Trabajo Práctico

FRBA Ofertas

Curso: K3522

Materia: Gestión de Datos

Alumno	Legajo
Ambrosini Nahuel	155 559 5
Álvarez Damián	164 172 4
Martin Daniel	159 186 1

Introducción

Dominio del Problema

El presente TP describe una situación de migración de un sistema antiguo a partir de una base de datos con una tabla única no normalizada.

A partir de la inferencia,deducción, consulta y análisis de las funcionalidades a proveer se diseñó el nuevo modelo de datos representado en un DER. El nuevo sistema será un sistema de ofertas de descuento mediante cupones. El software soporta tres tipos de usuario. A saber, el Usuario proveedor que publica ofertas de descuento, los clientes que pueden recargar saldo y comprar ofertas y el usuario administrador, encargado de facturar a los proveedores.

Un proveedor publica un cupón con cierto descuento sobre un producto. El cliente puede adquirirlo si tiene saldo en su cuenta y de esta forma aprovechar un mejor precio. Por otro lado el usuario administrador será el encargado de emitir las facturas regularmente. Este proceso será manual y el administrador podrá seleccionar el período a facturar y el proveedor para el cual emitir la factura. Dado que es un trabajo práctico en el contexto universitario, el dominio del problema se encuentra acotado. Una aclaración importante es que no tiene testing unitario ya que no fue requisito de aprobación. No obstante se hicieron las debidas pruebas funcionales para comprobar que el sistema contempla correctamente casos de borde emitiendo mensajes de error informativos que sugieren al usuario su buen uso. En un proyecto de software comercial el testing unitario sabemos que es indispensable ya que se está lidiando con dinero y cualquier falta puede ser motivo de sanción y/o pérdida de contrato/s.

Metodología de Trabajo

Para trabajar se adoptó GIT como sistema de versionado y Github como repositorio de código como es estándar en la industria. Se trabajó principalmente mediante whatsapp como vía de comunicación y discusión.

El respositorio se encuentra disponible en https://github.com/dadais216/gdd

No hubo planificación previa y delegación de tareas, lo cual definitivamente perjudicó los tiempos de entrega. Daniel impulsó el desarrollo del proyecto llevando a cabo gran parte de las features. Los problemas que eso ocasionó pudieron ser charlados y resueltos debidamente, permitiéndonos reconocer su esfuerzo y cambiar la actitud para conseguir un mejor resultado. Este detalle podría omitirse, pero resulta relevante ya que los trabajos grupales no evalúan sólo el contenido académico, que muchos ya lo tenemos claro por un empleo, sino la capacidad de interactuar en equipo y generar compromisos con el otro.

En ese sentido, la dinámica final del grupo fue propicia a la colaboración y trabajo en equipo, lo cual sin dudas es a destacar. No obstante sí existió un participante que no se comprometió en ningún momento lamentablemente y por ello no es incluido entre los integrantes de este trabajo práctico.

Tecnologías utilizadas

Como dictaba el TP se utilizó *C#* como lenguaje de programación y *.NET* como framework. El entorno de desarrollo fue mediante Visual Studio 2012 Express y Visual Studio 2019. El motor de base de datos fue SQL Server y para las pruebas se utilizó SQL Server Management Studio como administrador de bases de datos para realizar pruebas y consultas accesorias.

Desarrollo

Inicialización de la base de datos

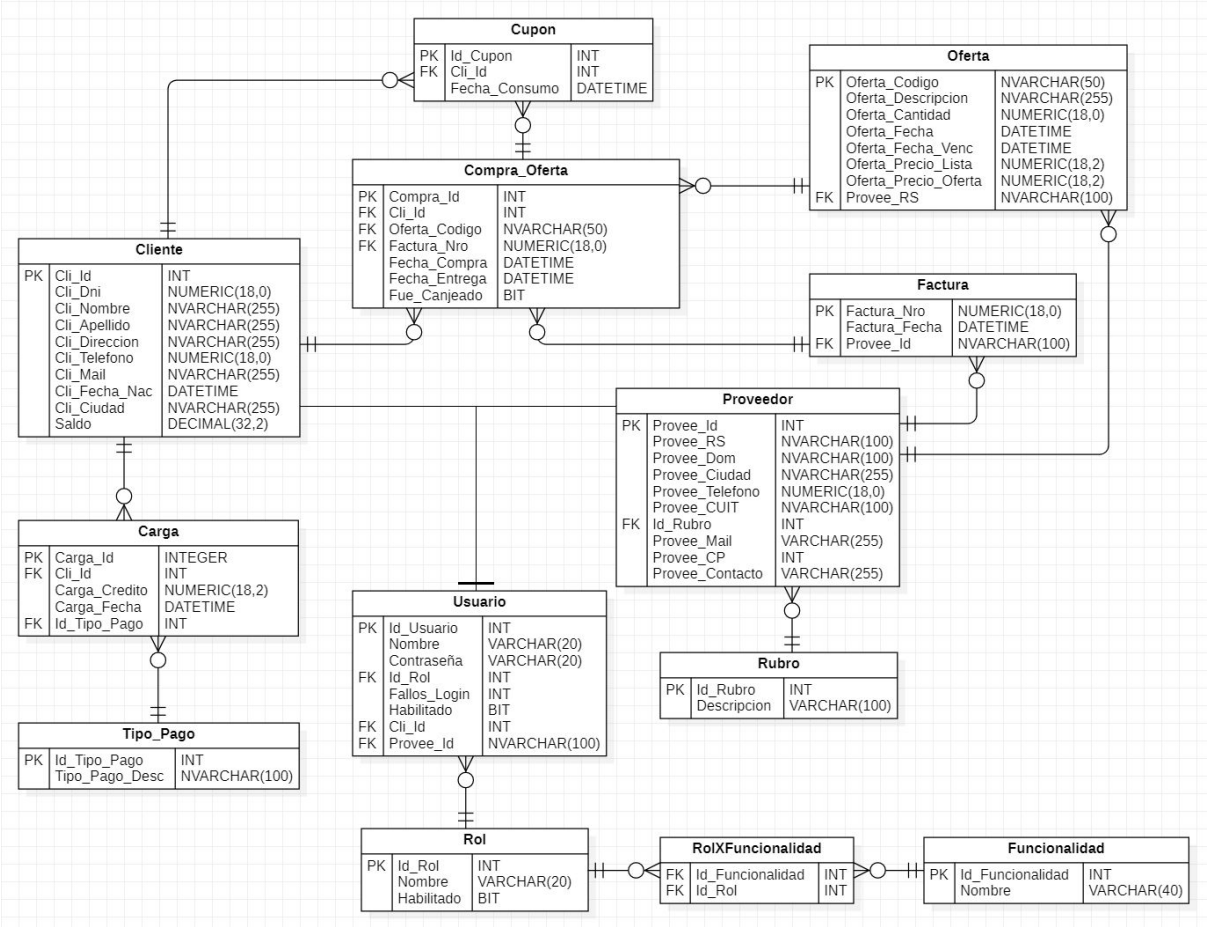
Como fue mencionado en la introducción el lenguaje C#, la base de datos SQL Server y .NET como framework para el desarrollo del TP. El código se encuentra disponible en github bajo la url https://github.com/dadais216/gdd.

Para inicializar la base de datos se entrega un archivo `script_creación_inicial.bat` que corre mediante `sqlcmd` el archivo sql que inicializa la base de datos y deja el resultado de la operación en un log.

Este comportamiento es análogo al que provee la cátedra para cargar la base de datos a normalizar. Si bien no es necesario ni fue solicitado, resulta útil al solo requerir tan solo un doble click.

Lo mismo se hizo para revertir la base al estado previo con `drop_schema.bat.` Los datos de autenticación por defecto son `gdCupon2019` como usuario y `gd2019` como password.

Diagrama Entidad-Relación



En el diagrama se pueden ver tablas maestras y transaccionales. Las maestras serían Usuario, Proveedor, Cliente y Oferta. La factura, las compras, los cupones, y las cargas surgen de la operatoria del producto. Se definieron FK a Tipo de pago y Rubro para permitir la extensibilidad ante nuevos tipos de pago y rubros de proveedores.

También se favoreció la escalabilidad con la tabla RolXFuncionalidad, permitiendo flexibilidad en la asignación de remoción de roles y su asociación a distintas funcionalidades más allá de lo solicitado en el TP.

Fundamentación de decisiones

El primer punto en el cual se tomaron decisiones fue en el modelado de datos a partir de la migración. En la mayoría de las tablas se decidió usar un id porque no había algún campo que cumpla las condiciones de clave primaria. Ya que si bien parecían ser únicos, no estaba garantizado que no pudiera haber otro registro con ese mismo valor y que sea válido.
Por ejemplo en la tabla cliente se podría pensar en usar dni como primary key, pero el dni es único solo dentro de un país y las ciudades que se encuentran en la base vieja indican que la organización opera con clientes de todo el mundo. Lo mismo fue observado con la razón social/ CUIT en proveedor, aunque estos fueron marcados como unique porque lo pedía el enunciado explícitamente.

Para evitar que existan clientes distintos con los mismos datos se hizo que el conjunto de varios de estos este bajo una constraint unique. No se usaron todos los datos porque la constraint crea un índice y este índice tiene un límite de bytes disponibles por entrada, y no resultaba conveniente ingresar todos los campos. Una opción distinta a esto hubiera sido usar un trigger o verificar con queries desde la aplicación antes de insertar campos nuevos, pero esto último nos pareció que dejaba una responsabilidad del lado de la aplicación que podría manejarse fácilmente desde la base. No optamos por triggers porque la constraint era más simple y cumplía la misma función, posiblemente de forma un poco más rápida ya que están optimizados para resolver este tipo de relaciones.

El enunciado parece indicar que se quieren tener campos de mail, código postal y contacto en proveedor. Como estos valores no figuran en la base vieja se optó por tenerlos en null y setearlos luego a través del abm.

En la carga de oferta nos dimos cuenta de que en la base vieja existían muchas ofertas iguales (misma descripción, mismas fechas de publicación y vencimiento),

con códigos distintos. Decidimos dejarlas como estaban, ya que el hecho de que tengan códigos distintos implica que fueron ofertas distintas. Aunque la alternativa de desvalorizar el significado del código y fusionar los registros iguales en uno solo es una opción a considerar, pensando que estos registros duplicados fueron causados por un error de la base vieja.
En la tabla factura usamos como pk el número de factura. Durante la carga se insertan los valores de número de factura de la base vieja, pero durante el funcionamiento su valor va a ser autogenerado. Se evaluó generar un identificador que permite facilitar las búsquedas como fue el modelo de datos durante la cursada, pero dados los datos disponibles parecía un esfuerzo adicional innecesario dado que las funcionalidades requeridas podían ser satisfechas con una performance satisfactorio sin necesidad de ese cambio.

La tabla compra oferta tuvo varias iteraciones. Algo interesante de su llenado en la migración es que usamos funciones agregadas para llenar los campos, aprovechando que las funciones agregadas aplicadas a un conjunto donde solo se tiene un valor y los demás null devuelve el valor, que es justo lo que necesitábamos. Hacer esto ahorra hacer varios joins y simplifica bastante el proceso. Se carga una columna fecha de entrega de esta forma, que luego se usa para generar la tabla cupones y se descarta.

Durante un periodo se mantuvo una relación uno a uno entre la compra oferta y el cupón que nace de esta, pero se terminó prefiriendo no mantenerlo porque no era algo solicitado. Los cupones ahora solo mantienen una relación con el cliente a que pertenecen. La compra oferta tiene un bit que indica si un cupón nació de esta, para evitar que cree cupones duplicados.

Se decidió que la compra oferta mantenga un campo que indique si pertenece a una factura, como parece ser que se hacía en la base vieja. Cuando una compra nueva se crea este campo esta en null, y durante la creación de una factura que la incluya se setea. Hay un chequeo en la facturación que impide facturar una compra oferta que ya pertenezca a una factura, para evitar contar una venta 2 veces por error.

Una idea alternativa que surgió a esto es no mantener esta relación. En este otro modelo para obtener las compra ofertas que pertenecen a una factura habría que buscar las compra ofertas que sean del mismo proveedor y hayan ocurrido entre un rango de fechas, que se guardaría en la factura. Este modelo podría ser mejor porque usa menos memoria, pero si se necesitan saber estas cosas requeriría más procesado, y haría el código para evitar que la facturación facture una misma compra oferta más de una vez un poco más complejo. Además de esto, la base vieja solo guarda la fecha de facturación, no el rango de fechas que fue facturado, por lo que no está totalmente claro cómo se migrarían. Como este modelo complicaba las cosas y no estaba claro si iba a ser mejor optamos por no hacerlo.

A los clientes nuevos se les asignan 200$ de carga desde la aplicación. Los clientes de la base vieja fueron migrados haciendo que arranquen con 0 de saldo, que luego se varió según las cargas y compras registradas. Durante este proceso se descubrió que todas las cargas registradas están asignadas a un mismo cliente, lo que es raro pero al menos por el momento se interpretó como verdadero, aunque probablemente sea un error de la base vieja sobre el cual la empresa deba tomar una decisión. Una consecuencia de esto es que los demás clientes que hicieron compras ahora tienen saldo negativo, algo que está soportado pero no ocurre durante el uso normal del sistema, y sería preferible que no pasara para poder usar números enteros no signados y aprovechar mejor la memoria.

En la carga se guarda el numero de tarjeta en caso de que la carga sea con una tarjeta de crédito o débito. Si es en efectivo, se deja el campo en null. Es una forma simple de guardar la información, tener una tabla aparte para manejar ese campo nos pareció una normalización innecesaria. Se comprobó que un numérico de 16 de precisión (16 decimales) es suficiente para guardar en el número.

Existe una tabla usuarios, que tiene una referencia a un cliente o un proveedor en caso de que sea alguno de estos. Un nombre único, un hash de su contraseña y algunos campos de utilidad.

Durante la migración el nombre de usuario fue la concatenación de los nombres y apellidos de los clientes para los clientes, y la razón social para proveedores. Estos nombres podrían haber sido no únicos en la base de datos vieja, pero da la casualidad de que si son únicos y esto se aprovechó para generar nombres simples y no complejizar de manera innecesaria.

Las contraseñas generadas son números aleatorios, porque es más seguro que tener una contraseña provisional basada en los datos del cliente. Estos números se guardaron en una tabla temporal de forma plana, con intención de ser enviados a los usuarios (de no hacer esto nadie sabría la contraseña de los mismos y las cuentas serían inaccesibles). Esta contraseña tiene intención de ser temporal, la idea es que los usuarios la cambien en su primer acceso. Los administradores son capaces de cambiar las contraseñas si esto se requiere frente a algún problema.
En la base de datos se guarda un hash encriptado mediante el algoritmo SHA-256 como fue requisito de cátedra. Investigando, adquirimos el conocimiento que este algoritmo ya no es vanguardia porque el cálculo del hash es muy rápido, permitiendo armar grandes bases de datos de hashes a partir de inputs conocidos. Con el tiempo, financiamiento y hardware adecuado, es posible vulnerar contraseñas encriptadas mediante esta función. Un algoritmo alternativo que artificialmente demora en la generación de hashes para mitigar el riesgo de brute-forcing es bcrypt. En un entorno corporativo donde se manejan datos de clientes y hay entidades regulatorias, este algoritmo podría ser utilizado en lugar de SHA-256.
Cuando el usuario ingresa la contraseña, esta se hashea y el resultado se compara contra el valor de la base de datos. Sólo es válido el login si ambos hashes coinciden. Vale mencionar que ninguna función de hash evita colisiones, pero estos algoritmos están probados y aceptados en la industria para minimizar la probabilidad hasta poder considerarla despreciable.

El enunciado hace referencia a que en el futuro el usuario va a tener varios roles. Como la utilidad de esto no estaba clara para nosotros, ni su implementación (podría implementarse de formas distintas a asignar varios roles directamente),

preferimos optar por no mantener una relación de muchos a muchos entre usuarios y rol de forma preventiva. Iba a ser un trabajo que no estaba claro si iba a dar frutos. Si en una versión futura se requiere que haya muchos roles, además de los cambios de interfaz y los chequeos sobre rol, podría cambiarse esa relación a muchos a muchos sin demasiados problemas, ya que no hay muchas cosas de código que choquen contra eso (solo la interfaz y los chequeos de rol, que están entre las cosas a cambiar para implementar la nueva lógica de todas formas).
Algo que termina resultado raro de esta forma de plantear las cosas es que cuando se elimina un rol los usuarios de ese rol quedan con ese campo vacío. Esto no es un problema en sí, ya que hay pocos lugares del código que miran ese campo y saben de manejar el nulo, es solo una consecuencia de haber planteado las cosas así. Si manejar el null específicamente es un problema se podría crear un rol que cumpla función de null object, o se podría determinar un rol default a asignar cuando se elimina un rol.

Después de desarrollar las funcionalidades se nos ocurrió que estas podrían haberse implementado como un campo de bits en rol en lugar del modelo totalmente normalizado que hicimos. Usar un int, por ejemplo, permitiría hasta 32 funcionalidades distintas. Hacer esto haría el código que trabaja con funcionalidades más rápido, más simple y fácil de usar. No nos tiramos por hacerlo porque no creemos que el trabajo apunta a hacer desnaturalizaciones fuertes de este tipo.

La pantalla principal de la aplicación muestra un botón por cada funcionalidad del rol del usuario que haya ingresado, que da acceso a esta funcionalidad. Algunas funcionalidades tienen chequeos internos sobre rol de todas formas, por ejemplo un administrador con acceso a todas las funcionalidades no tiene saldo y por lo tanto no puede comprar, el codigo para crear una oferta es distinto para proveedores y administradores.

Al inicio de la aplicación se inicia una conexión con la base de datos, que se mantiene durante todo el uso. Buscando en internet vimos que hay gente que recomienda no hacer esto, y abrir una conexión única para cada transacción. Como

hacer esto es más trabajo (para el programador y el programa) y los beneficios de hacerlo no son claros no lo hicimos. No tuvimos problemas con manejarnos con una conexión.

Para facilitar el desarrollo existe código que levanta la aplicación con un usuario ya logueado con el rol que se necesite. Este código no se compila en la versión que se entrega al usuario.

La cantidad de fallos del login se guarda en la base de datos, para mantener una consistencia si el usuario intenta hacer logins en conexiones distintas.
Si se supera la cantidad de fallos o el administrador lo determina, el usuario queda inhabilitado y no puede acceder al sistema.

El form de crear cliente y crear proveedor se reutilizan en sus abms y en sus registros. Esto los hace un poco menos flexibles porque se tienen que adaptar a ambos casos de uso pero ahorra código.

En los abm de cliente y proveedor se construye un único query según los filtros colocados.

Como no teníamos indicativo de cómo generar el código de oferta, decidimos generarlo como un valor alfanumérico aleatorio. En caso de colisión se intenta con otro valor. Se podría haber hecho algo secuencial que arranque desde el máximo de los insertados de la migración, pero como es algo que no está soportado nativamente en la base y implementarlo implicaría crear una nueva tabla donde guardar el valor de secuencia (no puede guardarse en la aplicación para evitar condiciones de carrera entre varias de ellas), lo que complicaría la solución, decidimos no hacerlo, no parece ser a lo que apunte el trabajo.

En varias operaciones importantes se usó un form extra para confirmar la acción.

Los campos ingresados por el usuario son validados en caso de ser validables, por ejemplo se verifica que los numeros son numeros.

Se usaron transacciones en las operaciones que involucran varias operaciones consecutivas para garantizar la integridad de los datos.

Durante el desarrollo no se hizo mucho uso de stored procedures y functions porque se percibió que el desarrollo era más rápido y el código más legible teniendo las queries usadas por el código en el código. Esto viene al costo de tener que enviar el string de la aplicación a la base cada vez que se usa, pero es un costo que no pareció ser relevante al menos por el momento. Mover los queries más grandes a stored procedures es un cambio que no llevaría mucho tiempo hacer si se cree necesario en un futuro.

En varias partes del código se hace uso de @@Identity para acceder al id de una tabla que acaba de ser insertada, así ahorrando tener que hacer un query para obtenerla. Usar esta funcionalidad es seguro, ya que el valor se mantiene consistente dentro de una misma conexión. Podrían generarse problemas si una tabla tiene un trigger que haga un insert, ya que actualizaría el @@Identity y puede que quien lo use no se percate. Pero como por el momento no tenemos triggers que hagan inserts este no es un problema, aunque es algo que hay que tener documentado por si en el futuro se usan.

Varias queries del código se hacen concatenando strings, porque resultaba más cómodo. Se tuvo cuidado de no hacer esto con datos de input de usuario, aún con usuarios administrativos. Estos datos son validados correctamente usando las librerías de sql de c# para evitar posibles inyecciones.

Se hizo una Stored Procedure para el proceso de facturación para que no se generen facturas sin compras asociadas o solo impacte en algunas compras. En un principio esto no era así y si bien se manejaba con lógica de manejo de excepciones en la aplicación, es una buena muestra de cuando usar stored procedures nos dan

un respaldo de que si las filas de una tabla son modificadas, lo harán de forma consistente. No obstante son decisiones de desarrollo que no son mandamientos. Dependiendo del conocimiento del equipo, el lenguaje de programación, el motor de base de datos y el problema a resolver, convendrá hacer una stored procedure o invocar una función en la aplicación y manejar las transacciones y eventuales excepciones a nivel aplicación y no motor de base de datos. En este tp lo hicimos para mostrar su ventaja en la simplificación del código de la aplicación.

Conclusiones

Mediante el trabajo práctico se pudo aprender los desafíos que presenta la migración de datos no normalizados. A la hora de encarar un proyecto a nivel profesional resulta de utilidad conocer los problemas que se pueden enfrentar y posibles formas de resolverlos. Estar familiarizado con los GROUP BY, los DISTINCT y las funciones de agregación es muy bueno a la hora de asegurarse no perderse ningún dato y a la vez no duplicarlos.

Por otro lado, a nivel personal para los integrantes del grupo que tienen experiencia laboral fue de gran utilidad trabajar cerca de la base de datos. En particular a Nahuel le sirvió para cambiar de opinión en que no siempre es mejor abstraer todo en un ORM e interactuar siempre con objetos del lenguaje. Si bien eso trae una facilidad en el desarrollo porque se puede pensar en objetos, es de gran utilidad e incluso necesario conocer las estructuras internas de la base de datos para ser consciente cuando una decisión que parece trivial puede penalizar la performance de la base de datos.

Adicionalmente, se pudo contrastar los conocimientos de la cátedra en los puestos laborales. El nuevo paradigma para APIs Graphql se apoya fuertemente en la traducción eficiente de querys escritas en SDL (Schema Definition Language) a SQL performante. En particular se observó como la acumulación de subselects penalizaba fuertemente la performance y como los joins facilitaban la tarea de ejecutar las querys. Contenido que fue visto en la materia.

También se aprendió de las ventajas de utilizar un framework de diseño de interfaces gráficas del tipo WYSIWYG (What You See Is What You Get) ya que acelera el tiempo de entrega de interfaces y permite poner el esfuerzo en la implementación en código.

Por otro lado en cuestiones organizativas, se confirmó la hipótesis que el trabajo grupal colaborativo sigue siendo un desafío. Principalmente para generar compromiso entre los integrantes y un clima de colaboración para un objetivo común.

Es de gran utilidad para la labor de ingeniero tener este tipo de experiencias, aunque no sean necesariamente placenteras. Si en algún momento alguno decidiera ir por una carrera profesional de gestión de proyectos, ya está entrenado para detectar alertas y posibles riesgos ante la falta de compromiso