Funciones, un algoritmo, BigQuery y la ausencia de frameworks

Functions, an algorithm, BigQuery, and the absence of frameworks

Pedro Cano[1]
pedroc777@gmail.com

Resumen: En este artículo vamos explicar cómo solucionar el siguiente problema: realizar una inserción de datos en una tabla de BigQuery usando el lenguaje de programación Python, cuando no podemos usar frameworks, chatGPT, GitHub Copilot, etc., para obtener una solución.

La solución se centra en desarrollar una serie de funciones que nos permitan validar los tipos de datos de los campos de la tabla involucrados en la inserción, ordenar los datos a insertar y darles la estructura requerida según los tipos de datos presentes. Para esto, también mostramos el desarrollo de un algoritmo centrado en realizar dichos procesos.

Palabras clave: datos, campos, funciones, algoritmo, Python, BigQuery, SQL.

Abstract: In this article, we are going to explain how to solve the following problem: perform a data insertion in a BigQuery table using the Python programming language, when we cannot use frameworks, chatGPT, GitHub Copilot, etc., to obtain a solution.

The solution focuses on developing a series of functions that allow us to validate the data types of the table fields involved in the insertion, order the data to be inserted and give them the required structure according to the data types present. For this, we also show the development of an algorithm focused on performing such processes.

Keywords: data, fields, functions, algorithm, Python, BigQuery, SQL.

[1] Universidad Abierta y a Distancia de México

1. Introducción

En este escrito pretendemos mostrar cómo se hace la inserción de datos en una tabla de BigQuery usando el lenguaje de programación Python. Esto es algo se ha realizado cientos de veces (y se seguirá haciendo) en la industria, pero la peculiaridad en este caso es que la explicación mostrada toma como base una hipotética aplicación en la que no pueden usarse frameworks, recursos como chatGPT o GitHub Copilot, bibliotecas que nos permitan realizar esta acción y recepción de datos de manera convencional (la razón es que en este contexto, dichas herramientas no tienen la solución para nuestro problema, no están integradas como tal o no se cuenta con los recursos para acceder a ellas). Lo único que recibe esta aplicación es una cadena de texto que nosotros debemos manipular para lograr nuestro objetivo.

Ahora bien, para poder desarrollar este escrito, primero explicamos a detalle el problema, es decir, damos un contexto del problema planteado arriba. Posteriormente mencionamos algunas cuestiones relacionadas con la función que realiza la inserción en BigQuery y pasamos a dar cuenta de la solución planteada, la cual se compone de una serie de funciones programadas en Python que permiten obtener las estructuras deseadas por BigQuery/SQL.

La descripción de esta solución se compone de varias partes. La primera de ellas es una función cuya misión es validar los tipos de datos de los campos involucrados en la inserción mencionada (para saber si son STRING, INTEGER, BOOLEAN, etc.). Posteriormente, se explica cómo esta función se usa para ejecutar cualquiera de las siguientes dos opciones: 1) la función que ordena los valores a insertar cuando todos los campos de la inserción son de tipo STRING; 2) la función que ordena los valores cuando uno o varios de estos no son de tipo STRING.

Esta última función es un tanto más extensa, pues conlleva el desarrollo de un algoritmo que llamamos *búsqueda por saltos*, el cual, con la ayuda de listas, contadores y ciclos for, nos ayuda a ordenar y dar formato a los datos involucrados en cada inserción. Por este motivo, la explicación de esta función implica también la descripción del algoritmo en abstracto, el algoritmo en el código y un ejemplo de cómo se ejecuta cada iteración del mismo. Una vez que se terminan estas explicaciones, también se termina el escrito, pues después de estos ordenamientos sólo resta ejecutar la inserción que nos permite realizar la inserción.

La última cuestión que debemos aclarar antes de empezar a desarrollar nuestros objetivos, es BigQuery.

Esta (Google, 2023) es una herramienta desarrollada por Google basada en una arquitectura sin servidor que permite almacenar grandes cantidades de información en la nube y que usa SQL para realizar varias de las operaciones pertenecientes a una base de datos tradicional, como consultas, inserciones, actualizaciones, procedimientos almacenados y demás. En este aspecto, junto con el desarrollo del algoritmo mencionado y en el hecho de no poder usar herramientas que faciliten la solución mencionada, recae la importancia del escrito, pues por un lado usamos una herramienta que se encuentra en auge (por el incremento del cómputo en la nube), y por otro nos ponemos en un escenario en el que, por ciertos motivos, nos vemos imposibilitados para usar las herramientas mencionadas.

2. Explicación del problema

Supongamos un escenario: necesitamos insertar información en un proyecto de BigQuery (Jenn, 2022[i]) (de ahora en adelante abreviado como BQ) y necesitamos que dicha información se obtenga a partir de una interfaz gráfica de usuario. Dicho de otro modo: necesitamos que una tabla de BQ se alimente de la información que un usuario le proporciona a través de una aplicación móvil o de escritorio mediante la escritura de los datos. Suena de lo más común, pero la particularidad de este escenario es que dicha aplicación sólo recibe datos mediante cadenas de texto, esto es, únicamente recibe expresiones del tipo "$dato_1, dato_2, dato_3, \dots, dato_n$" ¿Por qué?

Porque en este caso atípico no podemos usar frameworks u otras herramientas (como chatGPT o GitHub Copilot) que nos faciliten esta misión, también debemos suponer el frontend que manejamos en este escenario no nos permite recibir datos de tipo INTEGER, STRING, BOOLEAN o cualquier otro tipo de dato aceptado por BQ, porque no cuenta con las herramientas de desarrollo para ello. La razón para esto puede deberse a que nuestra aplicación se desarrolla en un entorno muy nuevo que no cuenta con integraciones de las herramientas mencionadas, falta presupuesto para pagar por alguna de ellas y demás; lo importante es que sólo puede recibir cadenas de texto y con ellas debemos realizar las inserciones.

Ahora bien, esto es un problema porque las sentencias de inserción en SQL (que es el lenguaje de programación usado por BQ) tienen dos requerimientos que entran en conflicto con el hecho de que nuestra aplicación hipotética sólo trabaje con cadenas de texto. El primero de ellos es la estructura o el orden que deben tener los datos para ser insertados:

```
INSERT INTO producto (productoNombre, productoDesc, productoPrecio, tPid, tiendaId) values
("iPhone XL","iPhone de última generación",20000.00, 2, 1), ("MOTO G","Almacenamiento: 16 Gb, SO:
Android",10000.00, 1, 2),
("RM","Teléfono pequeño y resitente",20000.00, 4, 3),
("ZTE","Almacenamiento: 4GB, SO: Android",6000.00, 3, 4),
("Nokia 5120","Teléfono austero",500.00, 2, 5),
("GALAXY Z Omega","Almacenamiento: 64 Gb, SO: Android",50000.00, 5, 5),
("iPhone 5","Almacenamiento: 60 gb, SO: iOS",6000.00, 1, 4),
("BlackBerry 7230","Almacenamiento: 4 Gb, SO: RM",1000.00, 2, 3),
("Xiaomi Redmi Note 8","Almacenamiento: 64 Gb, SO: Android 11 MIUI 12.5",30000.00, 5, 3),
("Huawei Nova 9","Almacenamiento: 64 Gb, SO: EMUI 12 ",12999.00, 3, 2),
("BlackBerry Z10","Almacenamiento: 32 Gb, SO: BlackBerry 10",15000.00, 4, 1);
```

Figura 1. Sentencia de inserción de SQL.

Como podemos observar, los datos están dispuestos en forma de filas y cada fila tiene un número de elementos que coincide con el número de campos en los que se quiere hacer la inserción. Si la aplicación recibe una cadena de texto en una única fila que no tiene saltos de línea y que tampoco cuenta con una especie de delimitador que indique dónde termina una fila de datos y empieza la siguiente (que es el caso en nuestro escenario), entonces vamos a recibir un error cuando intentemos hacer la inserción.

Por otra parte, en este ejemplo podemos observar que algunos datos que no son de tipo STRING, tienen que ser escritos en la sentencia sin los caracteres “” o „‟. En una cadena de texto del tipo “$dato_1, dato_2, dato_3, \dots, dato_n$”, al ser de tipo STRING, no es posible determinar si dato $dato_2$ es de tipo INTEGER o si $dato_3$ es BOOLEAN. Para BQ todo esto va a llegar como STRING, y cuando intente introducirlo en un campo que no es de ese tipo, se va a generar un error.

Por estos motivos, en los siguientes apartados explicaremos cómo resolver estos problemas mediante funciones desarrolladas en Python[ii]. Sin embargo, antes de explicar la solución debemos aclarar que existen dos casos que se relacionan con los requerimientos mencionados anteriormente: 1) la cadena de texto contiene datos que se van a insertar en campos cuyo tipo de dato es únicamente STRING; 2) la cadena de texto contiene datos que se van a insertar en campos en los que el tipo de dato de alguno de ellos es diferente de STRING[iii]. En el caso 1) únicamente se debe generar la estructura requerida por la inserción explicada arriba, es decir, se deben escribir los paréntesis, comas y “” o „‟ que delimitan a los datos de tipo STRING, así como sus respectivos saltos de línea. En el caso 2), también debe realizarse dicho procedimiento de escritura, pero deben omitirse “” o „‟ donde el tipo de dato no sea STRING. Dicho esto, es momento de explicar la solución.

3. **Consideraciones previas para la solución: la función que inserta los datos.**

El primer paso lógico para hacer una inserción es definir una función que contenga una sentencia del tipo `"""INSERT INTO `"""+dataSet+""".`"""+tabla+"""` ("""+campos+""") VALUES"""+valores+"""""""` donde *dataSet* es el conjunto de datos donde se encuentra la tabla en la que se van a escribir los datos, *tabla* es el nombre de ésta, *campos* se refiere a los campos y *valores* es el conjunto de valores que se va a insertar en la tabla. Sin embargo, en este escenario existen más pasos, pues debemos tener en cuenta las validaciones a realizar que se derivan de los dos casos explicados al final del apartado anterior, es decir, antes de realizar la inserción debemos validar si los campos involucrados en esta inserción son todos de tipo STRING, o si alguno de ellos no lo es. Después de esta validación, la función debe determinar si los datos se ordenan de una manera u otra, pues ambos casos requieren de procesos diferentes para su ordenamiento. Y, finalmente, debe construirse la sentencia SQL y ejecutarse la inserción. Teniendo todo esto en cuenta, la función de inserción tiene la siguiente estructura (Lakshmanan, 2021)[iv]:

```
def insercion (cliente,projectId:str,dataSet:str,tabla:str,campos:str,valores:str): listaCampos =
    campos.split(",")
    validacion = validarDatos(cliente,projectId,dataSet,tabla,campos)

    if(type(validacion).__name___== 'str'):
        valores = generarValoresString(valores,len(listaCampos)) else:
    valores = generarValoresNoString(validacion,valores)


    query_string = """INSERT INTO `"""+dataSet+""".`"""+tabla+"""` ("""+campos+""")
    VALUES"""+valores+"""""""
    query_job = cliente.query(query_string)
```

Figura 2. Función de inserción en Python para BQ.

La descripción de los parámetros de esta función es la siguiente:

1. *cliente*: es el objeto Client usado para ejecutar las sentencias SQL en BQ.
2. *projectId*: es el id del proyecto de BQ que se usa para extraer los metadatos en la función *validarDatos*().
3. *dataSet*: es el nombre del dataset del proyecto de BQ que se usa para extraer los metadatos en la función *validarDatos*().
4. *tabla*: es el nombre de la tabla de BQ de la que se van a extraer los metadatos en la función *validarDatos*().
5. *campos*: son los campos cuyos metadatos se van a extraer en la función *validarDatos*() y en los que se va a insertar información.
6. *valores*: los valores que se van a insertar en la tabla.

Por otro lado, *listaCampos* almacena la lista obtenida de dividir el parámetro *campos* usando una coma como parámetro. *validacion* almacena el resultado del método *validarDatos*() (este método se explica en el siguiente apartado). La estructura if-else nos ayuda a decidir si los campos a insertar son únicamente de tipo STRING o de otro tipo (siguiendo lo mencionado acerca de los casos en cuestión). En cualquiera de los dos casos, los métodos mencionados ordenan los datos de modo que se obtenga la estructura requerida por SQL (estos métodos se explicarán más adelante). Por último, se construye la sentencia SQL y se ejecuta la inserción.

Como podemos observar, realizar la inserción en este escenario requiere de varios subprocesos que cuentan con cierta complejidad y que serán explicados a continuación.

4. La función que valida los datos.

La función que valida los datos tiene el objetivo de terminar si en el conjunto de campos involucrados en la inserción existe alguno cuyo tipo de dato es diferente de STRING o si la totalidad de ellos son de este tipo. Una vez que se determina esto, la función devuelve una lista con los metadatos (Holowczak, 2022)[v] o propiedades de los campos (o más propiamente de la tabla) en los que se insertará la información. Necesitamos esto para saber qué función de ordenamiento usaremos después. La función tiene la siguiente estructura:

```
def validarDatos (cliente, projectId: str,dataSet: str,tabla: str,campos: str)->str: full_table_path =

    projectId  + "."+ dataSet + "." + tabla
    listaCampos = campos.split(",")
    metaDatosTabla = cliente.get_table(full_table_path) found = []

    for j in range(len(listaCampos)):
for k in range(len(metaDatosTabla.schema)): if(listaCampos[j] == metaDatosTabla.schema[k].name):
    found.append(metaDatosTabla.schema[k])



    for x in found:
if(x.field_type != "STRING"): retorno = found
        break else:
    retorno = campos



    return retorno
```

Figura 3. Función de validación de datos.

La descripción de los parámetros es la siguiente:

1. *cliente*: es el objeto Client usado para ejecutar las sentencias SQL de BQ.
2. *projectId*: es el id del proyecto de BQ donde se encuentra la tabla de la que vamos a extraer metadatos.
3. *dataSet*: es el nombre del dataset del proyecto de BQ en el que se encuentra la tabla de la que vamos a extraer metadatos.
4. *tabla*: es el nombre de la tabla de BQ de la que se van a extraer los metadatos.
5. *campos*: son los campos cuyos metadatos se van a extraer.

Por otro lado, las variables tienen las siguientes funciones: *full_table_path*: sirve para construir la ruta completa de

la tabla cuyos metadatos se van a extraer. *listaCampos*: almacena la lista obtenida al dividir la cadena recibida en *campos*, usando la coma como parámetro. *metaDatosTabla* almacena los metadatos de la tabla a la que se quiere insertar información, esto se hace en forma de objeto (de la clase Schema). Esto se logra usando la API de BQ/GCP y *projectId*, *dataSet* y *tabla* como parámetros. *found* es una lista en la que se van a guardar los campos cuyos tipos de dato sea diferente de STRING.

Ahora bien, los ciclos for anidados que se encuentran entre la línea 32 y la 35 tienen la siguiente lógica: el primero recorre, uno a uno, todos los elementos de *listaCampos*, mientras el segundo hace lo mismo, pero con la lista *metaDatos.schema*. Luego se encuentra un if en el que la condición es: si alguno de los elementos de *listaCampos* es igual a alguno de los nombres contenidos en *metaDatos.schema*, entonces guarda ese elemento en la lista *found*.

Una vez que se han terminado los recorridos mencionados, se recorre *found* con la intención de lograr el objetivo de *validarDatos*: la estructura for-in recorre elemento a elemento a *found* y dentro tiene otra estructura if- else, cuya condición es que si alguno de los tipos de datos de los campos guardados en la lista en cuestión es diferente de STRING, entonces *retorno* (la variable a retornar) almacena todo en la lista *found* y se detiene el ciclo. En otro caso, se termina de recorrer la lista y si todos los campos son de tipo STRING, se devuelve *retorno* pero almacenando la cadena de texto que contiene los campos involucrados.

Cuando ya se ha realizado este proceso, en la función *insertar*() se encuentra una variable que guarda los resultados de la función recién descrita. Luego, en la estructura if-else mencionada (Figura 2., líneas 133- 136) se determina si hay que usar el método que ordena los valores para campos cuyos tipos de datos son todos STRING o no. En el siguiente apartado describiremos *generarValoresString*(), que es el método que se ejecuta cuando *validarDatos*() devuelve una cadena de texto.

5. La función que ordena los valores cuando todos los campos son de tipo STRING.

generarValoresString() es una función que nos ayuda a obtener la estructura de datos requerida por BQ, a saber:

$$\begin{matrix}(d_1, & d_2, & d_3, & d_4), \\ (d_5, & d_6, & d_7 & d_8), \\ \vdots & \vdots & \vdots & \vdots \\ (d_{m-3}, & d_{m-2}, & d_{m-1} & d_m);\end{matrix} \qquad \textbf{(1)}$$

Donde $d_1, d_2, d_3, \ldots d_n$ representa el conjunto de datos dispuestos en renglones. En cada renglón, se encuentran paréntesis que encierran un determinado número de datos separados por una coma. El número de datos que se encuentra entre paréntesis depende del número de campos que se van a insertar. Por ejemplo, si
con una inserción vamos a agregar valores a 4 campos, entonces entre cada uno de los paréntesis habrá 4 datos. El número total de renglones va a depender de la cardinalidad del conjunto de datos a insertar[vi].

En este sentido, *generarValoresString*() nos ayuda a obtener una estructura como la mencionada cuando los campos de la cadena de texto recibida son todos de tipo STRING. La estructura de la función es la siguiente:

```
def generarValoresString(valores: str, numeroCampos: int)->str: listaCadena = valores.split(",")
    contador = 0 w = ""
    z = ""
    y = ""

while(contador < len(listaCadena)): for i in range(numeroCampos):
    y = listaCadena[contador]
    z += ("'" + y + "',")
    contador += 1

    w += "(" + z[:-1] + "),\n" z = ""

    return w[:-2] + ";"
```

Figura 4. Función que genera la estructura requerida por la inserción de SQL.

La descripción de los parámetros es la siguiente:

1. *valores*: se trata es una cadena de texto que contiene los valores a insertar en una tabla de BQ.
2. *numeroCampos*: es un número entero que nos indica el número de campos involucrados en la inserción de la tabla de BQ (cuando se llama la función, lo que se recibe aquí es el tamaño de la lista obtenida de dividir la cadena *campos,* lo cual se hace en la función *insercion*()).

Las variables declaradas entre las líneas 2 y 4 tienen las siguientes funciones. *listaCadena* almacena una lista que resulta de dividir la cadena *valores* usando una coma como parámetro. *contador* es una variable que nos sirve para extraer los elementos de *listaCadena* y para realizar el conteo de las iteraciones del ciclo while. w, z, y son variables que nos ayudan a realizar las concatenaciones mostradas arriba.

En cuanto al ciclo while, este nos ayuda a recorrer *listaCadena*. El número de iteraciones del ciclo for depende del número de campos a insertar, de modo que las concatenaciones mostradas ahí se realizan el mismo número de veces que *numeroCampos*. Así, por ejemplo, si *numeroCampos* = 4, entonces dichas concatenaciones se van a realizar 4 veces. Al mismo tiempo, esto también implica que se van a concatenar 4 elementos, es decir, el número de elementos de *listaCadena* que se van a concatenar es el mismo que el de *numeroCampos*. Dicho sea de paso: *numeroCampos* va a determinar el número de renglones de la estructura en cuestión.

Por otro lado, las concatenaciones se realizan de la siguiente manera:

1. El elemento extraído de *listaCadena* con *contador* se guarda en y.
2. En z se guarda y concatenado con la coma y las comillas que requiere SQL por tratarse de un dato de tipo STRING.
3. A *contador* se le suma 1 para pasar al siguiente elemento.
4. Cuando se acaban las iteraciones del ciclo for, la cadena concatenada en z se concatena con los paréntesis que delimitan a los datos del renglón y se guardan en w. En este punto también se verifica el valor de *contador* para ver si se sigue cumpliendo la condición del while. Si sí, se repite el proceso. En caso contrario se termina.
5. z se iguala con “” (en cierto sentido se vacía) para repetir el proceso y formar otro renglón. Esto se repite hasta que se acaben los elementos de *listaCadena* y se deje de cumplir la condición mencionada. En este caso, w es el valor que se regresa con ciertas modificaciones que eliminan caracteres que no se necesitan y agrega un ; para indicar que ahí se termina esa parte de la sentencia.

Una vez que se ejecuta esta función, la cadena que obtenemos es la siguiente (que tiene la estructura requerida por BQ y se obtiene a partir de la cadena de texto “Avenida 1,Python1,Avenida2,Python2”):

```
('Avenida 1','Python1'),
('Avenida2','Python2');
```

Figura 5. Resultado de ejecutar *generarValoresString*().

Como mencionamos, esta aplicación nos sirve para cuando los campos con los que estamos trabajando son todos de tipo STRING (la muestra de ello es la Figura 5). Sin embargo, cuando los datos son de tipo INTEGER, BOOLEAN, FLOAT, etc., o cuando en este conjunto hay campos de tipo STRING y otros, debemos realizar un procedimiento parecido, pero con ciertas modificaciones que aumentan un tanto la complejidad de lo ya explicado. Esto se trata en el siguiente apartado.

6. La función que ordena los valores cuando no todos los campos son de tipo STRING.

Como bien dijimos, el hecho de que en una inserción de SQL haya campos cuyo tipo de datos sea STRING, junto con campos cuyo tipo de dato es de cualquier otro, aumenta la complejidad de lo explicado en el apartado anterior, pues ahora los renglones de datos no tendrán esta estructura: $("d_1","d_2","d_3",\dots,"d_n")$, sino que tendrán otras como $(d_1,"d_2","d_3",\dots,d_n)$, $("d_1",d_2,d_3,\dots,d_n)$, etc., porque cualquiera de estos puede ser un entero, booleano, flotante, etc.

La complejidad en este contexto radica en saber qué elementos van a llevar comillas y cuáles no. Se ha desarrollado una solución para esto, pero antes debemos explicar ciertos aspectos necesarios para llegar a ella. Por lo pronto, empezaremos por decir que en lugar de mostrar la totalidad de la estructura de la función *generarValoresNoString*(), mostraremos secciones de la misma, pues cada sección tiene una misión específica e implica descripciones más puntuales que las de las funciones anteriores.

La primera sección es la de los parámetros y las variables iniciales:

```
def generarValoresNoString(validacion:list, valores: str)->str:

listaCadena = valores.split(",") listaValidacion = [] listaValidacion2 = []
w = ""
z = ""
diferencia = len(validacion)
```

Figura 6. Sección de parámetros y variables iniciales de la función *generarValoresNoString*().

Los parámetros *validación* y *valores*, y las variables *listaCadena*, w e y tienen exactamente las mismas funciones que en la función *generarValoresString*(), por ello no las explicaremos aquí. Por otro lado, *listaValidacion* y *listaValidacion2* son listas que nos permiten guardar por separado los resultados obtenidos en *validación* (esto lo explicaremos más adelante). *diferencia* es una variable que nos permite almacenar el tamaño de la lista *validacion*. Este valor será importante más adelante porque nos permitirá realizar procedimientos relevantes en esta función.

La siguiente sección es la de validación. Redundante, sí, pero aún con la validación ya obtenida en *validacion* necesitamos hacer otra por el siguiente motivo: la lista en cuestión contiene instancias de la clase *Schema* que indican el nombre y el tipo de dato de los campos involucrados en una inserción. Entre estos campos puede haber algunos que sean de STRING y otros de tipo diferente. En este caso, es necesario separar el conjunto de los que no son STRING de los que sí, para saber cuáles llevan comillas y cuáles no. Por ello, la sección de validación se estructura de la siguiente manera:

```
    for x in validacion:
if(x.field_type != 'STRING'): listaValidacion.append(validacion.index(x))

    for y in validacion:
if(y.field_type == 'STRING'): listaValidacion2.append(validacion.index(y))
```

Figura 7. Sección de parámetros y variables iniciales de la función *generarValoresNoString*().

Aquí se encuentran dos ciclos for-in. El primero recorre la totalidad de *validacion*. Recordemos que dicha lista contiene objetos de la clase *Schema*, razón por la cual en cada iteración se pregunta si la propiedad *field_type* de cada uno de estos objetos es diferente de STRING. Si esto es verdadero, entonces los índices en los que se encuentran tales objetos se almacenan en la lista *listaValidacion* con el método *append*(). En resumen: *listaValidacion* contiene los índices de *validacion* en los que se encuentran valores cuyo tipo de dato es diferente de STRING. La misma lógica se sigue en el segundo for-in, con la diferencia de que en *listaValidacion2* se almacenan los índices de los campos cuyo tipo de dato son STRING. Hacemos esto porque el saber en qué índices se encuentran estos campos nos ayudará más adelante (en conjunto con *diferencia*) a determinar qué datos deben ir entre comillas y cuáles no, pero aún faltan dos pasos (secciones) para llegar a ese punto.

La siguiente sección es la de igualación de tamaño entre listas. Esto más que se ser una necesidad de la función en cuestión, es un requerimiento de Python: si intentamos ejecutar esta función sin esta sección, entonces obtenemos el error "IndexError: list index out of range" debido a la diferencia de tamaños que existe a veces entre *listaValidacion* y *listaValidacion2*. Estas listas deben ser del mismo tamaño para que, cuando sean recorridas, no obtengamos el error mencionado[vii].

La estructura de la sección es la siguiente:

```
if(len(listaValidacion)>len(listaValidacion2)):
diff = len(listaValidacion) - len(listaValidacion2) for i in range(0,diff):
            listaValidacion2.append("-") elif(len(listaValidacion)<len(listaValidacion2)):
diff = len(listaValidacion2) - len(listaValidacion) for i in range(0,diff):
listaValidacion.append("-")
```

Figura 8. Sección de igualación entre listas de la función *generarValoresNoString*().

Entre las líneas 70 y 77 se encuentra una estructura if-elif. La parte del if verifica si el tamaño de *listaValidacion* es mayor que el tamaño de *listaValidacion2*. En caso afirmativo, el tamaño de *listaValidacion2* se resta al de *listaValidacion* y se usa *diff* (la diferencia) en un ciclo for in para llenar con guiones los índices que le faltan a *listaValidacion2* para tener el mismo tamaño de la otra lista[viii].

En la siguiente sección, y con el mismo objetivo de no obtener el error "IndexError: list index out of range", debemos igualar el tamaño de ambas listas con el tamaño de la lista *validación*.

La estructura de esta sección es la siguiente:

```
for i in range(0,diferencia-len(listaValidacion)): listaValidacion.append("-")
listaValidacion2.append("-")
```

Figura 9. Sección de igualación de listas con el tamaño de *validación* de la función *generarValoresNoString*().

Como se puede observar, se trata únicamente de un ciclo for que va de 0 a el número resultante de restar el tamaño de las listas obtenido en la sección anterior y el tamaño de la lista *validacion*[ix].

Ahora bien, todo lo que hemos venido explicando coincide en este punto, pues varios de estos aspectos se van a utilizar para explicar algo que llamamos algoritmo de *búsqueda por saltos*. Lo hemos llamado así a falta de un mejor término y porque requiere de ciertos "saltos" entre elementos (determinados por la variable *diferencia*) que nos ayudan a identificar qué elementos deben tener tal o cual formato. La explicación de este algoritmo se va a dividir en tres partes: 1) el algoritmo explicado en abstracto; 2) el algoritmo en el código (de la siguiente sección de la función); y 3) el algoritmo con un ejemplo.

7. Algoritmo de *búsqueda por saltos* (en abstracción)

Coloquialmente entendemos que un algoritmo es un conjunto finito de pasos claros y distintos que están destinados a resolver un problema[x]. El problema que tenemos aquí es: necesitamos generar con Python una estructura textual (aceptada por SQL) que nos permita insertar un conjunto de datos en una tabla de BQ, cuando los campos involucrados en la inserción tienen tipos de dato diferentes de STRING (todo esto con el trasfondo de nuestra aplicación peculiar mencionada al principio; los pasos a seguir se muestran más adelante).

Ahora bien, lo primero que debemos observar para resolver dicho problema es recordar parte de la estructura de la *Figura 1*, la cual antes de la palabra reservada *values* tiene los nombres de los campos en los que se van a insertar datos, y después (dispuestos en renglones y columnas) tiene a los datos que se van a insertar. Considerado esto, tenemos una figura como la siguiente:

```
    (productoNombre, productoDesc, productoPrecio, tPid, tiendaId) values
("iPhone XL","iPhone de última generación",20000.00, 2, 1), ("MOTO G","Almacenamiento: 16 Gb, SO:
Android",10000.00, 1, 2),
("RM","Teléfono pequeño y resitente",20000.00, 4, 3),
("ZTE","Almacenamiento: 4GB, SO: Android",6000.00, 3, 4),
("Nokia 5120","Teléfono austero",500.00, 2, 5),
("GALAXY Z Omega","Almacenamiento: 64 Gb, SO: Android",50000.00, 5, 5),
("iPhone 5","Almacenamiento: 60 gb, SO: iOS",6000.00, 1, 4),
("BlackBerry 7230","Almacenamiento: 4 Gb, SO: RM",1000.00, 2, 3),
("Xiaomi Redmi Note 8","Almacenamiento: 64 Gb, SO: Android 11 MIUI 12.5",30000.00, 5, 3),
("Huawei Nova 9","Almacenamiento: 64 Gb, SO: EMUI 12 ",12999.00, 3, 2),
("BlackBerry Z10","Almacenamiento: 32 Gb, SO: BlackBerry 10",15000.00, 4, 1);
```

Figura 10. Estructura de campos y datos de una sentencia de inserción.

Como podemos observar, en la parte posterior de *values* están los campos de la sentencia, los cuales determinan, por decirlo de algún modo, las columnas en las que se disponen los datos, pues debajo del nombre de cada campo, se encuentran datos que corresponden a sus respectivos tipos de dato: *productoNombre* es un campo de tipo STRING y los elementos encontrados abajo son de tipo STRING, por eso se escriben entre comillas; *productoPrecio* es de tipo FLOAT, y los datos encontrados abajo son de tipo FLOAT, lo que implica que no lleven comillas.
Podemos abstraer lo mostrado en la Figura 10 para obtener la siguiente estructura:

$$(c_0, \quad c_1, \quad c_2, \quad c_3) \quad \textbf{(2)}$$
$$(d_0, \quad d_1, \quad d_2, \quad d_3),$$
$$(d_4, \quad d_5, \quad d_6, \quad d_7), (d_8, \ d_9, \ d_{10}, \ d_{11}), \ (d_{12}, \ d_{13}, \ d_{14}, \ d_{15}), (d_{16}, \ d_{17}, \ d_{18}, \ d_{19}), \ (d_{20}, \ d_{21}, \ d_{22}, \ d_{23}), \ (d_{24}, \ d_{25}, \ d_{26}, \ d_{27}), \ (d_{28}, \ d_{29}, \ d_{30}, \ d_{31}), \ (d_{32}, \ d_{33}, \ d_{34}, \ d_{35}), \ (d_{36}, \quad d_{37}, \ d_{38}, \quad d_{39}),$$
$$(d_{40}, \quad d_{41}, \quad d_{42}, \quad d_{43});$$

Donde c_0 a c_3 simbolizan los campos involucrados en la inserción y d_0 a d_{43} representan los datos ubicados debajo que se van a insertar a la tabla. Podemos hacer una abstracción aún mayor y obtener una estructura como la siguiente:

$$\begin{matrix} (c_0, & c_1, & \cdots & c_n) \\ (d_0, & d_1, & d_2, & d_3), \\ (d_4, & d_5, & d_6, & d_7), \\ \vdots & \vdots & \vdots & \vdots \\ (d_{m+1}, & d_{m+2}, & \cdots & d_{m+n}); \end{matrix} \quad \textbf{(3)}$$

En esta expresión c y d tienen exactamente la misma función que en la anterior. n representa el número de campos, m representa la posición de cada dato en cada renglón y $m+n$ representa la última posición en un renglón de acuerdo al número total de campos. Antes de seguir avanzando debemos hacer algunas aclaraciones:

1. Es evidente que la estructura mostrada arriba se asemeja a una matriz, de hecho si nos centramos únicamente en la parte de los datos podríamos representarla como una. Sin embargo, dado que no estamos tratando únicamente con números, sino con diferentes tipos de datos, la consideraremos más como un arreglo.
2. Debemos notar (porque esto será importante en unos momentos) que n es la misma cantidad que *diferencia*, pues ambas nos indican el tamaño del conjunto de campos involucrados en nuestra inserción.
3. Las posiciones de los datos del primer renglón siempre coinciden con las posiciones de los campos. Es decir, d_0 siempre coincide con c_0, d_1 con c_1 y así sucesivamente[xi].

Una vez realizadas estas explicaciones, ya podemos empezar a explicar el algoritmo propiamente. Lo primero que debemos notar es que gracias a los arreglos descritos, podemos ver cuáles son los datos que le corresponden a cada campo. Por ejemplo, si tenemos un arreglo de 4 campos y 12 datos como el siguiente:

$$\begin{matrix} (c_0, & c_1, & c_2 & c_3) \\ (d_0, & d_1, & d_2, & d_3), \\ (d_4, & d_5, & d_6, & d_7), \\ (d_8, & d_9, & d_{10}, & d_{11}); \end{matrix} \quad \textbf{(4)}$$

En el que podemos observar que a c_0 le corresponden d_0, d_4,y d_8. Algo parecido pasa con el resto de campos. Para nosotros es evidente porque lo tenemos ordenado visualmente y podemos incluso señalarlo, pero para la computadora no. La máquina necesita que de algún modo, ya sea en Python, JavaScript o Java, le indiquemos qué elementos le corresponden a cada campo (pues esto también le ayudará a saber cuáles llevan comillas y cuáles no) y para que sepa esto, debemos indicárselo con n o *diferencia* y teniendo en cuenta la aclaración número tres. Por ejemplo, si queremos indicarle a la computadora qué datos le corresponden a c_1, primero debemos tener en cuenta que el primer dato que le corresponde es d_1. Luego, para saber cuáles son los datos siguientes, suponemos que $m = 1$ (por estar d_1 en la posición 1) y le sumamos $n = 4$. De este modo
$m + n = 1 + 4 = 5$, por lo tanto, el siguiente dato es d_5 por estar en la posición 5. Siguiendo esta misma lógica, el siguiente dato será d_9[xii].
Este dato (d_9) también será el último, pues como mencionamos hace algunas páginas, la división del número de datos entre el número de campos determina el número de renglones. De modo que si $|\{c_0, c_1, c_2, c_3\}| = 4$ y $|\{d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}\}| = 12$, entonces $12/4 = 3$, lo que nos indica que sólo habrá tres renglones en este arreglo y que a cada campo sólo le corresponden tres datos porque sólo existen tres renglones.

Ahora bien, supongamos que c_0 es de tipo BOOLEAN, c_1 es de tipo INTEGER, y c_2 y c_3 son de tipo STRING. Con lo ya explicado, sabemos qué elementos le corresponden a c_0 y c_1, los cuales no deben llevar comillas, mientras que los de c_2 y c_3 sí las llevan, por lo que la cadena que debemos pasar a la sentencia de inserción debe verse de la siguiente manera:

$$\begin{array}{llll} (c_0, & c_1, & c_2 & c_3) \\ (d_0, & d_1, & "d_2", & "d_3"), \\ (d_4, & d_5, & "d_6", & "d_7"), \\ (d_8, & d_9, & "d_{10}", & "d_{11}"); \end{array} \qquad \textbf{(5)}$$

Como podemos observar, d_0, d_4, d_8 y d_1, d_5, d_9 no llevan comillas, pues los tipos de dato de sus campos correspondientes no lo exigen. Los datos restantes sí las necesitan porque sus respectivos campos son de tipo STRING.

Antes de pasar a la siguiente sección, haremos un pequeño resumen de los pasos del algoritmo en cuestión:

1. Determinar el número de renglones del arreglo mediante la división del número de datos entre el número de campos.
2. Determinar el número de campos para saber qué tantos elementos se deben saltar en cada iteración (y qué tantos deben escribirse en cada renglón).
3. Determinar, a partir de las posiciones de los campos, qué datos serán los primeros involucrados al empezar a realizar las sumas (o saltos).
4. Realizar las sumas que nos permiten saber qué elementos le corresponden a cada campo.
5. Agregar o quitar las comillas a cada dato, según el tipo de dato del campo que le corresponde.
6. Detenerse hasta que se haya alcanzado el número de renglones y la totalidad de datos.

En la siguiente sección, explicaremos cómo todo esto se representa en el código. Será un tanto más complicado, pues dado que la máquina carece del recurso visual del arreglo, todo el proceso se realizará a través de ciclos for, listas y contadores.

8. Algoritmo de *búsqueda por saltos* (en código)

En esta sección mostraremos cómo se codifica el algoritmo de *búsqueda por saltos*. Por tanto, lo primero que haremos será mostrar una parte de la estructura que tiene el código en Python:

```
for i in range(0,len(listaCadena)): for j in range(0,diferencia):
    if(listaValidacion[j] == i):
    y = listaCadena[i]
    z += (""+y+",")
        listaValidacion[j] += diferencia elif(listaValidacion2[j] == i):
    y = listaCadena[i]
    z += ("'"+y+"',")
    listaValidacion2[j] += diferencia

    w += "" + z[:-1] + "," z = ""
```

Figura 11. Primera parte del código del algoritmo de *búsqueda por saltos*.

Como podemos observar, este código se compone de dos ciclos for, uno anidado dentro del otro. El primero de ellos recorre todos los elementos de *listaCadena* (que contiene los valores a insertar en la tabla de BQ), y el segundo sólo se repite un número de veces igual al número de campos que se van a usar en la inserción. Si bien en este punto no se forman propiamente los renglones, sí se toma en cuenta *diferencia* para tener un orden en la asignación de las comillas y llevar el conteo de los renglones para detenerse cuando sea necesario (primer paso del algoritmo). Esto también implica que por cada elemento recorrido de *listaCadena*, se van a ejecutar n (*diferencia*) iteraciones del segundo for (segundo paso del algoritmo).

Por otra parte, tenemos una estructura if-else. La lógica de esta estructura es la siguiente: si un elemento de *listaValidacion* (la lista que contiene los índices de los campos cuyo tipo de dato no es STRING) es igual al índice que lleva el conteo de *listaCadena* (i), entonces el elemento de *listaCadena* se guarda en la variable y; y se concatena con un carácter vacío y una coma para acumularse en z (sin comillas porque en este apartado se encuentran los datos que no son STRING); y por último, al elemento de *listaValidacion* se le suma diferencia.

Antes proseguir debemos aclarar ciertas cuestiones. La primera es que en las primeras iteraciones de este ciclo for (las primeras n iteraciones que coinciden con el número de campos involucrados en la inserción), el programa está determinando cuáles son los primeros datos que coinciden con los campos. Es decir, *listaValidacion* contiene los índices de los campos, pero también son los índices de los datos iniciales con los que se harán los saltos mencionados (lo cual es el tercer paso de nuestro algoritmo).

La segunda aclaración es que los saltos en esta parte del código se realizan con $listaValidacion[i] += diferencia$ (aquí se ejecuta el paso número cuatro del algoritmo[xiii]): una vez que se encuentra una coincidencia entre el elemento de esta lista y el índice del dato, al elemento de la lista se le suma *diferencia* para realizar el salto y obtener el siguiente índice, de modo que en la siguiente iteración, si vuelve a haber una coincidencia (entre el nuevo valor de *listaValidacion* y el índice del dato), se repetirá el proceso y se obtendrá el valor del siguiente dato a insertar.

La tercera aclaración es que el procedimiento del elif es exactamente igual al que acabamos de describir, con la diferencia de que en esta condición es donde se asignan las comillas a los datos indicados porque *listaValidacion2* contiene los índices de los campos que son de tipo STRING.

Una vez que se termina el proceso de validación y se alcanza el límite de iteraciones para el segundo for, se vuelve a hacer una concatenación (una por cada iteración del primer for) en la que a z se le quita el último elemento, se concatena con un carácter vacío y una coma, y se acumula en la variable w. Luego z se "vacía" y se repite el proceso hasta alcanzar el límite de *listaCadena*. Si bien en esta parte del código no se forman los renglones, sí se hace el conteo indicado para perfilar la construcción de los mismos.

Para la generación de los renglones se codificaron otras cuantas líneas que se explican a continuación:

```
    w = w[:-1]

    listStr = w.split(",") listStr2 = []
    contador = 0
    rango = len(listStr)/diferencia

for k in range(0, int(rango)): listStr2.append("")

    for i in range(0,int(rango)):
    for j in range(0, diferencia):
    listStr2[i] += (listStr[contador]+",") contador += 1

for i in range(0,len(listStr2)): listStr2[i] = "("+listStr2[i][:-1]+")"

    w = ""
for i in range(0,len(listStr2)): w +=  listStr2[i] + ",\n"

    w = w[:-2] + ";"
```

Figura 12. Segunda parte del código del algoritmo de *búsqueda por saltos* (generación de renglones).

En la línea 104 se retoma la variable *w* para quitarle el último carácter, pues este impide obtener la estructura deseada. Luego en *listStr* se almacena una lista que se obtiene de dividir la cadena obtenida en la línea recién descrita (se guardan los datos a insertar, pero esta vez con el formato que le corresponde a cada uno, con o sin comillas). *listStr2* nos permite inicializar una lista que usaremos más adelante. *contador* es un contador que nos permite realizar el conteo en loops posteriores. *rango* es la variable que almacena el número de renglones, el cual se obtiene, como dijimos, de dividir el tamaño del conjunto de datos entre el número de campos.

El loop de las líneas 111 y 112 tiene el objetivo de llenar *listStr2* con espacios vacíos para que tenga el mismo tamaño de *rango*. Hacemos esto para evitar el error "IndexError: list index out of range". Si no lo hacemos, como la lista no tiene elementos ni índices, cualquier operación que se realice sobre ella se considerará como fuera de rango.

El ciclo de las líneas 115 a 118 tiene el objetivo de agrupar los datos en renglones. El primer loop itera de 0 al número de renglones obtenido. El segundo va de 0 a diferencia, esto con el objetivo de agrupar el número debido de datos en cada renglón. Esto se hace obteniendo cada elemento de *listStr2* (con ayuda de *contador*[xiv]) para concatenarlo con una coma y acumular esta cadena en cada uno de los elementos de *listStr2*. Dicho de otra manera, el primer loop itera una vez por cada renglón, y en cada uno de estos renglones se contatena/acumula el número de datos que le corresponden.

En el loop de las líneas 120 y 121 únicamente se agregan los paréntesis que delimitan a cada renglón y se elimina el último carácter, que es una coma que está de más. En la línea 123 se "vacía" la variable *w* para poder concatenar en ella cada elemento de *listStr2*, junto con una coma y su respectivo salto de línea. Por último, a *w* se le quitan los últimos dos caracteres (un salto de línea y una coma que sobran) y se le agrega un ; para poder retornarla.

Aquí es donde se termina la explicación del algoritmo de *búsqueda por saltos*, junto con algunos recursos que se tuvieron que agregar para completarlo. También se finaliza la explicación de la función *generarValoresNoString*(). A continuación se muestra un ejemplo de cómo funciona este código.

9. Algoritmo de *búsqueda por saltos* (en acción)

Para este ejemplo supongamos que tenemos una tabla llamada *Proveedores*, a la que queremos insertar dos datos ("1,proveedor1,2,proveedor2") en los campos *proveedorId* y *proveedorNom*. *proveedorId* es de tipo INTEGER y *proveedorNom* es de tipo STRING.

Ahora bien (aquí vamos a omitir algunos pasos para hacer más dinámica esta explicación), lo primero que debemos considerar es que lista validación va a tener los siguientes elementos:

1. SchemaField('proveedorId', 'INTEGER', 'NULLABLE', None, (), None).
2. SchemaField('proveedorNom', 'STRING', 'NULLABLE', None, (), None).

Los cuales tienen información acerca de los campos mencionados. La propiedad que importa es la llamada *field_type,* pues es la que contiene el tipo de dato de cada campo. Los ciclos for que trabajan con las listas de validación son los que se encargan de usar esta propiedad para almacenar los índices de dichos campos. En este ejemplo, *listaValidacion* tiene el número 0 como elemento y *listaValidacion2* tiene el número 1.

Luego de hacer la igualación de tamaños, es momento de pasar a los ciclos for que realizan los saltos.

1. Primera iteración del for posterior $i = 0$:
   a. Primera iteración del for interior:
      i. $j = 0$.
      ii. $listaValidacion[j] = 0$ (es igual a i).
      iii. $listaCadena[0] = 1$.
      iv. 1 se concatena en z sin comillas y con una coma.
      v. $listaValidacion[j] += diferencia = 0 + 2$, por tanto $listaValidacion[0] = 2$.
   b. Segunda iteración del for interior:
      i. $j = 1$.
      ii. $listaValidacion[j] = '-'$, pasamos a elif.
      iii. $listaValidacion2[j] = '-'$.
2. Segunda iteración del for posterior $i = 1$:
   a. Primera iteración del for interior:
      i. $j = 0$.
      ii. $listaValidacion[j] = 2$, pasamos a elif.
      iii. $listaValidacion2[j] = 1$ (es igual a i).
      iv. $listaCadena[1] = proveedor1$.
      v. $proveedor1$ se concatena en z con comillas y con una coma.
      vi. $listaValidacion2[j] += diferencia = 1 + 2$, por tanto $listaValidacion2[0] = 3$.
   b. Segunda iteración del for interior:
      i. $j = 1$.
      ii. $listaValidacion[j] = '-'$, pasamos a elif.
      iii. $listaValidacion2[j] = '-'$.
3. Tercera iteración del for posterior $i = 2$:
      i. $j = 0$.
      ii. $listaValidacion[j] = 2$ (es igual a i).
      iii. $listaCadena[2] = 2$.
      iv. 2 se concatena en z sin comillas y con una coma.
      v. $listaValidacion[j] += diferencia = 2 + 2$, por tanto $listaValidacion[0] = 4$.
   b. Segunda iteración del for interior:
      i. $j = 1$.
      ii. $listaValidacion[j] = '-'$, pasamos a elif.
      iii. $listaValidacion2[j] = '-'$.
4. Cuarta iteración del for posterior $i = 3$:
   a. Primera iteración del for interior:
      i. $j = 0$.
      ii. $listaValidacion[j] = 4$, pasamos a elif.
      iii. $listaValidacion2[j] = 3$ (es igual a i).
      iv. $listaCadena[3] = proveedor2$.
      v. $proveedor2$ se concatena en z con comillas y con una coma.
      vi. $listaValidacion2[j] += diferencia = 3 + 2$, por tanto $listaValidacion2[0] = 5$.
   b. Segunda iteración del for interior:
      i. $j = 1$.
      ii. $listaValidacion[j] = '-'$, pasamos a elif.
      iii. $listaValidacion2[j] = '-'$
      iv. En este punto se terminan las iteraciones.

Cuando se terminan las operaciones, obtenemos una cadena de texto como la siguiente: "1,'proveedor1',2,'proveedor2',", la cual, después de ser tratada por la segunda sección del código del algoritmo, permite obtener la siguiente estructura:

```
(1,'proveedor1'),
(2,'proveedor2');
```

Figura 13. Resultado del algoritmo y la función *generarValoresNoString*().

Como podemos observar, los elementos que son de tipo STRING llevan sus respectivas „", y los que son de tipo diferente no la llevan. Además, los paréntesis, comas, saltos de línea y ; se encuentran en los lugares indicados. Una vez que hemos terminado esta explicación, debemos regresar a la función *inserción*().

10. De regreso a la función *inserción*()

Retomando la *Figura 2*, podemos ver que ya hemos explicado los elementos necesarios para que se realice la inserción: explicamos las funciones *validarDatos*(), *generarValoresString*() y *generarValoresNoString*(), las cuales nos ayudan a tener la estructura necesitada por BQ para que se puedan introducir los datos. Una vez que se ha realizado la validación de las líneas 133-136, todos los datos generados con estas funciones y los recibidos como parámetros, se concatenan en la variable *query_string* y luego esta se pasa como parámetro para realizar la inserción. Si no hay errores, se realizará este proceso.

11. Conclusiones

Primero debemos señalar lo obvio: la complejidad computacional de las funciones es $O(n^2)$, debido a los numerosos for anidados que encontramos a lo largo de ellas. Sabemos que esto puede afectar el desempeño de las funciones cuando nos encontremos en escenarios con grandes cantidades de información, por lo cual se debe buscar una alternativa con una complejidad menos agresiva. Suponemos, pues, que se pueden explorar otras opciones que usen estructuras de datos (como Great Learning, 2022, que usa una estructura de datos para revertir una lista ligada) con un enfoque iterativo, o incluso buscar alguna opción recursiva que nos permita resolver este problema sin necesidad de caer en el peor de los casos. Sin embargo, en este espacio no es posible realizar esta tarea porque nos encontramos con una primera aproximación a la resolución del problema planteado y al algoritmo de *búsqueda por saltos* (el cual también se encuentra ligado a $O(n^2)$). Además esto se saldría de los objetivos del escrito.

En este mismo sentido, también queremos resaltar la importancia del algoritmo recién mencionado. Si bien la función principal de éste es generar la estructura requerida por BQ, tenemos la suposición de que puede tener otras aplicaciones. Es decir, pensamos que puede ser usado para realizar alguna especie de búsqueda en estructuras parecidas al arreglo mostrado en páginas anteriores o podría, por ejemplo, usarse cuando se desee realizar una búsqueda en arreglos de dos dimensiones con técnicas parecidas a la *búsqueda por filas* o *búsqueda por columnas* (algoritmos que tienen estructuras similares al de *búsqueda por saltos*). Así también, las demostraciones por inducción matemática y demás cuestiones relacionadas con análisis y diseño de algoritmos quedan pendientes, pero no olvidadas, para momentos posteriores.

Por otra parte también estamos conscientes de una de las objeciones más grandes que pueden hacerse a este escrito: el uso de algún framework o de la API de Google Cloud para Python, e incluso en este momento el de una "inteligencia artificial", como chatGPTno GitHub Copilot, podrían haber hecho la inserción más sencilla, incluso hubiera hecho más sencillo el desarrollo de toda la aplicación. Sí, es verdad. Sin embargo, el escenario en el que nos pusimos (el de la aplicación que únicamente trabaja con cadenas de texto) también nos exigió pensar en cómo sería trabajar sin estas erramientas. No estamos en contra de su uso, incluso pensamos que estos recursos nos ayudan a ser más eficientes y nos dan el tiempo para pensar en cosas de mayor complejidad, pero también quisimos pensar en cómo sería estar en un escenario en el que hubiera que codificar desde 0 y donde no hubiera este tipo de herramientas para ayudarnos.

En relación con lo anterior, lo que aquí desarrollamos puede ser integrado a algún framework o alguna biblioteca cuya aplicación se centre en BQ o incluso en SQL y demás. Esto es, se puede usar lo aquí descrito para mejorar procesos (si es posible y a reserva de la complejidad mencionada) trabajados en Python o en cualquier otro lenguaje. Aquí usamos dicho lenguaje de programación debido a su popularidad en el rubro de los datos, pero si hay un lenguaje en el que se puedan declarar funciones y arreglos, lo mostrado aquí es perfectamente replicable (muestra de ello son las pruebas realizadas en JavaScript para combatir ciertos errores).

Esta replicabilidad se debe a las estructuras matemáticas y computacionales mostradas arriba, las cuales, a pesar de las diferencias sintácticas y lingüísticas a las que estamos acostumbrados, son las mismas y funcionan de la misma manera, al menos en los contextos relacionados con el cómputo. Con esto también queremos recalcar que uno de los retos para este trabajo fue el poner en ejercicio la lógica más que la codificación, pues aquí lo que implicó más tiempo y esfuerzo fue encontrar el algoritmo, el cual no depende del lenguaje de programación, sino, como acabamos de decir, de las estructuras matemáticas y computacionales que se nos pusieron en frente.

Esto último sirve como punto de partida (y como punto final para este escrito) para trabajar, en espacios posteriores, las cuestiones que se han dejado abiertas, pues éstas también van más a allá de codificar con x o y lenguaje de programación y se encuentran más orientadas a las estructuras de los datos, los algoritmos y las matemáticas.

12. Fuentes

- Algoritmo de búsqueda. (30 de noviembre de 2022). En *Wikipedia*. https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda
- Google. (2023) *¿Qué es BigQuery?*. Recuperado de: https://cloud.google.com/bigquery/docs/introduction?hl=es-419.
- Google. (2023) *Crear cuentas de servicio*. Recuperado de: https://support.google.com/a/answer/7378726?hl=es
- Holowczak. (2022, 19 de enero). *Python Programming with Google BigQuery*. Recuperado de: https://holowczak.com/python-programming-with-google-bigquery/7/
- Jenn, Jie. (2022, 13 de julio). *Getting Started With Google BigQuery API In Python* [video]. YouTube: https://www.youtube.com/watch?v=lLPdRRy7dfE&t=3s
- Lakshmanan, Lak. (2021, 10 de febrero). *How to trigger Cloud Run actions on BigQuery events*. Recuperado de: https://cloud.google.com/blog/topics/developers-practitioners/how-trigger-cloud-run- actions-bigquery-events
- Ponce, Jahaziel. (2021, 21 de febrero). Algoritmos de búsqueda. Recuperado de: https://jahazielponce.com/algoritmos-de-busqueda/

13. Notas

[i] Para realizar las inserciones en BQ usamos la librería de la API de ésta, siguiendo los pasos descritos por Jenn, 2022. De este recurso tomamos la manera de importar los módulos correspondientes y configurar el ambiente. Esto no se ve en el código citado en el artículo, pero es importante mencionarlo por el hecho de que toda esta solución tiene el objetivo de insertar información de BQ. De igual manera, esto nos ayudó, junto con Google, 2023, a configurar todo lo relacionado con la cuenta de servicio que da acceso a este recurso. Esto tampoco se encuentra en el artículo o en el código por cuestiones de privacidad.

[ii] En los siguientes apartados vamos citar código, pero para tener una referencia más general del mismo también se proporciona el siguiente repositorio de GitHub: link de github.

[iii] Cuando hablamos de tipos de datos diferentes de STRING, nos referimos principalmente a INTEGER, BOOLEAN, FLOAT y similares. Casos más específicos como el de DATE (que necesita formatos diferentes a los tipos de datos mencionados) no se tratan aquí, pero con algunas modificaciones al código presentado pueden trabajarse.

[iv] El artículo de Lakshmanan nos sirvió, principalmente, para saber cómo codificar las sentencias SQL dentro del código de Python: fue una bases para saber cómo usar las comillas (“”) dentro del flujo del código. Por otra parte y en menor medida, sirvió para confirmar que algo de este tipo puede hacer en BQ

[v] La idea de cómo obtener los metadatos viene de Holowczak, 2022. De aquí únicamente retomamos la sentencia que contiene la variable *full_table_path*, pues esta es la que nos ayuda a extraer la información requerida. La dejamos con este nombre como referencia y, como se puede observar, el tratamiento del objeto obtenido es diferente al encontrado en la fuente, pues la naturaleza de este artículo es un tanto diferente. Por otro lado, la idea de trabajar con los metadatos surgió de la necesidad de trabajar con varias tablas, es decir, el hecho de pensar en que esta solución se aplicara a cualquier tabla y no sólo a una con una estructura determinada, nos llevó a concluir que es mejor trabajar con los metadatos, pues con ellos podemos obtener el número de columnas y el tipo de cada una, lo cual es algo que se puede obtener de cualquier tabla y que puede funcionar con la solución. La ambición de esta especie de automatización viene del hecho de que, en la práctica, las tablas que encontramos tanto en las bases de datos convencionales como en BQ siempre son variables en cuanto a su estructura.

[vi] Aquí es conveniente señalar que, en caso de requerirse, para determinar el número de renglones se deben realizar los siguientes pasos:

1. Determinar el número de campos: $c_1, c_2, c_3, \dots, c_m$.
2. Determinar el número de datos: $d_1, d_2, d_3, \dots, d_n$.
3. Dividir el número de datos entre el número de campos: $n/m = r$, donde r es el número de renglones.

Por ejemplo, si tenemos 4 campos y 12 datos, entonces tendremos un total de 3 renglones. Además, cabe aclarar que, dentro y fuera de este contexto, para realizar una inserción en SQL es importante que el número de datos sea un múltiplo del número de campos. De esta manera el gestor de bases de datos (o en este caso BQ) no notificará sobre un error de exceso o falta de datos.

[vii] Si intentamos codificar esta misma función en JavaScript (el otro lenguaje en el que se hicieron pruebas para entender el porqué de esta situación) podremos notar que no se obtiene el mismo error, por lo que no tienen que codificarse esta sección y la siguiente. Por ello, esto es más un requerimiento de Python que una necesidad de lo aquí descrito.

[viii] Llenamos con guiones las listas con el único fin de igualar los tamaños entre las mismas y porque los guiones no se parecen a ninguna de las cadenas de texto que se usarán más adelante. Así no tenderemos problemas con los procedimientos mostrados posteriormente.

[ix] Esto se podría haber hecho todo en una única sección realizando la igualación de cada lista directamente con diferencia, pero lo hicimos así porque parece más explícito para entender la naturaleza de este problema.

[x] Para realizar este algoritmo nos basamos más en un análisis del problema al que nos enfrentamos. Este se encuentra descrito en los arreglos de datos y código descritos a lo largo del artículo. Sin embargo, el llamarle *búsqueda por saltos* viene del hecho en que observamos varios algoritmos de búsqueda para entender si lo que hicimos entraba dentro de esta categoría. En general, nos basamos en Ponce, 2021, Wikipedia, 2022, clases universitarias de algoritmos y estructuras de datos, y análisis y diseño de algoritmos..

[xi] Esto posiblemente sea objeto de una demostración por inducción matemática, pero como eso sobrepasa los objetivos de este escrito, la dejaremos para otra ocasión.

[xii] Este procedimiento es el que otorga al algoritmo el nombre de *búsqueda por saltos*, pues tener en cuenta el número de campos involucrados en la inserción, nos indica el número de elementos que se tienen que "saltar" en cada caso para encontrar qué elementos le corresponden a los campos. Este descubrimiento se hizo después de escribir todas estas estructuras en papel y observarlas.

[xiii] Los pasos 5 y 6 son más evidentes en el código, por eso no los vamos a señalar.

[xiv] *contador* se usa para retomar el conteo en el número en que quedó cada vez que se pasa al segundo loop. La necesidad de este contador también recae en el hecho de que $j = 0$ cada vez que se entra al segundo loop.