sort: { _id: 1 } }
])
```

16. Obtener el ticket promedio de compra (`$group` + `$avg`).

```
db.ordenes.aggregate([
  { $group: { _id: null, ticketPromedio: { $avg: "$total" } } },
  { $project: { _id: 0, ticketPromedio: 1 } }
])
```

◇ Ejercicios de Índices y Performance

17. Crear un índice en `clientes.email` y probar una búsqueda por email.

```
db.clientes.createIndex({ email: 1 }, { unique: true, sparse: true, name: "idx_email_1" })

db.clientes.find({ email: "ana@example.com" })
```

18. Crear un índice compuesto en `productos (categoria, precio)` y probar una consulta filtrando por categoría y ordenando por precio.

```
db.productos.createIndex({ categoria: 1, precio: 1 }, { name: "idx_categoria_1_precio_1" })

db.productos.find({ categoria: "calzado" }).sort({ precio: 1 })
```

19. Explicar con `explain("executionStats")` la diferencia entre buscar un producto con y sin índice.

Sin índice:

- winningPlan con COLLSCAN (recorre toda la colección).
- totalDocsExamined ≈ cantidad total de docs en productos.
- Puede aparecer un SORT en memoria (stage: SORT) si no hay índice que soporte el orden.
- executionTimeMillis más alto (depende del tamaño)

Con índice:

- winningPlan con IXSCAN sobre "idx_categoria_1_precio_1".
- totalKeysExamined y totalDocsExamined cercanos a nReturned (muy bajos).
- No aparece un SORT separado (el índice ya entrega en orden).
- executionTimeMillis sustancialmente menor.

◇ Ejercicios de Modelado

20. Diseñar un modelo de documento para almacenar reseñas de productos con: cliente, puntaje (1-5), comentario y fecha. ¿Conviene embeber las reseñas dentro de `productos` o tener una colección aparte? Justificar.

    Tener una colección aparte y en el producto tener un resumen de esas reseñas, ej promedio, cantidad, etc. Tener una colección dedicada simplifica para buscar todas las reseñas de un cliente o de varios productos (patrones de consulta)

21. Replantear el modelo de `ordenes` para incluir más datos del cliente en el momento de la compra (por ejemplo, dirección). ¿Conviene duplicar datos o mantener solo la referencia?

Duplicar. En el modelo de ordenes se guardaría clienteId como referencia al cliente vivo y una copia inmutable (snapshot) de los datos re levantes al momento de la compra (nombre, emails, direcciones, CUIT, etc.)

Si el cliente cambia su dirección o el producto cambia de nombre/precio, la orden debe seguir mostrando lo que era al comprar.

Tambien sirve ver una orden completa sin hacer múltiples lookup.