



# Northeastern University

## College of Engineering

Data Warehousing & Integration  
IE 6750  
FALL 2024

## FINAL REPORT

Group 01

Layashree Adepu

Pavithra Moorthy

Adepu.l@northeastern.edu

Moorthy.p@northeastern.edu

**Submission Date: 12/07/2024**

# BIKE FLOW ANALYTICS

## Problem Statement:

The Bay Wheels bike-sharing service faces challenges in maintaining bike availability across its network of stations. Some stations frequently run out of bikes, while others have too many, leading to user dissatisfaction and inefficiencies in the system. Currently, the service does not have a consolidated way to analyse key factors like ride patterns, peak times, and user behaviour to better manage bike distribution.

To address this, we will create a centralised on-premise database that collects trip data, station usage, and user information. This will help the company analyse patterns, improve bike reallocation, and optimize station management, ensuring bikes are available when and where they are needed most.

## Database Overview:

The operational database for the Bay Wheels bike-sharing service is designed to collect and organize key information from every trip, allowing for detailed analysis and decision-making.

Each column in the database captures specific aspects of the bike-sharing system, which can be leveraged to optimize operations and improve user experience

## Columns:

- duration\_sec: Total trip duration in seconds, useful for analyzing ride patterns and trip lengths.
- start\_time / end\_time: Start and end timestamps for each trip, essential for identifying peak usage periods.
- start\_station\_id / start\_station\_name: Unique ID and name of the station where the trip starts, used for tracking station popularity.
- start\_station\_latitude / start\_station\_longitude: Geographic coordinates of the start station, important for location-based analysis.
- end\_station\_id / end\_station\_name: Unique ID and name of the station where the trip ends, helping to analyze route patterns and station demand.
- end\_station\_latitude / end\_station\_longitude: Geographic coordinates of the end station, useful for geospatial studies.
- bike\_id: Unique identifier for each bike, allowing for maintenance tracking and usage patterns.
- user\_type: Indicates whether the rider is a Subscriber (member) or Customer (casual user), for behavior analysis.
- bike\_share\_for\_all\_trip: Identifies trips under the Bike Share for All program, important for evaluating program participation.

We have an extensive dataset that covers key aspects such as trip duration, start and end times, station details, bike IDs, and user types. To enhance the analysis and provide more insights, we plan to add additional columns like user details (e.g., age group and subscription type), distance traveled (calculated between start and end stations). These new columns will be generated or calculated based on existing data, allowing for deeper analysis of user behaviour, trip patterns, and operational efficiencies within the bike-sharing system.

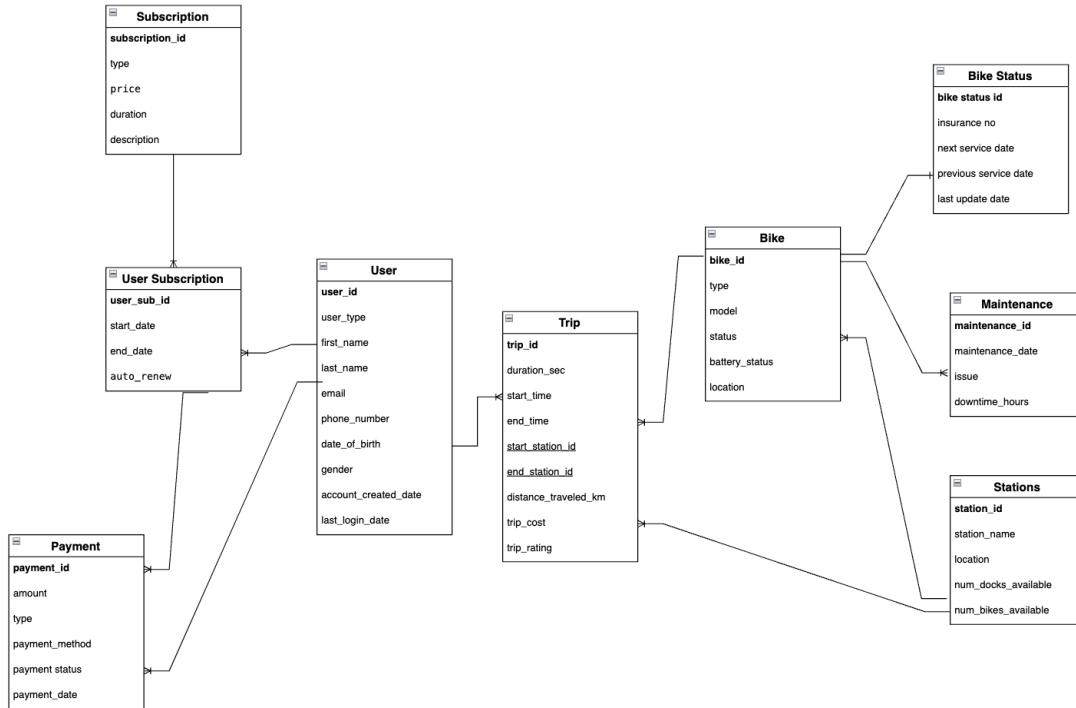
## What We Hope to Analyze:

In the future, we hope to analyze user behaviour patterns to gain insights into rental trends, peak usage times, and demographic preferences for Lyft Bikes. This analysis will help optimize fleet management by assessing bike distribution and maintenance needs, ensuring availability aligns with demand.

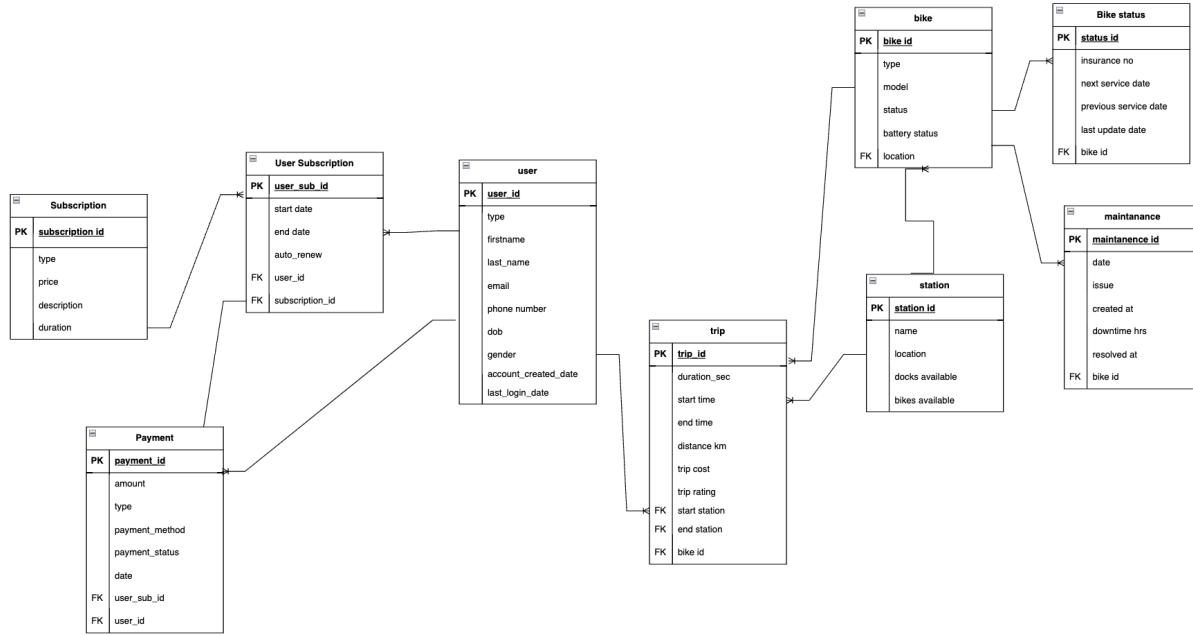
Additionally, I aim to evaluate the effectiveness of integration with public transportation systems, measuring the impact on rental rates and user satisfaction. Overall, these insights will guide improvements in service delivery and support Lyft's commitment to enhancing urban mobility.

- **Optimize Bike Reallocation:** Predict high-demand stations and recommend rebalancing of bikes across stations.
- **Peak Time Analysis:** Identify the busiest times of day/week to optimize bike and station management.
- **User Behaviour:** Analyse differences between casual users and subscribers to tailor marketing or service strategies.
- **Station Usage Efficiency:** Evaluate which stations are under- or over-utilized and plan for capacity adjustments.

ERD:



## Logical Model:



## Data Population Methodology

1. Data set upload as a csv file for trip table from Kaggle (90000 rows)
2. For other tables - <https://www.mockaroo.com>

### 1. Dimensions

- a. Bike
- b. Station
- c. User
- d. Subscription

### 2. Hierarchies

#### a. Date/Time Dimension

Hierarchy:

- o Year → Quarter → Month → Day
- o Year → Month → Week → Day → Hour

#### b. Location Dimension

Hierarchy:

- o Country → State → City → Zipcode
- o Country → Region → City → Zipcode

### 3. Measures

#### a. Trip Measures

- **Trip Count**: Total number of trips
- **Average Trip Duration**: Sum of all trip durations / Trip Count
- **Total Distance Traveled**: Sum of all distances for each trip
- **Trip Cost**: Sum of costs associated with all trips

- **Average Trip Rating:** Average of all ratings given for trips
- **Trips by Time of Day:** Number of trips during different parts of the day (morning, afternoon, evening, night)

#### b. User Measures

- **User Count:** Total number of unique users
- **New Users:** Number of users who signed up in a specific time period
- **Retention Rate:** Percentage of users who continue to use the service over a period of time
- **Average Subscription Length:** Average duration of active subscriptions per user
- **User Activity:** Count of trips per user over time

#### c. Bike Measures

- **Bike Utilization Rate:** Percentage of time a bike is in use vs. available
- **Maintenance Frequency:** Average number of maintenance events per bike
- **Mileage per Bike:** Total distance traveled by each bike
- **Downtime:** Total time each bike is out of service

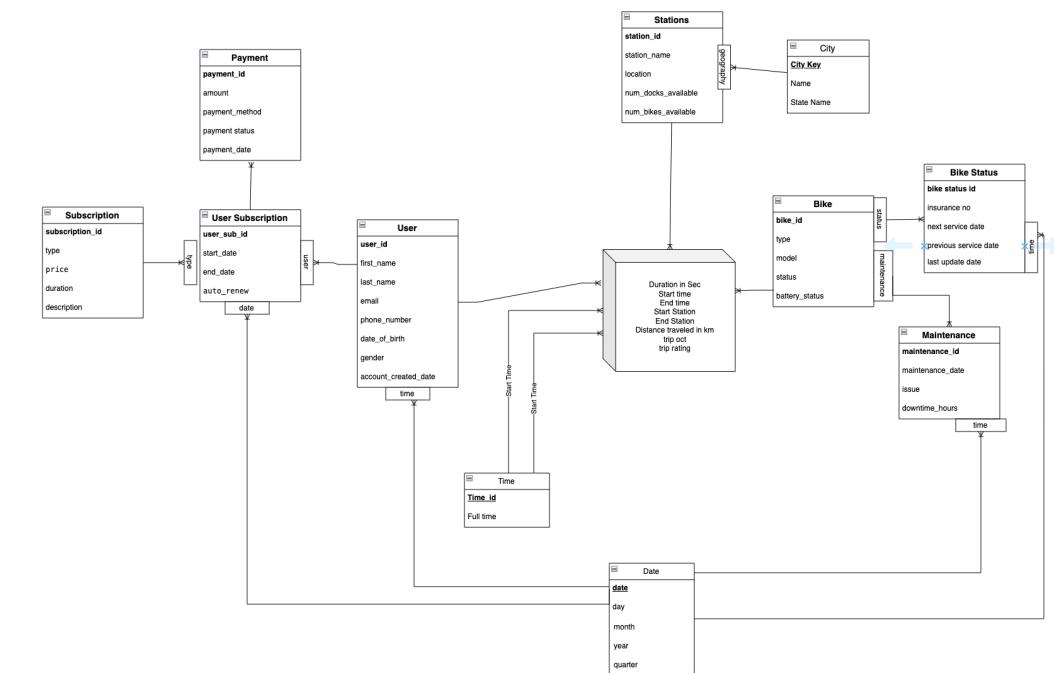
#### d. Station Measures

- **Station Usage:** Number of trips starting or ending at each station
- **Capacity Utilization:** Average percentage of docks occupied at a station
- **Peak Hours:** Most frequently used hours at a station
- **Top Stations:** Stations with the highest number of rentals or returns

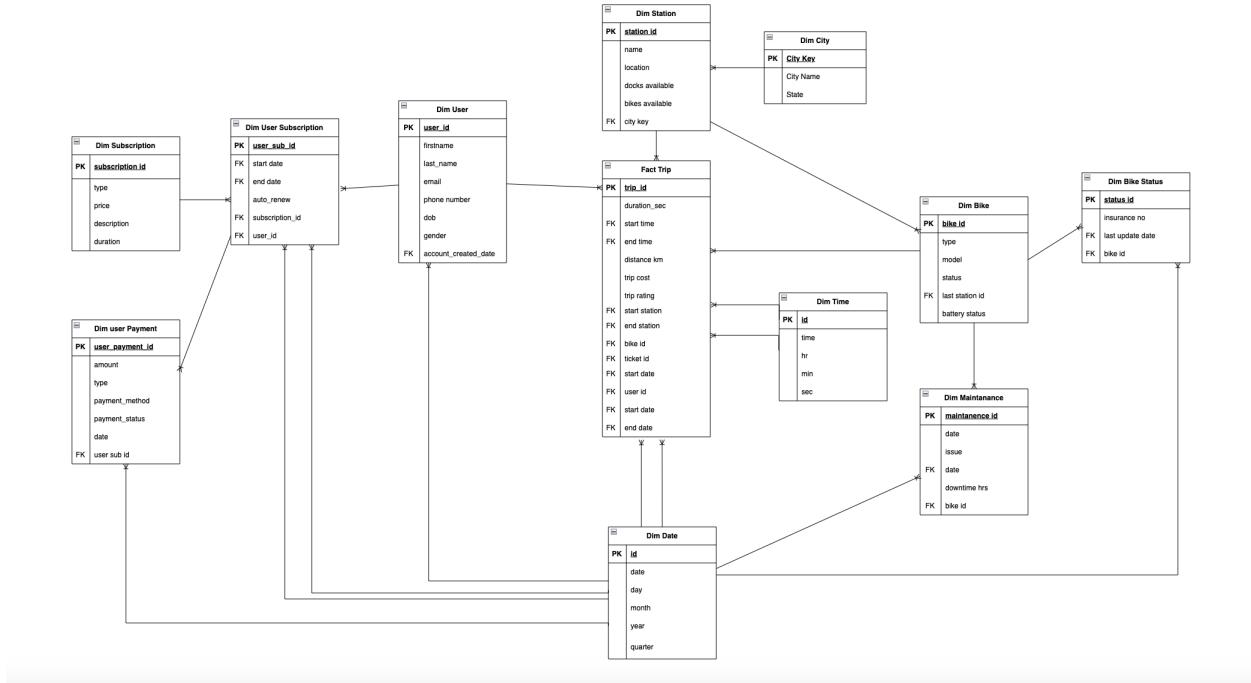
#### e. Financial Measures (Payments)

- **Total Revenue:** Sum of all payments
- **Average Payment per Trip:** Total Revenue / Trip Count
- **Subscription Revenue:** Total revenue from subscriptions

Dimensional Conceptual Model:



## Dimensional Logical Model:



## Implementation:

The screenshot shows the pgAdmin interface. The top bar has a title '<postgres> Script-1'. Below it is a toolbar with icons for file operations, search, and refresh. A dropdown menu is open, showing 'dim\_user 1 x'. The main area contains a SQL query editor with the following code:

```
select * from dim_user du
```

Below the query editor is a results grid titled 'dim\_user 1 x'. The grid has a header row with columns: user\_id, first\_name, last\_name, email, phone\_number, other\_details, and account\_created\_date. The 'user\_id' column is highlighted with a green circle icon.

The screenshot shows a PostgreSQL database interface with two tabs open. The top tab is titled 'Script-1' and contains the SQL query: 'select \* from dim\_subscription ds'. The bottom tab is titled 'dim\_subscription 1' and displays the results of the query in a grid format. The grid has columns: sub\_id, type, price, description, and duration\_in\_months. The 'sub\_id' column is currently sorted in ascending order. The 'type' column is sorted in descending order. The 'price' column is sorted in ascending order. The 'description' column is sorted in descending order. The 'duration\_in\_months' column is sorted in ascending order.

sub_id	type	price	description	duration_in_months
1	Subscription A	\$10.00	Description A	12
2	Subscription B	\$20.00	Description B	24
3	Subscription C	\$30.00	Description C	36
4	Subscription D	\$40.00	Description D	48
5	Subscription E	\$50.00	Description E	60

The screenshot shows a PostgreSQL database interface with the following details:

- Script-1**: The current tab is titled "Script-1".
- Query Editor**: The main area contains the SQL command: `select * from dim_bike db |`.
- Results Tab**: A results tab titled "dim\_bike 1" is open.
- Result Grid**: The grid displays columns: bike\_id, model, status, and battery\_status. The first row shows values: 123, bike, 1, and 1.
- Toolbar**: Standard database toolbar icons are visible.
- Status Bar**: The status bar at the bottom says "Enter a SQL expression to filter results (use Ctrl+Space)".

```
*<postgres> Script-1 ×
▶ └─ select * from dim_bike_status dbs
```

dim\_bike\_status 1 ×

```
select * from dim_bike_status dbs | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 bike_status_id	Az status	next_service_date	previous_service_date	last_update_date	123 bike_id

```
*<postgres> Script-1 ×
▶ └─ select * from dim_station ds
```

dim\_station 1 ×

```
select * from dim_station ds | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 station_id	Az station_name	Az location	123 num_bikes_available	123 num_docks_available

```
*<postgres> Script-1 × *<postgres> Script
▶ └─ select * from dim_support_tickets dst
```

dim\_support\_tickets 1 ×

```
select * from dim_support_tickets dst | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 ticket_id	Az issue_description	created_at	resolved_at	123 user_id

```
*<postgres> Script-1 ×
▶ └─ select * from dim_payment dp
```

dim\_payment 1 ×

```
select * from dim_payment dp | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 payment_id	Az payment_method	Az payment_status	payment_date

```
*<postgres> Script-1 × *<postgres> Script
▶ └─ select * from dim_maintenance dm
```

dim\_maintenance 1 ×

```
select * from dim_maintenance dm | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 maintenance_id	Az issue	123 downtime_hours	created_at	123 bike_id

```
*<postgres> Script-1 × *<postgres> Script
▶ └─ select * from dim_state ds
```

dim\_state 1 ×

```
select * from dim_state ds | Enter a SQL expression to filter results (use Ctrl+Space)
```

Grid	123 state_id	Az state_name

```

* <postgres> Script-1 × * <postgres> Script
  _ select * from dim_city dc |
dim_city 1 ×
  ⌂ select * from dim_city dc | Enter a SQL expression to filter results (use Ctrl+Space)
Grid ① 123 city_id ▾ A-Z city_name ▾
  ⌂

* <postgres> Script-1 × * <postgres> Script
  _ select * from dim_month dm |
dim_month 1 ×
  ⌂ select * from dim_month dm | Enter a SQL expression to filter results (use Ctrl+Space)
Grid ① 123 month_id ▾ 123 month_number ▾ A-Z month_name ▾
  ⌂

* <postgres> Script-1 × * <postgres> Script
  _ select * from dim_month dm |
dim_month 1 ×
  ⌂ select * from dim_month dm | Enter a SQL expression to filter results (use Ctrl+Space)
Grid ① 123 month_id ▾ 123 month_number ▾ A-Z month_name ▾
  ⌂

* <postgres> Script-1 × * <postgres> Script
  _ select * from dim_user_subscription dus |
dim_user_subscription 1 ×
  ⌂ select * from dim_user_subscription dus | Enter a SQL expression to filter results (use Ctrl+Space)
Grid ① 123 user_sub_id ▾ ② start_date ▾ ③ end_date ▾ 123 sub_id ▾ 123 user_id ▾
  ⌂

* <postgres> Script-1 × * <postgres> Script
  _ select * from fact_trip ft |
fact_trip 1 ×
  ⌂ select * from fact_trip ft | Enter a SQL expression to filter results (use Ctrl+Space)
Grid ① 123 trip_id ▾ 123 duration_in_sec ▾ ② start_time ▾ ③ end_time ▾ 123 distance_travelled_km ▾ 123 cost ▾ 123 start_station ▾ 123 end_station ▾ 123 bike_id ▾ 123 user_id ▾
  ⌂

```

## PRIMARY EVENTS

- Trips: A user rents a bike and returns it to another station.
- Payments: A user makes a payment for either a trip or a subscription.
- User Subscriptions: A user subscribes to a specific bike-sharing plan.
- Support Tickets: A user reports an issue or creates a support request.
- Bike Maintenance: A bike is taken out of service for repairs or maintenance.

## OLAP QUERIES

1. Total revenue generated from payments for subscriptions in the last month

```
SLICE(Dim_Payments, Date.Month = "September")
ROLL_UP(Dim_Payments, Dim_Subscription.sub_id)
SUM(Dim_Payments, amount)
```

2. How many trips were taken by users in each station in the last quarter

```
ROLL_UP(Fact_Trips, Month.Quarter)
SLICE(Fact_Trips, Month.Quarter = "Q3")
ROLL_UP(Fact_Trips, start_station)
COUNT(Fact_Trips, trip_id)
```

3. Average duration of trips taken by users who have an active subscription

```
SLICE(Dim_UserSubscription, end_date > NOW())
ROLL_UP(Fact_Trips, user_id)
AVG(Fact_Trips, duration)
```

4. Which bikes had the highest number of trips associated with them last month

```
SLICE(FactTrips, Date.Month = "September")
ROLL_UP(FactTrips, bike_id)
COUNT(FactTrips, trip_id)
ORDER BY COUNT(FactTrips, trip_id) DESC
```

5. What are the maintenance issues reported for bikes and how long were they down

```
ROLL_UP(Dim_Maintenance, bike_id)
SUM(Dim_Maintenance, downtime_hrs)
SELECT(Dim_Maintenance, issue)
```

6. Calculate total payments by user and by month

```
ROLL_UP(Dim_Payments, user_id, Date.Month)
SUM(Dim_Payments, amount)
```

7. Calculate total trips by user and bike type

```
ROLL_UP(Fact_Trips, user_id, Dim_Bike.type)
COUNT(Fact_Trips, trip_id)
```

## ETL Implementation in Talend:

### 1. ETL Execution

- Insert or update data with surrogate key

### 2. Additional Features

- Implemented Type 3 SCD
- Calculated measures:
  - Trip duration in minutes → Fact Trip

### 3. Data Sources:

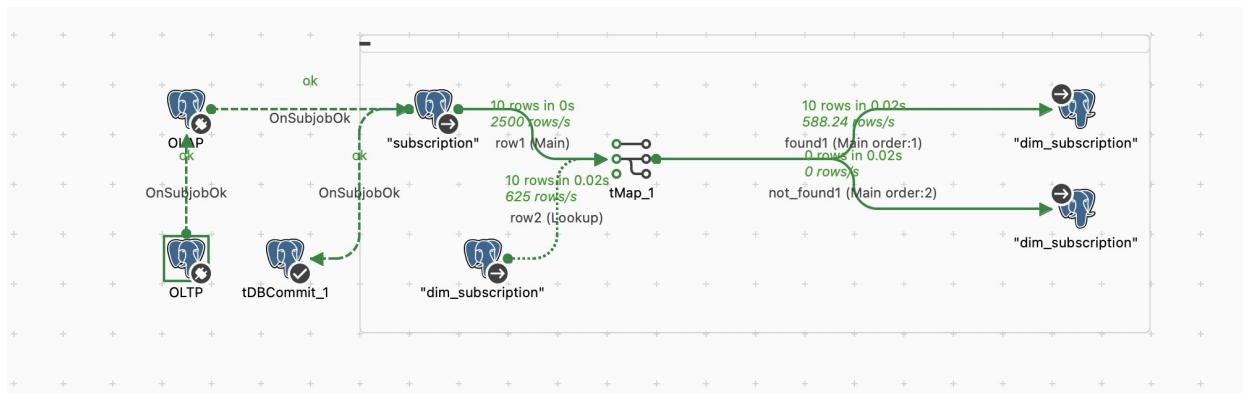
- CSV files - city
- XML files - date, time
- OLTP PostgresDB

### 4. Transformations:

- Converted to Date format
- Converted to Time format
- Converted to appropriate data types in the targetDB (int, decimal, String)

## ETL Execution

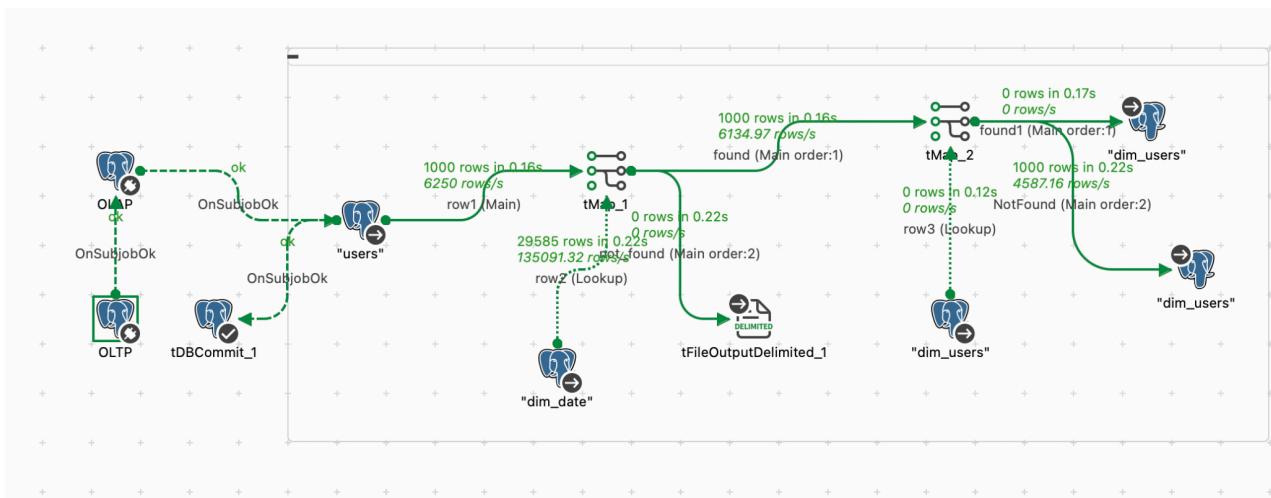
### 1. Dim\_Subscription (surrogate\_key : subscription\_dim\_



SQL

	subscription_dim_id [PK] integer	subscription_id integer	type character varying (50)	price numeric (10,2)	duration character varying (100)	description character varying (100)
1		873	1 Single Ride	3.99	30 minutes	Classic bike: 30 min for \$3.99, then \$0.30/min
2		874	2 Day Pass	15.00	1 day	Classic bike: 30 min free, then \$0.30/min
3		875	3 Month Pass	29.00	1 month	Classic bike: 45 min free, then \$0.20/min
4		876	4 Bay Wheels	150.00	1 year	Classic bike: 45 min free, then \$0.20/min
5		877	5 Lyft Pink	199.00	1 year	Classic bike: 45 min free, then \$0.20/min
6		878	6 Ebike - Single Ride	3.99	30 minutes	Ebike: \$3.99 unlock + \$0.30/min
7		879	7 Ebike - Day Pass	15.00	1 day	Ebike: Free unlock + \$0.30/min
8		880	8 Ebike - Month Pass	29.00	1 month	Ebike: Free unlock + \$0.15/min for 45 min
9		881	9 Ebike - Bay Wheels	150.00	1 year	Ebike: Free unlock + \$0.15/min for 45 min
10		882	10 Ebike - Lyft Pink	199.00	1 year	Ebike: Free unlock + \$0.15/min for 45 min

## 2. Dim\_Users (surrogate\_key : user\_dim\_id)



**pgAdmin 4**

**Object Explorer**

- Languages
- Publications
- Schemas (1)
  - public
    - Aggregates
    - Collations
    - Domains
    - FTS Configurations
    - FTS Dictionaries
    - FTS Parsers
    - FTS Templates
    - Foreign Tables
    - Functions
    - Materialized Views
    - Operators
    - Procedures
    - Sequences
  - Tables (12)
    - dim\_bike
    - dim\_bike\_insurance\_status
    - dim\_bike\_insurance\_status\_scd
    - dim\_city
    - dim\_date
    - dim\_maintenance
    - dim\_payment
    - dim\_stations
    - dim\_subscription
    - dim\_time
    - dim\_user\_subscription
    - dim\_users
  - Trigger Functions
  - Types
  - Views

**SQL**

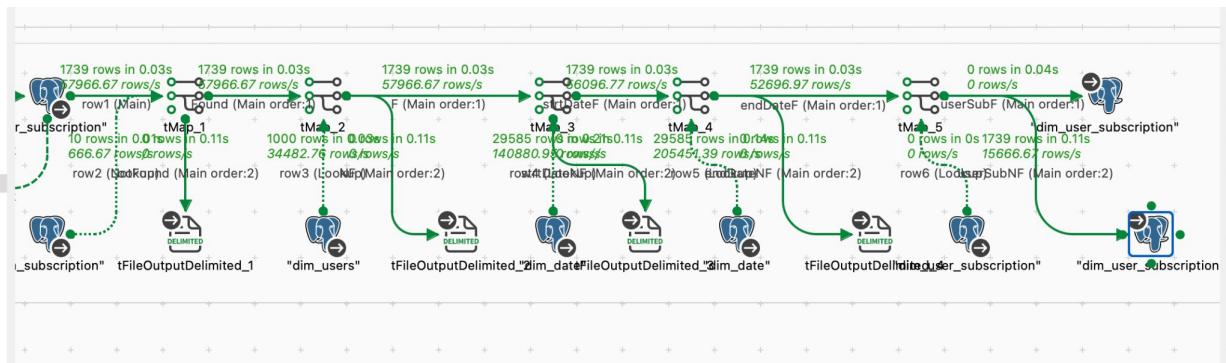
BikeFlowOLAP/postgres@PostgreSQL 17\*

**Data Output**

user_dim_id [PK] integer	user_id integer	first_name character varying(50)	last_name character varying(50)	email character varying(100)	phone_number character varying(20)	date_of_birth date	gender character varying(10)	account_created integer
1	3001	1 Diana	Rothert	drother0@technorati.com	727844608	1985-07-12	Female	
2	3002	2 Morganica	Beddin	mbeddin1@pagesperso-orange.fr	2355725276	1990-08-17	Female	
3	3003	3 Siouxie	Abrahams	sabrahams2@vk.com	5197018526	2011-12-04	Female	
4	3004	4 Marwin	Heamus	mheamus3@shareasale.com	474069737	2003-05-13	Male	
5	3005	5 Merwin	Pitcock	mpitcock4@topsy.com	1298497442	1988-12-28	Male	
6	3006	6 Sharlene	Somerville	ssomerville5@diegogogo.com	8403282478	2002-04-21	Female	
7	3007	7 Brent	MacConnelly	bmacconnelly6@freewebs.com	85001449	1979-03-28	Male	
8	3008	8 Noemi	Karpe	nkarpe7@china.com.cn	6235696477	2012-04-29	Female	
9	3009	9 Gertie	Greeveson	ggreeveson8@virginia.edu	344426262	1998-01-31	Female	
10	3010	10 Matty	Kislingbury	mkislingbury9@washington.edu	3169290649	2011-03-04	Male	
11	3011	11 Kirsti	Speller	kspeller@mit.edu	9506928800	1995-10-26	Female	
12	3012	12 Sophie	Harley	sharley@comsenz.com	6632208363	2004-10-15	Female	
13	3013	13 Karola	Abrahami	kabrahamic@alex.com	1012775461	1974-11-23	Female	
14	3014	14 Carlyne	Dempster	cdempster@google.co.jp	2517410615	2002-07-29	Female	
15	3015	15 Orsola	Worthing	oworthinge@blogs.com	1878564013	1999-10-08	Female	
16	3016	16 Thorsten	Paulusch	tpauluschf@umich.edu	801802946	1974-12-19	Male	
17	3017	17 Lauri	Lamminmaki	llamminmangi@51.la	85396567	1975-09-06	Female	
18	3018	18 Eavleen	Beggin	ebeggin@angelfire.com	822455250	2011-06-23	Female	
19	3019	19 Thaddeus	Tsimoneau	tsimoneau@indiatimes.com	9614410434	1994-06-10	Male	
20	3020	20 Ree	Crichmere	rcrichmere@constantcontact.com	4744881092	2008-01-27	Female	
21	3021	21 Alwyn	Harrod	aaharrod@dot.gov	8225301374	1993-08-14	Male	
22	3022	22 Perice	Valasek	pvalasek@mit.edu	3050813725	1982-09-09	Male	
23	3023	23 Kinnie	Snarr	ksnarrm@eventbrite.com	2057528065	2002-12-30	Male	
24	3024	24 Martelle	Rubenovic	mrubenovicn@twitter.com	728554178	1983-11-28	Female	

Total rows: 1000 of 1000    Query complete 00:00:00.144    Ln 184, Col 32

### 3. Dim\_User\_Subscription (surrogate\_key : user\_sub\_dim\_id)



**pgAdmin 4**

Object Explorer    SQL X BikeFlowOLTP/post... X BikeFlowOLAP/postgres@PostgreSQL 17\* X Processes X

Mon Nov 11 9:58PM

Object Explorer

- public
  - Aggregates
  - Collations
  - Domains
  - FTS Configurations
  - FTS Dictionaries
  - FTS Parsers
  - FTS Templates
  - Foreign Tables
  - Functions
  - Materialized Views
  - Operators
  - Procedures
  - Sequences
- Schemas (1)
  - public

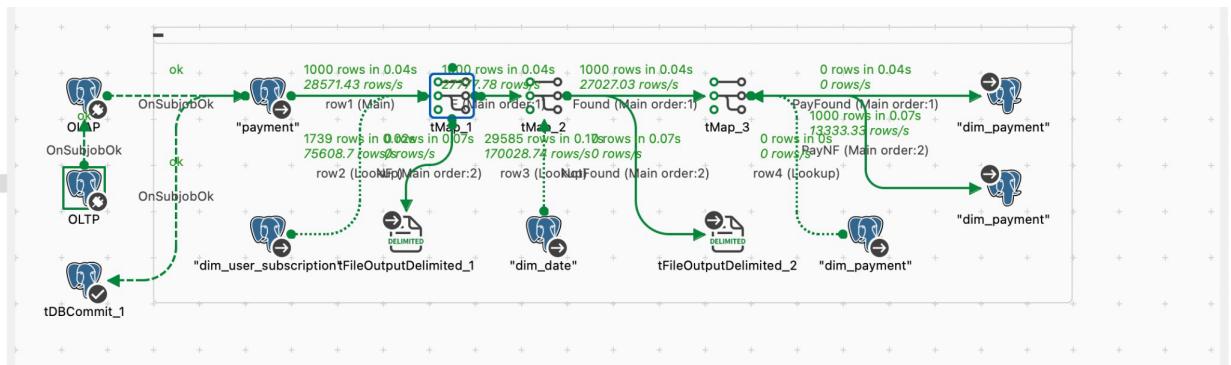
SQL

BikeFlowOLAP/postgres@PostgreSQL 17

	user_sub_dim_id	user_sub_id	user_id	subscription_id	start_date_id	end_date_id	auto_renew
1	1750	0	3001	879	27158	27158	false
2	1751	1	3002	877	26910	27276	true
3	1752	2	3004	876	26916	27282	true
4	1753	3	3004	878	27316	27316	false
5	1754	4	3005	880	27152	27183	false
6	1755	5	3006	880	27092	27123	true
7	1756	6	3006	873	27157	27157	true
8	1757	7	3006	875	27297	27327	false
9	1758	8	3007	882	26940	27306	false
10	1759	9	3009	878	27027	27027	true
11	1760	10	3009	881	27095	27460	false
12	1761	11	3010	878	27182	27182	true
13	1762	12	3010	879	27206	27236	true
14	1763	13	3010	882	27239	27604	false
15	1764	14	3011	880	27097	27128	true
16	1765	15	3011	880	27180	27210	true
17	1766	16	3011	881	27230	27595	false
18	1767	17	3013	874	26992	26992	false
19	1768	18	3013	882	27010	27376	false
20	1769	19	3014	879	27040	27040	true
21	1770	20	3014	873	27312	27312	true
22	1771	21	3015	882	26988	27354	false
23	1772	22	3016	880	26925	26955	false
24	1773	23	3016	879	27057	27057	true
25	1774	24	3016	873	27137	27137	false

Total rows: 1000 of 1739    Query complete 00:00:00.153    Ln 186, Col 44    Successfully

#### 4. Dim\_Payments (surrogate\_key : payment\_dim\_id)



pgAdmin 4 Object Tools Edit View Window Help pgAdmin 4 Mon Nov 11 9:59PM

Object Explorer SQL BikeFlowOLTP/postgres@BikeFlowOLTP PostgreSQL 17 Processes

Query History

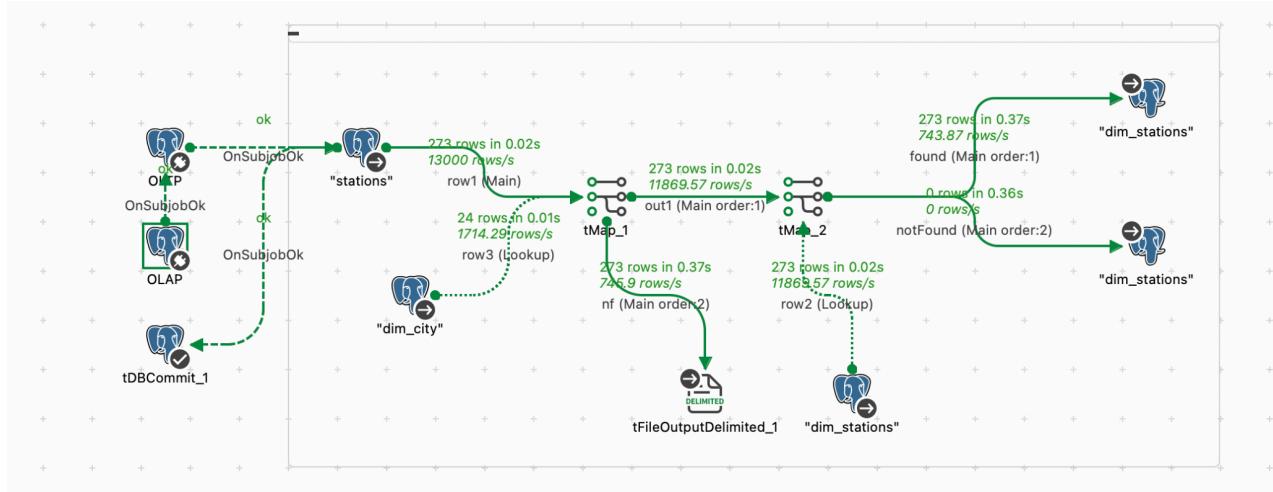
```
1.83 select * from public.dim_subscription ;
```

Data Output Messages Notifications

payment_dim_id	payment_id	user_sub_dim_id	amount	type	payment_method	payment_date_id
1	1001	1	2284	29.00	success	Gpay
2	1002	2	2144	199.00	success	Gpay
3	1003	3	3240	150.00	success	applepay
4	1004	4	2524	3.99	success	credit card
5	1005	5	2806	3.99	success	Gpay
6	1006	6	3438	15.00	success	applepay
7	1007	7	1777	15.00	success	debit card
8	1008	8	1903	150.00	success	applepay
9	1009	9	2908	199.00	success	credit card
10	1010	10	2932	15.00	failure	credit card
11	1011	11	2726	3.99	success	debit card
12	1012	12	3249	3.99	success	applepay
13	1013	13	2078	15.00	failure	debit card
14	1014	14	3055	15.00	success	Gpay
15	1015	15	3206	199.00	success	debit card
16	1016	16	2741	29.00	success	Gpay
17	1017	17	2971	29.00	success	credit card
18	1018	18	3329	3.99	success	credit card
19	1019	19	2616	150.00	success	debit card
20	1020	20	3249	3.99	success	applepay
21	1021	21	2863	199.00	success	debit card
22	1022	22	2861	15.00	success	credit card
23	1023	23	1857	199.00	success	applepay

Total rows: 1000 of 1000 Query complete 00:00:00.160 Ln 189, Col 34

## 5. Dim\_Stations (surrogate\_key : station\_dim\_id)

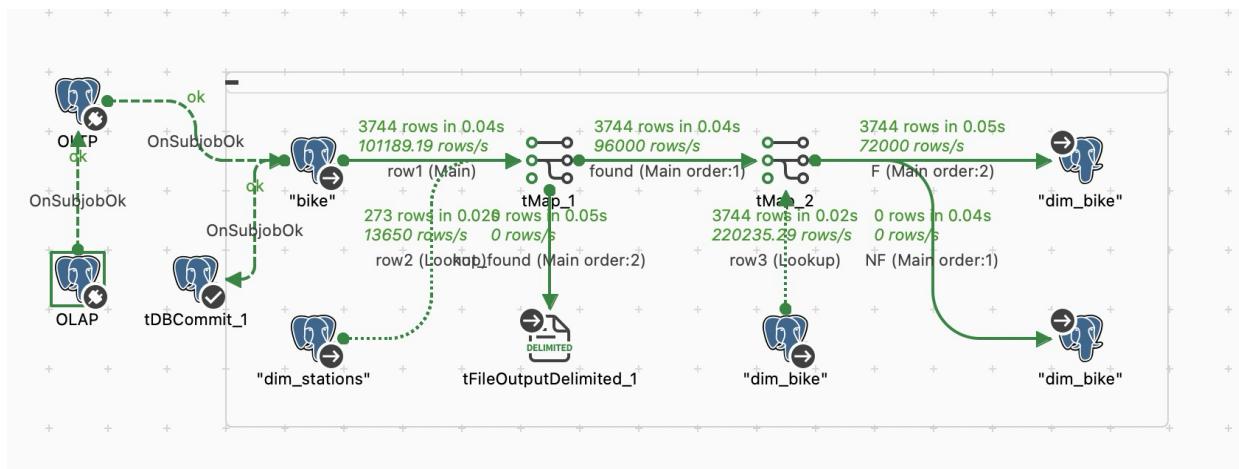


The screenshot shows the pgAdmin 4 interface with the 'BikeFlowOLAP' database selected. The 'Tables (12)' section is expanded, and the 'dim\_stations' table is selected. The table data is displayed in a grid format:

station_dim_id	station_id	station_location	station_lat	station_long	num_docks_available	num_bikes_available	city_id
1	883	72 Broadway at 30th St	37.8193814	-122.2619284	13	22	
2	884	73 Broadway at 40th St	37.8277573	-122.2567156	21	14	
3	885	74 Broadway at Battery St	37.79857211	-122.400869	30	5	
4	886	75 Broadway at Coronado Ave	37.8357883	-122.2516207	22	13	
5	887	76 Broadway at Kearny	37.79801364	-122.4059504	25	10	
6	888	77 Broderick St at Oak St	37.7730627	-122.4390777	26	9	
7	889	78 Bryant St at 15th St	37.7671004	-122.410662	27	8	
8	890	79 Bryant St at 6th St	37.77591022	-122.402575	18	17	
9	891	80 Bushrod Park	37.8465156	-122.2653043	19	16	
10	892	81 Cahill Park	37.32911867	-121.9045758	16	19	
11	893	82 California St at University Ave	37.87055533	-122.2797203	23	12	
12	894	83 Central Ave at Fell St	37.77331088	-122.4442926	25	10	
13	895	84 Cesar Chavez St at Dolores St	37.7478584	-122.4249863	20	15	
14	896	85 Channing Way at San Pablo Ave	37.8628271	-122.2902305	20	15	
15	897	86 Church St at Duboce Ave	37.7700831	-122.4291557	21	14	
16	898	87 Civic Center/UN Plaza BART Station (Market St at McAllister ...	37.7810737	-122.4117382	17	18	
17	899	88 Clay St at Battery St	37.795001	-122.39997	22	13	
18	900	89 Colin P Kelly Jr St at Townsend St (Temporary Site)	37.78138282	-122.3898411	20	15	
19	901	90 College Ave at Bryant Ave	37.8381269	-122.2512714	25	10	
20	902	91 College Ave at Harwood Ave	37.848152	-122.2521599	18	17	
21	903	92 College Ave at Taft Ave	37.8417999	-122.2515349	13	22	
22	904	93 Commercial St at Montgomery St	37.794231	-122.402923	11	24	
23	905	94 Cyril Magnin St at Ellis St	37.78588063	-122.408915	25	10	
24	906	95 Davis St at Jackson St	37.79728	-122.398436	13	22	

Total rows: 273 of 273    Query complete 00:00:00.155    Ln 182, Col 36

## 6. Dim\_bikes (surrogate\_key : bike\_dim\_id)



The screenshot shows the pgAdmin 4 interface with the following details:

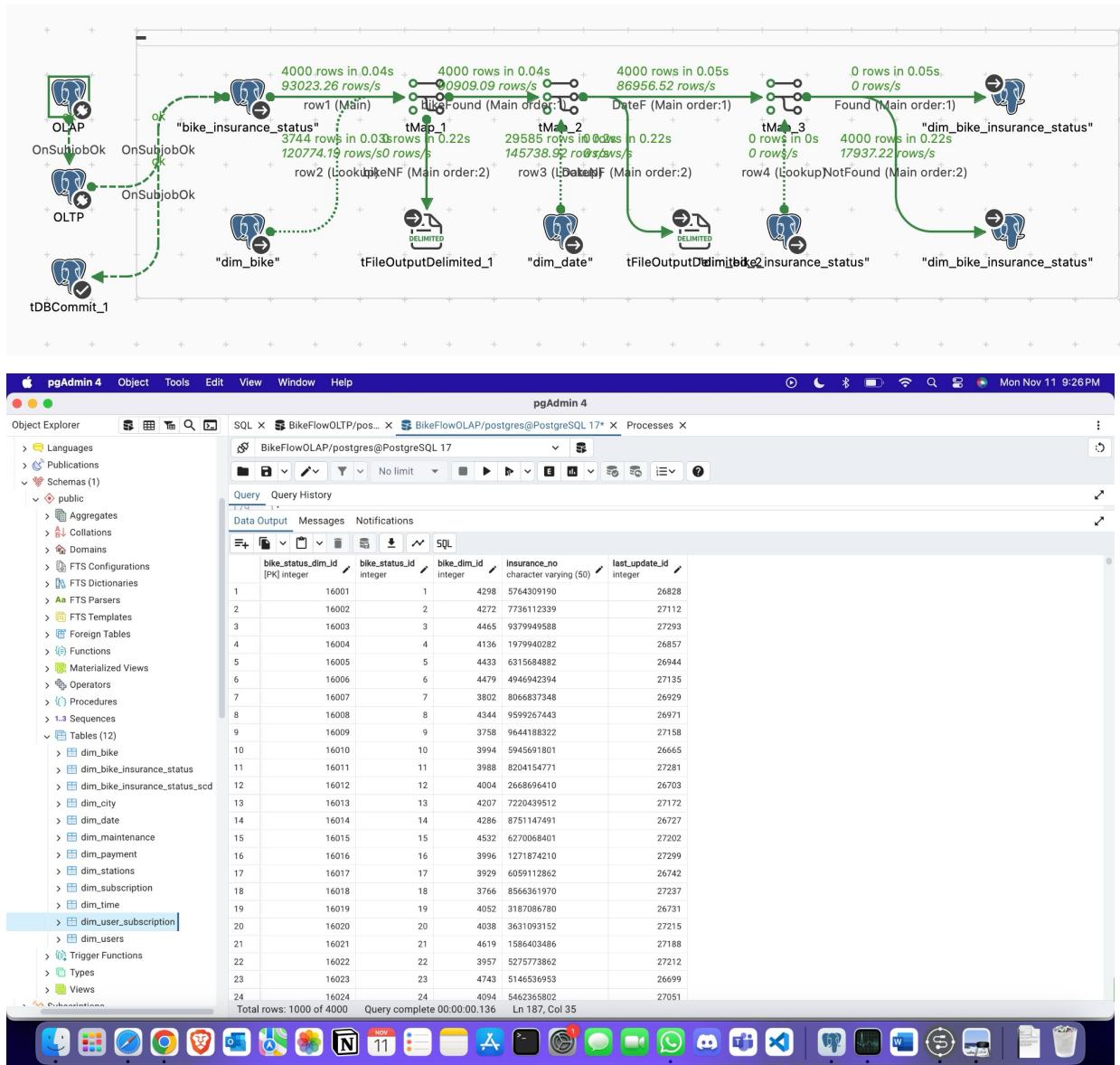
- Object Explorer:** Shows the database schema with various objects like Languages, Publications, Schemas, public, Aggregates, Collations, Domains, FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and multiple Tables (dim\_bike, dim\_bike\_insurance\_status, dim\_bike\_insurance\_status\_scd, dim\_city, dim\_date, dim\_maintenance, dim\_payment, dim\_stations, dim\_subscription, dim\_time, dim\_user\_subscription, dim\_users).
- SQL Editor:** The current connection is BikeFlowOLAP/postgres@PostgreSQL\_17\*. The query being run is:

bike_dim_id	bike_id	type	status	battery_status	last_station_id	
1	3745	1	Original Ebike	Maintenance	42	1128
2	3746	2	Classic Bike	Maintenance	40	1062
3	3747	3	Original Ebike	Maintenance	42	1055
4	3748	4	Nextgen Ebike	Maintenance	88	1075
5	3749	5	Nextgen Ebike	Maintenance	26	936
6	3750	6	Original Ebike	In-Use	71	1025
7	3751	7	Original Ebike	Available	44	913
8	3752	8	Classic Bike	Maintenance	42	892
9	3753	9	Nextgen Ebike	Available	7	957
10	3754	10	Scooter	In-Use	70	1092
11	3755	11	Classic Bike	Maintenance	87	1051
12	3756	12	Nextgen Ebike	Available	30	1107
13	3757	13	Original Ebike	In-Use	19	1075
14	3758	14	Classic Bike	Maintenance	94	1048
15	3759	15	Classic Bike	Maintenance	6	1044
16	3760	16	Nextgen Ebike	Available	8	884
17	3761	17	Scooter	In-Use	64	1036
18	3762	18	Classic Bike	Available	38	996
19	3763	19	Original Ebike	In-Use	99	923
20	3764	20	Nextgen Ebike	In-Use	50	937
21	3765	21	Classic Bike	Maintenance	46	978
22	3766	22	Scooter	Available	64	1054
23	3767	23	Original Ebike	In-Use	87	1103
24	3768	24	Scooter	Maintenance	36	1029
25	3769	25	Original Ebike	In-Use	63	895

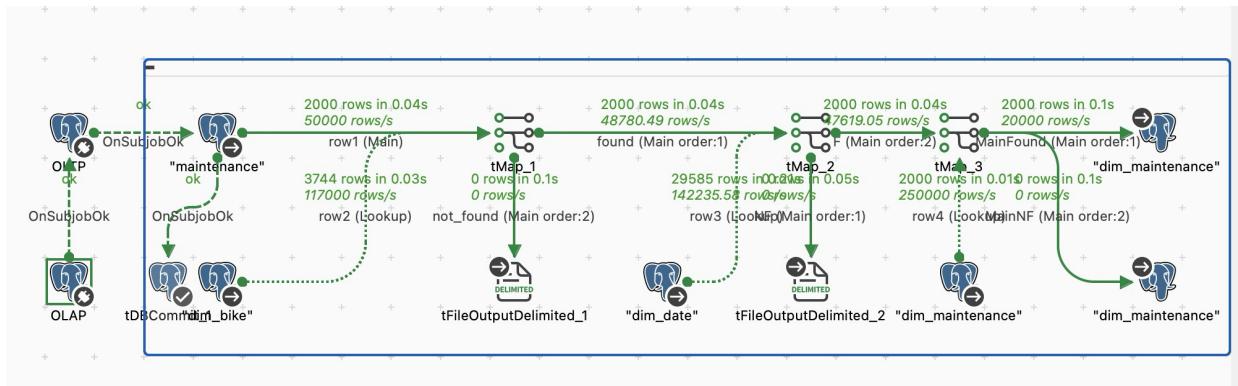
Total rows: 1000 of 3744 Query complete 00:00:00.158 Ln 181, Col 31

Successfully run. Total query runtime: 158 msec. 3744 rows affected.

## 7. Dim\_bike\_insurance\_status (surrogate\_key : bike\_status\_dim\_id)



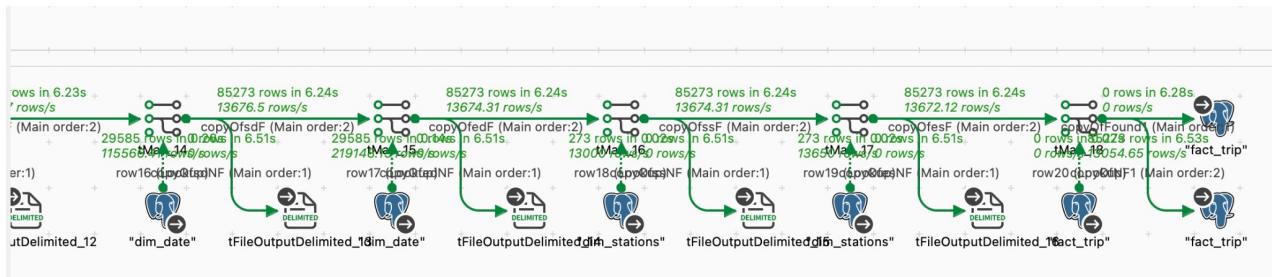
## 8. Dim\_bike\_maintenance(surrogate\_key : maintenance\_dim\_id)



maintenance_dim_id	maintenance_id	bike_dim_id	maintenance_date_id	issue	downtime_hours
1	2001	1	3771	27085 Motor Issues	5
2	2002	2	4085	27309 City Restrictions	1
3	2003	3	3815	27271 Bike Overuse or Shortage	4
4	2004	4	4110	27224 Broken Frame	2
5	2005	5	3903	27152 Improper Parking	4
6	2006	6	3924	27251 Broken Chain	2
7	2007	7	4206	27292 Improper Parking	3
8	2008	8	4689	27028 Accidents	9
9	2009	9	3917	27033 Battery Failure	10
10	2010	10	4607	27110 Overuse	8
11	2011	11	4241	27263 Charging Issues	4
12	2012	12	4357	27209 Broken Frame	10
13	2013	13	4315	26952 Insurance Issues	10
14	2014	14	4046	27297 Wear-out or Broken Pedals	1
15	2015	15	4717	27297 Accidents	5
16	2016	16	4294	27174 Low Availability at Stations	1
17	2017	17	3764	27264 Bike Overuse or Shortage	1
18	2018	18	4014	27074 Unreported Problems	5
19	2019	19	4009	27070 Insurance Issues	4
20	2020	20	4486	26933 Insurance Issues	2
21	2021	21	3837	27012 Motor Issues	7
22	2022	22	4285	27279 Brake Malfunction	2
??	20??	??	??	??	??

Total rows: 1000 of 2000 Query complete 00:00:00.135 Ln 188, Col 38

## 9. FACT\_TRIP (surrogate\_key : trip\_dim\_id)

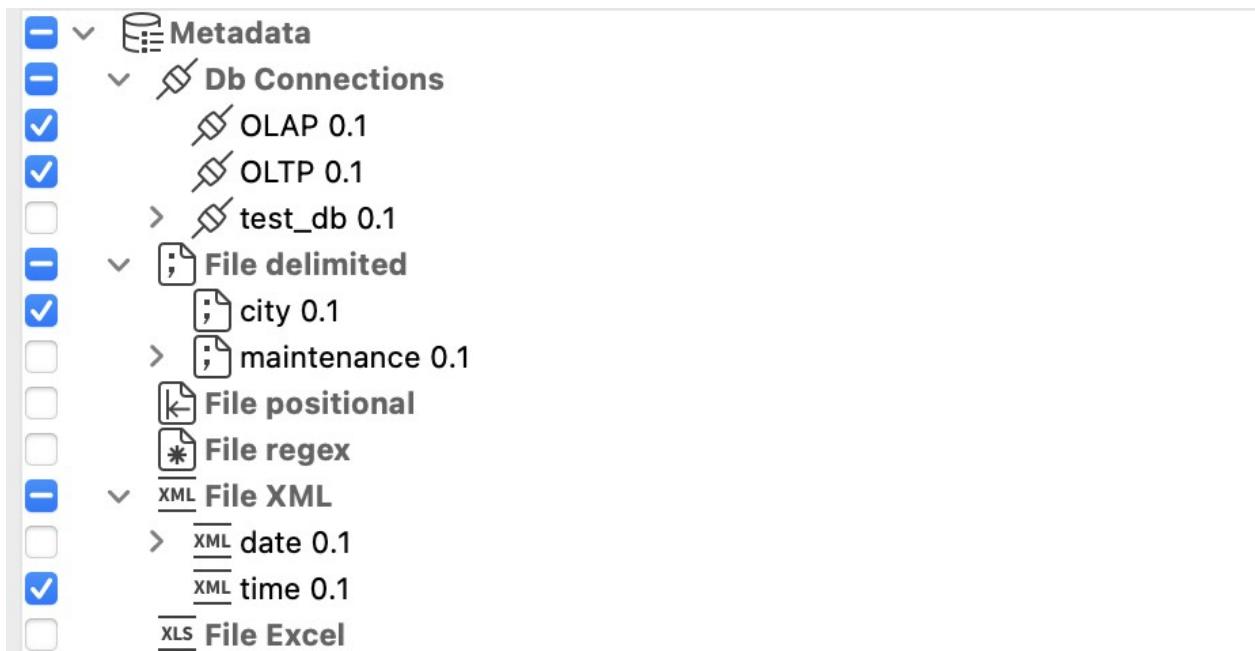


The screenshot shows the pgAdmin 4 interface with the following details:

- Object Explorer:** Shows the schema structure, including tables like dim\_bike, dim\_stations, and dim\_trip.
- SQL Tab:** A query window titled "BikeFlowOLTP/postgres@PostgreSQL 17\*" containing the SQL code for creating the FACT\_Trip table.
- Data Output Tab:** Displays the data for the newly created FACT\_Trip table, showing 1000 rows of data.
- Message Tab:** Shows the message "Query complete 00:00:00.134 Ln 237, Col 25".

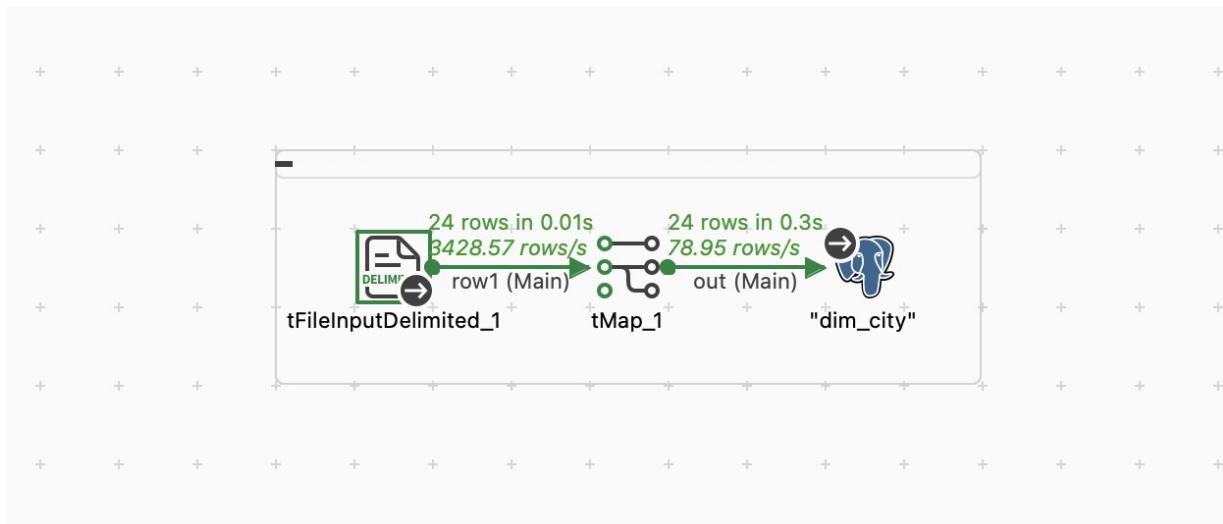
trip_dim_id	trip_id	user_dim_id	bike_dim_id	duration_sec	duration_min	start_time_id	end_time_id	start_date_id	end_date_id	start_station_id	end_station_id	
1	1	15644	3072	7000	275	4	66380	66656	24866	24868	1114	112'
2	2	70937	3934	4885	285	4	38717	39003	24847	24847	1114	112'
3	3	13094	3420	7093	684	11	53356	54040	24843	24843	1110	112'
4	4	13103	3086	7092	576	9	54729	55306	24865	24865	1110	112'
5	5	16019	3491	6988	519	8	54784	55303	24865	24865	1110	112'
6	6	16879	3431	6962	631	10	63738	64369	24847	24847	1110	112'
7	7	28141	3587	6614	461	7	54296	54758	24839	24839	1110	112'
8	8	49195	3314	5827	562	9	56991	57554	24850	24850	1110	112'
9	9	67864	3991	5027	684	11	36793	37478	24844	24844	1110	112'
10	10	70580	3236	4902	492	8	45766	46259	24852	24852	1110	112'
11	11	73318	3790	4798	637	10	48453	49090	24853	24853	1110	112'
12	12	76667	3212	4566	599	9	56964	57564	24850	24850	1110	112'
13	13	91938	3586	3904	449	7	43933	44383	24843	24843	1110	112'
14	14	22204	3993	6803	817	13	50866	51683	24850	24850	1102	112'
15	15	64913	3909	5159	682	11	35984	36667	24861	24861	1102	112'
16	16	67862	3882	5027	1872	31	31464	33337	24845	24846	1102	112'
17	17	51279	3874	5753	756	12	60798	61554	24841	24841	1099	112'
18	18	23666	3061	6759	1234	20	84786	86021	24867	24867	1097	112'
19	19	27659	3729	6629	2190	36	39756	41947	24848	24848	1096	112'
20	20	73057	3346	4805	2047	34	48524	50572	24843	24843	1096	112'
21	21	5050	3957	7350	1233	20	63399	64632	24867	24867	1094	112'
22	22	7920	3017	7259	971	16	67231	68202	24845	24845	1094	112'

## Data Sources:



## CSV:

### 1. Dim\_City



pgAdmin 4 Object Tools Edit View Window Help Mon Nov 11 9:20PM

Object Explorer SQL X BikeFlowOLTP/postgres X BikeFlowOLAP/postgres@PostgreSQL 17\* X Processes X

BikeFlowOLAP/postgres@PostgreSQL 17\* No limit

Query History Data Output Messages Notifications

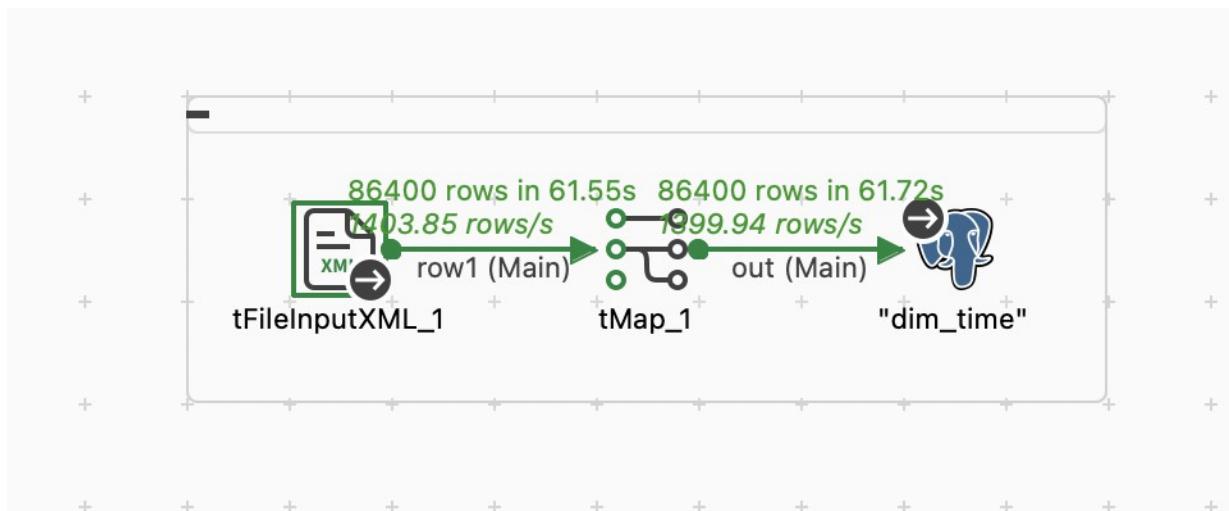
SQL

	city_id	city_name	state
1	1	San Francisco	California
2	2	San Diego	California
3	3	Sacramento	California
4	4	Fresno	California
5	5	Emeryville	California
6	6	Long Beach	California
7	7	Oakland	California
8	8	Bakersfield	California
9	9	Anaheim	California
10	10	Los Angeles	California
11	11	Santa Ana	California
12	12	Riverside	California
13	13	Stockton	California
14	14	Chula Vista	California
15	15	Irving	California
16	16	Berkeley	California
17	17	Modesto	California
18	18	Fontana	California
19	19	Moreno Valley	California
20	20	Alameda	California
21	21	Oxnard	California
22	22	Fremont	California
23	23	Glendale	California
24	24	San Jose	California

Total rows: 24 of 24 Query complete 00:00:00.136 Ln 191, Col 31

XML:

### 1. Dim\_Time



pgAdmin 4 Object Tools Edit View Window Help Mon Nov 11 9:21PM

Object Explorer SQL X BikeFlowOLTP/post... X BikeFlowOLAP/postgres@PostgreSQL 17\* X Processes X

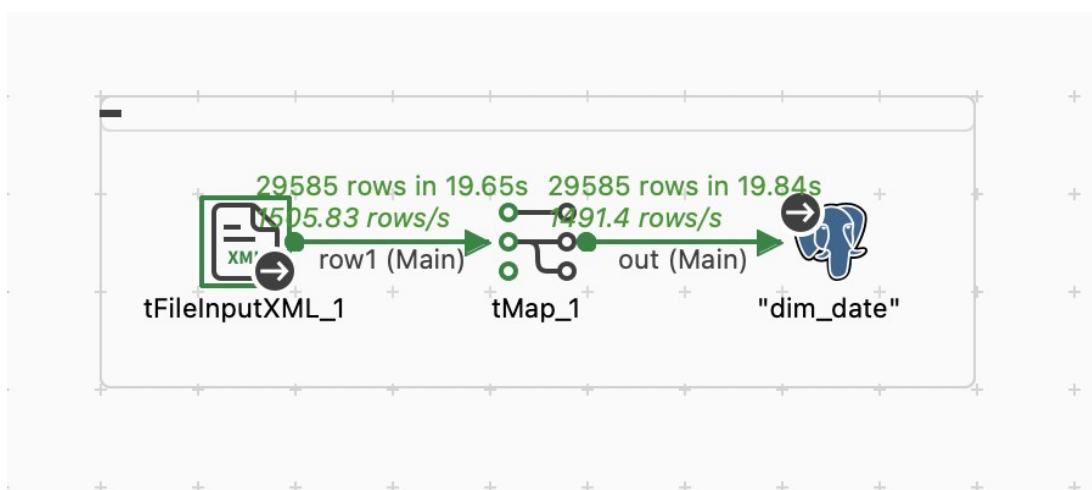
BikeFlowOLAP/postgres@PostgreSQL 17\* No limit

Query History Data Output Messages Notifications

time_id [PK] integer	time time without time zone	hour integer	minute integer	second integer
1	00:00:00	0	0	0
2	00:00:01	0	0	1
3	00:00:02	0	0	2
4	00:00:03	0	0	3
5	00:00:04	0	0	4
6	00:00:05	0	0	5
7	00:00:06	0	0	6
8	00:00:07	0	0	7
9	00:00:08	0	0	8
10	00:00:09	0	0	9
11	00:00:10	0	0	10
12	00:00:11	0	0	11
13	00:00:12	0	0	12
14	00:00:13	0	0	13
15	00:00:14	0	0	14
16	00:00:15	0	0	15
17	00:00:16	0	0	16
18	00:00:17	0	0	17
19	00:00:18	0	0	18
20	00:00:19	0	0	19
21	00:00:20	0	0	20
22	00:00:21	0	0	21
23	00:00:22	0	0	22
24	00:00:23	0	0	23

Total rows: 1000 of 86400 Query complete 00:00:00.147 Ln 190, Col 31

## 2. Dim\_Date



pgAdmin 4 Object Tools Edit View Window Help

BikeFlowOLAP/postgres@PostgreSQL 17\*

Object Explorer Schemas (1) public Tables (12) dim\_bike dim\_bike\_insurance\_status dim\_bike\_insurance\_status\_scd dim\_city dim\_date dim\_maintenance dim\_payment dim\_stations dim\_subscription dim\_time dim\_user\_subscription dim\_users Trigger Functions Types Views

Data Output Messages Notifications SQL

	date_id [PK] integer	full_date date	day_id integer	day_name character varying (10)	week_number integer	month_id integer	month_name character varying (10)	quarter integer	year integer
1	1	1950-01-01	1	Sunday	52	1	January	1	1950
2	2	1950-01-02	2	Monday	1	1	January	1	1950
3	3	1950-01-03	3	Tuesday	1	1	January	1	1950
4	4	1950-01-04	4	Wednesday	1	1	January	1	1950
5	5	1950-01-05	5	Thursday	1	1	January	1	1950
6	6	1950-01-06	6	Friday	1	1	January	1	1950
7	7	1950-01-07	7	Saturday	1	1	January	1	1950
8	8	1950-01-08	8	Sunday	1	1	January	1	1950
9	9	1950-01-09	9	Monday	2	1	January	1	1950
10	10	1950-01-10	10	Tuesday	2	1	January	1	1950
11	11	1950-01-11	11	Wednesday	2	1	January	1	1950
12	12	1950-01-12	12	Thursday	2	1	January	1	1950
13	13	1950-01-13	13	Friday	2	1	January	1	1950
14	14	1950-01-14	14	Saturday	2	1	January	1	1950
15	15	1950-01-15	15	Sunday	2	1	January	1	1950
16	16	1950-01-16	16	Monday	3	1	January	1	1950
17	17	1950-01-17	17	Tuesday	3	1	January	1	1950
18	18	1950-01-18	18	Wednesday	3	1	January	1	1950
19	19	1950-01-19	19	Thursday	3	1	January	1	1950
20	20	1950-01-20	20	Friday	3	1	January	1	1950
21	21	1950-01-21	21	Saturday	3	1	January	1	1950
22	22	1950-01-22	22	Sunday	3	1	January	1	1950
23	23	1950-01-23	23	Monday	4	1	January	1	1950
24	24	1950-01-24	24	Tuesday	4	1	January	1	1950
25	25	1950-01-25	25	Wednesday	4	1	January	1	1950

Total rows: 1000 of 29585 Query complete 00:00:00.136 Ln 192, Col 32

## Calculated Measure:

- Duration of the trip is converted to minutes

Talend Studio Talend Real-time Big Data Platform - tMap - tMap\_9

esF

Var Found1

Expression Builder

esF.duration\_sec / 60

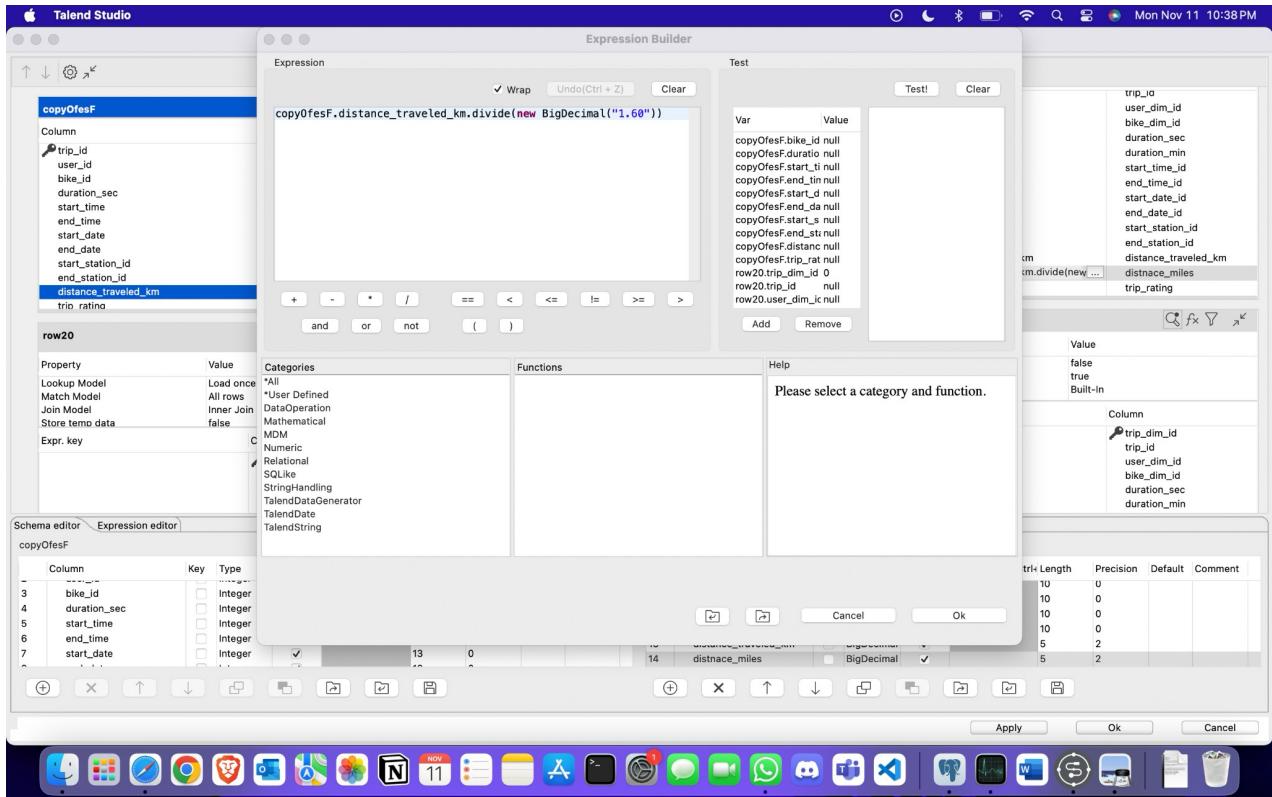
Categories Functions Help Please select a category and function.

Schema editor Expression editor esF

Column	Key	Type	Nullab	Date
3 bike_id		integer	✓	
4 duration_sec		integer	✓	
5 start_time		integer	✓	
6 end_time		integer	✓	
7 start_date		integer	✓	
8 end_date		integer	✓	
9 start_station_id		integer	✓	
10 end_station_id		integer	✓	
11 distance_traveled_km		double	✓	

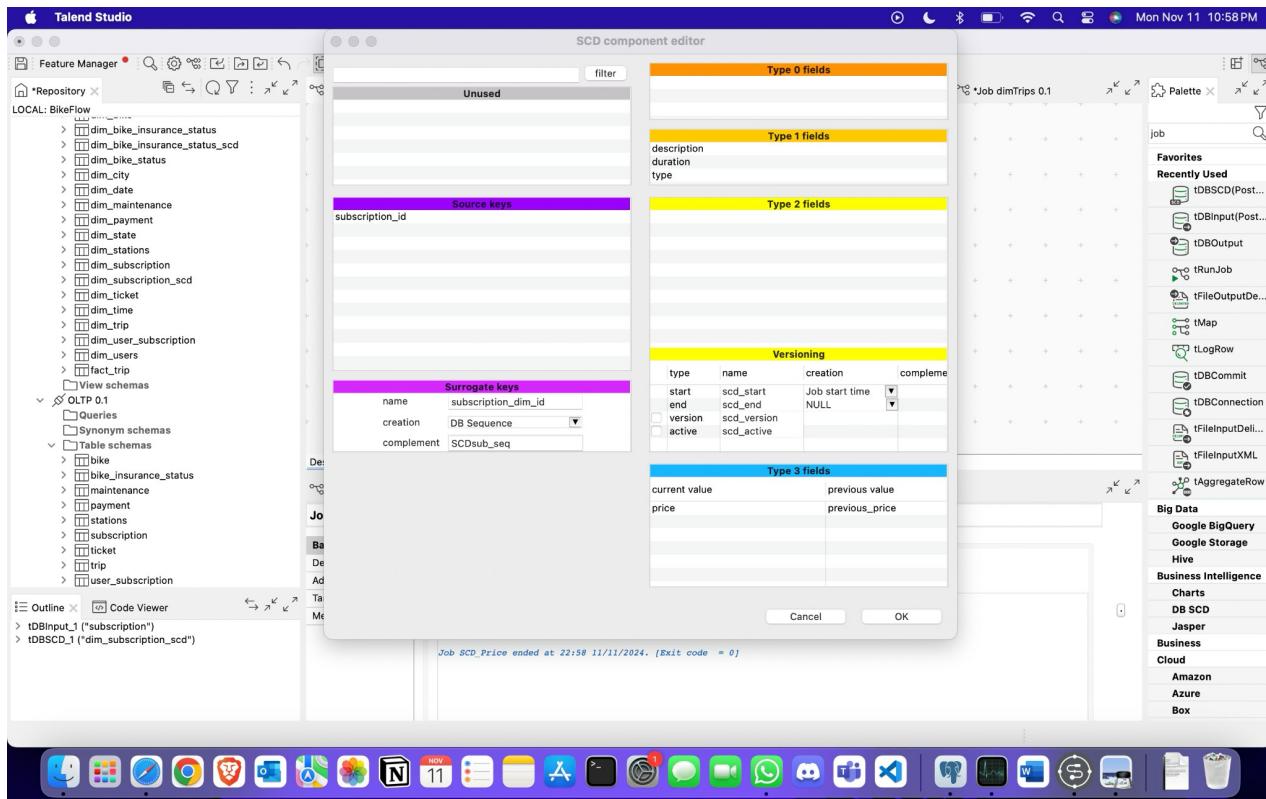
ib Date Pattern (Ctrl Length Precision Default Comment

## 2. Distance of the trip is converted to miles



SCD:

Type 3:



	subscription_dim_id [PK] integer	subscription_id integer	type character varying (50)	price numeric (10,2)	previous_price numeric (10,2)	duration character varying (100)	description character varying (100)
1		21	1 Single Ride	3.99	[null]	30 minutes	Classic bike: 30 min for \$3.99, then \$0.30/min
2		22	2 Day Pass	15.00	[null]	1 day	Classic bike: 30 min free, then \$0.30/min
3		23	3 Month Pass	29.00	[null]	1 month	Classic bike: 45 min free, then \$0.20/min
4		24	4 Bay Wheels	150.00	[null]	1 year	Classic bike: 45 min free, then \$0.20/min
5		25	5 Lyft Pink	199.00	[null]	1 year	Classic bike: 45 min free, then \$0.20/min
6		26	6 Ebike - Single Ride	3.99	[null]	30 minutes	Ebike: \$3.99 unlock + \$0.30/min
7		27	7 Ebike - Day Pass	15.00	[null]	1 day	Ebike: Free unlock + \$0.30/min
8		28	8 Ebike - Month Pass	29.00	[null]	1 month	Ebike: Free unlock + \$0.15/min for 45 min
9		29	9 Ebike - Bay Wheels	150.00	[null]	1 year	Ebike: Free unlock + \$0.15/min for 45 min
10		30	10 Ebike - Lyft Pink	199.00	[null]	1 year	Ebike: Free unlock + \$0.15/min for 45 min

pgAdmin 4 Object Tools Edit View Window Help

BikeFlowOLAP/postgres@PostgreSQL 17\* Mon Nov 11 11:05PM

Object Explorer

- domains
- FTS Configurations
- FTS Dictionaries
- FTS Parsers
- FTS Templates
- Foreign Tables
- Functions
- Materialized Views
- Operators
- Procedures
- Sequences
- Tables (14)
  - dim\_bike
  - dim\_bike\_insurance\_sta
  - dim\_bike\_insurance\_sts
  - dim\_city
  - dim\_date
  - dim\_maintenance
  - dim\_payment
  - dim\_stations
  - dim\_subscription
  - dim\_time
  - dim\_trip
  - dim\_user\_subscription
  - dim\_users
  - fact\_trip
- Trigger Functions
- Types
- Views
- Subscriptions
- BikeFlowOLTP
- Casts
- Catalogs
- Event Triggers

SQL > BikeFlowOLAP/postgres@PostgreSQL 17\* No limit

Query History

```

270
271   select * from public.dim_subscription_SCD ;
272
273
274   update public.dim_subscription_SCD set price=5.99 where subscription_id=1;
275
276
  
```

Data Output Messages Notifications

subscription_dim_id [PK] integer	subscription_id integer	type character varying (50)	price numeric (10,2)	previous_price numeric (10,2)	duration character varying (100)	description character varying (100)
1	22	2 Day Pass	15.00	[null]	1 day	Classic bike: 30 min free, then \$0.30/min
2	23	3 Month Pass	29.00	[null]	1 month	Classic bike: 45 min free, then \$0.20/min
3	24	4 Bay Wheels	150.00	[null]	1 year	Classic bike: 45 min free, then \$0.20/min
4	25	5 Lyft Pink	199.00	[null]	1 year	Classic bike: 45 min free, then \$0.20/min
5	26	6 Ebike - Single Ride	3.99	[null]	30 minutes	Ebike: \$3.99 unlock + \$0.30/min
6	27	7 Ebike - Day Pass	15.00	[null]	1 day	Ebike: Free unlock + \$0.30/min
7	28	8 Ebike - Month Pass	29.00	[null]	1 month	Ebike: Free unlock + \$0.15/min for 45 min
8	29	9 Ebike - Bay Wheels	150.00	[null]	1 year	Ebike: Free unlock + \$0.15/min for 45 min
9	30	10 Ebike - Lyft Pink	199.00	[null]	1 year	Ebike: Free unlock + \$0.15/min for 45 min
10	21	1 Single Ride	5.99	5.00	30 minutes	Classic bike: 30 min for \$3.99, then \$0.30/min

Total rows: 10 of 10 Query complete 00:00:00.099 Ln 271, Col 44

Transformations:

Changing Time formats:

Talend Studio Mon Nov 11 10:07 PM

Expression Builder

Expression: TalendDate.parseDate("HH:mm:ss", row1.time)

Test

Var	Value
row1.time_id	null
row1.time	null
row1.Hour	null
row1.Minute	null
row1.Second	null

Categories: All, User Defined, DateOperation, Mathematical, MDM, Numeric, Relational, SQL, StringHandling, TalendDataGenerator, TalendDate, TalendString

Functions:

Help: Please select a category and function.

Schema editor

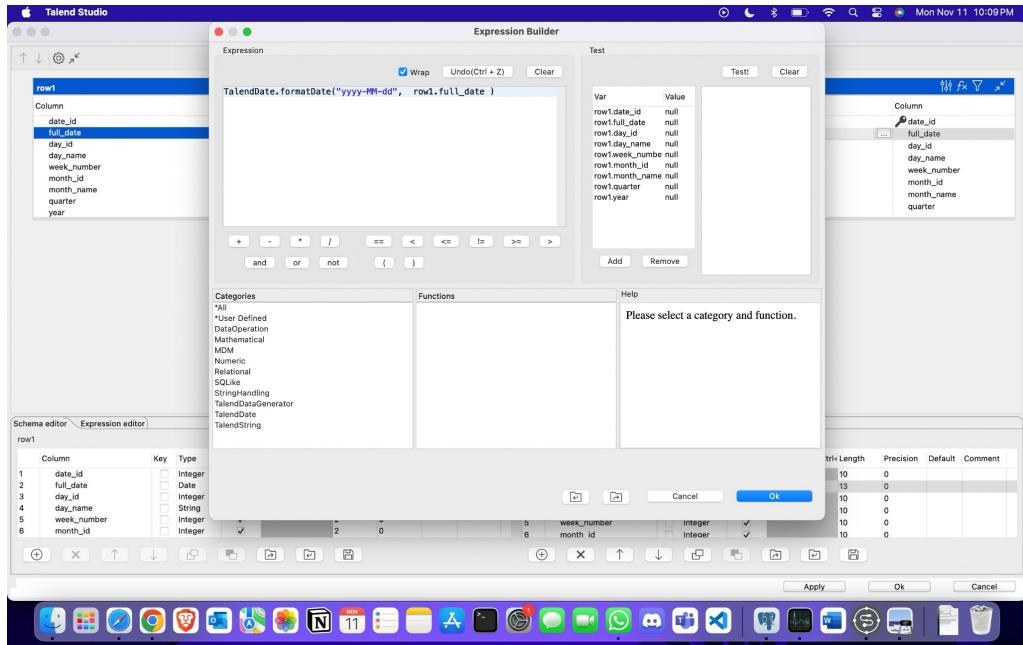
Column	Key	Type
1 time_id		Integer
2 time		String
3 Hour		Integer
4 Minute		Integer
5 Second		Integer

Expression editor

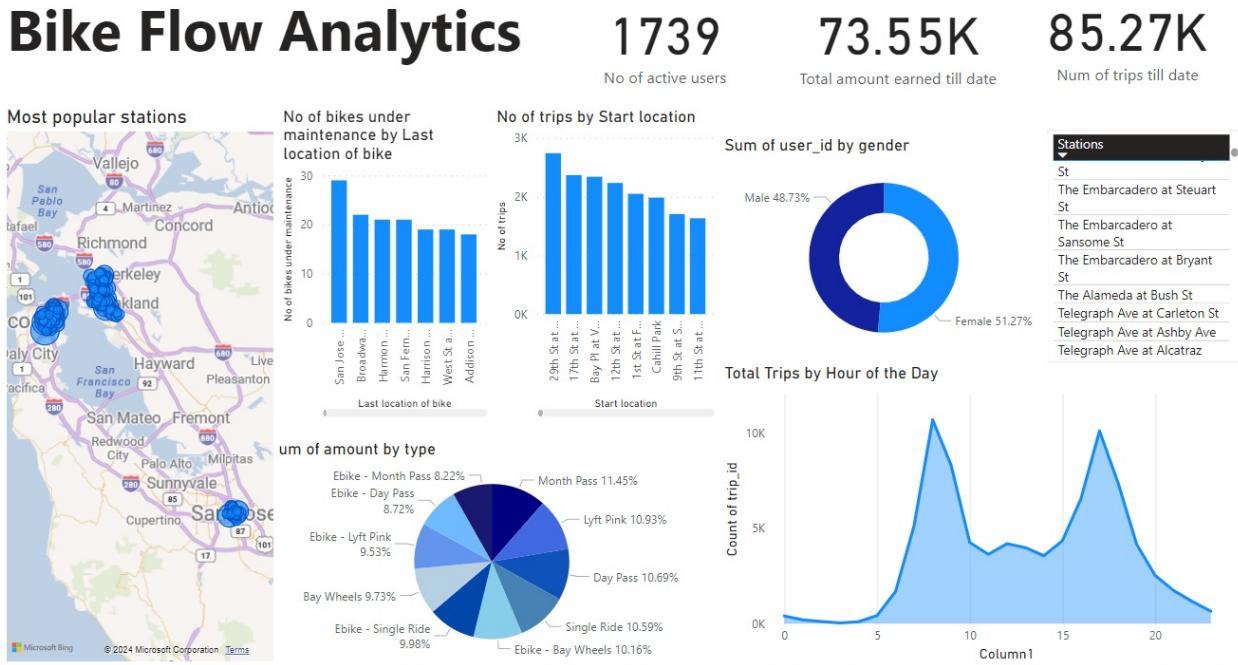
Column	Key	Type
1 time_id		Integer
2 time		String
3 Hour		Integer
4 Minute		Integer
5 Second		Integer

Ok Cancel

Changing Date Formats:



## DASHBOARD:



**GOAL** - The primary goals are to optimize bike availability during peak hours, strategically position bikes at high-demand stations, enhance operational efficiency through maintenance tracking, increase revenue via targeted marketing informed by user demographics, and improve overall user satisfaction by leveraging actionable insights from data analysis.

## KPI's

### Number of Active Users:

- 1,739 active users.
- Indicates overall user engagement and adoption of the service.

### Total Revenue Generated:

- \$73.55K total revenue to date.
- Measures financial performance and service profitability.

### Total Trips Taken:

- 85.27K trips to date.
- Reflects service utilization and user activity.

### Gender Distribution:

- Male: 48.73%, Female: 51.27%.
- Provides insights into the demographic breakdown of users.

### Peak Usage Hours:

- Total trips by the hour of the day, with distinct peaks during high-demand periods.
- Highlights key time windows for operational optimization.

### Most Popular Stations:

- Interactive map identifies high-traffic locations, such as Berkeley and San Francisco.
- Aids in strategic placement of bikes to maximize usage.

## Metrics:

### 1. Interactive Map:

- Clusters stations by popularity.
- High-density usage areas like Berkeley and San Francisco are clearly highlighted.

### 2. Maintenance Overview:

- Bar chart showing the number of bikes under maintenance categorized by the last known location.
- Supports tracking and management of service maintenance needs.

### 3. Trip Start Locations:

- Bar chart illustrating the volume of trips initiated from various start points.
- Pinpoints areas with the highest demand for services.

### 4. Revenue Breakdown by Bike Type:

- Pie chart summarizing revenue by bike type and pass type.
- Categories include Ebike (e.g., Month Pass: 11.45%, Day Pass: 8.22%) and Bay Wheels.

### 5. Hourly Trip Trends:

- Line chart representing total trips taken by the hour of the day.
- Helps identify peak and off-peak usage patterns for staffing and bike redistribution.

### 6. Filters and Drill-Through Features:

- Allows slicing data by station, bike type, or time period.
- Drill-through capabilities enable deeper insights into specific metrics like station performance or bike type preferences.

## Insights:

### 1. Peak Usage Identification:

- The line chart highlights morning (8-10 AM) and evening (4-6 PM) peaks, suggesting commuter-driven usage patterns. This indicates a need for increased bike availability during these times.

### 2. Popular Station Insights:

- Popular stations like "The Embarcadero at Steuart St" and others in San Francisco and Berkeley highlight areas requiring more bikes to meet demand.
- 3. **User Demographics:**
  - Assists in creating targeted marketing campaigns.
- 4. **Maintenance Tracking:**
  - Certain locations have a high number of bikes under maintenance, signaling potential inefficiencies in repair and redistribution processes.
- 5. **Revenue Optimization:**
  - Revenue is driven by short-term usage, as evidenced by high percentages for single rides and day passes, providing opportunities for promoting monthly pass subscriptions to enhance user retention.
- 6. **Demographics:**
  - A nearly equal gender distribution (Male: 48.73%, Female: 51.27%) shows balanced user engagement, indicating potential for gender-neutral marketing strategies.
- 7. **Bike Type Preference:**
  - E-bikes contribute significantly to trips and revenue, suggesting increased investment in e-bike availability and maintenance could boost growth.

# NYC RESTAURANT INSPECTION ANALYTICS

## Problem Statement:

The New York City Restaurant Inspection Results dataset faces challenges in maintaining food safety standards across its network of restaurants. Some establishments frequently fail inspections, while others consistently pass, leading to public health concerns and inefficiencies in the inspection system. Currently, the Department of Health and Mental Hygiene (DOHMH) does not have a consolidated way to analyze key factors like violation patterns, peak inspection times, and restaurant behavior to better manage food safety inspections.

To address this, we will create a centralized on-premise database that collects inspection data, violation details, and restaurant information. This will help the DOHMH analyze patterns, improve inspection allocation, and optimize restaurant management, ensuring food safety standards are maintained where and when they are needed most.

## Database Overview:

The operational database for the NYC Restaurant Inspection Results is designed to collect and organize key information from every inspection, allowing for detailed analysis and decision-making. Each column in the database captures specific aspects of the restaurant inspection system, which can be leveraged to optimize operations and improve public health outcomes.

## Dataset:

<https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/43nn-pn8j/data>

## Columns:

- CAMIS: Unique identifier for each restaurant, useful for tracking inspection history and patterns.
- DBA: "Doing Business As" name of the restaurant, essential for identifying specific establishments.
- BORO: Borough where the restaurant is located, used for geographic analysis of inspection results.
- BUILDING / STREET / ZIPCODE: Address details of the restaurant, important for location-based analysis.
- PHONE: Contact information for the restaurant, useful for follow-up communications.
- CUISINE DESCRIPTION: Type of cuisine served, helping to analyze patterns across different food types.
- INSPECTION DATE: Date of the inspection, essential for identifying trends over time.
- ACTION: Outcome of the inspection (e.g., violations cited, establishment closed), crucial for tracking enforcement actions.
- VIOLATION CODE / VIOLATION DESCRIPTION: Specific health code violations found during inspection, key for analyzing common issues.
- CRITICAL FLAG: Indicates whether a violation is critical or not, important for prioritizing health risks.
- SCORE: Numerical score assigned based on violations, useful for quantitative analysis of restaurant performance.
- GRADE: Letter grade assigned based on the inspection score, essential for public communication of food safety standards.
- GRADE DATE: Date when the grade was issued, helpful for tracking grade history.
- RECORD DATE: Date when the record was added to the database, important for data management.
- INSPECTION TYPE: Indicates the type of inspection conducted, useful for categorizing different inspection scenarios.

We have an extensive dataset that covers key aspects such as inspection dates, violation details, restaurant information, and grading history

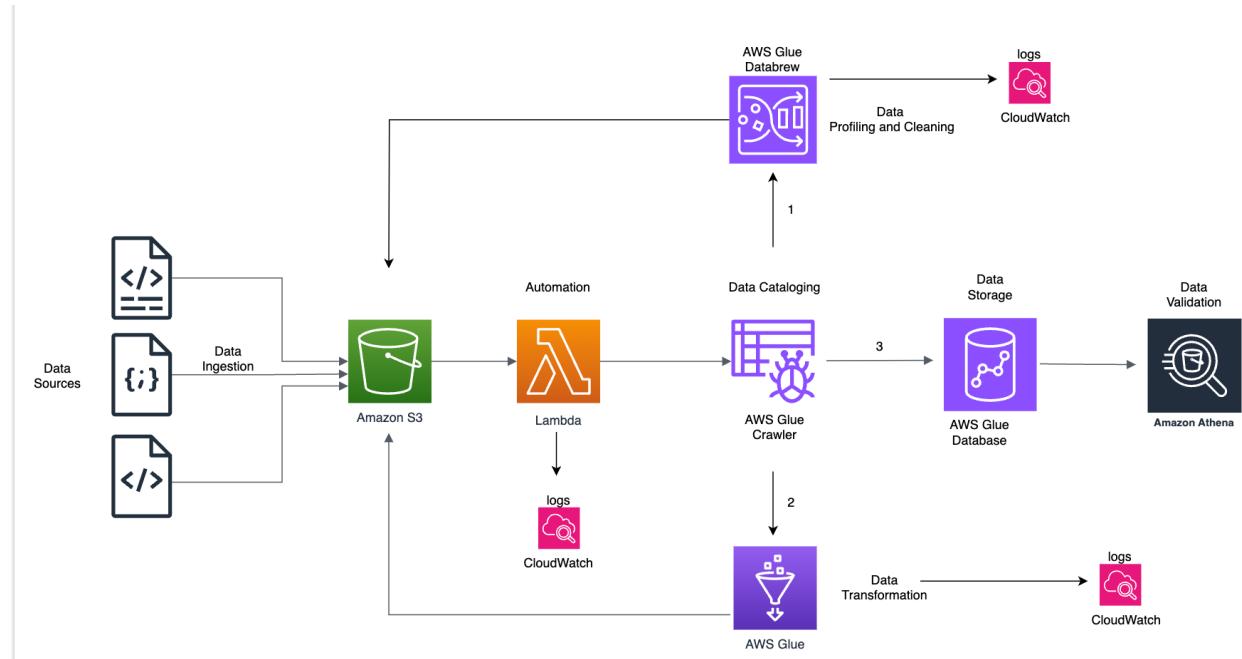
### What We Hope to Analyze:

In the future, we hope to analyze restaurant behavior patterns to gain insights into violation trends, peak inspection times, and cuisine-specific issues for NYC restaurants. This analysis will help optimize inspection management by assessing violation distribution and inspection frequency needs, ensuring food safety standards are maintained where they are needed most.

Additionally, we aim to evaluate the effectiveness of the grading system, measuring its impact on restaurant compliance and public health outcomes. Overall, these insights will guide improvements in inspection service delivery and support the DOHMH's commitment to enhancing food safety in New York City.

- **Optimize Inspection Allocation:** Predict high-risk restaurants and recommend prioritization of inspections across establishments.
- **Peak Time Analysis:** Identify the busiest times of year/week for violations to optimize inspection scheduling and staffing.
- **Restaurant Behavior:** Analyze differences between consistently compliant restaurants and frequent violators to tailor education and enforcement strategies.
- **Violation Pattern Analysis:** Evaluate which types of violations are most common and plan for targeted education and enforcement efforts.

### Data Pipeline Architecture:



## Data Sources:

- **Nyc\_Inspection.csv**
- **Date.json**

## AWS Tools:

- **S3**
- **Glue**
- **Glue DataBrew**
- **Lambda**
- **Athena**
- **Cloudwatch**

## Programming Language Used:

- **Python**

## PIPELINE:

### 1. Upload File to S3/de-cloud-project/input

- **AWS Service: Amazon S3**
- The user uploads raw data files to the S3/de-cloud-project/input bucket. These could be CSV, JSON, or other types of raw data files that need to be processed through the pipeline.
- The input data is stored in an S3 bucket as a starting point for processing.
- Nyc\_Inspection.csv uploaded to S3/de-cloud-project/input
- Date.json uploaded to S3/de-cloud-project/date

The screenshot shows the AWS S3 console interface. The left sidebar shows navigation links for Buckets, Access Grants, Access Points, Object Lambda Access Points, Multi-Region Access Points, Batch Operations, IAM Access Analyzer for S3, Block Public Access settings for this account, Storage Lens (Dashboards, Storage Lens groups, AWS Organizations settings), and a Feature spotlight. The main content area displays the 'Objects' tab for the 'de-cloud-project' bucket. It shows 8 objects: athena/, cleaned-archived/, cleaned/, date/, input-archived/, input/, transformed-archived/, and transformed/. The 'Actions' dropdown menu is open, showing options like Copy S3 URI, Copy URL, Download, Open, Delete, Actions (selected), Create folder, and Upload. The top right corner shows the user's name and session information.

Name	Type	Last modified	Size	Storage class
athena/	Folder	-	-	-
cleaned-archived/	Folder	-	-	-
cleaned/	Folder	-	-	-
date/	Folder	-	-	-
input-archived/	Folder	-	-	-
input/	Folder	-	-	-
transformed-archived/	Folder	-	-	-
transformed/	Folder	-	-	-

### 2. Data Crawler Job on S3/Bucket/Input

- **AWS Service: AWS Glue Crawler**

- The AWS Glue Crawler is responsible for discovering and cataloging the data in the S3/Bucket/Input. It inspects the files in the bucket, identifies their schema, and creates a catalog of the raw data.
- Create a catalog of the raw data in S3/de-cloud-project/input to be used for further processing and transformations.
- This step ensures that the metadata of the input data is available for any cleaning job that needs to run next

The screenshot shows the AWS Glue Table Overview page for a table named 'input'. The 'Table details' section includes:

- Name:** input
- Classification:** CSV
- Location:** s3://de-cloud-project/input/
- Description:** -
- Connection:** -
- Deprecated:** -
- Column statistics:** No statistics

The 'Last updated' field shows November 29, 2024 at 02:07:48. Below this, the 'Advanced properties' section is collapsed.

The 'Schema' tab is selected, showing a single column named 'camis' with a data type of 'bigint'. The schema table has the following structure:

#	Column name	Data type	Partition key	Comment
1	camis	bigint	-	-

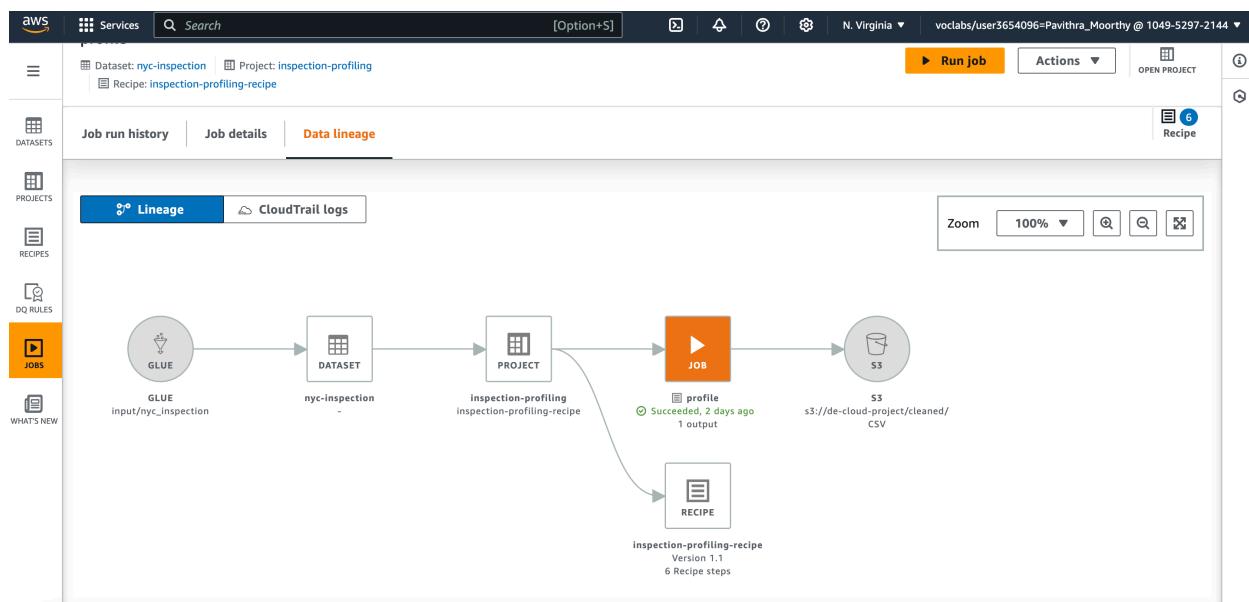
Buttons for 'Edit schema as JSON' and 'Edit schema' are located at the top right of the schema table.

### 3. Data Brewer Job on S3/Bucket/Input Catalog

- **AWS Service: AWS Glue Databrew Job**
- The Data Brewer Job processes the raw data catalog created by the Glue crawler. This job applies profiling and cleaning to the raw data and stores the output in S3/de-cloud-project/cleaned.
- This job can include tasks such as filtering out unnecessary data, correcting formats, removing errors

**inspection-profiling**

Project name inspection-profiling	Input dataset nyc-inspection DATA-CATALOG
Recipe name inspection-profiling-recipe	
Recipe steps (6)	
1. Delete column Location Point1 2. Replace text 0 with Unknown in BORO 3. Fill missing values with Unknown in CUISINE DESCRIPTION 4. Change format of INSPECTION DATE to yyyy-mm-dd 5. Change format of GRADE DATE to yyyy-mm-dd 6. Change format of RECORD DATE to yyyy-mm-dd	
<a href="#">Open project</a> <a href="#">Close</a>	



#### 4. Output Cleaned Data to S3/Bucket/Clean

- **AWS Service: Amazon S3**
- The clean data is stored in S3/Bucket/Clean, where it is now available for further processing or analysis.
- Store the cleaned data after processing.

Amazon S3 < Buckets > de-cloud-project

**Objects** | Properties | Permissions | Metrics | Management | Access Points

**Objects (8) Info**

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Name	Type	Last modified	Size	Storage class
athena/	Folder	-	-	-
cleaned-archived/	Folder	-	-	-
cleaned/	Folder	-	-	-
date/	Folder	-	-	-
input-archived/	Folder	-	-	-
input/	Folder	-	-	-
transformed-archived/	Folder	-	-	-
transformed/	Folder	-	-	-

## 5. Data Crawler Job on S3/Bucket/Clean

- **AWS Service: AWS Glue Crawler**
- The AWS Glue Crawler scans the cleaned data in S3/Bucket/Clean, updating or creating a new catalog in the Glue Data Catalog. It inspects the data format and structure of the cleaned data to ensure it is ready for further processing.
- Create a metadata catalog of the cleaned data to be used for the ETL jobs.
- This step ensures that the metadata of the cleaned data is available for any ETL job that needs to run next.

AWS Glue < Tables > cleaned

**cleaned**

Last updated (UTC) December 1, 2024 at 00:02:08 | Version 3 (Current version) | Actions

**Table overview** | Data quality - new

**Table details**

Name	cleaned	Classification	CSV
Database	nyc_inspection	Location	s3://de-cloud-project/cleaned/
Description	-	Connection	-
Last updated	November 29, 2024 at 05:12:33	Deprecated	-
<b>Advanced properties</b>			

**Schema** | Partitions | Indexes | Column statistics - new

**Schema (27)**

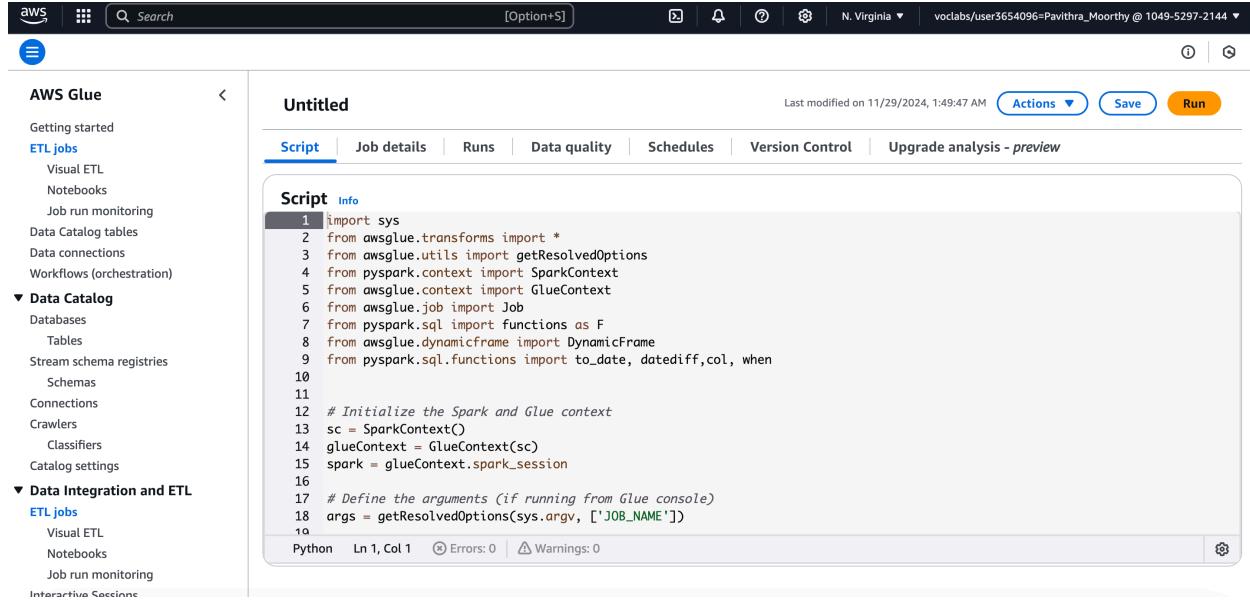
Edit schema as JSON | Edit schema

Filter schemas

## 6. Run ETL Job on Data from S3/Bucket/Clean

- **AWS Service: AWS Glue ETL Jobs**
- The ETL Job runs on the cleaned data cataloged by the Glue Crawler. This job performs Extract, Transform, and Load operations to transform the cleaned data into its final form.
- Transform the cleaned data into a format suitable for analytics, reporting, or machine learning, and store the results in the S3/Bucket/Transformed.
- Cleaned data and dates.json is combined

- Necessary fields are kept, others removed
- New measures (days took to give the results) are calculated



The screenshot shows the AWS Glue Script Editor interface. On the left, there's a navigation sidebar with sections like 'Getting started', 'ETL jobs' (selected), 'Data Catalog', 'Data Integration and ETL', and 'Interactive Sessions'. The main area is titled 'Untitled' and contains a Python script:

```

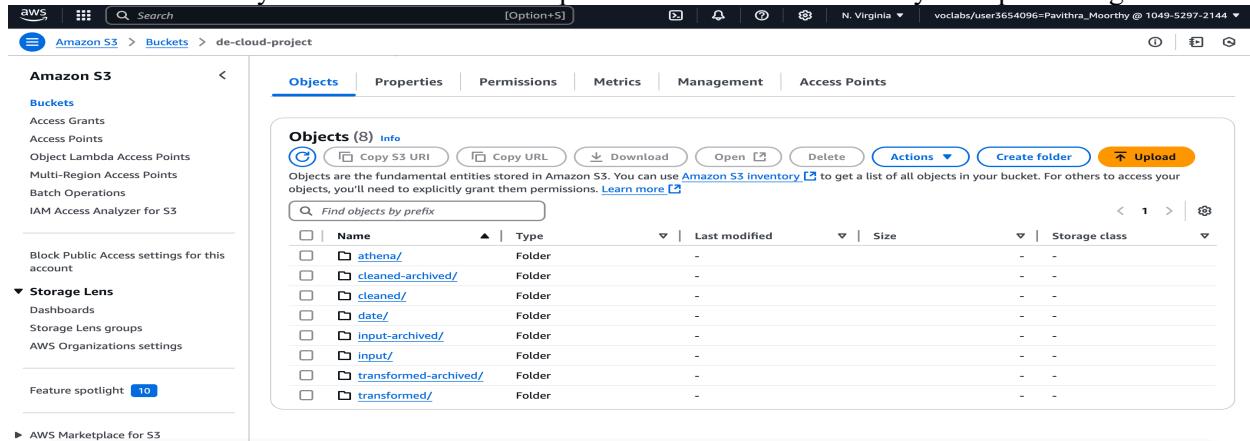
1 import sys
2 from awsglue.transforms import *
3 from awsglue.utils import getResolvedOptions
4 from pyspark.context import SparkContext
5 from awsglue.context import GlueContext
6 from awsglue.job import Job
7 from pyspark.sql import functions as F
8 from awsglue.dynamicframe import DynamicFrame
9 from pyspark.sql.functions import to_date, datediff, col, when
10
11
12 # Initialize the Spark and Glue context
13 sc = SparkContext()
14 glueContext = GlueContext(sc)
15 spark = glueContext.spark_session
16
17 # Define the arguments (if running from Glue console)
18 args = getResolvedOptions(sys.argv, ['JOB_NAME'])
19

```

Below the code editor, it says 'Python Ln 1, Col 1' and shows 'Errors: 0' and 'Warnings: 0'.

## 7. Output Transformed Data to S3/Bucket/Transformed

- **AWS Service: Amazon S3**
- The ETL process stores the transformed data in S3/Bucket/Transformed. This is the final processed data that is ready for analytics or further processing.
- Store the fully transformed data in a separate S3 bucket for final analysis or processing.



The screenshot shows the AWS S3 Buckets page. The left sidebar includes 'Buckets', 'Object Lambda Access Points', 'Multi-Region Access Points', 'Batch Operations', 'IAM Access Analyzer for S3', 'Block Public Access settings for this account', 'Storage Lens', and 'AWS Organizations settings'. The main area shows a table of objects in the 'de-cloud-project' bucket:

Name	Type	Last modified	Size	Storage class
athena/	Folder	-	-	-
cleaned-archived/	Folder	-	-	-
cleaned/	Folder	-	-	-
date/	Folder	-	-	-
input-archived/	Folder	-	-	-
input/	Folder	-	-	-
transformed-archived/	Folder	-	-	-
transformed/	Folder	-	-	-

## 8. Create Glue Crawler and Table for S3/Bucket/Transformed Data

- **AWS Service: AWS Glue Crawler**
- The **AWS Glue Crawler** is invoked to scan the transformed data in the S3/Bucket/Transformed creating a new catalog in the Glue Data Catalog.
- Create a metadata catalog of the cleaned data to be used for the creating tables.

The screenshot shows the AWS Glue Table Overview page for the 'inspection\_table'. The left sidebar includes sections for Getting started, ETL jobs, Visual ETL, Notebooks, Job run monitoring, Data Catalog tables, Data connections, Workflows (orchestration), Data Catalog (Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings), and Data Integration and ETL (ETL jobs, Visual ETL, Notebooks, Job run monitoring). The main content area displays the 'Table details' for 'inspection\_table', showing its Name (inspection\_table), Classification (-), Location (s3://de-cloud-project/transformed/), and Connection (-). It also shows the last updated date (November 29, 2024 at 07:04:17). Below this is the 'Advanced properties' section. At the bottom, there are tabs for Schema, Partitions, Indexes, and Column statistics - new. The Schema tab is selected, showing a table structure with columns: #, Column name, Data type, Partition key, and Comment. A search bar labeled 'Filter schemas' is present. Buttons for 'Edit schema as JSON' and 'Edit schema' are located at the top right of the schema table.

## 9. Querying using Athena

- **AWS Service: AWS Athena**
- To validate and query the database

The screenshot shows the Amazon Athena Query editor. On the left, the sidebar lists the database 'nyc\_inspection\_db' and the table 'inspection\_table'. The main area contains a SQL query window with the following content:

```

6
7
8
9
10
11
12
13
14
15 select * from inspection_table;
SQL Ln 15, Col 32

```

Below the query window are buttons for Run, Explain, Cancel, Clear, and Create. A checkbox for 'Reuse query results up to 60 minutes ago' is checked. The results are displayed in a table with the following data:

#	camis	dba	boro	building	street	zipcode	phone
1	50014885	CAMPANIA COAL FIRED PIZZA	Staten Island	3900	RICHMOND AVENUE	10312	71822
2	41022480	LA CANOA	Queens	551	ONONDAGA AVENUE	11705	71845

## LAMBDA FUNCTIONS:

### 1. Lambda Function Trigger on upload File in S3/Bucket/Input

- A Lambda function is triggered automatically when a new file is uploaded to S3/Bucket/Input. The function initiates the glue data crawler and glue data brewer jobs. And Move files from

S3/Bucket /transformed to S3/Bucket /transformed-archived and move files from S3/Bucket/cleaned to S3/Bucket/cleaned-archived

```

import json
import boto3
import logging
import time

# Initialize boto3 clients
glue_client = boto3.client('glue')
databrew_client = boto3.client('databrew')
lambda_client = boto3.client('lambda')
s3_client = boto3.client('s3')

# Update Lambda function configuration (e.g., set timeout to 200 seconds)
lambda_client.update_function_configuration(
    FunctionName='s3-databrew-task1',
    Timeout=200 # Timeout in seconds
)

def lambda_handler(event, context):
    # Log the event (S3 event)
    logging.info("Received event: %s" % event)

```

## 2. Lambda Function Trigger on upload File in S3/Bucket/Clean

- A Lambda function is triggered when a new file is uploaded to S3/Bucket/Clean. This function starts the next set of tasks, including running the glue data crawler on the clean data and performing the glue ETL job process.

```

import json
import boto3
import time

# Initialize boto3 clients for Glue and Lambda
glue_client = boto3.client('glue')
lambda_client = boto3.client('lambda')

# Update Lambda function configuration (e.g., set timeout to 500 seconds)
lambda_client.update_function_configuration(
    FunctionName='s3-etl-task2',
    Timeout=500 # Timeout in seconds
)

def lambda_handler(event, context):
    # Print the incoming event for debugging purposes
    print("Received event: %s" % event)

    # Define names for Glue Crawler and ETL Job
    glue_crawler_name = "cleaned"
    glue_job_name = "Untitled" # Replace with your Glue ETL job name

```

## 3. Lambda Function Trigger on upload File in S3/Bucket/Transformed

- A Lambda function is triggered when a new file is uploaded to S3/Bucket/Transformed. This function will trigger the Glue Data Crawler job and create database and table and insert to Glue Database via Athena to use in analytics or querying.

```

import boto3
import os
import csv
import json
import tempfile
from botocore.exceptions import ClientError

lambda_client = boto3.client('lambda')

# Update Lambda function configuration (e.g., set timeout to 60 seconds)
lambda_client.update_function_configuration(
    FunctionName='s3-glue',
    Timeout=200 # Timeout in seconds
)

# Initialize AWS clients
s3_client = boto3.client('s3')
glue_client = boto3.client('glue')

# Define your Glue Database and Table name
DATABASE_NAME = 'NYC_INSPECTION_DB'
TABLE_NAME = 'INSPECTION_TABLE'

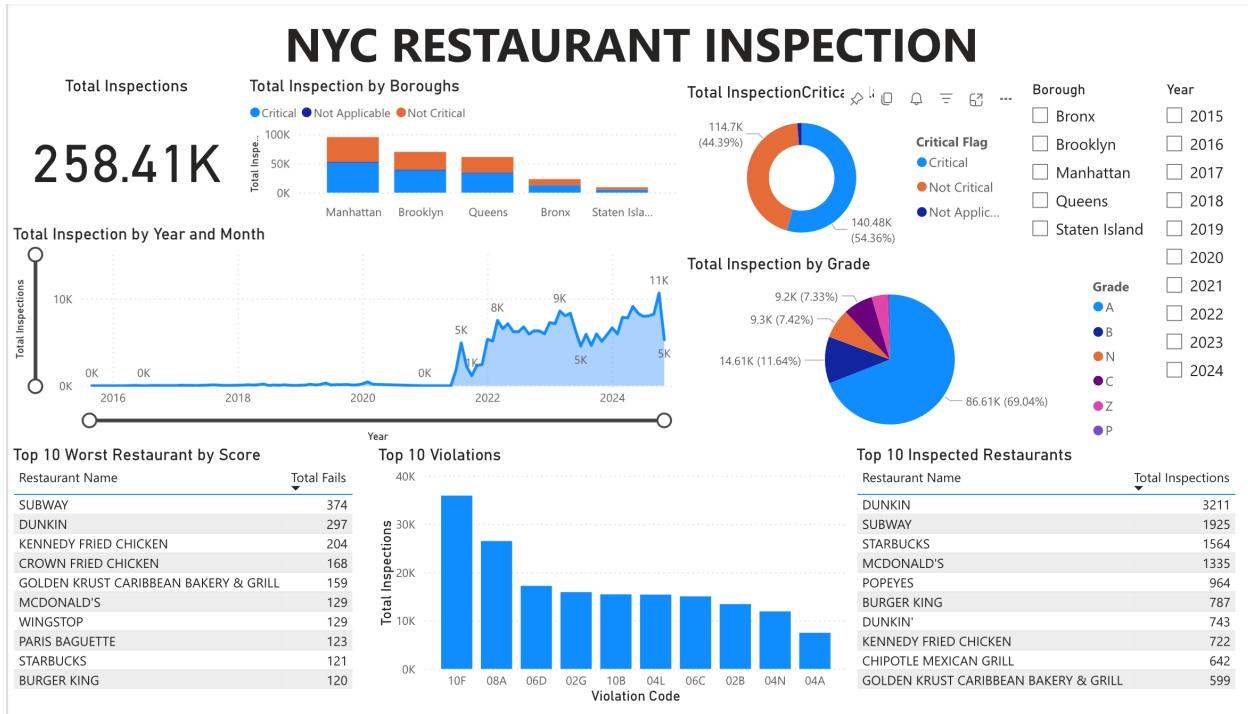
# S3 bucket and path for the incoming files
S3_BUCKET = 'de-cloud-project'
S3_PREFIX = 'transformed/'

# ...

```

**4. Cloudwatch:** Consists of all the logs and outputs of all jobs.

## DASHBOARD:



**GOAL** - The goal is to improve food safety compliance by reducing critical violations by 10%, increasing Grade A establishments by 15%, optimizing inspection resource allocation across boroughs, and implementing corrective measures for repeat offenders, ensuring public health and equitable regulatory enforcement.

## KPI's

### Total Inspections:

- 258.41K inspections conducted to date, indicating extensive monitoring efforts.

### Inspection Criticality:

- **Critical:** 54.36% (140.48K inspections).
- **Not Critical:** 45.64% (114.7K inspections).
- Highlights the proportion of critical issues requiring immediate action.

### Grade Distribution:

- **A:** 69.04% (86.61K inspections).
- **B:** 11.64% (14.61K).
- Majority of establishments are compliant with hygiene and safety standards.

### Top 10 Inspected Restaurants:

- Dunkin' leads with 3,211 inspections, followed by Subway (1,925) and Starbucks (1,564).
- Reflects targeted inspections on popular chains.

### Worst Offenders by Failures:

- Subway: 374 failures.
- Dunkin': 297 failures.
- Highlights problematic establishments requiring focused corrective measures.

### Violation Analysis:

- Top Violation Code: "10F" (40K cases).
- Indicates recurring food safety or hygiene issues.

## Metrics:

1. **Inspection Trends:**
  - Line chart illustrating inspection growth from 2016 to 2024.
  - Recent peak in inspections (11K in late 2024), showing increased regulatory activity.
2. **Year-Wise/ Borough-Wise Drill Down Analysis:**
  - Interactive filters allow users to analyze total inspections by specific years, revealing trends over time.
  - Enables granular analysis of inspection activity within each borough
3. **Borough-Wise Inspections:**
  - Manhattan: Highest activity across boroughs.
  - Staten Island: Minimal activity.
  - Enables geographic analysis of compliance efforts.
4. **Violation Codes:**
  - Bar chart showing top 10 violation codes.
  - Common violations highlight areas of non-compliance, aiding targeted training.
5. **Grade Breakdown:**
  - Pie chart distribution of grades (A, B, Z, etc.).
  - Majority receive "A," but ~30% require improvements.
6. **Restaurant Rankings:**
  - Top inspected chains like Dunkin' and Subway are both heavily monitored and among the worst offenders.

## Insights:

1. **Training and Education:**
  - The high proportion of "Grade B" (11.64%) and other grades indicates a need for training on food safety and hygiene standards to improve compliance.

2. **Focus on Critical Violations:**
  - Over 54% of inspections flagged as critical highlight a significant proportion of establishments with serious violations.
3. **Target Problematic Establishments:**
  - High failure rates in chains like Subway (374 failures) and Dunkin' (297 failures) indicate a need for targeted audits and enforcement to address recurring compliance issues
4. **Violation Patterns:**
  - High counts for specific violations (e.g., "10F") suggest recurring issues like improper food handling or hygiene.
5. **Geographic Strategy:**
  - Manhattan sees the highest inspection activity, suggesting a need for rebalancing resources to focus on underserved boroughs like Staten Island, which may face lower regulatory oversight.
6. **Seasonal and Monthly Trends:**
  - Inspection peaks toward the end of the year (e.g., Nov 2024 with 11K inspections) indicate potential operational stress or intensified campaigns. Planning inspection cycles evenly across months may enhance efficiency.
7. **Resource Allocation:**
  - Chains with high inspection counts like Dunkin' (3,211 inspections) and Subway (1,925 inspections) likely consume significant inspection resources. Analyzing these inspection frequencies against the risk can improve resource efficiency.



