Trabajo Práctico: Modelado de Documentos e Índices en MongoDB

Parte 1 – Modelado de Documentos

1. Teoría (preguntas)

a) Explica con tus palabras las diferencias entre modelo embebido y modelo con referencias en MongoDB.

- **Modelo embebido (embedded documents):**
  - La información relacionada se guarda dentro del mismo documento.
  - Todo está en una sola colección.
  - Ventaja: acceso rápido porque no hay que hacer joins ni múltiples consultas.
  - Desventaja: si los datos embebidos crecen demasiado, el documento puede volverse muy grande (MongoDB tiene un límite de 16 MB por documento).

- **Modelo con referencias (referenced documents):**
  - La información relacionada se guarda en colecciones separadas.
  - Los documentos se conectan mediante un identificador (_id u otro campo).
  - Ventaja: mayor flexibilidad y normalización, evita duplicación excesiva de datos.
  - Desventaja: necesitas múltiples consultas o un lookup para juntar la información.

b) ¿En qué casos conviene usar documentos embebidos y en cuáles referencias? Da al menos un ejemplo de cada caso.

Conviene usar embebidos cuando: La relación es uno a pocos y los datos relacionados casi siempre se consultan juntos.

```
{
  "_id": 1,
  "nombre": "Juan Pérez",
  "email": "juan@mail.com",
  "direcciones": [
    { "calle": "San Martín 123", "ciudad": "Buenos Aires" },
```

```
    { "calle": "Belgrano 456", "ciudad": "Mendoza" }
  ]
}
```

Conviene embebido, porque las direcciones pertenecen siempre al mismo usuario.

Conviene usar referencias cuando: La relación es muchos a muchos o los datos relacionados se comparten entre documentos.

```
// Colección productos
{
  "_id": 101,
  "nombre": "Guitarra Fender",
  "precio": 1200
}
// Colección pedidos
{
  "_id": 5001,
  "cliente": "Juan Pérez",
  "productos": [101, 102, 103]  // referencias a productos
}
```

Conviene referencias, porque los mismos productos pueden aparecer en distintos pedidos sin duplicar información.

c) ¿Qué son los query patterns y por qué son importantes al diseñar un modelo de datos?

Los query patterns son los patrones de consulta más frecuentes que va a ejecutar tu aplicación sobre la base de datos. Son consultas frecuentes.

Son importantes porque:

- Guían cómo modelar los documentos: embebidos vs referencias.
- Permiten optimizar el rendimiento, evitando joins innecesarios.
- Ayudan a decidir qué campos indexar para acelerar las consultas.

Ejemplo:

Si en una app de pedidos la mayoría de las consultas son “traer un pedido con todos sus productos y datos del cliente”, conviene embebido.

Pero si las consultas más frecuentes son “traer todos los pedidos de un producto específico”, conviene referencias + un índice en productos.

2. Práctica (ejercicio de diseño)

Ejercicio 1 – Red Social

Todos estos ejercicios se pueden entregar en formato:

- **Esquema JSON** (cómo sería el documento en MongoDB).

- Justificación escrita: ¿por qué embebido o referencias según los query patterns?

Queremos modelar una aplicación tipo **Twitter**:

- Usuarios tienen: nombre, email, fecha de registro.

- Cada publicación (tweet) tiene: texto, fecha, autor, likes, comentarios.

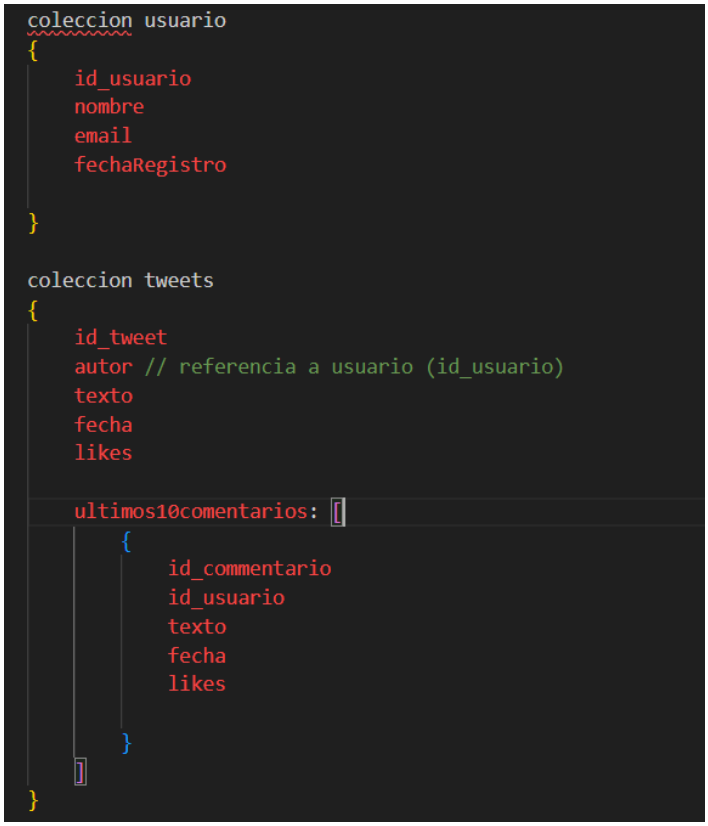
- Cada comentario tiene: autor, fecha, texto.
- Justificar:

    - ¿Qué diseño conviene si la consulta más frecuente es: *"Ver los últimos 10 comentarios de un tweet"*?

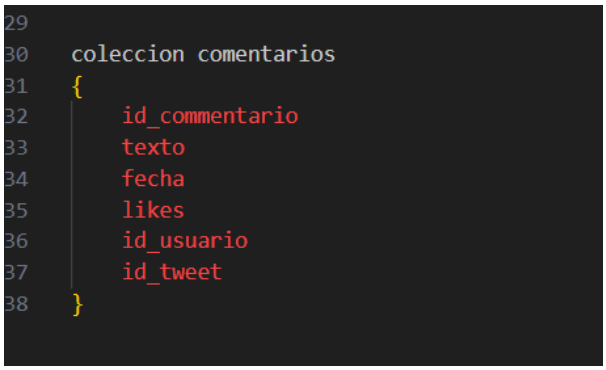
      Embebido: Guardar los comentarios completos en colección aparte.

- ¿Y si la consulta más frecuente es: *"Ver todos los comentarios de un usuario"*?

Referencia: Todos los comentarios en colección comments indexados por autorId.



```
coleccion usuario
{
    id_usuario
    nombre
    email
    fechaRegistro

}

coleccion tweets
{
    id_tweet
    autor // referencia a usuario (id_usuario)
    texto
    fecha
    likes

    ultimos10comentarios: [
        {
            id_commentario
            id_usuario
            texto
            fecha
            likes

        }
    ]
}
```



```
29
30  coleccion comentarios
31  {
32      id_commentario
33      texto
34      fecha
35      likes
36      id_usuario
37      id_tweet
38  }
```

Ejercicio 2 – Plataforma de Cursos Online

Cada curso tiene: título, descripción, profesor, lista de lecciones.
Cada lección tiene: título, duración, materiales.
Cada alumno puede inscribirse a varios cursos.

Lopez Celina COM3

Tareas:

1. Modelar la relación **curso – lecciones** (embebido o referencias).

2. Modelar la relación **alumno – cursos** (embebido o referencias).

3. Pensar consultas frecuentes:

    - Obtener todas las lecciones de un curso: embebido
    - Ver todos los cursos de un alumno: referencia por tabla intermedia

    - Listar los alumnos de un curso.
      ¿Cuál de los modelos se adapta mejor?: referencia por tabla intermedia

```
coleccion profesores
{
    id_profesor
    nombre
    email
    fechaRegistro
}

coleccion alumnos
{
    id_alumno
    nombre
    email
    fechaRegistro
}
```

```
coleccion cursos
{
    _id_curso
    titulo
    descripcion
    id_profesor
    lecciones: [       // EMBEBIDO: 1 curso a muchas lecciones
        {
            id_leccion
            titulo
            duracionMin
            materiales: [   // EMBEBIDO: 1 leccion a muchos materiales
                {
                    id_material
                    tipo
                    url
                }
            ]
        }
    ],
}
```

```
// REFERENCIA: Tabla puente para la relacion alumno - curso (muchos a muchos)
coleccion inscripciones
{
    id_inscripcion
    alumnoId    // ref a id_alumno
    cursoId     // ref a id_curso
    fechaInscripcion
}
```

Ejercicio 3 – Blog de Noticias

Cada artículo tiene: título, cuerpo, autor, fecha, etiquetas.
Los usuarios pueden guardar artículos como “favoritos”.

Tareas:

1. Modelar la relación **artículo – etiquetas** (embebido o referencias).
2. Modelar la relación **usuario – favoritos**. (embebido o referencias).
3. Consulta común: *"Buscar todos los artículos de una etiqueta"*.
    - ¿Qué diseño es más eficiente? Embebido.
4. Consulta común: *"Obtener todos los favoritos de un usuario"*.
    - ¿Qué diseño elegís? embebido

{} bb.json

```
1
2   coleccion articulo
3   {
4       id_articulo
5       titulo
6       cuerpo
7       autor
8       fecha
9       etiquetas { id_etiqueta}, label_etiqueta //EMBEBIDO
10  }
11
12  coleccion usuario
13  {
14      id_usuario
15      articulos_favoritos { id_articulo, titulo, cuerpo, autor, fecha, etiquetas[]} //EMBEBIDO
16  }
17
18
```

Ejercicio 4 – Sistema de Gestión de Hospital

- Pacientes: nombre, DNI, fecha de nacimiento.
- Médicos: nombre, especialidad.
- Turnos: fecha, paciente, médico.
- Historias clínicas: diagnósticos, tratamientos, estudios.

Tareas:

1. Modelar la relación **paciente – historia clínica**.
2. Modelar la relación **paciente – turnos**.
3. Pensar consultas:
    - Buscar todos los turnos de un médico en un rango de fechas.
      ```
      db.turnos.find({
        "medico.id_medico": 123,
        fecha: { $gte: ISODate("2025-09-01"), $lte: ISODate("2025-09-10") }
      })
      .sort({ fecha: 1 })
      ```
    - Obtener toda la historia clínica de un paciente.
      ```
      db.pacientes.findOne(
        { _id: 123},
        { nombre: 1, dni: 1, diagnosticos: 1, tratamientos: 1, estudios: 1 }
      )
      ```
    - Listar todos los pacientes que tienen un diagnóstico específico.
      Justificar qué modelo es mejor para cada caso.

      ```
      db.pacientes.find(

        { "diagnosticos.nombre": "Hipertensión" },

        { nombre: 1, dni: 1 }

      )
      ```

Lopez Celina COM3

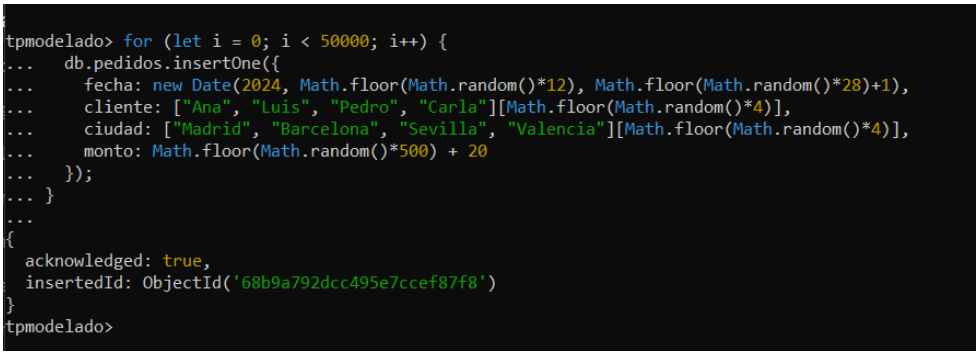
```
coleccion pacientes
{
    id_paciente
    nobmre
    dni
    fecha_nacimiento
    historia_clinica:[ // EMBEBIDA
        diagnosticos [{nombre1, fecha1, nota1}, {nombre2, fecha2, nota2}]
        tratamientos[{nombre1, desde1, hasat1}, {nombre2, desde2, hasta2}]
        estudios [{tipo1, fecha1, resultado1}, {tipo1, fecha2, resultado2}]
    ]

}

coleccion medicos
{
    id_medico
    nombre
    especialidad
}

coleccion turnos
{
    id_turno
    fecha
    paciente {id_paciente, nombre, dni, fecha_nacimiento}
    medico {id_medico, nombre, especialidad}
}

coleccion historias clinicas
{
    id_historiaclinica
    diagnosticos [{nombre1, fecha1, nota1}, {nombre2, fecha2, nota2}]
    tratamientos[{nombre1, desde1, hasat1}, {nombre2, desde2, hasta2}]
    estudios [{tipo1, fecha1, resultado1}, {tipo1, fecha2, resultado2}]

}
```

Parte 2 – Índices y Performance

1. Preparación

Crear una colección `pedidos` con datos de prueba:

```
for (let i = 0; i < 50000; i++) {
  db.pedidos.insertOne({
    fecha: new Date(2024, Math.floor(Math.random()*12),
Math.floor(Math.random()*28)+1),
    cliente: ["Ana", "Luis", "Pedro",
"Carla"][Math.floor(Math.random()*4)],
    ciudad: ["Madrid", "Barcelona", "Sevilla",
"Valencia"][Math.floor(Math.random()*4)],
    monto: Math.floor(Math.random()*500) + 20
  });
}
```



---

2. Consultas sin índice

Ejecutar las siguientes queries y medir con `explain("executionStats")`:

```
db.pedidos.find({ ciudad: "Madrid" }).explain("executionStats")
db.pedidos.find({ monto: { $gt: 300 } }).explain("executionStats")
db.pedidos.find({ ciudad: "Madrid", monto: { $gt: 300 }
}).explain("executionStats")
```

Preguntas:

- ¿Qué stage aparece (`COLLSCAN` o `IXSCAN`)?

COLLSCAN

- ¿Cuántos documentos se examinan?
  totalDocsExamined: 50000
- ¿Cuál fue el tiempo de ejecución?

  executionTimeMillis: 24

---

3. Crear índices

a) Crear un **índice simple** en `ciudad` y repetir la primera query.

Stage: IXSCAN

totalDocsExamined: 12337

executionTimeMillis: 41

b) Crear un **índice simple** en `monto` y repetir la segunda query.

Stage: IXSCAN

totalDocsExamined: 21883

executionTimeMillis: 45

c) Crear un **índice compuesto** en `{ ciudad: 1, monto: 1 }` y repetir la tercera query.

Stage: IXSCAN

totalDocsExamined: 5410

executionTimeMillis: 11

Preguntas:

- ¿Cómo cambiaron los resultados de `executionStats`?

  El stage cambio de COLLSCAN a IXSCAN -> mas rápido

- ¿Qué diferencia hay en `keysExamined` y `docsExamined` antes y después?

Antes: keysExamined siempre 0 porque no hay índice
Despues: keysExamined igual a docsExamined

- ¿El orden de los campos en el índice afecta las consultas? Probar invirtiendo el orden.

  Invierto el índice: db.pedidos.createIndex({ monto: 1, ciudad: 1 })

  db.pedidos.find({monto: { $gt: 300 }, ciudad: "Madrid", }) .explain("executionStats")

  totalKeysExamined: 5848
  totalDocsExamined: 5410
  executionTimeMillis: 28

  Si, afecta. Es mucho mas eficiente con el índice { ciudad, monto } porque primero acota por ciudad.

Parte 3 – Reflexión final

1. ¿Cómo influye el **diseño del modelo de documentos** en la necesidad de índices?

   Consultas frecuentes dictan índices: si el modelo te obliga a consultar por ciudad o por monto+ciudad, necesitás índices que lo soporten ({ ciudad:1 }, { ciudad:1, monto:1 }).

   Patrón igualdad + rango: el orden del índice debe seguir el patrón de la consulta → igualdades primero, rangos después.

   Denormalización vs joins: en MongoDB, al evitar joins, todo se resuelve con filtros en una colección: más presión sobre índices correctos.

   Cardinalidad/selectividad: campos con alta selectividad (muchos valores únicos o distribución sesgada) se benefician más del índice; el diseño debería favorecer filtros selectivos.

2. ¿Por qué no es recomendable crear índices para todos los campos?
   Costo de escritura: cada insert/update actualiza todos los índices → más latencia y CPU.

   Espacio y RAM: índices ocupan disco y memoria; menos working set efectivo en caché.

Planificador y fragmentación: demasiados índices confunden al planner y pueden degradar planes; además, más mantenimiento (rebuild/compact).

Límites prácticos: hay límites por colección y por documento; además, el overhead operativo crece.
Conclusión: indexa solo lo que responda a consultas reales y críticas (por volumen o latencia).

3. ¿Qué diferencias notaste al indexar un campo string (`ciudad`) frente a un campo numérico (`monto`)?

   Mas eficiencia con el filtro primero por ciudad:
   ciudad (4 valores) → baja selectividad; el índice filtra menos (ej. ~25% del total).
   monto (muchos valores) → más selectivo en igualdad; en rangos depende del corte (ej. >300 ~40%).

Tamaño del índice: strings pueden ser más grandes (claves más largas): índice más pesado; números son compactos.

4. Imagina que tu aplicación crece y ahora se hacen muchas consultas por **rango de fechas**.

   - ¿Qué índice agregarías?
     db.pedidos.createIndex({ fecha: 1 })

     o combinado por ej
     db.pedidos.createIndex({ ciudad: 1, fecha: 1 })

   - ¿Cómo justificarías esa decisión?

     Patrón de acceso: muchas queries del tipo “entre fechas” necesitan que el índice limite por rango sin COLLSCAN.

     Orden del índice: pongo primero los predicados de igualdad (si existen, ej. ciudad) y luego fecha (rango/orden).

     Métricas: se espera bajar keysExamined/docsExamined y executionTimeMillis de forma similar al ejercicio anterior.