Homework 5

Khang Thai

2025-05-31

Question 1)

```
library(RMariaDB)
```

```
## Warning: package 'RMariaDB' was built under R version 4.3.3
```

```
library(DBI)
```

```
## Warning: package 'DBI' was built under R version 4.3.3
```

```
con <- dbConnect(RMariaDB::MariaDB(),
host = "relational.fel.cvut.cz",
port = 3306,
username = "guest",
password = "ctu-relational",
dbname = "SFScores"
)
```

(a)

```
dbGetQuery(con, "
  SELECT avg(avg_day_between)
  FROM (
    SELECT business_id,
      DATEDIFF(MAX(date), MIN(date)) / (COUNT(*) - 1) AS        avg_day_between
    FROM inspections
    WHERE type = 'Routine - Unscheduled'
    GROUP BY business_id
    HAVING COUNT(*) > 1
  ) AS avg_intervals;
")
```

```
##   avg(avg_day_between)
## 1             376.5305
```

(b)

```
monthly_score <- dbGetQuery(con, "
  SELECT MONTH(date) AS month, AVG(score) AS avg_score
  FROM inspections
  WHERE score IS NOT NULL
  GROUP BY month
```

```
")
monthly_score
```

```
##    month avg_score
## 1      1   90.8880
## 2      2   90.5037
## 3      3   91.0659
## 4      4   91.4891
## 5      5   90.3538
## 6      6   90.6790
## 7      7   89.7380
## 8      8   90.4024
## 9      9   90.0393
## 10    10   90.9969
## 11    11   91.1792
## 12    12   90.3708
```

```
cor(monthly_score$month, monthly_score$avg_score)
```

```
## [1] -0.2037836
```

(c)

```
dbGetQuery(con, "
  WITH inspection_score AS (
    SELECT business_id, date, score,
    LAG(score) OVER (PARTITION BY business_id ORDER BY date) AS previous_score
    FROM inspections
    WHERE score IS NOT NULL
  ), businesses_with_drop AS (
    SELECT DISTINCT business_id
    FROM inspection_score
    WHERE score < previous_score
  ), nondecrease_or_uninspected AS (
    SELECT COUNT(*) AS count_nondecrease
    FROM businesses
    LEFT JOIN businesses_with_drop ON businesses.business_id = businesses_with_drop.business_id
    WHERE businesses_with_drop.business_id IS NULL
  ), total_business_count AS (
    SELECT COUNT(*) AS total_count
    FROM businesses
  )
  SELECT n.count_nondecrease, t.total_count,
  ROUND(n.count_nondecrease * 100.0 / t.total_count, 2) AS percentage_nondecrease
  FROM nondecrease_or_uninspected n, total_business_count t;
")
```

```
##   count_nondecrease total_count percentage_nondecrease
## 1              3530        6358                  55.52
```

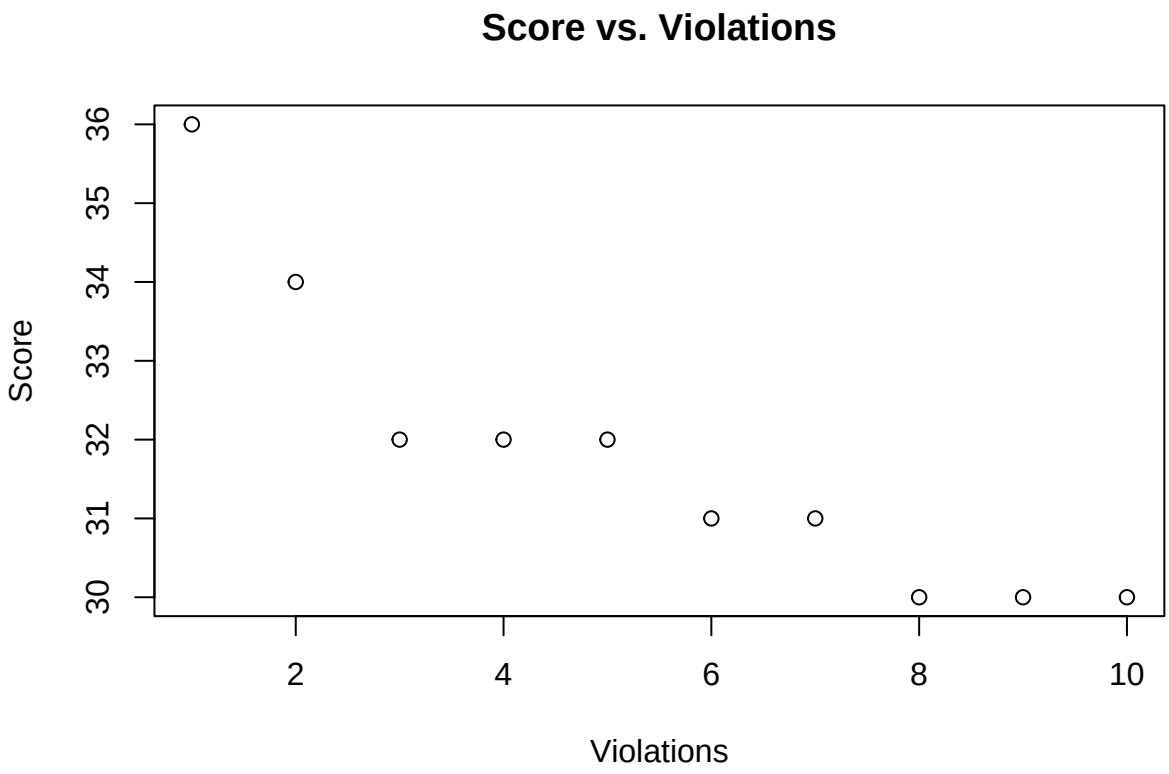
Question 2)

(a)

```
data <- dbGetQuery(con, "
  WITH ranked_inspections AS (
    SELECT i.business_id, i.date, i.score, COUNT(score) AS n_violations,
      SUM(CASE WHEN v.risk_category = 'Low Risk' THEN 1 ELSE 0 END) AS n_low_risk,
      SUM(CASE WHEN v.risk_category = 'Moderate Risk' THEN 1 ELSE 0 END) AS n_moderate_risk,
      SUM(CASE WHEN v.risk_category = 'High Risk' THEN 1 ELSE 0 END) AS n_high_risk,
      RANK() OVER (PARTITION BY i.business_id ORDER BY i.date DESC) as rn
  FROM inspections i
  JOIN violations v ON i.business_id = v.business_id
  GROUP BY i.business_id, i.date, i.business_id, i.score
  )
  SELECT business_id, date, n_violations, n_low_risk, n_moderate_risk, n_high_risk
  FROM ranked_inspections
  WHERE rn = 1
  ORDER BY n_violations DESC
  LIMIT 10;
")
data
```

```
##    business_id       date n_violations n_low_risk n_moderate_risk n_high_risk
## 1        70996 2016-06-21           36         18              13           5
## 2         7643 2016-09-27           34         19               8           7
## 3        76441 2016-05-17           32         20              10           2
## 4        60115 2016-08-09           32         16              11           5
## 5         7216 2016-09-24           32         17              12           3
## 6        18480 2016-09-20           31         16              10           5
## 7        25572 2016-09-08           31         11              13           7
## 8         3459 2016-01-13           30         15               9           6
## 9        17570 2016-08-29           30         17               5           8
## 10        7772 2016-07-07           30         19               8           3
```

```
plot(data$n_violations, data$score, main = "Score vs. Violations", xlab = "Violations", ylab = "Score")
```



Those with a higher score tend to have a lower violation risk than those who have a lower score.

(b)

```
dbGetQuery(con, "
  WITH violation_categorized AS (
    SELECT
      CASE
        WHEN LOWER(v.description) LIKE '%food%' THEN 'food'
        WHEN LOWER(v.description) LIKE '%plumbing%' THEN 'plumbing'
        WHEN LOWER(v.description) LIKE '%utensils%' THEN 'utensils'
        ELSE 'other'
      END AS category
    FROM violations v
    JOIN inspections i ON v.business_id = i.business_id
    WHERE i.score <= 80
  )
  SELECT category,
    COUNT(*) AS count,
    ROUND(100.0 * COUNT(*) / SUM(COUNT(*)) OVER (), 2) AS percentage
  FROM violation_categorized
  GROUP BY category;
")
```

```
##   category count percentage
## 1     food 10846      43.08
```

```
## 2    other 11879      47.18
## 3 plumbing   404       1.60
## 4 utensils  2047       8.13
```

(c)

```
dbGetQuery(con, "
  SELECT i.business_id, b.name, COUNT(*) AS total_inspection
  FROM inspections i
  JOIN businesses b ON i.business_id = b.business_id
  LEFT JOIN violations v ON i.business_id = v.business_id
  WHERE v.business_id IS NULL
  GROUP BY i.business_id, b.name
  ORDER BY total_inspection DESC
  LIMIT 10;
")
```

```
##    business_id                            name total_inspection
## 1        81682                     The Flame LLC               12
## 2        77850           Blue Fin Sushi & Lounge               10
## 3        80585                      High Treason                9
## 4        84470                Cream of Stonestown                8
## 5        79121                       Black Sands                8
## 6        75303 Second Act Marketplace and Events                8
## 7        71640                  Beluga Restaurant                8
## 8         3519                       Eclipse Cafe                8
## 9        76431                    Humphry Slocombe                8
## 10       75465                      The Pizza Shop                8
```

(d)

```
dbGetQuery(con, "
  WITH inspection_rank AS (
    SELECT b.postal_code, i.score,
      RANK() OVER (PARTITION BY b.postal_code ORDER BY i.score) AS rn,
      COUNT(*) OVER (PARTITION BY b.postal_code) AS total_count
    FROM businesses b
    JOIN inspections i on b.business_id = i.business_id
  ),
  at_least_30 AS (
    SELECT *
    FROM inspection_rank
    HAVING total_count >= 30
  ),
  median_score AS (
    SELECT postal_code, score AS median_score
    FROM at_least_30
    WHERE rn = (total_count + 1) / 2
  )
  SELECT postal_code, median_score
  FROM median_score
  ORDER BY median_score DESC
  LIMIT 10;
```

```
")
```

```
##   postal_code median_score
## 1       94108           75
## 2       94108           75
## 3       94108           75
## 4       94108           75
## 5       94108           75
## 6       94108           75
## 7       94108           75
## 8       94116           74
## 9       94116           74
```

Question 3)

```
library(reticulate)
```

```
## Warning: package 'reticulate' was built under R version 4.3.3
```

```
virtualenv_install("stats167_venv", packages = c("pandas", "matplotlib", "seaborn"))
```

```
## Using virtual environment "stats167_venv" ...
```

```
## + "C:/Users/kunfu/OneDrive/Documents/.virtualenvs/stats167_venv/Scripts/python.exe" -m pip install --
```

```
use_virtualenv("stats167_venv", required = TRUE)
```

(a)

```
import sqlite3
import pandas as pd

con = sqlite3.connect("rideshare.db")

query = """
  WITH hourly_count AS (
    SELECT sub_type,
      strftime('%Y-%m-%d', start_date) AS start_day,
      strftime('%H', start_date) AS start_hour,
      COUNT(*) AS n_hourly_trips
    FROM trips
    WHERE strftime('%Y', start_date) = '2012'
      AND strftime('%m', start_date) BETWEEN '05' AND '11'
    GROUP BY sub_type, start_day, start_hour
  ), avg_trips AS(
    SELECT sub_type, start_day, start_hour, n_hourly_trips,
      AVG(n_hourly_trips) OVER (
      PARTITION BY sub_type, start_hour
      ORDER BY start_day ROWS BETWEEN 27 PRECEDING
      AND CURRENT ROW ) AS rolling_avg_28d_trips
  FROM hourly_count
  ), ranked AS (
    SELECT *,
      RANK() OVER (PARTITION BY sub_type
      ORDER BY rolling_avg_28d_trips DESC) as rnk
```

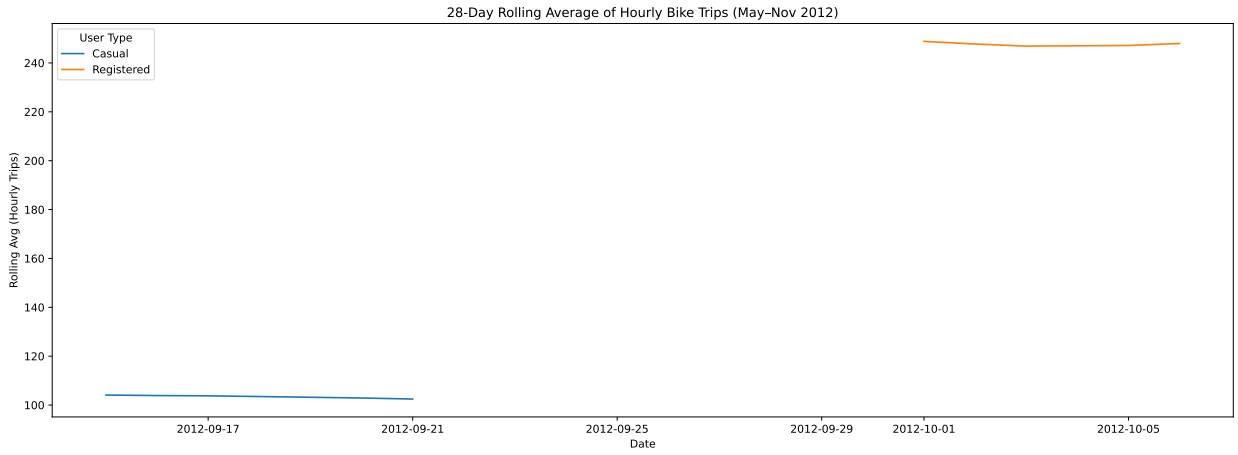
```
    FROM avg_trips
  )
  SELECT sub_type, start_day, start_hour, n_hourly_trips,        rolling_avg_28d_trips
  FROM ranked
  WHERE rnk <= 5;
"""
df = pd.read_sql_query(query, con)
print(df)
```

```
##       sub_type   start_day start_hour  n_hourly_trips  rolling_avg_28d_trips
## 0       Casual  2012-09-15         17             192             104.107143
## 1       Casual  2012-09-16         17             130             103.892857
## 2       Casual  2012-09-17         17              84             103.785714
## 3       Casual  2012-09-20         17              81             102.892857
## 4       Casual  2012-09-21         17              72             102.464286
## 5   Registered  2012-10-01         17             369             248.821429
## 6   Registered  2012-10-06         17             102             247.964286
## 7   Registered  2012-10-02         17             186             247.750000
## 8   Registered  2012-10-05         17             326             247.142857
## 9   Registered  2012-10-03         17             250             246.892857
```

```
df['start_day'] = pd.to_datetime(df['start_day'])
df['start_hour'] = df['start_hour'].astype(int)
```

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(16, 6))
sns.lineplot(data=df, x='start_day', y='rolling_avg_28d_trips', hue='sub_type')
plt.title('28-Day Rolling Average of Hourly Bike Trips (May-Nov 2012)')
plt.xlabel('Date')
plt.ylabel('Rolling Avg (Hourly Trips)')
plt.legend(title='User Type')
plt.tight_layout()
plt.show()
```



Registered users consistenly have a higher and more stable 28-day rolling average of bike trips compared to casual users. Casual users show more fluctuation.

(b)

```
con = sqlite3.connect("rideshare.db")
query = """
  WITH start_counts AS (
    SELECT
      start_station AS station,
      strftime('%Y-%m', start_date) AS month,
      COUNT(*) n_start_trips
    FROM trips
    GROUP BY station, month
  ), end_counts AS (
    SELECT
      end_station AS station,
      strftime('%Y-%m', end_date) AS month,
      COUNT(*) AS n_end_trips
    FROM trips
    GROUP BY station, month
  ), month_stats AS (
    SELECT
      s.station,
      s.month,
      s.n_start_trips,
      e.n_end_trips
    FROM start_counts s
    LEFT JOIN end_counts e ON s.station = e.station
      AND s.month = e.month
  ), avg_stats AS (
    SELECT station,
      AVG(n_start_trips) AS avg_monthly_start_trips,
      AVG(n_end_trips) AS avg_monthly_end_trips,
      CASE
        WHEN AVG(n_end_trips) = 0 THEN NULL
        ELSE CAST(AVG(n_start_trips) AS FLOAT) /
        AVG(n_end_trips)
      END AS avg_start_end_ratio
    FROM month_stats
    GROUP BY station
  ) SELECT
      station,
      avg_monthly_start_trips,
      avg_monthly_end_trips,
      avg_start_end_ratio
    FROM avg_stats
    WHERE avg_start_end_ratio IS NOT NULL
    ORDER BY avg_start_end_ratio DESC
    LIMIT 6;

"""

df = pd.read_sql_query(query, con)
print(df)
```

```
##    station  avg_monthly_start_trips  avg_monthly_end_trips  avg_start_end_ratio
```

```
## 0      123               53.666667             40.000000            1.341667
## 1       77              203.916667            156.916667            1.299522
## 2      144              352.000000            277.200000            1.269841
## 3      143              217.400000            171.400000            1.268378
## 4      119              208.000000            165.500000            1.256798
## 5      108               30.333333             25.000000            1.213333
```

Question 4)

(a)

NoSQL - Document database: Since the data is user-specific, it can be store as a document and NoSQL can handle large-scale, semi-structured data.

(b)

NoSQL - Key-Value: Key-Value stores are in-memory databased optimized for low-latency, requires fast read and lookup.

(c)

SQL

(d)

NoSQL - Wide-Column: The data is typically high volume and schema flexibility helps adapt as new data fields are added.

(e)

NoSQL - Document database: Requires handling multiple active sessions at once and requires fast writes.

Question 5)

Quorum consistency is when there is a series of nodes in a distributed system agreeing on a read or write operation. Quorum consistency is typically useful in balancing consistency and availability by allowing tunable consistency. The effect on consistency, availability, and latency is typically due either increasing or decreasing the size of the quorum.

When deciding on suitable consistency level, it is good to consider the criticality of the data consistency as well as the latency sensitivity. We also want to make sure that read and write quorums overlap so that it reads the latest writes.