Campus Santa Fe

4. Normalización de la base de datos del reto apoyándose en recursos de IA generativa

Erick Alonso Morales Dieguez | A01029293

Fausto Izquierdo Véjar | A01785221

Sergio Jiawei Xuan | A01784695

Jueves 29 de mayo 2025

Profesores:

Esteban Castillo Juarez

Gilberto Echeverría Furió

Octavio Navarro Hinojosa

Contexto

Desde un inicio hemos trabajado con un formato para desarrollar la base de datos llamado DBML (Data Base Markup Language), visualizando el ERD con el aplicativo web dbdiagram.io.

Además, al desarrollar la base de datos, hemos revisado constantemente que cumpla con ciertos criterios de normalización extraídos de la materia de IOT, donde nos enseñaron teoría sobre ello.

Proceso

Se hizo una versión inicial de la base de datos en la primera tarea, de la cual tuvimos retroalimentación las subsecuentes 2 semanas.

Tras las últimas revisiones de la base de datos con el profesor Esteban aplicamos los cambios que consideramos adecuados, lo que nos llevó a nuestra versión 3 de la base de datos.

Extrajimos los criterios de normalización de la presentación Normalización ubicada en: https://experiencia21.tec.mx/courses/575641/pages/diapositivas-de-bases-de-datos?module_item_id=34864631

Le pedimos a la IA analizar nuestra versión 3, teniendo en cuenta los criterios de normalización, para identificar los puntos de mejora

El modelo utilizado fue ChatGPT o4-mini. Tras pensar 23 segundos esto fue lo que redactó:

La versión “bare-metal” del esquema V3 adolece de varios problemas de diseño que, aunque no violen de forma estricta las reglas formales, rompen los principios prácticos de normalización:

1. Dependencias transitivas en la tabla enemy_types

- Se mezclan atributos de enemigo genérico (base_hp, base_damage, etc.) con detalles de jefe vía la FK boss_id.
- Cualquier consulta que necesite información de jefe (como max_hp o description) pasa por esa llave, pero esos atributos no dependen directamente de la PK (enemy_id), sino de boss_id.
- Esa relación crea una dependencia transitoria:
  enemy_id → boss_id → boss_details.*
- En consecuencia, si cambian los datos del jefe, hay riesgo de inconsistencias si no se actualizan correctamente en ambas entidades.

2. Campo redundante y ambigüedad semántica: is_boss

- El booleano is_boss duplica la información que ya aporta boss_id (NULL versus valor).
- Esto genera anomalías de actualización: ¿qué pasa si alguien marca is_boss = true pero olvida asignar boss_id, o viceversa?
- Viola el principio de “no almacenar la misma información dos veces” y complica las reglas de integridad.

3. Enums incrustados en lugar de tablas de dominio

- Valores como event_type_enum, weapon_slot_enum, upgrade_type_enum, boss_result_enum y el libre item_type en shop_purchases están definidos en código o como tipos de columna.
- Al crecer el proyecto, añadir un nuevo tipo de evento o de arma exige cambiar el DDL y redeploy de la app, en lugar de un simple INSERT en una tabla de lookup.
- Esto va contra la flexibilidad y crea “siloblocks” de lógica de negocio dispersos: mantenimiento y reporting se vuelven más costosos.

4. Atributos de contexto sin lookup ni FK

- Campos como player_events.weapon_type o shop_purchases.item_name quedan sueltos, sin referencia a un catálogo de ítems.
- Se fomenta la aparición de valores erráticos (“sword”, “espada”, “swd”), con el consiguiente riesgo de anomalías de inserción y actualización.

5. Ausencia de subtipo explícito para jefes

- Idealmente, “enemigo” y “jefe” deberían modelarse con herencia o mediante dos tablas (padre: enemy_types; subtipo: boss_details).
- El diseño actual obliga a almacenar columnas nulas en todas las filas que no son jefes, y a gestionar lógica de filtrado en cada consulta.

En suma, aunque cada tabla tiene su PK y no hay violaciones graves de 1FN o 2FN, la mezcla de enums estáticos, la duplicación de información (is_boss vs. boss_id), las dependencias transitivas ocultas y la falta de tablas de dominio—hacen que el modelo no sea mantenible ni escalable. No es un “3FN puro” en la práctica empresarial: sufre anomalías de actualización, inserción y borrado que terminan encareciendo los ciclos de desarrollo y operación.

Una vez identificados los puntos de mejora, le pedimos al modelo ChatGPT o4-mini-high desarrollar un V4 del esquema, dándole el contexto previo además de también adjuntar los criterios para normalización.

Nos entregó el esquema en formato dbml y posteriormente lo visualizamos con dbdiagram.io (Tanto el esquema V1, V2, V3 y V4 están adjuntos en la carpeta de esta actividad).

Posterior a la normalización, le pedimos que describiera los cambios que había implementado. Esto fue lo que contestó:

---

1. Gestión de dominios (ENUMs → tablas lookup)

Problema pre-3FN	Impacto	Solución post-3FN
ENUMs embebidos en cada tabla (event_type, weapon_slot, etc.)	Cada vez que sumabas un nuevo valor, tocaba hacer ALTER TYPE o recompilar, con riesgo de replicar valores distintos en distintas tablas.	Tablas de dominio (event_types, weapon_slots, …) centralizan valores válidos y garantizan integridad referencial, facilitando extensión y reporting.

Beneficio clave: Cambios de negocio (p.ej. nuevos tipos de evento) se implementan insertando una fila en la tabla correspondiente, sin tocar la estructura de las tablas de datos transaccionales.

2. Eliminación de dependencias parciales y transitivas

- **Antes:**
    - enemy_types incluía is_boss, boss_id y campos propios de jefes (max_hp, description) → datos nulos para no-jefes y duplicación de “floor” entre enemigos y jefes.
    - Tablas como player_upgrades y weapon_upgrades_temp mezclaban lógicas de estado permanente y temporal con claves compuestas no explícitas.
- **Después:**
    - **Supertipo/Subtipo**: enemy_types + boss_details (PK=enemy_id) sacan los atributos de jefe a su propia tabla.
    - **Dependencias claras**: todo atributo no-clave depende directamente de la PK, sin columnas que dependan de otras no-clave.
    - **Claves compuestas explícitas**: índices PK en player_upgrades, equipped_weapons, weapon_upgrades_temp aseguran unicidad y evitan duplicados accidentales.

Beneficio clave: Cero valores nulos innecesarios, reducción de espacio, mejora de consultas JOIN y mantenimiento de reglas de integridad.

3. Separación de catálogos vs. transacciones

Pre-3FN	Post-3FN
Mescla catálogo de ítems (item_type como texto libre) con transacciones (shop_purchases)	item_types como lookup, FK desde shop_purchases
Tablas de eventos (player_events, boss_encounters) con campos de dominio inline	Relaciones FK a event_types, boss_results

Beneficio clave: Auditoría y análisis de BI—puedes consultar fácilmente cuántas compras de cada tipo de ítem, sin filtrar strings arbitrarios.

4. Integridad referencial reforzada

- **Clave foránea obligatoria**: todas las FKs son NOT NULL cuando corresponda (p.ej. run_id, user_id), lo cual evita registros huérfanos.
- **Unicidad garantizada**: índices PK compuestos previenen duplicados lógicos (p.ej. un arma equipada en el mismo slot dos veces).
- **Actualizaciones en cascada/controladas**: separar dominios permite políticas de borrado/actualización más estrictas.

5. Escalabilidad y mantenimiento

- **Flexibilidad**: para añadir un nuevo tipo de upgrade, basta con insertar en upgrade_types; ningún ALTER TABLE en tablas de negocio.
- **Performance**: reducir ancho de fila y redundancia mejora cacheo, estadísticas de PostgreSQL y planes de ejecución.

- **Documentación implícita**: cada lookup table actúa como documentación viva de valores válidos.

Conclusión

Al normalizar a 3FN:

1. **Redujimos redundancias** y valores nulos.
2. **Centralizamos dominios** para acelerar cambios y análisis.
3. **Fortalecimos la integridad referencial**, reduciendo errores en producción.
4. **Mejoramos la escalabilidad** y mantenibilidad del modelo de datos.

Este rediseño garantiza que futuros cambios—tanto de reglas de negocio como de reporting—sean rápidos, predecibles y con mínimo riesgo de inconsistencias.
