Práctica 7 – Normalización

# Preguntas de repaso

1. Las anomalías de inserción, borrado y modificación aparecen cuando uno quiere guardar tablas que representan NATURAL JOINS enteras, sin haber hecho descomposición.
   1. **Inserción:** hay veces que se quiere ingresar una tupla con una cantidad acotada de atributos y el resto de ellos queda incompleto; en ese caso se generan NULLs. Otro caso es cuando queremos agregar todos los atributos relacionados a un atributo en particular que sería clave de ellos y al haber tantos repetidos se pueden ingresar errores (por ejemplo, un empleado también ingresa el código único de su proyecto y su título, su descripción, etc., pero ese título puede tener errores dado un mismo código único que aparece en varias entradas, como ser ‘Presentación’ y ‘Presentasion’).
   2. **Borrado:** hay veces que un atributo que no es clave en la relación a nivel general, en realidad sí es clave de alguno otro y al eliminar una entrada respecto de la clave general se pierde ese registro. Por ejemplo, si borramos el único empleado que se corresponde al proyecto con código 2, ese proyecto también deja de existir.
   3. **Modificación:** si se repite un atributo muchas veces (redundancia) es necesario tener mucho cuidado si se requiere hacer una actualización porque podría no afectar a todas las entradas y quedar inconsistente (modificar el nombre del proyecto con código 5).
2. Hay varios problemas que pueden traer los NULLs
   1. Espacio ocupado innecesariamente, en particular en columnas que raramente tienen un dato almacenado.
   2. Genera dudas respecto a su significado: ¿quiere decir que no existe valor porque no aplica o que todavía no se agregó y está pendiente, u otra cosa?
   3. Los NATURAL JOINs con tablas que tienen muchos NULLs terminan simulando un OUTER JOIN aunque no se quiera.
   4. Las funciones de agregación COUNT, SUM, MAX, etc. no siempre funcionan como se espera y hay que agregar casos borde con CASE WHEN o similares.

Las tuplas espúreas aparecen cuando una relación/tabla es mal descompuesta en subrelaciones o tablas más chicas y no se asignaron correctamente las claves primarias y foráneas, por lo que al hacer un JOIN en reversa para generar la tabla original aparecen duplicados en atributos que generan tuplas que originalmente no existían. Se previene forzando que haya claves primarias y foráneas correctas.

1. Una dependencia funcional **X** 🡪 **Y** es una restricción sobre dos conjuntos de atributos X e Y de una relación/tabla R y satisface que para cada par cualquiera de tuplas en la relación, si tienen el mismo valor en X entonces también lo deben tener en Y. Por eso Y depende (funcionalmente) de X. Las dependencias funcionales deben ser determinadas por quien administra la base de datos, dado que requiere un entendimiento de su semántica y cómo interactúan los valores y atributos dentro de la relación.
2. Porque desconocemos la semántica por sobre los atributos, dependencias entre ellos, valores esperados, etc.
3. La 2FN pide que todos los atributos no primos dependan de forma completa de ~~la clave principal~~ **cualquier clave candidata (inclusive la principal)**, es decir, realmente dependa de cada uno de esos atributos que componen la clave principal. Al pedir esto, evita escenarios en donde uno o más atributos están generando redundancia dado que simplemente se podría hacer una nueva tabla con esos atributos, eligiendo como su clave primaria ese subconjunto de la clave principal de la tabla grande original.

\* Puede ser que la clave principal tenga un solo atributo y haya alguna candidata que tenga más de uno y que esa candidata vaya en contra de la 2FN (porque hay un atributo no primo que depende parcialmente de esa candidata compuesta). En ese caso, la relación NO ESTÁ EN 2FN; es necesario descomponer.

1. La 3FN pide, además de estar en 2FN, que ningún atributo no primo dependa transitivamente de la clave principal (es decir, que suceda que hay un atributo no primo B que depende de otro atributo no primo A y uno depende funcionalmente del otro, por lo que se podría hacer PK → A y A → B). Pedir esto evita que haya un escenario similar al de 2FN de redundancia, pero esta vez un atributo no primo A podría usarse como clave principal en otra tabla, sacando al atributo no primo B de la tabla original y dejándolo en ésta segunda tabla.

\*1FN – Sólo pide que haya atributos atómicos. Hay varias formas de resolverlo pero en la cátedra recomiendan separar el atributo no atómico en otra tabla, junto a lo que era la clave en esa tabla original (y algún identificador, tipo Teléfono 1, 2, 3, etc.).

\* Las 3FN y Boyce-Codd están pensadas para que eviten el problema de Lossless Joins.

Ej. de descomposición para que quede Lossless Join:

R = { A, B, C } en donde DFs = { { A, B 🡪 C }, { C 🡪 B} }

Se puede descomponer en R1 = { A, C } y R2 = { C, B }. Esto cumple la propiedad Lossless Join.

# Ejercicios

1. Las tuplas espúreas aparecieron porque la tabla fue mal dividida en EMP\_LOCS. Ahí la clave tuvo que haber sido por ejemplo el DNI, no el nombre del empleado ni la ubicación del proyecto porque no se puede determinar un nombre por una ubicación ni viceversa, puede haber duplicados de ambos lados (una ubicación tiene muchos nombres y un nombre muchas ubicaciones). Cuando el campo UbicacionProyecto se usa como clave vinculante en el JOIN entonces aparecen asignaciones incorrectas a nivel DNI, lo que no debería pasar.
2. Las claves deberían ser las más abarcativas, es decir, { NumeroPedido, CodigoObjeto }. Hacer una NATURAL JOIN básicamente le agregaría a cada entrada de LINEA\_PEDIDO tres columnas/atributos más, pertenecientes a PEDIDO (FechaPedido, NumeroCliente, CosteTotal). Esto tendría mucha redundancia, porque podría haber cien productos por pedido y cada uno repetiría los tres campos mencionados. No estaría ni siquiera en 2FN, si definimos que la nueva clave es { NumeroPedido, CodigoObjeto }, dado que esos tres atributos que vinieron de PEDIDO son atributos no primos que dependen **parcialmente** y no completamente de la clave principal (dependen sólo de NumeroPedido).

Las DFs resultantes son:

{ NumeroPedido → FechaPedido, NumeroCliente, CosteTotal }

{ NumeroPedido, CodigoObjeto → CantidadSolicitada, PrecioTotal, PorcentajeDto }

1. R está en 1FN porque los valores (se supone) son atómicos.

R no está en 2FN porque no sucede que todos los atributos no primos (FechaVenta, PorcentajeComision, Descuento) dependen de forma completa de los atributos de la clave primaria. Por ejemplo { NumeroVendedor → Descuento } implica que Descuento depende parcialmente de la clave principal.

R no está en 3FN porque no está en 2FN.

Para normalizar esta tabla se podría separar en las siguientes tablas:

R1 = { NumeroCoche, NumeroVendedor, FechaVenta }

R2 = { FechaVenta, Descuento }

R3 = { NumeroVendedor, PorcentajeComision }

Las DFs serían:

{ NumeroCoche, NumeroVendedor → FechaVenta }

{ FechaVenta → Descuento }

{ NumeroVendedor → PorcentajeComision }

1. ¡¿Suponemos que TituloLibro es clave principal?! ¿Cuál es la clave si no?

R está en 1FN porque (se supone) sus atributos contienen valores atómicos.

R está en 2FN porque (se supone) la clave principal es TituloLibro y contiene un solo atributo, así que no puede haber atributos no primos que sean parcialmente dependientes de la clave principal…

R no está en 3FN porque hay atributos no primos que se pueden poner entre medio de la clave principal y otro atributo no primo y generar transitividad (por ejemplo TituloLibro → Autor y Autor → AfiliacionAutor).

No hay pasos para aplicar sobre la clave principal porque ya está compuesta de un solo atributo, pero sí se pueden aplicar pasos a algunos atributos no primos. Dado que ListaPrecio depende funcionalmente de TipoLibro y AfiliacionAutor igualmente de Autor, se pueden mover ambos atributos a relaciones más chicas.

R1 = { TituloLibro, Autor, TipoLibro, Editorial }

R2 = { Autor, AfiliacionAutor }

R3 = { TipoLibro, ListaPrecio }

1. DISCO(NumeroSerie, Fabricante, Modelo, Version, Capacidad, Distribuidor)

DF1 = { NumeroSerie, Fabricante → Modelo, Version, Capacidad, Distribuidor }

DF2 = { Modelo → Fabricante }

DF3 = { Version → Modelo }

DF4 = { Modelo, Fabricante → Capacidad } = { Modelo → Capacidad }

Se podría descomponer y pasar a 3FN:

R1 = { NumeroSerie, Fabricante, Version, Distribuidor }

R2 = { Version, Modelo }

R3 = { Modelo, Fabricante, Capacidad }

1. Si denotamos a { IdCoche, Extra } como clave primaria de R, la relación no está en 3FN sólo por no estar en 2FN.

R no está en 2FN porque hay atributos no primos que dependen parcialmente de la clave principal. Por ejemplo, PrecioExtra depende funcionalmente sólo de Extra y no de IdCoche y de igual forma PrecioDescontado depende funcionalmente sólo de IdCoche y no de Extra.

# Ejercicio de repaso de Normalización

Dada la relación y las DFs

PROYECCION (Título, Formato, Director, Nacionalidad\_Director, Precio, Duración, Puntuación)

DF1 = { Titulo → Director }  
DF2 = { Director → Nacionalidad\_Director }  
DF3 = { Titulo, Formato → Precio }  
DF4 = { Titulo → Duración }  
DF5 = { Título → Puntuación }

1. No está en 2FN porque hay atributos no primos que dependen parcialmente de la clave principal { Titulo, Formato }. Por ejemplo, Director depende funcionalmente sólo de Titulo. No está en 3FN por no estar en 2FN.
2. Propuesta de descomposición en 2FN: separar los atributos que generan conflicto (que dependen parcialmente de la clave principal).

R1 = { Titulo, Formato, Precio }

R2 = { Titulo, Director, Nacionalidad\_Director, Duracion, Puntuacion }

DF1\_1 = { Titulo, Formato → Precio }

DF2\_1 = { Titulo → Director, Duración, Puntuación }

DF2\_2 = { Director, Nacionalidad\_Director }

1. La descomposición anterior es 2FN pero no es 3FN porque ocurre que hay un atributo no primo que puede usarse para realizar una transitividad entre la clave primaria y otro atributo no primo, por ejemplo Titulo → Director y Director → Nacionalidad\_Director en R2

Propuesta de descomposición en 3FN:

R1 = { Titulo, Formato, Precio }

R2 = { Titulo, Director, Duración, Puntuación }

R3 = { Director, Nacionalidad\_Director }

DF1\_1 = { Titulo, Formato → Precio }

DF2\_1 = { Titulo → Director, Duración, Puntuación }

DF3\_1 = { Director → Nacionalidad\_Director }

Si volvemos a unir R = R1 U R2 U R3 y sus dependencias funcionales DF1\_1 U DF2\_1 U DF3\_1 y aplicamos el algoritmo de chequeo de Loseless Join, pasa la prueba.