**Composite adentro**

Ese profe, con su *"esto afuera se hace distinto, y mejor"*, tiró una verdad que parece mínima pero es oro puro.

**Entonces, vamos al grano. ¿Cómo se resuelve esto en sistemas reales?**

Desde la experiencia: muchas veces se hace **con menos patrones y más sentido común.**

**El problema:**

* Un Usuario tiene uno o más Roles
* Cada Rol tiene varios Permisos
* Pero también, **cada Usuario puede tener permisos adicionales (especiales)** (fuera de los roles)

**En la vida real, ¿cómo se hace?**

**1. Se rompe el Composite (en la práctica)**

En sistemas reales, no se representa todo como un árbol complejo. **Se separan bien las entidades:**

* Permisos (elementales, atómicos)
* Roles (colección de permisos)
* Usuarios (colección de roles + colección de permisos individuales)

Y se consulta así:

PermisosDelUsuario =

UNION(

SELECT Permisos FROM Roles WHERE Rol IN (RolesDelUsuario),

SELECT Permisos FROM Usuario WHERE UsuarioID = ?

)

¡Listo! No hay Composite, ni recursión, ni locura. **Sólo uniones bien claras**.

**2. Se evita la recursividad**

La gran mayoría de los sistemas grandes **evita tener roles dentro de roles**, porque eso hace que la lógica sea:

* Difícil de auditar
* Difícil de predecir
* Y muy costosa en performance

Por eso: **Roles = conjuntos planos de permisos.** Y si necesitás combinarlos, armás un nuevo rol con esa combinación, y listo.

**3. ¿Y qué patrón se usa entonces?**

No es exactamente un patrón formal como Composite. Pero se podría decir que es una forma simplificada de **Policy Object**, o incluso una especie de **Data-Driven RBAC (Role-Based Access Control)**.

Es más: muchos frameworks modernos ya trabajan con eso:

* **ASP.NET Core Identity**
* **Spring Security**
* **Django** Todos usan una estructura parecida: Users → Roles → Permissions
* permisos extras directamente asociados al usuario.

**¿Entonces el profe tenía razón?**

¡Sí! **En el mundo real no se usa Composite para esto.** Porque se vuelve inmanejable y sobrediseñado. Se usa un diseño simple, plano, con uniones y una lógica clara.

*La solución en el TFC fue mucho más sensata y cercana a lo que se hace realmente*.

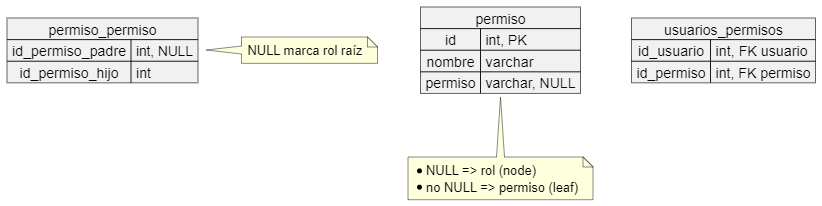
## Defectos de diseño en una gestión de roles y permisos especiales

Este caso muestra una serie de errores de diseño que pueden rastrearse al uso inadecuado del modelo relacional:

1. Desnaturalización de estructuras jerárquicas al forzar su representación relacional.
2. Introducción de complejidad artificial mediante CTEs recursivos.
3. Reducción de la mantenibilidad por aumento innecesario del código.
4. Falta de correspondencia entre diseño lógico y estructura de persistencia.

**Desnaturalización de estructuras jerárquicas al forzar su representación relacional**

En el diseño original, se emplean tres tablas:

[[1]](#footnote-1)

Con estas tablas, la aplicación ejecuta esta consulta recursiva para reconstruir el árbol de permisos de un rol dado:

WITH PermissionTree AS

(

SELECT ph.id\_permiso\_padre AS ParentId,

ph.id\_permiso\_hijo AS ChildId

FROM permiso\_permiso ph

WHERE ph.id\_permiso\_padre IS NULL

OR ph.id\_permiso\_padre = @RootRoleId

UNION ALL

SELECT ph.id\_permiso\_padre AS ParentId,

ph.id\_permiso\_hijo AS ChildId

FROM permiso\_permiso ph

JOIN PermissionTree pt

ON pt.ChildId = ph.id\_permiso\_padre

)

SELECT pt.ParentId,

pt.ChildId,

p.id AS PermissionId,

p.nombre AS PermissionName,

p.permiso AS PermissionType

FROM PermissionTree pt

JOIN permiso p

ON p.id = pt.ChildId;

**Nota**: esta construcción SQL se apoya en una **CTE** (Common Table Expression) para iterar sobre la jerarquía.

A continuación, el código cliente interpreta PermissionType (columna permiso) para distinguir nodos de hojas:

string code = reader["permiso"] as string;

Componente c = string.IsNullOrEmpty(code)

? new RoleNode() // nodo (rol)

: new PermissionLeaf(); // hoja (permiso especial)

**Introducción de complejidad artificial mediante CTEs recursivos**

A continuación, se presentan las principales fallas estructurales observadas en el modelo relacional utilizado para gestionar roles y permisos, analizadas desde la perspectiva del diseño orientado a objetos y la eficiencia en la representación de jerarquías.

**Semántica implícita en la estructura de datos.** El modelo relacional utilizado mezcla roles y permisos especiales en una misma tabla (permiso), utilizando el valor NULL en la columna permiso como señal implícita para distinguir nodos (roles) de hojas (permisos especiales). Esta estrategia introduce una ambigüedad semántica que obliga a que el código cliente interprete los valores nulos en lugar de apoyarse en una tipificación clara y explícita. Esto rompe con el principio de claridad del dominio y dificulta la validación estática del modelo.

**Recursividad forzada mediante consultas CTE.** Para reconstruir la jerarquía de roles y permisos, se recurre a una expresión común de tabla (CTE) recursiva que emula una estructura de árbol. Este enfoque, aunque técnicamente válido, implica una complejidad considerable tanto en la construcción como en el mantenimiento de las consultas. Además, la recursividad explícita en SQL se vuelve frágil ante cambios en la jerarquía o en la lógica de los permisos, y no refleja naturalmente la topología de los datos.

**Acoplamiento entre lógica de presentación y esquema de datos.** La lógica necesaria para interpretar si un nodo es un rol o un permiso especial está embebida en el código cliente, lo que genera un fuerte acoplamiento entre la estructura del esquema y la lógica de presentación o negocio. Este acoplamiento dificulta el mantenimiento, ya que cualquier cambio en la forma de distinguir nodos y hojas requiere modificaciones tanto en la base de datos como en el código de la aplicación.

**Violación del principio de intención única.** El esquema mezcla responsabilidades heterogéneas en una misma tabla. Desde el punto de vista de la modelación, sería más claro y coherente separar en tablas distintas los conceptos de Role, Permission y RolePermission. La combinación de todos en un mismo esquema relacional obliga a reconstruir manualmente la jerarquía y su tipificación, tanto en SQL como en código, lo que va en contra de la intención única y reduce la legibilidad del diseño.

Estas fallas no solo introducen complejidad técnica innecesaria, sino que también afectan la mantenibilidad y claridad semántica del sistema. La lógica jerárquica, al estar implementada en un esquema plano, exige soluciones técnicas que terminan desviando el foco del dominio hacia los detalles de implementación.

En la siguiente sección se explora una alternativa basada en modelos documentales que resuelve estas deficiencias de forma natural, alineando la persistencia con la topología de los datos y reduciendo significativamente la complejidad del diseño.

**Reducción de la mantenibilidad por aumento innecesario del código**

Frente a las limitaciones del modelo relacional descritas anteriormente, una alternativa más alineada con la topología jerárquica del dominio es la utilización de un modelo documental, como MongoDB o LiteDB. Este enfoque permite representar estructuras de árbol de manera nativa, con una semántica clara y sin necesidad de recursión ni tipificación implícita.

Consideremos el mismo caso: un usuario con múltiples roles, y cada rol con una lista de permisos asociados, además de contar con permisos especiales fuera de rol. En un modelo documental, toda esta estructura puede representarse como un único documento embebido, lo que refleja directamente la jerarquía sin necesidad de un esquema plano ni consultas recursivas.

{

"\_id": ObjectId("64b2c3..."),

"username": "alice",

"roles": [

{

"id": 1, "name": "Manager",

"permissions": [

"CREATE\_REPORT",

"VIEW\_REPORT",

"APPROVE\_EXPENSE"

]

},

{

"id": 2, "name": "Auditor",

"permissions": [

"VIEW\_REPORT",

"EXPORT\_AUDIT"

]

}

],

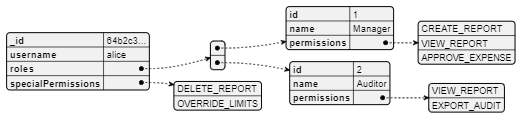
"specialPermissions": [

"DELETE\_REPORT",

"OVERRIDE\_LIMITS"

]

}

[[2]](#footnote-2)

Esta representación distingue explícitamente entre los permisos heredados por rol y los permisos especiales, sin recurrir a campos nulos ni reglas de interpretación en el código. La semántica del dominio está completamente contenida en la estructura del documento.

**Código en C# (LiteDB)**

En una implementación con LiteDB, la recuperación de los permisos efectivos de un usuario es directa:

var user = db.GetCollection<User>("users").FindOne(u => u.Username == "alice");

var rolePerms = user.Roles.SelectMany(r => r.Permissions).Distinct();

var effectivePerms = rolePerms.Union(user.SpecialPermissions).ToList();

Esta lógica, clara y expresiva, contrasta con la complejidad del modelo relacional, donde el ensamblado jerárquico requiere múltiples consultas, CTEs y lógica condicional para diferenciar tipos de nodo.

**Ventajas del modelo documental**

* **Representación nativa de jerarquías:** la estructura árbol se conserva directamente en el documento, sin necesidad de reconstrucción ni consultas recursivas.
* **Semántica explícita:** la distinción entre roles y permisos especiales está dada por la estructura del documento, no por convenciones implícitas o nulos.
* **Consulta unificada:** basta con leer un solo documento para obtener toda la información relevante, eliminando la necesidad de JOINs o recorridos de tablas.
* **Desacoplamiento de lógica de dominio y persistencia:** el modelo refleja fielmente el diseño de clases en C#, permitiendo una deserialización directa sin lógica adicional.

Este modelo no solo resuelve las deficiencias observadas en el enfoque relacional, sino que además simplifica el mantenimiento, facilita la evolución del sistema y mejora el rendimiento en operaciones comunes como la lectura de permisos efectivos.

En la siguiente sección, se comparan ambos enfoques en detalle para mostrar cómo el modelo documental supera las barreras impuestas por la tradición relacional en dominios jerárquicos.

**Falta de correspondencia entre diseño lógico y estructura de persistencia**

La siguiente tabla resume las diferencias clave entre el enfoque relacional tradicional (basado en CTEs y composición recursiva) y el modelo documental, aplicado a la gestión de roles y permisos especiales. Se analizan aspectos fundamentales como la representación de jerarquía, la semántica de nodos y hojas, la complejidad de consultas, y la mantenibilidad del sistema.

|  |  |  |
| --- | --- | --- |
| **Aspecto** | **Modelo relacional (CTE + Composite)** | **Modelo documental (NoSQL)** |
| **Modelado de jerarquía** | Árbol emulado mediante consultas recursivas (CTE), sin soporte nativo | Árbol representado directamente en el documento, sin reconstrucción |
| **Distinción nodo/hoja** | Implícita (basada en NULL en el campo permiso y lógica en el cliente) | Explícita: roles y permisos definidos con estructuras separadas (e.g., permissions, specialPermissions) |
| **Consultas** | Requieren CTE recursivas + múltiples JOINs; alto costo y complejidad | Lectura directa de un único documento; consultas simples y eficientes |
| **Lógica en el cliente** | Necesaria para interpretar tipo de nodo y ensamblar la jerarquía desde resultados planos | Innecesaria: deserialización directa a objetos con estructura completa |
| **Mantenibilidad** | Fragilidad alta ante cambios en jerarquía o tipado; consultas extensas | Modificaciones simples (agregar/eliminar roles o permisos implica editar el documento JSON) |
| **Alineación topológica** | Forzada: topología de árbol acíclico representada artificialmente en un esquema plano | Natural: la estructura del documento respeta la topología del dominio |

**Conclusión de la comparación**

Esta comparación evidencia que el modelo documental no solo **resuelve las ineficiencias** del enfoque relacional tradicional, sino que también **alinea la estructura de persistencia con la semántica del dominio**, respetando la topología jerárquica sin ambigüedades ni lógica redundante.

Mientras el modelo relacional requiere reconstruir jerarquías a partir de estructuras planas y depender en convenciones frágiles (como nulos), el modelo documental permite una representación directa, clara y expresiva. Esta coherencia estructural reduce la complejidad del sistema, mejora su mantenibilidad y valida la hipótesis de que **la topología debe guiar la elección del modelo de persistencia**.

**Conclusión**

El análisis del sistema de gestión de roles y permisos revela de forma clara la importancia de **alinear el modelo de persistencia con la topología de los datos**. El modelo relacional tradicional, aunque ampliamente adoptado, introduce una complejidad innecesaria cuando se aplica a estructuras jerárquicas. La necesidad de consultas recursivas (CTE), la distinción semántica basada en nulos, y la lógica interpretativa embebida en el código cliente son síntomas de una **disonancia estructural** entre el modelo de datos y la realidad del dominio.

En contraste, el modelo documental demuestra ser más **natural, claro y eficiente** para representar jerarquías acíclicas como la que se encuentra en este caso. Su capacidad para encapsular toda la jerarquía en un único documento, con una semántica explícita y alineada con el diseño orientado a objetos, elimina la recursión, simplifica la lógica de negocio y mejora la mantenibilidad del sistema.

Este caso concreto valida empíricamente la **hipótesis del criterio basado en topología**:

“Cuando los datos conforman una jerarquía acíclica, el modelo documental maximiza claridad, eficiencia y capacidad de evolución.”

Además, refuerza una premisa central del diseño orientado a objetos: que **la estructura del dominio debe prevalecer sobre la inercia tecnológica**. Elegir un modelo de persistencia que respete la topología del problema no solo mejora el rendimiento técnico, sino que también honra la semántica del sistema y facilita su evolución futura.

1. **PlantUML:**

   @startuml

   map permiso\_permiso {

   id\_permiso\_padre => int, NULL

   id\_permiso\_hijo => int

   }

   note right of permiso\_permiso

   NULL marca rol raíz

   end note

   map permiso {

   id => int, PK

   nombre => varchar

   permiso => varchar, NULL

   }

   note bottom of permiso

   \* NULL => rol (node)

   \* no NULL => permiso (leaf)

   end note

   map usuarios\_permisos {

   id\_usuario => int, FK usuario

   id\_permiso => int, FK permiso

   }

   @enduml [↑](#footnote-ref-1)
2. **PlantUML:**

   @startjson

   {

   "\_id": "64b2c3...",

   "username": "alice",

   "roles": [

   {

   "id": 1, "name": "Manager",

   "permissions": [

   "CREATE\_REPORT",

   "VIEW\_REPORT",

   "APPROVE\_EXPENSE"

   ]

   },

   {

   "id": 2, "name": "Auditor",

   "permissions": [

   "VIEW\_REPORT",

   "EXPORT\_AUDIT"

   ]

   }

   ],

   "specialPermissions": [

   "DELETE\_REPORT",

   "OVERRIDE\_LIMITS"

   ]

   }

   @endjson [↑](#footnote-ref-2)