DD360 README

June 17, 2023

1 Instalación

1. Clonar el repositorio con:

```
git clone https://github.com/Jazpy/DD360.git
```

2. Instalar paquetes necesarios con:

```
pip install -r requirements.txt
```

3. Para una "corrida" manual se puede ejecutar el script de bash simplemente con:

```
./manual.sh
```

   Esto creará archivos intermedios resultantes del scraper en `json/`, actualizará la base de datos en `db/meteo.db`, y escribirá reportes en `parquet/` y en `csv/`

   Alternativamente se puede agregar el directorio `src` a la variable `dags_folder` en `airflow.cfg`, y habilitar el DAG llamado `simple_dag` que aparecerá inmediatamente. Éste se ejecutará 1 vez cada hora.

2 Explicación de código

2.1 `scraper.py`

Esta clase se encarga de hacer las consultas a `meteored.mx` y extraer la información relevante. La función principal es `scrape()`:

```
def scrape(self):
    '''Consulta los URLs guardados y regresa la información relevante.

    Regresa:
        ret: lista de tuplas de la forma (código respuesta,
            (distancia, fecha y hora, temperatura, humedad)), un elemento
            por cada URL.
    '''
    # Obtener respuestas crudas
    responses = []
    for url in self.urls:
        try:
            # Hacer consulta y guardar código de respuesta.
            # TODO: Tal vez sería bueno hacer esto async.
            response = requests.get(url)
            code = response.status_code

            # Recupera texto solo si fue una consulta exitosa.
            text = response.text if code == 200 else None
        except Exception as e: # Errores de conexión por ejemplo.
            code, text = -1, None
        finally:
            responses.append((code, text))

    ret = []
    for code, text in responses:
        if code == -1 or not text:
            ret.append((code, (None, None, None, None)))
            continue

        # Representación jerárquica del texto.
        parsed = bs(text, 'html.parser')

        # Extraer las cadenas de los valores relevantes.
        dist = self.__match(str(parsed.find(id='dist_cant')))
        temp = self.__match(str(parsed.find(id='ult_dato_temp')))
        humd = self.__match(str(parsed.find(id='ult_dato_hum')))
        updt = self.__match(str(parsed.find(id='fecha_act_dato')))

        # Transformar datos a sus tipos correctos si se recuperaron
        # de manera correcta.
        temp = float(temp) if temp else None
        humd = float(humd) if humd else None

        if updt:
            # Agregar 'Z' al tiempo para indicar UTC y convertir a
            # un objeto datetime.
            updated_dt = date_parser.parse(updt + 'Z')
            # Transformar a timestamp de UNIX.
            updt = int(time.mktime(updated_dt.timetuple()))

        # Agregar a la salida
        ret.append((code, (dist, updt, temp, humd)))

    return ret
```

Básicamente, iteramos sobre cada URL de interés, estos URLs se le dan a la clase `MeteoScraper` cuando se crea un objeto. Para cada URL obtenemos una respuesta, o si por cualquier razón no hay una respuesta tomamos un valor default de -1 como código de respuesta. Sería bueno hacer esta parte del código asíncrona, ya que podría ser extremadamente tardada si tuviéramos más URLs.

Teniendo todas las respuestas, utilizamos bibliotecas como BeautifulSoup para parsear el HTML respuesta, y sobre este árbol parseado buscamos los spans que tienen la información relevante. Si buscáramos más datos en vez de solo 4, sería conveniente iterar sobre una lista de spans de interés que pueda ser configurada. Se utiliza una expresión regular muy sencilla y endeble para obtener el valor dentro del span, sería mejor reemplazarla por una llamada a BeautifulSoup que pudiera extraer el valor de cualquier span.

Si por cualquier razón no obtenemos un valor válido del span, asignamos un valor de `None` que en la base de datos se volverá un `NULL`. Para la hora de actualizado, la convertimos en un timestamp de `UNIX` ya que facilitará mucho todo el resto de la ejecución. Puede transformarse a una cadena fácil de leer cuando sea, pero eso sería hasta la etapa de interfaz con el usuario.

En un código mejor preparado, sería importante validar los URLs antes de hacer las consultas.

2.2 packer.py

El siguiente archivo es el más sencillo del repositorio. Lo único importante a mencionar es que crea un objeto de tipo `MeteoScraper`, y utiliza a éste para obtener los datos de `meteored.mx`. Posteriormente, los empaqueta como archivos `.json` para que luego sean consumidos por el manejador de la base de datos.

```python
def pack(self):
    '''Llama al scraper y guarda resultados como archivos JSON

    Arroja:
        EnvironmentError: no se pudo escribir al archivo JSON.
    '''
    # Para el ID de estos queries.
    ts = int(dt.utcnow().timestamp())

    # Hacer queries a meteored.
    responses = self.scraper.scrape()

    # Transformar a diccionarios.
    dics = []
    for i, (city, query, response) in enumerate(
        zip(self.cities, self.urls, responses)):
        response_dic = {
            'query':       query,
            'city':        city,
            'code':        response[0],
            'distance':    response[1][0],
            'update':      response[1][1],
            'temperature': response[1][2],
            'humidity':    response[1][3],
            'run':         ts,
            'id':          f'{ts}_{i}'
        }

        dics.append(response_dic)

    # Guardar diccionarios en un archivo JSON.
    try:
        with open(f'json/{ts}.json', 'w') as out_f:
            out_f.write(json.dumps(dics))
    except EnvironmentError as e:
        print('Error escribiendo JSON!')
        raise e
```

Esencialmente, el propósito de esta clase es que el manejador de la base de datos no tenga que llamar al scraper directamente, y que el scraper pueda mantenerse agnóstico al formato `.json`. Esta distinción también sería útil si en un futuro se quisiera tener varios tipos de scrapers trabajando independientemente, con la clase `Packer` funcionando como un tipo de aglomerador de datos que pueda procesar diferentes formatos.

2.3 db_manager.py

Finalmente la clase `db_manager.py` se encarga de escribir y leer a la base de datos. El primer método es `__add_response()`. Este método es llamado por `commit_latest()`, que simplemente parsea un archivo `.json` y envía los datos directamente. Dentro de `__add_response()`, primero se verifica si la ciudad que se está reportando ya existe en la base de datos, de no ser así, es añadida. Luego se agrega el ID de la consulta junto con

el código de respuesta a la tabla de códigos de respuesta, y finalmente se agregan los valores meteorológicos a la tabla de reportes, después de una sanitización muy básica.

```python
def __add_response(self, city, dist, temp, humd, ts, q_id, code):
    '''Agrega los datos asociados a una consulta a la BD

    Argumentos:
        city (str): ciudad objeto.
        dist (str): distancia de estación.
        temp (float): temperatura.
        humd (float): humedad.
        ts (int): tiempo de actualización.
        q_id (str): id único de la consulta.
        code (int): código de respuesta.

    Arroja:
        ConnectionError: no se pudo conectar a la BD.
        SQLite3.Error: error al insertas filas.
    '''
    # Conexión a SQLite
    try:
        conn = sqlite3.connect(self.db_path)
    except Exception as e:
        print('Error conectándose a la BD!')
        raise ConnectionError

    # Agregar ciudad si es necesario
    cursor = conn.execute('select id from cities where '
        f'name="{city}" limit 1')
    row = cursor.fetchone()
    if not row:
        conn.execute(f'insert into cities (name) values ("{city}")')
        cursor = conn.execute('select id from cities where '
            f'name="{city}" limit 1')
        row = cursor.fetchone()
    city_id = row[0]

    # Agregar a tabla de códigos
    conn.execute('insert into codes (id, code) values '
        f'("{q_id}", {code})')

    # TODO: agregar sanitizador que haga esto de manera más general.
    # Agregar a tabla principal de consultas
    dist = f'"{dist}"' if dist else 'NULL'
    temp = temp if temp else 'NULL'
    humd = humd if humd else 'NULL'
    ts   = ts   if ts   else 'NULL'
    conn.execute('insert into reports (distance, temperature, '
        'humidity, updated, code_id, city_id) values '
        f'({dist}, {temp}, {humd}, {ts}, "{q_id}", {city_id})')

    conn.commit()
    conn.close()
```

Cambios importantes que me gustaría hacer serían una sanitización mucho más robusta de los datos, ya que en este momento se asume que no habrá ninguna entrada maligna. También me gustaría que la conexión con la base de datos pudiera mantenerse abierta para cada reporte y así evitar el overhead de abrirla y cerrarla, pero eso

también implicaría un monitoreo más sofisticado del estado de la conexión y el estatus de las diferentes transacciones. En este momento si hay calquier problema de conexión, simplemente no se hace ningún cambio a la base de datos y se procede con la siguiente ciudad de la corrida.

En cuanto a la leída de la base de datos, está la función `__get_checkpoint()`. Esta función obtiene temperaturas o humedades mínimas, máximas, promedio, y más recientes para cada ciudad que exista en la base de datos en ese momento. Todo se arregla a un diccionario que luego es convertido a un `dataframe` de `pandas` para fácilmente escribir tanto los archivos `parquet` como `.csv`.

```
def __get_checkpoint(self):
    '''Recupera los datos para un checkpoint de parquet en un diccionario

    Regresa:
        ret (dictionary): diccionario con reporte promedio y más reciente
        para cada ciudad, formato para pandas.

    Arroja:
        ConnectionError: no se pudo conectar a la BD.
        SQLite3.Error: error al buscar filas.
    '''
    # Conexión a SQLite
    try:
        conn = sqlite3.connect(self.db_path)
    except Exception as e:
        print('Error conectándose a la BD!')
        raise ConnectionError

    ret = defaultdict(list)

    # Iteramos sobre cada ciudad, construyendo un reporte a la vez
    cursor = conn.execute('select * from cities')
    rows   = cursor.fetchall()
    for row in rows:
        c_id = row[0]
        ret['ciudad'].append(row[1])

        # Temperatura mínima, máxima, y promedio
        cursor = conn.execute(f'select max(temperature) from reports where
        ret['max_temp'].append(cursor.fetchone()[0])
        cursor = conn.execute(f'select min(temperature) from reports where
        ret['min_temp'].append(cursor.fetchone()[0])
        cursor = conn.execute(f'select avg(temperature) from reports where
        ret['avg_temp'].append(cursor.fetchone()[0])

        # Humedad mínima, máxima, y promedio
        cursor = conn.execute(f'select max(humidity) from reports where ci
        ret['max_humid'].append(cursor.fetchone()[0])
        cursor = conn.execute(f'select min(humidity) from reports where ci
        ret['min_humid'].append(cursor.fetchone()[0])
        cursor = conn.execute(f'select avg(humidity) from reports where ci
        ret['avg_humid'].append(cursor.fetchone()[0])

        # Valores más recientes
        cursor = conn.execute(f'select max(updated) from reports where cit
        ts     = cursor.fetchone()[0]

        if ts:
            cursor = conn.execute(f'select temperature, humidity from repo
            row    = cursor.fetchone()
            curr_t, curr_h = row[0], row[1]
        else:
            curr_t, curr_h = None, None

        ret['curr_temp'].append(curr_t)
        ret['curr_humid'].append(curr_h)
        ret['updated'].append(ts)

    conn.close()
    return ret
```

Cambios importantes que podrían hacerse incluyen tener menos valores hardcodeados, por ejemplo podría haber una función que tome como argumento la columna, la función (min, max, avg), y la ciudad para obtener esos valores de manera genérica. También sería buena idea almacenar el promedio, mínimo, y máximo de las columnas relevantes para poder actualizar estos valores en tiempo constante en vez de lineal sobre la cantidad de hileras.

2.4 schema

El schema de la base de datos es muy sencillo. Tenemos 3 tablas:

```
CREATE TABLE cities(
    id integer primary key autoincrement not null unique,
    name text not null unique
);

CREATE TABLE codes(
    id text primary key not null unique,
    code int not nul
);

CREATE TABLE reports(
    id integer primary key autoincrement not null unique,
    distance text,
    temperature real,
    humidity real,
    updated integer,
    code_id text not null unique,
    city_id integer not null,
    foreign key(code_id) references codes(id),
    foreign key(city_id) references cities(id
);
```

La primera tabla solo tiene nombres de ciudades y IDs únicos, como piden los requerimientos de la prueba.

La segunda tabla tiene un identificador único para toda consulta que se hace a través del scraper (timestamp + índice en la corrida) y el código de respuesta. Elegí este identificador porque usar solamente el timestamp de la corrida o de la actualización no garantiza que sea único, y esto da flexibilidad para agregar más distintivos a la cadena que representen por ejemplo el scraper que hizo la consulta, o cuál evento en la nube de muchos fue el gatillo que resultó en la consulta específica.

Finalmente, la tercera tabla guarda los datos relevantes, y llaves foráneas tanto a la tabla de ciudades como a la tabla de códigos de respuesta. Guardar la fecha de

actualización como un timestamp de UNIX hace que sea muy fácil determinar rangos de tiempo para consultas a la base de datos. Dado que los requerimientos piden que la tabla de códigos de respuesta sea distinta a la de los datos reportados, esta configuración no tiene ninguna duplicación de datos innecesaria.