How to run the file:

* Unzip the jar file: case-study-server.jar
* From terminal / CLI, launch the case study server on port 8081: java -jar case-study-server 8081
* Make sure your machine has python3.5x.
* pip install pipreqs
* pip install -r requirements.txt

Finally, run python uptake.py

How it works:

Connect to the appropriate API endpoint. For this project, our api end point is http://localhost:8081/.

We also have three query gateways:

\* /sites

\* /turbines

\* /signals

Using helper functions, the app makes request to each of the gateways and grabs all incoming json data.

It performs basic cleaning to extract incoming json into a pandas dataframe and then saves all the data into a temporary location raw\_data/ at the current working directory.

The output from the location raw\_data can is to be used next to ingest into a Uptake’s Database Platform.

Using a dummy gmail account, I wrote the connection to Big Query and created the following tables (including a denormalized table for fast lookups). Unfortunately due to Google’s oauth protection. I am including the screenshot layout of the tables at the end of this documentation.

Additionally, I have included a set of create table statements.

**Architecture:**

I have designed layout, attached in PDF.

Since the case study mentions using any technology of your choice, I am basing my design of a Google Big Query.

Processing:

Connection is established from a compute engine (Unix based cluster or cloud cluster such as Google Compute Engine) using a python script. Data is downloaded temporarily into an internal storage.

Transformation and cleaning is done using pandas / pyspark can also be used. Data is now being transformed and ready to be pushed onto Uptakes DB platforms.

Logs and Checks (not explicitly mentioned in the design)

1. A Data Profile can be written to perform some health checks:
   1. Stage everything as string datatype or transform and upload. In the either events, the some common checks are:
      1. Schema mismatch,
      2. Removing any null values and data type mismatch.
      3. Rollback the latest push in the event of a failure.
2. Upload log script to keep a running track of the jobs.
   1. Tables update
   2. Number of new rows added / removed.
   3. Time of upload, script name, cluster name etc. to identify the source of log.
   4. Log notification in case of failure. This can be setup using a bash.

Raw Database:

Since given dataset is not so massive, I have assumed that scalability and fast access would be a concern at some point.

I have chosen Big Query as my primary database since it supports operations on Petabytes of data. Raw Order of data is maintained in Big Query. For analytical purpose, the dataset can also be denormalized to fast access and lookups.

Processing – Aggregation and Grouping

Raw order of data can then be processed right from Big Query, using SQL script that can be automated or to an external process using .py jobs utilizing pandas and pyspark to aggregated and prepare final reporting tables.

This also helps avoid continuous writes and reads to the RAW order table, as this causes latency. Reporting tables are the available in a Big Query warehouse.

Reporting Tables

After processing, aggregated datasets are pushed onto new reporting tables. Views can also be utilized for reporting here. These reporting tables are much smaller that the raw dataset and are focused on specific data models.

In case of custom web interfaces, Mongo DB or Postgres can be used as the provide high ACID features.

Front End and Visualization – Reporting

This can be either a web interface or a reporting tools that’s powered by the data coming from the Reporting tables.

Most reporting tools Tableau or Power BI utilize caching of the dataset. In case of a custom web interface D3 or other Charts.js API can be used to design these dashboards.

**Entity Relationship:**

Sites and Turbine tables have a 1:N relationship. Site\_id is the Foreign Key in Turbine table.

Sites

*Siteid (primary key)*

Turbine

*Turbine\_id (primary key)*

*Site\_id ( FK)*

Turbine and Signals tables again have a 1:N relationship. TurbineId is the foreign key.

Turbine

*Turbine\_id (primary key)*

*Site\_id ( FK)*

Signals

TurbineId (Foreign key)

S\_no (custom Primary key)

SiteTurbine table is built to establish the many to many relationship between Sites and Signals. Sites and Signal tables have a N:M relationship, or a many to many relationship. We can create an intermediate table to establish this connection.

SiteTurbine

*Siteid*

*Turbineid*

Connection to Big Query

Using a dummy gmail account, I wrote the connection to Big Query and created the following tables (including a denormalized table for fast lookups). Unfortunately due to Google’s oauth protection. I am including the screenshot layout of the tables below.

*Big Query Table Schema overview.*

*A screenshot of a cell phone

Description automatically generated*

*A screenshot of a cell phone

Description automatically generated*A screenshot of a cell phone

Description automatically generated

Denormalized Table for Analytics

*A screenshot of a cell phone

Description automatically generated*