Tim Stewart
B. S. in Computer Science

Task 1 Report for VDM1

Executive Summary

Our imaginary DVD Rental Company has done impressive business in recent times. There were 16,044 DVD rentals[1] of the 4,581 DVDs[2] in our stores' inventory during the 9-month period[3] reflected in our current `dvdrental` database, which produced a total of $61,321.04 in sales.[4] And while these aggregate figures sound impressive, in reality they are not very useful in providing us with specific and actionable business intelligence that we can use to better satisfy our customers and increase profits.

Therefore, in this report we will investigate which film categories (e.g., Action, Animation, Children, etc.) were responsible for the greatest dollar-value of rental sales on weekends during the 9 months of rental sales data available in `dvdrental`. This specific line of inquiry will guide our future DVD purchases for our inventory, because it will tell us about the best-selling film categories that people want to watch on weekends.

For our purposes, we define "the weekend" as from Friday 5 p.m. until Sunday 11:59 p.m. The `rental.rental_date` and `payment.payment_date` columns in our Company's `dvdrental` database are stored in UTC time (+00, or no time zone), so we will need to transform these timestamps into local timestamps based on each customer's city and country on file. To obtain the UTC offset information, a Python program written by myself was used to produce a supplementary PostgreSQL table named `utc_offsets_for_cities` that was loaded into the `dvdrental` database and used in some of the queries.

Please note that four additional separate files were also submitted with this report:
• **load-Tim-Stewart-functions-procedures-table.sql** — All the SQL code I wrote for this project
• **tz.py** — The Python program I wrote to look up time zones for the cities in the `city` table
• **cities_countries.txt** — The query results I used as input to my Python program
• **Tim Stewart, id 001476583, VDM1.pdf** — This report as a PDF in case the fonts break in the .docx

Introduction

The well-known sample database `dvdrental`[5] was analyzed using SQL queries and procedures to produce a report of how much money was made per film category from rental sales on weekends using

[1] `SELECT count(*) FROM rental;`
[2] `SELECT DISTINCT COUNT(*) FROM inventory;`
[3] Or 8 months 20 days, to be exact: `SELECT AGE( (SELECT rental_date FROM rental ORDER BY rental_date DESC LIMIT 1), (SELECT rental_date FROM rental ORDER BY rental_date ASC LIMIT 1) );`
[4] `SELECT SUM(amount) FROM payment;`
[5] The `dvdrental` sample database is sometimes also known by the name Pagila in the PostgreSQL world and by the name Sakila in the MySQL world. The Sakila database is the original version of the database, and it was

all available rental transaction data. The weekend was defined as starting at Friday 5 p.m. (i.e., 17:00) in each customer's local time zone and continuing through all day Saturday and all day Sunday. Since the `dvdrental` database stores the timestamp of rental transactions without any time zone (i.e., in UTC time, or +00 time), a transformation needed to be applied to convert each rental timestamp into the local time zone of the customer's city.

A. 1–3. Data Sources and Data Product

The data that this report is based on was provided in the form of the `dvdrental` sample database which was pre-installed for my use in an online virtual lab environment. More specifically, I used the following tables from the `dvdrental` database: `rental`, `inventory`, `film`, `film_category`, `category`, `customer`, `address`, `city`, and `country`. Although many fields from these tables were accessed by queries and procedures to produce the detailed section of this report, please note that the summary table shows just the `category.name` and aggregated `payment.amount` fields.

The raw data was aggregated and partly transformed to produce a summary report of the amount of weekend (in local time) rental sales broken out by film category.

A. 4. Transformation of Transaction Timestamps to Local Time Zones

The `payment.payment_date` and `rental.rental_date` columns of the `dvdrental` database were created using PostgreSQL's data type `timestamp without time zone not null`.[6] Not using time zones for timestamps stored in the database is an excellent design choice. The possible alternative of storing timestamps with a local time zone creates the risk of corrupt data analysis if timestamps are improperly converted when they are compared, or if timestamps aren't converted at all and analysts unwittingly compare them anyway.

But while it's wise to store the rental transaction timestamps without time zones, that means that any analysis involving the customer's purchasing habits in local time will require a transformation of the timestamps into ones that express the customer's local time.

Therefore, a transformation function will be applied to the `payment.payment_date` field to convert these timestamp values into the customer's local time zone so that rental sales only during each customer's local weekend timeframe will be aggregated. Since time zone and UTC offset information are not provided in the `dvdrental` database, I wrote a Python program (see Appendix A) that accepts the list of cities and countries from `dvdrental`'s city and country tables as input and looks up the latitude and longitude coordinates for each city and then looks up the UTC offsets for the latitude and longitude coordinates using the Python packages geopy.geocoders[7] and timezonefinders.[8] The resulting UTC

published by the MySQL company in March 2006 (see https://dev.mysql.com/doc/sakila/en/sakila-history.html, accessed April 28, 2022). Pagila is an adaptation of the Sakila database schema and data to the PostgreSQL idiom that was published in 2006 shortly after Sakila's release (see https://wiki.postgresql.org/wiki/Sample_Databases, accessed April 28, 2022).

[6] I confirmed this by examining the output of the commands `pg_dump.exe -st rental dvdrental` and `pg_dump.exe -st payment dvdrental`, which I executed from the virtual lab's PowerShell prompt.

[7] https://github.com/geopy/geopy, accessed April 28, 2022.

[8] https://github.com/jannikmi/timezonefinder, accessed April 28, 2022.

offsets were put into a SQL file (see Appendix B) which was loaded into the `dvdrental` database in my virtual lab environment prior to running my queries and procedures.

A. 5. Business Uses of the Detailed and Summary Reports

The `detailed_table` is an intermediate step between the raw data that we start with and the `summary_table` which is the end goal of this report. In that sense, the main purpose of the `detailed_table` is to support the computation of the resulting `summary_table`. However, the `detailed_table` is also an excellent starting point for almost any data analysis that involves the film categories and local date and time of the rental transactions. For example, the data in the `detailed_table` could be used to explore which film categories are popular at different times of the week. By contrast, the `summary_table` very narrowly answers just the question of which film categories made the most money in rentals during the weekend timeframe. The advantage of the `summary_table` data is that it has a focus for a very specific business question, without any other columns or transformations that might serve as a distraction.

Another advantage of separating out the `detailed_table` from the `summary_table` is that the queries that populate the two tables could possibly be run on two different computer servers or at two different times of day because the SQL code that populates the `detailed_table` is much more intensive and demanding than the code that populates the `summary_table`. The `detailed_table` executes a query across six `JOIN`ed tables and then performs a complex timestamp transformation function on each row of the raw data it collects. By comparison, the `summary_table` performs a simple `SUM()` of amounts by film category and sorts 16 categories in `DESC`ending `ORDER`. The imaginary DVD Rental Company might take advantage of the difference in computing power needed for these two tables by, for example, calculating the data for the `detailed_table` at night when the database servers are unused and available for giant queries and jobs. Or the data for the `detailed_table` might be calculated using a powerful cloud-based server so that the queries can be calculated faster. By comparison, the data for the `summary_table` could be easily calculated on a less-powerful server or perhaps even on the database user's own workstation.

A. 6. Frequency of Refreshing the Reports

Since this report analyzes weekend sales data, the natural conclusion would be that the detailed and summary reports should be refreshed on a weekly basis in order to capture the most recent weekend's worth of rental sales data. In practical terms, it is my recommendation that the detailed and summary reports be refreshed weekly on Tuesdays so that there is a short amount of time for the store locations to enter their weekend rental sales data into the database but so that there is still most of the week left for the refreshed reports to support business decisions that need to be made by the end of the week.

B. Creating the Tables for the Reports

The SQL code below is responsible for creating the `detailed_table` and `summary_table` that are used to create the business reports. The `detailed_table` will contain raw data extracted from the tables `category`, `rental`, and `payment`. As noted earlier in part A section 4, the `rental.rental_date` UTC timestamps in the raw data will be transformed by the function `get_local_timestamp()` into timestamps that are localized to the customer's city.

```
/*
```

```
   Create `detailed_table`
*/
CREATE TABLE detailed_table (
   detailed_id SERIAL PRIMARY KEY,
   category TEXT NOT NULL,
   rental_date_local TIMESTAMP WITHOUT TIME ZONE NOT NULL,
   rental_id INTEGER UNIQUE NOT NULL,
   amount NUMERIC(8,2)
);
ALTER TABLE ONLY public.detailed_table
   ADD CONSTRAINT detailed_table_rental_id_fkey
   FOREIGN KEY (rental_id)
   REFERENCES public.rental(rental_id)
   ON UPDATE CASCADE
   ON DELETE RESTRICT;

/*
   Create `summary_table`
*/
CREATE TABLE summary_table (
   category TEXT PRIMARY KEY NOT NULL,
   rental_sales numeric(8,2) NOT NULL
);
```

C. SQL Code for Extraction from `dvdrental` and Loading into `detailed_table`

The SQL code below extracts raw data from the `dvdrental` database and performs a transformation on the `rental.rental_date` values so they reflect the local time of the rental customer's city.

```
/*
   Extract raw data from `dvdrental` to `detailed_table`, using the
   "ON CONFLICT ... DO NOTHING" technique to avoid duplicating rows
*/
INSERT INTO detailed_table (category, rental_date_local, rental_id, amount)
   (SELECT
      category.name AS "Category",
      get_local_timestamp(rental.rental_id) AS "Rental Date (Local)",
      rental.rental_id AS "Rental ID",
      payment.amount AS "Amount"
   FROM payment
      JOIN rental USING (rental_id)
      JOIN inventory USING (inventory_id)
      JOIN film USING (film_id)
      JOIN film_category USING (film_id)
      JOIN category USING (category_id)
   ) ON CONFLICT (rental_id) DO NOTHING;
```

The extraction code shown above uses the `ON CONFLICT ... DO NOTHING` qualifier to avoid duplicating rows in `detailed_table`. In other words, if `detailed_table` already contains a row with the same `rental_id` as a relation in the raw data, it won't `INSERT`ed again.

The project requirements state that the SQL code should "verify the data's accuracy." To meet this requirement, I have made sure that my `SELECT` query uses six `JOIN`ed tables. How does `JOIN`ing help verify the data's accuracy? Five of these six tables have foreign key constraints (i.e., `FK CONSTRAINT`), which require the data values of `category_id`, `film_id`, `inventory_id`, `rental_id`, and `payment_id` to be valid references. If by some accident an erroneous integer is used in place of any valid id numbers during the query, the query will automatically fail due to a foreign key constraint violation, giving the database user or database administrator an opportunity to investigate and correct the problem.

D. SQL Code for Transforming UTC Timestamps to Local Time Zones

As noted earlier in part A section 4, storing timestamps in the database without time zones is a best practice that prevents a wide variety of time zone conversion errors and problems. But this also means that when we want to analyze customer rental habits in the context of the customer's local time zone, we have to perform a transformation.

There were two options available for coding this transformation. We could use the time zone for the city that is linked to the store's address, or we could use the time zone for the city that is linked to the customer's address. Since the imaginary DVD Rental Company wants to be as customer-focused as possible, I decided to obtain the local time zone information based on the customer's address.

The function shown below queries a table named `utc_offsets_for_cities`, which was not provided in the pre-installed `dvdrental` database in the online lab environment. As noted earlier in part A section 4, I created this table and added it to my instance of the `dvdrental` database in order to complete this project accurately using the local time zone of each customer. I wrote a Python program (see Appendix A) that looked up the time zones of each city listed in `dvdrental`'s `city` table and then looked up the numeric offset from UTC time that was associated with each time zone. I made sure that the Python program outputted the results in a format that made it easy to create a PostgreSQL table. Finally I added the new table to my instance of the `dvdrental` database. More information about my process of obtaining the time zones and UTC offset information can be found in Appendix A.

```
/*
   Function: get_local_timestamp(the_rental_id INTEGER)
*/
CREATE OR REPLACE FUNCTION get_local_timestamp(the_rental_id INTEGER)
RETURNS TIMESTAMP AS $$
DECLARE
   the_customer_city_id INTEGER;
   the_utc_offset_in_hours REAL;
   the_utc_offset_in_seconds INTEGER;
   the_rental_timestamp TIMESTAMP;
   the_local_timestamp TIMESTAMP;
BEGIN
   the_customer_city_id :=
      (SELECT address.city_id
         FROM rental
            JOIN customer USING (customer_id)
            JOIN address USING (address_id)
         WHERE rental.rental_id = the_rental_id);
   the_utc_offset_in_hours :=
      (SELECT utc_offset
         FROM utc_offsets_for_cities
         WHERE city_id = the_customer_city_id);
   the_utc_offset_in_seconds :=
      (SELECT CAST(the_utc_offset_in_hours * 3600 AS INTEGER));
   the_rental_timestamp :=
      (SELECT rental.rental_date
         FROM rental
         WHERE rental.rental_id = the_rental_id);   /* continued on next page */
   the_local_timestamp :=
      (SELECT (the_rental_timestamp + (INTERVAL '1 sec' * the_utc_offset_in_seconds)));
   RETURN the_local_timestamp;
END; $$ LANGUAGE 'plpgsql';
```

E. Creating a Trigger on the `detailed_table` Which Updates the `summary_table`

The code for this SQL trigger updates the data in `summary_table` whenever an `INSERT` statement occurs on `detailed_table`. Note that it would of course also be possible to create additional `TRIGGER`s that update the data in `summary_table` whenever rows in `detailed_table` are `DELETE`d or `UPDATE`d, in case deletions and updates at some point become a routine activity on `detailed_table`.

```
/*
   Update `summary_table`
*/
CREATE OR REPLACE FUNCTION summary_table_update()
RETURNS trigger AS $$
BEGIN
   DELETE FROM summary_table;
   INSERT INTO summary_table (category, rental_sales)
      SELECT
         category,
         SUM(amount)
      FROM
         detailed_table
      WHERE
         is_weekend(rental_date_local) = true
      GROUP BY category
      ORDER BY SUM(amount) DESC;
   RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER update_summary_table
   AFTER INSERT ON detailed_table
   FOR EACH STATEMENT
   EXECUTE PROCEDURE summary_table_update();
```

For the sake of completeness, what follows is the PostgreSQL code I wrote for the `is_weekend()` function that was used in the `WHERE` clause of the `summary_table_update()` function shown immediately above. This `is_weekend()` function encodes the business logic about the definition of weekend discussed earlier in the Executive Summary and in the Introduction. As a courtesy reminder, for the purposes of our data analysis “the weekend” is defined as “Friday after 5 p.m. and all day Saturday and all day Sunday, all in the customer’s local time.” Note that in `is_weekend()`, `timestamp`s for Fridays are cast to the `time` data type for direct comparison to the time value '17:00:00' (i.e., 5 p.m.).

```
CREATE OR REPLACE FUNCTION is_weekend(the_ts TIMESTAMP)
RETURNS BOOLEAN AS $$
BEGIN
   IF
      (day_of_week(the_ts) = 5 AND the_ts::time >= '17:00:00'::time)
      OR day_of_week(the_ts) = 6
      OR day_of_week(the_ts) = 0
   THEN
      RETURN true;
   ELSE
      RETURN false;   /* continued on next page */
   END IF;
END; $$ LANGUAGE 'plpgsql';
```

Likewise for the sake of completeness, here is the code for the `day_of_week()` function I wrote as a utility function for use in the `IF` clause of the preceding `is_weekend()` function.

```
CREATE OR REPLACE FUNCTION day_of_week(the_ts TIMESTAMP)
RETURNS INTEGER AS $$
```

```
DECLARE
   the_dow INTEGER;
BEGIN
   SELECT EXTRACT(DOW FROM the_ts) INTO the_dow;
   RETURN the_dow;
END;
$$ LANGUAGE 'plpgsql';
```

F. Using a Scheduled Procedure to Refresh the Detailed and Summary Tables

The data in the detailed and summary reports is only as good as it is fresh, meaning that if the data has become stale then it becomes less useful as a guide to which film categories customers are currently interested in renting on the weekends. As noted earlier in part A section 6, it makes sense for the weekend rental sales data to be updated weekly, whenever there is a new weekend's worth of data to incorporate into our analysis. The following SQL code refreshes the detailed and summary tables by executing the Extraction, Loading, and Transformation (ETL) activities described in parts C and D above.

```
CREATE OR REPLACE PROCEDURE refresh_detailed_and_summary_tables()
/*
   Procedure: `refresh_detailed_and_summary_tables()`
   Purpose:   Refresh the detailed_table and summary_table to include recent
              weekend rental sales
   Author:    Tim Stewart
   Frequency: call this procedure weekly on Tuesday mornings
*/
LANGUAGE 'plpgsql' AS $$
BEGIN
/*  Clear contents of `detailed_table` */
DELETE FROM detailed_table;
/*  Clear contents of `summary_table` */
DELETE FROM summary_table;
/*  Load raw data into `detailed_table`. Note that because we already have a
   trigger installed on `detailed_table`, we don't have to explicitly refresh
   the `summary_table`. The `INSERT INTO detailed_table` statement will
   automatically invoke the trigger `update_summary_table` that is shown in
   Part E of the report.
*/
INSERT INTO detailed_table (category, rental_date_local, rental_id, amount)
   (SELECT
      category.name AS "Category",
      get_local_timestamp(rental.rental_id) AS "Rental Date (Local)",
      rental.rental_id AS "Rental ID",
      payment.amount AS "Amount"
   FROM payment
      JOIN rental USING (rental_id)
      JOIN inventory USING (inventory_id)
      JOIN film USING (film_id)
      JOIN film_category USING (film_id)
      JOIN category USING (category_id)
   ) ON CONFLICT (rental_id) DO NOTHING;
END; $$;
```

To ensure data freshness, the `refresh_detailed_and_summary_tables()` procedure should be called regularly. As noted earlier in part A section 6, my recommendation is for the imaginary DVD Rental Company to refresh the weekend rental sales data on a weekly basis, preferably as early in the week as possible. To provide some buffer time for the raw weekend rental sales figures to be loaded into the database, I recommend for a stored procedure such as this one to be run on Tuesday morning at the beginning of the business day.

A process or plan is only as strong as its weakest link, so executing this stored procedure on a weekly basis shouldn't be in the hands of a particular employee or department to always remember to manually run it. A much more reliable business process is automating the procedure so it happens at the same time and in the same way every week without human error. All the major operating systems have the functionality to regularly execute a command on a schedule. On Linux and other Unix-like systems, the `cron` utility is a well-known and reliable means of automating recurring tasks. For example, a `crontab` line for running this stored procedure every Tuesday at 9 a.m. might look as follows:

```
0 9 * * TUE /usr/bin/psql -U postgres -d dvdrental -c "call refresh_detailed_and_summary_tables();"
```

The Microsoft Windows operating system provides a similar functionality to run specified commands at certain times on certain days via its built-in Task Scheduler application.

G. Panopto Video Recording

A personal video presentation forms part G of this report. It can be accessed using the link that was submitted with this report.

Conclusion

In this report we have seen how the raw data in `dvdrental` can be extracted, loaded, and transformed in order to answer a narrowly defined business question such as which film categories produce the most sales rentals on the weekends in customers' local time zones. Producing the summary data entailed the use of several `JOIN`ed tables and a complex transformation. Additionally I chose to write a Python program to generate a supplementary table of UTC offsets of customers' local time zones.

See below for the final output of the `summary_table` (as shown in two columns). Our imaginary DVD Rental Company should stock up on Sports and Sci-Fi DVDs and stay away from Travel and Music DVDs if they want to give the customers what they want and boost those future weekend rental sales!

```
dvdrental=# select * from summary_table;

  category   | rental_sales
-------------+--------------
 Sports      |      1692.36
 Sci-Fi      |      1459.65
 Drama       |      1392.63
 Animation   |      1353.69
 Action      |      1339.65
 Family      |      1338.49
 New         |      1335.22
 Games       |      1301.27
 Documentary |      1224.97
 Foreign     |      1220.05
 Comedy      |      1218.46
 Classics    |      1164.06
 Children    |      1146.09
 Horror      |      1076.51
 Travel      |      1049.54
 Music       |       971.65
(16 rows)
```

Appendix A. Python Program to Look Up Local Time Zones for Customers' Cities

I had to think outside the box to figure out how to find the UTC offsets for the 600 cities[9] in `dvdrental`'s `city` table. Since PostgreSQL does not have a built-in function for looking up UTC offsets or time zones for geographic locations, I had to make use of an accessory program in the form of a Python program I wrote expressly for this project that used the two well-known Python packages geopy and timezonefinder.

My first step was to get the list of cities and countries from `dvdrental`. Using a simple query,[10] I obtained a text file with all the cities and their corresponding countries. The first 10 and last 10 lines of the resulting text file are shown below. (Note that I later on deleted the column headers, which I have highlighted in yellow here, so it would be simpler for Python to parse just the rows of actual data.)

```
 city_id |            city             | country_id |                country
---------+-----------------------------+------------+-----------------------------------
       1 | A Corua (La Corua)          |         87 | Spain
       2 | Abha                        |         82 | Saudi Arabia
       3 | Abu Dhabi                   |        101 | United Arab Emirates
       4 | Acua                        |         60 | Mexico
       5 | Adana                       |         97 | Turkey
       6 | Addis Abeba                 |         31 | Ethiopia
       7 | Aden                        |        107 | Yemen
       8 | Adoni                       |         44 | India
       9 | Ahmadnagar                  |         44 | India
      10 | Akishima                    |         50 | Japan
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     590 | Yuncheng                    |         23 | China
     591 | Yuzhou                      |         23 | China
     592 | Zalantun                    |         23 | China
     593 | Zanzibar                    |         93 | Tanzania
     594 | Zaoyang                     |         23 | China
     595 | Zapopan                     |         60 | Mexico
     596 | Zaria                       |         69 | Nigeria
     597 | Zeleznogorsk                |         80 | Russian Federation
     598 | Zhezqazghan                 |         51 | Kazakstan
     599 | Zhoushan                    |         23 | China
     600 | Ziguinchor                  |         83 | Senegal
(600 rows)
```

On the next page the complete listing of my Python program is given. For ease of reading, the code is displayed using colorful syntax highlighting which makes the variables, punctuation, and Python keywords easier to distinguish.

[9] `SELECT COUNT(*) FROM city;`

[10] `SELECT ci.city_id, ci.city, co.country_id, co.country FROM city AS ci JOIN country AS co USING (country_id);`

```python
from geopy.geocoders import Nominatim
import datetime
import pytz
import timezonefinder

def pretty_print_offset(utc_offset_str):
    if utc_offset_str[0] == '-':
        sign = -1.0
    else:
        sign = 1.0
    hours = float(utc_offset_str[1:3])
    if utc_offset_str[-2:] == '30':
        half = 0.5
    else:
        half = 0.0
    if half == 0.5:
        return float(sign * (hours + half))
    else:
        return int(sign * hours)

if __name__ == "__main__":
    with open("cities_countries.txt", mode="r", encoding="utf-8") as f:
        cities_countries = f.read().splitlines()

    cities_countries_tuples = []
    for each_c_c in cities_countries:
        cities_countries_tuples.append([datum.strip() for datum in each_c_c.split("|")])

    geolocator = Nominatim(user_agent="Tim Stewart")
    tf = timezonefinder.TimezoneFinder()
    for x in cities_countries_tuples:
        geo_location = ""
        try:
            geo_location = geolocator.geocode(f"{x[1]}, {x[3]}")
        except Exception as e:
            print(f"Error: {e}")

        if geo_location:
            tz_name = tf.timezone_at(lng=geo_location.longitude, lat=geo_location.latitude)
            utc_offset = datetime.datetime.now(pytz.timezone(tz_name)).strftime('%z')
            if x[0] == 600:
                punc = ";"
            else:
                punc = ","
            print(f"({x[0]}, '{x[1]}', {pretty_print_offset(utc_offset)}){punc}")
        else:
            print(f"geo_location couldn't be found for city with city_id = {x[0]}")

    exit()
```

But when I executed the program, I noticed a problem! I discovered that dozens of city names in the `city` table in the `dvdrental` database are missing their special characters. Apparently at some point in the past, the database files for the `dvdrental` database were saved with an incorrect text encoding that caused all characters with special accent marks to disappear from the names of the cities. For example, the true name of the city called 'A Corua (La Corua)' in the `city` table is actually supposed to be 'A Coruña (La Coruña)'. Likewise the city 'Acua' is actually supposed to be named 'Acuña'. And the city 'Balaiha' is actually supposed to be named 'Balašiha'. As I investigated deeper into this problem, I discovered that a wide variety of special characters, including *á, â, ç, é, İ, í, ñ, ó, ö,* and *š*, had vanished from the names of these international cities, resulting in corrupted city names for 65 of the 600 cities in the table. Furthermore, this encoding corruption didn’t just happen in the `dvdrental` database: Even the official Sakila database that is available for download from mysql.com[11] had the same text encoding corruption in the city names. In the end I did have to do some manual research using Wikipedia and online atlases to track down the correct names and time zones for those 65 cities that appear in misspelled and corrupted form in dvdrental's city table, since my Python program wasn't able to process the misspelled and corrupted city names. On another positive note, I and a friend of mine filed a bug report with the MySQL company, and they told us they will have these corrupted city names fixed in an upcoming version of the Sakila sample database.[12]

Shown below are the first 10 and last 10 lines of the output of the Python program when I was finally able to obtain time zones and offsets for all the cities (even the ones that were missing characters!).

```
(1, 'A Corua (La Corua)', 2),
(2, 'Abha', 3),
(3, 'Abu Dhabi', 4),
(4, 'Acua', -5),
(5, 'Adana', 3),
(6, 'Addis Abeba', 3),
(7, 'Aden', 3),
(8, 'Adoni', 5.5),
(9, 'Ahmadnagar', 5.5),
(10, 'Akishima', 9),
. . . . . . . . . . . . . . . . .
(590, 'Yuncheng', 8),
(591, 'Yuzhou', 8),
(592, 'Zalantun', 8),
(593, 'Zanzibar', 3),
(594, 'Zaoyang', 8),
(595, 'Zapopan', -5),
(596, 'Zaria', 1),
(597, 'Zeleznogorsk', 7),
(598, 'Zhezqazghan', 6),
(599, 'Zhoushan', 8),
(600, 'Ziguinchor', 0);
```

At this point it was a simple matter to add CREATE TABLE and INSERT INTO statements before the list of cities and their UTC offsets and execute the combined code while I was in the dvdrental database in my virtual lab environment to create the utc_offsets_for_cities table I needed for my queries. For the complete code listing of the CREATE TABLE, INSERT INTO, and VALUES statements that I used to create the special utc_offsets_for_cities table, please see Appendix B.

[11] https://dev.mysql.com/doc/index-other.html, accessed April 29, 2022.

[12] With the help of my friend Vladislav Sokol, we filed a bug report with the actual MySQL company about the text encoding corruption that occurred in the Sakila database that the dvdrental database is based on. One of MySQL's software developers, Philip Olson, confirmed to us that this is the first time anybody has reported this issue to them and that they will fix it in an upcoming release of the Sakila sample database. You can see my and Vladislav's actual bug report here, including MySQL's responses: https://bugs.mysql.com/bug.php?id=106951, accessed April 25, 2022. To add to the excitement over finding a bug in a widely used database, the corrupted city names bug is just one of *two* bugs I discovered while working on this dvdrental project. The second bug, which was also swiftly filed with the MySQL company, was regarding some payment rows that are missing rental_id foreign keys, resulting in $9.95 worth of transactions that are not associated with any rental. This bug report can also be found online: https://bugs.mysql.com/bug.php?id=107158, accessed April 29, 2022. It gives me a warm feeling to help our wider database community by contributing bug reports to improve the quality of the software and learning tools we use. Hopefully future students who use dvdrental will have an improve database to use!

Appendix B. Code Listing to Create and Populate the Supplementary Table `utc_offsets_for_cities`

```
\connect dvdrental

CREATE TABLE utc_offsets_for_cities (
   city_id SMALLINT PRIMARY KEY NOT NULL,
   city_name TEXT NOT NULL,
   utc_offset REAL NOT NULL
);

INSERT INTO utc_offsets_for_cities VALUES
(1, 'A Corua (La Corua)', 2),
(2, 'Abha', 3),
(3, 'Abu Dhabi', 4),
(4, 'Acua', -5),
(5, 'Adana', 3),
(6, 'Addis Abeba', 3),
(7, 'Aden', 3),
(8, 'Adoni', 5.5),
(9, 'Ahmadnagar', 5.5),
(10, 'Akishima', 9),
(11, 'Akron', -4),
(12, 'al-Ayn', 4),
(13, 'al-Hawiya', 3),
(14, 'al-Manama', 3),
(15, 'al-Qadarif', 2),
(16, 'al-Qatif', 3),
(17, 'Alessandria', 2),
(18, 'Allappuzha (Alleppey)', 5.5),
(19, 'Allende', -5),
(20, 'Almirante Brown', -3),
(21, 'Alvorada', -3),
(22, 'Ambattur', 5.5),
(23, 'Amersfoort', 2),
(24, 'Amroha', 5.5),
(25, 'Angra dos Reis', -3),
(26, 'Anpolis', -3),
(27, 'Antofagasta', -4),
(28, 'Aparecida de Goinia', -3),
(29, 'Apeldoorn', 2),
(30, 'Araatuba', -3),
(31, 'Arak', 4.5),
(32, 'Arecibo', -4),
(33, 'Arlington', -5),
(34, 'Ashdod', 3),
(35, 'Ashgabat', 5),
(36, 'Ashqelon', 3),
(37, 'Asuncin', -4),
(38, 'Athenai', 3),
(39, 'Atinsk', 7),
(40, 'Atlixco', -5),
(41, 'Augusta-Richmond County', -4),
(42, 'Aurora', -5),
(43, 'Avellaneda', -3),
(44, 'Bag', -3),
(45, 'Baha Blanca', -3),
(46, 'Baicheng', 8),
(47, 'Baiyin', 8),
(48, 'Baku', 4),
(49, 'Balaiha', 3),
(50, 'Balikesir', 3),
(51, 'Balurghat', 5.5),
(52, 'Bamenda', 1),
(53, 'Bandar Seri Begawan', 8),
(54, 'Banjul', 0),
(55, 'Barcelona', -4),
(56, 'Basel', 2),
(57, 'Bat Yam', 3),
(58, 'Batman', 3),
(59, 'Batna', 1),
(60, 'Battambang', 7),
(61, 'Baybay', 8),
(62, 'Bayugan', 8),
(63, 'Bchar', 1),
(64, 'Beira', 2),
(65, 'Bellevue', -7),
(66, 'Belm', -3),
(67, 'Benguela', 1),
(68, 'Beni-Mellal', 0),
(69, 'Benin City', 1),
(70, 'Bergamo', 2),
(71, 'Berhampore (Baharampur)', 5.5),
(72, 'Bern', 2),
(73, 'Bhavnagar', 5.5),
(74, 'Bhilwara', 5.5),
(75, 'Bhimavaram', 5.5),
(76, 'Bhopal', 5.5),
(77, 'Bhusawal', 5.5),
(78, 'Bijapur', 5.5),
(79, 'Bilbays', 2),
(80, 'Binzhou', 8),
(81, 'Birgunj', 104),
(82, 'Bislig', 8),
(83, 'Blumenau', -3),
(84, 'Boa Vista', -4),
(85, 'Boksburg', 2),
(86, 'Botosani', 3),
(87, 'Botshabelo', 2),
(88, 'Bradford', 1),
(89, 'Braslia', -4),
(90, 'Bratislava', 2),
(91, 'Brescia', 2),
(92, 'Brest', 2),
(93, 'Brindisi', 2),
(94, 'Brockton', -4),
(95, 'Bucuresti', 3),
(96, 'Buenaventura', -5),
(97, 'Bydgoszcz', 2),
(98, 'Cabuyao', 8),
(99, 'Callao', -5),
(100, 'Cam Ranh', 7),
(101, 'Cape Coral', -4),
(102, 'Caracas', -4),
(103, 'Carmen', -5),
(104, 'Cavite', 8),
(105, 'Cayenne', -3),
(106, 'Celaya', -5),
(107, 'Chandrapur', 5.5),
(108, 'Changhwa', 8),
(109, 'Changzhou', 8),
(110, 'Chapra', 5.5),
(111, 'Charlotte Amalie', -4),
(112, 'Chatsworth', 2),
(113, 'Cheju', 9),
(114, 'Chiayi', 8),
(115, 'Chisinau', 3),
(116, 'Chungho', 8),
(117, 'Cianjur', 7),
(118, 'Ciomas', 7),
(119, 'Ciparay', 7),
(120, 'Citrus Heights', -7),
(121, 'Citt del Vaticano', 2),
(122, 'Ciudad del Este', -4),
(123, 'Clarksville', -5),
(124, 'Coacalco de Berriozbal', -5),
(125, 'Coatzacoalcos', -5),
(126, 'Compton', -7),
(127, 'Coquimbo', -4),
(128, 'Crdoba', -3),
(129, 'Cuauhtmoc', -5),
(130, 'Cuautla', -5),
(131, 'Cuernavaca', -5),
(132, 'Cuman', -4),
(133, 'Czestochowa', 2),
(134, 'Dadu', 5),
(135, 'Dallas', -5),
(136, 'Datong', 8),
(137, 'Daugavpils', 3),
(138, 'Davao', 8),
(139, 'Daxian', 8),
(140, 'Dayton', -4),
(141, 'Deba Habe', 1),
(142, 'Denizli', 3),
(143, 'Dhaka', 6),
(144, 'Dhule (Dhulia)', 5.5),
(145, 'Dongying', 8),
(146, 'Donostia-San Sebastin', 2),
(147, 'Dos Quebradas', -5),
(148, 'Duisburg', 2),
(149, 'Dundee', 1),
(150, 'Dzerzinsk', 3),
(151, 'Ede', 2),
(152, 'Effon-Alaiye', 1),
(153, 'El Alto', -4),
(154, 'El Fuerte', -6),
(155, 'El Monte', -7),
(156, 'Elista', 3),
(157, 'Emeishan', 8),
(158, 'Emmen', 2),
(159, 'Enshi', 8),
(160, 'Erlangen', 2),
(161, 'Escobar', -3),
(162, 'Esfahan', 4.5),
(163, 'Eskisehir', 3),
(164, 'Etawah', 5.5),
(165, 'Ezeiza', -3),
(166, 'Ezhou', 8),
(167, 'Faaa', -10),
(168, 'Fengshan', 8),
(169, 'Firozabad', 5.5),
(170, 'Florencia', -5),
(171, 'Fontana', -7),
(172, 'Fukuyama', 9),
(173, 'Funafuti', 12),
(174, 'Fuyu', 8),
(175, 'Fuzhou', 8),
(176, 'Gandhinagar', 5.5),
(177, 'Garden Grove', -7),
(178, 'Garland', -5),
(179, 'Gatineau', -4),
(180, 'Gaziantep', 3),
(181, 'Gijn', 2),
(182, 'Gingoog', 8),
(183, 'Goinia', -3),
(184, 'Gorontalo', 8),
(185, 'Grand Prairie', -5),
(186, 'Graz', 2),
(187, 'Greensboro', -4),
(188, 'Guadalajara', -5),
(189, 'Guaruj', -3),
(190, 'guas Lindas de Gois', -3),
(191, 'Gulbarga', 5.5),
(192, 'Hagonoy', 8),
(193, 'Haining', 8),
(194, 'Haiphong', 7),
(195, 'Haldia', 5.5),
(196, 'Halifax', -3),
(197, 'Halisahar', 5.5),
(198, 'Halle/Saale', 2),
(199, 'Hami', 8),
(200, 'Hamilton', 12),
(201, 'Hanoi', 7),
(202, 'Hidalgo', -5),
(203, 'Higashiosaka', 9),
(204, 'Hino', 9),
(205, 'Hiroshima', 9),
(206, 'Hodeida', 3),
(207, 'Hohhot', 8),
(208, 'Hoshiarpur', 5.5),
(209, 'Hsichuh', 8),
(210, 'Huaian', 8),
(211, 'Hubli-Dharwad', 5.5),
(212, 'Huejutla de Reyes', -5),
(213, 'Huixquilucan', -5),
(214, 'Hunuco', -5),
(215, 'Ibirit', -3),
(216, 'Idfu', 2),
(217, 'Ife', 1),
(218, 'Ikerre', 1),
(219, 'Iligan', 8),
(220, 'Ilorin', 1),
(221, 'Imus', 8),
(222, 'Inegl', 3),
(223, 'Ipoh', 8),
(224, 'Isesaki', 9),
(225, 'Ivanovo', 3),
(226, 'Iwaki', 9),
(227, 'Iwakuni', 9),
(228, 'Iwatsuki', 9),
(229, 'Izumisano', 9),
(230, 'Jaffna', 5.5),
(231, 'Jaipur', 5.5),
(232, 'Jakarta', 7),
(233, 'Jalib al-Shuyukh', 3),
(234, 'Jamalpur', 6),
(235, 'Jaroslavl', 3),
(236, 'Jastrzebie-Zdrj', 2),
(237, 'Jedda', 3),
(238, 'Jelets', 3),
(239, 'Jhansi', 5.5),
(240, 'Jinchang', 8),
(241, 'Jining', 8),
(242, 'Jinzhou', 8),
(243, 'Jodhpur', 5.5),
(244, 'Johannesburg', 2),
(245, 'Joliet', -5),
(246, 'Jos Azueta', -5),
(247, 'Juazeiro do Norte', -3),
(248, 'Juiz de Fora', -3),
(249, 'Junan', 8),
(250, 'Jurez', -5),
(251, 'Kabul', 4.5),
(252, 'Kaduna', 1),
(253, 'Kakamigahara', 9),
(254, 'Kaliningrad', 2),
(255, 'Kalisz', 2),
(256, 'Kamakura', 9),
(257, 'Kamarhati', 5.5),
(258, 'Kamjanets-Podilskyi', 3),
(259, 'Kamyin', 3),
(260, 'Kanazawa', 9),
(261, 'Kanchrapara', 5.5),
(262, 'Kansas City', -5),
(263, 'Karnal', 5.5),
(264, 'Katihar', 5.5),
(265, 'Kermanshah', 4.5),
(266, 'Kilis', 3),
(267, 'Kimberley', 2),
(268, 'Kimchon', 9),
(269, 'Kingstown', -4),
(270, 'Kirovo-Tepetsk', 3),
(271, 'Kisumu', 3),
(272, 'Kitwe', 2),
(273, 'Klerksdorp', 2),
(274, 'Kolpino', 3),
(275, 'Konotop', 3),
(276, 'Koriyama', 9),
(277, 'Korla', 8),
(278, 'Korolev', 3),
(279, 'Kowloon and New Kowloon', 8),
(280, 'Kragujevac', 2),
(281, 'Ktahya', 3),
(282, 'Kuching', 8),
(283, 'Kumbakonam', 5.5),
(284, 'Kurashiki', 9),
(285, 'Kurgan', 5),
(286, 'Kursk', 3),
(287, 'Kuwana', 9),
(288, 'La Paz', -5),
(289, 'La Plata', -3),
(290, 'La Romana', -4),
(291, 'Laiwu', 8),
(292, 'Lancaster', -4),
(293, 'Laohekou', 8),
(294, 'Lapu-Lapu', 8),
(295, 'Laredo', -5),
(296, 'Lausanne', 2),
(297, 'Le Mans', 2),
(298, 'Lengshuijiang', 8),
(299, 'Leshan', 8),
(300, 'Lethbridge', -6),
(301, 'Lhokseumawe', 7),
(302, 'Liaocheng', 8),
(303, 'Liepaja', 3),
(304, 'Lilongwe', 2),
(305, 'Lima', -5),
(306, 'Lincoln', -5),
(307, 'Linz', 2),
(308, 'Lipetsk', 3),
(309, 'Livorno', 2),
(310, 'Ljubertsy', 3),
(311, 'Loja', -5),
(312, 'London', 1),
(313, 'London', -4),
(314, 'Lublin', 2),
(315, 'Lubumbashi', 2),
(316, 'Lungtan', 8),
(317, 'Luzinia', -3),
(318, 'Madiun', 7),
(319, 'Mahajanga', 3),
(320, 'Maikop', 3),
(321, 'Malm', 2),
(322, 'Manchester', -4),
(323, 'Mandaluyong', 8),
(324, 'Mandi Bahauddin', 5),
(325, 'Mannheim', 2),
(326, 'Maracabo', -4),
(327, 'Mardan', 5),
(328, 'Maring', -3),
(329, 'Masqat', 4),
(330, 'Matamoros', -5),
(331, 'Matsue', 9),
(332, 'Meixian', 8),
(333, 'Memphis', -5),
(334, 'Merlo', -3),
(335, 'Mexicali', -7),
(336, 'Miraj', 5.5),
(337, 'Mit Ghamr', 2),
(338, 'Miyakonojo', 9),
(339, 'Mogiljov', 3),
(340, 'Molodetno', 3),
(341, 'Monclova', -5),
(342, 'Monywa', 6.5),
(343, 'Moscow', 3),
(344, 'Mosul', 3),
(345, 'Mukateve', 3),
(346, 'Munger (Monghyr)', 5.5),
(347, 'Mwanza', 3),
(348, 'Mwene-Ditu', 2),
(349, 'Myingyan', 6.5),
(350, 'Mysore', 5.5),
(351, 'Naala-Porto', 2),
(352, 'Nabereznyje Telny', 3),
(353, 'Nador', 0),
(354, 'Nagaon', 5.5),
(355, 'Nagareyama', 9),
(356, 'Najafabad', 4.5),
(357, 'Naju', 9),
(358, 'Nakhon Sawan', 7),
(359, 'Nam Dinh', 7),
(360, 'Namibe', 1),
(361, 'Nantou', 8),
(362, 'Nanyang', 8),
(363, 'NDjamna', 1),
(364, 'Newcastle', 2),
(365, 'Nezahualcyotl', -5),
(366, 'Nha Trang', 7),
(367, 'Niznekamsk', 3),
(368, 'Novi Sad', 2),
(369, 'Novoterkassk', 3),
(370, 'Nukualofa', 13),
(371, 'Nuuk', -2),
(372, 'Nyeri', 3),
(373, 'Ocumare del Tuy', -4),
(374, 'Ogbomosho', 1),
(375, 'Okara', 5),
(376, 'Okayama', 9),
(377, 'Okinawa', 9),
(378, 'Olomouc', 2),
(379, 'Omdurman', 2),
(380, 'Omiya', 9),
(381, 'Ondo', 1),
(382, 'Onomichi', 9),
(383, 'Oshawa', -4),
(384, 'Osmaniye', 3),
(385, 'ostka', 3),
(386, 'Otsu', 9),
(387, 'Oulu', 3),
(388, 'Ourense (Orense)', 2),
(389, 'Owo', 1),
(390, 'Oyo', 1),
(391, 'Ozamis', 8),
(392, 'Paarl', 2),
(393, 'Pachuca de Soto', -5),
(394, 'Pak Kret', 7),
(395, 'Palghat (Palakkad)', 5.5),
(396, 'Pangkal Pinang', 7),
(397, 'Papeete', -10),
(398, 'Parbhani', 5.5),
(399, 'Pathankot', 5.5),
(400, 'Patiala', 5.5),
(401, 'Patras', 3),
(402, 'Pavlodar', 6),
(403, 'Pemalang', 7),
(404, 'Peoria', -5),
(405, 'Pereira', -5),
(406, 'Phnom Penh', 7),
(407, 'Pingxiang', 8),
(408, 'Pjatigorsk', 3),
(409, 'Plock', 2),
(410, 'Po', 2),
(411, 'Ponce', -4),
(412, 'Pontianak', 7),
(413, 'Poos de Caldas', -3),
(414, 'Portoviejo', -5),
(415, 'Probolinggo', 7),
(416, 'Pudukkottai', 5.5),
(417, 'Pune', 5.5),
(418, 'Purnea (Purnia)', 5.5),
(419, 'Purwakarta', 7),
(420, 'Pyongyang', 9),
(421, 'Qalyub', 2),
(422, 'Qinhuangdao', 8),
(423, 'Qomsheh', 4.5),
(424, 'Quilmes', -3),
(425, 'Rae Bareli', 5.5),
(426, 'Rajkot', 5.5),
(427, 'Rampur', 5.5),
(428, 'Rancagua', -4),
(429, 'Ranchi', 5.5),
(430, 'Richmond Hill', -4),
(431, 'Rio Claro', -3),
(432, 'Rizhao', 8),
(433, 'Roanoke', -4),
(434, 'Robamba', -5),
(435, 'Rockford', -5),
(436, 'Ruse', 3),
(437, 'Rustenburg', 2),
(438, 's-Hertogenbosch', 2),
(439, 'Saarbrcken', 2),
(440, 'Sagamihara', 9),
(441, 'Saint Louis', -5),
(442, 'Saint-Denis', 4),
(443, 'Sal', -4),
(444, 'Salala', 4),
(445, 'Salamanca', -5),
(446, 'Salinas', -7),
(447, 'Salzburg', 2),
(448, 'Sambhal', 5.5),
(449, 'San Bernardino', -7),
(450, 'San Felipe de Puerto Plata', -4),
(451, 'San Felipe del Progreso', -5),
(452, 'San Juan Bautista Tuxtepec', -5),
(453, 'San Lorenzo', -4),
(454, 'San Miguel de Tucumn', -3),
(455, 'Sanaa', 3),
(456, 'Santa Brbara dOeste', -3),
(457, 'Santa F', -3),
(458, 'Santa Rosa', 8),
(459, 'Santiago de Compostela', 2),
(460, 'Santiago de los Caballeros', -4),
(461, 'Santo Andr', -3),
(462, 'Sanya', 8),
(463, 'Sasebo', 9),
(464, 'Satna', 5.5),
(465, 'Sawhaj', 2),
(466, 'Serpuhov', 3),
(467, 'Shahr-e Kord', 4.5),
(468, 'Shanwei', 8),
(469, 'Shaoguan', 8),
(470, 'Sharja', 4),
(471, 'Shenzhen', 8),
(472, 'Shikarpur', 5),
(473, 'Shimoga', 5.5),
(474, 'Shimonoseki', 9),
(475, 'Shivapuri', 5.5),
(476, 'Shubra al-Khayma', 2),
(477, 'Siegen', 2),
(478, 'Siliguri (Shiliguri)', 5.5),
(479, 'Simferopol', 3),
(480, 'Sincelejo', -5),
(481, 'Sirjan', 4.5),
(482, 'Sivas', 3),
(483, 'Skikda', 1),
(484, 'Smolensk', 3),
(485, 'So Bernardo do Campo', -3),
(486, 'So Leopoldo', -3),
(487, 'Sogamoso', -5),
(488, 'Sokoto', 1),
(489, 'Songkhla', 7),
(490, 'Sorocaba', -3),
(491, 'Soshanguve', 2),
(492, 'Sousse', 1),
(493, 'South Hill', -4),
(494, 'Southampton', 1),
(495, 'Southend-on-Sea', 1),
(496, 'Southport', 1),
(497, 'Springs', 2),
(498, 'Stara Zagora', 3),
(499, 'Sterling Heights', -4),
(500, 'Stockport', 1),
(501, 'Sucre', -4),
(502, 'Suihua', 8),
(503, 'Sullana', -5),
(504, 'Sultanbeyli', 3),
(505, 'Sumqayit', 4),
(506, 'Sumy', 3),
(507, 'Sungai Petani', 8),
(508, 'Sunnyvale', -7),
(509, 'Surakarta', 7),
(510, 'Syktyvkar', 3),
(511, 'Syrakusa', 2),
(512, 'Szkesfehrvr', 2),
(513, 'Tabora', 3),
(514, 'Tabriz', 4.5),
(515, 'Tabuk', 3),
(516, 'Tafuna', -11),
(517, 'Taguig', 8),
(518, 'Taizz', 3),
(519, 'Talavera', 8),
(520, 'Tallahassee', -4),
(521, 'Tama', 9),
(522, 'Tambaram', 5.5),
(523, 'Tanauan', 8),
(524, 'Tandil', -3),
(525, 'Tangail', 6),
(526, 'Tanshui', 8),
(527, 'Tanza', 8),
(528, 'Tarlac', 8),
(529, 'Tarsus', 3),
(530, 'Tartu', 3),
(531, 'Teboksary', 3),
(532, 'Tegal', 7),
(533, 'Tel Aviv-Jaffa', 3),
(534, 'Tete', 2),
(535, 'Tianjin', 8),
(536, 'Tiefa', 8),
(537, 'Tieli', 8),
(538, 'Tokat', 3),
(539, 'Tonghae', 9),
(540, 'Tongliao', 8),
(541, 'Torren', 2),
(542, 'Touliu', 8),
(543, 'Toulon', 2),
(544, 'Toulouse', 2),
(545, 'Trshavn', 1),
(546, 'Tsaotun', 8),
(547, 'Tsuyama', 9),
(548, 'Tuguegarao', 8),
(549, 'Tychy', 2),
(550, 'Udaipur', 5.5),
(551, 'Udine', 2),
(552, 'Ueda', 9),
(553, 'Uijongbu', 9),
(554, 'Uluberia', 5.5),
(555, 'Urawa', 9),
(556, 'Uruapan', -5),
(557, 'Usak', 3),
(558, 'Usolje-Sibirskoje', 8),
(559, 'Uttarpara-Kotrung', 5.5),
(560, 'Vaduz', 2),
(561, 'Valencia', -4),
(562, 'Valle de la Pascua', -4),
(563, 'Valle de Santiago', -5),
(564, 'Valparai', 5.5),
(565, 'Vancouver', -7),
(566, 'Varanasi (Benares)', 5.5),
(567, 'Vicente Lpez', -3),
(568, 'Vijayawada', 5.5),
(569, 'Vila Velha', -3),
(570, 'Vilnius', 3),
(571, 'Vinh', 7),
(572, 'Vitria de Santo Anto', -3),
(573, 'Warren', -5),
(574, 'Weifang', 8),
(575, 'Witten', 2),
(576, 'Woodridge', 8),
(577, 'Wroclaw', 2),
(578, 'Xiangfan', 8),
(579, 'Xiangtan', 8),
(580, 'Xintai', 8),
(581, 'Xinxiang', 8),
(582, 'Yamuna Nagar', 5.5),
(583, 'Yangor', 12),
(584, 'Yantai', 8),
(585, 'Yaound', 1),
(586, 'Yerevan', 4),
(587, 'Yinchuan', 8),
(588, 'Yingkou', 8),
(589, 'York', 1),
(590, 'Yuncheng', 8),
(591, 'Yuzhou', 8),
(592, 'Zalantun', 8),
(593, 'Zanzibar', 3),
(594, 'Zaoyang', 8),
(595, 'Zapopan', -5),
(596, 'Zaria', 1),
(597, 'Zeleznogorsk', 7),
(598, 'Zhezqazghan', 6),
(599, 'Zhoushan', 8),
(600, 'Ziguinchor', 0);
```