Mohammad A. Rehman – Sales Data Solution

This is a guide outlining the tooling requirements for the sales data solution, the set-up process for the data pipelines, the SQL scripts required for the database objects and data aggregations, and the PowerBI visualisations to illustrate analytical metrics pertaining to the sales, user, and geographical data.

1) Software and services used within the sales data solution

- Azure Storage Account – Data Lake Gen 2
- Azure Key Vault
- Azure Data Factory (an ARM template is provided to terraform the ADF instance)
- PostgreSQL
- PowerBI

2) Setting up the data pipeline

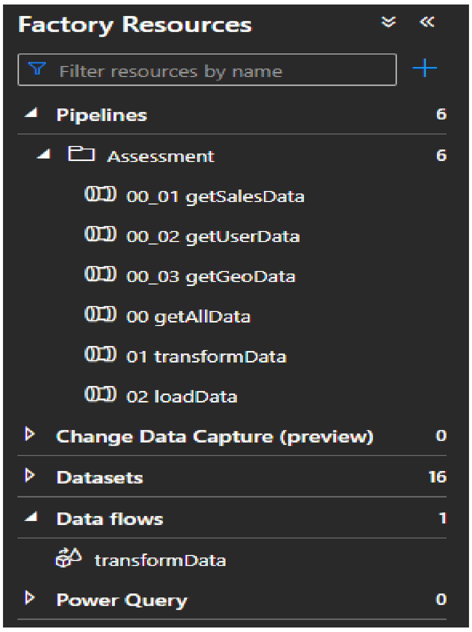
An azure storage account is initially created to store the source sales dataset, as well as any objects (csv and parquet) files that will be generated during the data sourcing and transformation processes.

Name	Last modified	Anonymous access level	Lease state
$logs	3/25/2024, 8:28:18 AM	Private	Available
raw-lake	3/26/2024, 8:50:13 AM	Private	Available
sales-data	3/25/2024, 8:29:00 AM	Private	Available
silver-lake	3/26/2024, 1:33:38 PM	Private	Available

- raw-lake contains a parquet copy of all files (geo_data, sales_data, and user_data) that are received from the various sources provided, for historization.
- sales-data contains the original sales_data.csv provided.
- silver-lake contains a parquet copy of all cleaned and transformed datasets generated using the pipelines (these will populate the following tables in Postgres: sales_per_user and sales_with_geo_info).

An azure key vault is created to store the weather API appid as a secret.

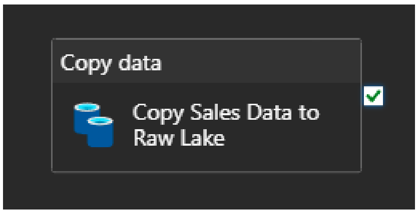
Name	Type	Status	Expiration date
weatherAPI-appid		✓ Enabled	



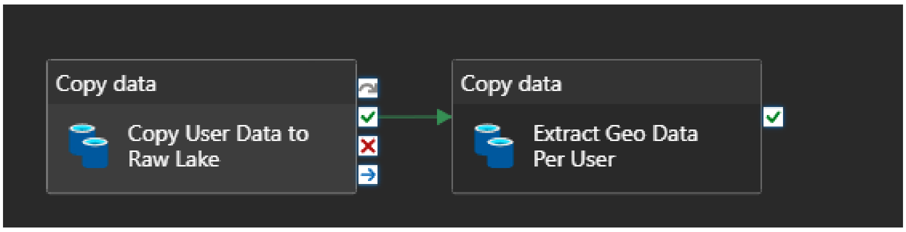
An Azure Data Factory v2 is created to build the pipelines required for data acquisition (user and weather), data transformation, loading into Postgres, and triggering the pipelines.

The pipelines within ADF are set up as follows:

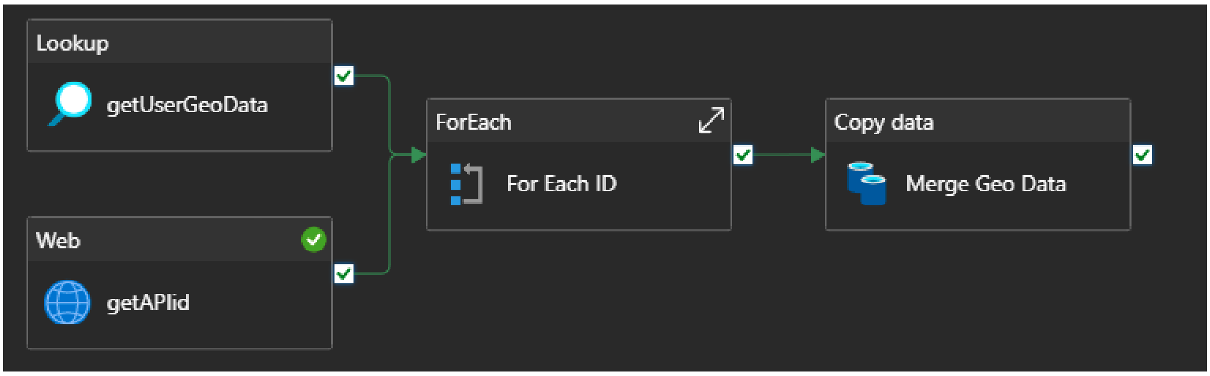
1) Get each of the datasets from their various sources.
   00_01 getSalesData:



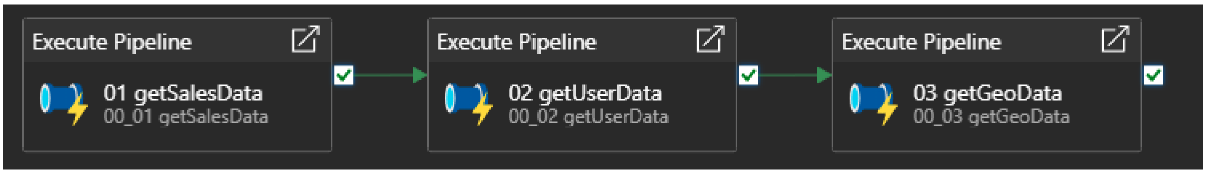
00_02 getUserData:



00_03 getGeoData:



All these pipelines can be executed sequentially, in one run by chaining their respective executions in one pipeline, as shown in 00 getAllData:



As user and weather are in .json format, each nested collection needs to be flattened; this is done within the Copy data activity's mapping specification:

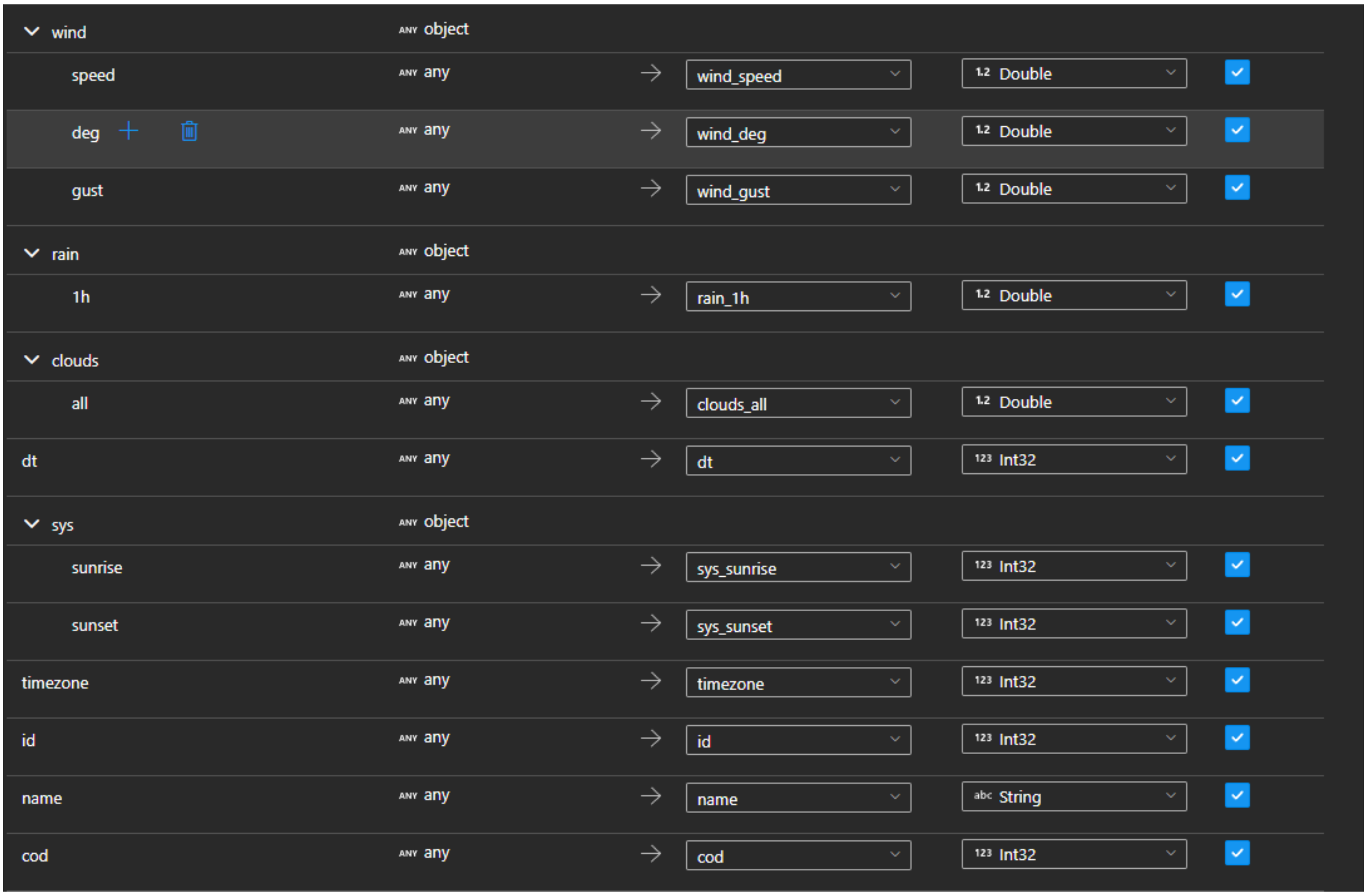
User mapping:

Name	Type	Collection reference	Column name	Type	Include
id	ANY any	→	id	123 Int32	✓
name	ANY any	→	name	abc String	✓
username	ANY any	→	username	abc String	✓
email	ANY any	→	email	abc String	✓
address	ANY object				
street	ANY any	→	street	abc String	✓
suite	ANY any	→	suite	abc String	✓

city	any	→	city	String	✓
zipcode	any	→	zipcode	String	✓
geo	object				
lat	any	→	lat	Decimal	✓
lng	any	→	lng	Decimal	✓
phone	any	→	phone	String	✓
website	any	→	website	String	✓
company	object				
name	any	→	company_name	String	✓
catchPhrase	any	→	catchPhrase	String	✓
bs	any	→	bs	String	✓

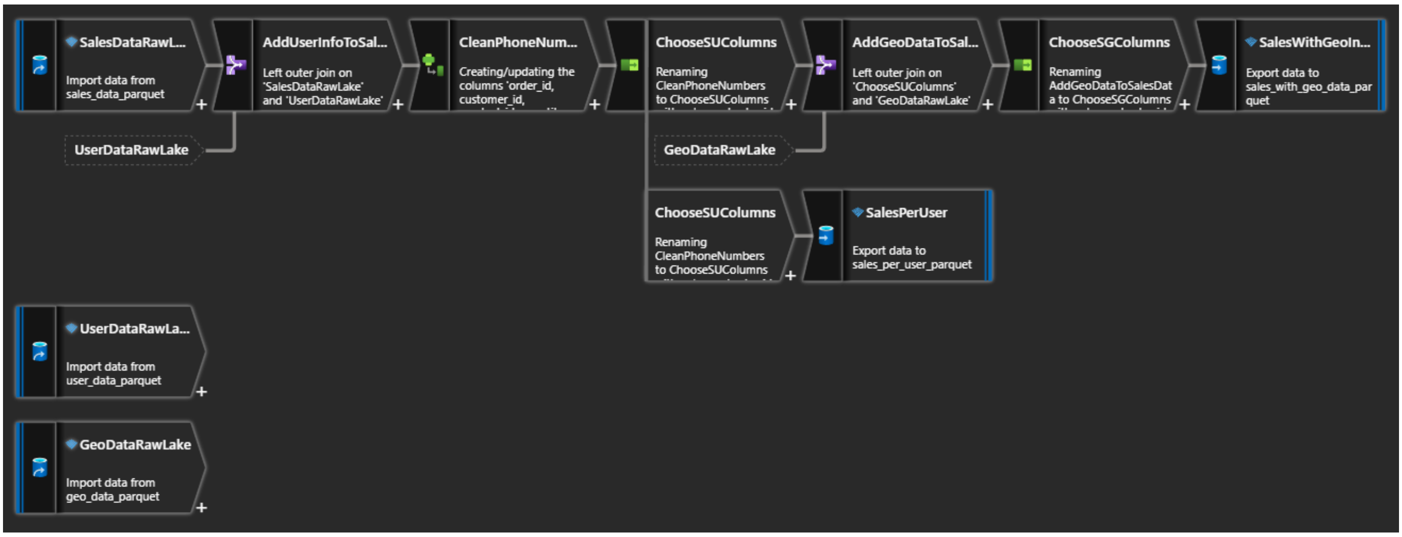
WeatherAPI mapping:

Name	Type	Collection reference	Column name	Type	Include
coord	object				
lon	any	→	lon	Decimal	✓
lat	any	→	lat	Decimal	✓
$['weather'][0]['id']	any	→	weather_id	Int32	✓
$['weather'][0]['main']	any	→	weather_main	String	✓
$['weather'][0]['description']	any	→	weather_description	String	✓
$['weather'][0]['icon']	any	→	weather_icon	String	✓
base	any	→	base	String	✓
main	object				
temp	any	→	temp	Double	✓
feels_like	any	→	feels_like	Double	✓
temp_min	any	→	temp_min	Double	✓
temp_max	any	→	temp_max	Double	✓
pressure	any	→	pressure	Double	✓
humidity	any	→	humidity	Double	✓
sea_level	any	→	sea_level	Double	✓
grnd_level	any	→	grnd_level	Double	✓
visibility	any	→	visibility	Double	✓



Executing 00 getAllData runs each of the 00_ pipelines to retrieve the datasets from their respective sources and stores them in the raw-lake folder. This ensures that we have a copy of the source data, along with metadata around when the data was retrieved from its source.

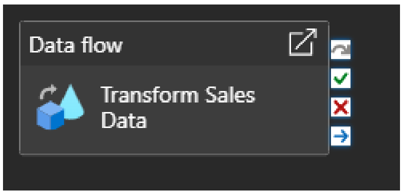
Next, the data can be cleaned (in the case of phone numbers which have different formats and some with extensions; a regexReplace() call is made to ensure we only have [^0-9] characters, after using split on 'x' to drop the extension portions where they exist), and transformed into a schema that is fit for the required aggregations and analytics required. A dataflow, transformData, is created to achieve this.



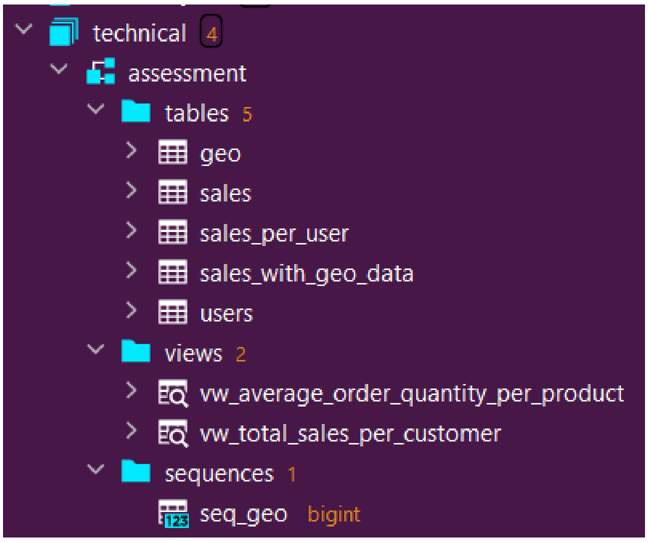
To preserve the sales records, left joins are done on both user and weather (geo) datasets, where sales' 'customer_id' field joins to user on user's 'id' field, and sales' 'lat' and 'lon' fields join to geo on geo's 'lat' and 'lon' fields.

The sinks, SalesPerUser and SalesWithGeoInfo, each write the transformed datasets, in parquet file format, to the silver-lake, which is used to store a copy of transformed datasets.

To trigger this dataflow, it is encapsulated in the 01 transformData pipeline, which executes the dataflow as illustrated in the transformation logic above. 01 transformData:



Having acquired all the source data and transforming it into a schema viable for down stream analytics, the data can be loaded from the raw-lake and silver-lake into Postgres once the required objects have been created in Postgres. Within Postgres, a database called ‘technical’ is created, with a schema called ‘assessment’. This database contains the structured tables, views, and sequences required to perform aggregation and analytics:



Of interest to us, are two tables, namely: sales_per_user and sales_with_geo_data, as they are the basis of all analytics and visualisation that will happen in due course. Below is the DDL for these two tables, illustrating the required fields and their data types – note that there is no primary key set as ‘order_id’ is not a unique field:

sales_per_user:

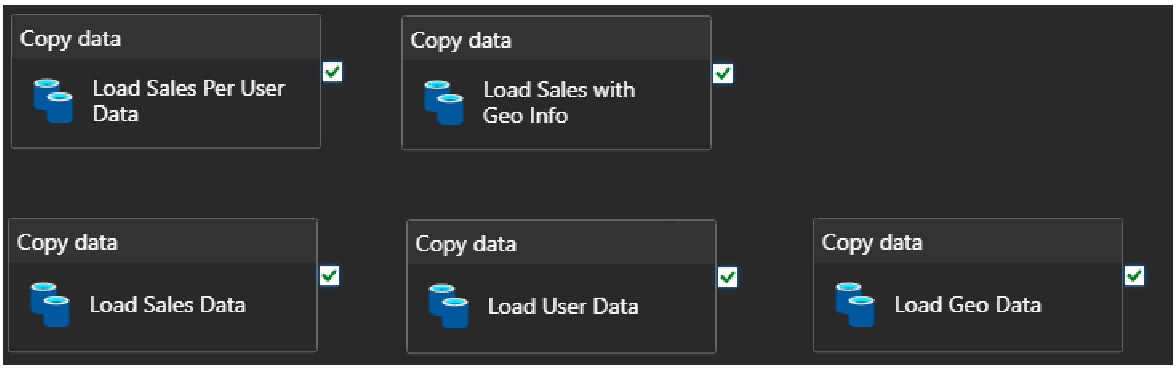
```
create table if not exists assessment.sales_per_user
(
    order_id     integer not null,
    customer_id  integer,
    product_id   integer,
    quantity     integer,
    price        double precision,
    order_date   date,
    name         varchar(256),
    username     varchar(256),
    email        varchar(256),
    street       varchar(256),
    suite        varchar(256),
    city         varchar(256),
    zipcode      varchar(10),
    latitute     numeric(10, 8),
    longitude    numeric(11, 8),
    phone        varchar(20),
    website      varchar(256),
    company_name varchar(256),
    catch_phrase varchar(256),
    bs           varchar(256)
);
```

sales_with_geo_data:

```
create table if not exists assessment.sales_with_geo_data
(
    order_id              integer,
    customer_id           integer,
    product_id            integer,
    quantity              integer,
    price                 double precision,
    order_date            date,
    latitute              numeric(10, 8),
    longitude             numeric(11, 8),
    weather_id            integer,
    weather_main          varchar(256),
    weather_description varchar(256),
    weather_icon          varchar(256),
    base                  varchar(256),
    temperature           double precision,
    feels_like            double precision,
    temp_min              double precision,
    temp_max              double precision,
    pressure              double precision,
    humidity              double precision,
    sea_level             double precision,
    grnd_level            double precision,
    visibility            double precision,
    wind_speed            double precision,
    wind_deg              double precision,
    wind_gust             double precision,
    rain_1h               double precision,
    clouds_all            double precision,
    dt                    integer,
    sys_sunrise           integer,
    sys_sunset            integer,
    timezone              integer,
    id                    integer,
    name                  varchar(256),
    cod                   integer
);
```

Having the necessary tables created within the database, we can now go back to ADF and run a pipeline that will load data from the raw-lake and silver-lake, into each of the tables we need to populate:

02 loadData:



With this successful pipeline run, the ETL portion of the sales data solution is complete. Next, we will move back to Postgres to create the views that will contain the aggregations required, as well as the source data needed to create the visualisations that are shown in PowerBI later.

3) Creating views to store aggregations

As the data flow in ADF has transformed the various source datasets into 2 datasets that we can directly use for analytics, and we have loaded those 2 datasets into our database, we can create views that contain the aggregation logic given by business/consumers to easily obtain information they require relating to metrics they are looking for. Here are two examples of views that are generated to showcase the aggregation logic:

vw_total_sales_per_customer:

```
create or replace view assessment.vw_total_sales_per_customer(name, total_sales_amount)
as
SELECT sales_per_user.name,
       round(sum(sales_per_user.quantity::double precision *
sales_per_user.price)::numeric, 2) AS total_sales_amount
FROM assessment.sales_per_user
GROUP BY sales_per_user.name
ORDER BY sales_per_user.name;
```

vw_average_order_quantity_per_product:

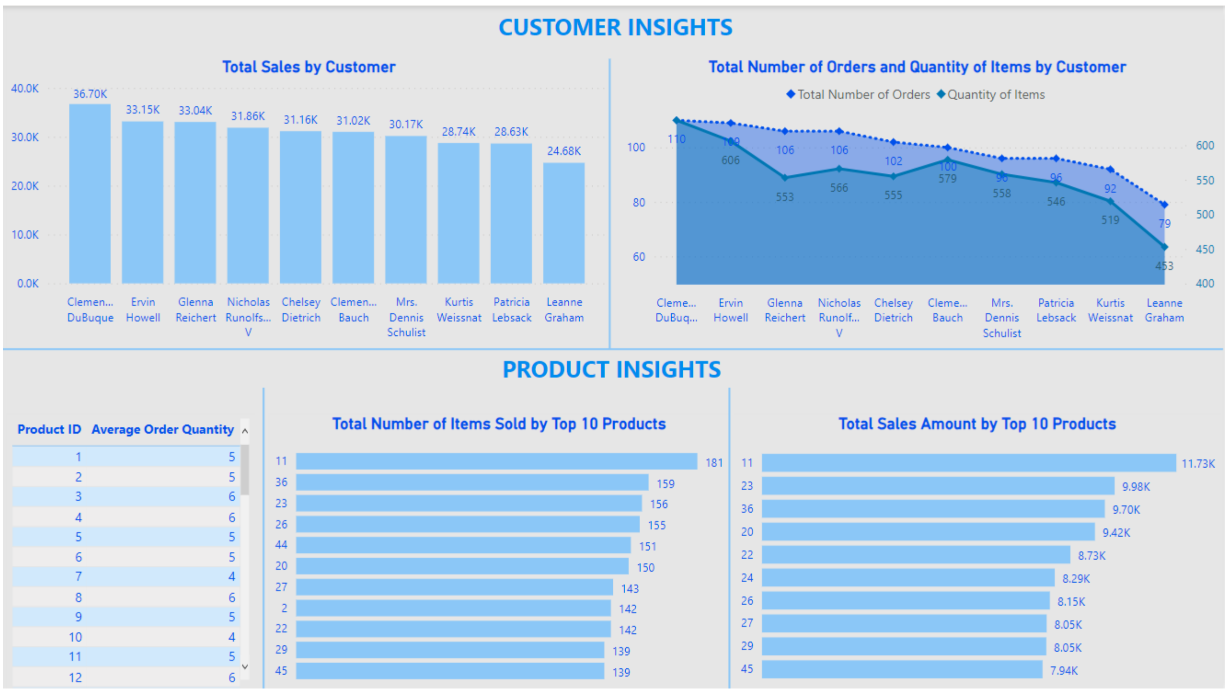
```
create or replace view assessment.vw_average_order_quantity_per_product(product_id,
avg_order_quantity) as
SELECT sales.product_id,
       sum(sales.quantity) / count(DISTINCT sales.order_id) AS avg_order_quantity
FROM assessment.sales
GROUP BY sales.product_id
ORDER BY sales.product_id;
```

Note: 'DISTINCT' is used in the count of order_id as the 'order_id' field is not unique and in a production setting, we could potentially have an order inclusive of different customers but the same product, and in such a situation, not using 'DISTINCT' would incorrectly increase the number of orders.

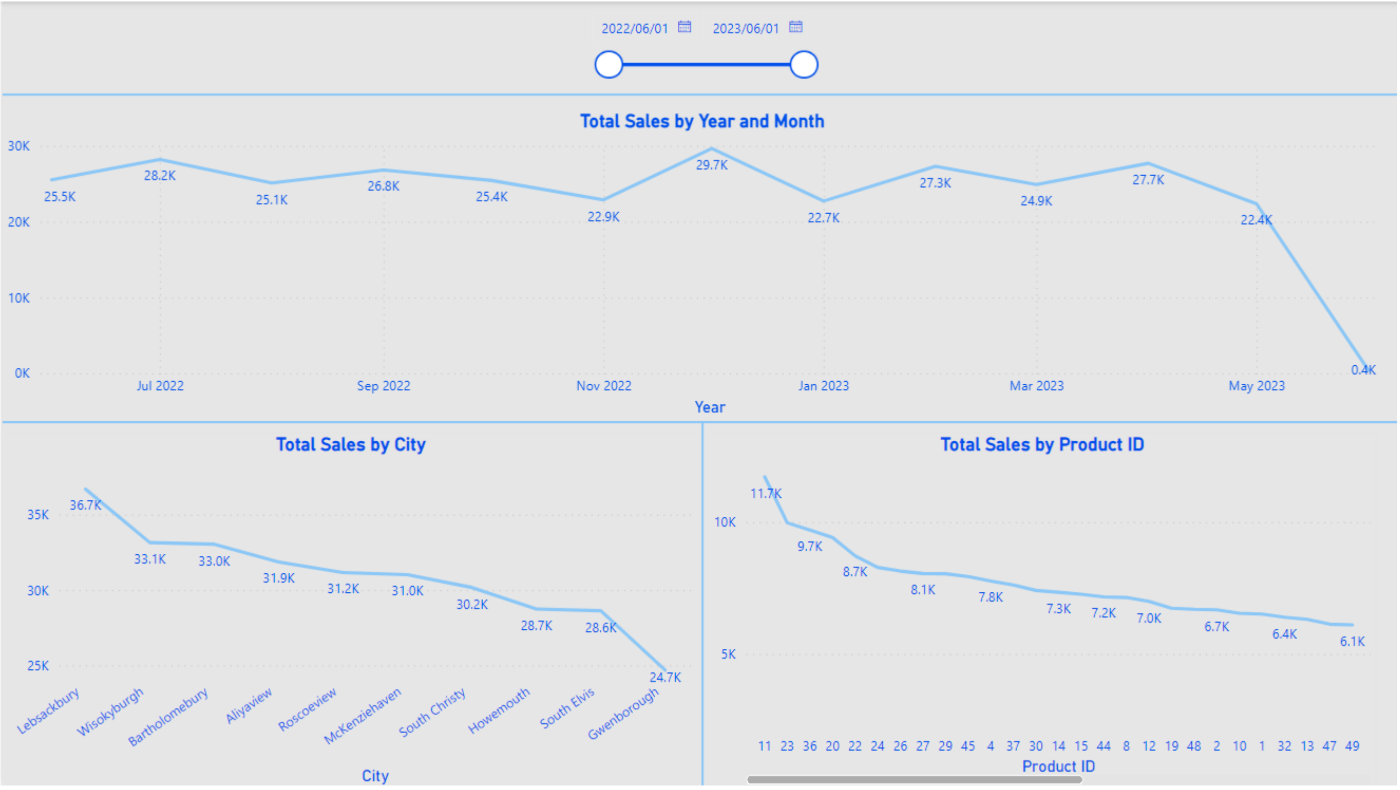
4) Visualising the aggregations and additional analytics

Instead of creating views for each metric of analysis that may be required, a business intelligence tool can be used to visualise the aggregations pertaining to each of the metrics, as well as include additional metrics that may be useful to business users or leadership. PowerBI is used to create these reports showcasing the various aggregations that can be performed on the sales data:

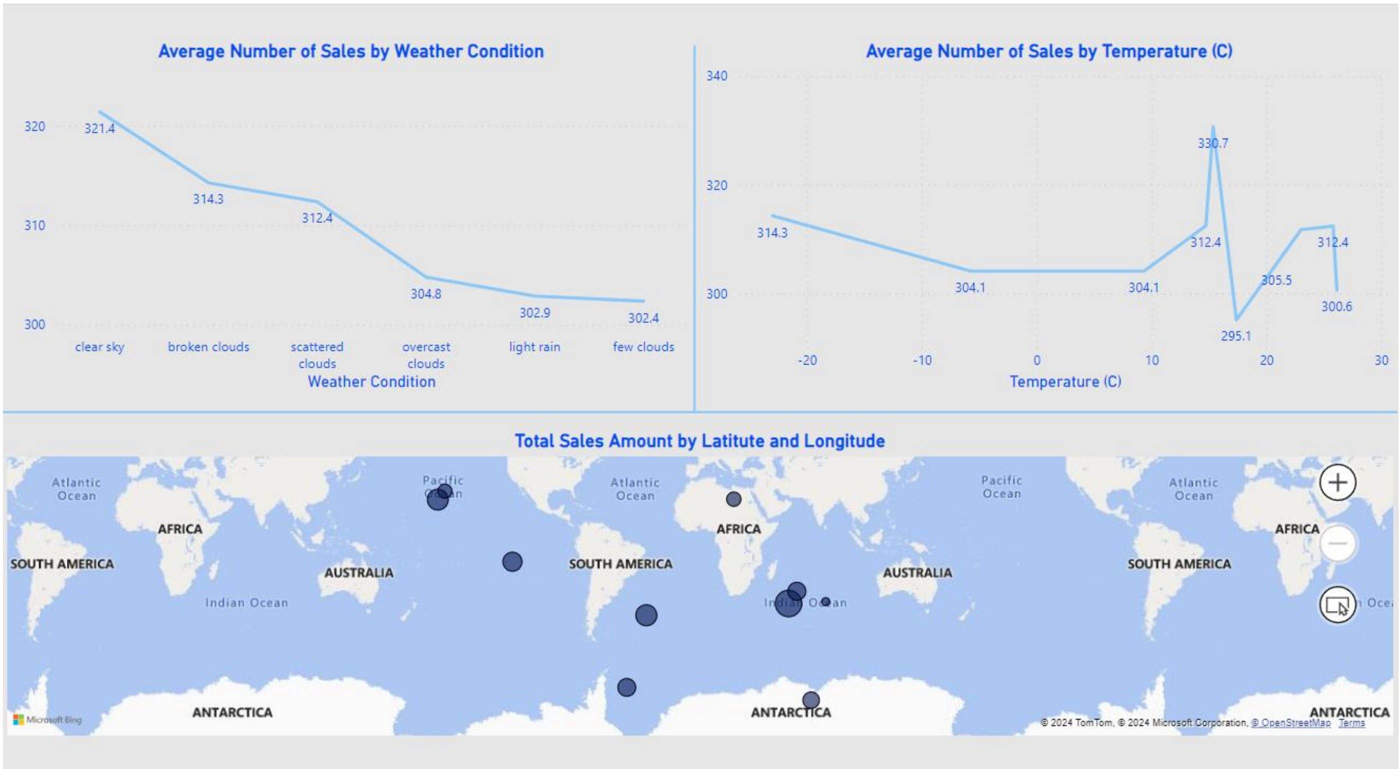
Sales by User:



Sales Trends:



Sales by Geo:



Not only do these visualisation allow us to quickly illustrate business-actionable metrics for business, but they also communicate to us as data engineers when there are inaccuracies in data - for example, looking at the 'Total Sales Amount by Latitude and Longitude' map in 'Sales by Geo', we can see that the latitude and longitude information provided in the user data source is incorrect, as those values correspond to places where a consumer couldn't physically be present.

In conclusion, having a good understanding of sourcing, transforming, loading, warehousing, aggregating and visualising data (i.e. being proficient in the end-to-end cycle of a data solution) is powerful for a data engineer to perform their tasks, while also ensuring that data is being used and consumed is representative of what it is supposed to imply in the real world.