PRE-EMPLOYMENT TEST

1. Problem 1: Architecture

In situations where the volume of data gathering surpasses the system's processing capacity, a strategic revision of the architecture is necessary. Implementing **batch processing** becomes instrumental in efficiently managing extensive datasets by breaking them into manageable chunks, facilitating parallel processing. Examples of batch processing tools include Apache Hadoop, Apache Spark, and Apache Flink. These frameworks enable the distributed storage and processing of large datasets with features like MapReduce and in-memory processing.

The adoption of **data sharding** is pivotal to distribute data across numerous processing nodes, with each node handling a specific subset, thereby ensuring horizontal scalability. Database choices supporting data sharding include MongoDB for NoSQL data distribution, MySQL configured for sharding, and CockroachDB offering automatic sharding and strong consistency.

Embracing **Dynamic Scaling** leverages the auto-scaling capabilities offered by cloud providers, allowing the system to automatically adjust its capacity in response to varying data influx. Tools like Amazon EC2 Auto Scaling, Kubernetes, and Google Kubernetes Engine (GKE) provide dynamic scaling features. For instance, Amazon EC2 Auto Scaling adjusts the number of EC2 instances based on user-defined policies, while Kubernetes and GKE automatically scale containerized applications by adjusting the number of running containers.

To address the challenge of parallelizing the slow API, the Python library **asyncio** emerges as a robust solution. Its asynchronous programming paradigm enables concurrent execution of tasks, effectively utilizing system resources and mitigating the impact of API latency.

2. Problem 2: Data management

The provided Python script, ***data_management_ferran.py*** utilizes the pandas and numpy libraries to transform electricity consumption data from an Excel file, ***data_2019.xls,*** into a time series with a regular daily frequency. The script's purpose is to process billing information spanning a year, identifying and addressing gaps in the data, and ultimately saving the transformed time series as a CSV file.

The initial step involves reading the data from the Excel file using pandas. The script focuses on the first three columns of the dataset, ensuring that only relevant information is considered for further processing. Any rows containing NaN values are dropped to enhance the consistency of the dataset.

The date columns, *d_ini* and *d_end,* are then converted to the datetime format, facilitating subsequent date-related operations.

To create the expanded time series, the script initializes an empty DataFrame named ***expanded_df*** with columns *date* and *kwh* to store the daily consumption data. Another DataFrame, ***final_csv***, is constructed, containing all dates in daily frequency within a specified date range, *2019-01-01* and the largest number in the *d_end* column of the initial Excel.

The script iterates over the rows of the original DataFrame, generating date ranges, from the *d_init* and *d_end* columns, and corresponding consumption values to expand the time series. The expanded time series DataFrame is merged with the final DataFrame based on the *date* column, if a gap is detected a NaN value is added, consolidating the transformed data.

3. Problem 3: Linked data

I have used the following sparql queries to solve the proposed problems:

List of all buildings in the file:

```
def get_all_buildings_query():
    return """
    SELECT ?building ?buildingName
    WHERE {
        ?building rdf:type s4blg:Building ;
        foaf:name ?buildingName.
    }
    """
```

List of all buildings of a type:

```
def get_all_type_buildings_query(type):
    return f"""
    SELECT ?building ?buildingName ?buildingType
    WHERE {{
        ?building rdf:type s4blg:Building ;
            foaf:name ?buildingName ;
            ex:tipus ?buildingType.
    FILTER (?buildingType = "{type}")
    }}
    """
```

List of all devices linked to a building:

```
def get_devices_of_buildings_query(building):
    return f"""
    SELECT ?device ?deviceType
        WHERE {{
        {building} s4blg:contains ?device.
        ?device rdf:type s4blg:Device ;
                ex:tipus ?deviceType.
        }}
    """
```

The response obtained when the python script, query.py was executed was as follows:

```
(base) ferran@ferran:~/Escritorio/cimne/preEmploymentTest/linked_data$ python query.py
ALL BUILDING URI
(rdflib.term.URIRef('http://example.org/b1'), rdflib.term.Literal('Edifici A'))
(rdflib.term.URIRef('http://example.org/b2'), rdflib.term.Literal('Edifici B'))
(rdflib.term.URIRef('http://example.org/b3'), rdflib.term.Literal('Edifici C'))
(rdflib.term.URIRef('http://example.org/b4'), rdflib.term.Literal('Edifici D'))
ALL OFFICE BUILDINGS
(rdflib.term.URIRef('http://example.org/b1'), rdflib.term.Literal('Edifici A'), rdflib.term.Literal('Oficina'))
DEVICES AND TYPE OF BUILDING B2
(rdflib.term.URIRef('http://example.org/d7'), rdflib.term.Literal('TEMPREATURA'))
(rdflib.term.URIRef('http://example.org/d8'), rdflib.term.Literal('ELECTRIC'))
```

4. Problem 4: REST api

The Flask REST API, implemented in the Rest_API.py script, is designed using the Flask framework in Python. This API interacts with a MySQL database through the mysql-connector library, defining specific endpoints for tasks such as initializing the database, creating tables, and retrieving time-series data. The API runs on port 5000 and provides a structured interface for data management.

For the database initialization process, the create_db function connects to MySQL, reads a password from a file, and creates the daily_time_series_db if it doesn't already exist. The create_table function is responsible for creating the daily_time_series table within the database, specifying columns for ID, date, and kWh.

To insert data into the database, the insert_data function reads time-series data from the time_series_daily.csv file. It then inserts this data into the daily_time_series table using the REPLACE INTO SQL statement, ensuring data integrity.

The Flask API defines two main endpoints. The '/' endpoint triggers the initialization of the database, creation of the table, and insertion of data. The '/get_data' endpoint allows users to retrieve time-series data within a specified date range by sending GET requests.

Docker is employed to containerize the application. The Dockerfile specifies the steps to create an image for the Flask API, including setting the working directory, installing dependencies, and exposing port 5000. The docker-compose.yaml file orchestrates the deployment, defining services for both the API and a MySQL

database. The API service depends on the database service, ensuring proper initialization order.

After executing the *docker-compose up* command the Docker containers are initiated. Subsequently, navigating to http://localhost:5000 in a web browser triggers the creation of the database, table, and the insertion of data into the specified table. To retrieve data for a specific date range, a GET request to http://localhost:5000/get_data?start_date=2019-01-01&end_date=2019-01-10 provides the relevant information.