



---

## Análisis GTFS Vancouver

Trabajo Final

73.82 - Bases de Datos Espaciales y de Movilidad

**Autores:**

**Tomas Camilo Gay Bare<sup>1</sup>**

**Manuel E. Dithurbide<sup>2</sup>**

Diciembre 2025

---

<sup>1</sup>tgaybare@itba.edu.ar

<sup>2</sup>mdithurbide@itba.edu.ar

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Metodología de replicación</b>	<b>3</b>
2.1. Instalación de dependencias . . . . .	3
2.2. Requisitos de la base de datos . . . . .	3
2.3. Flujo GTFS estático . . . . .	4
2.3.1. Descarga y preprocesamiento . . . . .	4
2.3.2. Importación de tablas GTFS . . . . .	4
2.3.3. Creación de estructuras MobilityDB . . . . .	4
2.3.4. Carga de densidad poblacional (opcional) . . . . .	4
2.3.5. Consultas y visualizaciones estáticas . . . . .	5
2.4. Flujo GTFS-Realtime . . . . .	5
2.4.1. Configuración y tablas realtime . . . . .	5
2.4.2. Ingesta de feeds GTFS-Realtime . . . . .	5
2.4.3. Construcción de trayectorias reales . . . . .	6
2.4.4. Análisis puntual de un recorrido . . . . .	6
2.4.5. Consultas y análisis agregados con GTFS-Realtime . . . . .	6
<b>3. Análisis de GTFS estático</b>	<b>7</b>
3.1. Distribución de tipos de rutas . . . . .	7
3.2. Densidad de rutas por segmento . . . . .	9
3.3. Duplicación de rutas . . . . .	10
3.4. Análisis de velocidades . . . . .	11
3.5. Accesibilidad a estadios . . . . .	13
<b>4. Análisis con GTFS-Realtime</b>	<b>15</b>
4.1. Velocidad real vs. velocidad programada . . . . .	15
4.2. Desempeño de horarios (schedule times) . . . . .	17
4.3. Segmentos de demora y patrones de congestión . . . . .	18
4.4. Regularidad de headways y <i>bus bunching</i> . . . . .	20



# 1. Introducción

Este proyecto analiza el sistema de transporte de Vancouver utilizando datos GTFS estáticos y GTFS-Realtime de TransLink. Se implementa un pipeline reproducible sobre PostgreSQL con extensiones PostGIS y MobilityDB, utilizando Python para análisis y visualización. El pipeline permite caracterizar la red estática, comparar planificación vs. operación y generar visualizaciones para identificar problemas de servicio.

## 2. Metodología de replicación

En esta sección se documenta paso a paso el pipeline completo, desde la instalación hasta la generación de los resultados presentados en las secciones siguientes.

### 2.1. Instalación de dependencias

Situarse en la raíz del repositorio y ejecutar:

```
pip install -r requirements.txt
pip install -r static_analysis/requirements.txt
pip install -r realtime_analysis/requirements.txt
npm install -g gtfs-via-postgres
```

Configurar la conexión a la base de datos exportando las variables de entorno:

```
export PGHOST=localhost
export PGPORT=5432
export PGUSER=postgres
export PGPASSWORD=postgres
export PGDATABASE=gtfs
```

Para hacer estas variables persistentes, agregarlas al perfil del shell (e.g., `~/.bashrc`, `~/.zshrc`).

### 2.2. Requisitos de la base de datos

Antes de ejecutar cualquier comando, asegurarse de que:

- La base de datos `gtfs` existe y está corriendo.
- Las extensiones PostGIS y MobilityDB están instaladas y habilitadas.
- Las variables de entorno de conexión están exportadas (ver sección anterior).

Se puede verificar la conexión y las extensiones con:

```
psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE \
-c "SELECT_PostGIS_version(), MobilityDB_version();"
```

## 2.3. Flujo GTFS estático

### 2.3.1. Descarga y preprocesamiento

Ejecutar el script de descarga y limpieza:

```
bash static_analysis/data/download_data.sh
```

Este paso genera los archivos prunados en static\_analysis/data/gtfs\_pruned/.

### 2.3.2. Importación de tablas GTFS

Con la base de datos en marcha, importar las tablas:

```
cd static_analysis/data/gtfs_pruned
gtfs-to-sql --require-dependencies -- *.txt \
| psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE
```

### 2.3.3. Creación de estructuras MobilityDB

Desde la raíz del repositorio:

```
cat static_analysis/data_loading/mobilitydb_import.sql \
| psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE
```

Este script genera tablas como scheduled\_trips\_mdb, route\_segments, vistas agregadas y columnas geométricas.

### 2.3.4. Carga de densidad poblacional (opcional)

Para los análisis que combinan densidad poblacional y cobertura de transporte, se utiliza un GeoJSON de áreas censales de Vancouver. El siguiente script calcula la densidad (pop/a) e importa la capa population\_areas en la base de datos:

```
python static_analysis/data/download_population_data.py \
--geo static_analysis/data/population/vancouver_geo.geojson
```

### 2.3.5. Consultas y visualizaciones estáticas

Las vistas materializadas y las gráficas de análisis se generan con un único comando que orquesta las consultas SQL y los scripts de visualización:

```
cd static_analysis/queries  
python run_all_analyses.py
```

Este comando:

- Ejecuta los archivos SQL en `static_analysis/queries/sql/`, creando vistas materializadas `qgis_*` para visualización en QGIS (densidad de rutas, segmentos de velocidad, población vs. transporte, etc.).
- Ejecuta los scripts de `static_analysis/queries/visualizations/`, generando gráficos en formato PNG (histogramas, distribuciones por ruta, comparaciones de velocidad, métricas de proximidad a estadios, etc.).

Los resultados se guardan en `static_analysis/queries/results/`, organizados por tipo de análisis. Las vistas `qgis_*` pueden cargarse directamente en QGIS para producir mapas más detallados.

## 2.4. Flujo GTFS-Realtime

### 2.4.1. Configuración y tablas realtime

Instalar dependencias adicionales y configurar la API de TransLink:

```
pip install -r realtime_analysis/requirements.txt  
export TRANSLINK_GTFSR_API_KEY="your-translink-api-key"
```

Crear las tablas necesarias para datos realtime:

```
cat realtime_analysis/realtime_schema.sql \  
| psql -h $PGHOST -p $PGPORT -U $PGUSER -d $PGDATABASE
```

### 2.4.2. Ingesta de feeds GTFS-Realtime

Ejecutar el proceso de ingestá durante un intervalo de tiempo:

```
python -m realtime_analysis.ingest_realtime \  
--duration-minutes 20 \  
--poll-interval 30
```

Este comando consulta periódicamente los endpoints de posiciones y actualizaciones de viaje, almacenando los mensajes en tablas `realtime_*`.

### **2.4.3. Construcción de trayectorias reales**

A partir de los puntos GPS se construyen trayectorias *map-matched* sobre los recorridos estáticos:

```
python -m realtime_analysis.build_realtime_trajectories --hours  
2
```

Las trayectorias resultantes se almacenan en `realtime_trips_mdb`. Los puntos GPS crudos se deduplican y se ajustan (*snap*) a las formas programadas antes de insertarse en la tabla.

### **2.4.4. Análisis puntual de un recorrido**

Para comparar detalladamente un viaje programado vs. su ejecución real:

```
python -m realtime_analysis.analyze_realtime \  
--route-short-name 99
```

Los resultados se escriben en `realtime_analysis/output/` como mapas HTML, gráficos de tiempos y archivos CSV con métricas por segmento.

### **2.4.5. Consultas y análisis agregados con GTFS-Realtime**

Las vistas materializadas y los análisis agregados sobre datos realtime se generan de forma análoga al flujo estático, a través de un único script orquestador:

```
cd realtime_analysis/queries  
python run_all_analyses.py
```

Este comando ejecuta los archivos SQL de `realtime_analysis/queries/sql/`, creando vistas `realtime_*` y `qgis_realtime_*`, y luego corre los scripts de visualización para producir gráficos en PNG y resúmenes en CSV.

También es posible ejecutar cada análisis de forma individual:

```
cd realtime_analysis/queries/visualizations  
python speed_vs_schedule_analysis.py  
python schedule_times_analysis.py  
python delay_segments_analysis.py  
python headway_analysis.py
```

Los resultados se almacenan en `realtime_analysis/queries/results/`, organizados por tipo de análisis.

### 3. Análisis de GTFS estático

#### 3.1. Distribución de tipos de rutas

La primera consulta caracteriza la composición de la red en términos de tipo de servicio (bus, subway, rail, ferry, etc.), utilizando las tablas GTFS routes y un CTE de mapeo de códigos:

```
WITH route_types(route_type, name) AS (
    SELECT '0', 'streetcar' UNION
    SELECT '1', 'subway' UNION
    SELECT '2', 'rail' UNION
    SELECT '3', 'bus' UNION
    SELECT '4', 'ferry' UNION
    SELECT '11', 'trolley'
),
route_groups AS (
    SELECT
        route_type,
        COUNT(*) AS qty,
        ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (), 2) AS perc
    FROM routes
    GROUP BY route_type
)
SELECT name, qty, perc
FROM route_groups g JOIN route_types t ON g.route_type = t.
    route_type
ORDER BY perc DESC;
```

Esta consulta devuelve, para cada modo, la cantidad de rutas y su porcentaje sobre el total. En la práctica, complementamos esta tabla con un mapa de la red completa y un gráfico de distribución de viajes por ruta, mostrados en las Figuras 1 y 2.

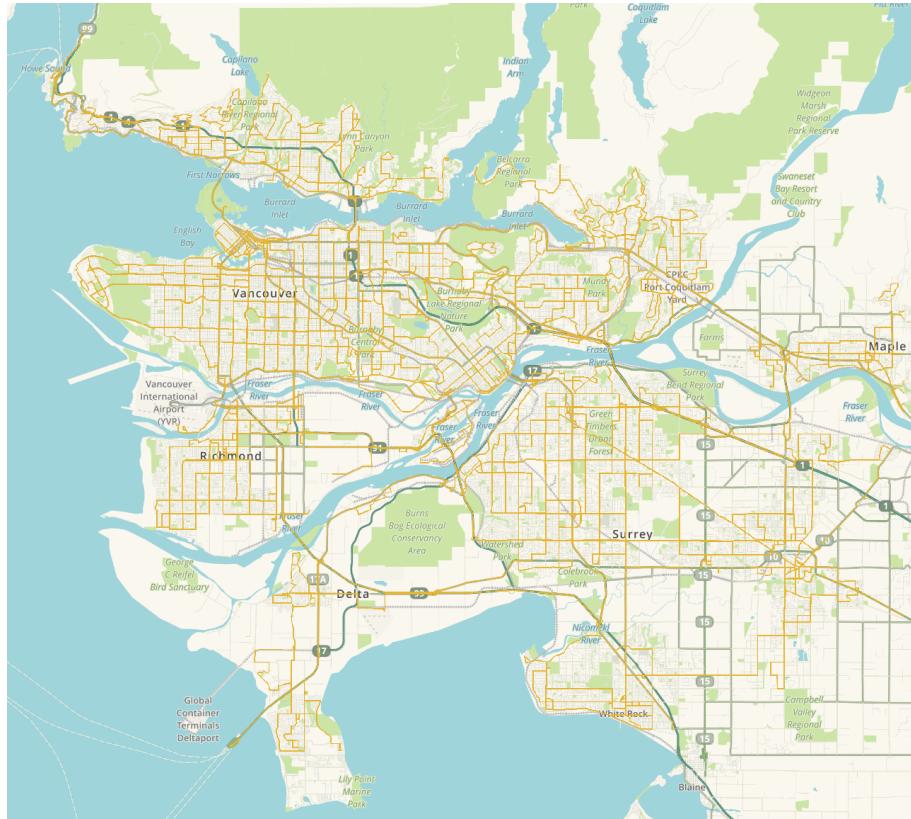


Figura 1: Mapa de la red de rutas de Vancouver a partir de GTFS estático.

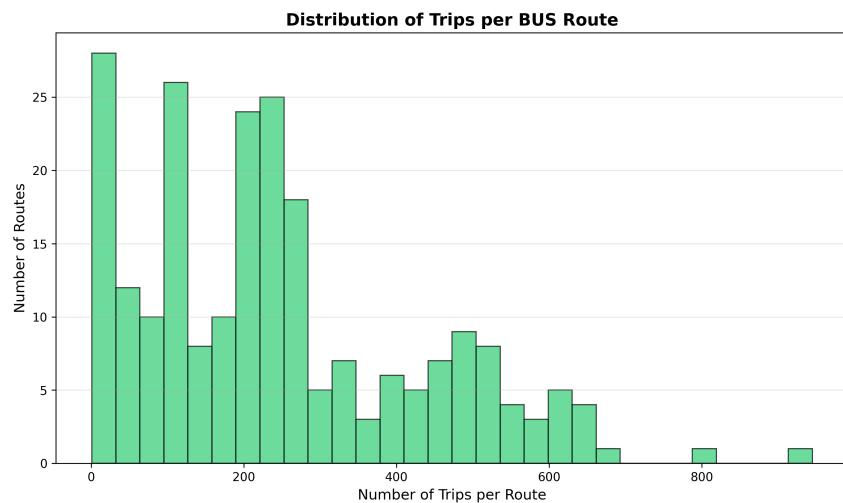


Figura 2: Distribución del número de viajes por ruta.

En el mapa se observa una red especialmente densa en el centro de Vancouver y en los corredores troncales hacia el este y el sur. El histograma muestra que un subconjunto reducido de rutas concentra un gran número de viajes diarios, mientras que la mayoría opera con menor frecuencia, típicamente en zonas periféricas.

### 3.2. Densidad de rutas por segmento

Para estudiar la densidad de servicio sobre el espacio urbano se construye la vista materializada `segment_route_density`, basada en la tabla `route_segments`:

```
DROP MATERIALIZED VIEW IF EXISTS segment_route_density;
CREATE MATERIALIZED VIEW segment_route_density AS
SELECT
    stop1_id || stop2_id AS segment_id,
    seg_geom,
    COUNT(DISTINCT route_id) AS num_routes
FROM route_segments
WHERE seg_geom IS NOT NULL
GROUP BY stop1_id, stop2_id, seg_geom;
```

El script `route_density_analysis.py` resume esta información y genera un histograma de densidad (Figura 3), mientras que las vistas materializadas se utilizan para construir un mapa de densidad de segmentos (Figura 4).

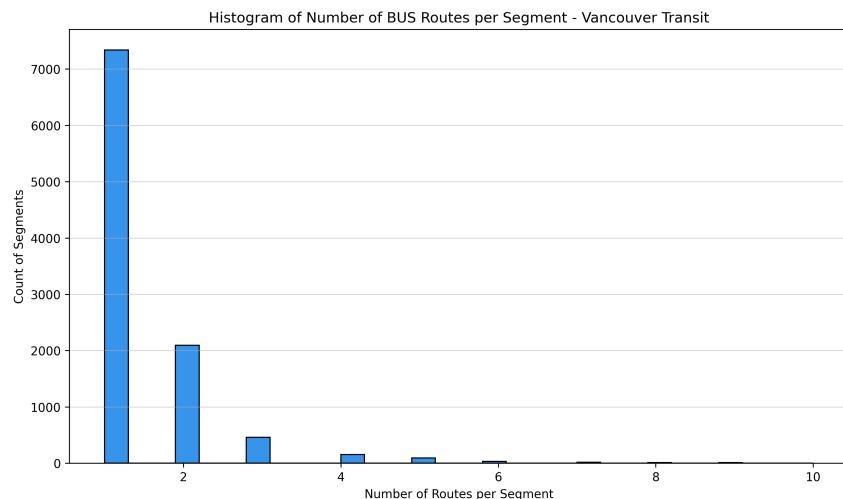


Figura 3: Histograma de cantidad de rutas por segmento.

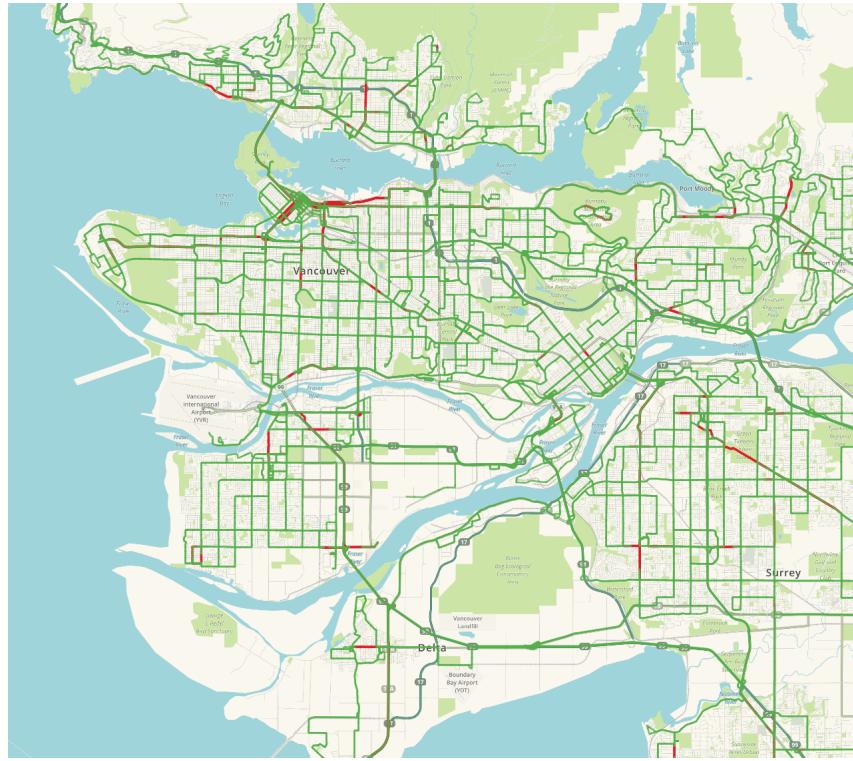


Figura 4: Mapa de densidad de rutas sobre la red de segmentos.

El histograma evidencia que la mayoría de segmentos tiene pocas rutas superpuestas, mientras que un número reducido de corredores concentra un alto nivel de servicio. En el mapa estos corredores de alta densidad aparecen como líneas más intensas alrededor del centro y en ejes troncales, mientras que las zonas periféricas muestran menor superposición de rutas.

### 3.3. Duplicación de rutas

La duplicación de recorridos se analiza mediante el cálculo de segmentos compartidos entre pares de rutas. La siguiente consulta (fragmento) construye la vista `route_duplication`:

```

DROP MATERIALIZED VIEW IF EXISTS route_duplication;
CREATE MATERIALIZED VIEW route_duplication AS
WITH route_segment_pairs AS (
    SELECT DISTINCT
        rs1.route_id AS route1,
        rs2.route_id AS route2,
        COUNT(DISTINCT CONCAT(rs1.stop1_id, rs1.stop2_id)) AS
            shared_segments
    FROM route_segments rs1
    JOIN route_segments rs2
        ON rs1.stop1_id = rs2.stop1_id
        AND rs1.stop2_id = rs2.stop2_id
)
```

```

    AND rs1.route_id < rs2.route_id
  WHERE rs1.seg_geom IS NOT NULL
    AND rs2.seg_geom IS NOT NULL
 GROUP BY rs1.route_id, rs2.route_id
 HAVING COUNT(DISTINCT CONCAT(rs1.stop1_id, rs1.stop2_id)) >= 5
)
SELECT *
FROM route_segment_pairs;

```

Sobre esta vista se calcula el porcentaje de solapamiento relativo para cada par de rutas y se identifica el conjunto `highly_duplicated_routes`, es decir, las rutas que comparten una fracción significativa de su recorrido con varias otras.

El script `route_duplication_analysis.py` produce la matriz de calor y estadísticos que se presentan en la Figura 5.

#### Falta generar resultado.

Genere el archivo

`.. /static/analysis/queries/results/route_duplication/route_duplication_heatmap.png` ejecutando

Figura 5: Matriz de calor de solapamiento entre rutas (`route_duplication_heatmap.png`).

### 3.4. Análisis de velocidades

La velocidad promedio por segmento y por ruta se obtiene a partir de la duración entre arribo a paradas consecutivas y la longitud del segmento geométrico:

```

DROP MATERIALIZED VIEW IF EXISTS schedule_speeds;
CREATE MATERIALIZED VIEW schedule_speeds AS
SELECT
  s.route_id || stop1_sequence || stop2_sequence AS id,
  AVG(seg_length / EXTRACT(EPOCH FROM
    (stop2_arrival_time - stop1_arrival_time)) * 3.6) AS
    speed_kmh,
  seg_geom
FROM route_segments s
WHERE stop2_arrival_time <> stop1_arrival_time
  AND seg_length > 0
GROUP BY s.route_id, stop1_sequence, stop2_sequence, seg_geom;

```

A partir de esta vista se derivan estadísticas agregadas por ruta (`route_speed_stats`) y se identifican segmentos con velocidades inusualmente altas (`high_speed_segments`). El script `speed_analysis.py` genera tanto mapas como gráficos resumiendo estas métricas, mostrados en las Figuras 6 y 7.

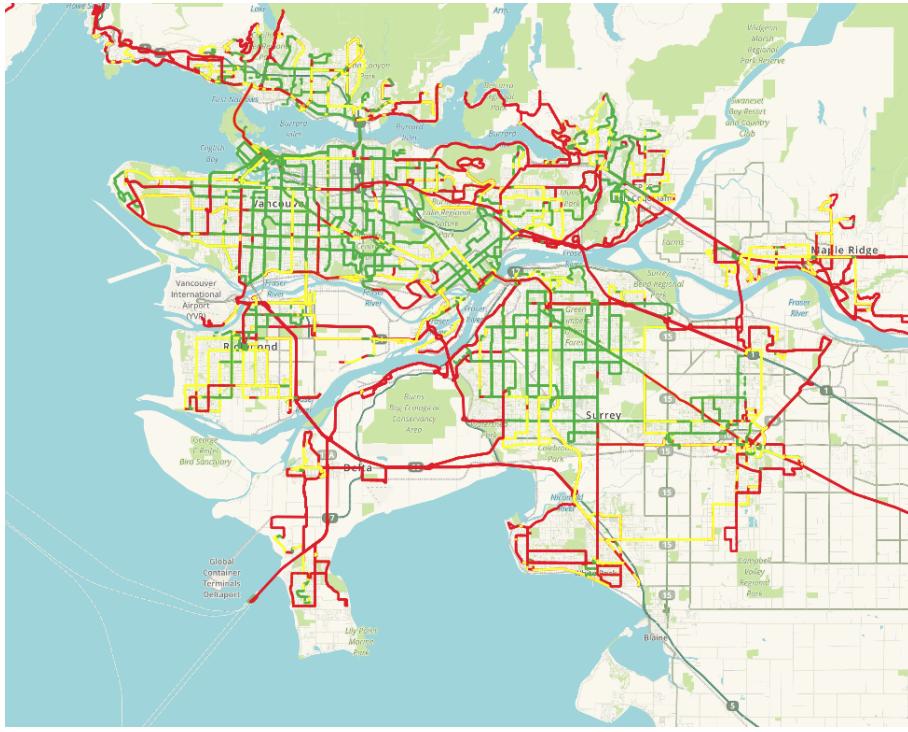


Figura 6: Mapa de velocidades promedio sobre la red de segmentos.

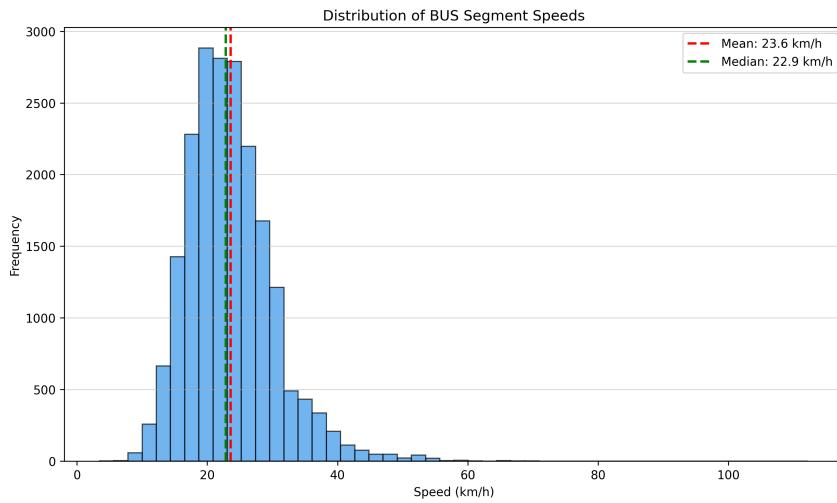


Figura 7: Histograma de velocidades programadas por segmento.

En el mapa se aprecian claramente los tramos más rápidos (corredores de acceso rápido y autopistas urbanas) frente a zonas de menor velocidad en el centro y en arterias congestionadas. El histograma muestra una distribución concentrada en torno a velocidades medianas (20–40 km/h), con colas hacia valores muy bajos que corresponden a tramos con fuerte fricción (semáforos, giros, áreas céntricas).

### 3.5. Accesibilidad a estadios

Se modelan estadios y arenas deportivas como puntos de interés dentro del área de estudio. La tabla football\_stadiums se puebla con coordenadas de BC Place, Rogers Arena y Pacific Coliseum, y se construyen métricas de acceso en transporte público:

```
CREATE TABLE football_stadiums (
    id serial PRIMARY KEY,
    name text NOT NULL,
    team text,
    latitude float,
    longitude float,
    geom geometry(Point, 4326)
);

CREATE MATERIALIZED VIEW stadium_transit_access AS
SELECT
    s.name AS stadium_name,
    s.team,
    (SELECT COUNT(*)
     FROM stops st
     WHERE ST_DistanceSphere(s.geom, st.stop_loc::geometry) <= 500)
        AS stops_500m,
    (SELECT COUNT(DISTINCT t.route_id)
     FROM stops st
     JOIN stop_times stt ON st.stop_id = stt.stop_id
     JOIN trips t ON stt.trip_id = t.trip_id
     WHERE ST_DistanceSphere(s.geom, st.stop_loc::geometry) <= 500)
        AS unique_routes_500m
FROM football_stadiums s;
```

El script stadium\_proximity\_analysis.py resume el número de viajes por franja horaria y genera tanto mapas como gráficos de demanda hacia cada estadio.

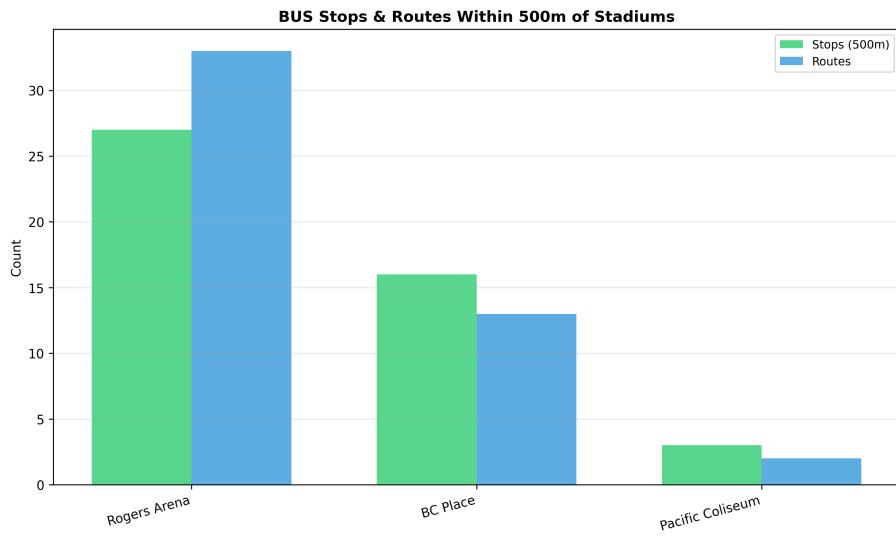


Figura 8: Mapa de estadios, paradas cercanas y rutas que los sirven.

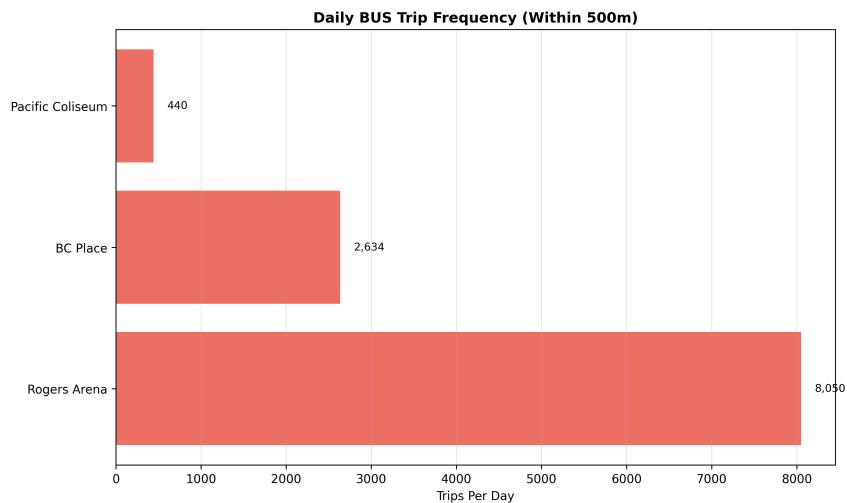


Figura 9: Viajes diarios que pasan cerca de cada estadio.

En el mapa se aprecia que BC Place y Rogers Arena se encuentran en un nodo concentrado de rutas y paradas, mientras que otros recintos presentan menor densidad de servicio en su entorno inmediato. El gráfico de viajes diarios muestra que los estadios céntricos reciben significativamente más oferta de transporte que aquellos ubicados en zonas residenciales o periféricas.

## 4. Análisis con GTFS-Realtime

Los análisis sobre datos en tiempo real se basan en las trayectorias *map-matched* almacenadas en `realtime_trips_mdb`, las cuales se comparan con sus equivalentes

programados en scheduled\_trips\_mdb.

#### 4.1. Velocidad real vs. velocidad programada

La comparación de velocidades se realiza mediante la vista materializada realtime\_speed\_comparison que calcula las velocidades programadas y observadas para cada segmento:

```
DROP MATERIALIZED VIEW IF EXISTS realtime_speed_comparison;
CREATE MATERIALIZED VIEW realtime_speed_comparison AS
WITH with_next_stop AS (
    SELECT
        d.trip_instance_id,
        d.trip_id,
        d.route_id,
        d.service_date,
        d.stop_sequence,
        d.stop_id,
        d.actual_arrival,
        LEAD(d.stop_sequence) OVER w AS next_stop_sequence,
        LEAD(d.stop_id) OVER w AS next_stop_id,
        LEAD(d.actual_arrival) OVER w AS next_actual_arrival
    FROM rt_trip_updates_deduped d
    WINDOW w AS (PARTITION BY d.trip_instance_id ORDER BY d.
        stop_sequence)
)
SELECT
    w.trip_instance_id,
    w.trip_id,
    r.route_short_name,
    w.route_id,
    rs.seg_length AS segment_length_m,
    EXTRACT(EPOCH FROM (rs.stop2_arrival_time - rs.
        stop1_arrival_time)) AS scheduled_seconds,
    EXTRACT(EPOCH FROM (w.next_actual_arrival - w.actual_arrival)) AS actual_seconds,
    (rs.seg_length / NULLIF(EXTRACT(EPOCH FROM (rs.
        stop2_arrival_time - rs.stop1_arrival_time)), 0) * 3.6) AS
        scheduled_speed_kmh,
    (rs.seg_length / NULLIF(EXTRACT(EPOCH FROM (w.
        next_actual_arrival - w.actual_arrival)), 0) * 3.6) AS
        actual_speed_kmh
FROM with_next_stop w
JOIN route_segments rs
    ON rs.trip_id = w.trip_id
    AND rs.stop1_sequence = w.stop_sequence
LEFT JOIN routes r ON r.route_id = w.route_id
WHERE w.next_actual_arrival IS NOT NULL
    AND rs.seg_length > 10
```

```

AND EXTRACT(EPOCH FROM (rs.stop2_arrival_time - rs.
stop1_arrival_time)) > 0
AND EXTRACT(EPOCH FROM (w.next_actual_arrival - w.
actual_arrival)) > 0;

```

Los resultados de esta consulta se visualizan mediante un conjunto de gráficos que comparan velocidades programadas vs. observadas. En la Figura 10 se ilustra el diagrama de dispersión para una corrida de análisis, donde cada punto representa un segmento entre paradas consecutivas y se aprecia que muchos puntos quedan por debajo de la línea de igualdad (velocidades reales menores a las programadas), sobre todo en los tramos teóricamente más rápidos.

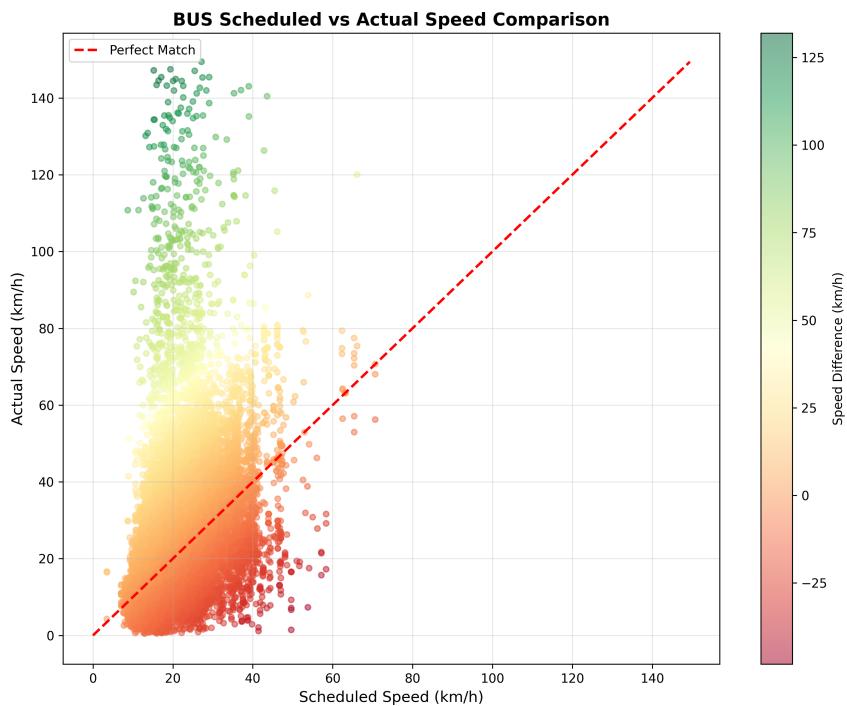


Figura 10: Comparación de velocidad programada vs. observada.

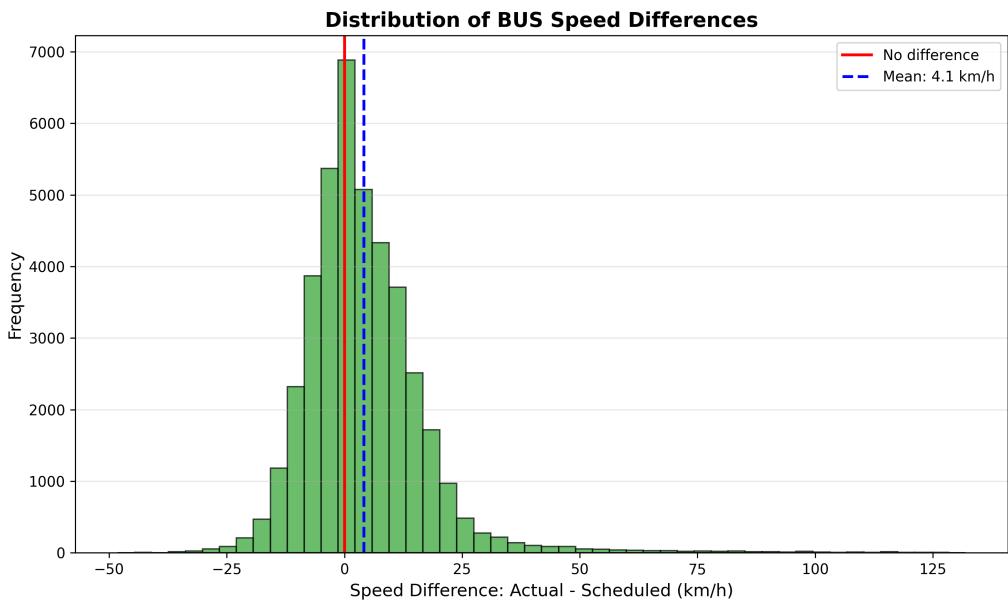


Figura 11: Distribución de la diferencia de velocidad (real – programada).

El histograma de diferencias (Figura ??) muestra una distribución aproximadamente unimodal, centrada levemente por encima de cero: en promedio los buses circulan unos pocos km/h más rápido que lo previsto, pero con colas hacia valores positivos que evi-dencian episodios de sobreejecución y hacia valores negativos asociadas a tramos muy congestionados.

## 4.2. Desempeño de horarios (schedule times)

La comparación de horarios programados vs. observados se realiza mediante la vista materializada `realtime_schedule_times`:

```
DROP MATERIALIZED VIEW IF EXISTS realtime_schedule_times;
CREATE MATERIALIZED VIEW realtime_schedule_times AS
SELECT
    d.trip_instance_id,
    d.trip_id,
    r.route_short_name,
    r.route_long_name,
    d.route_id,
    d.service_date,
    d.stop_sequence,
    d.stop_id,
    s.stop_name,
    ts.arrival_time AS scheduled_arrival_interval,
    d.actual_arrival,
    d.actual_departure,
    d.arrival_delay_seconds,
```

```

d.departure_delay_seconds,
d.arrival_delay_seconds / 60.0 AS delay_minutes,
EXTRACT(hour FROM d.actual_arrival) AS hour_of_day,
EXTRACT(dow FROM d.actual_arrival) AS day_of_week,
CASE
    WHEN EXTRACT(dow FROM d.actual_arrival) IN (0, 6) THEN '
        Weekend'
    ELSE 'Weekday'
END AS day_type
FROM rt_trip_updates_deduped d
JOIN routes r ON r.route_id = d.route_id
LEFT JOIN stops s ON s.stop_id = d.stop_id
LEFT JOIN transit_stops ts
    ON ts.trip_id = d.trip_id
    AND ts.stop_sequence = d.stop_sequence
WHERE d.arrival_delay_seconds IS NOT NULL;

```

Sobre esta vista se construyen distintos gráficos que resumen el comportamiento de las demoras (histogramas, boxplots por hora, promedios por ruta, etc.). En la Figura 11 se muestra el histograma de demoras generado para una corrida específica, donde se observa una cola hacia valores positivos que indica predominio de demoras por encima del horario programado.

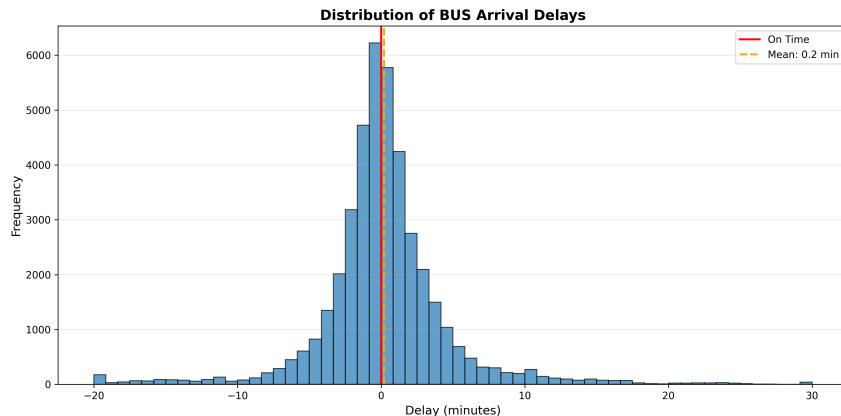


Figura 12: Distribución de demoras respecto al horario programado.

### 4.3. Segmentos de demora y patrones de congestión

El análisis de demoras por segmento se basa en la vista materializada realtime\_delay\_analysis:

```

DROP MATERIALIZED VIEW IF EXISTS realtime_delay_analysis;
CREATE MATERIALIZED VIEW realtime_delay_analysis AS
WITH with_next AS (
    SELECT
        d.trip_instance_id,
        d.trip_id,

```

```

d.route_id,
d.service_date,
d.stop_sequence AS from_seq,
d.stop_id AS from_stop_id,
d.actual_arrival AS from_arrival,
d.arrival_delay_seconds AS from_delay,
LEAD(d.stop_sequence) OVER w AS to_seq,
LEAD(d.stop_id) OVER w AS to_stop_id,
LEAD(d.actual_arrival) OVER w AS to_arrival,
LEAD(d.arrival_delay_seconds) OVER w AS to_delay
FROM rt_trip_updates_deduped d
WINDOW w AS (PARTITION BY d.trip_instance_id ORDER BY d.
stop_sequence)
)
SELECT
w.trip_instance_id,
w.trip_id,
r.route_short_name,
w.route_id,
w.from_seq,
w.to_seq,
rs.seg_length AS segment_length_m,
rs.seg_geom,
EXTRACT(EPOCH FROM (rs.stop2_arrival_time - rs.
stop1_arrival_time)) AS scheduled_seconds,
EXTRACT(EPOCH FROM (w.to_arrival - w.from_arrival)) AS
actual_seconds,
(w.to_delay - w.from_delay) AS segment_delay_change,
(EXTRACT(EPOCH FROM (w.to_arrival - w.from_arrival)) -
EXTRACT(EPOCH FROM (rs.stop2_arrival_time - rs.
stop1_arrival_time))) / 60.0 AS segment_delay_minutes,
EXTRACT(hour FROM w.from_arrival) AS hour_of_day,
EXTRACT(dow FROM w.from_arrival) AS day_of_week,
CASE
WHEN EXTRACT(dow FROM w.from_arrival) IN (0, 6) THEN '
Weekend'
ELSE 'Weekday'
END AS day_type,
CASE
WHEN EXTRACT(hour FROM w.from_arrival) BETWEEN 7 AND 9
THEN 'Morning_Rush'
WHEN EXTRACT(hour FROM w.from_arrival) BETWEEN 16 AND 18
THEN 'Evening_Rush'
WHEN EXTRACT(hour FROM w.from_arrival) BETWEEN 9 AND 16
THEN 'Midday'
WHEN EXTRACT(hour FROM w.from_arrival) BETWEEN 18 AND 22
THEN 'Evening'
ELSE 'Night'

```

```

END AS time_period
FROM with_next w
JOIN route_segments rs
  ON rs.trip_id = w.trip_id
  AND rs.stop1_sequence = w.from_seq
LEFT JOIN routes r ON r.route_id = w.route_id
WHERE w.to_arrival IS NOT NULL
  AND rs.seg_length > 10
  AND EXTRACT(EPOCH FROM (rs.stop2_arrival_time - rs.
    stop1_arrival_time)) > 0
  AND EXTRACT(EPOCH FROM (w.to_arrival - w.from_arrival)) > 0;

```

El análisis agrega esta información en distintos gráficos (por hora, por tipo de día, por severidad de la demora, etc.). En particular, la Figura 12 resume gráficamente los segmentos más afectados en términos de demora promedio.

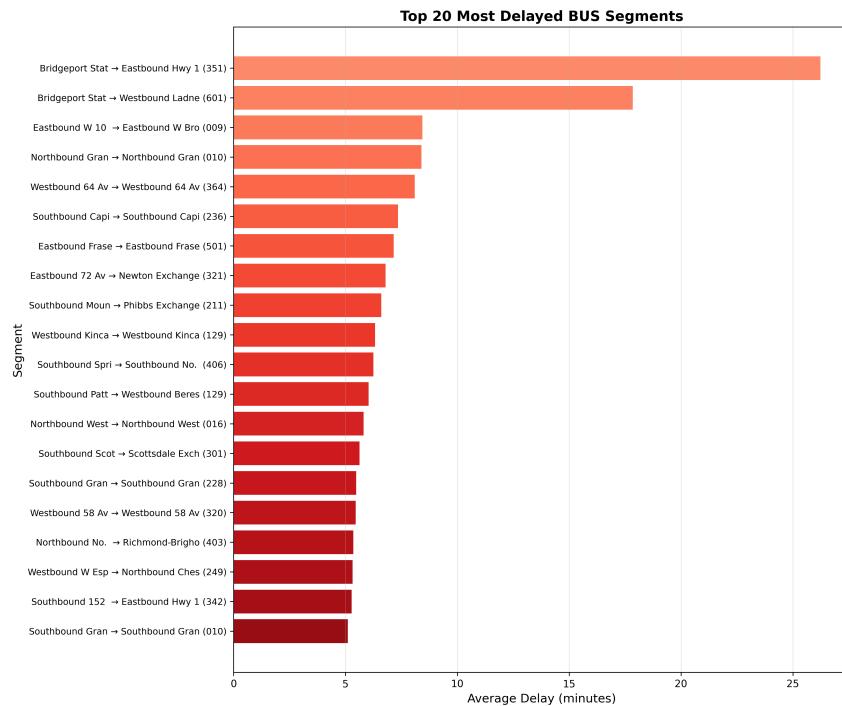


Figura 13: Segmentos con mayores demoras promedio.

#### 4.4. Regularidad de headways y *bus bunching*

El análisis de headways se basa en la vista materializada `realtime_headway_stats`, que calcula los intervalos entre vehículos consecutivos:

```

DROP MATERIALIZED VIEW IF EXISTS realtime_headway_stats;
CREATE MATERIALIZED VIEW realtime_headway_stats AS
WITH stop_arrivals AS (
  SELECT

```

```

        rtu.route_id,
        r.route_short_name,
        rtu.stop_id,
        s.stop_name,
        rtu.trip_instance_id,
        rtu.trip_id,
        rtu.arrival_time,
EXTRACT(hour FROM rtu.arrival_time) AS hour_of_day,
EXTRACT(dow FROM rtu.arrival_time) AS day_of_week,
CASE
    WHEN EXTRACT(dow FROM rtu.arrival_time) IN (0, 6) THEN '
        Weekend'
    ELSE 'Weekday'
END AS day_type,
CASE
    WHEN EXTRACT(hour FROM rtu.arrival_time) BETWEEN 7 AND
        9 THEN 'Morning_Rush'
    WHEN EXTRACT(hour FROM rtu.arrival_time) BETWEEN 16 AND
        18 THEN 'Evening_Rush'
    WHEN EXTRACT(hour FROM rtu.arrival_time) BETWEEN 9 AND
        16 THEN 'Midday'
    WHEN EXTRACT(hour FROM rtu.arrival_time) BETWEEN 18 AND
        22 THEN 'Evening'
    ELSE 'Night'
END AS time_period
FROM rt_trip_updates rtu
JOIN routes r ON r.route_id = rtu.route_id
LEFT JOIN stops s ON s.stop_id = rtu.stop_id
WHERE rtu.arrival_time IS NOT NULL
    AND rtu.stop_id IS NOT NULL
),
with_prev AS (
    SELECT
        *,
        LAG(arrival_time) OVER (
            PARTITION BY route_id, stop_id
            ORDER BY arrival_time
        ) AS prev_arrival,
        LAG(trip_instance_id) OVER (
            PARTITION BY route_id, stop_id
            ORDER BY arrival_time
        ) AS prev_trip_instance_id
    FROM stop_arrivals
)
SELECT
    route_id,
    route_short_name,
    stop_id,

```

```

stop_name,
trip_instance_id,
prev_trip_instance_id,
arrival_time,
prev_arrival,
EXTRACT(EPOCH FROM (arrival_time - prev_arrival)) / 60.0 AS
    headway_minutes,
hour_of_day,
day_of_week,
day_type,
time_period
FROM with_prev
WHERE prev_arrival IS NOT NULL
AND trip_instance_id != prev_trip_instance_id
AND EXTRACT(EPOCH FROM (arrival_time - prev_arrival)) > 0
AND EXTRACT(EPOCH FROM (arrival_time - prev_arrival)) < 7200;

```

El script `headway_analysis.py` utiliza esta vista para categorizar los headways como:

- **Bunched** (< 3 minutos): vehículos demasiado próximos.
- **Good** (3–10 minutos): intervalo deseable en rutas frecuentes.
- **Acceptable** (10–20 minutos): servicio menos frecuente pero razonable.
- **Gap** (> 20 minutos): esperas prolongadas.

La Figura 13 muestra la distribución de headways obtenida a partir de los registros realtime.

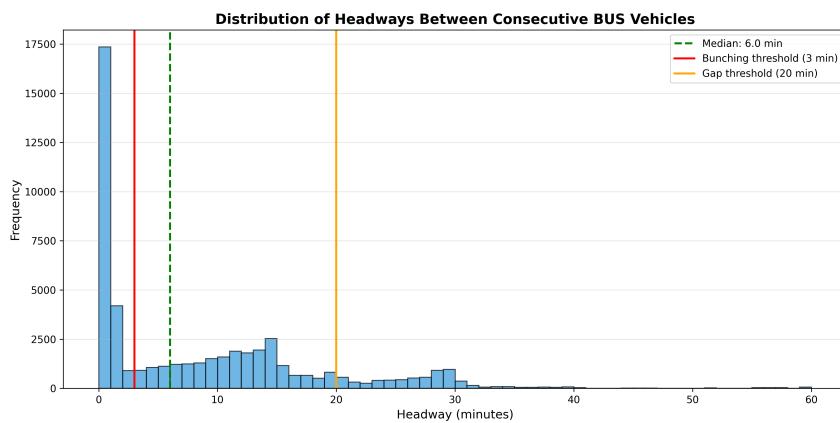


Figura 14: Distribución de headways entre vehículos.

## **5. Discusión y trabajo futuro**

El pipeline desarrollado permite integrar datos GTFS estáticos y GTFS-Realtime en una única base de datos espacial y reproducir de forma sistemática una serie de consultas y visualizaciones sobre la red de transporte de Vancouver. La separación clara entre scripts de ingesta, consultas SQL y scripts de visualización facilita la extensión del análisis a nuevas ciudades o a períodos de tiempo adicionales.

Como trabajo futuro se identifican varias líneas de avance: incorporar análisis de robustez ante fallas (por ejemplo, cierres de estaciones), extender las métricas de confiabilidad al nivel de pasajero (tiempos puerta a puerta) e integrar datos de demanda (conteos o validaciones) para estudiar la relación entre oferta, congestión y ocupación vehicular.