**Informe primera parte segundo trabajo de Bases de Datos II – 2021-1**

***Integrantes:***

*Aristizabal Giraldo Salomé - Rendón Giraldo Sebastián - Valencia Zapata Santiago Alexis*

**Primer punto: Blockchain empresa**

La solución de este punto se compone de los archivos comandosMultichain.txt, cada una de las líneas de código de este archivo deben correrse por separado en la consola en el orden en que aparecen en el archivo, adicionalmente cabe destacar que la línea de código multichaind empresa -daemon debe correrse en una consola por separado, esta consola debe permanecer en ejecución para poder ejecutar el código correctamente.

En este archivo se crea la blockchain llamada “empresa”, posteriormente se le crean 2 streams “vendedores” y “ganancias”.

Para el Smart filter del stream ganancia se verifica inicialmente que la clave cedula sea única, si la longitud de la clave es diferente de 1 retorna un error, posteriormente se verifica que exista la clave, para esto se asegura que la longitud de la clave sea mayor a 1 y que no sea igual a null, seguidamente se verifica que el formato sea json usando .format, luego se verifica que los valores del stream ganancia sean únicos, que su longitud sea mayor a 1 y que no sean nulos ya que son datos obligatorios, finalmente si se ingresa un dato diferente a numero toma el valor de NaN y genera un error.

Para el Smart filter del stream ganancia se verifica inicialmente que la clave cedula sea única, adicionalmente que su longitud sea mayor a 0 y que sea diferente de null ya que este campo es obligatorio, posteriormente se verifica que el formato de los datos ingresados sea Json, este es el único formato que se recibe. Una vez se asegure de que el formato es correcto se toman los demás datos del stream ganancia, la cantidad de datos ingresados sin contar la cedula debe ser igual a 2, ya que solo quedan los campos de nombre y teléfono, posteriormente se verifica que el nombre no sea null y su longitud sea mayor a 1 ya que este campo es obligatorio, y se hace lo mismo con teléfono.

Se obtienen las direcciones, y se procede a activar los Smart filter, cabe destacar que para que estos funcionen debe reemplazar en ambos comandos [dirección] por la dirección que obtuvo en el paso anterior, para que pueda ejecutar estos comandos correctamente, el resto del código son datos de prueba, no es necesario ejecutarlos.

La solución de este punto se compone por 3 archivos ‘.sql’ guardados en la carpeta Punto 1, los cuales se deben ejecutar en el siguiente orden para la correcta implementación de la solución: TABLA\_AUXILIAR, GRANJA\_CERDOS\_PACKAGE, PEDIDO.

El archivo TABLA\_AUXILIAR implementa un bloque PL/SQL con un EXECUTE INMEDIATE para crear una tabla llamada CERDOXCAMION en la cual, más adelante se irán guardando los cerdos que ya hayan sido utilizados para cubrir el pedido y no volverlos a usar al realizar las cuentas, esta contiene la referencia de un cerdo y la del respectivo camión al que fue asignado.

El archivo GRANJA\_CERDOS\_PACKAGE encontramos la definición de tipos de tipos de arrays como fila\_camion, fila\_cerdo y matriz\_booleana, que son usadas al momento de generar el informe, tanto dentro del paquete, como en el archivo PEDIDO. También se construyen las funciones: matriz\_falso\_verdadero, pseo\_cerdos y elegir\_cerdos, las cuales son llamadas por el archivo PEDIDO, y cuya funcionalidad será explicada cuándo entremos en contexto a la solución.

El archivo PEDIDO es un PROCEDURE nombrado pedidos el cual recibe un input que corresponde al peso del pedido que se desea realizar, este tipo de dato es el mismo utilizado en la tabla CERDO.pesokilos, el cual debe ser un entero positivo. Para ejecutarlo en consola se debe utilizar un EXECUTE pedidos(kilos); en el cual los kilos se reemplazan por el valor de kilos deseado por el cliente.

Luego de esta línea, encontramos el IS donde declaramos todas las variables que se irán utilizando más adelante, y luego comienza la ejecución del código. Primero se asigna a la variable ‘pedido’ el valor ingresado al llamar al EXECUTE pedidos(); y luego borramos de la tabla CERDOXCAMION cualquier eventual dato residual de alguna consulta anterior. Posteriormente realizamos un BULK COLLECT de los camiones disponibles ordenados de mayor a menor capacidad. En caso de que no haya camiones disponibles, se salta la excepción ‘no\_se\_puede’, creada en la sección DECLARE, la cual nos lleva a un bloque de excepción dónde esta se resuelve imprimiendo el mensaje de ‘El pedido no se puede satisfacer’.

En caso contrario, usamos un FOR para recorrer la lista de camiones disponibles. En cada iteración calculamos el peso que aún falta por enviar, restando el peso ya enviado al pedido total, luego calculamos el peso máximo que podremos llevar en el camión de la iteración actual. Este peso máximo se calcula como el mínimo entre el peso que aún hace falta llevar, y la máxima capacidad del camión de la iteración actual. Luego se realiza otro BULK COLLECT para buscar los posibles cerdos que, según el peso máximo calculado, el camión está en capacidad de llevar, exceptuando a los que se encuentren en la tabla CERDOXCAMION.

Si la lista de cerdos elegibles queda vacía luego del BULK COLLECT, puede haber dos posibles escenarios, el primero es que nos encontremos en el primer camión de la iteración. Esto significa que el camión de mayor capacidad no está en condiciones de llevar ningún cerdo de la tabla CERDO, o en su defecto, que esta está vacía. Ambas situaciones implican que el pedido no puede ser satisfecho, por lo que de nuevo se salta la excepción ‘no\_se\_puede’ y se interrumpe la ejecución. El segundo escenario es que no nos encontremos con el primer camión, esto quiere decir que hay parte del pedido que pudo ser cubierto con al menos otro camión. En este caso sencillamente se interrumpen las iteraciones con un EXIT, puesto que ninguno de los camiones siguientes podrá contribuir con el pedido, pero se continua con la ejecución porque los anteriores camiones generaron información para satisfacer el pedido.

Por otra parte, si la lista de cerdos no queda vacía, debemos seleccionar un subconjunto de cerdos que, entre todos, llenen el camión lo más que puedan hasta el límite del peso máximo. A este problema se le conoce como el ‘Subset Sum Problem’. Para solucionarlo, se debe generar una matriz de falsos y verdaderos donde cada fila representa, para nosotros, un cerdito con su respectivo peso, y cada columna, un posible valor de suma que se puede conseguir con la suma de los pesos de los cerditos, también les agregamos la fila 0, que representa a ningún cerdito, y la columna 0 que representa la suma de 0. Generamos columnas de 0 hasta el peso máximo. Luego llenamos toda la columna de 0 con Verdaderos, y toda la fila de 0 con Falsos, a excepción de la posición (0,0) que queda verdadera. Para el resto de posiciones de la matriz, su valor se definirá de la siguiente: El valor de la posición (x,y) con ‘x’ como fila, y ‘y’ como columna, será verdadera si (x-1,y) también es verdadera, o si existe una posición (x-1,y-k) dónde ‘k’ es el peso del cerdito que representa la fila actual. Si esta existe y es verdadera, el valor de la posición (x,y) también será verdadera, si no se cumple alguna de las dos condiciones, el valor de (x,y) será falso. Esta matriz se genera con la función ‘matriz\_falso\_verdadero’ que recibe como argumento al array de los cerditos seleccionados y al peso máximo.

Luego de generar la matriz, con esta podemos calcular el peso máximo que podremos llevar con los cerditos. Para esto, la función ‘peso\_cerdos’ usa la matriz generada anteriormente, el listado de cerdos y el peso máximo que se puede llevar en esta iteración, para recorrer la matriz en su última fila, desde su última columna hacia la primera. En la columna que se encuentre un Verdadero, se sabrá que el peso que esa columna representa será el peso que se va a llevar en esta iteración, y este es el valor que la función devuelve.

Finalmente, usamos la función ‘elegir\_cerdos’, la cual hace uso del peso calculado en ‘peso\_cerdos’, la matriz de falsos y verdaderos y la lista de cerdos, para escoger a los cerdos que van a ser llevados en el camión de la iteración actual. Para esto, empezamos a recorrer la matriz desde la última fila de la columna que representa al peso ingresado a la función. Si nos encontramos en la posición (x,y), verificamos el valor de la posición (x-1,y), si este es verdadero, saltamos a la posición (x-1,y) y repetimos el proceso; si este es falso, agregamos a una lista de cerdos elegidos al cerdo que representa la fila ‘x’ y saltamos a la posición (x-1,y-k) dónde ‘k’ es el peso del cerdito que representa la fila ‘x’, y repetimos el proceso hasta que lleguemos a la columna 0, dónde nos detenemos. Esta función nos retorna la lista de cerdos que vamos a despachar en el camión actual.

Una vez sabemos los cerdos que podemos despachar, procedemos a imprimir el informe para el camión. Además, agregamos a los cerditos elegidos a la tabla CERDOXCAMION, para evitar que sean tenidos en cuenta en el futuro. Por último, actualizamos los valores del peso ya enviado y vaciamos otras variables que se requieren vacías para la siguiente iteración

Al finalizar las iteraciones que se puedan realizar, imprimimos el final del informe que condensa la cantidad del pedido total, lo que se cumplió y lo que faltó por cumplir. Y así finaliza la ejecución del programa.

La inspiración para la solución de este problema surge del siguiente video en el minuto 8:54 <https://youtu.be/34l1kTIQCIA?t=534>

**Segundo punto: Triggers para la tabla Individuo**

La solución de este punto se dividió en varios archivos ‘.sql’, ubicados en la carpeta Punto 2, dónde cada uno se llama (nombre\_del\_trigger)\_trigger.sql.

**Punto a:** Para este punto se creó un trigger llamado ‘ingreso\_cero\_hijos’ del tipo ‘BEFORE INSERT FOR EACH ROW’ para la tabla Individuo, que solo se dispara cuándo el nuevo atributo de número de hijos es diferente de 0. Este trigger lanza un error con la función RAISE\_APLICATION\_ERROR, la cual le indica al usuario que solo se pueden crear individuos con 0 hijos.

**Punto b:** Para este punto se creó un trigger llamado ‘ingreso\_con\_padre’ del tipo ‘BEFORE INSERT FOR EACH ROW’ para la tabla Individuo, que solo se dispara cuándo el nuevo atributo de padre es diferente de NULL, y se ejecuta después de ‘ingreso\_cero\_hijos’. Este trigger ejecuta un UPDATE para la tabla Individuo y le suma 1 al número de hijos del individuo cuyo código coincida con el código del padre del individuo que está ingresando.

**Punto c y d:** Para este punto se creó un COMPOUND TRIGGER llamado ‘borrado\_con\_padres\_o\_hijos’, del tipo FOR DELETE para la tabla individuo, que nos actualiza los datos referentes a padres o hijos en caso de que el individuo los tenga. Para esto creó un BEFORE STATEMENT dónde se creó un array con los datos originales de los individuos, y posteriormente se hace un UPDATE en la tabla individuo, para asignarle como NULL a todos los valores de la columna PADRE, esto para evitar problemas con la referencia que maneja esta columna con respecto a la columna CODIGO al momento de borrar un individuo.

Luego creamos un BEFORE EACH ROW, dónde obtenemos y guardamos los datos OLD del individuo que vamos a borrar. Dado que el valor de PADRE se cambió a NULL en el anterior bloque, procedemos a buscar el valor real de este campo en la lista generada antes del cambio, también en el anterior bloque.

Después de obtener todos los datos requeridos, generamos un AFTER STATEMENT dónde si el individuo borrado tenía padre, a este le realizamos un UPDATE en la columna ‘nro\_hijos’ restándole 1 al valor que ya tenía. Finalmente, se le asignan los valores originales de la columna PADRE que quedaron almacenados en el array del primer bloque, excepto a los hijos del individuo que se acabó de borrar, por lo que estos quedan en NULL.

**Punto e:** La solución de este punto se hizo con un trigger llamado ‘actualizar\_valor\_individuo’, el cual es un COMPOUND TRIGGER del tipo FOR UPDATE que solo funciona para la columna VALOR en la tabla INDIVIDUO. Este es ejecutado después del trigger ‘actualizar\_codigo\_individuo’ (Solución punto f) para evitar inconsistencias en caso de que se activen los dos al mismo tiempo. En este trigger generamos un bloque BEFORE EACH ROW dónde calculamos la diferencia entre el viajo valor y el valor a actualizar. En caso de que esta diferencia sea mayor a 0, verificamos si es mayor a 5. Si esto no se cumple, lanzamos una excepción con RAISE\_APLICATION\_ERROR en la cual se indica que Solo se permiten aumentos de 5 unidades o más en el valor del individuo. Con esta excepción, el programa terminaría su ejecución. Si por el contrario, la diferencia es mayo o igual a 5, actualizamos el NEW.VALOR = OLD.VALOR+2 y calculamos lo que se le va a adicionar al valor de un posible hijo, que sería la diferencia menos 2, así mismo, guardamos el nuevo código del individuo y el número de hijos que tiene, para poder usar esta información posteriormente. Si ninguno de estos escenarios ocurre, el UPDATE se realiza tal cual como se estableció desde el principio.

Luego generamos un AFTER STATEMENT donde verificamos si la diferencia es mayor o igual a 5 y si el número de hijos es mayor a 0, pues en caso contrario, no se debe hacer nada más con el UPDATE. En caso de que se cumplan estas dos condiciones, generamos una lista de los hijos del individuo que se está actualizando y seleccionamos el que tenga menor valor. Si este hijo elegido tiene hijos, los guardamos en otra lista para asegurarnos de no perder la referencia, pues luego de esto procederemos a hacerle un DELETE, y enseguida se le hace un INSERT con los mismos valores, a excepción del VALOR, dónde se le coloca el valor adicionándole lo que su padre le ha dejado, y dejamos el número de hijos en 0 para no generar problemas con el trigger del punto a. En caso de que este hijo elegido tuviese hijos, le hacemos un UPDATE para colocarle el número de hijos que tenía antes del DELETE y luego a cada uno de sus hijos se le actualiza el PADRE al valor del nuevo código del individuo.

**Punto f:** La solución de este punto se refleja en el trigger ‘actualizar\_codigo\_individuo’, que es un COMPOUND TRIGGER del tipo FOR UPDATE para la columna CODIGO de la tabla INDIVIDUO. Este, al igual que la solución del punto c y d, tiene un BEFORE STATEMENT dónde se genera un array de todos los individuos y sus valores originales, para luego asignar el valor de NULL para toda la columna PADRE y evitar complicaciones con la referencia que esta columna tiene con la columna CODIGO.

En un BEFORE EACH ROW guardamos tanto el código viejo, como el nuevo del individuo a actualizar, y luego en un AFTER STATEMENT, después de la actualización, recuperamos los valores originales de la columna PADRE para todos los valores, excepto para los hijos del individuo, en el cual se asigna el valor del nuevo código del individuo actualizado.