The Data Driller - A Property Pipeline Project

1. Introduction

Project Statement

This project involves developing a data pipeline to ingest and consolidate time-distributed HDB resale flat datasets. The team has implemented an end-to-end data integration workflow using Python by extracting data from the data.gov.sg API, transforms it to a consistent schema, and loads it into a PostgreSQL database, serving as a single source of truth for downstream analytics.

Project Goals

The primary objectives of this project are to:

- Construct a resilient data crawler
- Design and implement an optimised database schema
- Document a clear user guide for seamless operation
- Conduct preliminary data analysis to demonstrate utility

Project Plan & Timeline

The project is organised into three distinct phases, mirroring the structure of this report:

Phase 1: The Blueprint - Comprehensive research and architectural design.

Phase 2: The Build - Development of the data crawler and database schema.

Phase 3: The Payoff - Data loading, preliminary analysis, and final report compilation.

2. Research & Analysis

Data Source Research

Our primary data source is the data.gov.sg API. It uses pagination, serving data in chunks. To handle this, our crawler first determines the total number of records, then concurrently fetches all pages using asynchronous aiohttp requests. This parallel fetching minimises data retrieval time and ensures efficient handling of large datasets.

Furthermore, a ful dataset covering the period from 1990 to the present is not provided as a single file by the API. We have identified five distinct CSV datasets which, when combined, provide a comprehensive view of the entire period up till 2025. Our method involves fetching each of these five separate datasets and consolidating them into a single, cohesive DataFrame for analysis.

Technology Stack Review

We chose Python for its extensive library ecosystem and readability, making it the ideal language for data projects. For our database, we selected PostgreSQL for its reliability and performance with structured data.

For the python libraries used, we went with:

- ***aiohttp*** for asynchronous requests, allowing us to fetch data much faster.

- ***nest_asyncio*** to allow the asynchronous code to run in nested environments like Jupyter Notebook and Visual Studio.
- ***numpy*** for efficient numerical operations and data handling within the DataFrame.
- ***pandas*** to manage data transformation by using the ***pandas.DataFrame*** object for operations such as data formatting and normalisation.
- ***math*** for mathematical functions, specifically using the math.ceil function to accurately determine the total number of pages to be fetched from the API.
- ***SQLAlchemy*** to manage the database connection and make loading data feel like a breeze.

System Requirements & Use Case
The system is built on a few core requirements.

- **Robust Error Handling**: The system ensures robustness by checking the API's success key in the JSON response with try-except blocks to gracefully handle network errors and API failures.
- **Efficient Data Fetching**: We use Python's asyncio and aiohttp to perform non-blocking I/O operations. This allows the crawler to make multiple simultaneous API requests, reducing the overall time required for data extraction compared to a sequential approach.
- **Structured Data Output**: The raw JSON data from the API is transformed into a tabular structure using a pandas.DataFrame, ensuring all records conform to a consistent schema.

3. Database Schema & Structure:

Conceptual Design
For this project, we have chosen the star schema Entity-Relationship Model (ERM). This model, a form of denormalisation, is ideal for our analytical goals as it is designed to prioritize query performance by minimising the number of joins required for SQL queries.

This structure provides a clear distinction between the central fact table (resale_transactions)—which holds the core metrics like price and floor_area_sqm—and its related dimension tables (location, flat_details, and time). This directly addresses our primary use case: enabling rapid, high-level analysis.

Logical Design
Our logical design for the star schema is as follows:

Fact Table: resale_transactions

- flat_id (INTEGER, FOREIGN KEY): Links to the flat dimension table.
- location_id (INTEGER, FOREIGN KEY): Links to the location dimension table.
- storey_id (INTEGER, FOREIGN KEY): Links to the storey dimension table.
- time_id (INTEGER, FOREIGN KEY): Links to the time dimension table.
- resale_price (NUMERIC): The final price of the flat.
- price_per_sqm (NUMERIC): The calculated price per square meter.
- remaining_lease_years (NUMERIC): The calculated remaining lease in years.

Dimension Table: flat_details

- flat_id (INTEGER, PRIMARY KEY): A unique identifier for each flat type-model combination.
- flat_type (VARCHAR): The type of flat (e.g., '3 ROOM').
- flat_model (VARCHAR): The flat model (e.g., 'New Generation').
- floor_area_sqm (NUMERIC): The size of the flat in square meters.
- lease_commence_date (INTEGER): The year the lease commenced.

Dimension Table: location

- location_id (INTEGER, PRIMARY KEY): A unique identifier for each town-street combination.
- town (VARCHAR): The HDB town.
- street_name (VARCHAR): The street where the flat is located.
- block (VARCHAR): The block number.

Dimension Table: time

- time_id (INTEGER, PRIMARY KEY): A unique identifier for each month-year combination.
- year (INTEGER): The year of the transaction.
- month (INTEGER): The month of the transaction.

Dimension Table: storey

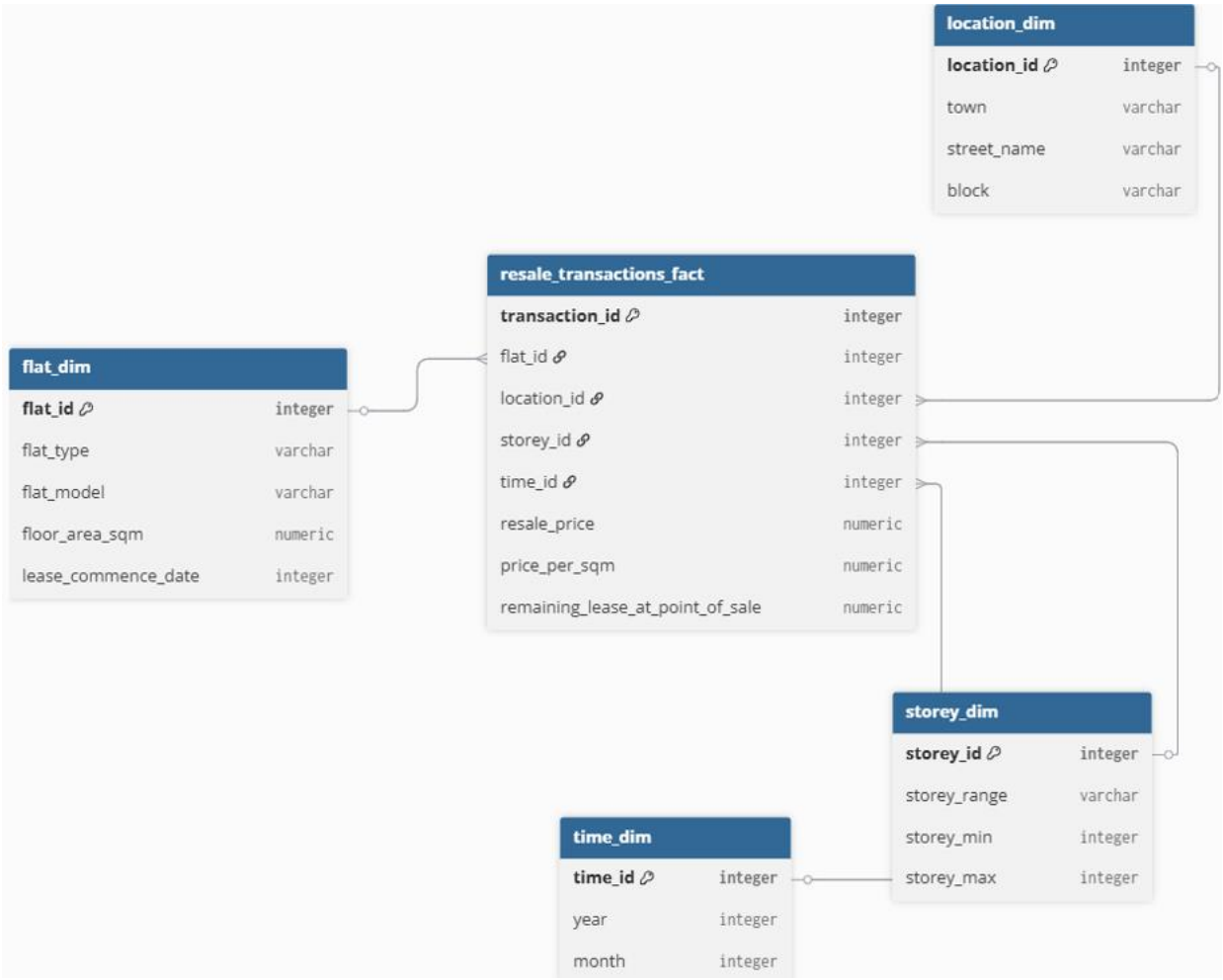
- storey_id (INTEGER, PRIMARY KEY): A unique identifier for each storey range.
- storey_range (VARCHAR): The range of floors for the flat (e.g., '10 to 12').
- storey_min (INTEGER): The minimum storey number.
- storey_max (INTEGER): The maximum storey number.

Justification

We chose a denormalised star schema over a normalised one because it is ideal for the read-intensive, analytical goals, prioritising query performance by minimising the number of joins.

Key advantages are:

- Optimised performance: By reducing joins and including derived metrics directly in the fact table, we achieve rapid data aggregation and faster analysis.
- Data integrity: The schema has been refined to better isolate unique dimensional attributes, improving data integrity and reducing redundancy.
- Simplified queries: The clear separation of fact and dimension tables makes it easier to write simple, efficient queries for business intelligence.



4. ETL Process

The process of getting the data from the source to our database follows a clear, three-stage flow:

- **Extract:** The Python script uses aiohttp to send asynchronous HTTP requests to the data.gov.sg API. It programmatically navigates the API's pagination, retrieving all historical records, which are returned in a JSON format. The data is stored in a temporary structure for the next stage.

- **Transform:** Once extracted, the data undergoes a series of transformations. We use the pandas and numpy libraries to clean the data, handle any missing values, and calculate new metrics not provided by the API, such as price_per_sqm and remaining_lease_years. This stage also standardises column names and data types to ensure they match our defined schema.

- **Load:** The transformed data is structured into a star schema before loading into the PostgreSQL database. This process is handled by a Python script using the SQLAlchemy library, which provides a simple and reliable way to interact with the database.

    - The loading process first generates four dimension tables (flat_dim, location_dim, storey_dim, and time_dim) from the cleaned DataFrame, assigning a unique, synthetic primary key to each.

- The resale_transactions fact table is then built by merging the data with these new keys.
- Finally, all five tables are loaded into PostgreSQL using pandas.to_sql with SQLAlchemy, a process that ensures referential integrity and maintains the star schema.

Essentially, we are taking the raw, unprocessed data and putting it through a digital assembly line so it comes out perfectly ready for analysis.

5. User Guide for the Crawler

Prerequisites To run the data crawler successfully, you need the following software and libraries installed on your system:

- **Python 3.8+:** The core programming language for the script.
- **PostgreSQL:** The database system for long-term data storage.
- **Python Libraries:** You can install all necessary libraries using pip.

Setup Before running the script, you must configure your environment and credentials.

1. **Database Configuration:** Update the database connection string in the configuration file (config.ini or similar) to match your PostgreSQL setup, including the username, password, host, and database name.
2. **API Key:** The data.gov.sg API does not require an API key for public datasets, so no key setup is necessary.
3. **File Paths:** Ensure the script has the necessary permissions to create and write to the local CSV file path specified in the configuration.

Running in a Jupyter Notebook If you prefer an interactive environment, you can run the code within a Jupyter Notebook.

1. **Install Jupyter:** If you don't have it, install it with pip install jupyterlab.
2. **Launch Jupyter:** Open a terminal in your project directory and run jupyter lab.
3. **Create a New Notebook:** A new tab will open in your browser. Create a new Python 3 notebook.
4. **Copy and Paste:** Copy the crawler code from your Python script and paste it into a code cell in the notebook.
5. **Run Cells:** Execute the cells sequentially. You can use this to run different parts of the ETL process in a step-by-step manner, inspecting the DataFrame after the transformation stage to verify data quality before the final load.

6. Results & Analysis

Please refer to the accompanying presentation for a more detailed explanation.