DSAI Project Team 6

Members: Hanafi, Ismail, Kok Siang, Shaun, Xiangrong, Zachary

Introduction (Dataset used)

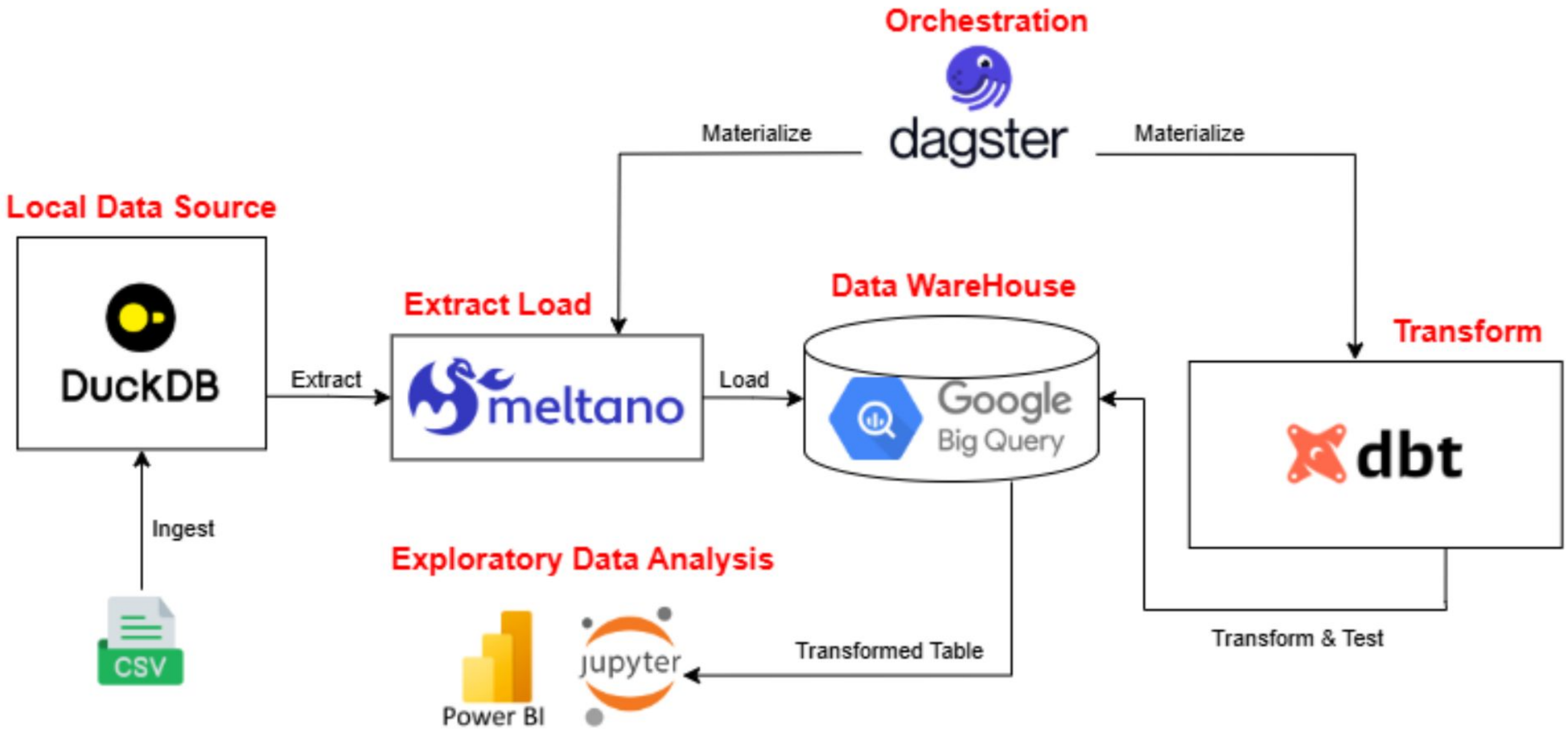
Dataset:

- **Citi Bike Trip Histories** (CSV files)from https://citibikenyc.com/system-data
    - JC-202505-citibike-tripdata (Jersey City May 2025)
- **Data includes:**
    - Ride ID
    - Start Time, Ended Time
    - Start and End Station:
        - ID
        - Name
        - Latitude
        - Longitude
    - Member or Casual

Introduction (Project Goal)

- Build an automated ELT Data Pipeline
- Extract valuable business insights from the dataset, focusing on:
    - Usage trends
    - Revenue and Trip Count Composition
    - Start and End Station Popularity
    - Anomaly Detection

Introduction (ELT Data Warehouse Architecture)



EDA (Usage Trends)

Analyse Usage Trends throughout the Day

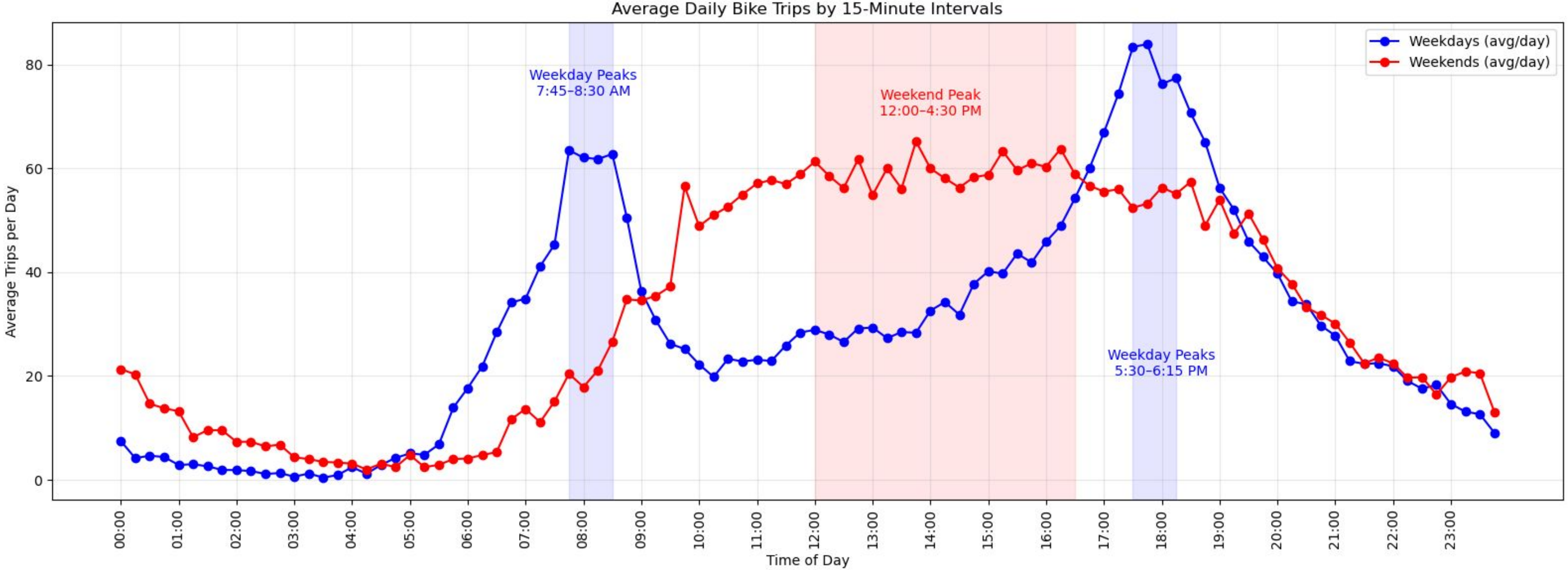
- Avg Trip Count per Day in 15 mins intervals
- Separated by Weekday and Weekends

Findings

- Peak timings on **Weekdays** aligns with commute times.
- Peak timings on **Weekends** shows a more leisure-oriented usage.

Action to take

- Implement Increase price rate on peak periods for casual users



EDA (Revenue and Trip Composition)

Analyse Revenue and Trip Count Composition

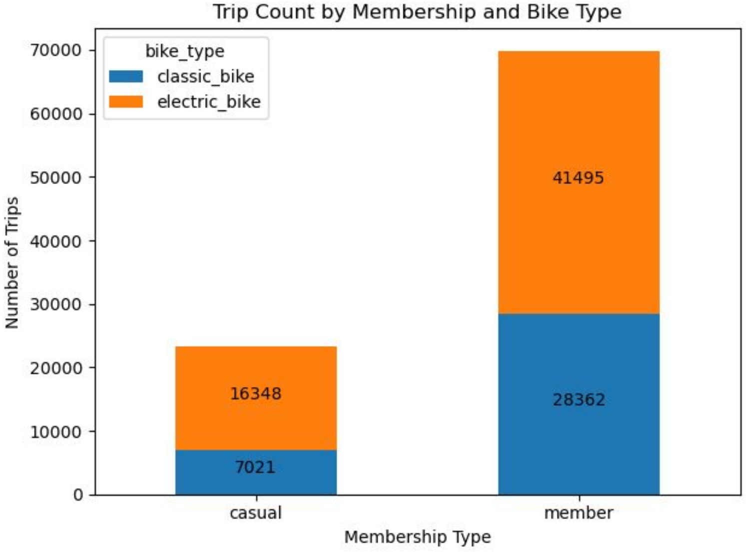
- Based on Membership and Bike Type

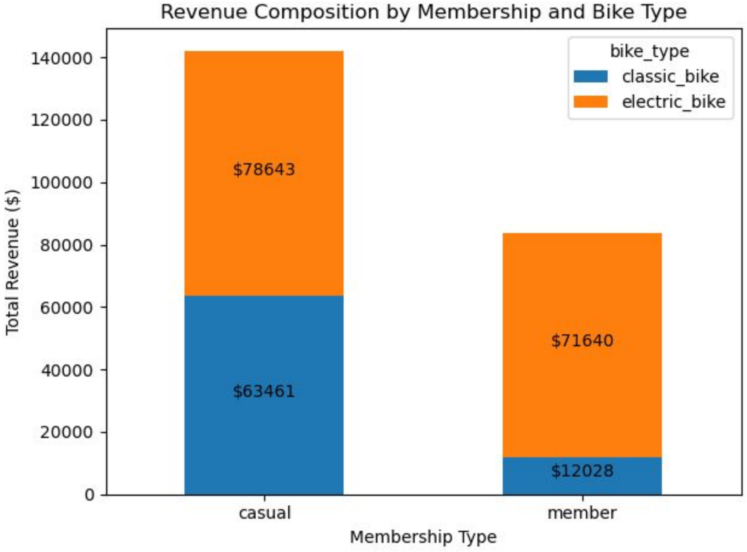
Findings

- **Casual** users generates more revenue despite having lower trip count in both **classic** and **electrical** bike
- **Classic** bike generates lower revenue per trip for **member** user

Action to take

- Increase the **electric** bike rates for **casual** users to encourage them to use the **classic** bike





EDA using Jupyter Notebook

Revenue Insights

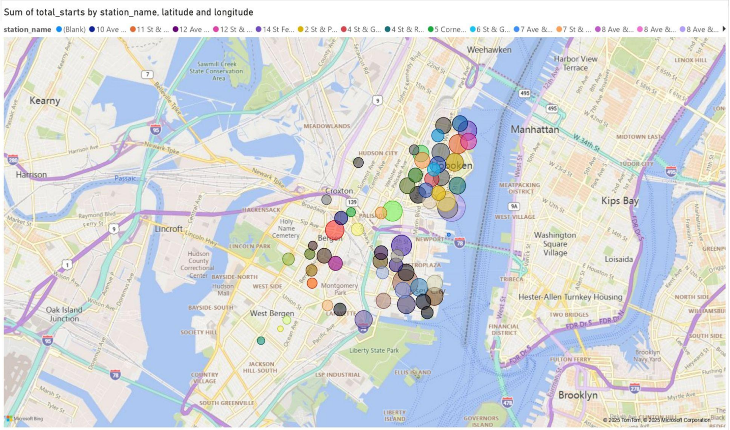
- Price Plan

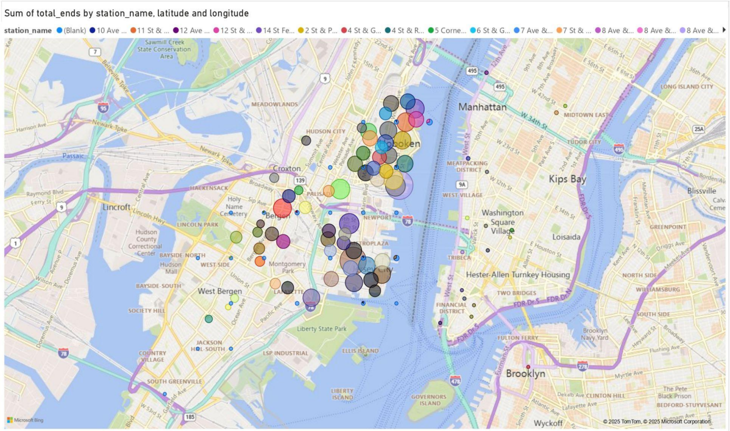
price_plans

Row	price_plan_id	membership_type	bike_type	unlock_fee	per_minute_...	included_mins	valid_from	valid_to
1	1	casual	classic_bike	4.99	0.38	30	2025-01-01	9999-12-31
2	2	casual	electric_bike	0.0	0.38	0	2025-01-01	9999-12-31
3	3	member	classic_bike	0.0	0.25	45	2025-01-01	9999-12-31
4	4	member	electric_bike	0.0	0.25	0	2025-01-01	9999-12-31

EDA (Start and End Station Popularity)

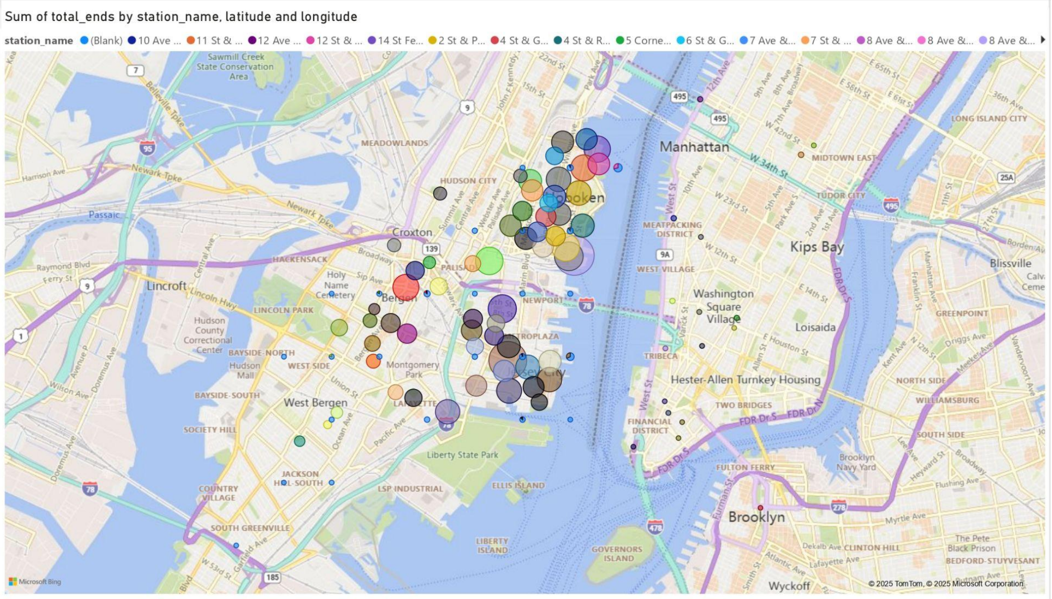
- Calculated based on station_name, latitude and longitude, and total_starts and total_ends
- Findings: The popular bike stations are gathered around Hoboken and Jersey City, which is where the metro areas are. Bergen and Montgomery Park also seem to have a fairly big cluster
- Action: Develop more bike stations at Bergen and Montgomery Park





EDA (End Station Popularity)

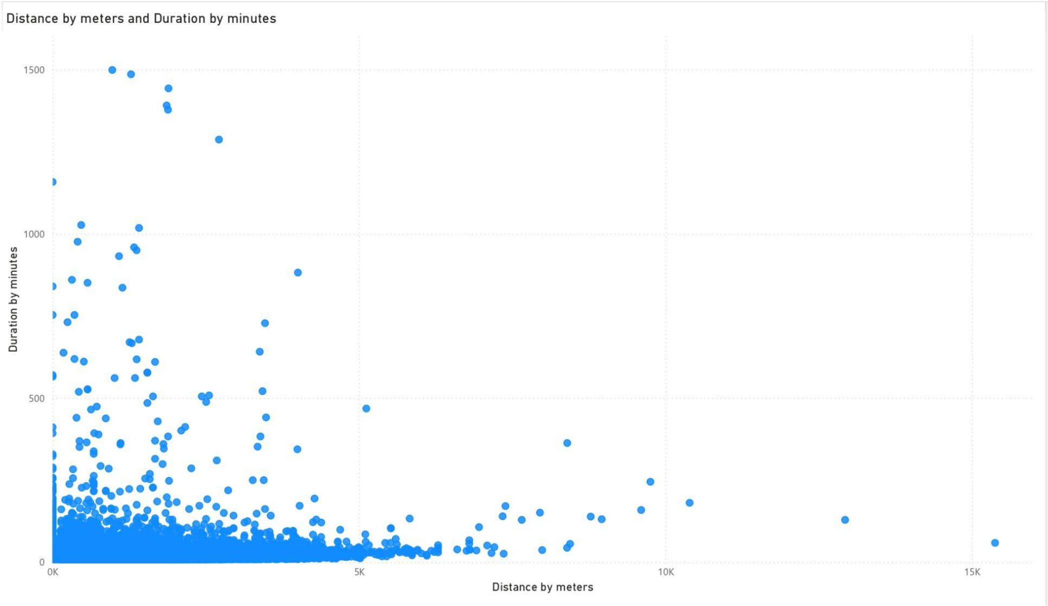
- Calculated based on station_name, latitude and longitude, and total_starts and total_ends



EDA using Power BI

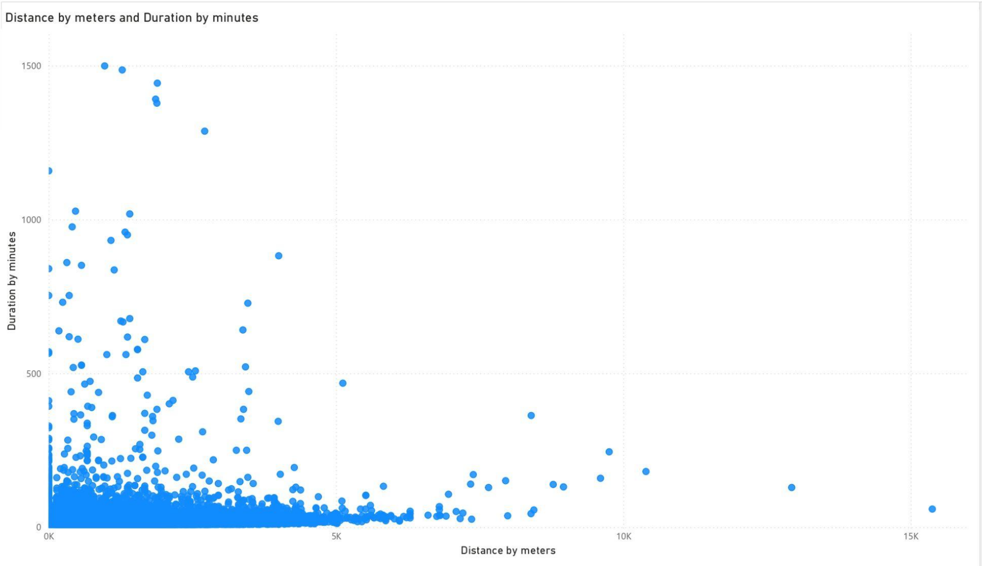
Anomaly Detection

- Based on data from duration_minutes and distance_metres
- Findings:



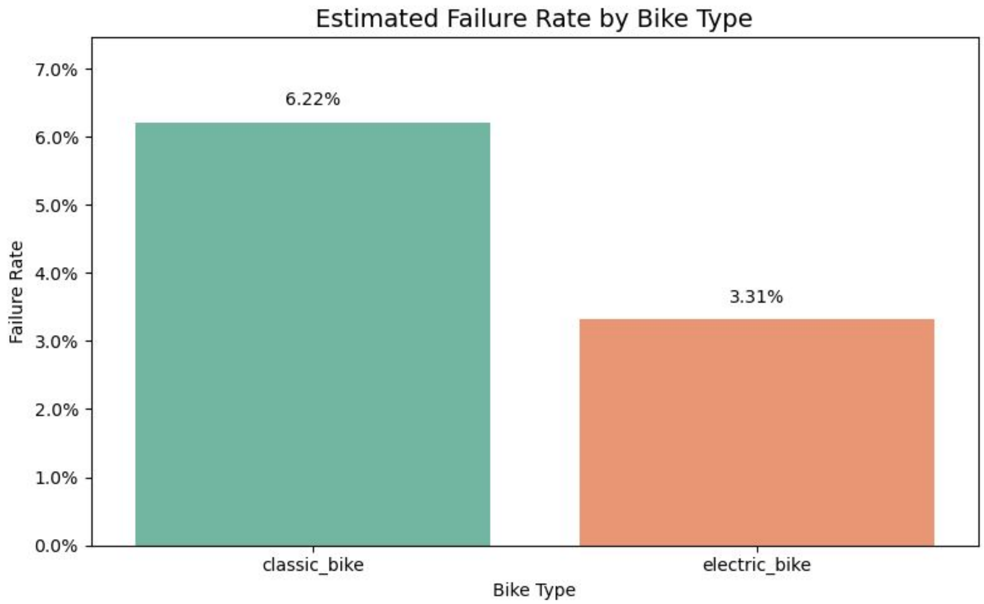
EDA (Anomaly Detection)

- Based on data from duration_minutes and distance_metres
- Findings: Most data are clustered within the range of 500 minutes and 5km, but there are outliers found
- Actions to take: Investigate the outliers and find out the cause



EDA (Potential Failure Rate)

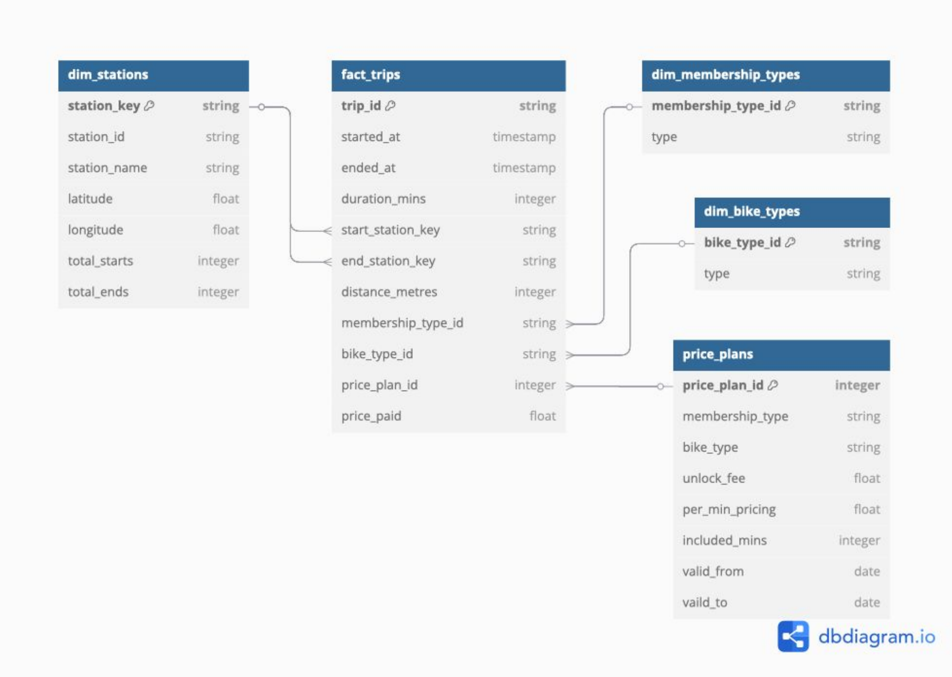
- Calculated using Jupyter Notebook based on outliers in duration_minutes and distance_metres
- Findings: Classic bikes seem to have a potentially higher failure rate than electric bikes
- Action to take: investigate the failure rate of the classic bikes, and see how it can be reduced



dbdiagram

Sample row:

trip_id	membership_type_id	start_time_id	end_time_id	duration_minutes	distance_m	price paid
abc123	abz_12456	20250618_08	20250618_08	12	2500	3.50



In our star schema the **fact_trips** table uses these five foreign-key columns to join out to the dimension tables:

- **start_station_key** → **dim_stations.station_key**
- **end_station_key** → **dim_stations.station_key**
- **bike_type_id** → **dim_bike_types.bike_type_id**
- **membership_type_id** → **dim_membership_types.membership_type_id**
- **price_plan_id** → **dim_price_plans.price_plan_id**

dbdiagram

In our Citibike star schema, every dimension table row (“one” side) can relate to multiple trip records in the fact table (“many” side)

Ref: fact_trips.bike_type_id > dim_bike_types.bike_type_id

Ref: fact_trips.start_station_key > dim_stations.station_key

Ref: fact_trips.end_station_key > dim_stations.station_key

Ref: fact_trips.membership_type_id > dim_membership_types.membership_type_id

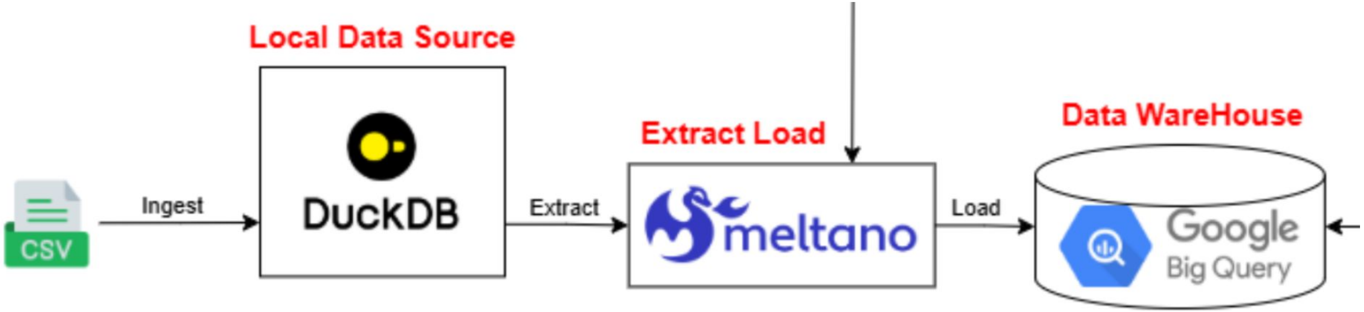
Ref: fact_trips.price_plan_id > dim_price_plans.price_plan_id

Sample Table:

bike_type_id	bike_type
1	Classic Bike
2	E-Bike

trip_id	bike_type_id	...other columns...
trip_001	1	...
trip_002	1	...
...		
trip_50	2	...

Ingestion



- Data is downloaded as csv file from the citybikenyc website and load into DuckDB (local data source).
- We use Meltano to orchestrate an EL pipeline
  - **Extractor (tap-duckdb)**: to stream the raw data out of DuckDB
  - **Loader (target-bigquery)**: will receive that stream and load it into BigQuery inside Google Cloud Project
- Once the raw data is available in BigQuery, we can use dbt to clean, test and model the newly-ingested data.

Key implementation logic for target tables (1/2)

SQL code snippet for **fact_trips** table

```sql
SELECT
    CAST(ride_id AS STRING) AS trip_id,
    CAST(started_at AS TIMESTAMP) AS started_at,
    CAST(ended_at AS TIMESTAMP) AS ended_at,
    CAST(TIMESTAMP_DIFF(CAST(ended_at AS TIMESTAMP), CAST(started_at AS TIMESTAMP), MINUTE) AS INT64) AS duration_mins,
    CAST(start_stations.station_key AS STRING) AS start_station_key,
    CAST(end_stations.station_key AS STRING) AS end_station_key,
    CAST(
        ST_DISTANCE(
            ST_GEOGPOINT(start_stations.longitude, start_stations.latitude),
            ST_GEOGPOINT(end_stations.longitude, end_stations.latitude),
            TRUE
        ) AS INT64
    ) AS distance_metres,
    CAST(membership_types.membership_type_id AS STRING) AS membership_type_id,
    CAST(bike_types.bike_type_id AS STRING) AS bike_type_id,
    CAST(price_plans.price_plan_id AS INT64) AS price_plan_id,
    CAST(
        ROUND(
            COALESCE(price_plans.unlock_fee, 0) +
            COALESCE(
                CASE
                    WHEN CAST(TIMESTAMP_DIFF(CAST(ended_at AS TIMESTAMP), CAST(started_at AS TIMESTAMP), MINUTE) AS INT64) > COALESCE(price_plans.included_mins, 0)
                    THEN (CAST(TIMESTAMP_DIFF(CAST(ended_at AS TIMESTAMP), CAST(started_at AS TIMESTAMP), MINUTE) AS INT64) - COALESCE(price_plans.included_mins, 0))
                        * COALESCE(price_plans.per_minute_pricing, 0)
                    ELSE 0
                END,
                0
            ),
            2
        ) AS FLOAT64
    ) AS price_paid
FROM
    {{ source(env_var('BIGQUERY_SOURCE_DATASET'), env_var('BIGQUERY_RAW_DATA_TABLE')) }} source_data
INNER JOIN {{ ref('dim_stations') }} start_stations
    ON COALESCE(source_data.start_station_id, '') = COALESCE(start_stations.station_id, '')
    AND COALESCE(source_data.start_station_name, '') = COALESCE(start_stations.station_name, '')
    AND COALESCE(CAST(source_data.start_lat AS FLOAT64), 0.0) = COALESCE(CAST(start_stations.latitude AS FLOAT64), 0.0)
    AND COALESCE(CAST(source_data.start_lng AS FLOAT64), 0.0) = COALESCE(CAST(start_stations.longitude AS FLOAT64), 0.0)
```

New features

- duration_mins: computing difference between start and end times
- distance_metres: using BigQuery's functions (ST_DISTANCE, ST_GEOPOINT)
- price_paid: referencing price_plans seed table to compute price paid per trip

Defensive coding

- Explicit casting to mitigate inference differences between dbt and BigQuery
- Using COALESCE function to mitigate errors due to missing/null values
- Using common environment variables by implementing python-dotenv

Key implementation logic for target tables (2/2)

SQL code snippet for dim_stations table

```sql
WITH stations AS (
    SELECT
        station_id,
        station_name,
        latitude,
        longitude,
        SUM(CASE WHEN station_role = 'start' THEN 1 ELSE 0 END) AS total_starts,
        SUM(CASE WHEN station_role = 'end' THEN 1 ELSE 0 END) AS total_ends
    FROM (
        SELECT
            start_station_id AS station_id,
            start_station_name AS station_name,
            start_lat AS latitude,
            start_lng AS longitude,
            'start' AS station_role
        FROM {{ source(env_var('BIGQUERY_SOURCE_DATASET'), env_var('BIGQUERY_RAW_DATA_TABLE')) }}
        UNION ALL
        SELECT
            end_station_id AS station_id,
            end_station_name AS station_name,
            end_lat AS latitude,
            end_lng AS longitude,
            'end' AS station_role
        FROM {{ source(env_var('BIGQUERY_SOURCE_DATASET'), env_var('BIGQUERY_RAW_DATA_TABLE')) }}
    ) t
    GROUP BY station_id, station_name, latitude, longitude
)
SELECT
    {{ dbt_utils.generate_surrogate_key(['station_id', 'station_name', 'latitude', 'longitude']) }} station_key,
    CAST(station_id AS STRING) AS station_id,
    CAST(station_name AS STRING) AS station_name,
    CAST(latitude AS FLOAT64) AS latitude,
    CAST(longitude AS FLOAT64) AS longitude,
    CAST(total_starts AS INT64) AS total_starts,
    CAST(total_ends AS INT64) AS total_ends
FROM
    stations
```

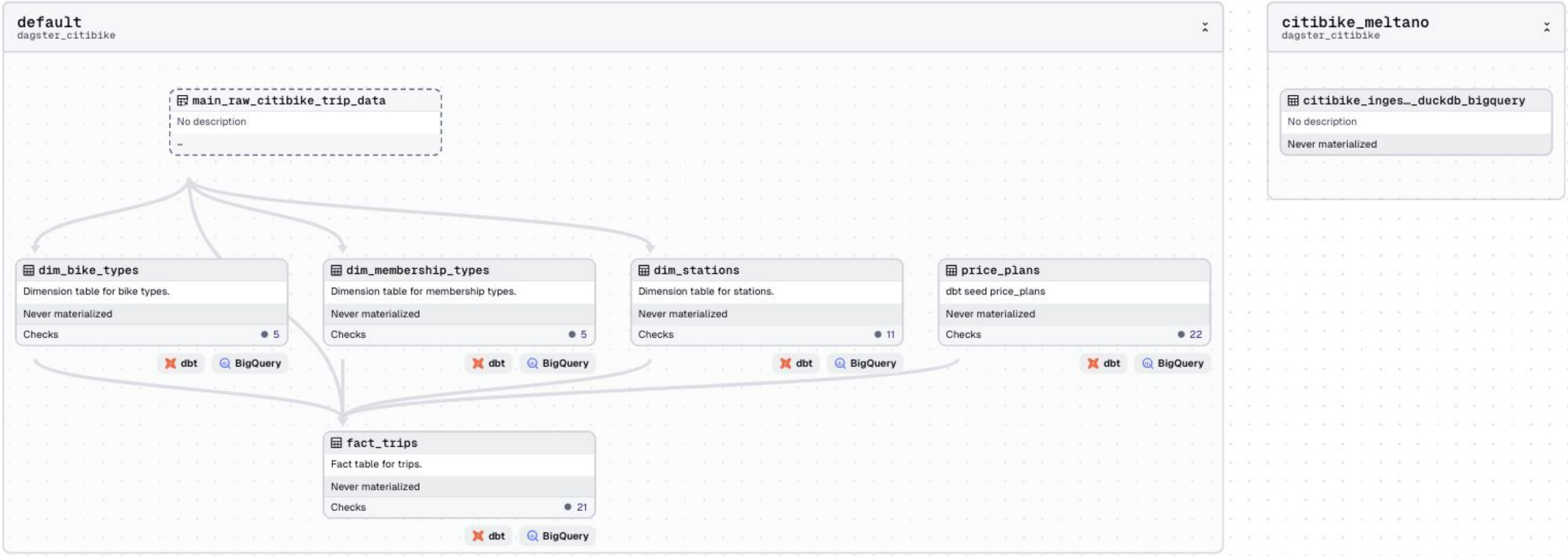
New features

- Every station as an unique record: using CTE to combine all start and end stations into a temp table and group them by four columns to uniquely identify every station
- total_starts, total_ends: using a temp column to store roles of every station and counting how many times each station is used as the start station for trips, likewise for end station

Data quality

- Generating primary key based on four columns to mitigate incomplete source data which can give rise to downstream discrepancies (e.g., distance computation with null or zero results)

Dagster Orchestration



Dagster - Assets

```python
@asset(group_name="citibike_meltano")
def citibike_ingestion_duckdb_bigquery():
    # Run Meltano ELT: DuckDB → BigQuery
    result = subprocess.run(
        meltano_args,
        cwd=meltano_dir,
        capture_output=True,
        text=True
    )
    print(result.stdout, result.stderr)
```

```python
@dbt_assets(manifest=dbt_manifest_path)
def citibike_dbt(context:
AssetExecutionContext, dbt: DbtCliResource):
    # Run dbt build
    yield from dbt.cli(["build"],
    context=context).stream()
```

Resources

```python
# Ensure the following absolute paths point
to the Meltano and dbt projects.
meltano_dir = EnvVar("MELTANO_PROJECT_ROOT").
get_value()

dbt_dir = EnvVar("DBT_PROJECT_ROOT").get_value
()

# Meltano resources
meltano_tap = "tap-duckdb"
meltano_target = "target-bigquery"
meltano_args = ["meltano", "run",
meltano_tap, meltano_target]

# Dbt resources
dbt_manifest_path = f"{dbt_dir}/target/
manifest.json"
Tabnine | Edit | Test | Explain | Document
@resource
def dbt_bigquery():
    return DbtCliResource(
        project_dir=dbt_dir)
```

Definitions

```python
all_assets = load_assets_from_modules
([meltano_assets,dbt_assets])

defs = Definitions(
    assets=all_assets,
    resources={
        "dbt": dbt_bigquery,
    }
)
```