WMS Tandil

Trabajo Práctico Especial 2019

Base De Datos I

Grupo *3*

Integrantes:

248570 - Horacio, Casado

249039 - Thomas, Ojeda

248557 - Matias, Morandeira

Indice

Consideraciones generales

Para el siguiente trabajo se consideró que el resto de restricciones especificadas en el enunciado están en funcionamiento, con el fin de enfocarnos en realizar solo los servicios y controles requeridos, asumimos que ante cualquier acción sobre el esquema, los datos involucrados serán consistentes todo el tiempo.

A. Ajuste del Esquema

Se ajustó el esquema según lo descrito en el enunciado, modificando los nombres de las tablas con el prefijo **G06.** Para la inserción de datos sobre las tablas **g06_movimiento_cc** y **g06_linea_alquiler** agregamos las secuencias **sq_g06_movimiento_cc_id_movimiento** y **sq_g06_linea_alquiler_id_liquidacion**, para poder realizar las inserciones asegurando siempre un id distinto para cada inserción.

B1. Elaboración de Restricciones

B1.2) Un movimiento de salida debe referenciar, en orden cronológico, al último movimiento interno, si los tuviera, o al movimiento de entrada (respecto del mismo pallet).

Analisis

Se realizó un análisis de las tablas involucradas y los campos de dichas tablas que pueden activar la restricción:

TABLA	INSERT	UPDATE	DELETE
movimiento	✔	fecha ✔	❌
		cod_pallet ✔	
		fecha ✔	

Para esta restricción nos evocamos a controlar sólo las operaciones de los movimientos de salida (**tipo = 'S**') en la tabla **movimiento**, ya que ninguna acción sobre otra tabla puede cambiar la relación de los movimientos de salida con los demás movimientos. Por lo tanto podemos concluir que esta restricción es de tabla.

Las situaciones que se controlaron tanto en el **UPDATE** como en el **INSERT** fueron las siguientes:

- La **fecha** del **id_mov_ant** (movimiento anterior) debe ser previa a la del movimiento a insertar o a actualizar independientemente si es el más reciente o no.
- El **cod_pallet** (pallet que se está movimiento) del **id_mov_ant** debe ser el mismo que el movimiento de salida.
- Basándonos en lo anterior, no se puede cambiar el **cod_pallet** sin cambiar en conjunto con el **id_mov_ant** ya que el movimiento anterior no sería del mismo pallet que el movimiento de salida.

- No puede haber más de un movimiento de salida por pallet.

En el caso del **DELETE** no es necesario realizar ningún control porque, en caso de querer borrarse un movimiento de salida válido, dicha acción no producirá ningún cambio que pueda activar el control de la restricción.

Implementación

1) Restricción en *SQL estándar declarativo*
Constraint: **ck_g06_mov_anterior_no_consistente**

Esta se codificó bajo el siguiente razonamiento:
Chequear que **no exista** un movimiento de salida cuyo movimiento anterior sea diferente al movimiento de entrada o al interno más reciente del pallet, o que la fecha del movimiento de salida sea previa a la fecha definida en el movimiento anterior. Además chequear que **no exista** más de un movimiento de salida por pallet.

2) Restricción en PostgreSQL
Función:

- **trfn_g06_mov_anterior_no_consistente()**

Triggers:

- **tr_g06_mov_anterior_no_consistente_insert**
- **tr_g06_mov_anterior_no_consistente_update**

Definimos una función que realiza el control basándose en el mismo razonamiento comentado antes, con la única diferencia de que se sacó provecho de las posibilidades que este recurso ofrece: solo realizamos los controles a partir de la tupla que se modifica o se inserta a partir de (NEW y OLD), esto nos da la ventaja de evitar hacer un control sobre todos los datos de tabla (caso declarativo) y realizar un control más eficiente. Como consecuencia, una vez creado el trigger que utiliza la función sobre la tabla movimiento, dicho trigger no verifica la consistencia de los datos ya cargados (caso declarativo).

3) Sentencias de modificación e inserción que promueven la activación de la restricción:
Dentro del script soluciones, inmediatamente después de la implementación de las restricciones se ubican todos los ejemplos de prueba, si tomamos el primer insert de prueba donde el **id_mov_ant = 3** y lo ejecutamos, se activa el trigger y dentro de la función, luego de calcularse el movimiento más reciente del pallet cuyo id es 4, se realiza una comparación entre ambos valores (4 <> 3) y como son diferentes levanta la excepción: *El movimiento anterior no es el más reciente del pallet.*

B2. Reglas del Negocio

B2.1) Mantener actualizado automáticamente el saldo de cada cliente.

El cumplimiento de esta regla del negocio implica un accionar bajo los siguientes eventos críticos:

TABLA	INSERT	UPDATE	DELETE
movimiento_cc	✔	cuit_cuil✔	✔
		importe✔	

El trigger y función que implementan esta regla en el script G06_Soluciones.sql son **tr_g06_actualizar_saldo** y **trfn_g06_actualizar_saldo**.

Las inserciones y las eliminaciones implican claramente un incremento o decremento del **saldo** de un cliente. Si una tupla en **g06_movimiento_cc** se corresponde a un movimiento de credito, entonces la modificación del **cuit_cuil** de dicho movimiento implica una modificación tanto en el **saldo** del cliente viejo como en el del actual. Lo mismo ocurre con movimientos de débito, pero la modificación de **cuit_cuil** en estos llevará a la base de datos a un estado inconsistente (los movimientos de débito están relacionados a los alquileres de un cliente ya establecido). La modificación del importe también implica una actualización en el **saldo** del dueño de la cuenta.

En el script G06_Soluciones.sql se especifica un ejemplo en el cual se inserta una fila cuyo importe es 100$ y luego se modifica. Para actualizar el saldo del cliente, se busca su fila en la tabla **g06_cliente** y se le suma el nuevo importe. Para las eliminaciones se resta, y para cualquier actualización se resta el viejo **importe** al **saldo** antiguo cliente, y se suma el nuevo **importe** al **saldo** del nuevo cliente (si el cliente no cambia, el nuevo y viejo cliente son el mismo). El buen manejo de actualizaciones de **importe** es crucial ya que el servicio **C3** los realiza frecuentemente.

C. Servicios

C1.1) Generar una lista de las estanterías que en este momento tienen más de cierto porcentaje (configurable) de las posiciones ocupadas.

Al tratarse de un servicio que tiene un parámetro (el porcentaje configurable) decidimos utilizar una función **fn_g06_estanterias_ocupadas_mas_de(_porcentaje real)** que devuelva una tabla con los datos solicitados y que reciba como parámetro el porcentaje, ya que con una vista no sería posible realizar una consulta parametrizada. Básicamente para cada estantería se calcula la cantidad de posiciones que tiene y se compara ese valor con la cantidad que están ocupadas para determinar si el porcentaje es mayor que el dado por parámetro.

Para este servicio se consideró que las posiciones solo tienen tres estados posibles, los mencionados en el trabajo. Quedando que el estado de cualquier posición tiene que ser uno de los siguientes valores $\{LIBRE, RESERVADO, OCUPADO\}$.

C2.2) Devolver todos los datos de las posiciones (estantería, fila y nro de posición) que cambiaron de zona al reconfigurar el depósito, indicando si están ocupadas o no actualmente.

Dada la simpleza del servicio decidimos implementarlo con una vista **g06_ultimos_cambios_de_zona** que selecciona los datos completos de todas posiciones en **g06_posicion** que se encuentran en un conjunto de posiciones en **g06_zona_pocicion** que poseen la fecha de alta más reciente (la misma para todas) y que no aparecen solo una vez en

dicha tabla, es decir, no es la primer asignación de zona que tiene la posición. Esta vista es actualizable tanto en el estándar de SQL como en PostgreSQL ya que posee la clave de la tabla **g06_posicion**, no utiliza ninguna función de agregación, ninguna subconsulta en el select, tampoco distinct y no requiere de ningún join.

Consideramos que podían existir dos formas de actualización de zonas, una que involucre a todas las posiciones de las cuales algunas podrían permanecer con la misma zona y otras no, y otra que cada actualización solo involucra a posiciones que realmente cambiaron de zona. Nuestra implementación se basó en esta última.

C3) Diariamente se debe actualizar la Cuenta Corriente de cada cliente con los alquileres que tiene activos, agregando un movimiento de débito (importe negativo) en la cuenta corriente del mismo.

Este servicio se implementó con un stored procedure **pr_g06_generar_facturas()** ya que se debe encargar de generar nueva información en la base diariamente. Más específicamente genera nuevos movimientos de cuenta corriente, es decir, tuplas en **g06_movimiento_cc** y en **g06_linea_alquiler**.

Para realizar esto, primero se obtienen todos los clientes con alquileres activos en **g06_alquiler**, considerando un alquiler activo como aquel cuya **fecha_hasta** es nula y su fecha de inicio es menor o igual a la fecha actual. Con cada uno de estos clientes se genera una nueva tupla en **g06_movimiento_cc**. También se consulta en la tabla **alquiler** por los alquileres pertenecientes cada cliente. Con ellos, se consulta la tabla **g06_alquiler_posicion** por las posiciones de dichos alquileres, y por cada una de ellas se genera una nueva tupla en **g06_linea_alquiler**, referenciando a la tupla **g06_movimiento_cc** correspondiente al cliente.

Esta implementación no requiere el uso de tablas auxiliares y sigue un esquema muy similar a una consulta de múltiples tablas ensambladas con join. Además las proyecciones sobre las tablas ayudan a que no se maneje un volumen de datos excesivo.

En el script G06_Soluciones.sql se incluyó un ejemplo detallado al final del sector de este servicio, en el cual a partir de clientes y alquileres añadidos se ejecuta el procedimiento, que genera las nuevas filas correspondientes a una facturación diaria para cada cliente.

D. Vistas

D1.1) Listar los datos de todos los clientes junto con el último movimiento de pago de mayor importe que cada uno de éstos realizó en los últimos 12 meses, en el caso que corresponda.

La vista planteada en esta ocasión **g06_clientes_ultimos_pagos_relevantes** selecciona todos los datos de **g06_cliente** junto con todos los datos del último movimiento de cuenta corriente **(g06_movimiento_cc)** de pago de mayor importe en el último año. En PostgreSQL esta vista no es automáticamente actualizable, debido a que se utiliza un join. Mientras tanto en el estándar de SQL resulta automáticamente actualizable, pero solo los datos de la tabla **g06_movimiento_cc**, esto se debe a que es la tabla de la cual se preserva la clave. Se decidió mostrar todos los datos de **g06_movimiento_cc** ya que si no se seleccionan todos, a la hora de hacer el trigger instead

of no tendríamos los datos que se quieren ingresar y los tendríamos que “inventar” o elegir algunos valores por defectos para poder insertar los datos en estas situaciones.

Para implementar el trigger instead of decidimos que las inserciones se realizarán como si este tuviera un local check option, ya que consideramos que si se quiere insertar algo en la vista, esto debería ser mostrado luego de la inserción en esta. En el caso de que se quiera insertar datos que no van a ser mostrados en la vista debido a sus condiciones tiene más sentido que sean ingresados a través de la tablas correspondientes. Otra aclaración relevante es que siempre se van a insertar nuevos movimientos de cuenta corriente, nunca se va a actualizar un movimiento existente. Esto lo decidimos así dado que estos son datos que vienen a representar comprobantes y que la modificación de estos podría incurrir en situaciones de inconsistencias entre los comprobantes reales y los cargados en el sistema. Para evitar inconsistencias cuando se quiere un insertar un registro a la vista, y el cliente ya existía la decisión que tomamos es no modificar los datos del cliente, lo único que sé hace es insertar, si se cumplen las condiciones mencionadas anteriormente, el movimiento de cuenta corriente.

D2.1) Listar todos los datos de las posiciones de la fila número 5 en adelante que nunca han sido alquiladas

La vista **g06_posiciones_nunca_alquiladas_fila_5_en_adelante** planteada en esta ocasión selecciona todos los datos de **posicion** donde el **nro_fila** es mayor o igual a cinco y además no existe ningún registro de alquiler para cada una de estas en **alquiler_posicion**. Haciendo el mismo análisis que para la vista del servicio C2.2 también podemos afirmar que esta vista es actualizable tanto en el estándar como en PostgreSQL.

Dado que esta vista está hecha sólo en base a una tabla, es equivalente un **LOCAL CHECK OPTION** que un **CASCADED CHECK OPTION**. Si consideramos el segundo ejemplo del script seguido de la implementación de la vista, con esta definida sin ningún check, se ejecuta el insert de manera exitosa, pero si mostramos los datos de la vista, no la encontraremos entre estos, ya que dicha tupla posee un **nro_fila = 4**. Pero sí la vista hubiera contado con check option, esta misma inserción no se hubiera realizado debido a que no cumple la condición de la vista.

Conclusión

Se pudo realizar de forma exitosa todas las restricciones, reglas de negocio, servicios y vistas solicitadas. Los resultados de las pruebas realizadas respaldan el correcto funcionamiento de cada implementación, ya que se obtienen resultados esperados, salvo en ocasiones en las cuales los datos ingresados no son consistentes debido a un mal uso. Suponiendo que los usuarios no son capaces de realizar tales modificaciones, la base de datos implementada en el presente trabajo sirve como un modelo a escala de un sistema mayor y más complejo, pero con características fundamentales muy parecidas. Con ella es fácil comprender la dificultad que presenta mantener y administrar este sistema.