Funciones

`def parse_date_from_mongo_format` La usaremos para convertir las fechas que se encuentran en formato ISO o numberLong, las cuales están dentro de diccionarios respectivos

{"$date": "2020-01-01T00:00:00.000Z"}

{"$date": {"$numberLong":-199843200000}}

Primero debemos comprobar que el dato recibido sea un diccionario, y que tenga la clave de $date, sino es así retornamos None y salimos de la función.

Obtenemos el valor que posee el campo $date y pasamos a los condicionales, para saber cual es su caso.

1. El tiempo en milisegundos (numberLong): Si el valor es numérico, se hace la operación de
    - Tomar el valor numérico y dividirlo en 1000, de forma que obtenemos los segundos (Iniciamos con el tiempo en milisegundos y pasamos a segundos)
    - Con los segundos hacemos uso de datetime.fromtimestamp() para que los convierta en un objeto datetime para Python y poder usarlas después.
2. En formato ISO: Si el valor es un string, lo intentará convertirlo al formato estándar
3. Si no cumple ningún caso, retorna None y salimos de la función

---

`def transformation_dim_dates` *Primera función para trabajar sobre nuestros datos*

Definimos un set (colección de elementos únicos), recorremos cada elemento del archivo `transaction`, buscando acceder a las transacciones anidadas y el campo date.

Se recorre cada una de estas de forma que obtengamos la fecha y posteriormente la formatearemos, para evitar que haya problemas a la hora de agregarla a la base de datos, para finalmente agregarlos al set.

Ahora hemos de recolectar los datos que vamos a insertar, para lo cual recorremos el set, pero ordenado para agregarlo a la base de datos. Generamos las variables que serán usadas para el registro de estas, y son agregadas a la lista.

Por último, se cargan en la base de datos usando `executemany`, que permite insertar **varios registros en una sola operación**.

`def transformation_dim_symbol` *Segunda función para trabajar sobre nuestros datos*

Definimos nuevamente un set, de forma que tengamos de manera única y sin duplicados, recorremos de manera similar, dentro de las cuentas, después dentro de las transacciones para obtener los **symbols** y agregar dicho elemento al set.

Ahora de forma similar que el anterior agregamos los símbolos a la lista. IMPORTANTE se debe agregar con *** symbol, *** debido a que SQLite espera tuplas para agregar.

Una vez que ya están los datos, se realiza la inserción de los valores a la tabla correspondiente, tal cual previamente se realizó.

** Cambio 06/06, tras la necesidad de pasar los datos que ya se habían cargado, se ejecuta la consulta establecida para devolver todos los symbol obteniendo `{symbol_name : id_symbol}`

`def transformation_dim_customers` *Tercera función para trabajar los datos*

Seguimos una estructura ordenada. Hemos de definir una lista, la que usaremos para guardar los elementos que componen cada una de las filas, con los datos que obtenemos del archivo “customers”. Una observación importante, cuando obtenemos la “birthdate” esta debe ser formateada con la primera función definida, para que se pueda guardar en la base de datos, de caso contrario no estaríamos cumpliendo la normalización de los datos propuestas.

Una vez que obtuvimos estos datos iniciales toca el turno de los tier y beneficios que poseen, como estos son un diccionario (tier) y listas (beneficios), tenemos que recorrerlos para obtener la data del diccionario y extraer los beneficios, una consideración que se tuvo para esto fue la de buscar únicamente los **tier que**

esten estado ‘active’ y devolver el valor (bronze, silver, gold) y obviamente cargar la lista de los beneficios con los que tiene el usuario

Convertimos dicha lista en un string, ya que usamos formato TEXT en el campo y procedemos a cargar la base de datos.

** Cambio 06/06, tras la necesidad de pasar los datos que ya se habían cargado, se ejecuta la consulta establecida para devolver todas las “natural_keys” obteniendo

`{customer_natural_key : id_customer}`

`def transformation_dim_accounts` *Cuarta función para trabajar los datos*

Definimos la lista para guardar las cuentas que están registradas, obtenemos los valores de “account_id”, y ‘limit’ este último haciendo la operación de que si no tiene algo, se le asigne 0, en cambio se le pase el valor que posee (De igual forma ningún valor de los de limit está vacio)

Dentro de estos datos están los productos, los cuales como ya vienen en una lista, podemos pasarlos rápidamente a string (teniendo como resultado el tipo de dato TEXT para la base de datos)

Cargamos los datos en la lista inicial, para posteriormente cargarla a la tabla respectiva

** **Cambio 06/06** Tras generar la relación entre las tablas de accounts y customers, se hace uso del resultado obtenido en `relation_customers_and_accounts` y el los datos procesados al cargar la tabla de `customers` para obtener los `id_customers` , asegurando la correcta inserción en la tabla de accounts.

Además se presentan lo previamente mencionado, obtemos el map de las cuentas

`{id_account_natural : ID_ACCOUNT_UNIQUE}`

`def transformation_dim_type_transactions` *Quinta función para trabajar los datos*

** tot: type of transaction

Definimos nuestra lista para guardar los dos tipos de transacción “sell” y “buy”, estas se encuentran en una lista, por lo cual hacemos un recorrido anidado,

inicialmente para recorrer los datos y el segundo para recorrer cada una de las transacciones, de esta forma vamos a obtener el 'transaction_code' que al final corresponde a los ya mencionados.

Estos son agregados al set, para evitar duplicidad, y tener únicamente que hacemos, y posteriormente los añadimos en una lista, para la inserción en la base de datos

** Importante, como hacemos uso de **executemany** tenemos que tener una tupla de datos, por lo cual se define como "**tot,".** Si no se quisiese usar la operación previa, sería una inserción individual para cada una

** Cambio 06/06, tras la necesidad de pasar los datos que ya se habían cargado, se ejecuta la consulta establecida para devolver todas las "natural_keys" obteniendo

`{name_type_transaction:  id_type_transaction}`

`def relation_customers_and_accounts` *Función intermedia para mapear cuentas y usuarios*

La función está pensada para realizar la asignación de la clave natural (username+name) con los id de la cuenta que son encontrados en las lista de cada usuario, por eso se usa 'customers_data' de forma que asignaríamos el id encontrado a dicha clave, dando como resultado.

- es necesario mencionar que si llegan a existir dos cuentas con el mismo id de cuenta, esta se le sobrescribirá al último que es ingresado, lo cual correspondería al caso del `id_account = 627788`

`[{id_account: customer_natural_key}]` mostrando todos los que le pertenecen a un único usuario

---

`def transformation_fact_transactions` *Sexta función para trabajar los datos*

En esta última hacemos uso de la información previa en específico los mapas obtenidos tras hacer la carga de las tablas anteriores.

Mientras vamos iterando sobre los datos de las transacciones, ingresamos a estudiar lo que posee el `account_id` de la transacción, guardando la variable para ser usada y hacer la conexión hacia las cuentas y el dueño de dicha cuenta, esto

lo hacemos con las variables `account_id_unique, customers_accounts` mientras que `customers_id_og` será usada para hacer la conexión entre las tablas de customers y accounts.

Una vez que se generaron las relaciones, ha de comprobar que las fechas estén en un formato correcto, haciendo uso de la función de formateo. Una vez que se han comprobado los datos previos, se procede a ir obteniendo cada uno de los datos que se encuentra dentro de esta, para obtener el monto, precio, total, el código de transacción donde volvemos a usar un mapa para relacionarlo con nuestra tabla de `Type of Transaction`, y lo mismo para los `symbols` .

Una vez que ya está todo, se carga la lista y se inserta en la base de datos, terminando la carga de estos terminando así el proceso de carga.