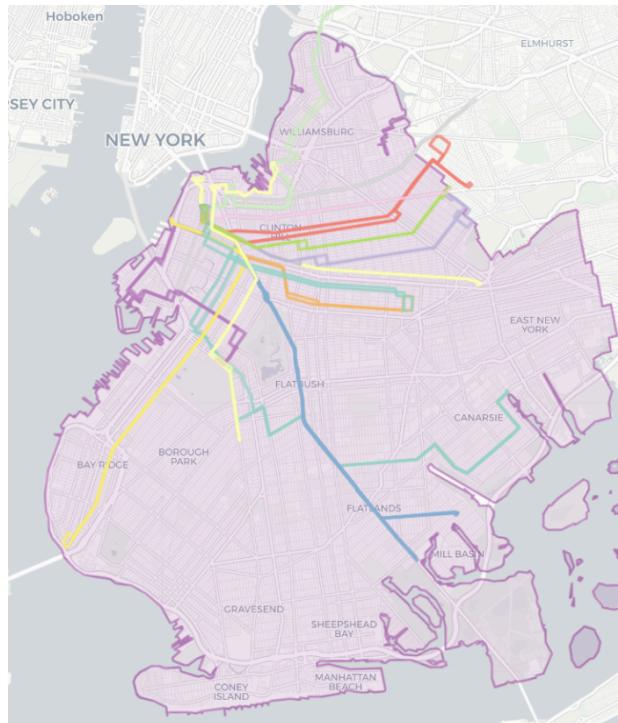


73.38 - Bases de Datos Espaciales y de Movilidad

8 Julio de 2025



Entrega parcial del TP Final - GTFS Scheduled - Nueva York

Jonathan Daniel Liu, 62533

Camila Di Toro, 62576

Profesor:

Alejandro Ariel Vaisman



Índice

Índice.....	1
1. Introducción.....	3
2. Fuentes de datos.....	5
2.1. GTFS Scheduled.....	5
2.2. GTFS Real Time.....	5
2.3. Borough administration data.....	6
3. Carga de datos de GTFS Schedule Data en PostgreSQL.....	6
3.1. Manhattan.....	7
Tabla agency.....	8
Tabla stops.....	8
Tabla routes.....	10
Tabla calendar.....	11
Tabla calendar_dates.....	11
Tabla shapes.....	12
Tabla trips.....	12
Tabla frequencies.....	13
Tabla stop_times.....	13
3.3. Manhattan y Brooklyn utilizando los .sql.....	14
3.4. Bronx, Queens, Staten Island utilizando los .sql.....	15
3.5. MTA Bus Company.....	16
Tabla agency.....	16
Tabla stops.....	17
Tabla routes.....	18
Tabla calendar.....	20
Tabla calendar_dates.....	20
Tabla shapes.....	20
Tabla trips.....	21
Tabla frequencies.....	21
Tabla stop_times.....	22
4. Carga de shapefiles.....	23
5. Visualización de rutas.....	24
5.1 Visualización por shape_id.....	24
5.2 Visualización por agencia.....	26
5.3 Visualización por tipo de ruta.....	31
6. Agrupación de rutas por segmento.....	34
6.1 Agrupación de rutas de Brooklyn (MTA NYCT).....	34
6.2 Agrupación de rutas de Brooklyn (MTA NYCT y MTA BC).....	39
7. Identificación de Hotspots y Duplicación de Rutas.....	43
7.1 Visualización de los viajes proyectados por segmento.....	44
7.2 Identificación de hotspots mediante una grilla espacial.....	48
7.3 Evaluación de duplicación en recorridos de buses.....	59

8. Velocidades.....	68
8.1 Velocidades por segmento en Brooklyn y Queens.....	69
8.2 Comparación de velocidades.....	78
8.2.1 Comparación de promedio por hora y por dia.....	79
8.2.2 Comparación de velocidades entre distritos.....	84

1. Introducción

Este informe corresponde a una entrega parcial del trabajo final de la materia **73.38 - Bases de Datos Espaciales y de Movilidad**, cuyo objetivo es analizar el sistema de transporte utilizando datos en formato GTFS (General Transit Feed Specification). En particular, se trabajó con los datos de la ciudad de Nueva York, una metrópolis con una extensa y compleja red de transporte urbano. La disponibilidad tanto de datos Scheduled como Real Time permite explorar distintos aspectos del funcionamiento del sistema de buses y su planificación. En esta primera parte del trabajo, el foco estuvo puesto en la exploración, carga, visualización y análisis de los datos Scheduled.

A lo largo del informe se observa una creciente complejidad en los análisis, en línea con el aprendizaje adquirido durante el desarrollo del proyecto. Las primeras secciones se centran en la carga y organización de los datos (Secciones 2 a 4), mientras que las Secciones 5 y 6 presentan visualizaciones y análisis exploratorio, basados principalmente en ejemplos y técnicas discutidas en el libro del curso. Finalmente, las Secciones 7 y 8 desarrollan estudios más elaborados, incluyendo la identificación de zonas con alta superposición de recorridos y el análisis de velocidades planificadas en distintos distritos de la ciudad, lo que permite obtener hallazgos más profundos sobre el sistema de transporte.

Este trabajo sienta las bases para una segunda etapa en la que se incorporarán datos en tiempo real, con el objetivo de contrastar la planificación con la operación efectiva del sistema.

Para complementar este informe, se incluyen dos recursos adicionales que permiten reproducir y explorar los resultados del trabajo:

[Repositorio en GitHub](#): contiene todos los scripts utilizados para el análisis. Se recomienda comenzar por el archivo `README.md`, donde se detallan los requerimientos, la configuración necesaria y la organización del código. La estructura del repositorio sigue el mismo esquema que el informe, lo que facilita la lectura conjunta y la comprensión de cada sección.

[Carpeta en Google Drive](#): organizada también por secciones, contiene los datos utilizados y las visualizaciones generadas con folium.

2. Fuentes de datos

2.1. GTFS Scheduled

Fuente de datos de GTFS Scheduled para buses de Nueva York:
<https://www.mta.info/developers>.

Los datos están divididos en seis archivos separados según región y operador:

- Bronx¹: Incluye rutas locales y la M100. Excluye Bx23 y rutas BxM.
- Brooklyn²: Incluye la mayoría de rutas locales de Brooklyn y algunas de Queens. Excluye B100, B103 y rutas BM.
- Manhattan³: Contiene rutas de Manhattan, pero no incluye la M100.
- Queens⁴: Incluye la mayoría de rutas locales de Queens y otras expressways como QM63, QM64, QM68. No incluye rutas operadas por MTA Bus Company.
- Staten Island⁵: Incluye todas las rutas de la isla.
- MTA Bus Company⁶: Contiene rutas que no están en los otros archivos, como B100, B103, Bx23, BxM y varias rutas adicionales en Queens.

Los datos provistos son válidos para el tercer quarter del año 2025, que incluye los meses de Julio, Agosto y Septiembre

Sugerimos descargar los .zip de [esta carpeta de Google Drive](#) para obtener los mismos resultados que los presentados a lo largo del desarrollo del informe. Durante el desarrollo del trabajo, nos encontramos con que en algunas ocasiones los .zip de la página del mta son cargados con defectos, como headers corruptos. Estos defectos en ocasiones impiden su procesamiento y análisis. Los datos de esa carpeta fueron descargados el día 26 de junio de 2025 y serán los utilizados para todo el proyecto, dado que son válidos hasta septiembre de 2025.

2.2. GTFS Real Time

Se agrega esa fuente de datos por completitud, a pesar de que no se utilizará en esta entrega parcial.

¹ https://rrgtfsfeeds.s3.amazonaws.com/gtfs_bx.zip

² https://rrgtfsfeeds.s3.amazonaws.com/gtfs_b.zip

³ https://rrgtfsfeeds.s3.amazonaws.com/gtfs_m.zip

⁴ https://rrgtfsfeeds.s3.amazonaws.com/gtfs_q.zip

⁵ https://rrgtfsfeeds.s3.amazonaws.com/gtfs_si.zip

⁶ https://rrgtfsfeeds.s3.amazonaws.com/gtfs_busco.zip

GTFS Real Time para buses de Nueva York (requiere una API Key)
<https://bt.mta.info/wiki/Developers/GTFSRT>:

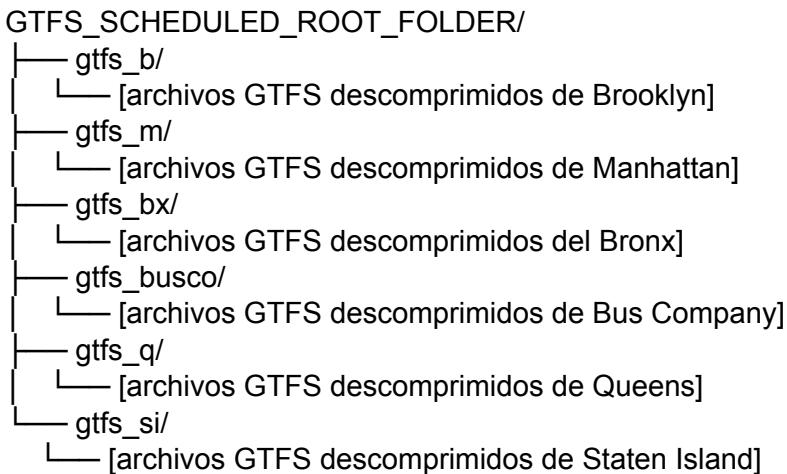
2.3. Borough administration data

Los datos utilizados para obtener los shapefiles con las geometrías de los distritos de Nueva York se encuentran en [esta carpeta de Drive](#) y fueron obtenidos de [este](#) sitio

3. Carga de datos de GTFS Schedule Data en PostgreSQL

El objetivo de esta sección es unificar los datos de todos los archivos GTFS Scheduled de la ciudad de Nueva York, con el fin de centralizar la información en un único schema de la base de datos. Esto permite facilitar el análisis y, al mismo tiempo, identificar posibles diferencias o inconsistencias entre los distintos conjuntos de datos.

Primero descargamos todos los archivos .zip en una misma carpeta GTFS_SCHEDULED_ROOT_FOLDER y descomprimimos cada uno de los archivos en sus respectivas carpetas: gtfs_b, gtfs_m, gtfs_bx, gtfs_busco, gtfs_q, gtfs_si.



En las secciones 3.1 y 3.2., se describen los pasos para cargar en PostgreSQL los datos de Manhattan y de Brooklyn, detallando los pasos y verificaciones realizadas para unificar los datos de ambos archivos GTFS. También estarán disponibles en [esta carpeta de git](#) los .sql necesarios para cargar los datos siguiendo las instrucciones de la sección 3.3.

Los archivos de Brooklyn, Manhattan, Bronx, Queens y Staten Islands son manejados por la misma agency: *MTA New York City Transit*. Esto hace que la combinación de datos no presente conflictos. Por esto se proveerán únicamente los .sql y las instrucciones para utilizarlos en la sección 3.4., dado que la lógica aplicada es la misma para Bronx, Queens y Staten Islands.

En la sección 3.5 se explicará cómo cargar los datos de Bus Company, donde surgieron algunos conflictos por leves diferencias de datos entre las agencies. Se explicará también cuáles fueron las decisiones tomadas.

3.1. Manhattan

Siguiendo los pasos del libro del curso⁷, es posible ver como cargar los datos de una de las regiones, en nuestro caso, Manhattan, utilizando `gtfs-via-postgres`⁸. Primero, creamos la base de datos donde cargaremos los datos, por ejemplo, `transport_ny`.

Dentro de la carpeta `GTFS_SCHEDULED_ROOT_FOLDER/gtfs_m` podemos correr los siguientes comandos:

```
export PGUSER=user
export PGPASSWORD=pass
export PGDATABASE=transport_ny
npm exec -- gtfs-to-sql --require-dependencies -- *.txt | psql -b
```

En Windows:

```
$env:PGUSER = "postgres"
$env:PGPASSWORD = "postgres"
$env:PGDATABASE = "transport_ny"

$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }

npm exec -- gtfs-to-sql --require-dependencies -- $txtFiles | psql -b
```

3.2. Brooklyn

La carga directa con `gtfs-to-sql` va a funcionar para la primera región. Para trabajar con las siguientes regiones es necesario encontrar una forma de juntar los datos de ambas regiones. Para esto, insertamos los datos de una segunda región (Brooklyn) en un

⁷ Mahmoud Sakr, Alejandro A. Vaisman, and Esteban Zimányi. Mobility Data Science - From Data to Insights. Data-Centric Systems and Applications. Springer, 2025.

⁸ <https://www.npmjs.com/package/gtfs-via-postgres/v/3.0.2>

nuevo schema. Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_b, corremos:

```
npm exec -- gtfs-to-sql --require-dependencies --schema brooklyn  
-- *.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }  
  
npm exec -- gtfs-to-sql --require-dependencies --schema brooklyn --  
$txtFiles | psql -b
```

Tabla agency

Ambos schemas tienen una única entrada en la tabla agency con los mismos datos

Tabla stops

Ahora hay que mover los datos del schema brooklyn al schema public, verificando que no haya inconsistencias.

Se eligió comenzar con la tabla stops. Lo primero que se puede observar es que hay ids de paradas que se encuentran repetidos, sin embargo, si se comparan los datos es posible ver que las ubicaciones y los nombres de las paradas son exactamente iguales.

```
SELECT  
    b.stop_id,  
    b.stop_name as brooklyn_name,  
    p.stop_name as public_name,  
    ST_Distance(b.stop_loc, p.stop_loc) as distance  
FROM brooklyn.stops b  
JOIN public.stops p ON b.stop_id = p.stop_id  
ORDER BY ST_Distance(b.stop_loc, p.stop_loc) DESC;
```

Esta query da el siguiente resultado:

	stop_id	brooklyn_name	public_name	distance
1	400070	TRINITY PL/RECTOR ST	TRINITY PL/RECTOR ST	0
2	400191	MADISON AV/E 37 ST	MADISON AV/E 37 ST	0
3	400723	6 AV/WAVERLY PL	6 AV/WAVERLY PL	0
4	400910	CHURCH ST/CHAMBERS ST	CHURCH ST/CHAMBERS ST	0
5	400917	6 AV/SPRING ST	6 AV/SPRING ST	0
6	402128	W 23 ST/6 AV	W 23 ST/6 AV	0
7	402130	E 23 ST/BROADWAY	E 23 ST/BROADWAY	0
8	402134	E 23 ST/3 AV	E 23 ST/3 AV	0
9	402143	E 23 ST/1 AV	E 23 ST/1 AV	0
10	402145	E 23 ST/3 AV	E 23 ST/3 AV	0
11	403647	ALLEN ST/DELANCEY ST	ALLEN ST/DELANCEY ST	0
12	403847	6 AV/WEST BROADWAY	6 AV/WEST BROADWAY	0
13	404225	BROADWAY/BARCLAY ST	BROADWAY/BARCLAY ST	0

Se puede observar que los nombres coinciden y que la distancia es cero, por lo que se puede insertar los datos de las paradas ignorando los ids que presentan conflictos.

Se verifica primero cuántas paradas hay en cada schema

```
select count(*) from public.stops; --1828
select count(*) from brooklyn.stops; --4629
```

Luego se insertan las paradas con distinto stop_id

```
INSERT INTO public.stops
SELECT
    stop_id,
    stop_code,
    stop_name,
    stop_desc,
    stop_loc,
    zone_id,
    stop_url,
    location_type::text::public.location_type_val,
    parent_station,
    stop_timezone,
    wheelchair_boarding::text::public.wheelchair_boarding_val,
    level_id,
    platform_code
FROM brooklyn.stops
ON CONFLICT (stop_id) DO NOTHING
```

Por último, se verifica que se hayan insertado correctamente

```
select count(*) from public.stops; --6444=1828+4629-13
```

Tabla routes

En este caso, ambas tablas contienen las mismas 323 entradas, y por lo tanto, no será necesario insertarlas de nuevo.

Ambas tienen 323 entradas

```
select count(*) from public.routes; --323
select count(*) from brooklyn.routes; --323
```

Ambas tienen los mismos ids

```
SELECT count(b.route_id)
FROM brooklyn.routes b
INNER JOIN public.routes p ON b.route_id = p.route_id; --323
```

Los mismos ids contienen la misma información

```
SELECT
    b.route_id
FROM brooklyn.routes b
JOIN public.routes p ON b.route_id = p.route_id
WHERE (b.agency_id, b.route_short_name, b.route_long_name,
b.route_desc, b.route_type, b.route_url, b.route_color,
b.route_text_color, b.route_sort_order)
<> (p.agency_id, p.route_short_name, p.route_long_name,
p.route_desc, p.route_type, p.route_url, p.route_color,
p.route_text_color, p.route_sort_order);
```

El resultado de esta query es vacío.

Tabla calendar

En este caso, no hay ids repetidos:

```
SELECT count(b.service_id)
FROM brooklyn.calendar b
INNER JOIN public.calendar p ON b.service_id = p.service_id; --0
```

Se contabiliza cuántas entradas tiene cada una de las tablas:

```
select count(*) from public.calendar; --16
select count(*) from brooklyn.calendar; --29
```

Luego se insertan los datos directamente:

```
INSERT INTO public.calendar (service_id, monday, tuesday,
wednesday, thursday, friday, saturday, sunday, start_date,
end_date)
SELECT
    service_id,
    monday::text::availability,
    tuesday::text::availability,
    wednesday::text::availability,
    thursday::text::availability,
    friday::text::availability,
    saturday::text::availability,
    sunday::text::availability,
    start_date,
    end_date
FROM brooklyn.calendar;
```

Por último, se verifica que se hayan insertado correctamente

```
select count(*) from public.calendar; --45=16+29
```

Tabla calendar_dates

Parte de la pk de la tabla calendar_dates es la pk de la tabla calendar, service_id. Por lo que si no hay service_ids repetidos, tampoco habrá entradas en la tabla calendar_dates que presenten conflictos.

En todos los casos se verifica que la cantidad de datos insertada sea la correcta.

Dejaremos de mostrar las queries para no extender tanto el informe

```
INSERT INTO public.calendar_dates
SELECT
    service_id,
    date,
    exception_type::text::exception_type_v
FROM brooklyn.calendar_dates;
```

Tabla shapes

En este caso, no hay shape_ids repetidos:

```
SELECT count(b.shape_id)
FROM brooklyn.shapes b
INNER JOIN public.shapes p ON b.shape_id = p.shape_id; --0
```

Se insertan los datos directamente:

```
INSERT INTO public.shapes (shape_id, shape_pt_sequence,
shape_dist_traveled, shape_pt_loc)
SELECT
    shape_id,
    shape_pt_sequence,
    shape_dist_traveled,
    shape_pt_loc
FROM brooklyn.shapes;
```

Tabla trips

En este caso, no hay ids repetidos:

```
SELECT count(b.route_id)
FROM brooklyn.trips b
INNER JOIN public.trips p ON b.route_id = p.route_id; --0
```

Se insertan los datos directamente:

```
INSERT INTO public.trips (trip_id, route_id, service_id,
trip_headsign, trip_short_name,
direction_id, block_id, shape_id,
wheelchair_accessible, bikes_allowed)
SELECT
    trip_id,
    route_id,
    service_id,
    trip_headsign,
    trip_short_name,
    direction_id,
    block_id,
    shape_id,
    wheelchair_accessible::text::wheelchair_accessibility,
    bikes_allowed::text::bikes_allowance
FROM brooklyn.trips;
```

Tabla frequencies

Ninguno de los dos esquemas tiene entradas, por lo que no hay nada que hacer

```
select count(*) from public.frequencies; --0
select count(*) from brooklyn.frequencies; --0
```

Notar que no hay un archivo llamado frequencies.txt en ninguno de los .zip, por lo que este resultado es el esperado.

Tabla stop_times

Parte de la pk the la tabla stop_times es el trip_id de la tabla trips. Como no había trip_ids repetidos, es posible insertar los datos directamente.

```
INSERT INTO public.stop_times (trip_id, arrival_time,
departure_time, stop_id, stop_sequence, stop_sequence_consec,
stop_headsign, pickup_type,
drop_off_type, shape_dist_traveled, timepoint, trip_start_time)
SELECT
    trip_id,
    arrival_time,
    departure_time,
    stop_id,
    stop_sequence,
    stop_sequence_consec,
    stop_headsign,
    pickup_type::text::pickup_drop_off_type,
    drop_off_type::text::pickup_drop_off_type,
    shape_dist_traveled,
    timepoint::text::timepoint_v,
    trip_start_time
FROM brooklyn.stop_times;
```

3.3. Manhattan y Brooklyn utilizando los .sql

En GTFS_SCHEDULED_ROOT_FOLDER descargar y descomprimir los .zip de los archivos GTFS que pueden obtenerse de [esta carpeta](#). También contar con una base de datos con el nombre transport_ny y dentro de esa base de datos crear el schema brooklyn.

Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_m, correr los siguientes comandos:

```
export PGUSER=user
export PGPASSWORD=pass
export PGDATABASE=transport_ny
npm exec -- gtfs-to-sql --require-dependencies -- *.txt | psql -b
```

En Windows:

```
$env:PGUSER = "user"
$env:PGPASSWORD = "pass"
$env:PGDATABASE = "transport_ny"

$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }

npm exec -- gtfs-to-sql --require-dependencies -- $txtFiles | psql -b
```

Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_b, correr el siguiente comando:

```
npm exec -- gtfs-to-sql --require-dependencies --schema brooklyn
-- *.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }

npm exec -- gtfs-to-sql --require-dependencies --schema brooklyn --
$txtFiles | psql -b
```

Finalmente, correr dentro de la base de datos el archivo brooklyn_load.sql, disponible en [este link](#).

3.4. Bronx, Queens, Staten Island utilizando los .sql

Antes de comenzar, asegurarse de tener las variables de entorno PGUSER, PGPASSWORD, PGDATABASE previamente seteadas

Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_bx, correr los siguientes comandos:

```
npm exec -- gtfs-to-sql --require-dependencies --schema bronx --  
*.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }  
  
npm exec -- gtfs-to-sql --require-dependencies --schema bronx --  
$txtFiles | psql -b
```

Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_q, correr los siguientes comandos:

```
npm exec -- gtfs-to-sql --require-dependencies --schema queens --  
*.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }  
  
npm exec -- gtfs-to-sql --require-dependencies --schema queens --  
$txtFiles | psql -b
```

Dentro de la carpeta GTFS_SCHEDULED_ROOT_FOLDER/gtfs_si, correr los siguientes comandos:

```
npm exec -- gtfs-to-sql --require-dependencies --schema island --  
*.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }  
  
npm exec -- gtfs-to-sql --require-dependencies --schema island --  
$txtFiles | psql -b
```

Finalmente, correr los .sql de carga, disponibles en [esta carpeta](#):

```
queens_load.sql, bronx_load.sql, island_load.sql
```

3.5. MTA Bus Company

Finalmente, insertar los datos del último .zip gtfs en un nuevo schema. Dentro de la carpeta `GTFS_SCHEDULED_ROOT_FOLDER/gtfs_busco`, correr:

```
npm exec -- gtfs-to-sql --require-dependencies --schema busco --
*.txt | psql -b
```

En Windows:

```
$txtFiles = Get-ChildItem -Filter *.txt | ForEach-Object { $_.Name }

npm exec -- gtfs-to-sql --require-dependencies --schema busco --
$txtFiles | psql -b
```

Los pasos descritos a continuación pueden realizarse corriendo el archivo `busco_load.sql` disponible en [esta carpeta](#).

Tabla agency

Cada esquema contiene su propia agencia:

```
select * from public.agency;
```

agency_id	agency_name	agency_url	agency_timezone	agency_lang	agency_phone
1	MTA NYCT	MTA New York City Transit	http://www.mta.info	America/New_York	en

```
select * from busco.agency;
```

agency_id	agency_name	agency_url	agency_timezone	agency_lang	agency_phone
1	MTABC	MTA Bus Company	http://www.mta.info	America/New_York	<null>

Se insertan los datos de la segunda agency:

```
INSERT INTO public.agency
SELECT * FROM busco.agency;
```

Tabla stops

Se identificaron paradas con el mismo stop_id pero con ubicaciones distintas. Para detectarlas se utilizó la siguiente consulta:

```
SELECT
    b.stop_id,
    b.stop_name as brooklyn_name,
    p.stop_name as public_name,
    ST_Distance(b.stop_loc, p.stop_loc) as distance
FROM brooklyn.stops b
JOIN public.stops p ON b.stop_id = p.stop_id
ORDER BY ST_Distance(b.stop_loc, p.stop_loc) DESC;
```

En total se detectaron 779 casos conflictivos. Las primeras 10 entradas del resultado de las 779 entradas conflictivas son las siguientes:

	stop_id	busco_name	public_name	distance
1	505239	DOUGLASTON PKWY/DOUGLASTON PLAZA	DOUGLASTON PKWY/DOUGLASTON PLAZA	147.88693906
2	701020	JAMAICA CENTER/BAY D	JAMAICA CENTER/BAY D	111.66258059
3	501044	BELL BLVD/212 ST	BELL BLVD/212 ST	106.92648432
4	404028	BROADWAY/W 219 ST	BROADWAY/W 220 ST	100.83847296
5	307181	FOUNTAIN AV/BROOKLYN DEVELOPMENTAL CENTER	FOUNTAIN AV/BROOKLYN DEVELOPMENTAL CENTER	97.47583873
6	501563	46 AV/171 ST	46 AV/171 ST	89.20197964
7	552284	FRANCIS LEWIS BLVD/147 DR	FRANCIS LEWIS BLVD/147 DR	88.93027625
8	104619	HUTCHINSON RIVER PKWY/BOLLER AV	HUTCHINSON RIVER PKWY/BOLLER AV	87.81660649
9	501269	CROSS ISLAND PKWY/ESTATES LN	CROSS ISLAND PKWY/ESTATES LN	85.85534302
10	400120	5 AV/E 81 ST	5 AV/E 80 ST	85.47576098

En la mayoría de ellos los nombres de las paradas coincidían y las distancias eran menores a una cuadra. Se decidió priorizar las ubicaciones provistas por la agency *MTA New York City Transit* (MTA NYCT) (presente en el esquema *public*), a costa de perder precisión en la información de *MTA Bus Company* (MTA BC). Se insertaron únicamente las paradas con stop_id no presentes en *public*:

```
INSERT INTO public.stops
SELECT
    stop_id,
    stop_code,
    stop_name,
    stop_desc,
```

```

stop_loc,
zone_id,
stop_url,
location_type::text::public.location_type_val,
parent_station,
stop_timezone,
wheelchair_boarding::text::public.wheelchair_boarding_val,
level_id,
platform_code
FROM busco.stops
ON CONFLICT (stop_id) DO NOTHING

```

Tabla routes

Se identificaron diferencias en 4 rutas con el mismo route_id, que presentaban atributos distintos. Se utilizó la siguiente consulta para detectarlas:

```

SELECT
    b.route_id,
    b.route_short_name AS busco_route_short_name,
    p.route_short_name AS public_route_short_name,
    b.route_long_name AS busco_route_long_name,
    p.route_long_name AS public_route_long_name,
    b.route_desc AS busco_route_desc,
    p.route_desc AS public_route_desc,
    b.route_type AS busco_route_type,
    p.route_type AS public_route_type,
    b.route_url AS busco_route_url,
    p.route_url AS public_route_url,
    b.route_color AS busco_route_color,
    p.route_color AS public_route_color,
    b.route_text_color AS busco_route_text_color,
    p.route_text_color AS public_route_text_color,
    b.route_sort_order AS busco_route_sort_order,
    p.route_sort_order AS public_route_sort_order
FROM busco.routes b
JOIN public.routes p ON b.route_id = p.route_id
WHERE (b.route_short_name, b.route_long_name, b.route_desc,
b.route_type,
    b.route_url, b.route_color, b.route_text_color,
b.route_sort_order)

```

```
IS DISTINCT FROM (p.route_short_name, p.route_long_name,
p.route_desc, p.route_type,
    p.route_url, p.route_color, p.route_text_color,
p.route_sort_order);
```

	public_route_short_name	busco_route.long_name	public_route.long_name	busco_route_desc	public_route_desc
Q26	College Point - Fresh Meadows	Fresh Meadows - College Point	Via College Pt Blvd / Parsons Blvd / 46Th Av	Via Hollis Court Blvd / 46th Av / Parsons Blvd	
Q28	Flushing - Bay Terrace Rush	Bayside - Flushing	Via Northern Blvd / Francis Lewis Blvd / Crocheron Av	Via Northern Blvd / 32nd Av / Corporal Kennedy St	
Q32	Jackson Heights - Penn Station (Midtown)	Jackson Heights - Penn Station	Via Roosevelt Av / Queens Blvd / 5Th Av / Madison Av	Via Roosevelt Av / Queens Blvd / Fifth Av / Madison Av	
Q110	Jamaica Avenue East	Floral Park - Jamaica	Via Jamaica Av.	Via Jamaica Av / Jericho Turnpike	

Se insertaron todas las rutas con `route_id` no presentes en `public.routes`, nuevamente priorizando los datos de MTA NYCT:

```
INSERT INTO public.routes (
    route_id,
    agency_id,
    route_short_name,
    route_long_name,
    route_desc,
    route_type,
    route_url,
    route_color,
    route_text_color,
    route_sort_order
)
SELECT
    route_id,
    agency_id,
    route_short_name,
    route_long_name,
    route_desc,
    route_type::text::route_type_val,
    route_url,
    route_color,
    route_text_color,
    route_sort_order
FROM busco.routes
ON CONFLICT (route_id) DO NOTHING;
```

Tabla calendar

No se detectaron IDs repetidos

```
SELECT count(b.service_id)
FROM busco.calendar b
INNER JOIN public.calendar p ON b.service_id = p.service_id;
```

Se insertaron todos los datos directamente:

```
INSERT INTO public.calendar (service_id, monday, tuesday,
wednesday, thursday, friday, saturday, sunday, start_date,
end_date)
SELECT
    service_id,
    monday::text::availability,
    tuesday::text::availability,
    wednesday::text::availability,
    thursday::text::availability,
    friday::text::availability,
    saturday::text::availability,
    sunday::text::availability,
    start_date,
    end_date
FROM busco.calendar;
```

Tabla calendar_dates

Es posible insertar los datos directamente, no hay conflictos:

```
INSERT INTO public.calendar_dates
SELECT
    service_id,
    date,
    exception_type::text::exception_type_v
FROM busco.calendar_dates;
```

Tabla shapes

No se encontraron shape_ids repetidos:

```
SELECT count(b.shape_id)
FROM busco.shapes b
INNER JOIN public.shapes p ON b.shape_id = p.shape_id; --0
```

Se insertaron los datos directamente:

```
INSERT INTO public.shapes (shape_id, shape_pt_sequence,
shape_dist_traveled, shape_pt_loc)
SELECT
    shape_id,
    shape_pt_sequence,
    shape_dist_traveled,
    shape_pt_loc
FROM busco.shapes;
```

Tabla trips

No se encontraron trip_id repetidos:

```
SELECT count(b.route_id)
FROM busco.trips b
INNER JOIN public.trips p ON b.route_id = p.route_id; --0
```

Se insertaron los datos directamente:

```
INSERT INTO public.trips (trip_id, route_id, service_id,
trip_headsign, trip_short_name,
direction_id, block_id, shape_id,
wheelchair_accessible, bikes_allowed)
SELECT
    trip_id,
    route_id,
    service_id,
    trip_headsign,
    trip_short_name,
    direction_id,
    block_id,
    shape_id,
    wheelchair_accessible::text::wheelchair_accessibility,
    bikes_allowed::text::bikes_allowance
FROM busco.trips;
```

Tabla frequencies

Ninguno de los dos esquemas tiene entradas, por lo que no hay nada que hacer.

```
select count(*) from public.frequencies; --0
select count(*) from busco.frequencies; --0
```

Tabla stop_times

Parte de la pk the la tabla stop_times es el trip_id de la tabla trips. Como no había trip_ids repetidos, se insertaron todos los datos directamente:

```
INSERT INTO public.stop_times (trip_id, arrival_time,
departure_time, stop_id, stop_sequence, stop_sequence_consec,
stop_headsign, pickup_type,
drop_off_type, shape_dist_traveled, timepoint, trip_start_time)
```

```
SELECT
    trip_id,
    arrival_time,
    departure_time,
    stop_id,
    stop_sequence,
    stop_sequence_consec,
    stop_headsign,
    pickup_type::text::pickup_drop_off_type,
    drop_off_type::text::pickup_drop_off_type,
    shape_dist_traveled,
    timepoint::text::timepoint_v,
    trip_start_time
FROM busco.stop_times;
```

4. Carga de shapefiles

Se utilizaron shapefiles correspondientes a los distritos de la ciudad de Nueva York, que contienen las geometrías necesarias para realizar visualizaciones y consultas espaciales a lo largo del trabajo. Los shapefiles se encuentran disponibles en [esta carpeta](#).

Originalmente, las geometrías están referenciadas en el sistema NAD83 / New York Long Island (EPSG:2263)⁹, adecuado para esta región pero con unidades en pies.

Para facilitar distintos tipos de análisis, se agregaron nuevas columnas con geometrías transformadas a dos sistemas adicionales:

EPSG:4326 (WGS 84): usado en los datos GTFS, en ocasiones se utiliza en conjunto con los datos GTFS que vienen previamente definidos con este srid.

EPSG:32118 (NAD83 / New York East - en metros)¹⁰: útil para análisis espaciales que requieren unidades métricas, permitiendo realizar cálculos más rápidos. Por ejemplo, nos permite calcular fácilmente distancias en metros para calcular km/h en las secciones de velocidad sin hacer la conversión a pies

```
-- 1. Agregar columnas para las geometrías transformadas
ALTER TABLE public.ny_adm ADD COLUMN geom_4326
geometry(MultiPolygon, 4326);
ALTER TABLE public.ny_adm ADD COLUMN geom_32118
geometry(MultiPolygon, 32118);

-- 2. Actualizar las nuevas columnas con las transformaciones
UPDATE public.ny_adm
SET
    geom_4326 = ST_Transform(geom, 4326),
    geom_32118 = ST_Transform(geom, 32118);
```

⁹ <https://epsg.io/2263>

¹⁰ <https://epsg.io/32118>

5. Visualización de rutas

El objetivo de esta sección es mostrar diferentes visualizaciones de las rutas de bus de Nueva York. Como primer acercamiento, simplemente se mostrarán las shapes cargadas a partir de la vista `shapes_aggregated` que fueron generadas utilizando la herramienta `gtfs-via-postgres` en la sección 3.

Luego, se agruparán estas shapes según la ruta a la que pertenezcan, para visualizarlas distinguiendo primero según a qué agency pertenecen y luego según el tipo de ruta.

Dentro de [esta carpeta](#) del repositorio de github de este trabajo pueden encontrarse el código utilizado para esta sección.

Dentro de [esta carpeta](#) de drive, se encuentran los .html de las visualizaciones realizadas para que el lector pueda interactuar sin tener que seguir todos los pasos necesarios. Notar que, dado que el espacio es limitado y algunas visualizaciones son pesadas, estos datos estarán disponibles únicamente hasta el 31 de Julio de 2025.

5.1 Visualización por shape_id

Esta es la primera visualización y la más simple que fue realizada en base a lo descrito en la sección 11.2.2 del libro del curso.

El siguiente código permite generar un mapa interactivo utilizando folium, que muestra los recorridos de todas las rutas cargadas para Nueva York.

```
def map_individual_lines():
    sql = 'SELECT * FROM shapes_aggregated;'
    try:
        engine = get_engine()
        shapes_gdf = gpd.read_postgis(sql, engine,
        geom_col='shape', crs='EPSG:4326')
        print(f"Loaded {len(shapes_gdf)} transit line records")
```

```
except Exception as e:  
    print(f"Error loading data: {e}")  
    return None  
  
# Crear el mapa centrado en Nueva York  
map_indiv_lines = f.Map(location=[40.7128, -74.0060],  
tiles='CartoDB positron',  
                        zoom_start=13, control_scale=True)  
  
# Agrupar por shape_id y agregar las líneas al mapa  
route_count = 0  
for shape_id, shape_group in shapes_gdf.groupby('shape_id'):  
    feature_group = f.FeatureGroup(name=f"Route {shape_id}",  
show=True)  
  
    for geometry in shape_group.geometries:  
        if geometry.geom_type == 'LineString':  
            coords = [(lat, lon) for lon, lat in  
geometry.coords]  
            f.PolyLine(locations=coords, color="blue",  
weight=2, opacity=0.8).add_to(feature_group)  
  
    feature_group.add_to(map_indiv_lines)  
    route_count += 1  
  
f.LayerControl(collapsed=False).add_to(map_indiv_lines)  
print(f"Created map with {route_count} transit routes")  
return map_indiv_lines
```

La visualización generada luce de la siguiente forma:

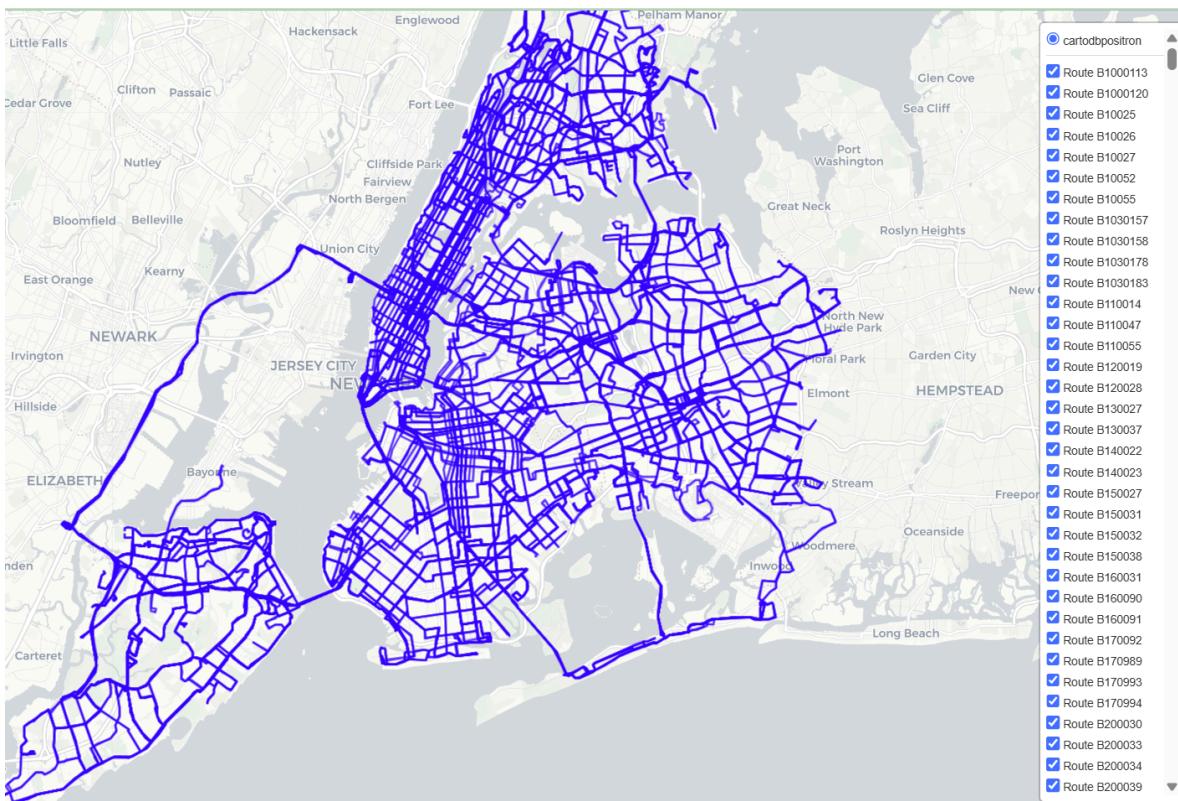


Figura 5.1-1: Visualización de todas las shapes de rutas de bus en Nueva York, obtenidas desde la vista `shapes_aggregated`.

En la imagen se puede observar un mapa interactivo de la ciudad de Nueva York que muestra todos los recorridos de buses cargados desde la vista `shapes_aggregated`, generada con la herramienta `gtfs-via-postgres`. Cada línea azul representa el trazado de una ruta de bus (`shape`) identificada por su `shape_id`. Este primer acercamiento proporciona una vista general del sistema de transporte público por bus en toda el área.

5.2 Visualización por agencia

En esta segunda visualización, se busca analizar las rutas diferenciando por la agencia que las opera. Como se mencionó anteriormente, el conjunto de datos GTFS utilizado incluye información de dos agencias distintas:

- MTA NYCT: MTA New York City Transit
- MTABC: MTA Bus Company

Dado que ambas forman parte del sistema de transporte de la ciudad pero gestionan diferentes rutas, resulta interesante mostrar su cobertura de forma diferenciada. Para ello, se creó una vista auxiliar en la base de datos que relaciona las rutas (`routes`), los viajes (`trips`) y los trazados de las rutas (`shapes_aggregated`). Esta vista, llamada `route_shapes_view`, permite acceder a la geometría de cada trazado junto con su información de agencia y tipo de ruta, agrupada por `shape_id`.

```
CREATE VIEW route_shapes_view AS
SELECT
    r.route_id,
    r.route_type,
    r.agency_id,
    s.shape_id,
    s.shape
FROM
    routes r
JOIN trips t ON r.route_id = t.route_id
JOIN shapes_aggregated s ON t.shape_id = s.shape_id
GROUP BY s.shape_id, s.shape, r.agency_id, r.route_type,
r.route_id;
```

A partir de esta vista, se desarrolló el script en Python que genera un mapa interactivo utilizando la librería folium, donde se representan las rutas diferenciadas según la agencia que las opera. Esto se logró aplicando distintos colores a las geometrías según el valor de `agency_id`. A continuación se presenta un breve análisis realizado a partir de la visualización generada

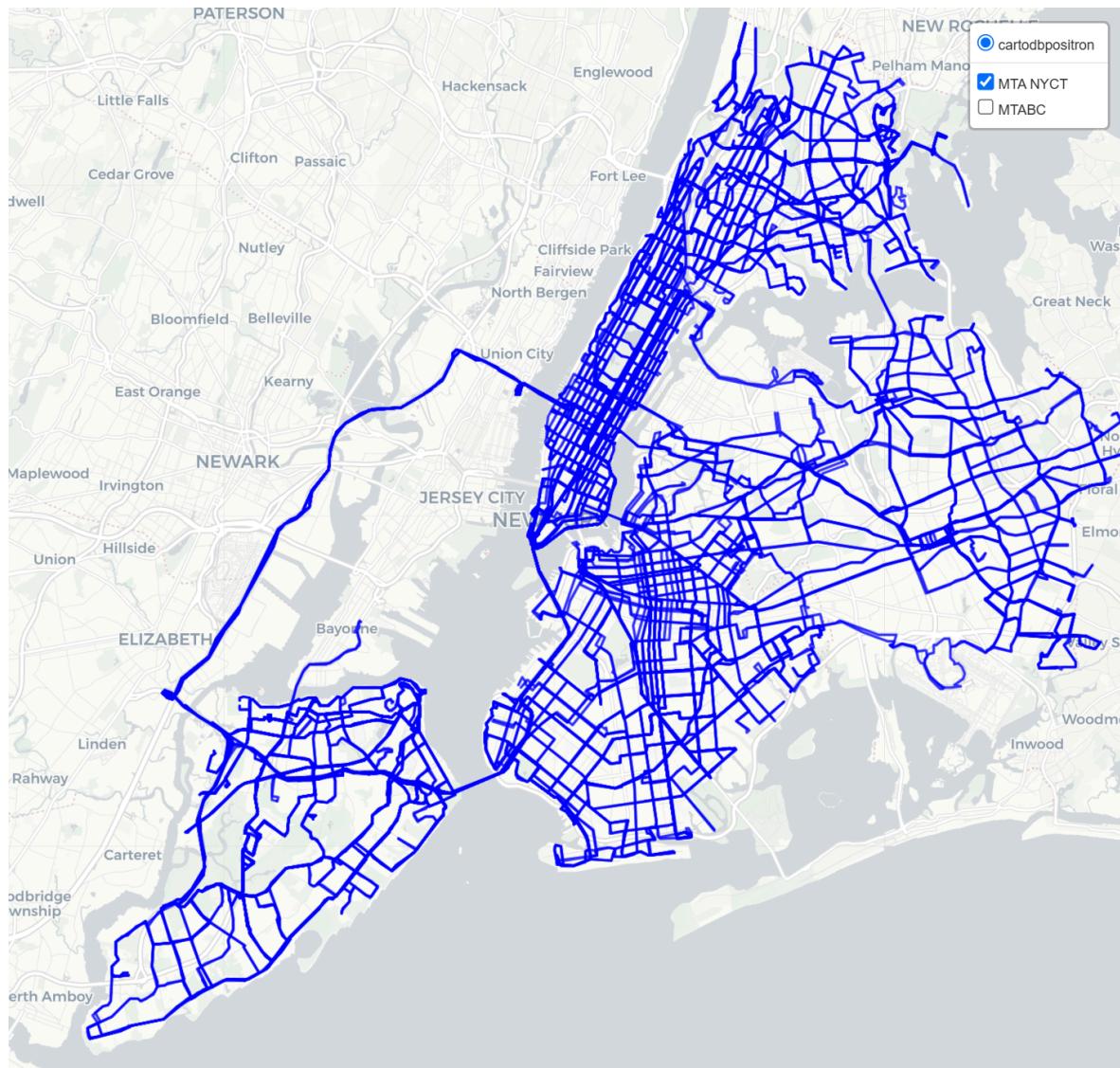


Figura 5.2-1: Visualización de las rutas de bus MTA NYCT

Se observa que MTA NYCT presenta una cobertura densa en Manhattan, Brooklyn, Bronx y Staten Island, pero con escasa presencia en partes de Queens, esto resalta la necesidad de considerar datos de MTA BC para lograr una visión completa.

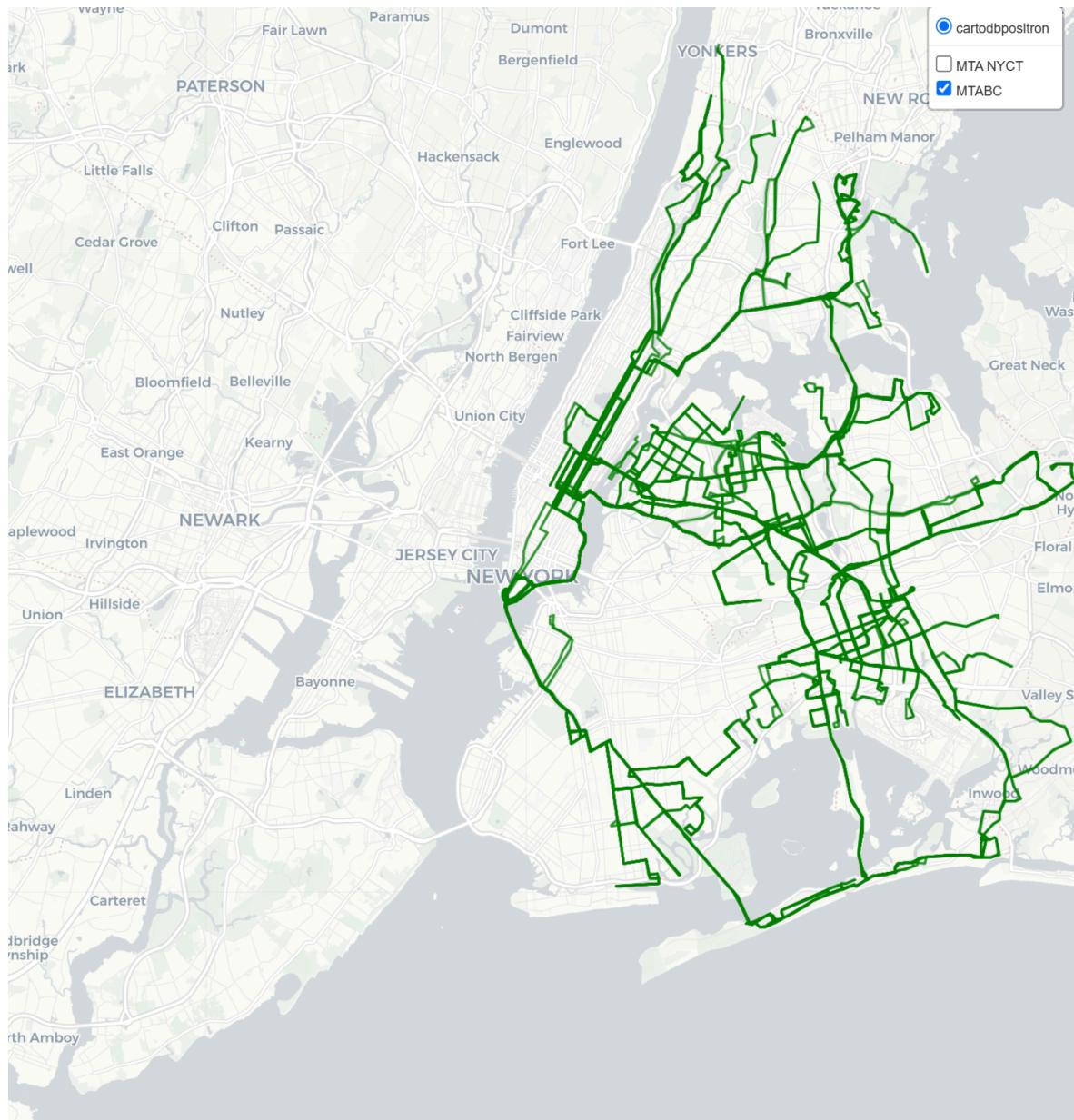


Figura 5.2-2: Visualización de las rutas de bus MTA BC

Las rutas verdes de MTA BC se concentran principalmente en Queens. Hay algunas rutas que pasan por Manhattan y Bronx, y escasa presencia en Brooklyn. Cubre zonas donde vimos que MTA NYCT tiene escasa o nula presencia. Esta visualización confirma el carácter complementario entre ambas agencias.

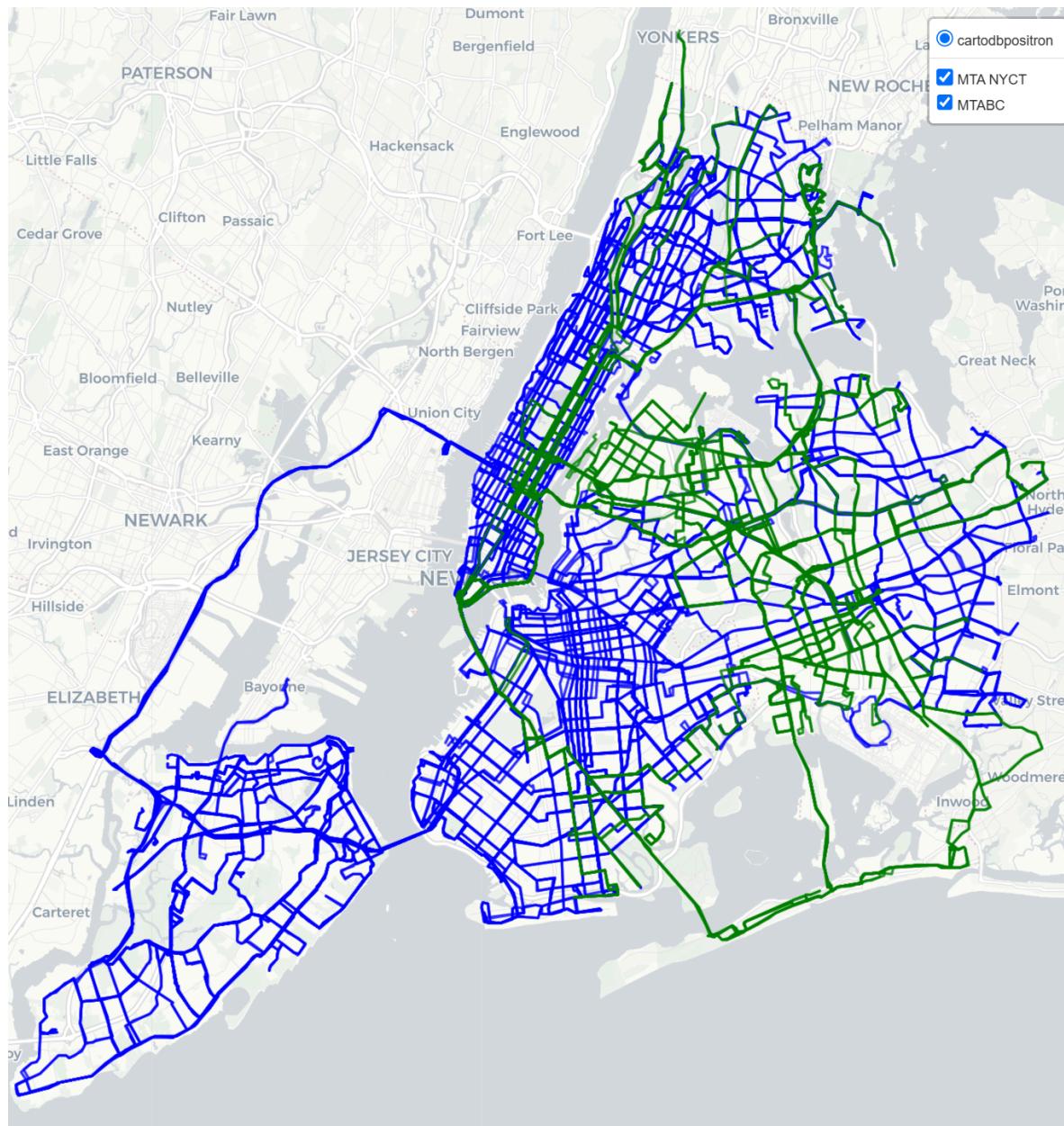


Figura 5.2-3: Rutas de ambas agencias en un mismo mapa. Azul: MTA NYCT, Verde: MTABC.

La **figura 5.2-3** permite ver mejor la distribución de las rutas observadas. La superposición de rutas evidencia cómo cada agencia prioriza distintas regiones. Esto evidencia que para comprender en profundidad el sistema de transporte por bus en Nueva York es imprescindible considerar los datos de ambas agencias.

5.3 Visualización por tipo de ruta

En esta sección se presentan las rutas agrupadas según su tipo (`route_type`).

Esta visualización se realizó a partir de una curiosidad que surgió al explorar los datos: en la tabla routes, además del valor esperado 3 (correspondiente a buses comunes), apareció también el valor 711.

Según la documentación oficial de GTFS¹¹, el valor 3 representa rutas de tipo Bus, abarcando servicios tanto de corta como de larga distancia. Por otro lado, 711 no forma parte del estándar básico, pero pertenece a una extensión propuesta por Google¹² y se interpreta como Shuttle Bus.

En la **Figura 5.3-1** se observan claramente diferenciadas las rutas según su tipo. Las rutas tipo 3, en rojo, conforman una red muy extensa que cubre prácticamente todo el territorio de Nueva York. Por otro lado, las rutas tipo 711, en violeta, representan una cantidad muy reducida dentro del conjunto total.

Esta visualización pone en evidencia que la gran mayoría de las rutas corresponden al tipo 3, mientras que las de tipo 711 tienen una presencia acotada y específica.

¹¹ <https://gtfs.org/documentation/schedule/reference/#routestxt>

¹² <https://developers.google.com/transit/gtfs/reference/extended-route-types>

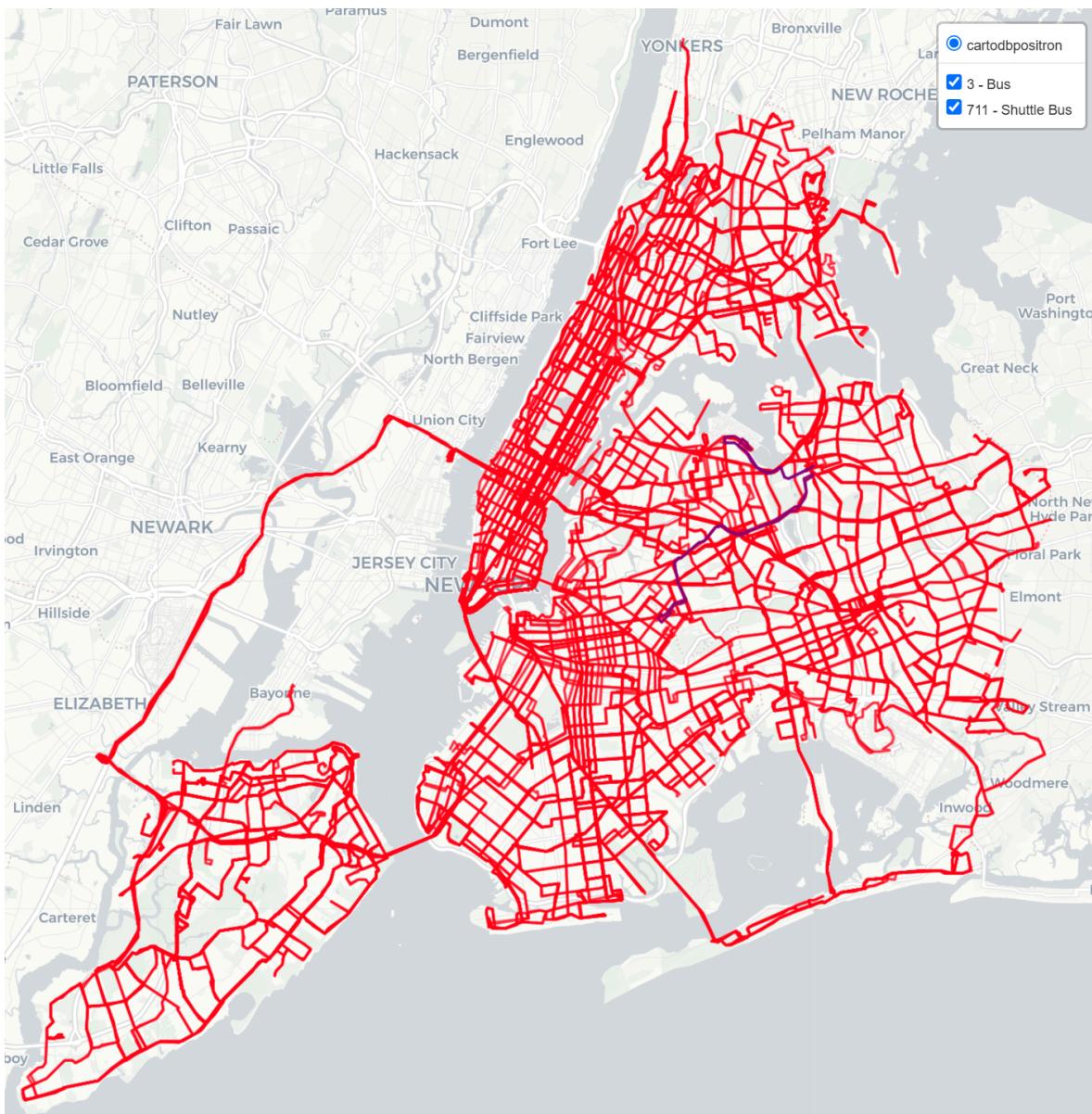


Figura 5.3-1: Rutas del sistema de transporte agrupadas según su route_type. En rojo se muestran los buses (tipo 3) y en violeta los shuttle buses (tipo 711), cuya presencia es mínima.

Inspeccionando un poco los datos, es posible verificar que hay 40 rutas definidas con el route_type 711. Sin embargo, solo 2 de ellas tienen trips definidos.

<pre>SELECT count(*), r.route_type FROM routes r group by r.route_type;</pre>	<pre>SELECT distinct r.route_id FROM routes r JOIN trips t ON r.route_id = t.route_id WHERE r.route_type = '711' ORDER BY r.route_id;</pre>
---	---

count	route_type
371	3
40	711

route_id
Q90
Q98

Lo cual es consistente con los datos observados en la visualización.

6. Agrupación de rutas por segmento

En esta sección se busca descomponer las rutas del sistema de transporte en segmentos, es decir, tramos entre dos paradas consecutivas. El objetivo es identificar la cantidad de rutas y de viajes que circulan por cada segmento.

Trabajar a nivel de segmento permite detectar, por ejemplo, tramos con alta superposición de servicios (posibles candidatos a optimización), o bien para calcular la velocidad a partir de los tiempos de partida y llegada entre los extremos de un segmento y su longitud. En ese sentido, esta sección no solo tiene un valor descriptivo, sino que también sienta la base para análisis más complejos que se realizarán en las próximas secciones.

Para facilitar el desarrollo, el proceso se aplica al distrito de Brooklyn utilizando un único archivo gtfs, lo que permite validar la metodología. Este primer análisis se realiza siguiendo la misma metodología que se utiliza en el libro del curso. Luego, se combina el archivo gtfs de Brooklyn con el de MTA Bus Company.

Los datos utilizados corresponden al período comprendido entre el 15 y el 31 de julio de 2025. La segmentación se realiza utilizando la librería `gtfs_functions`, que permite generar automáticamente los tramos a partir de los datos GTFS. Estos segmentos se almacenan en la base de datos y se cruzan con los viajes programados para calcular cuántos servicios atraviesan cada tramo.

El código utilizado en esta sección se encuentran dentro de [esta carpeta](#) del repositorio de GitHub del trabajo y las visualizaciones dentro de [esta carpeta](#) de Drive

6.1 Agrupación de rutas de Brooklyn (MTA NYCT)

En esta primera etapa del análisis, se trabaja exclusivamente con los datos de MTA NYCT. El objetivo es generar los segmentos de todas las rutas programadas que operan en

el distrito durante el período del 15 al 31 de julio de 2025, y visualizar cómo se distribuyen los viajes a lo largo de estos tramos.

Para obtener los segmentos, se utiliza la función `gtfs.Feed` de la librería `gtfs_functions`, que permite extraer directamente los tramos entre paradas consecutivas desde un archivo GTFS. En este caso, se utiliza el archivo correspondiente a Brooklyn (`gtfs_b.zip`) y los segmentos generados se guardan en la base de datos, en el schema `brooklyn`. Esto puede verse en el código que se muestra a continuación:

```
def load_gtfs_feed(file_path, start_date, end_date):
    feed = gtfs.Feed(file_path, start_date=start_date,
end_date=end_date)
    return feed.segments

def save_to_postgis(gdf, table_name, schema, if_exists):
    engine = get_engine()
    gdf.to_postgis(table_name, engine, schema=schema,
if_exists=if_exists)
    print(f"Data saved to '{schema}.{table_name}' in database
'{DB_CONFIG['database']}'.")"

if __name__ == "__main__":
    base_dir = os.path.dirname(__file__)
    start_date = '2025-07-15'
    end_date = '2025-07-31'

    gtfs_path = os.path.join(base_dir,
"../GTFS_SCHEDULED_ROOT_FOLDER/gtfs/gtfs_b.zip")
    segments_gdf = gtfs.Feed(gtfs_path, start_date=start_date,
end_date=end_date).segments
    print(f"Saving {len(segments_gdf)} rows from gtfs_b.zip")
    save_to_postgis(
        segments_gdf,
        table_name="segments",
        schema="brooklyn",
        if_exists='replace'
    )
```

Luego, se crea la vista a partir de estos segmentos que los segmentos generados con la vista `connections`, que contiene todos los viajes programados durante el período seleccionado.

```

CREATE VIEW brooklyn.segments_load AS
    SELECT c.from_stop_id, c.from_stop_name, c.to_stop_id,
c.to_stop_name, s.geometry,
    COUNT(trip_id) AS notrips,
    string_agg(DISTINCT c.route_short_name, ',') AS routes
    FROM brooklyn.segments s
    JOIN brooklyn.connections c
        ON s.route_id = c.route_id
        AND s.direction_id = c.direction_id
        AND s.start_stop_id = c.from_stop_id
        AND s.end_stop_id = c.to_stop_id
    WHERE date BETWEEN '2025-07-15' AND '2025-07-31'
    GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id,
c.to_stop_name, s.geometry;

```

Para cada segmento, se calcula:

- notrips: el total de viajes programados que pasan por ese segmento.
- routes: las rutas distintas que utilizan ese tramo.

Para dar un vistazo a los datos generados, es posible realizar unas simples consultas para ver que se generaron 5049 segmentos con un promedio de 2844.52 trips cada uno. El rango de viajes va desde 27097, para el segmento con más viajes a 65, para el segmento con menos viajes. Los 3 segmentos con más viajes son:

	from_stop_name	to_stop_name	routes	notrips
1	FLATBUSH AV/NOSTRAND AV	FLATBUSH AV/E 29 ST	B41	27097
2	GLENWOOD RD/NOSTRAND AV	GLENWOOD RD/FLATBUSH AV	B6	20620
3	RALPH AV/FLATLANDS AV	GLENWOOD RD/RALPH AV	B6	20620

Como se puede ver hay un único segmento con muchos más viajes que el resto, se tendrá esto en cuenta al limitar el valor máximo de la cantidad de viajes en la próxima visualización.

Finalmente, con los datos ya agregados, se utiliza la librería Folium para generar un mapa interactivo donde cada segmento se representa con un color que varía en función del número de viajes que lo recorren:

```

def load_segments_from_postgis(engine):
    query = """
        SELECT * FROM brooklyn.segments_load;
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def visualize_segments(segment_data, cutoff=20000):
    transport_map = f.Map(location=[40.7128, -74.0060],
    tiles='CartoDB positron', zoom_start=11)

    colormap = cm.LinearColormap(
    colors=['white', 'yellow', 'orange', 'red', 'darkred'],
    vmin=0,
    vmax=cutoff,
    caption='Trip Count'
    )
    colormap.add_to(transport_map)

    for _, segment in segment_data.iterrows():
        trip_count = min(segment['notrips'], cutoff)
        color = colormap(trip_count)
        geo_json = f.GeoJson(
            data={
                "type": "Feature",
                "geometry": segment['geometry'].__geo_interface__,
                "properties": {
                    "routes": segment['routes'],
                    "from_stop_name": segment['from_stop_name'],
                    "to_stop_name": segment['to_stop_name'],
                    "notrips": segment['notrips']
                }
            },
            style_function=lambda x, color= color: {
                'color': color, 'weight': 3, 'opacity': 0.7
            },
            tooltip=GeoJsonTooltip(
                fields=['routes', 'from_stop_name',
                'to_stop_name', 'notrips'],
                aliases=['Routes', 'From Stop', 'To Stop', 'Trip
Count'],
                localize=True
            )
        )
        geo_json.add_to(transport_map)

    return transport_map

```

El resultado de esta visualización es el siguiente:

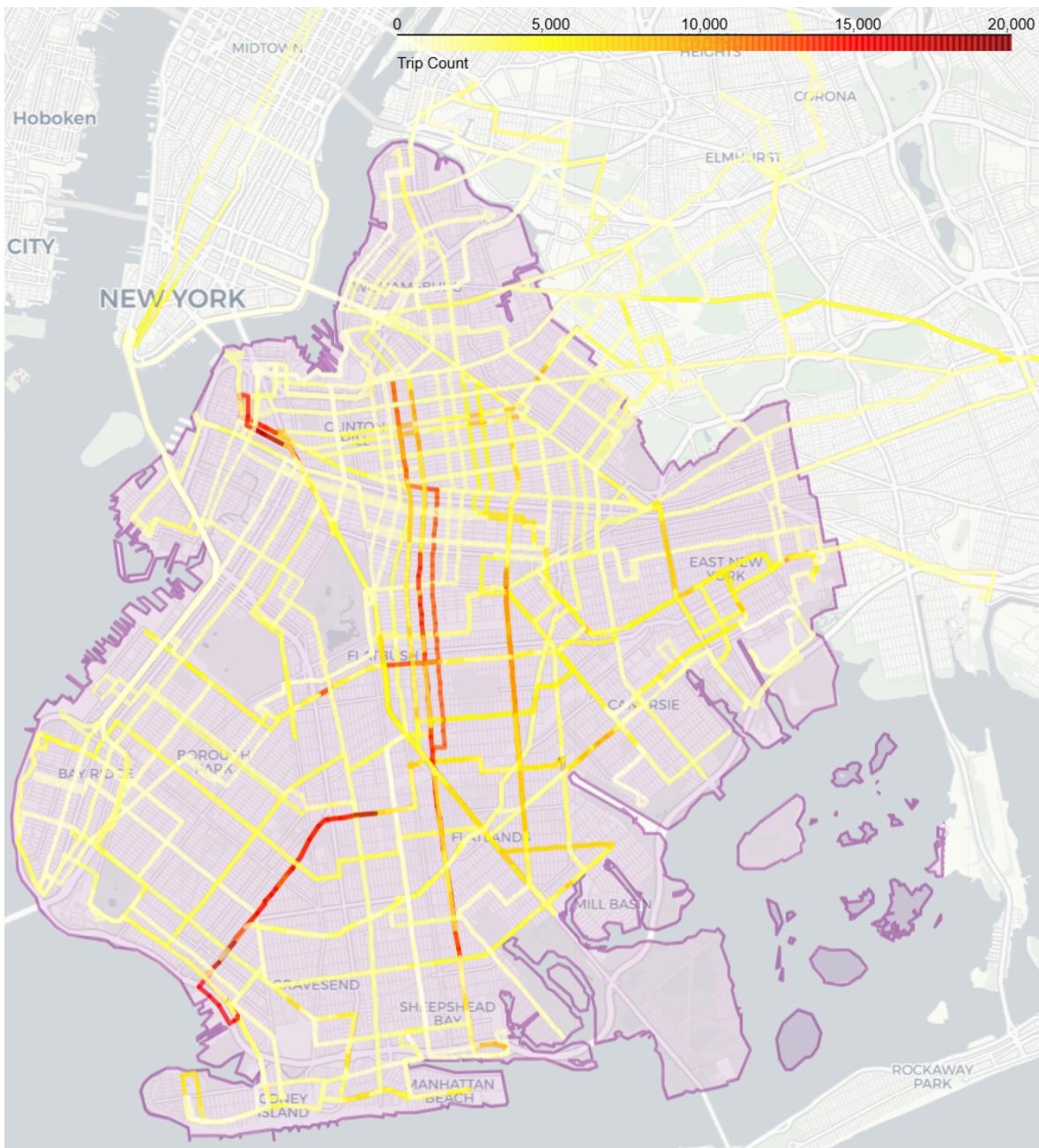


Figura 6.1-1: Mapa de segmentos en Brooklyn según cantidad de viajes (15 al 31 de julio de 2025). El color indica la cantidad de servicios programados que atraviesan cada tramo. Datos de MTA NYCT

A partir del mapa se identifican patrones relevantes:

La mayor concentración de servicios se da en el centro del distrito, donde se observan muchos segmentos en rojo oscuro, lo que indica una alta frecuencia de viajes y la presencia de múltiples rutas que comparten tramos.

En las zonas periféricas, los segmentos tienden a aparecer en colores más claros, reflejando una menor cantidad de servicios. Esto puede deberse a una menor demanda, pero también puede señalarn posibles problemas de cobertura o frecuencia.

6.2 Agrupación de rutas de Brooklyn (MTA NYCT y MTA BC)

Utilizando el mismo código que en la sección anterior, generamos los segmentos para MTA BC. En este caso, se utiliza el archivo correspondiente `gtfs_busco.zip` y los segmentos generados se guardan en la base de datos, en el esquema `busco`. Esto puede verse en el código que se muestra a continuación:

```
def load_gtfs_feed(file_path, start_date, end_date):
    feed = gtfs.Feed(file_path, start_date=start_date,
end_date=end_date)
    return feed.segments

def save_to_postgis(gdf, table_name, schema, if_exists):
    engine = get_engine()
    gdf.to_postgis(table_name, engine, schema=schema,
if_exists=if_exists)
    print(f"Data saved to '{schema}.{table_name}' in database
'{DB_CONFIG['database']}'.")"

if __name__ == "__main__":
    base_dir = os.path.dirname(__file__)
    start_date = '2025-07-15'
    end_date = '2025-07-31'

    gtfs_path = os.path.join(base_dir,
"../GTFS_SCHEDULED_ROOT_FOLDER/gtfs/gtfs_busco.zip")
    segments_gdf = gtfs.Feed(gtfs_path, start_date=start_date,
end_date=end_date).segments
    print(f"Saving {len(segments_gdf)} rows from gtfs_b.zip")
    save_to_postgis(
        segments_gdf,
        table_name="segments",
        schema="busco",
        if_exists='replace'
    )
```

Es importante notar que no es posible utilizar el modo `append` que nos provee `gtfs_functions`, dado que esto no tendría en cuenta la incorporación de nuevas rutas.

Se puede crear también la vista en este schema

```
CREATE VIEW busco.segments_load AS
    SELECT c.from_stop_id, c.from_stop_name, c.to_stop_id,
    c.to_stop_name, s.geometry,
        COUNT(trip_id) AS notrips,
        string_agg(DISTINCT c.route_short_name, ',') AS routes
    FROM busco.segments s
    JOIN busco.connections c
        ON s.route_id = c.route_id
        AND s.direction_id = c.direction_id
        AND s.start_stop_id = c.from_stop_id
        AND s.end_stop_id = c.to_stop_id
    WHERE date BETWEEN '2025-07-15' AND '2025-07-31'
    GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id,
    c.to_stop_name, s.geometry;
```

Y luego unir la vista del esquema busco con la de brooklyn para tener los datos completos.

```
CREATE VIEW brooklyn.segments_all AS
SELECT
    from_stop_id,
    from_stop_name,
    to_stop_id,
    to_stop_name,
    geometry,
    SUM(notrips) AS notrips,
    string_agg(routes, ',') AS routes
FROM (
    SELECT * FROM brooklyn.segments_load
    UNION ALL
    SELECT * FROM busco.segments_load
) AS all_segments
GROUP BY
    from_stop_id,
    from_stop_name,
    to_stop_id,
    to_stop_name,
    geometry;
```

En este caso, se generaron 8623 segmentos con un promedio de 2571.53 trips cada uno. El rango de viajes va desde 27097, para el segmento con más viajes, a 13, para el segmento con menos viajes. Los 3 segmentos con más viajes son, al igual que antes:

	<input type="checkbox"/> from_stop_name	<input type="checkbox"/> to_stop_name	<input type="checkbox"/> routes	<input type="checkbox"/> notrips
1	FLATBUSH AV/NOSTRAND AV	FLATBUSH AV/E 29 ST	B41	27097
2	GLENWOOD RD/NOSTRAND AV	GLENWOOD RD/FLATBUSH AV	B6	20620
3	RALPH AV/FLATLANDS AV	GLENWOOD RD/RALPH AV	B6	20620

La visualización fue realizada con el mismo código que antes, simplemente cambiando la vista utilizada.

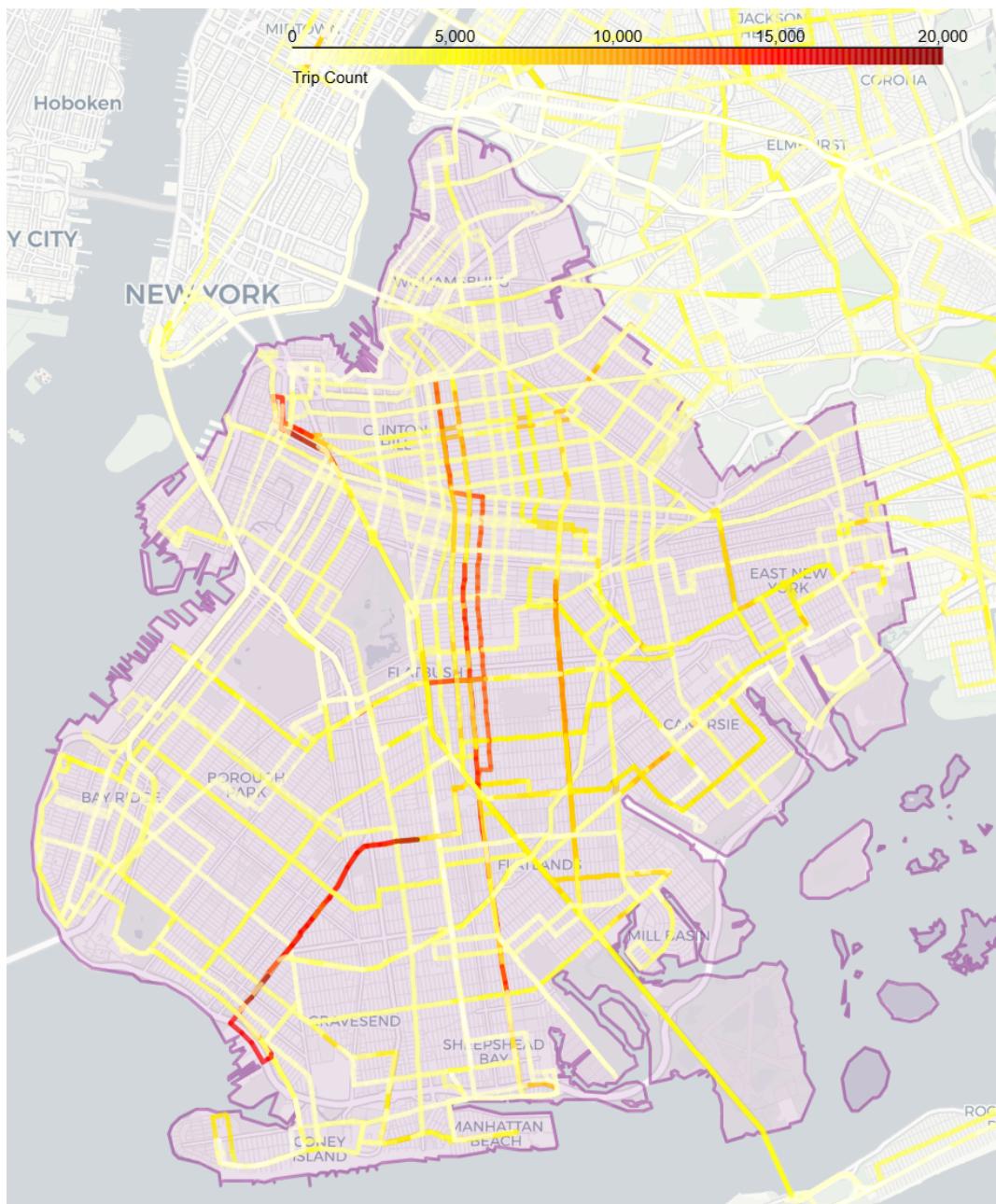


Figura 6.2-1: Mapa de segmentos en Brooklyn según cantidad de viajes (15 al 31 de julio de 2025). El color indica la cantidad de servicios programados para cada tramo. Datos de MTA NYCT y MTA BC

Si bien la estructura general de los segmentos con mayor cantidad de viajes se mantiene respecto de la **figura 6.1-1** (con una fuerte concentración en los corredores troncales del centro del distrito), la incorporación de MTA BC permite captar mayor cobertura en áreas que antes aparecían con menor densidad o directamente no figuraban.

Un ejemplo claro se da en la zona sureste de Brooklyn, donde se observa una mayor presencia de segmentos activos en barrios como Mill Basin y Georgetown. Estas zonas habían quedado prácticamente fuera del mapa en la visualización anterior.

Además, se observa que las rutas de MTA BC permiten una conexión directa entre Brooklyn y Queens a través del Marine Parkway Bridge, algo no capturado en la visualización original.

7. Identificación de *Hotspots* y Duplicación de Rutas

En esta parte del análisis se quiere identificar las zonas de Brooklyn donde se planifica una alta concentración de circulación de buses, que se llamarán *hotspots*. Un *hotspot* es un área donde, en un momento determinado, se espera que pasen muchos servicios. Detectar estas zonas resulta útil para entender la distribución espacial y temporal de la oferta de transporte, y también para evaluar si existen rutas que podrían estar solapándose más de lo necesario.

Para llevar adelante este análisis, se trabaja con una ventana temporal acotada: el lunes 21 de julio de 2025 entre las 9 y las 10 de la mañana. Para poder hablar de duplicación de recorridos es necesario que los viajes ocurran en el mismo momento. No alcanza con saber que varias rutas pasan por una misma calle a lo largo del día; lo relevante es que lo hagan al mismo tiempo.

En la sección 7.1 se visualiza la cantidad de viajes proyectados por segmento durante ese horario, como se hizo en secciones anteriores. Sin embargo, rápidamente se evidencia que este enfoque tiene limitaciones a la hora de detectar *hotspots*. Como se cuenta la cantidad de viajes por cada segmento de forma individual, si dos segmentos se superponen parcialmente pero no coinciden exactamente, el conteo queda disperso. En vez de identificar un punto con alta densidad de tránsito, aparecen múltiples segmentos cercanos con valores fragmentados, sin capturar la intensidad real del área.

Para resolver este problema, en la sección 7.2 se propone el uso de una grilla espacial de celdas regulares de 500x500 metros. A partir de esta grilla, se contabiliza la cantidad total de viajes que atraviesan cada celda. De esta manera, se logra una representación más precisa del espacio recorrido por los buses, agrupando adecuadamente los segmentos próximos y revelando con mayor claridad los *hotspots* de la red.

Finalmente, en la sección 7.3 se analizan las rutas que atraviesan esas celdas con alta densidad. El objetivo es detectar posibles casos de duplicación: situaciones en las que varias rutas comparten tramos muy similares, durante el mismo rango horario. Este tipo de superposición podría reflejar una planificación redundante.

El código utilizado en esta sección se encuentra dentro de [esta carpeta](#) del repositorio de GitHub del trabajo y las visualizaciones dentro de [esta carpeta](#) de Drive.

7.1 Visualización de los viajes proyectados por segmento

El primer paso del análisis consiste en observar cómo se distribuyen los viajes planificados para un día y horario específico: el lunes 21 de julio de 2025 entre las 9 y las 10 de la mañana. Para eso, se construyó una vista que filtra los datos de los esquemas `busco` y `brooklyn`, considerando únicamente los viajes que atraviesan el distrito de Brooklyn durante ese horario. Al igual que en la sección anterior, la vista agrupa los segmentos por origen y destino, y calcula cuántos viajes los recorren en ese intervalo, así como las rutas involucradas.

```
CREATE VIEW brooklyn.segments_9_10 AS
WITH boroughs AS (
    SELECT boroname, geom_4326 as geom
    FROM public.ny_adm
    WHERE boroname IN ('Brooklyn')
),
busco_morning AS (
    SELECT
        c.from_stop_id,
        c.from_stop_name,
        c.to_stop_id,
        c.to_stop_name,
        s.geometry,
        COUNT(c.trip_id) AS notrips,
        STRING_AGG(DISTINCT c.route_short_name, ',') AS routes
    FROM busco.segments s
    JOIN busco.connections c
        ON s.route_id = c.route_id
        AND s.direction_id = c.direction_id
        AND s.start_stop_id = c.from_stop_id
        AND s.end_stop_id = c.to_stop_id
```

```

JOIN boroughs b
    ON ST_Intersects(b.geom, s.geometry)
WHERE date = '2025-07-21'
    AND t_arrival >= '2025-07-21 09:00:00 America/New_York'
    AND t_arrival < '2025-07-21 10:00:00 America/New_York'
    GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id,
c.to_stop_name, s.geometry
),
brooklyn_morning AS (
    SELECT
        c.from_stop_id,
        c.from_stop_name,
        c.to_stop_id,
        c.to_stop_name,
        s.geometry,
        COUNT(c.trip_id) AS notrips,
        STRING_AGG(DISTINCT c.route_short_name, ',') AS routes
    FROM brooklyn.segments s
    JOIN brooklyn.connections c
        ON s.route_id = c.route_id
        AND s.direction_id = c.direction_id
        AND s.start_stop_id = c.from_stop_id
        AND s.end_stop_id = c.to_stop_id
    JOIN boroughs b
        ON ST_Intersects(b.geom, s.geometry)
    WHERE date = '2025-07-21'
        AND t_arrival >= '2025-07-21 09:00:00 America/New_York'
        AND t_arrival < '2025-07-21 10:00:00 America/New_York'
        GROUP BY c.from_stop_id, c.from_stop_name, c.to_stop_id,
c.to_stop_name, s.geometry
)

SELECT
    from_stop_id,
    from_stop_name,
    to_stop_id,
    to_stop_name,
    geometry,
    SUM(notrips) AS notrips,
    STRING_AGG(routes, ',') AS routes
FROM (
    SELECT * FROM busco_morning
    UNION ALL
    SELECT * FROM brooklyn_morning
) AS all_segments
GROUP BY
    from_stop_id,
    from_stop_name,
    to_stop_id,
    to_stop_name,
    geometry;

```

Los datos obtenidos utilizan una visualización que muestra la cantidad de viajes (`notrips`) por cada segmento de la red, utilizando el mismo código que en la sección anterior, pero adaptado para trabajar con la vista recién creada. Se establece un valor máximo de corte en 80 para la escala de colores, de forma de mantener la consistencia con los nuevos datos. El resultado se muestra en la **Figura 7.1-1**.

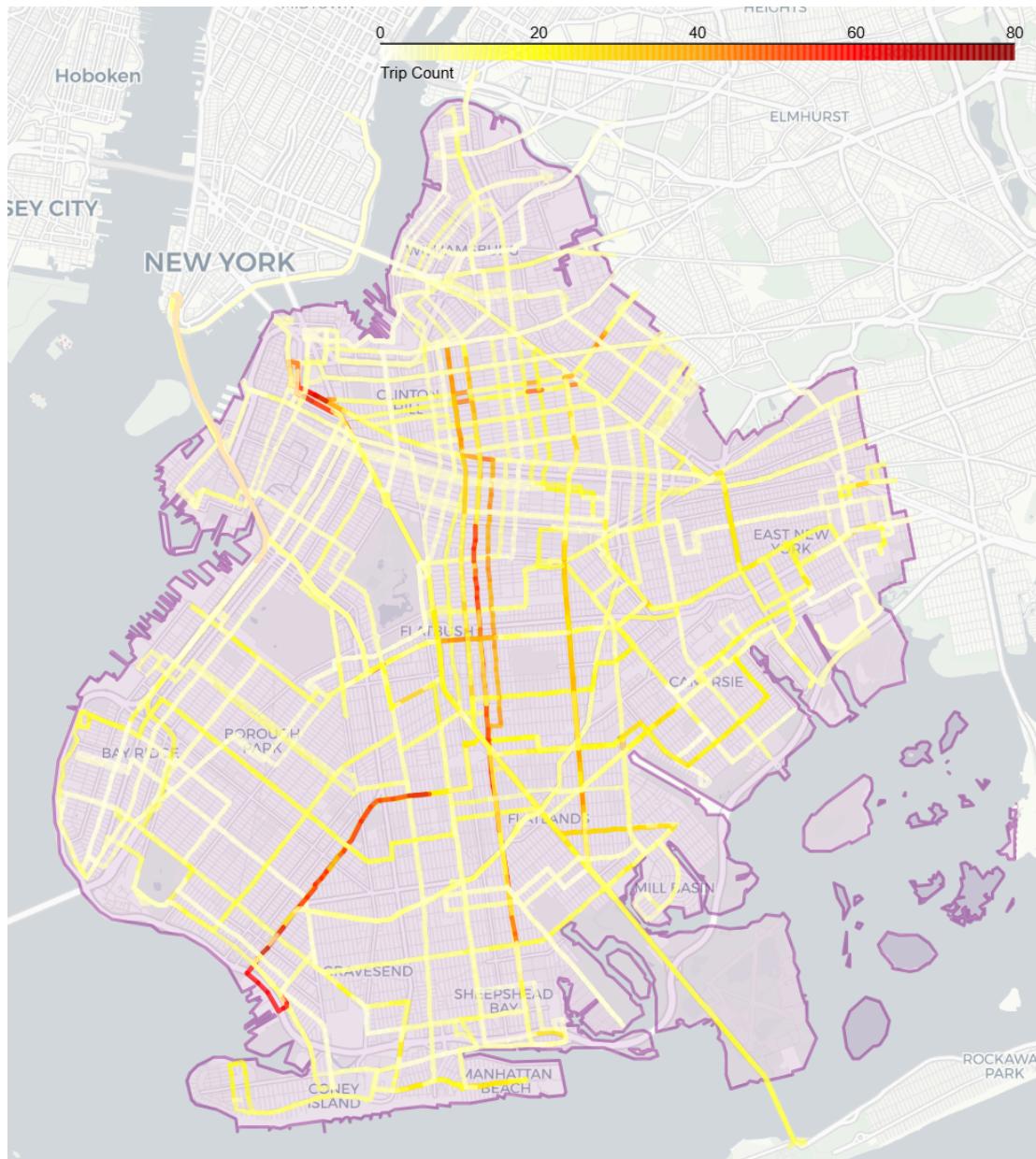


Figura 7.1-1: Cantidad de viajes por segmento en Brooklyn, lunes 21/07/2025 entre las 9 y las 10 AM. Los colores indican el número de viajes planificados: desde blanco y amarillo (pocos viajes) hasta rojo oscuro (muchos viajes). Datos de MTA NYCT y MTA BC

Esta visualización permite identificar de forma general los segmentos más utilizados, pero también revela una limitación importante: al contar los viajes por segmento de manera individual, no se capta correctamente la acumulación de tránsito en zonas donde varios segmentos se superponen sin coincidir exactamente. Esto puede hacer que la densidad de tránsito planificada parezca dispersa en lugar de concentrada, dificultando la identificación precisa de los *hotspots*, como se puede observar en la **Figura 7.1-2**.

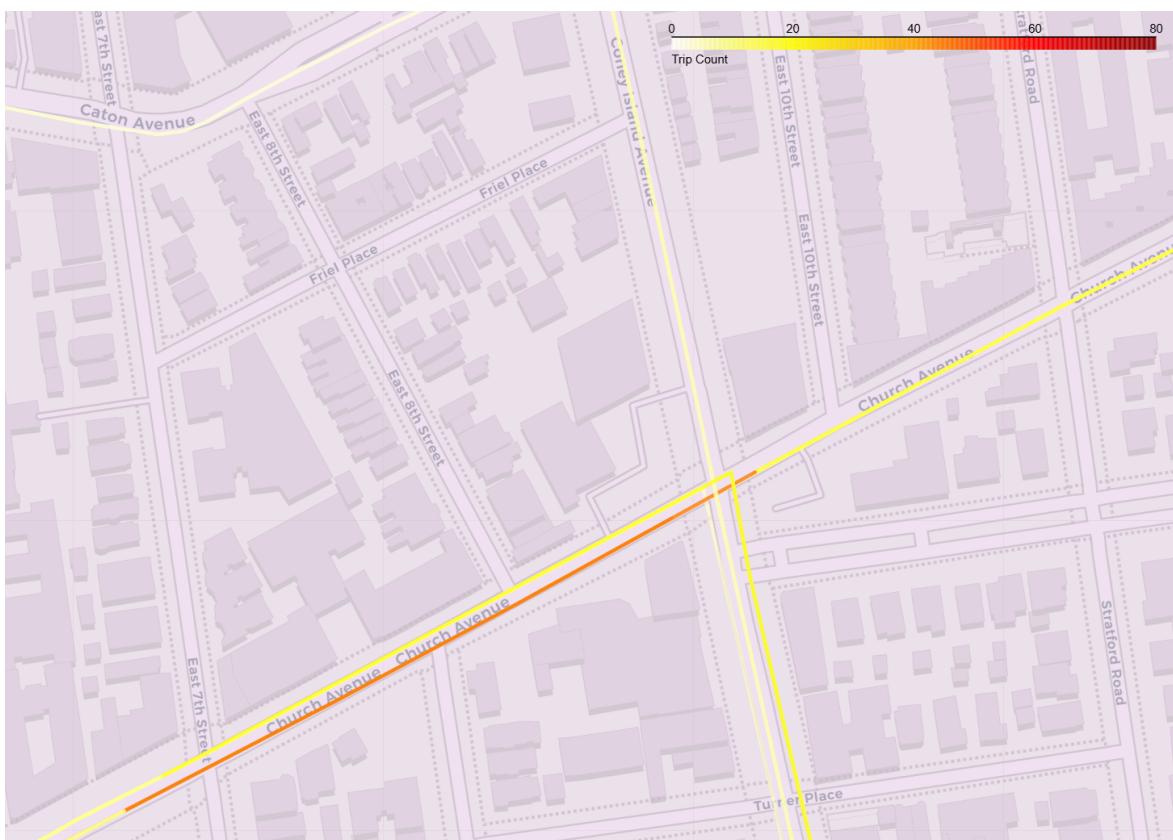


Figura 7.1-2: Ejemplo de superposición de segmentos en Church Avenue y Coney Island Avenue. Aunque los recorridos pasan por las mismas calles, los segmentos son distintos y cada uno tiene su propio valor de `notrips`. Esto fragmenta la representación del tránsito en la zona.

Esta limitación sugiere que el enfoque por segmento, aunque útil para ciertos análisis, no es suficiente para identificar correctamente las zonas con mayor circulación planificada. La superposición parcial de segmentos impide que se visualice claramente la intensidad del tránsito en puntos específicos del territorio. Por esta razón, en la próxima sección se propone un enfoque alternativo basado en una grilla espacial, que permite

capturar de manera más precisa los lugares por donde efectivamente circulan más buses, independientemente de la geometría exacta de cada recorrido.

7.2 Identificación de *hotspots* mediante una grilla espacial

Para superar las limitaciones del enfoque anterior, en esta sección se propone una metodología alternativa basada en el uso de una grilla espacial regular. Se construirán grillas de celdas de diferentes tamaños sobre el distrito de Brooklyn, y se contabilizará cuántos viajes planificados atraviesan cada celda durante el mismo intervalo horario (lunes 21/07 de 9 a 10 h). Se evaluaron distintos tamaños de grilla 100 m, 200 m, 500 m). Se explica cómo construir la grilla de 100m pero el procedimiento es análogo para todas.

```
CREATE TABLE brooklyn.grid_100m_32118 AS
WITH brooklyn_geom AS (
    SELECT geom_32118
    FROM public.ny_adm
    WHERE boroname = 'Brooklyn'
),
raw_grid AS (
    SELECT grid.geom
    FROM ST_SquareGrid(
        100,
        (SELECT ST_Envelope(geom_32118) FROM brooklyn_geom)
    ) AS grid
),
filtered_grid AS (
    SELECT grid.geom
    FROM raw_grid AS grid, brooklyn_geom
    WHERE ST_Intersects(grid.geom, brooklyn_geom.geom_32118)
)
SELECT
    row_number() OVER () AS grid_id,
    geom,
    0 as notrips,
    '' as routes
FROM filtered_grid;
```

La grilla se crea mediante la función `ST_SquareGrid`¹³ de PostGIS, que permite generar un grid regular de celdas cuadradas a partir de un `Envelope`. Se filtran las celdas que efectivamente intersectan con Brooklyn. A cada celda se le asigna un `grid_id`, y se

¹³ https://postgis.net/docs/ST_SquareGrid.html

initializan los valores de `notrips` y `routes` en cero y vacío, respectivamente, para poder cargarlos posteriormente.

El siguiente código permite visualizar la grilla generada:

```
def load_grid_100m(engine):
    query = """
        SELECT grid_id, ST_Transform(geom, 4326) AS geometry
        FROM brooklyn.grid_100m_32118;
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def visualize_grid(grid_gdf):
    # Centrar el mapa más o menos en Brooklyn
    transport_map = f.Map(location=[40.65, -73.95],
                           tiles='CartoDB positron', zoom_start=12)

    for _, row in grid_gdf.iterrows():
        geojson = f.GeoJson(
            data={
                "type": "Feature",
                "geometry": row['geometry'].__geo_interface__,
                "properties": {"grid_id": row["grid_id"]}
            },
            style_function=lambda x: {
                'fillColor': '#0080ff',
                'color': '#004080',
                'weight': 1,
                'fillOpacity': 0.2,
                'opacity': 0.5
            },
            tooltip=f"Grid ID: {row['grid_id']}"
        )
        geojson.add_to(transport_map)

    return transport_map
```

El resultado es el siguiente

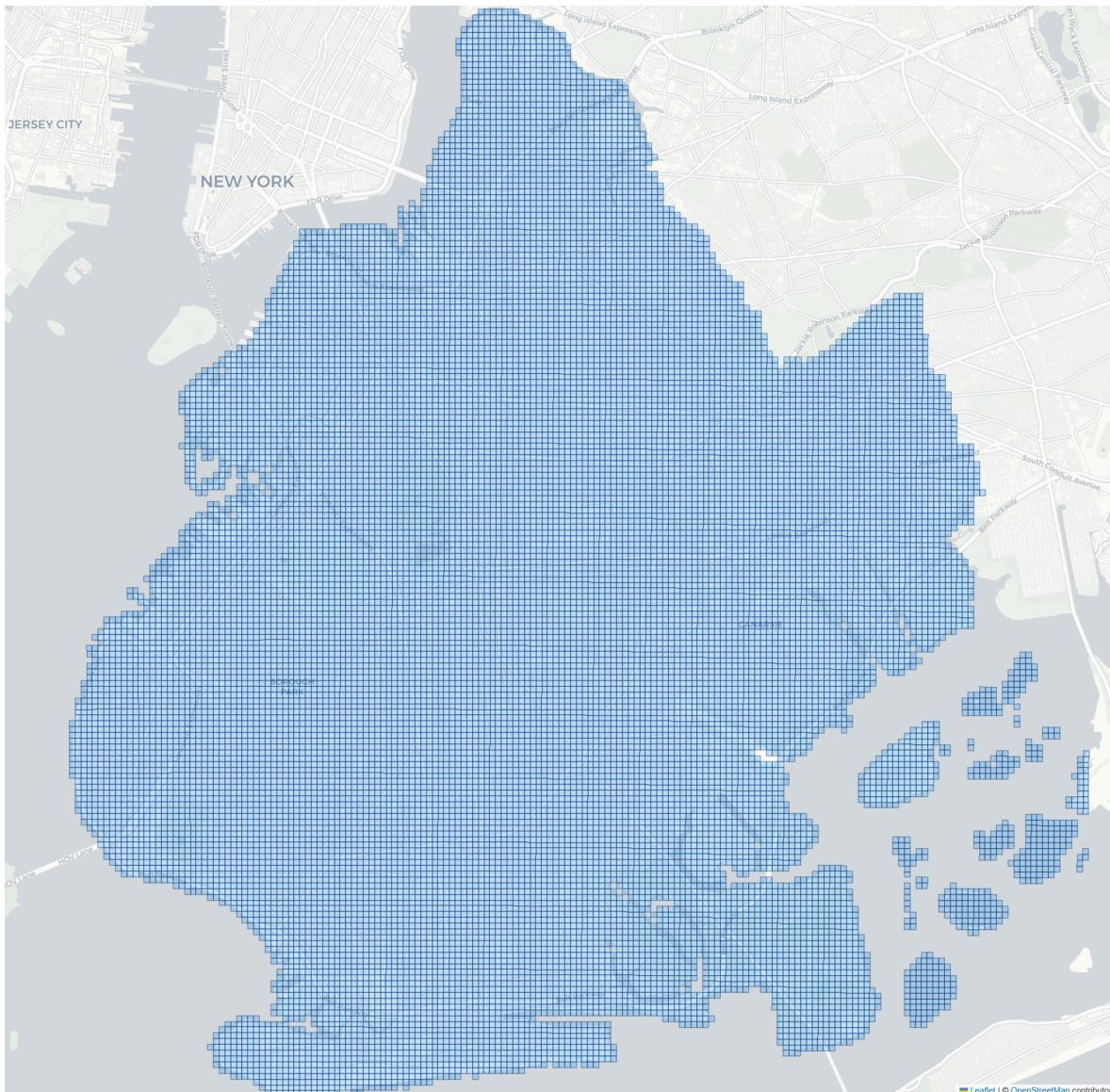


Figura 7.2-1: Grilla de celdas de 100x100 metros generada sobre el distrito de Brooklyn.

Una vez creada la grilla, el siguiente paso es contar cuántos viajes planificados cruzan cada una de sus celdas. Para ello, se utiliza una vista que calcula cuántos segmentos atraviesan cada celda, tomando como referencia el punto medio del segmento. Este punto se ubica dentro de una única celda mediante `ST_Contains`. De esta manera, cada segmento se asigna a una sola celda, evitando el doble conteo en celdas adyacentes.

Se pueden actualizar los campos `notrips` y `routes` de la tabla de grilla:

```
UPDATE brooklyn.grid_100m_32118 g
SET
    notrips = sub.total_notrips,
    routes = sub.routes
FROM (
    SELECT
        g.grid_id,
        SUM(s.notrips) AS total_notrips,
        STRING_AGG(DISTINCT s.routes, ',') AS routes
    FROM brooklyn.grid_100m_32118 g
    JOIN brooklyn.segments_9_10 s
        ON ST_Contains(
            g.geom,
            ST_Transform(
                ST_LineInterpolatePoint(s.geometry, 0.5),
                32118
            )
        )
    GROUP BY g.grid_id
) AS sub
WHERE g.grid_id = sub.grid_id;
```

Este enfoque permite simplificar el cálculo al contar cada segmento una sola vez, en su ubicación más representativa.

Para visualizar los resultados sobre la grilla ya poblada con los conteos de viajes, se utiliza el siguiente código en Python:

```
def load_grid_100m(engine):
    query = """
        SELECT grid_id, notrips, routes, ST_Transform(geom, 4326) AS geometry
        FROM brooklyn.grid_100m_32118;
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def load_brooklyn_boundary(engine):
    query = """
        SELECT ST_Transform(geom_32118, 4326) AS geometry
        FROM public.ny_adm
        WHERE boroname = 'Brooklyn';
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def visualize_grid(grid_gdf, brooklyn_boundary, cutoff=100):
    transport_map = f.Map(location=[40.65, -73.95],
```

```
tiles='CartoDB positron', zoom_start=11)

# Colormap for notrips
colormap = cm.LinearColormap(
    colors=['white', 'yellow', 'orange', 'red', 'darkred'],
    vmin=0,
    vmax=cutoff,
    caption='Trip Count'
)
colormap.add_to(transport_map)

# --- Borde de Brooklyn ---
for _, row in brooklyn_boundary.iterrows():
    geojson = f.GeoJson(
        data={
            "type": "Feature",
            "geometry": row['geometry'].__geo_interface__,
            "properties": {}
        },
        style_function=lambda x: {
            'fillColor': 'transparent',
            'color': 'purple',
            'weight': 2,
            'fillOpacity': 0.0,
            'opacity': 0.7
        }
    )
    geojson.add_to(transport_map)

# --- Cuadrículas coloreadas según notrips ---
for _, row in grid_gdf.iterrows():
    trip_count = min(row['notrips'], cutoff)
    color = colormap(trip_count)

    geo_json = f.GeoJson(
        data={
            "type": "Feature",
            "geometry": row['geometry'].__geo_interface__,
            "properties": {
                "grid_id": row['grid_id'],
                "notrips": row['notrips'],
                "routes": row['routes']
            }
        },
        style_function=lambda x, color=color: {
            'fillColor': color,
            'color': 'grey',
            'weight': 0.5,
            'fillOpacity': 0.6,
            'opacity': 0.3
        },
        tooltip=GeoJsonTooltip(
            fields=['grid_id', 'notrips', 'routes'],
            aliases=['Grid ID', 'Trip Count', 'Routes'],
        )
    )
    geo_json.add_to(transport_map)
```

```
        localize=True
    )
geo_json.add_to(transport_map)

return transport_map
```

Este código permite visualizar cada celda coloreada según la cantidad de viajes que contiene, utilizando una escala de color progresiva desde blanco (pocos viajes) hasta rojo oscuro (muchos viajes). También se superpone el contorno del distrito de Brooklyn como referencia espacial.

El resultado es el siguiente:

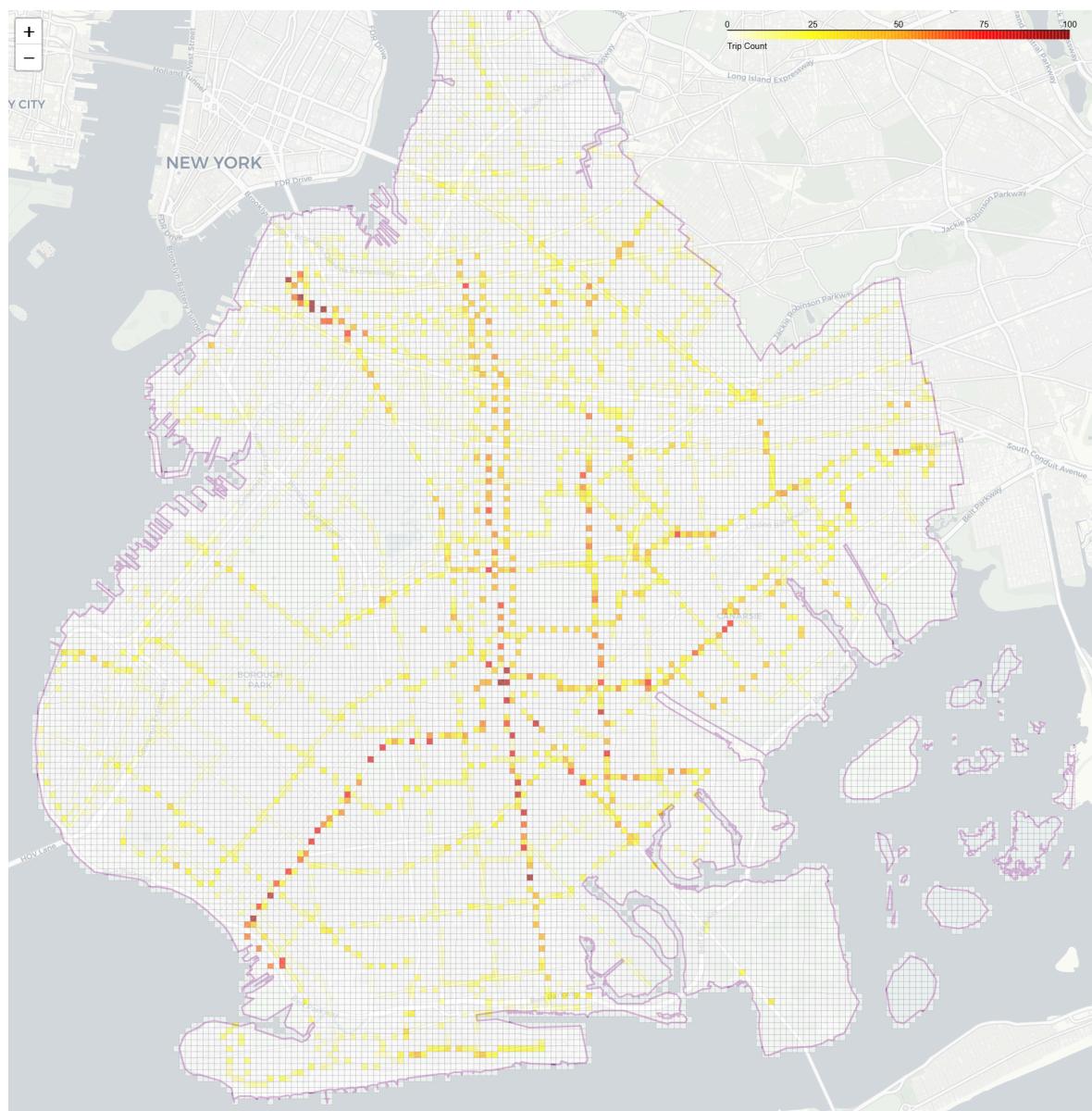


Figura 7.2-2: Visualización de la grilla de 100x100 metros con sus respectivos notrips, entre las 9 y 10 h del lunes 21/07/2025.

Este tamaño de grilla nos permite detectar con bastante precisión calles que concentran una gran cantidad de viajes planificados.

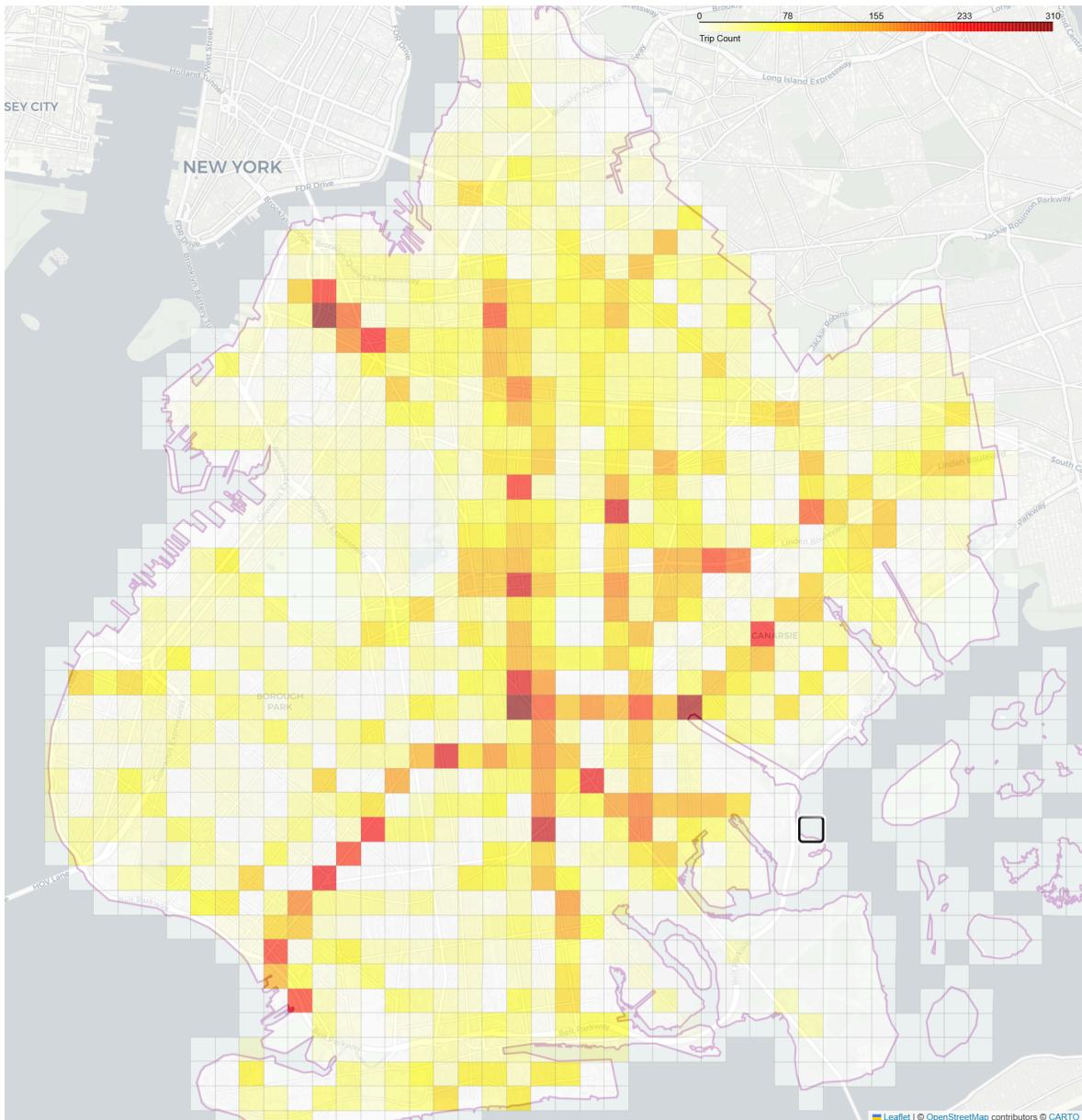


Figura 7.2-3: Visualización de la grilla de 200x200 metros con sus respectivos `notrips`, entre las 9 y 10 h del lunes 21/07/2025.

La visualización en la **Figura 7.2-3** nos permite acercarnos a una representación más espacial de la densidad de tránsito, resaltando tanto zonas como calles específicas con una alta cantidad de viajes proyectados.

Si llevamos el análisis a una grilla más gruesa, como la de 500x500 metros, se empieza a consolidar aún más la información, destacando áreas completas que presentan una concentración notable de tránsito:

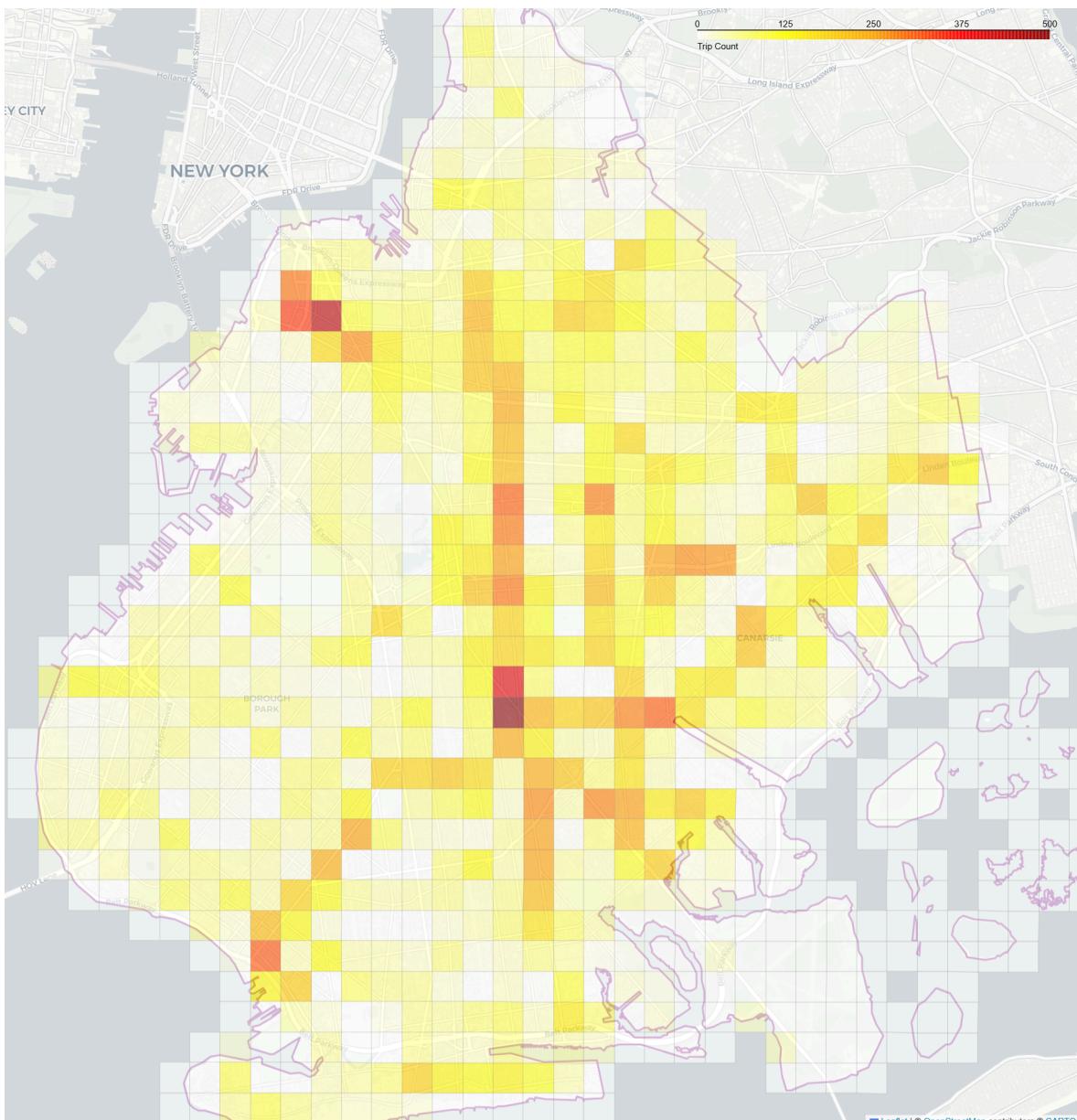


Figura 7.2-4: Visualización de la grilla de 500x500 metros con sus respectivos `notrips`, entre las 9 y 10 h del lunes 21/07/2025.

En la **Figura 7.2-4**, se destacan claramente dos regiones con una concentración significativamente mayor de viajes: la zona de Downtown Brooklyn y el cruce entre Nostrand Avenue y Flatbush Avenue. Tomaremos como *hotspots* dos celdas representativas de cada una de estas regiones:

- Downtown Brooklyn: celdas con grid_id 186 y 217 (ver Figura 7.2-5).
- Nostrand–Flatbush Intersection: celdas con grid_id 405 y 406 (ver Figura 7.2-6).

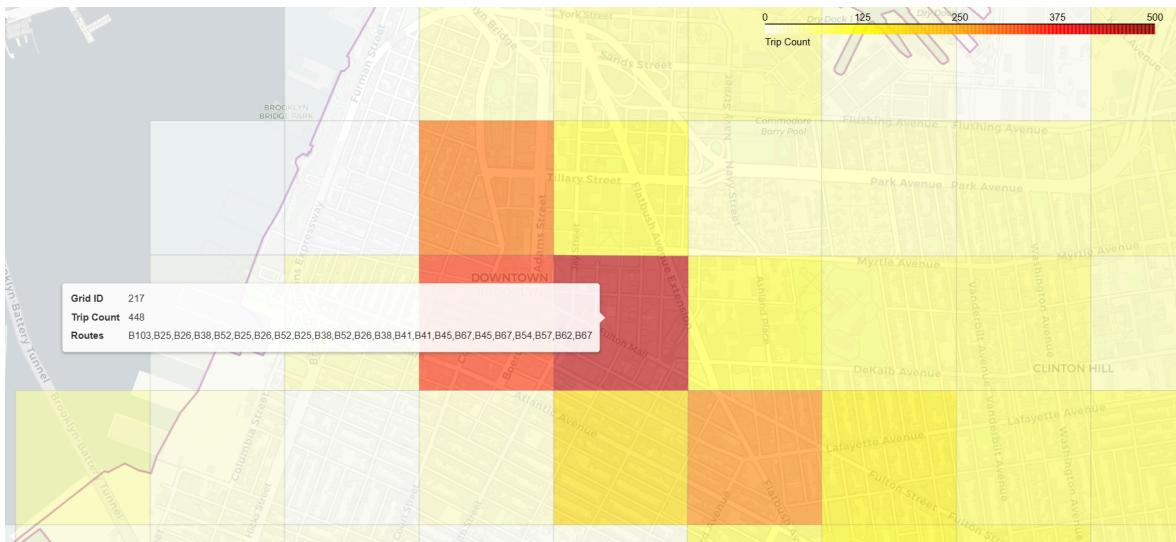


Figura 7.2-5: Zoom sobre la grilla de 500x500 m en la región de Downtown Brooklyn.

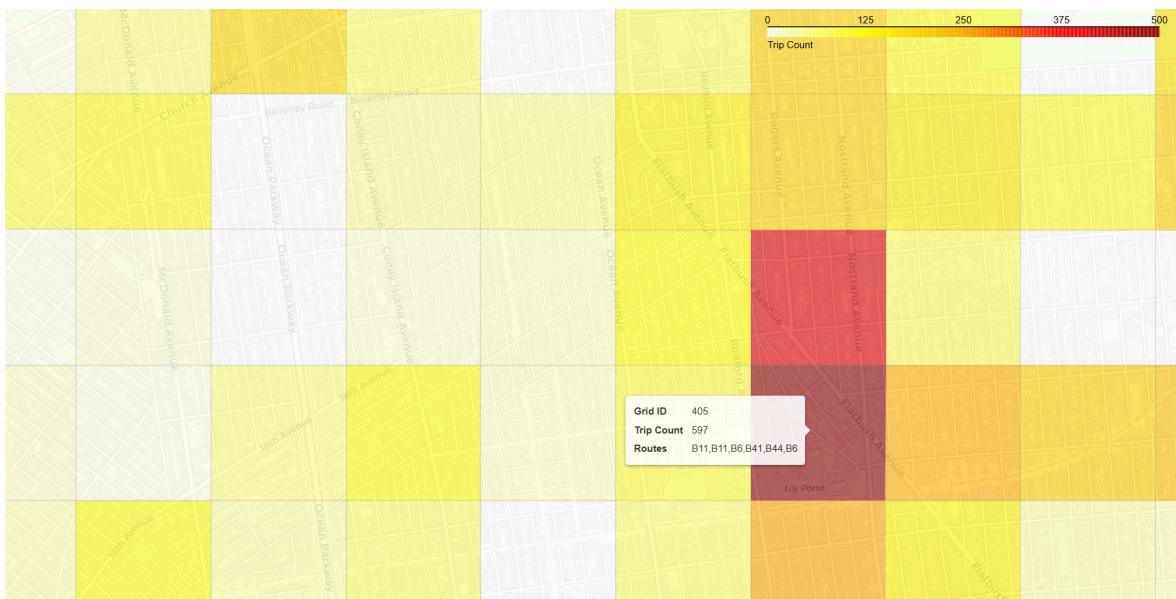


Figura 7.2-5: Zoom sobre la grilla de 500x500 m en la región de Downtown Brooklyn.

Cabe destacar que si se utilizara únicamente el enfoque por segmentos, la alta densidad de tránsito en Downtown Brooklyn resultaría visible, ya que los segmentos

coinciden espacialmente en gran medida. Sin embargo, en la intersección de Nostrand y Flatbush esto no se destacaría de la misma forma: al haber una mayor dispersión y superposición de segmentos, la densidad real quedaría enmascarada, se podría evidenciar la carga de Flatbush Avenue, pero no se evidenciaría tan claramente que hay particularmente más carga en su intersección con Nostrand Avenue. Esto se ilustra en la **Figura 7.2-7**, donde viendo ambos enfoques uno al lado del otro, podemos ver que el enfoque planteado en esta sección permite identificar más claramente que hay dos sectores con una gran cantidad de viajes planificados.

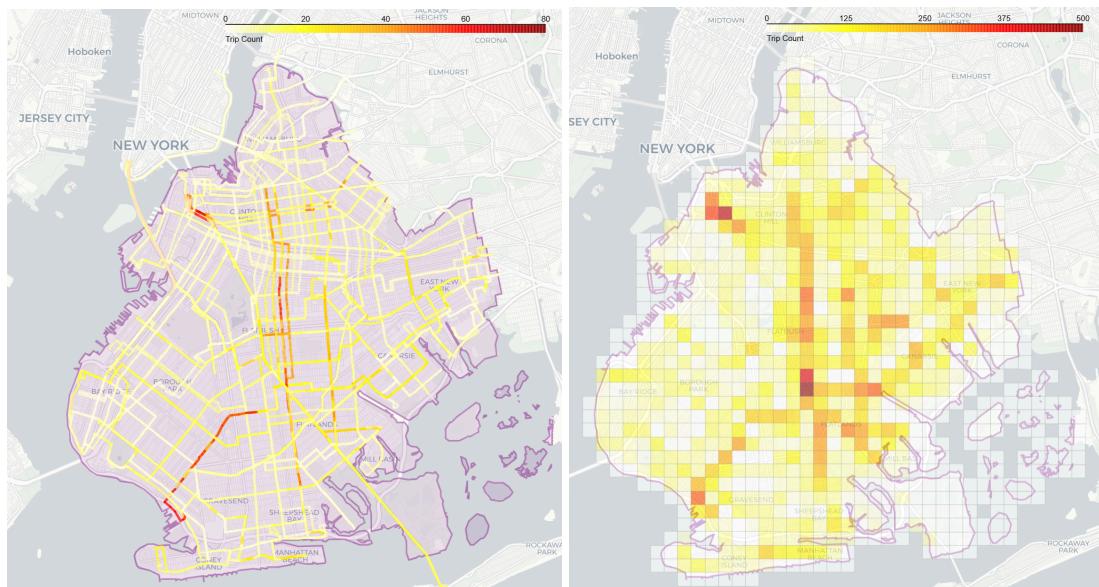


Figura 7.2-7: Comparación entre el enfoque por segmentos (izquierda) y el enfoque por grilla (derecha).

En resumen, este enfoque basado en grillas permitió identificar de manera precisa dos zonas del distrito con una densidad de tránsito planificado significativamente mayor al resto. Se trata de una representación mucho más robusta que el conteo por segmentos individuales, ya que evita los problemas de superposición parcial y ofrece una visión más adecuada de la circulación real esperada. Estas zonas seleccionadas servirán como caso de estudio en la siguiente sección, donde se analizará la posibilidad de duplicaciones en los recorridos que las atraviesan.

7.3 Evaluación de duplicación en recorridos de buses

Una vez identificados los *hotspots*, se procede a analizar qué rutas de bus atraviesan esas áreas. El objetivo principal es detectar posibles situaciones de duplicación entre rutas distintas: es decir, casos en los que varias líneas no solo pasan por las zonas de mayor tránsito planificado, sino que además comparten tramos significativos a lo largo de su recorrido, en el horario analizado. Cabe aclarar que este análisis no considera situaciones de duplicación dentro de una misma ruta (por ejemplo, múltiples viajes del mismo ramal o de ramales diferentes). En esos casos, se consideró que el solapamiento podría estar relacionado con decisiones operativas sobre frecuencia o cobertura horaria, más que con una redundancia estructural entre líneas distintas del sistema.

Se analizará únicamente el *hotspot* de Downtown Brooklyn, dado que analizar el otro *hotspot* identificado requeriría el mismo procedimiento, por lo que a los efectos de este trabajo sería redundante.

Para llevar a cabo este análisis, se utiliza la tabla `shapes_aggregated`, que representa los recorridos reales de cada ruta. Luego, se filtran aquellas rutas que intersectan los *hotspots* identificados previamente, y que estarán activas durante el horario de análisis (lunes 21 de julio de 2025, entre las 9 y las 10 h).

Debido a que esta operación espacial puede ser costosa, primero se transforman las geometrías al sistema proyectado EPSG:32118 y se crean índices espaciales para acelerar la ejecución:

```
CREATE MATERIALIZED VIEW busco.shapes_aggregated_32118 AS
SELECT
    shape_id,
    ST_Transform(ST_SetSRID(shape, 4326), 32118) AS shape
FROM busco.shapes_aggregated;
CREATE MATERIALIZED VIEW brooklyn.shapes_aggregated_32118 AS
SELECT
    shape_id,
    ST_Transform(ST_SetSRID(shape, 4326), 32118) AS shape
```

```
FROM brooklyn.shapes_aggregated;
CREATE INDEX idx_segments_aggregated_32118_geom
ON busco.shapes_aggregated_32118
USING GIST (shape);

CREATE INDEX idx_segments_aggregated_32118_geom
ON brooklyn.shapes_aggregated_32118
USING GIST (shape);
```

Luego, se define una vista que recopila todos los recorridos que intersectan al menos una de las celdas del *hotspot* (IDs 186 y 217) en el horario de interés. La consulta toma datos de ambos esquemas (*busco* y *brooklyn*).

```
CREATE VIEW brooklyn.hot_spot_routes AS
WITH grids AS (
    SELECT *
    FROM brooklyn.grid_500m_32118
    WHERE grid_id IN (186, 217)
),
busco_trips_shapes AS (
    SELECT
        c.direction_id,
        c.trip_id,
        t.route_id,
        r.route_short_name,
        t.trip_headsign,
        t.shape_id,
        s.shape AS shape_geom
    FROM busco.connections c
    JOIN busco.trips t
        ON c.trip_id = t.trip_id
    JOIN busco.shapes_aggregated_32118 s
        ON t.shape_id = s.shape_id
    JOIN busco.routes r
        ON t.route_id = r.route_id
    JOIN grids g
        ON ST_Intersects(s.shape, g.geom)
    WHERE c.date = '2025-07-21'
        AND c.t_arrival >= '2025-07-21 09:00:00 America/New_York'
        AND c.t_arrival < '2025-07-21 10:00:00 America/New_York'
),
brooklyn_trips_shapes AS (
    SELECT
        c.direction_id,
        c.trip_id,
        t.route_id,
        r.route_short_name,
        t.trip_headsign,
        t.shape_id,
        s.shape AS shape_geom
```

```

FROM brooklyn.connections c
JOIN brooklyn.trips t
  ON c.trip_id = t.trip_id
JOIN brooklyn.shapes_aggregated_32118 s
  ON t.shape_id = s.shape_id
JOIN brooklyn.routes r
  ON t.route_id = r.route_id
JOIN grids g
  ON ST_Intersects(s.shape, g.geom)
WHERE c.date = '2025-07-21'
  AND c.t_arrival >= '2025-07-21 09:00:00 America/New_York'
  AND c.t_arrival < '2025-07-21 10:00:00 America/New_York'
)

SELECT DISTINCT ON (shape_id) *
FROM (
  SELECT * FROM busco_trips_shapes
  UNION ALL
  SELECT * FROM brooklyn_trips_shapes
) all_trips;

```

El resultado se visualiza utilizando un mapa interactivo en Folium. Cada recorrido se colorea de acuerdo a su `route_id`, y las rutas se agrupan por `shape_id`, lo que permite comparar visualmente los distintos recorridos involucrados en los hotspots:

```

def create_routes_map(routes_gdf, brooklyn_boundary=None):
    base_map = f.Map(location=[40.65, -73.95], tiles='CartoDB
positron', zoom_start=11)
    # --- Color por route_id con Set3 ---
    route_ids = routes_gdf['route_id'].unique()
    color_palette = sns.color_palette("Set3",
len(route_ids)).as_hex()
    color_map = dict(zip(route_ids, color_palette))

    # --- Group by shape_id ---
    route_count = 0
    for shape_id, shape_group in routes_gdf.groupby('shape_id'):
        route_id = shape_group['route_id'].iloc[0]
        route_name = shape_group['route_short_name'].iloc[0]
        direction = shape_group['direction_id'].iloc[0]
        color = color_map.get(route_id, 'black')

        layer_name = f'{route_name} (dir: {direction}, shape_id:
{shape_id})'
        feature_group = f.FeatureGroup(name=layer_name,
show=True)

        for geometry in shape_group.geometry:
            if geometry.geom_type == 'LineString':
                coords = [(lat, lon) for lon, lat in
geometry.coords]
                feature_group.add_line(coords, color=color,
width=3)

```

```
geometry.coords]
    f.PolyLine(
        locations=coords,
        color=color,
        weight=3,
        opacity=0.8,
        tooltip=layer_name
    ).add_to(feature_group)
else:
    print(f"[WARN] Geometry type not supported:
{geometry.geom_type} (shape_id: {shape_id})")

feature_group.add_to(base_map)
route_count += 1

f.LayerControl(collapsed=False).add_to(base_map)
print(f"Map contains {route_count} transit routes.")
return base_map
```

El resultado es el siguiente:

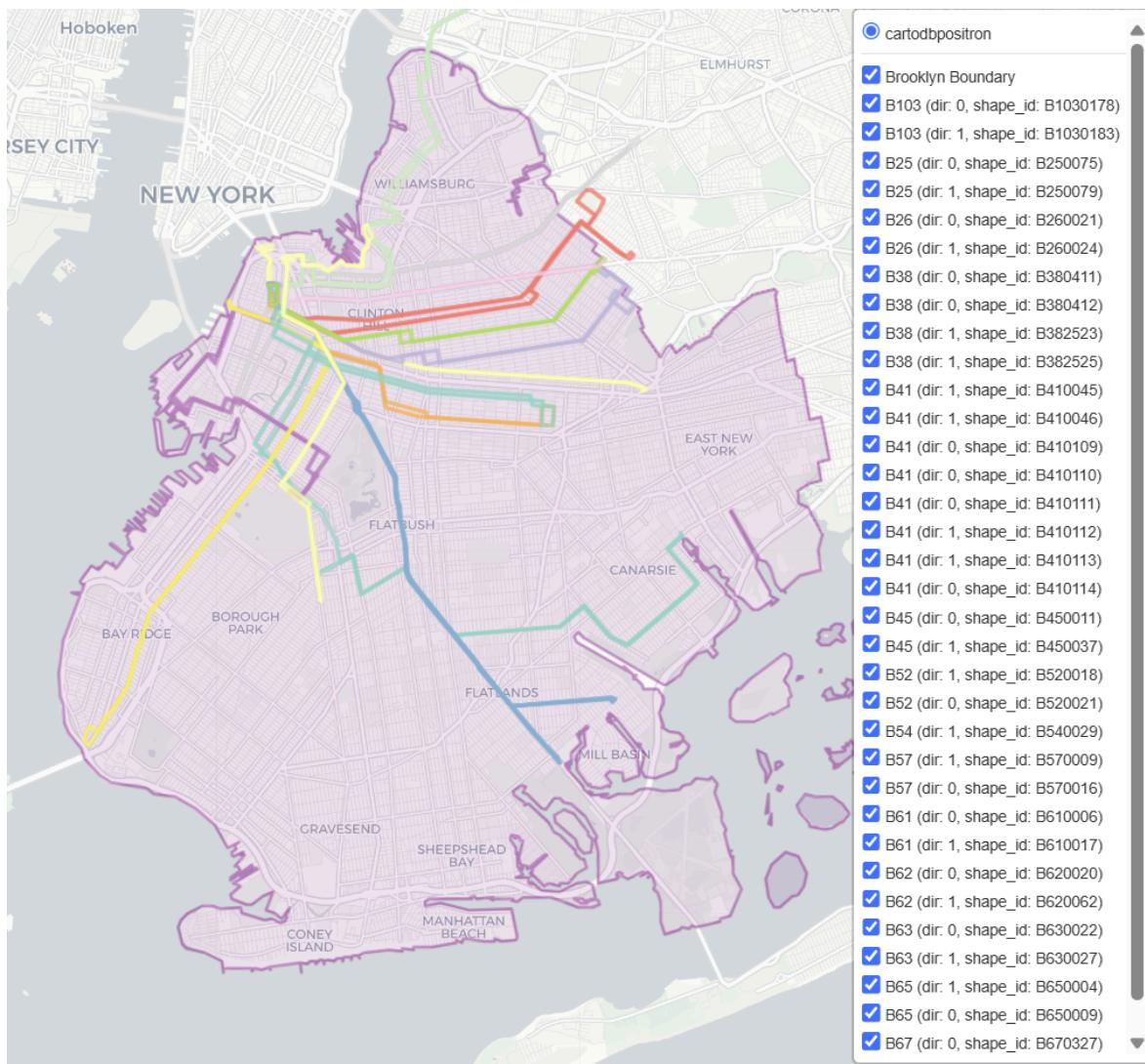


Figura 7.3-1: Rutas que atraviesan las celdas hotspot (IDs 186 y 217) entre las 9 y 10 h del lunes 21/07/2025. Cada recorrido está coloreado según su `route_id`.

Para facilitar la comparación visual entre los recorridos que atraviesan los *hotspots*, se seleccionaron distintos subconjuntos de rutas en función de su cercanía espacial y posibles similitudes. En la **Figura 7.3-2** se presentan los recorridos de las rutas 62, 57, 38, 52, 54, 26, 41, 103 y 63.

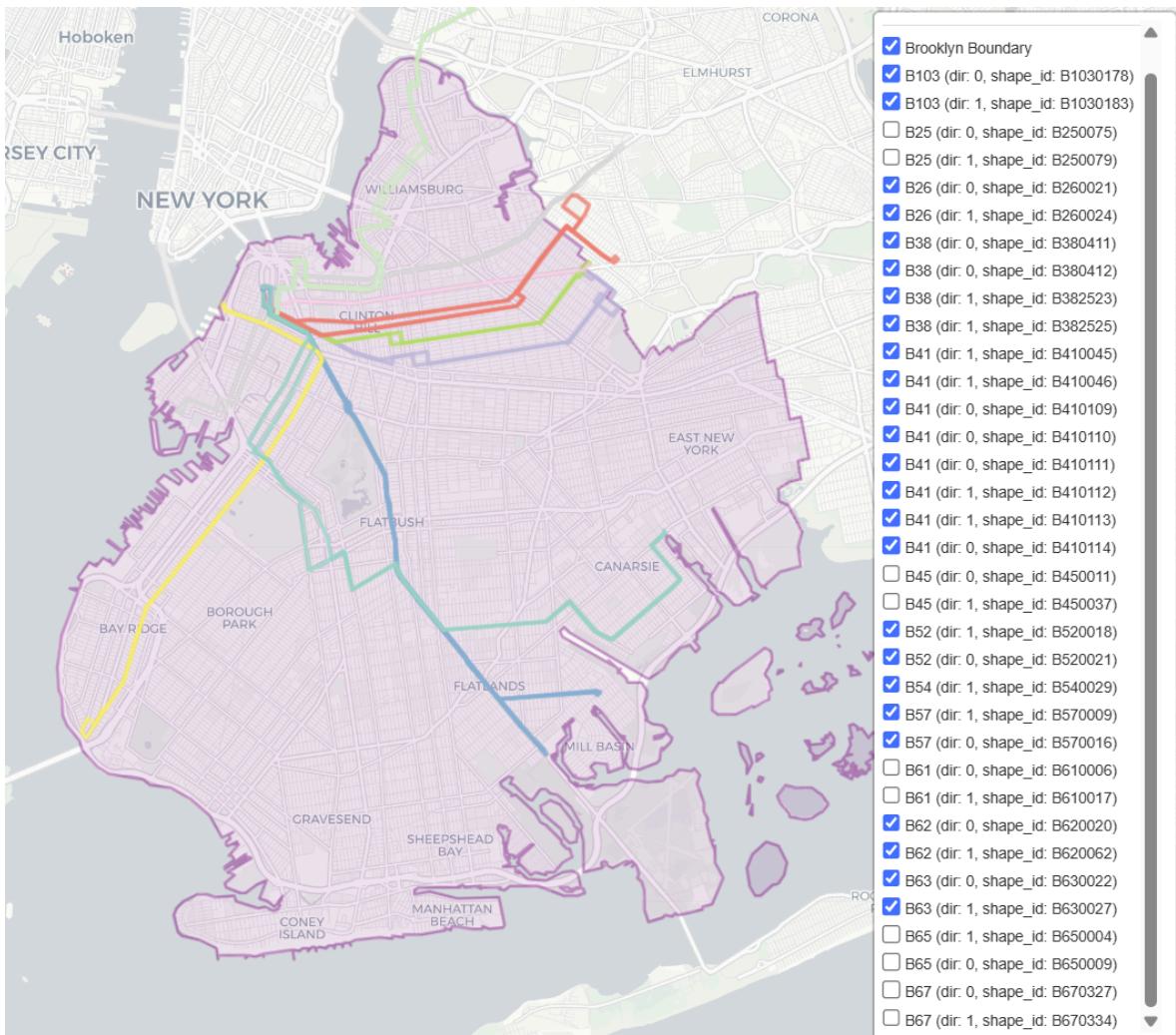


Figura 7.3-2: Rutas que atraviesan las celdas hotspot (IDs 186 y 217) entre las 9 y 10 h del lunes 21/07/2025.
Se encuentran seleccionados los route_ids 62, 57, 38, 52, 54, 26, 41, 103 y 63.

Las líneas 62 (verde agua), 57 (gris) y 38 (naranja) conectan el centro de Brooklyn con diferentes zonas de Queens, utilizando trayectorias que atraviesan sectores distintos del distrito, sin mostrar superposición significativa. En cambio, las rutas 52 (verde), 54 (rosa) y 26 (lila) se dirigen hacia áreas similares en el límite entre Brooklyn y Queens, aunque lo hacen por corredores internos bien diferenciados. En este grupo, podría considerarse que existe una leve duplicación entre las líneas 52 y 26, ya que ambas recorren zonas paralelas con una separación aproximada de cinco cuadras durante gran parte del trayecto. Por su parte, las rutas 41 (azul), 103 (verde oscuro) y 63 (amarillo) no solo difieren en su destino, sino que también son las únicas del conjunto que se extienden hacia sectores más

periféricos como Flatlands, Canarsie y Bay Ridge, mostrando trayectos claramente disociados del resto.

Otra selección, ilustrada en la **Figura 7.3-3**, muestra las rutas 25, 61, 62 y 67. Aunque todas intersectan alguna de las celdas consideradas *hotspots*, sus trayectos se despliegan por sectores disjuntos del distrito, sin evidenciar coincidencias significativas en sus recorridos principales.

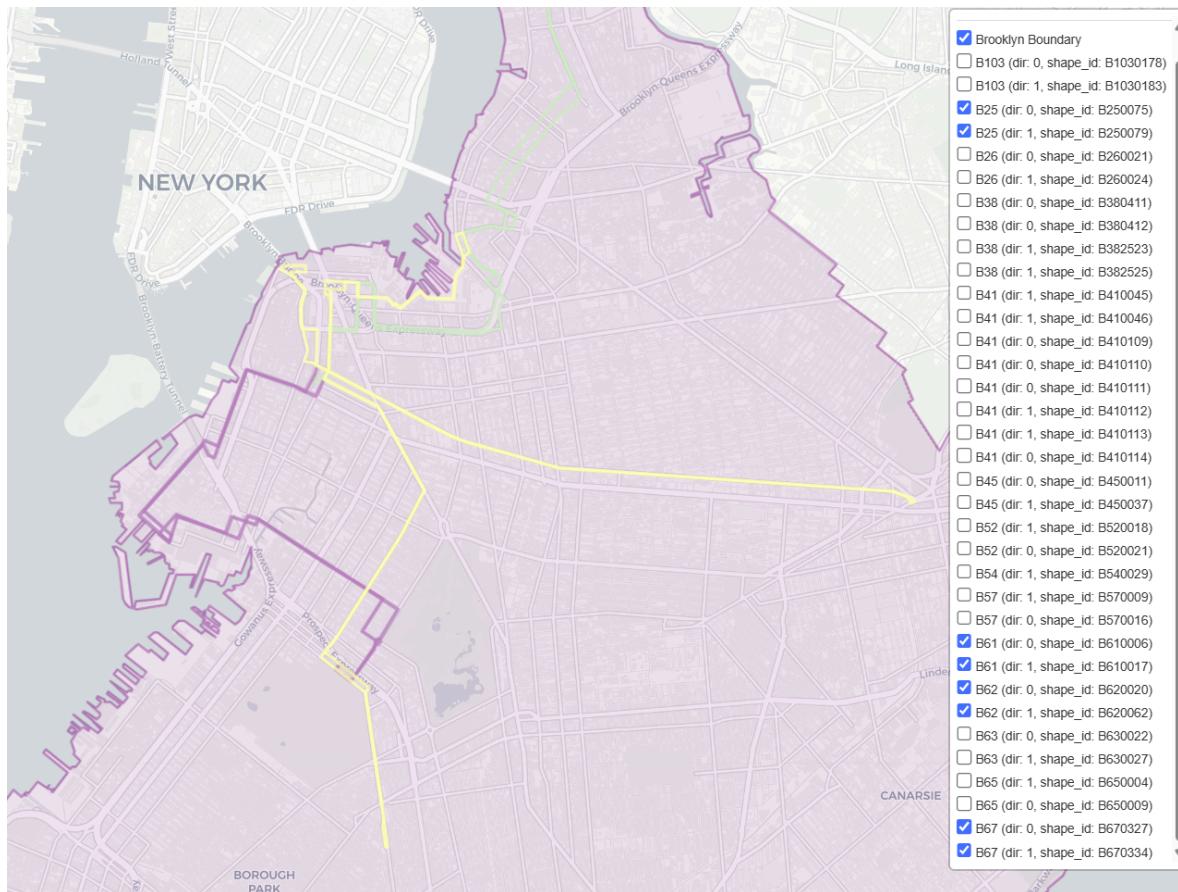


Figura 7.3-3: Rutas que atraviesan las celdas hotspot (IDs 186 y 217) entre las 9 y 10 h del lunes 21/07/2025.
Se encuentran seleccionados los route_ids 25, 61, 62 y 67.

Finalmente, en la **Figura 7.3-4**, se presentan las rutas 45 y 65. A diferencia de los casos anteriores, estas dos líneas exhiben recorridos muy similares a lo largo de varios tramos, lo que sugiere un posible caso de duplicación. Ambas atraviesan zonas centrales

del distrito siguiendo trayectorias casi paralelas, lo que podría implicar una oferta redundante de servicio.

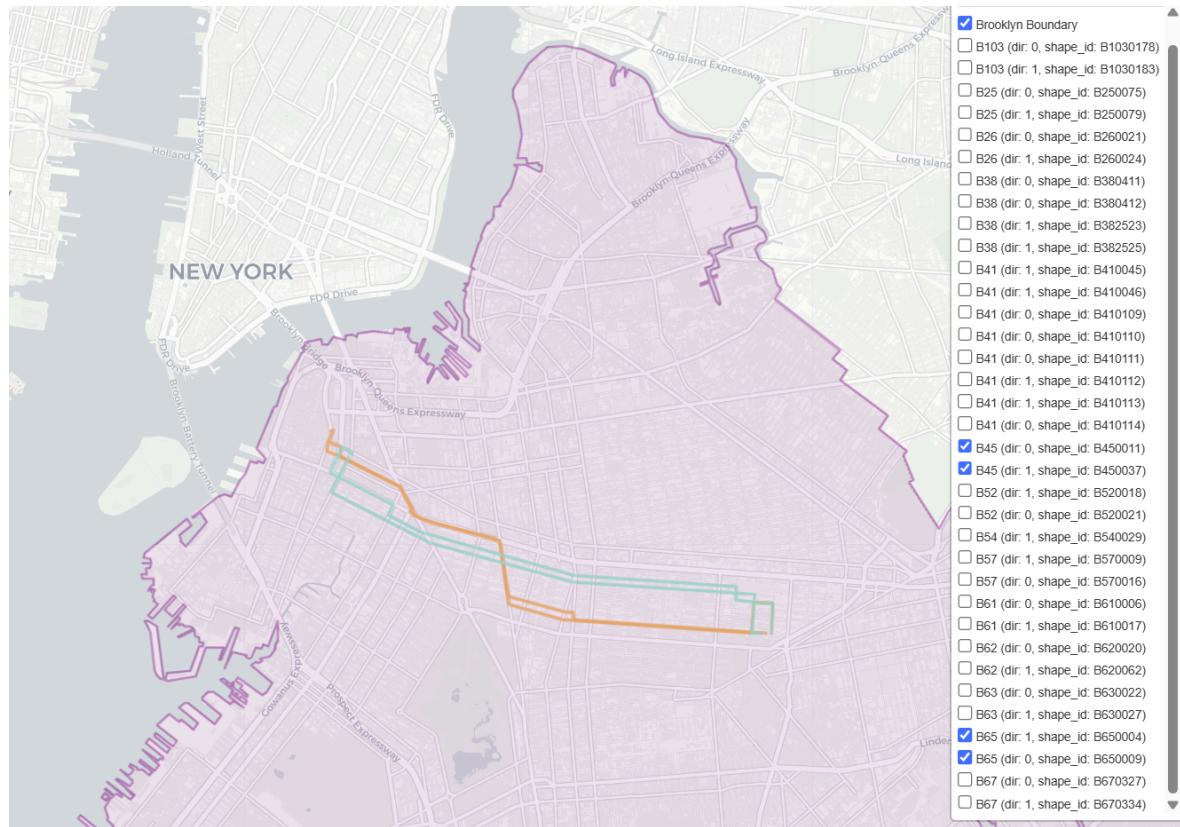


Figura 7.3-4: Rutas que atraviesan las celdas hotspot (IDs 186 y 217) entre las 9 y 10 h del lunes 21/07/2025.
Se encuentran seleccionados los route_ids 45, 65.

Sin embargo, para confirmar que se trata efectivamente de una duplicación operativa, sería necesario contar con información adicional. Factores como la demanda real de pasajeros en esos tramos o el rol que cumple cada ruta dentro del sistema pueden justificar que existan recorridos similares. Por ejemplo, si en esa zona hay una gran cantidad de personas que necesitan moverse en el mismo horario, puede ser razonable que varias líneas coincidan parcialmente. También podría analizarse si es más eficiente operar dos rutas parcialmente solapadas con menor frecuencia de viajes, en lugar de concentrar todos los viajes en una sola ruta sobrecargada. Sin datos sobre cómo se utiliza el sistema

en la práctica, no se puede afirmar que la superposición sea un problema que deba corregirse.

Para concluir, el análisis precedente permitió identificar grupos de rutas con tramos coincidentes en zonas de alta circulación, lo que podría ser indicio de cierta redundancia operativa. Si bien no se puede concluir que estas superposiciones representen ineficiencias sin una evaluación más completa, los resultados obtenidos podrían servir como base para diseñar estudios más detallados que evalúen estos aspectos en mayor profundidad.

8. Velocidades

El análisis de la velocidad es clave para entender el rendimiento y la eficiencia del sistema de transporte público. Esta sección se enfoca en el cálculo, la visualización y la comparación de las velocidades promedio planificadas para los segmentos de rutas, con el objetivo de identificar cómo se tienen en cuenta patrones temporales (tales como las horas pico), diferencias entre días laborables y no laborables, y contrastes entre distritos.

Para realizar este análisis, es necesario contar previamente con los segmentos generados a partir de los archivos GTFS de Brooklyn, Queens y la empresa MTA Bus Company (MTA BC).

El estudio se basa en los días 20 y 21 de julio de 2025, correspondientes a un domingo y un lunes, respectivamente. Esta selección permite contrastar el comportamiento del sistema entre un día laboral típico y un día de fin de semana.

Esta elección de distritos y fechas introduce una complejidad adicional en comparación con la sección anterior: ahora se trabaja con datos provenientes de dos regiones, tres archivos GTFS distintos y dos días completos. En contraste, el análisis anterior se centraba únicamente en una franja horaria de un solo día, de un único distrito. Esta mayor escala de datos obliga a prestar especial atención a la eficiencia de las consultas.

A continuación, se presenta el código utilizado para cargar los datos necesarios a partir de la base ya procesada, siguiendo un procedimiento similar al realizado en la sección 6:

```
import gtfs_functions as gtfs
import os

from db_utils.db_connection import save_to_postgis

def load_gtfs_feed(file_path, start_date, end_date):
```

```
feed = gtfs.Feed(file_path, start_date=start_date,
end_date=end_date)
    return feed.segments

if __name__ == "__main__":
    base_dir = os.path.dirname(__file__)
    start_date = '2025-07-20'
    end_date = '2025-07-21'

    gtfs_files = [
        "gtfs_b.zip",
        "gtfs_busco.zip",
        "gtfs_q.zip",
    ]

    schema = [
        "brooklyn",
        "busco",
        "queens",
    ]

    for i, filename in enumerate(gtfs_files):
        gtfs_path = os.path.join(base_dir,
"../GTFS_SCHEDULED_ROOT_FOLDER/gtfs", filename)
        segments_gdf = gtfs.Feed(gtfs_path,
start_date=start_date, end_date=end_date).segments
        print(f"Saving {len(segments_gdf)} rows from {filename}")
        save_to_postgis(
            segments_gdf,
            table_name="segments",
            schema=schema[i],
            if_exists='replace'
        )
```

El código utilizado en esta sección se encuentran dentro de [esta carpeta](#) del repositorio de GitHub del trabajo y las visualizaciones dentro de [esta carpeta](#) de Drive.

8.1 Velocidades por segmento en Brooklyn y Queens

Con el fin de establecer un panorama general inicial, se propone visualizar primero los segmentos con su velocidad promedio.

Primero, se calcula en vistas materializadas los segmentos en EPSG:32118. Se mantiene su versión en 4326. Al mismo tiempo, se conserva una versión en EPSG:4326, que se utilizará para la visualización en mapas interactivos con Folium.

A continuación, se muestra únicamente la vista correspondiente al distrito de queens. Se generan vistas análogas en los esquemas busco y brooklyn, cada una con sus respectivos índices para optimizar el rendimiento de las consultas.

```
CREATE MATERIALIZED VIEW queens.segments_32218 AS
SELECT
    s.*,
    st_length(geometry) as length,
    ST_Transform(ST_SetSRID(geometry, 4326), 32118) AS
geometry_32118
FROM queens.segments s;

CREATE INDEX idx_segments_32118_geom
ON queens.segments_32218
USING GIST (geometry_32118);
```

Se utiliza una vista materializada que consolida la información proveniente de los tres archivos GTFS, incluyendo las velocidades ya calculadas para cada connection planificada asociada a un segmento. Cada connection representa el recorrido previsto que un colectivo debería realizar entre los extremos de un segmento. A partir del horario programado, es posible estimar el tiempo de viaje y, con ello, calcular la velocidad planificada para ese trayecto.

```
CREATE MATERIALIZED VIEW segments_with_speed_32118 AS
WITH borough_geom AS (
    SELECT ST_Union(geom_32118) AS geom
    FROM public.ny_adm
    WHERE boroname IN ('Brooklyn', 'Queens')
),
all_connections AS (
    SELECT 'queens' AS source, s.shape_id, s.geometry,
    s.geometry_32118, st_length(s.geometry_32118) as length,
    c.trip_id, c.from_stop_id, c.to_stop_id,
    c.t_departure, c.t_arrival, date, from_stop_name,
    to_stop_name
    FROM queens.segments_32118 s
    JOIN queens.connections c
        ON s.route_id = c.route_id AND s.direction_id =
    c.direction_id
        AND s.start_stop_id = c.from_stop_id AND s.end_stop_id =
    c.to_stop_id
    UNION ALL
```

```
SELECT 'busco', s.shape_id, s.geometry, s.geometry_32118,
st_length(s.geometry_32118) as length, c.trip_id, c.from_stop_id,
c.to_stop_id,
      c.t_departure, c.t_arrival, date, from_stop_name,
to_stop_name
  FROM busco.segments_32118 s
  JOIN busco.connections c
    ON s.route_id = c.route_id AND s.direction_id =
c.direction_id
   AND s.start_stop_id = c.from_stop_id AND s.end_stop_id =
c.to_stop_id

UNION ALL

SELECT 'brooklyn', s.shape_id, s.geometry, s.geometry_32118,
st_length(s.geometry_32118) as length, c.trip_id, c.from_stop_id,
c.to_stop_id,
      c.t_departure, c.t_arrival, date, from_stop_name,
to_stop_name
  FROM brooklyn.segments_32118 s
  JOIN brooklyn.connections c
    ON s.route_id = c.route_id AND s.direction_id =
c.direction_id
   AND s.start_stop_id = c.from_stop_id AND s.end_stop_id =
c.to_stop_id
)
SELECT
  ac.source,
  ac.length,
  ac.geometry_32118,
  ac.from_stop_id,
  ac.to_stop_id,
  ac.shape_id,
  ac.geometry,
  ac.t_arrival,
  ac.t_departure,
  ac.from_stop_name,
  ac.to_stop_name,
  (ac.length /
    EXTRACT(EPOCH FROM (ac.t_arrival - ac.t_departure)))*3.6 AS
speed
FROM all_connections ac, borough_geom bg
WHERE ac.t_arrival > ac.t_departure
  AND ac.date >= '2025-07-20' AND ac.date < '2025-07-22'
  AND ST_Intersects(ac.geometry_32118, bg.geom)
```

Antes de visualizar la información, verificamos cual es la velocidad máxima para asignar la escala

```
with avg as (
```

```
SELECT
    geometry,
    from_stop_name,
    to_stop_name,
    AVG(speed) AS avg_speed
FROM segments_with_speed_32118
GROUP BY from_stop_id, to_stop_id, geometry, from_stop_name,
to_stop_name)
select max(avg_speed) from avg;
```

La velocidad promedio máxima se aproxima a los 110 km/h. Elevado para un bus en una ciudad.

El siguiente fragmento de código es parte del utilizado para graficar las velocidades, el código completo se encuentra en el repositorio de github.

```
def load_geom_boundary(engine):
    query = """
        SELECT geom as geometry
        FROM public.ny_adm
        WHERE boroname = 'Brooklyn' OR boroname = 'Queens';
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def load_segments_from_postgis(engine):
    query = """
        SELECT
            geometry,
            from_stop_name,
            to_stop_name,
            AVG(speed) AS avg_speed
        FROM segments_with_speed_32118
        GROUP BY from_stop_id, to_stop_id, geometry, from_stop_name,
        to_stop_name;
    """
    return gpd.read_postgis(query, engine, geom_col='geometry')

def visualize(segment_data, brooklyn_boundary=None, cutoff=55):
    transport_map = f.Map(location=[40.65, -73.95],
    tiles='CartoDB positron', zoom_start=11)

    colormap = cm.LinearColormap(
        colors=['white', 'yellow', 'orange', 'red', 'darkred'],
        vmin=0,
        vmax=cutoff,
        caption='Speed'
    )
    colormap.add_to(transport_map)
```

```
# --- Agregar geometría de Brooklyn al fondo ---
if brooklyn_boundary is not None:
    for _, row in brooklyn_boundary.iterrows():
        geojson = f.GeoJson(
            data={
                "type": "Feature",
                "geometry": row['geometry'].__geo_interface__,
                "properties": {},
            },
            style_function=lambda x: {
                'fillColor': 'purple',
                'color': 'purple',
                'weight': 2,
                'fillOpacity': 0.1,
                'opacity': 0.4
            }
        )
        geojson.add_to(transport_map)

# --- Agregar segmentos ---
for _, segment in segment_data.iterrows():
    trip_count = min(segment['avg_speed'], cutoff)
    color = colormap(trip_count)
    geo_json = f.GeoJson(
        data={
            "type": "Feature",
            "geometry": segment['geometry'].__geo_interface__,
            "properties": {
                "from_stop_name": segment['from_stop_name'],
                "to_stop_name": segment['to_stop_name'],
                "avg_speed": segment['avg_speed']
            }
        },
        style_function=lambda x, color=color: {
            'color': color, 'weight': 3, 'opacity': 0.7
        },
        tooltip=GeoJsonTooltip(
            fields=['from_stop_name', 'to_stop_name',
'avg_speed'],
            aliases=['From Stop', 'To Stop', 'Speed'],
            localize=True
        )
    )
    geo_json.add_to(transport_map)

return transport_map
```

En el fondo se muestran las geometrías de los distritos de Brooklyn y Queens, también es posible ver las velocidades junto con un color que indica que tan alta es la magnitud de la velocidad.

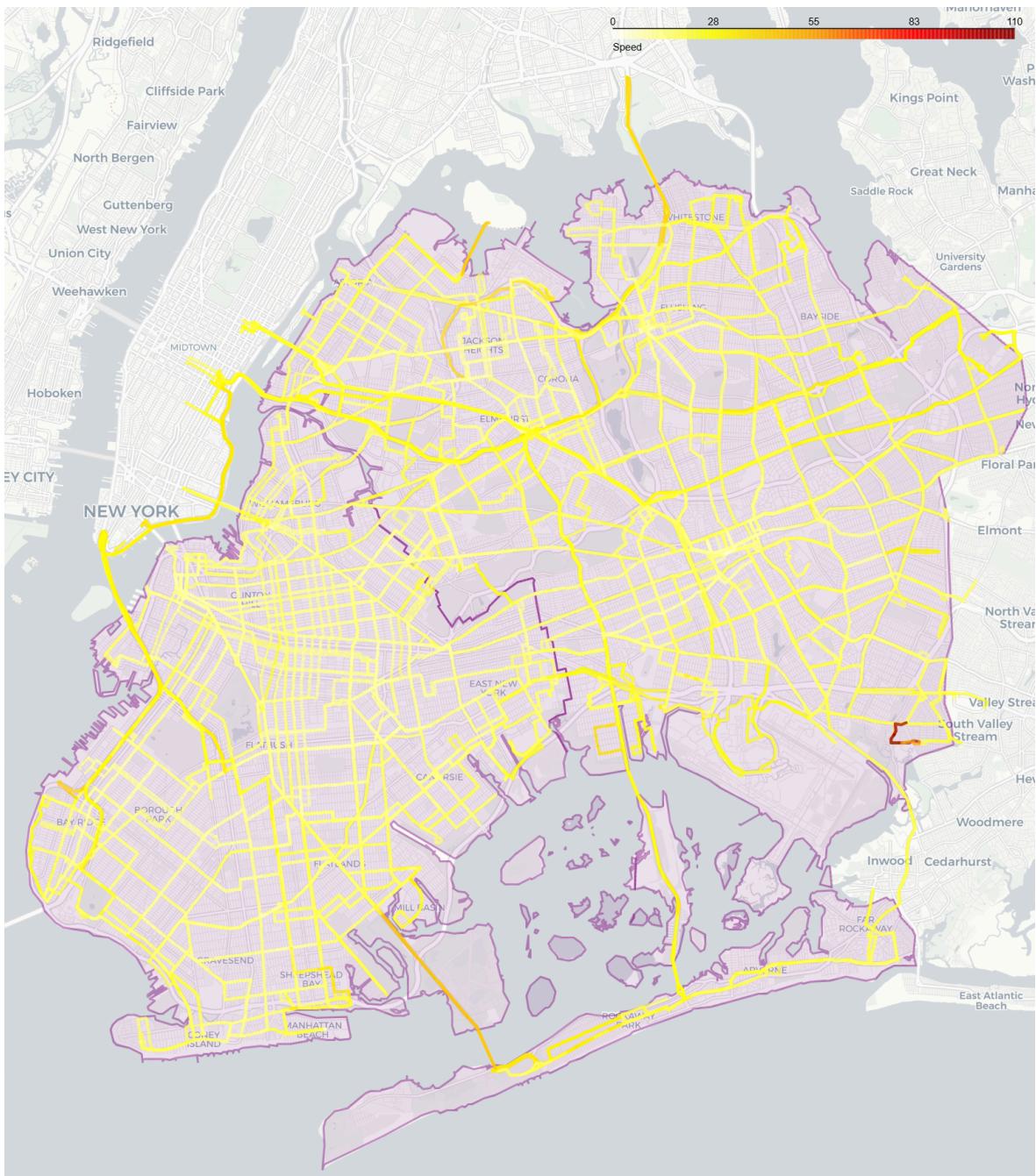


Figura 8.1-1: Mapa de segmentos en Brooklyn y Queens según su velocidad (20 al 21 de julio de 2025).
Datos de MTA NYCT (Queens y Brooklyn) y MTA BC

Es posible notar que hay un sector muy pequeño en el que las velocidades son mayores a 100 km/h. Si investigamos un poco que ocurre en estos casos con esta query

```
SELECT t_arrival, t_departure, length FROM
segments_with_speed_32118 where speed > 100;
```

Cuyo resultado contiene entradas similares entre sí:

	t_arrival	t_departure	st_length
1	2025-07-20 17:00:21.000000 +00:00	2025-07-20 17:00:00.000000 +00:00	604.3972552886155
2	2025-07-20 17:01:00.000000 +00:00	2025-07-20 17:00:37.000000 +00:00	668.0242911374605
3	2025-07-20 23:30:37.000000 +00:00	2025-07-20 23:30:21.000000 +00:00	491.2409238091945
4	2025-07-21 08:48:21.000000 +00:00	2025-07-21 08:48:00.000000 +00:00	604.3972552886155
5	2025-07-21 08:49:00.000000 +00:00	2025-07-21 08:48:37.000000 +00:00	668.0242911374605
6	2025-07-21 10:33:21.000000 +00:00	2025-07-21 10:33:00.000000 +00:00	604.3972552886155
7	2025-07-21 10:34:00.000000 +00:00	2025-07-21 10:33:37.000000 +00:00	668.0242911374605
8	2025-07-21 16:58:37.000000 +00:00	2025-07-21 16:58:21.000000 +00:00	491.2409238091945
9	2025-07-22 01:05:21.000000 +00:00	2025-07-22 01:05:00.000000 +00:00	604.3972552886155
10	2025-07-22 01:06:00.000000 +00:00	2025-07-22 01:05:37.000000 +00:00	668.0242911374605
11	2025-07-21 03:30:37.000000 +00:00	2025-07-21 03:30:21.000000 +00:00	491.2409238091945
12	2025-07-21 08:00:37.000000 +00:00	2025-07-21 08:00:21.000000 +00:00	491.2409238091945
13	2025-07-22 00:05:21.000000 +00:00	2025-07-22 00:05:00.000000 +00:00	604.3972552886155
14	2025-07-22 00:06:00.000000 +00:00	2025-07-22 00:05:37.000000 +00:00	668.0242911374605
15	2025-07-21 11:33:21.000000 +00:00	2025-07-21 11:33:00.000000 +00:00	604.3972552886155

Lo que se observa es que en la mayoría de los casos el tiempo asignado parece ser muy corto, pero la distancia entre paradas es razonable. Aunque habría que hacer un análisis en mayor profundidad para concluir que hay un error en los tiempos asignados para este recorrido, la velocidad mayor a 100 km/h no parece adecuada para la zona en la que ocurre. Estos valores serán considerados outliers y serán ignorados para evitar distorsiones en el análisis.

Es posible también ver que no hay ningún segmento con velocidad en el rango [80,100), por lo que considerar velocidades máximas de 80 km/h para los buses parece razonable. Esta modificación se incluirá directamente en la vista materializada inicial para no olvidar tenerla en cuenta a lo largo de las diferentes queries realizadas en esta sección.

Esta query

```
SELECT t_arrival, t_departure, length FROM
segments_with_speed_32118 where speed < 100 and speed >= 80;
```

Devuelve un conjunto vacío.

Luego de eliminar esos valores considerados outliers, la velocidad máxima promedio es aproximadamente 55km/h. Ajustando ese valor máximo en la visualización y el resultado es el siguiente:



Figura 8.1-2: Mapa de segmentos en Brooklyn y Queens según su velocidad (20 al 21 de julio de 2025).
Datos de MTA NYCT (Queens y Brooklyn) y MTA BC luego de eliminar posibles outliers

La Figura muestra el comportamiento de las velocidades promedio de los servicios de bus en Brooklyn y Queens. Se observa que las velocidades más bajas (0–20 km/h), representadas por tonalidades amarillas, predominan en zonas centrales y densamente

urbanizadas, como Downtown Brooklyn o Bushwick, donde la superposición de rutas y la frecuencia de paradas limitan la fluidez del tránsito.

En contraste, los segmentos más rápidos (mayores a 40 km/h, en rojo oscuro) se concentran en zonas periféricas y corredores rápidos, como los que cruzan hacia Rikers (Fig 8.1-3), o hacia Manhattan a través de Whitestone Expressway (Fig 8.1-4)



Figura 8.1-3: Corredor de alta velocidad en dirección a Rikers Island, ubicado en el norte de Queens. El tramo, representado en tonos rojo oscuro, evidencia velocidades promedio superiores a 40 km/h

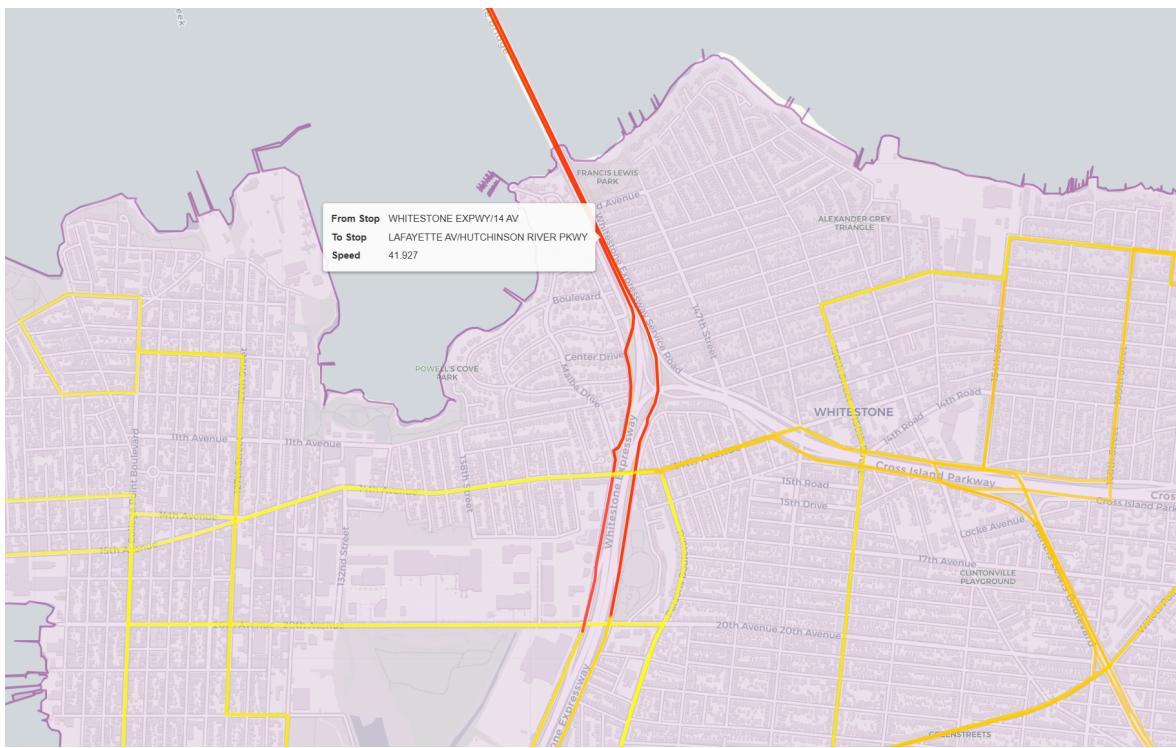


Figura 8.1-4: Tramo correspondiente a la Whitestone Expressway, una autopista que conecta el norte de Queens con Manhattan.

Estos tramos de mayor velocidad funcionan como ejes de conexión entre los núcleos urbanos más lentos y otras zonas de la ciudad. Actúan como corredores principales, permitiendo que los buses se desplacen más rápido por fuera de las áreas más congestionadas.

8.2 Comparación de velocidades

Con el objetivo de entender mejor cómo varía la velocidad del transporte público a lo largo del día y según el tipo de jornada, en esta sección comparamos las velocidades promedio por hora durante un domingo y un lunes (20 y 21 de julio de 2025). Observaremos cómo influye el día de la semana en la velocidad de circulación de los buses y si aparecen diferencias marcadas en las horas pico.

El análisis se enfoca en comparar, hora por hora, qué tan rápido se mueven los buses en un día de fin de semana (con menos tráfico y menos demanda) frente a un día laboral típico, donde se espera más congestión y frecuencias más ajustadas. Esto permite detectar patrones interesantes como:

- ¿A qué hora del día los buses circulan más rápido?
- ¿El domingo mantiene velocidades más constantes?
- ¿El lunes sufre caídas fuertes en ciertos horarios clave (como 8 a 9 AM o 17 a 19 PM)?

Más adelante, se analizará si hay diferencias de velocidades entre los distritos (Brooklyn vs. Queens).

8.2.1 Comparación de promedio por hora y por dia

En esta sección se analiza cómo varía la velocidad promedio de los buses en cada hora del día, comparando domingo y lunes por separado. Esto permite observar con más claridad las horas pico, los momentos de mayor fluidez y cómo varía la velocidad dentro del mismo día para ambos días analizados.

La siguiente consulta agrupa todos los segmentos de viaje registrados entre el domingo 20 de julio y el lunes 21 de julio de 2025, y calcula el promedio de velocidad por cada hora del día, separando los resultados por jornada:

```
SELECT
    CASE EXTRACT(DOW FROM t_arrival AT TIME ZONE
'America/New_York')
        WHEN 0 THEN 'Sunday'
        WHEN 1 THEN 'Monday'
        ELSE 'unknown'
    END AS weekday,
    EXTRACT(HOUR FROM t_arrival AT TIME ZONE
'America/New_York') AS hour,
    COUNT(*) AS num_segments,
    AVG(speed) AS avg_speed
FROM segments_with_speed_32118
```

```
    WHERE t_arrival >= '2025-07-20 00:00:00 America/New_York'  
    AND t_arrival < '2025-07-22 00:00:00 America/New_York'  
        GROUP BY weekday, hour  
        ORDER BY weekday, hour;
```

Es importante tener en cuenta que `t_arrival` esta almacenado en UTC-0, debemos realizar los ajustes necesarios para poder trabajar con horarios de Nueva York. Queremos analizar qué ocurrió en la ciudad en su hora pico.

Los gráficos se generan automáticamente para cada día con el siguiente script en Python:

```
def plot_speed_histograms():  
    engine = get_engine()  
  
    query = f"""  
        SELECT  
            CASE EXTRACT(DOW FROM t_arrival AT TIME ZONE  
'America/New_York')  
                WHEN 0 THEN 'Sunday'  
                WHEN 1 THEN 'Monday'  
                ELSE 'unknown'  
            END AS weekday,  
            EXTRACT(HOUR FROM t_arrival AT TIME ZONE  
'America/New_York') AS hour,  
            COUNT(*) AS num_segments,  
            AVG(speed) AS avg_speed  
            FROM segments_with_speed_32118  
            WHERE t_arrival >= '2025-07-20 00:00:00 America/New_York'  
            AND t_arrival < '2025-07-22 00:00:00 America/New_York'  
            GROUP BY weekday, hour  
            ORDER BY weekday, hour;  
        """  
  
    df = pd.read_sql(query, engine)  
  
    sns.set(style="whitegrid")  
  
    full_hours = pd.Series(range(24), name="hour")  
  
    for day in df['weekday'].unique():  
        day_df = df[df['weekday'] == day].copy()  
        day_df['hour'] = day_df['hour'].astype(int)  
  
        # Reindex to ensure all hours 0-23 are included  
        day_df = full_hours.to_frame().merge(day_df, on="hour",  
how="left")  
        day_df['weekday'] = day
```

```
day_df['num_segments'] =  
day_df['num_segments'].fillna(0).astype(int)  
day_df['avg_speed'] = day_df['avg_speed'].fillna(0)  
  
total_segments = day_df['num_segments'].sum()  
  
plt.figure(figsize=(10, 6))  
sns.barplot(x='hour', y='avg_speed', data=day_df,  
color='blue')  
plt.title(f"{day} - Average Speed by  
Hour\n{int(total_segments)} segments considered")  
plt.xlabel("Hour of Day")  
plt.ylabel("Average Speed (km/h)")  
plt.xticks(range(0, 24))  
plt.tight_layout()  
  
plt.savefig(f"outputs/histograms/speed_histogram_{day.lower()}.png")  
plt.close()  
print(f"Saved: speed_histogram_{day.lower()}.png")
```

El script en Python se encarga de automatizar la generación de gráficos que muestran la velocidad promedio de los buses por hora, diferenciando cada jornada. Para eso, primero ejecuta una consulta SQL que calcula la velocidad promedio (`avg_speed`) y la cantidad de segmentos (`num_segments`) por hora, separando los valores para el domingo y el lunes. Luego, carga esos resultados en un DataFrame y genera un gráfico de barras por cada día. Cada gráfico muestra cómo varía la velocidad a lo largo del día, incluye la cantidad total de segmentos considerados, y se guarda automáticamente como imagen.

Los resultados son los siguientes:

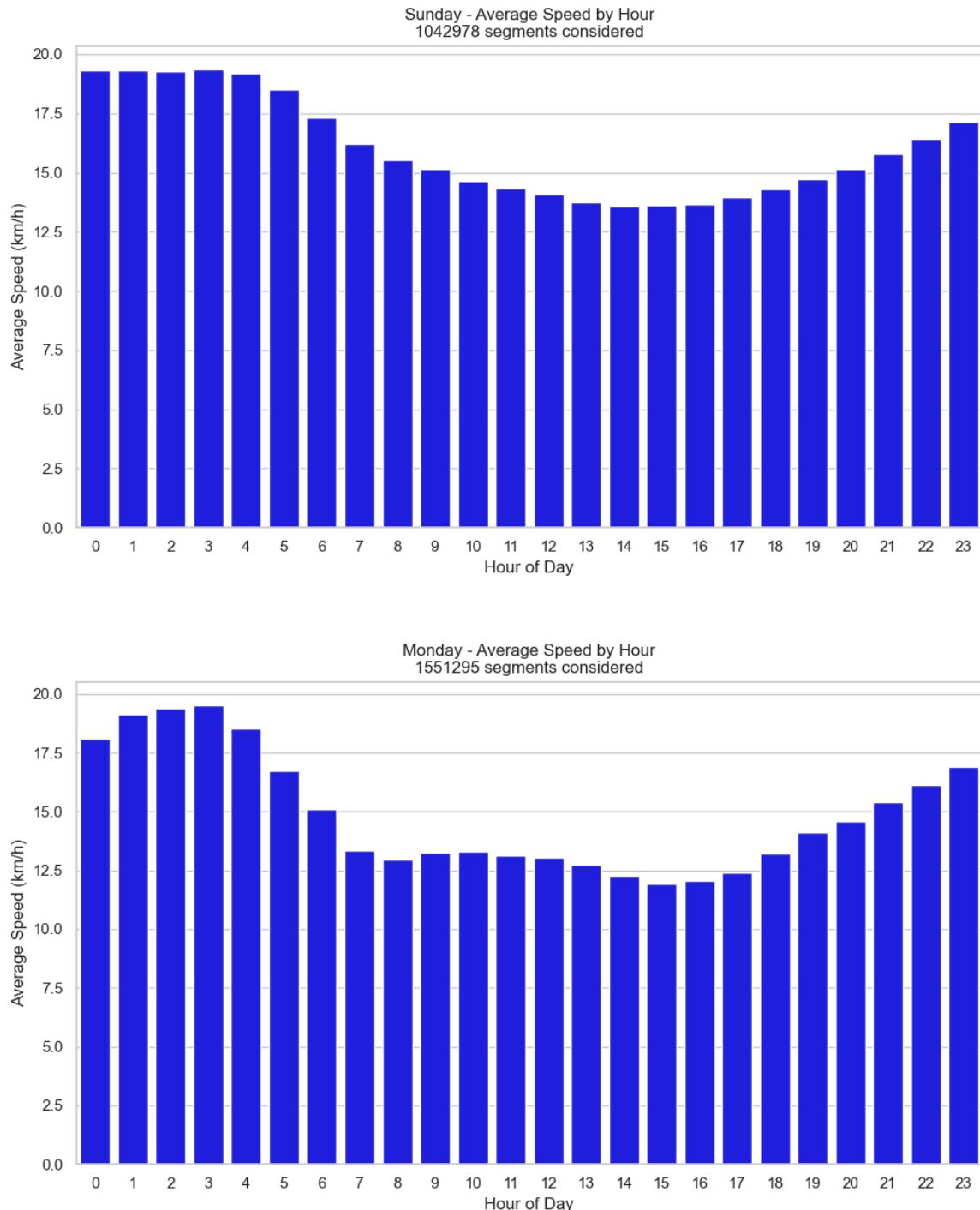


Figura 8.2.1-1: Velocidad promedio por hora del día, comparando domingo (arriba) y lunes (abajo). Cada barra representa la velocidad media de todos los segmentos recorridos en esa franja horaria

En el gráfico del domingo se observa que las velocidades se mantienen altas durante la madrugada (cerca de 19.5 km/h entre las 0 y las 4 AM), bajan progresivamente a partir de las 6 AM y se estabilizan en torno a los 13.5–14 km/h entre las 12 y 17 hs. A partir de las 18 hs, la velocidad vuelve a subir de forma constante hasta alcanzar valores cercanos a los de la madrugada.

Los buses se mueven a una velocidad relativamente alta y estable durante casi todo el día. Se puede distinguir una caída cerca del mediodía, pero no es tan pronunciada, lo que tiene sentido para un día con menor tráfico y menos volumen de pasajeros.

El lunes exhibe un patrón diferente: si bien también comienza con velocidades altas en la madrugada (hasta 19.5 km/h), a partir de las 6 AM se observa una caída abrupta. El peor momento se da entre las 8 y las 9 AM, donde la velocidad baja a 13 km/h o menos, señal clara del impacto de la hora pico de la mañana.

Durante el mediodía no mejora demasiado: la velocidad se mantiene en un rango bajo hasta las 17 hs, donde aparece la segunda gran congestión: la hora pico de la tarde, con valores apenas por encima de 12 km/h. Recién después de las 18 hs se nota una recuperación, que continúa hasta la medianoche.

El efecto del tráfico laboral es muy evidente. Las dos horas pico (mañana y tarde) están claramente marcadas con caídas importantes en la velocidad. Además, el rango entre las 10 y las 16 hs tampoco logra recuperar un ritmo fluido, lo que sugiere una congestión más prolongada durante el día.

Con esta información, podemos responder las preguntas que nos hicimos inicialmente, podemos ver que:

- ¿A qué hora los buses circulan más rápido? Durante la madrugada (0–4 AM) y la noche (22–23 hs) en ambos días. Pero el domingo mantiene estos valores por más tiempo.

- ¿El domingo tiene velocidades más constantes? Sí. El gráfico muestra una caída suave y una recuperación ordenada, sin grandes sobresaltos.
- ¿El lunes sufre más en ciertos horarios? Sin dudas. Las caídas fuertes en la mañana (8–9 hs) y la tarde (17–18 hs) coinciden con las horas pico esperadas. Además, la recuperación es mucho más lenta que la caída.

8.2.2 Comparación de velocidades entre distritos

En esta sección se compara la velocidad proyectada promedio de los segmentos en los distritos Brooklyn y Queens, para evaluar si existen diferencias. Para ello, se utilizó la información segmentada del día lunes 21 de julio de 2025, una jornada laboral típica.

La siguiente consulta, intersecta los segmentos con las geometrías administrativas de los distritos para identificar a qué borough pertenece cada trayecto.

```
WITH boroughs AS (
    SELECT boroname, geom_4326 AS geom
    FROM public.ny_adm
    WHERE boroname IN ('Brooklyn', 'Queens')
), segments_with_borough AS (
    SELECT
        s.*,
        b.boroname AS borough
    FROM segments_with_speed_32118 s
    JOIN boroughs b
        ON ST_Intersects(s.geometry, b.geom)
    WHERE s.t_arrival >= '2025-07-21 00:00:00 America/New_York'
        AND s.t_arrival < '2025-07-22 00:00:00 America/New_York'
)
SELECT
    borough,
    COUNT(*) AS num_segments,
    AVG(speed) AS avg_speed
FROM segments_with_borough
GROUP BY borough
ORDER BY borough;
```

Los resultados obtenidos fueron los siguientes:

- Brooklyn: 858,771 segmentos — Velocidad promedio: 12.71 km/h
- Queens: 697,566 segmentos — Velocidad promedio: 15.26 km/h

Estos datos muestran que, en promedio, se proyecta que en Queens los trayectos son más rápidos que en Brooklyn.

Esta diferencia podría explicarse por diversos factores. Queens tiene más tramos que circulan por corredores rápidos, autopistas urbanas o avenidas con menor densidad peatonal. En cambio, en Brooklyn, muchos servicios atraviesan zonas más densas. Llegar a una conclusión precisa sobre las causas de este fenómeno profundizar en el análisis. La diferencia de más de 2.5 km/h en promedio es significativa considerando que se trata de miles de segmentos.

Luego de ver estos valores, que parecen bajos en comparación a las velocidades promedio observadas, surge curiosidad acerca de a qué se deben estos números. No se profundizará en este aspecto para no hacer excesivamente extenso el análisis, pero sería interesante indagar al respecto.