Modelado de datos tickets

Candidato: Nicolás Díaz

Fecha: 25 de Octubre 2025

Version: 1.0

1. Entendimiento del dominio.

1.1. Descripción general del Sistema.

El sistema gestiona tickets de soporte que evolucionan a lo largo de su ciclo de vida. Cada ticket puede cambiar de estado, ser clasificado jerárquicamente, recibir comentarios y ser asignado a diferentes usuarios. Todas estas modificaciones quedan registradas en un histórico de acciones para trazabilidad y auditoría.

1.2. Conceptos clave:

1.2.1. Ticket.

Representa un caso de soporte con:

- Identificador único.
- Estado actual.
- Clasificación jerárquica.
- Fechas de creación, actualización y cierre.
- Histórico completo de acciones.

1.2.2. Estados del ticket.

Los tickets transitan por los siguientes estados:

- **open:** Ticket recién creado o reabierto.
- **in_progress:** Ticket en proceso de atención.
- **closed:** Ticket resuelto y cerrado.

1.2.3. Clasificadores jerárquicos.

Estructura de árbol que organiza los tickets por categorías. Características:

- Estructura de árbol con raíz y nodos.
- Un ticket puede estar clasificado en **un único nodo** por cada raíz jerárquica.
- Los nodos pueden tener múltiples niveles de profundidad.
- Al filtrar por un nodo padre, se incluyen todos sus nodos hijos.

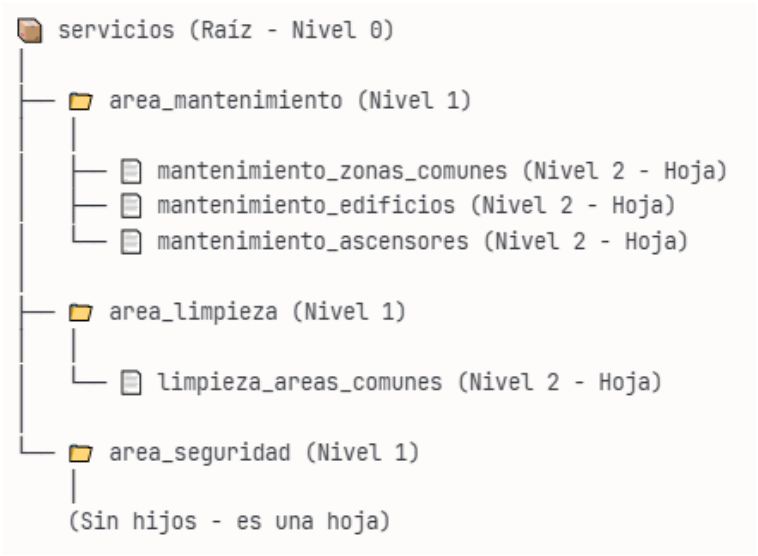
1.2.4. Histórico de acciones:

Registro inmutable de todas las modificaciones, incluyendo:

- **state_change:** Cambio de estado.
- **classification_change:** Cambio de clasificación.
- **comment:** Comentario agregado.

- **assignment:** Asignación a usuario.

1.3. Estructura jerárquica de clasificadores.



1.3.1. Características:

- Profundidad variable: Sin límite teórico de niveles (aunque generalmente 3-4 niveles máximo).
- Cada nodo hijo tiene un único nodo padre.
- Los nodos hoja (sin hijos) representan las categorías más específicas.
- Filtrar por un nodo padre incluye automáticamente todos sus nodos hijos y descendientes.

1.4. Supuestos de negocio.

Para completar la solución ante información no especificada, se asumen las siguientes reglas:

1.4.1. Estados:

- Solo existen tres estados: **open**, **in_progress**, **closed**.
- Un ticket siempre inicia en estado **open**.
- El modelo es flexible para agregar más estados en el futuro.

1.4.2. Reaperturas:

- Una reapertura ocurre cuando un ticket pasa de estado **closed** a **open.**
- Se cuenta cada vez que esto sucede (un ticket puede reabrirse múltiples veces).

1.4.3. Estado al final del periodo.

- Se debe determinar qué estado tenía el ticket exactamente al momento endDate.
- Esto requiere consultar el histórico para "viajar en el tiempo".
- Si el ticket no existía al final del período, no se incluye.

1.4.4. Sobre métricas:

- **Ingreso:** Cuando se crea el ticket (**created_at** dentro del rango).
- **Cierre:** Cuando un ticket cambia a estado **closed** (acción **state_change** → **closed**).
- **Reapertura:** Cuando un ticket **closed** cambia a open nuevamente.

1.4.5. Clasificadores:

- El sistema soporta clasificadores jerárquicos genéricos (no limitado a una estructura específica).
- Por defecto, se asume una única raíz jerárquica, pero el modelo es extensible a múltiples raíces.
- Cada raíz puede tener su propia estructura y profundidad.
- Un ticket debe estar clasificado en exactamente un nodo por cada raíz jerárquica existente.
- Los nombres de nodos son flexibles y configurables según las necesidades del negocio.

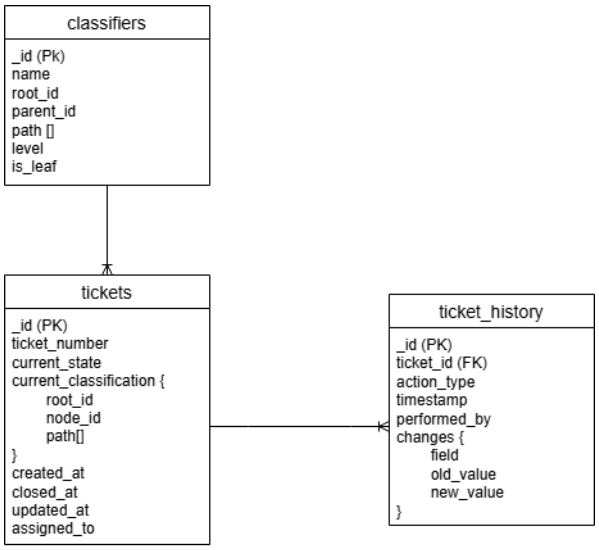
2. Modelo de datos.

2.1. Estrategia de diseño:

Para este sistema de tickets se optó por un diseño **híbrido** que combina:

- **Datos actuales normalizados** para consultas rápidas del estado presente
- **Datos históricos denormalizados** para auditoría y análisis temporal
- **Catálogo de clasificadores** separado para mantener la integridad referencial.

2.2. Modelo de entidad relación.



2.3. Colecciones de MongoDB.

Se propone crear tres colecciones. Ver implementación en: https://github.com/drawnick1214/DataModelingMongoDB/blob/master/mongodb/01-create-collections.js

2.3.1. classifiers:

Almacena la estructura jerárquica completa de clasificadores. Esta es una colección de **catálogo** se pretende que se modifique raramente es reutilizable y facilita validaciones de integridad referencial.

- **Estructura del documento:**

```
{
        "_id": "node_001",
        "name": "Nombre del Nodo",
        "root_id": "root_001",
        "parent_id": "parent_node_id",
        "path": ["root_001", "parent_node_id", "node_001"],
        "level": 2,
        "is_leaf": false,
        }
}
```

- **Diccionario de datos:**

Campo	Tipo	Descripción
_id	String	Identificador único del nodo (slug o código)
name	String	Nombre descriptivo mostrado al usuario
root_id	String	ID de la raíz jerárquica a la que pertenece
parent_id	String/Null	ID del nodo padre (null si es raíz)
path	Array[String]	Ruta completa desde la raíz hasta este nodo
level	Integer	Nivel de profundidad (0=raíz, 1=primer nivel, etc.)
is_leaf	Boolean	Indica si el nodo no tiene hijos
metadata	Object	Información adicional opcional

2.3.2. tickets:

Almacena el **estado actual** de cada ticket. Esta es la colección principal para consultas operacionales, es pequeña y rápida para consultas actuales.

- **Estructura del documento:**

```
{
        _id: ObjectId("..."),
        ticket_number: "TKT-2025-0001",
        current_state: "open",                  // open | in_progress | closed
        current_classification: {
                root_id: "servicios",
                node_id: "area_mantenimiento",
```

```
            path: ["servicios", "area_mantenimiento"] // DENORMALIZADO
        },
        created_at: ISODate("2025-01-15T10:30:00Z"),
        closed_at: null,
        updated_at: ISODate("2025-01-15T10:30:00Z"),
        assigned_to: "user_123"
}
```

- **Diccionario de datos:**

Campo	Tipo	Descripción
_id	ObjectId	Identificador único del ticket (generado por MongoDB)
ticket_number	String	Número legible para humanos (TKT-YYYY-NNNN)
current_state	String	Estado actual (open, in_progress, closed)
current_classification	Object	Clasificación actual con ruta completa denormalizada
created_at	ISODate	Fecha y hora de creación del ticket
closed_at	ISODate/Null	Fecha y hora del cierre (null si está abierto)
updated_at	ISODate	Fecha de última modificación
assigned_to	String/Null	Usuario asignado actualmente
metadata	Object	Campos adicionales según necesidades del negocio

2.3.3. ticket_history.

Almacena el **histórico inmutable** de todas las acciones sobre tickets. Esta colección crece indefinidamente. Se decide separar de tickets, y se pretende el uso de índices optimizados para cada tipo de consulta.

- **Estructura del documento:**

```
{
        _id: ObjectId("..."),
        ticket_id: ObjectId("..."),       // FK a tickets
        action_type: "state_change",
        timestamp: ISODate("2025-01-20T14:45:00Z"),
        performed_by: "user_789",
        changes: {
                field: "state",
                old_value: "open",
                new_value: "in_progress"
        }
}
```

- **Diccionario de acciones del atributo (action_type):**

Tipo	Descripción	Campos en changes
state_change	Cambio de estado	field: "state", old_value, new_value
classification_change	Cambio de clasificación	field: "classification", old_value, new_value
comment	Comentario agregado	content: "texto del comentario"

assignment	Asignación de usuario	field: "assigned_to", old_value, new_value
ticket_created	Creación del ticket	initial_state, initial_classification

2.4. Justificación de decisiones de diseño.

La relación entre tickets y **`ticket_history`** es 1:N, donde la referencia está en **`ticket_history.ticket_id`** que referencia **`ticket_id`** de la colección **`tickets`**. En este caso no se mantiene referencia inversa ya que el histórico puede crecer indefinidamente.

Entonces la forma de saber más detalles acerca de ticket es el campo **`timestamp`** en **`ticket_history`** que es la clave para consultar el histórico completo. Permitiendo reconstruir el estado del ticket en cualquier momento del tiempo. Con índice compuesto (**`ticket_id`**, **`timestamp`**) las consultas son muy eficientes. El campo **`updated_at`** en **`tickets`** sirve como referencia rápida de la última modificación.

3. Índices.

3.1. Estrategia de indexación.

Una estrategia de indexación bien diseñada es **crítica** para el performance del sistema, especialmente cuando el volumen de **`tickets`** crece a miles o millones de registros. Los principios aplicados para la indexación en este modelo de datos fueron:

- Indexar campos usados frecuentemente en filtros y ordenamiento.
- Crear índices compuestos para queries que combinan múltiples campos.
- Considerar el orden de campos en índices compuestos (selectividad).
- Balancear performance de lectura vs overhead de escritura.
- Evitar sobre-indexación (cada índice consume espacio y ralentiza inserts/updates).
- Todas las colecciones tienen un índice de _id.

3.2. Índices para la colección '**`classifiers`**'.

Como esta colección es pequeña y estática, como un catálogo, se necesitan índices mínimos. Ver implementación en: https://github.com/drawnick1214/DataModelingMongoDB/blob/master/mongodb/02-create-indexes.js

3.2.1. Índice de navegación jerárquica.

Índice para búsqueda por raíz y padre, cuyo propósito es listar todos los nodos de una raíz específica y encontrar los hijos directos de un nodo. El siguiente es la creación del índice:

```
db.classifiers.createIndex({ "root_id": 1, "parent_id": 1 })
```

Este es un ejemplo de los queries optimizados:

```
// Todos los nodos de una raíz
```

```
db.classifiers.find({ root_id: "servicios" })

// Hijos directos de un nodo
db.classifiers.find({ parent_id: "area_mantenimiento" })
```

3.2.2. *Índice de búsqueda de descendientes.*

Índice para búsquedas por elementos del path, que sirve para encontrar todos los descendientes de un nodo, útil si hay jerarquías profundas. El siguiente es la creación del índice:

```
db.classifiers.createIndex({ "path": 1 })
```

Este es un ejemplo de la query:

```
// Todos los descendientes de area_mantenimiento
db.classifiers.find({ path: "area_mantenimiento" })
// Retorna: area_mantenimiento, mantenimiento_zonas_comunes,
//          mantenimiento_edificios, mantenimiento_ascensores
```

3.3. índices para la colección '**tickets**'.

Esta es la **colección crítica** donde el diseño de índices impacta directamente el performance del sistema.

3.3.1. *Índice de número de Ticket:*

Esto garantiza la unicidad de números de ticket y garantiza la búsqueda rápida por número de ticket (común en interfaces de usuario). El siguiente es la creación del índice:

```
db.tickets.createIndex({ "ticket_number": 1 }, { unique: true })
```

Este es un ejemplo de la query:

```
db.tickets.findOne({ ticket_number: "TKT-2025-0001" }) // O(log n) - típicamente <
1ms
```

3.3.2. *Índice de estado actual del ticket.*

Índice para filtra el ticket por estado actual. El siguiente es la creación del índice:

```
db.tickets.createIndex({ "current_state": 1 })
```

Este es un ejemplo de la query:

```
// Dashboard: "Tickets abiertos"
db.tickets.find({ current_state: "open" })
```

```
// Métrica: Conteo por estado
db.tickets.countDocuments({ current_state: "closed" })
```

3.3.3. Rango de Fechas:

Índice para filtro por rango de fechas, esto filtra tickets que estuvieron abiertos durante un rango de fechas específico, y también hace el cálculo de métricas en periodos específicos. El siguiente es la creación del índice:

```
db.tickets.createIndex({ "created_at": 1, "closed_at": 1 })
```

Este es un ejemplo de la query:

```
// Tickets abiertos durante enero 2025
db.tickets.find({
  $or: [
    { created_at: { $lt: endDate }, closed_at: { $gte: startDate } },
    { created_at: { $lt: endDate }, closed_at: null }
  ]
})

// Tickets creados en un período (ingresos)
db.tickets.find({
  created_at: { $gte: startDate, $lt: endDate }
})
```

Orden de campos justificado:

- **created_at primero:**
    - Siempre tiene valor (NOT NULL)
    - Mayor selectividad
    - Usado en más queries
- **closed_at segundo:**
    - Puede ser NULL (~20% de tickets)
    - Menor selectividad
    - Usado en combinación con created_at

3.3.4. Índice de clasificación jerárquica.

índice para filtro por clasificación jerárquica. Filtrar tickets para nodo clasificador y todos sus descendientes. Soporta el filtro jerárquico más importante del sistema. El siguiente es la creación del índice:

```
db.tickets.createIndex({ "current_classification.path": 1 })
```

Este es un ejemplo de la query:

```
// Tickets de "area_mantenimiento" y todos sus hijos
db.tickets.find({
```

```
  "current_classification.path": "area_mantenimiento"
})
// Encuentra tickets en: area_mantenimiento, mantenimiento_zonas_comunes,
//                       mantenimiento_edificios, mantenimiento_ascensores
```

3.3.5. *Índice nodo específico:*

índice clasificación por nodo específico. Su propósito es filtrar tickets de un nodo específico, además de validar y reportar detalladamente. El siguiente es la creación del índice:

```
db.tickets.createIndex({ "current_classification.node_id": 1 })
```

Este es un ejemplo de la query:

```
// Solo tickets de este nodo específico
db.tickets.find({
  "current_classification.node_id": "area_mantenimiento"
})
// NO incluye mantenimiento_zonas_comunes, etc.
```

3.3.6. *Índice consulta principal compuesta.*

Índice compuesto para filtros combinados, soportar la query principal del sistema que combina todos los filtros. Genera un máximo performance para el caso de uso más común. El siguiente es la creación del índice:

```
db.tickets.createIndex({
  "current_state": 1,
  "current_classification.path": 1,
  "created_at": 1,
  "closed_at": 1
})
```

Este es un ejemplo de la query:

```
db.tickets.find({
  current_state: "open",
  "current_classification.path": "area_mantenimiento",
  $or: [
    { created_at: { $lt: endDate }, closed_at: { $gte: startDate } },
    { created_at: { $lt: endDate }, closed_at: null }
  ]
})
```

Orden de campos (de más a menos selectivo):

- **current_state** (primero)
    - Reduce a ~20% de tickets

    - Alta frecuencia de uso
- **current_classification.path** (segundo)
    - Reduce a ~5-30% adicional
    - Depende de la jerarquía
- **created_at** (tercero)
    - Ordena temporalmente
    - Permite range queries eficientes
- **closed_at** (cuarto)
    - Filtra por cierre
    - Puede ser NULL

3.4. Índices para la colección '**ticket_history**'.

Esta colección crece indefinidamente y requiere índices eficientes para no degradar con el tiempo.

3.4.1. Índice histórico por ticket.

Consulta de histórico por ticket. Cuyo propósito es obtener todo el histórico de un ticket ordenado cronológicamente y reconstruir el estado de un ticket en cualquier momento del tiempo. El siguiente es la creación del índice:

```
db.ticket_history.createIndex({ "ticket_id": 1, "timestamp": 1 })
```

Este es un ejemplo de la query:

```
// Todo el histórico
db.ticket_history.find({
  ticket_id: ObjectId("...")
}).sort({ timestamp: 1 })

// Última acción
db.ticket_history.find({
  ticket_id: ObjectId("...")
}).sort({ timestamp: -1 }).limit(1)

// Acciones en rango
db.ticket_history.find({
  ticket_id: ObjectId("..."),
  timestamp: { $gte: startDate, $lt: endDate }
})
```

Orden de campos:

- **ticket_id primero:** Filtra 99.9% de documentos irrelevantes
- **timestamp segundo:** Permite ordenamiento sin sort en memoria

3.4.2. Índice de métricas agregadas.

índice de consulta por tipo de acción y fecha, esto calcula métricas agregadas en rangos de fecha, además reporta por tipo de acción. El siguiente es la creación del índice:

```
db.ticket_history.createIndex({ "action_type": 1, "timestamp": 1 })
```

Este es un ejemplo de la query:

```
// Reaperturas en enero
db.ticket_history.find({
  action_type: "state_change",
  "changes.old_value": "closed",
  "changes.new_value": "open",
  timestamp: { $gte: startDate, $lt: endDate }
})

// Cierres en enero
db.ticket_history.find({
  action_type: "state_change",
  "changes.new_value": "closed",
  timestamp: { $gte: startDate, $lt: endDate }
})
```

Orden de campos:

- **action_type primero:**
  - Cardinalidad media (5 valores)
  - Selectividad alta para métricas específicas (~20% por tipo)
- **timestamp segundo:**
  - Permite filtros de rango eficientes
  - Ordena cronológicamente

3.4.3. Limpieza de datos antiguos.

Índice de consulta por timestamp, esto es funcional para archivar o eliminar datos históricos antiguos cumpliendo políticas de retención de datos. El siguiente es la creación del índice:

```
// Eliminar histórico > 5 años
db.ticket_history.deleteMany({
  timestamp: { $lt: ISODate("2020-01-01") }
})

// Contar acciones antiguas
db.ticket_history.countDocuments({
  timestamp: { $lt: cutoffDate }
})
```

Este es un ejemplo de la query:

```
// Proceso batch mensual de archivado
db.ticket_history.aggregate([
  {
    $match: {
      timestamp: { $lt: archiveDate }
    }
  },
```

```
  { $out: "ticket_history_archive" }
])  action_type: "state_change",
  "changes.new_value": "closed",
  timestamp: { $gte: startDate, $lt: endDate }
})
```

4. Consultas y Filtros.

Ver implementación en:
https://github.com/drawnick1214/DataModelingMongoDB/blob/master/mongodb/04-queries.js

4.1. Lógica de Filtros.

4.1.1. Filtro A: Rango de fechas.

Tickets abiertos o en gestión durante [startDate, enDate].

```
{
      $or: [
            { created_at: { $lt: endDate }, closed_at: { $gte: startDate } },
            { created_at: { $lt: endDate }, closed_at: null }
      ]
}
```

4.1.2. Filtro B: Estado.

- Caso simple: Estado actual -> **`current_state`**: '**`open`**'
- Caso complejo: Estado al final del periodo, requiere consultar histórico.

4.1.3. Filtro C: Clasificadores Jerárquicos.

Búsqueda por path incluye nodo padre y todos sus descendientes.

```
{ "current_classification.path": "area_mantenimiento" }
// Incluye: area_mantenimiento, mantenimiento_zonas_comunes, etc.
```

4.2. Consultas implementadas.

4.2.1. Lista de casos.

Calcula los tickets que existieron durante el periodo con filtros aplicados.

```
getTickets({ startDate, endDate, state, classifierIds, page, pageSize })
```

4.2.2. Cantidad de reaperturas.

Calcula los eventos donde state: closed → open (un ticket puede reabrirse múltiples veces).

```
countReopenings({ startDate, endDate, classifierIds })
```

4.2.3. *Cantidad de ingresos.*

Calcula tickets con created_at en el rango.

```
countTicketCierres({ startDate, endDate, classifierIds })
```

4.2.4. *Cantidad de cierres.*

Calcula eventos donde **`state`** es **`closed`**.

```
getTicketActions({ ticketId, actionTypes, startDate, endDate, page, pageSize })
```

4.2.5. *Lista de acciones.*

Calcula histórico de acciones con filtros.

```
getTicketActions({ ticketId, actionTypes, startDate, endDate, page, pageSize })
```

5. Dificultades y limitaciones.

5.1. Dificultades principales.

5.1.1. *Estado Histórico:*

- **Problema:** Determinar estado de un ticket en fecha pasada es computacionalmente costoso.
- **Solución:** Agregación con $lookup para encontrar último cambio de estado antes de endDate.
- **Limitación:** Con millones de tickets puede tomar varios segundos. Alternativa: materializar snapshots diarios.

5.1.2. *Jerarquías dinámicas:*

- **Problema:** Si un nodo cambia de padre, tickets mantienen path antiguo.
- **Solución:** Proceso batch de migración (ver código en documentación extendida).

5.1.3. *Múltiples raíces:*

- **Problema:** Filtrar por clasificadores de diferentes raíces requiere $and.
- **Solución:** Validación + construcción correcta del filtro.

5.2. Limitaciones conocidas.

- **Consistencia:** No implementa transacciones ACID (requiere replica set para producción).
- **Concurrencia:** Sin optimistic locking (recomendado para producción).
- **Búsqueda texto:** No incluida (considerar Elasticsearch).
- **Volumen extremo:** Consultas analíticas en 10M+ tickets → considerar data warehouse.

6. Implementación y uso.

6.1. Setup Rápido.

mongosh < mongodb/01-create-collections.js

mongosh < mongodb/02-create-indexes.js

mongosh < mongodb/03-seed-data.js

mongosh < examples/query-examples.js

6.2. Ejemplos de uso.

```
// Conectar a MongoDB
use capta_tickets_db

// Cargar funciones
load('mongodb/04-queries.js')

// Ejecutar consulta
await getTickets({
startDate: ISODate("2025-01-01"),
endDate: ISODate("2025-02-01"),
state: "open",
classifierIds: ["area_mantenimiento"],
page: 1,
pageSize: 50
})
```

7. Repositorio.

Por favor acceda al repositorio disponible en:

https://github.com/drawnick1214/DataModelingMongoDB.git