Divvy Tripdata Database

Introduction

To investigate ride behavior differences between casual and member users and uncover temporal and spatial patterns in ride activity, a comprehensive and well-structured database is essential. The analysis focuses on understanding how ride patterns vary across time—daily, weekly, and seasonally—and space—stations and routes—while identifying trends in ride duration, station popularity, and overall demand. These insights are critical for guiding Divvy's operational decisions and marketing strategies.

The source data for this project consists of 12 monthly Divvy trip datasets for the year 2024, containing ride-level information such as ride identifiers, timestamps, start and end stations, and user type (casual vs. member). To efficiently support analysis, a relational database will be designed to:

- Consolidate the monthly datasets into a single, queryable structure.
- Maintain data integrity with primary keys and appropriate data types for timestamps, text fields, and identifiers.
- Enable temporal analysis by storing ride start and end times in a standardized timestamp format.
- Support spatial analysis by including station names and IDs, allowing examination of station popularity and route patterns.
- Facilitate user segmentation by distinguishing between casual and member riders.

By implementing this database, analysts will be able to efficiently query and aggregate data, uncover patterns in ride behavior, and generate actionable insights for Divvy's operational planning and marketing initiatives.

Database Creation

Enviroment Setup

In this setup chunk, I prepare the R environment for the project. I suppress warnings and messages, set the CRAN mirror to avoid installation errors, and ensure that all the necessary packages (RPostgres, DBI, ini, tidyverse, readr, glue, ISOweek) are installed and loaded. This guarantees that all dependencies are available before running any database or data processing operations.

```
knitr::opts_chunk$set(echo = TRUE, message = FALSE, warning = FALSE)

# Show all rows in tibbles/data frames
options(dplyr.print_min = Inf, dplyr.print_max = Inf)


# Set default CRAN mirror (fixes the error)
options(repos = c(CRAN = "https://cloud.r-project.org"))

#Install the following libraries if required
# List of required packages
required_pkgs <- c("RPostgres", "DBI", "ini", "tidyverse", "readr", "glue", "ISOweek")

# Install and load each package
for (pkg in required_pkgs) {
  if (!requireNamespace(pkg, quietly = TRUE)) {
    install.packages(pkg)
```

```
  }
  library(pkg, character.only = TRUE)
}
```

Database Connection

Here, I read my PostgreSQL database credentials from a secure .ini configuration file and establish a connection using the RPostgres driver. I wrap the connection inside a tryCatch() block to handle connection errors gracefully. Once the connection is established, I register it for use by all subsequent SQL chunks.

```
# Read config
config <- read.ini("db_config.ini")
db <- config$postgresql

# Safe database connection
tryCatch({
  con <- dbConnect(
    Postgres(),
    host = db$host,
    dbname = db$database,
    user = db$user,
    password = db$password,
    port = as.integer(db$port)
  )
}, error = function(e) {
  stop("Database connection failed: ", e$message)
})

# Register connection for SQL chunks
knitr::opts_chunk$set(connection = con)
```

Prerequisites

I enable the PostgreSQL pgcrypto extension, which allows me to use functions for encryption, hashing, and UUID generation. This is especially helpful for generating unique identifiers or securing sensitive fields.

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

In this chunk, I enable the `btree_gin` extension to improve query performance on composite or full-text indexes. This will enhance the speed of analytical queries and materialized view refreshes later in the project.

```
CREATE EXTENSION IF NOT EXISTS btree_gin;
```

Schema & Base Tables

I create a dedicated schema named divvy. This schema acts as a container for all Divvy-related tables, views, and functions, keeping the project organized and logically separated from other database objects.

```
CREATE SCHEMA IF NOT EXISTS divvy;
```

This R chunk automatically creates 12 monthly staging tables, one for each month of 2024. Each table structure matches the columns of the Divvy CSV data files. Before creating a new table, I drop any existing table of the same name to avoid conflicts or outdated structures.

```
months <- c("january","february","march","april","may","june",
            "july","august","september","october","november","december")
```

```
for (m in months) {
  sql <- glue("
    CREATE TABLE IF NOT EXISTS divvy.{m} (
      ride_id             TEXT PRIMARY KEY,
      rideable_type       TEXT,
      started_at          TIMESTAMP,
      ended_at            TIMESTAMP,
      start_station_name  TEXT,
      start_station_id    TEXT,
      end_station_name    TEXT,
      end_station_id      TEXT,
      member_casual       TEXT
    );
  ")
  # Drop old table if exists
  dbExecute(con, glue("DROP TABLE IF EXISTS divvy.{m} CASCADE;"))

  # Create new table
  dbExecute(con, sql)
}
```

Load the CSVs

I loop through all twelve monthly CSV files, load each into R, clean and align the column names with the expected schema, and then write each dataset into its corresponding monthly table inside the PostgreSQL database. This ensures each month's data is properly imported and stored for further processing.

```
# month name for a given numeric month
month_name <- function(m) tolower(
  format(as.Date(paste0("2024-", sprintf("%02d", m), "-01")), "%B"))

# loop and load
for (m in 1:12) {
  fname <- sprintf("resources/data/2024%02d-divvy-tripdata.csv", m)
  tbl   <- month_name(m)

  message("Loading: ", fname, " -> divvy.", tbl)
  df <- readr::read_csv(fname, show_col_types = FALSE)

  # select/rename only the columns we expect
  expect <- c("ride_id","rideable_type","started_at","ended_at",
              "start_station_name","start_station_id",
              "end_station_name","end_station_id","member_casual")
  df <- df[, intersect(expect, names(df))]

  DBI::dbWriteTable(
    con,
    name = DBI::Id(schema="divvy", table=tbl),
    value = df,
    append = TRUE,
    row.names = FALSE
  )
}
```

Normalize & Combine

We'll build a normalized core: - dim_station: unique stations by station_id with most recent name seen (names can drift). - dim_date: calendar table for temporal joins. - dim_member_type, dim_bike_type: small lookup tables. - fact_trips: the single deduplicated table referencing dimensions.

Dimension tables **Station dimension** I create a dimension table called `dim_station` to store information about each bike station. This includes station ID, name, and coordinates (latitude and longitude). It provides a reference point for mapping and spatial analysis.

```
CREATE TABLE IF NOT EXISTS divvy.dim_station (
  station_id   TEXT PRIMARY KEY,
  station_name TEXT,
  latitude     DOUBLE PRECISION,
  longitude    DOUBLE PRECISION,
  updated_at   TIMESTAMP DEFAULT now()
);
```

Member type dimension This chunk defines the `dim_member_type` table, which holds information about user types—whether the rider is a "member" or "casual." It's a small but essential lookup table that helps enforce referential integrity and simplifies queries that aggregate data by rider type.

```
CREATE TABLE IF NOT EXISTS divvy.dim_member_type (
  member_type_id SMALLSERIAL PRIMARY KEY,
  member_casual  TEXT UNIQUE
);
```

I populate the dim_member_type table with the two standard values: "member" and "casual." I include an ON CONFLICT DO NOTHING clause so that the insert operation won't fail if these records already exist.

```
INSERT INTO divvy.dim_member_type(member_casual)
VALUES ('member'),('casual')
ON CONFLICT (member_casual) DO NOTHING;
```

Here, I create another dimension table called `dim_bike_type`. This table stores unique bike types, such as "electric," "classic," or "docked." By separating these into a dimension, I make the schema more normalized and efficient to query.

```
CREATE TABLE IF NOT EXISTS divvy.dim_bike_type (
  bike_type_id SMALLSERIAL PRIMARY KEY,
  rideable_type TEXT UNIQUE
);
```

I insert the known bike types into `dim_bike_type`, ensuring that any duplicates are ignored using ON CONFLICT DO NOTHING. This provides a clean and consistent reference for all rides in the fact table.

```
INSERT INTO divvy.dim_bike_type(rideable_type)
SELECT DISTINCT rideable_type
FROM (
  SELECT rideable_type FROM divvy.january
  UNION ALL SELECT rideable_type FROM divvy.february
  UNION ALL SELECT rideable_type FROM divvy.march
  UNION ALL SELECT rideable_type FROM divvy.april
  UNION ALL SELECT rideable_type FROM divvy.may
  UNION ALL SELECT rideable_type FROM divvy.june
  UNION ALL SELECT rideable_type FROM divvy.july
  UNION ALL SELECT rideable_type FROM divvy.august
  UNION ALL SELECT rideable_type FROM divvy.september
```

```
  UNION ALL SELECT rideable_type FROM divvy.october
  UNION ALL SELECT rideable_type FROM divvy.november
  UNION ALL SELECT rideable_type FROM divvy.december
) s
WHERE rideable_type IS NOT NULL
ON CONFLICT (rideable_type) DO NOTHING;
```

Build/refresh dim_station from both start and end stations

```
WITH stations AS (
  SELECT start_station_id AS station_id, max(start_station_name) AS station_name
  FROM (
    SELECT * FROM divvy.january
    UNION ALL SELECT * FROM divvy.february
    UNION ALL SELECT * FROM divvy.march
    UNION ALL SELECT * FROM divvy.april
    UNION ALL SELECT * FROM divvy.may
    UNION ALL SELECT * FROM divvy.june
    UNION ALL SELECT * FROM divvy.july
    UNION ALL SELECT * FROM divvy.august
    UNION ALL SELECT * FROM divvy.september
    UNION ALL SELECT * FROM divvy.october
    UNION ALL SELECT * FROM divvy.november
    UNION ALL SELECT * FROM divvy.december
  ) t
  WHERE start_station_id IS NOT NULL
  GROUP BY start_station_id
  UNION
  SELECT end_station_id AS station_id, max(end_station_name) AS station_name
  FROM (
    SELECT * FROM divvy.january
    UNION ALL SELECT * FROM divvy.february
    UNION ALL SELECT * FROM divvy.march
    UNION ALL SELECT * FROM divvy.april
    UNION ALL SELECT * FROM divvy.may
    UNION ALL SELECT * FROM divvy.june
    UNION ALL SELECT * FROM divvy.july
    UNION ALL SELECT * FROM divvy.august
    UNION ALL SELECT * FROM divvy.september
    UNION ALL SELECT * FROM divvy.october
    UNION ALL SELECT * FROM divvy.november
    UNION ALL SELECT * FROM divvy.december
  ) t
  WHERE end_station_id IS NOT NULL
  GROUP BY end_station_id
)
INSERT INTO divvy.dim_station(station_id, station_name)
SELECT DISTINCT ON (station_id) station_id, station_name
FROM stations
ORDER BY station_id, station_name
ON CONFLICT (station_id) DO UPDATE
SET station_name = EXCLUDED.station_name,
    updated_at   = now();
```

Date dimension

```
CREATE TABLE IF NOT EXISTS divvy.dim_date (
  date_id        DATE PRIMARY KEY,
  year           INT,
  quarter        INT,
  month          INT,
  week           INT,
  day            INT,
  day_of_week    INT,
  is_weekend     BOOLEAN
);
```

Populate for 2024

```
INSERT INTO divvy.dim_date(date_id, year, quarter, month, week, day, day_of_week, is_weekend)
SELECT d::date,
       EXTRACT(YEAR   FROM d)::int,
       EXTRACT(QUARTER FROM d)::int,
       EXTRACT(MONTH  FROM d)::int,
       EXTRACT(WEEK   FROM d)::int,
       EXTRACT(DAY    FROM d)::int,
       EXTRACT(DOW    FROM d)::int,
       (EXTRACT(DOW FROM d) IN (0,6))::boolean
FROM generate_series('2024-01-01'::date, '2024-12-31'::date, '1 day') AS s(d)
ON CONFLICT (date_id) DO NOTHING;
```

Fact table In this SQL chunk, I define the central fact table — `fact_trips`. It contains every individual trip record with foreign keys linking to the relevant dimensions (`bike_type_id`, `member_type_id`, and `station_id`). This is the heart of the star schema that enables efficient analytics and aggregations.

```
CREATE TABLE IF NOT EXISTS divvy.fact_trips (
  ride_id            TEXT PRIMARY KEY,
  bike_type_id       SMALLINT REFERENCES divvy.dim_bike_type(bike_type_id),
  member_type_id     SMALLINT REFERENCES divvy.dim_member_type(member_type_id),
  started_at         TIMESTAMP NOT NULL,
  ended_at           TIMESTAMP NOT NULL,
  start_station_id   TEXT REFERENCES divvy.dim_station(station_id),
  end_station_id     TEXT REFERENCES divvy.dim_station(station_id),

  -- Generated duration in minutes (kept in the fact for performance)
  ride_length_min    DOUBLE PRECISION GENERATED ALWAYS AS
                      (EXTRACT(EPOCH FROM (ended_at - started_at))/60.0) STORED,

  -- Date keys for fast joins to dim_date
  started_date       DATE GENERATED ALWAYS AS (started_at::date) STORED,
  ended_date         DATE GENERATED ALWAYS AS (ended_at::date) STORED,

  -- Basic data quality checks
  CONSTRAINT chk_positive_duration CHECK (ended_at >= started_at)
);
```

Helper mappings Temporary mapping tables for bike table upserts.

```sql
CREATE TEMP TABLE tmp_bike_map AS
SELECT rideable_type, bike_type_id FROM divvy.dim_bike_type;
```

Temporary mapping tables for member table upserts.

```sql
CREATE TEMP TABLE tmp_member_map AS
SELECT member_casual, member_type_id FROM divvy.dim_member_type;
```

Upsert into fact table This is where I load all twelve months of trip data from the staging tables into the main fact_trips table. During insertion, I join with the dimension tables to replace text values (like rideable_type and member_casual) with their corresponding foreign key IDs. This ensures data consistency and enforces referential integrity.

```sql
INSERT INTO divvy.fact_trips
(ride_id, bike_type_id, member_type_id, started_at, ended_at,
 start_station_id, end_station_id)
SELECT
  t.ride_id,
  b.bike_type_id,
  mt.member_type_id,
  t.started_at,
  t.ended_at,
  t.start_station_id,
  t.end_station_id
FROM (
  SELECT * FROM divvy.january
  UNION ALL SELECT * FROM divvy.february
  UNION ALL SELECT * FROM divvy.march
  UNION ALL SELECT * FROM divvy.april
  UNION ALL SELECT * FROM divvy.may
  UNION ALL SELECT * FROM divvy.june
  UNION ALL SELECT * FROM divvy.july
  UNION ALL SELECT * FROM divvy.august
  UNION ALL SELECT * FROM divvy.september
  UNION ALL SELECT * FROM divvy.october
  UNION ALL SELECT * FROM divvy.november
  UNION ALL SELECT * FROM divvy.december
) t
LEFT JOIN tmp_bike_map b ON t.rideable_type = b.rideable_type
LEFT JOIN tmp_member_map mt ON t.member_casual  = mt.member_casual
-- Basic sanity: require started_at/ended_at and member/bike maps
WHERE t.ride_id IS NOT NULL
  AND t.started_at IS NOT NULL
  AND t.ended_at   IS NOT NULL
  AND t.ended_at > t.started_at
  AND mt.member_type_id IS NOT NULL
  AND b.bike_type_id   IS NOT NULL
ON CONFLICT (ride_id) DO NOTHING;
```

Indexes

I added this index to make time-bounded scans fast (hours, days, months) without sweeping the entire table. It powers range filters like WHERE started_at BETWEEN ... and prepares data quickly for trend views. I accept a bit of write overhead in exchange for much faster reads, and I run ANALYZE after monthly loads.

```
CREATE INDEX IF NOT EXISTS idx_fact_trips_started_at
ON divvy.fact_trips(started_at);
```

Daily reporting is a core workflow for me, so I index a stored DATE rather than recalculating from the timestamp each time. This speeds WHERE started_date BETWEEN ... and GROUP BY started_date in my daily trends, with minimal storage cost. After bulk loads, I refresh stats and confirm plans with EXPLAIN.

```
CREATE INDEX IF NOT EXISTS idx_fact_trips_started_date
ON divvy.fact_trips(started_date);
```

Member vs casual is my primary segmentation, so I keep a lightweight index here to accelerate filters, splits, and joins to dim_member_type. On its own the selectivity is modest, but it shines when paired with date or station predicates in real queries.

```
CREATE INDEX IF NOT EXISTS idx_fact_member_type
ON divvy.fact_trips(member_type_id);
```

I frequently compare KPIs by bike class (classic/electric/docked). This compact index supports those slices and composes nicely with time and location filters. It's small, low-maintenance, and consistently helpful during EDA.

```
CREATE INDEX IF NOT EXISTS idx_fact_bike_type
ON divvy.fact_trips(bike_type_id);
```

Origin-focused analyses (busiest starts, station drill-downs) rely on quick point lookups. This index speeds WHERE start_station_id = ... and plays well with date filters for station heatmaps and dashboards. I keep an eye on skew in case a few stations dominate.

```
CREATE INDEX IF NOT EXISTS idx_fact_start_station
ON divvy.fact_trips(start_station_id);
```

Destination-centric questions (popular drop-offs, rebalancing signals) need the same fast path. This mirrors the start-station index to support flexible route exploration and joins to dim_station, with minimal overhead.

```
CREATE INDEX IF NOT EXISTS idx_fact_end_station
ON divvy.fact_trips(end_station_id);
```

Route (OD pair) analytics are central to flow insights. Ordering the columns as (start, end) matches my most common filter (origin first) and enables prefix scans for "all destinations from A." If most route queries add tight date windows, I may introduce a targeted three-column variant for hot paths.

```
CREATE INDEX IF NOT EXISTS idx_fact_route
ON divvy.fact_trips(start_station_id, end_station_id);
```

I routinely trim outliers and analyze duration-bounded subsets, so this index speeds WHERE ride_length_min BETWEEN .... Its value is highest when combined with time or station filters; if "valid ranges only" become standard, I'll consider a partial index to cut scan costs.

```
CREATE INDEX IF NOT EXISTS idx_fact_duration
ON divvy.fact_trips(ride_length_min);
```

This is my workhorse for "member vs casual by day." Equality on member type followed by a date range aligns perfectly with common filters and delivers efficient scans for daily summaries and dashboards. I keep the standalone started_date index for date-only workloads.

```
CREATE INDEX IF NOT EXISTS idx_fact_member_date
ON divvy.fact_trips(member_type_id, started_date);
```

Views

In this view, I combined all the essential trip details with computed fields like ride duration and date attributes. My goal was to enrich the base trip data for easier querying later on. By including calculated columns such as ride_length_min and day_of_week, I can perform time-based and duration-based analyses without recalculating these metrics repeatedly.

```
CREATE OR REPLACE VIEW divvy.vw_trips_enriched AS
SELECT
  f.ride_id,
  bt.rideable_type,
  mt.member_casual,
  f.started_at, f.ended_at, f.ride_length_min,
  f.start_station_id, s1.station_name AS start_station_name,
  f.end_station_id,   s2.station_name AS end_station_name,
  f.started_date, d.year, d.month, d.week, d.day_of_week, d.is_weekend
FROM divvy.fact_trips f
JOIN divvy.dim_bike_type   bt ON f.bike_type_id   = bt.bike_type_id
JOIN divvy.dim_member_type mt ON f.member_type_id = mt.member_type_id
LEFT JOIN divvy.dim_station s1 ON f.start_station_id = s1.station_id
LEFT JOIN divvy.dim_station s2 ON f.end_station_id   = s2.station_id
LEFT JOIN divvy.dim_date    d  ON f.started_date     = d.date_id;
```

I created this view to analyze ride activity on a daily basis. It summarizes the number of trips taken by members and casual users for each calendar date. This structure allows me to quickly visualize and compare daily ride patterns, identify high-demand days, and observe behavioral differences between user groups over time.

```
CREATE OR REPLACE VIEW divvy.vw_daily_counts AS
SELECT
  started_date,
  member_casual,
  COUNT(*) AS rides,
  AVG(ride_length_min) AS avg_min
FROM divvy.vw_trips_enriched
GROUP BY started_date, member_casual;
```

This view aggregates rides into weekly periods using ISO week notation. I use it to examine broader temporal trends and seasonal patterns. By grouping trips by week and user type, I can evaluate week-over-week changes, measure sustained demand, and identify potential seasonal effects in ride behavior.

```
CREATE OR REPLACE VIEW divvy.vw_weekly_counts AS
SELECT
  d.year,
  d.week,
  t.member_casual,
  COUNT(*) AS rides
FROM divvy.vw_trips_enriched t
JOIN divvy.dim_date d ON t.started_date = d.date_id
GROUP BY d.year, d.week, t.member_casual;
```

I built this view to pinpoint the most popular starting stations across the Divvy system. It ranks stations based on the number of trips initiated from each location. This helps me identify high-traffic areas, assess spatial demand distribution, and inform potential rebalancing or marketing strategies for specific neighborhoods.

```
CREATE OR REPLACE VIEW divvy.vw_top_start_stations AS
SELECT
```

```
  start_station_id,
  start_station_name,
  COUNT(*) AS rides
FROM divvy.vw_trips_enriched
GROUP BY start_station_id, start_station_name
ORDER BY rides DESC;
```

This view focuses on trip flows between start and end stations — essentially capturing the most common routes taken by users. By counting and grouping rides by both origin and destination, I can visualize movement patterns within the network, identify frequent commuter paths, and support infrastructure or operational planning.

```
CREATE OR REPLACE VIEW divvy.vw_route_counts AS
SELECT
  start_station_id, start_station_name,
  end_station_id,   end_station_name,
  COUNT(*) AS rides
FROM divvy.vw_trips_enriched
GROUP BY start_station_id, start_station_name, end_station_id, end_station_name;
```

I define a materialized view called `mv_monthly_summary` that pre-aggregates key metrics like total rides and average durations by month and user type. This helps me quickly run time-based analyses without recalculating from raw trip data every time.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS divvy.mv_monthly_summary AS
SELECT
  d.year, d.month,
  t.member_casual,
  COUNT(*) AS rides,
  AVG(t.ride_length_min) AS avg_min
FROM divvy.vw_trips_enriched t
JOIN divvy.dim_date d ON t.started_date = d.date_id
GROUP BY d.year, d.month, t.member_casual;
```

In this chunk, I include a command to refresh the materialized view when new trip data is added. This ensures that the summary statistics remain up to date with the latest data in the database.

```
REFRESH MATERIALIZED VIEW divvy.mv_monthly_summary;
```

Helper Functions

Normalize station name (trim/lower) for QA joins

```
CREATE OR REPLACE FUNCTION divvy.norm_station_name(txt TEXT)
RETURNS TEXT LANGUAGE sql IMMUTABLE AS $$
  SELECT NULLIF(regexp_replace(lower(trim(txt)), '\s+', ' ', 'g'),'');
$$;
```

Quick route key (for joining/BI)

```
CREATE OR REPLACE FUNCTION divvy.route_key(start_id TEXT, end_id TEXT)
RETURNS TEXT LANGUAGE sql IMMUTABLE AS $$
  SELECT start_id || '→' || end_id;
$$;
```

Exploratory Data Analysis (SQL)

Segment share

Segment share over time (daily)

```
SELECT *
FROM divvy.vw_daily_counts
ORDER BY started_date, member_casual;
```

Table 1: Displaying records 1 - 10

started_date	member_casual	rides	avg_min
2024-01-01	casual	1180	39.48705
2024-01-01	member	2478	11.56964
2024-01-02	casual	1154	15.43339
2024-01-02	member	5374	9.50178
2024-01-03	casual	1337	18.29907
2024-01-03	member	6127	10.33107
2024-01-04	casual	1507	17.11458
2024-01-04	member	6607	10.70772
2024-01-05	casual	1522	14.50350
2024-01-05	member	5861	11.00943

Weekly trend

Weekly trend by segment

```
SELECT *
FROM divvy.vw_weekly_counts
ORDER BY year, week, member_casual;
```

Table 2: Displaying records 1 - 10

year	week	member_casual	rides
2024	1	casual	10881
2024	1	member	39720
2024	2	casual	4909
2024	2	member	27196
2024	3	casual	2287
2024	3	member	13996
2024	4	casual	5589
2024	4	member	27178
2024	5	casual	8912
2024	5	member	39279

Duration distribution

```
SELECT
  width_bucket(ride_length_min, 0, 120, 12) AS bin, -- 10 min bins up to 120
  COUNT(*) AS rides
FROM divvy.vw_trips_enriched
WHERE ride_length_min BETWEEN 0 AND 180
```

```
GROUP BY bin
ORDER BY bin;
```

Table 3: Displaying records 1 - 10

bin	rides
1	3010305
2	1676020
3	588650
4	249257
5	120302
6	61554
7	37838
8	25390
9	17591
10	13096

Top 10 start/end stations

This query identifies the top start stations by total rides, ranking them in descending order. I join fact_trips with dim_station to retrieve human-readable names and then group by start_station_name. By analyzing the top stations for casual and member users separately, I can identify geographic hotspots — members may cluster around transit hubs or offices, while casuals may favor tourist or recreational areas. This insight supports both operational logistics (bike placement) and marketing segmentation.

```
SELECT * FROM divvy.vw_top_start_stations LIMIT 10;
```

Table 4: Displaying records 1 - 10

start_station_id	start_station_name	rides
NA	NA	1073546
13022	Streeter Dr & Grand Ave	66017
13300	DuSable Lake Shore Dr & Monroe St	43963
KA1503000043	Kingsbury St & Kinzie St	39657
13042	Michigan Ave & Oak St	39631
LF-005	DuSable Lake Shore Dr & North Blvd	39618
TA1307000039	Clark St & Elm St	35542
WL-012	Clinton St & Washington Blvd	34509
13008	Millennium Park	33165
TA1305000032	Clinton St & Madison St	33023

Similar to the previous chunk, this query ranks the top end stations by trip count. Analyzing both start and end station patterns helps reveal common travel flows — whether users tend to return bikes at the same stations or follow distinct routes. Comparing casual and member drop-off behavior highlights differences in trip intent: members may end rides near workplaces or transit, while casuals might finish near parks or attractions. This spatial perspective is crucial for demand forecasting and route planning.

```
SELECT
  end_station_id, end_station_name, COUNT(*) AS rides
FROM divvy.vw_trips_enriched
GROUP BY end_station_id, end_station_name
ORDER BY rides DESC
LIMIT 10;
```

Table 5: Displaying records 1 - 10

end_station_id	end_station_name	rides
NA	NA	1104115
13022	Streeter Dr & Grand Ave	67209
LF-005	DuSable Lake Shore Dr & North Blvd	43083
13300	DuSable Lake Shore Dr & Monroe St	42777
13042	Michigan Ave & Oak St	39564
KA1503000043	Kingsbury St & Kinzie St	39108
WL-012	Clinton St & Washington Blvd	34929
TA1307000039	Clark St & Elm St	34911
TA1305000032	Clinton St & Madison St	33792
13008	Millennium Park	33296

Top routes

In this SQL chunk, I identify the most common start–end station combinations, or "routes," based on the number of rides taken along each. I join the tables vw_route_counts and vw_trips_enriched using station IDs and group by start station, end station, and user type (member_casual). Then I order the results by descending ride counts and limit to the top 10.

```
SELECT
  start_station_name, end_station_name, member_casual,
  COUNT(*) AS rides
FROM divvy.vw_trips_enriched
GROUP BY start_station_name, end_station_name, member_casual
ORDER BY rides DESC
LIMIT 10;
```

Table 6: Displaying records 1 - 10

start_station_name	end_station_name	member_casual	rides
NA	NA	member	290355
NA	NA	casual	235647
Streeter Dr & Grand Ave	Streeter Dr & Grand Ave	casual	8871
DuSable Lake Shore Dr & Monroe St	DuSable Lake Shore Dr & Monroe St	casual	7242
State St & 33rd St	Calumet Ave & 33rd St	member	5587
Calumet Ave & 33rd St	State St & 33rd St	member	5506
DuSable Lake Shore Dr & Monroe St	Streeter Dr & Grand Ave	casual	5265
Michigan Ave & Oak St	Michigan Ave & Oak St	casual	4619
Ellis Ave & 60th St	University Ave & 57th St	member	3988
Ellis Ave & 60th St	Ellis Ave & 55th St	member	3976

Peak by hour and weekday

```
SELECT
  EXTRACT(DOW FROM started_at) AS dow,
  EXTRACT(HOUR FROM started_at) AS hr,
  COUNT(*) AS rides
FROM divvy.fact_trips f
GROUP BY 1,2
ORDER BY 1,2;
```

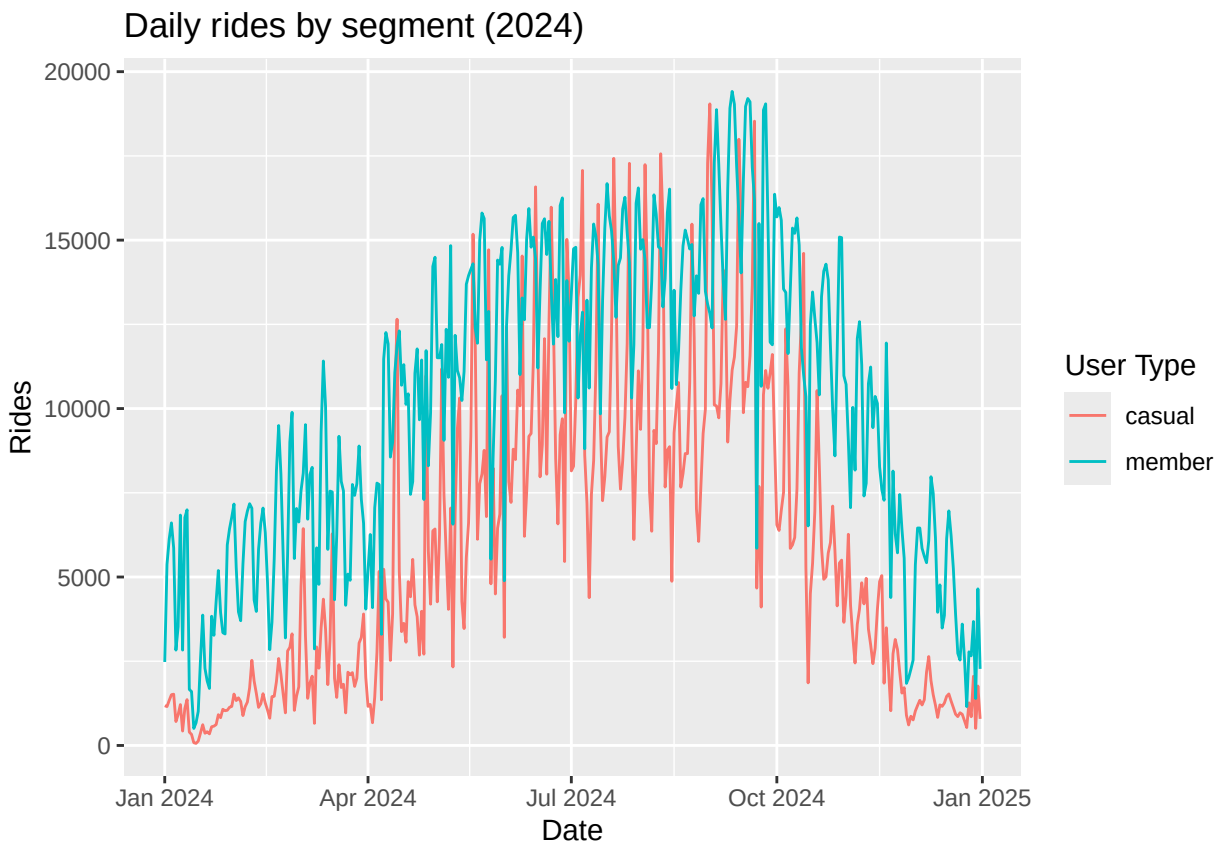
Table 7: Displaying records 1 - 10

dow	hr	rides
0	0	18682
0	1	13038
0	2	8080
0	3	4692
0	4	3356
0	5	3872
0	6	7646
0	7	12684
0	8	21681
0	9	34499

Exploratory Data Analysis (R)

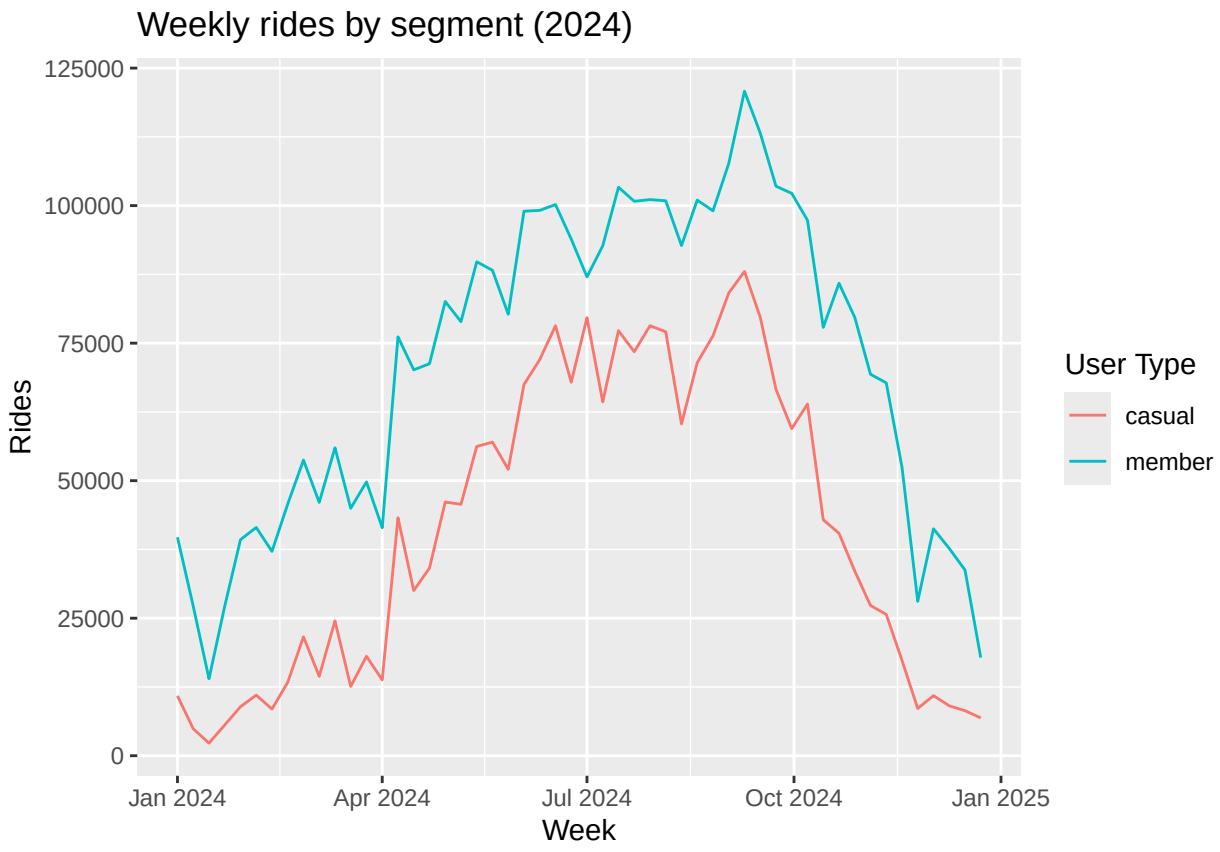
Daily rides by segment

```
# 8.1 Daily rides by segment
daily <- dbGetQuery(con, "SELECT started_date, member_casual, rides
                          FROM divvy.vw_daily_counts")
ggplot(daily, aes(as.Date(started_date), rides, color = member_casual)) +
  geom_line() +
  labs(title = "Daily rides by segment (2024)",
       x = "Date", y = "Rides", color = "User Type")
```



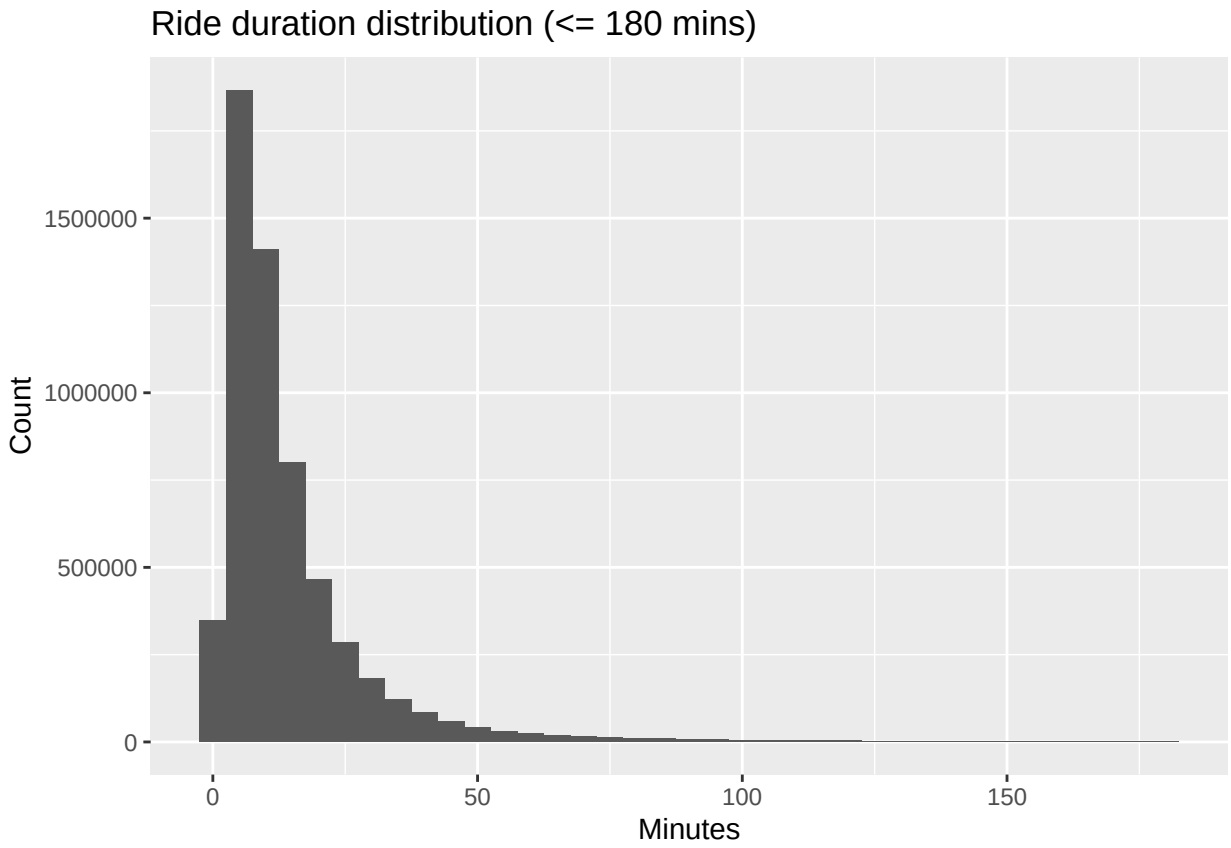
Weekly trend

```
# 8.2 Weekly trend
weekly <- dbGetQuery(con, "SELECT year, week, member_casual, rides
                           FROM divvy.vw_weekly_counts")
weekly$date <- ISOweek2date(paste0(weekly$year, "-W", sprintf("%02d", weekly$week), "-1"))
ggplot(weekly, aes(date, rides, color = member_casual)) +
  geom_line() +
  labs(title = "Weekly rides by segment (2024)",
       x = "Week", y = "Rides", color = "User Type")
```



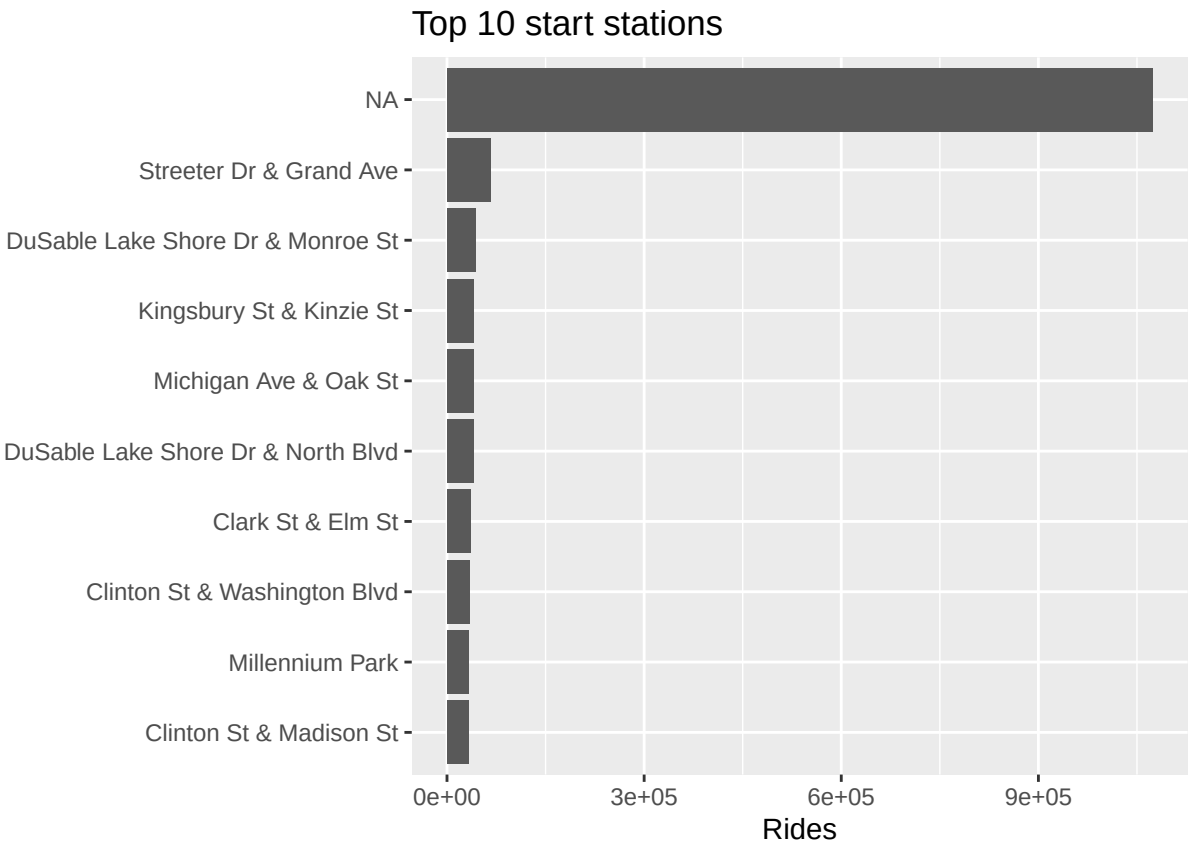
Duration distribution

```
# 8.3 Duration distribution
dur <- dbGetQuery(con, "
  SELECT ride_length_min
  FROM divvy.vw_trips_enriched
  WHERE ride_length_min BETWEEN 0 AND 180
")
ggplot(dur, aes(ride_length_min)) +
  geom_histogram(binwidth = 5) +
  labs(title = "Ride duration distribution (<= 180 mins)",
       x = "Minutes", y = "Count")
```



Top stations

```
# 8.4 Top stations
topstarts <- dbGetQuery(con, "
                        SELECT *
                        FROM divvy.vw_top_start_stations
                        LIMIT 10")
topstarts <- topstarts %>% mutate(rank = row_number())
ggplot(topstarts, aes(reorder(start_station_name, rides), rides)) +
  geom_col() +
  coord_flip() +
  labs(title="Top 10 start stations", x="", y="Rides")
```



Top routes by segment

```
routes <- dbGetQuery(con, "
  SELECT
    rc.start_station_name,
    rc.end_station_name,
    te.member_casual,
    rc.rides
  FROM divvy.vw_route_counts AS rc
  JOIN divvy.vw_trips_enriched AS te
    ON rc.start_station_id = te.start_station_id
   AND rc.end_station_id = te.end_station_id
  GROUP BY rc.start_station_name, rc.end_station_name, te.member_casual, rc.rides
  ORDER BY rc.rides DESC
  LIMIT 30;
")
```

Data Quality & Maintenance

Quick checks

I check for missing data in key columns such as start and end stations or bike type. This helps me identify potential data quality issues or incomplete records that may need cleaning before analysis.

```
SELECT
  SUM((started_at IS NULL)::int) AS missing_started,
  SUM((ended_at   IS NULL)::int) AS missing_ended,
```

```
  SUM((ride_id    IS NULL)::int) AS missing_ids
FROM divvy.fact_trips;
```

Table 8: 1 records

missing_started	missing_ended	missing_ids
0	0	0

I look for duplicate ride_id entries in the fact table. Since ride_id should be unique for each trip, this step ensures the integrity of the dataset and prevents double-counting in later analysis.

```
SELECT ride_id, COUNT(*) FROM divvy.fact_trips
GROUP BY ride_id HAVING COUNT(*) > 1;
```

Table 9: 0 records

ride_id	count

Check for negative or zero durations (should be 0 due to CHECK constraint)

```
SELECT COUNT(*) FROM divvy.fact_trips
WHERE ended_at < started_at OR ride_length_min < 0;
```

Table 10: 1 records

count
0

Refresh cadence

```
-- When new months arrive, re-run:
-- 1) load into staging,
-- 2) INSERT .. ON CONFLICT into dimensions & fact,
-- 3) REFRESH materialized views
REFRESH MATERIALIZED VIEW divvy.mv_monthly_summary;
```

The END