# SQL Database Design and Implementation: Melbourne Dark Web Crime Syndicate

JEREMY KLAUBER - 30368676

SUBMISSION DATE: September 25th

Contents

[SQL Database Design and Implementation: Melbourne Dark Web Crime Syndicate 1](#_Toc178718736)

[Summary 2](#_Toc178718737)

[Scenario 2](#_Toc178718738)

[Data Creation and Insertion 2](#_Toc178718739)

[Table Creation 2](#_Toc178718740)

[Polygons 3](#_Toc178718741)

[Polygon data was inserted into the SQL table to represent key geographical areas of interest such as city districts and high-risk zones. The polygon coordinates were extracted from Google Maps, ensuring accuracy in reflecting the real-world critical infrastructure. 3](#_Toc178718742)

[Each polygon was created using the **ST\_Polygon** function in **PostGIS**, and these polygons will be used to assess the proximity of key loggers to strategic locations. 3](#_Toc178718743)

[Points 3](#_Toc178718744)

[SQL Queries 5](#_Toc178718745)

[Query 1 – GROUP BY 5](#_Toc178718746)

[Query 2 – HAVING 6](#_Toc178718747)

[Query 3 – Inner SQL (SELECT)\ 6](#_Toc178718748)

[Spatial SQL Extension 7](#_Toc178718749)

[Point Inside Polygon 7](#_Toc178718750)

[Nearest Polygon to Point 7](#_Toc178718751)

[Extended SQL Functionality 8](#_Toc178718752)

[PL/SQL Block 8](#_Toc178718753)

[Stored Procedure 9](#_Toc178718754)

[Trigger 10](#_Toc178718755)

[Intersection Over Union and KML Extraction 11](#_Toc178718756)

[IoU calculations 11](#_Toc178718757)

[KML Code and Results 12](#_Toc178718758)

[Visualization in Google Earth 13](#_Toc178718759)

[Theory and Context 14](#_Toc178718760)

[Additional Spatial Queries 14](#_Toc178718761)

[Reflection on SQL and Spatial Functionality 15](#_Toc178718762)

[Conclusion 17](#_Toc178718763)

[Appendix 18](#_Toc178718764)

## Summary

This report covers the various development strategies used in spatial data management systems. This system will cover geographic data points and polygons, designed and queried using PostgresSQL with PostGIS extensions, gathering 10 polygons and 100 points in a scenario constructed. SQL queries and PL/SQL procedures were employed to analyze spatial relationships, including point-in-polygon and nearest-polygon operations. The results were exported to KML format and visualized in Google Earth.

## Scenario

The scenario centers on a dark web crime syndicate that has covertly deployed key loggers across strategic locations in Melbourne. Tasked by the Australian Security Intelligence Organization (ASIO), the mission is to conduct a comprehensive sweep of the city using advanced spatial data analysis. The city has been divided into polygons, each representing critical zones such as government buildings, transportation hubs, and financial districts. Points scattered across these zones signify potential key logger placements. Using spatial queries, the operation identifies whether these key logger points fall within the designated zones or lie outside of them. For points located outside of polygons, the nearest zone is calculated to determine proximity and potential threat level, aiding ASIO in prioritizing its investigation and response efforts.

## Data Creation and Insertion

### Table Creation

#### Explanation

The tables for polygons and points were created to store spatial data representing key zones and potential key logger locations.

A screenshot of a computer code

Description automatically generated

### Polygons

#### Explanation

### Polygon data was inserted into the SQL table to represent key geographical areas of interest such as city districts and high-risk zones. The polygon coordinates were extracted from Google Maps, ensuring accuracy in reflecting the real-world critical infrastructure.

### Each polygon was created using the **ST\_Polygon** function in **PostGIS**, and these polygons will be used to assess the proximity of key loggers to strategic locations.

A computer screen with numbers and letters

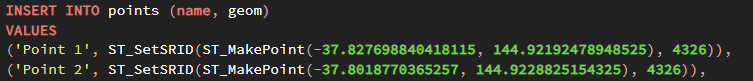
Description automatically generated

### Points

#### Explanation

Points were generated using a Python script that created random geographic coordinates within the Melbourne city boundary. The script ensured a diverse spread of 100 points, each representing a potential key logger location.

These points were then inserted into the database using the **ST\_Point** function in **PostGIS**, which converts latitude and longitude coordinates into spatial data.



## SQL Queries

### Query 1 – GROUP BY

#### Explanation

This query joins the points and polygons tables and groups the results by polygon name to count how many points are inside each polygon.

* **LEFT JOIN**: Joins the *points* table to the *polygon* table, using the **ST\_Within** function to check if each points lies within a polygon.
* **GROUP BY**: Groups the result by *polygon\_name*, allowing the user to count how many points are inside each polygon.
* **COUNT**: Uses **COUNT(pt.name)** to count the total number of points within each polygon.
* **ORDER BY *point\_count* *ASC***: Orders the results in ascending order based on the count of points within each polygon.

A computer screen with white text

Description automatically generatedA screenshot of a black screen

Description automatically generated

### Query 2 – HAVING

#### Explanation

* **HAVING COUNT(pt.name) > 3**: This condition filters out any polygons that have 3 or fewer points. Only polygons with more than 3 points are displayed in the results.

A computer screen with white text

Description automatically generatedA screenshot of a black screen

Description automatically generated

### Query 3 – Inner SQL (SELECT)\

#### Explanation

* **Inner Query**: the subequery (**SELECT p1.name, COUNT(p1.name))** calculates the number of points within each polygon, grouping them by *polygon\_name*.
* **Outer Query**: the outer query joins the *polygon* table with the subquery result (*point\_summary*) and selects the polygon with the highest point count.
* **ORDER BY point\_count DESC LIMIT 1**: Orders the result by the point count in descending order and limits the output to the polygon with the highest number of points.

A computer screen shot of a black background

Description automatically generatedA black and white text

Description automatically generated

## Spatial SQL Extension

### Point Inside Polygon

#### Explanation

* **ST\_Within**: A **PostGIS** spatial function that checks whether the geometry of one feature (point) is fully contained within the geometry of another feature (polygon).
* The function returns **TRUE** if the point’s geometry is spatially located inside the polygon’s geometry.
* This ensures that each row in the result set represents a point that falls within a polygon, helping to identify the location of key loggers within designated critical zones.

A black background with white text

Description automatically generated A black rectangular table with white text

Description automatically generated

### Nearest Polygon to Point

#### Explanation

* **ST\_Distance(pt.geom, p.geom):** Another **PostGIS** function that calculates the shortest distance between the geometries of two features, one being the point (**pt.geom**) and the other being the polygon(**p.geom**)
* The distance is measured in the units defined by the spatial reference system of the geometries, typically in meters. This function is used to identify which polygon is closest to a given point, providing valuable proximity data for points located outside polygons.

A black screen with white text

Description automatically generated A screenshot of a phone

Description automatically generated

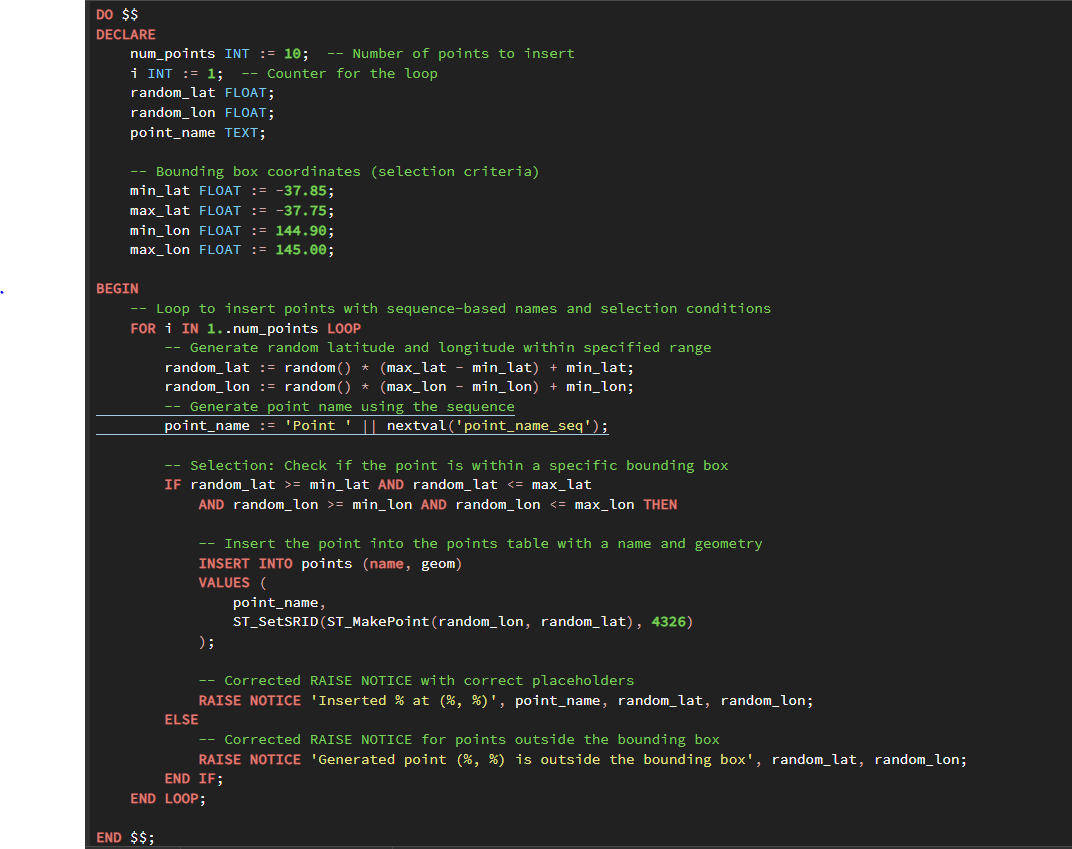
## Extended SQL Functionality

### PL/SQL Block

#### Explanation

This **PL/SQL block** is used to insert multiple points into the *points* table, each with a unique name and within a specified bounding box.

* **Sequence**: The block uses the **nextval(‘point\_name\_sq’)** function to generate unique point names, such as “*Point 1*”, “*Point 2*”, etc.
* **Iteration**: A **FOR** loop is employed to insert multiple points. The loop runs a specified number of times, generating and inserting a point during each iteration.
* **Selection**: An **IF** condition checks whether the generated point’s latitude and longitude fall within a predefined bounding box before inserting it into the *points* table. This ensures that only valid points are inserted within the target geographical area.

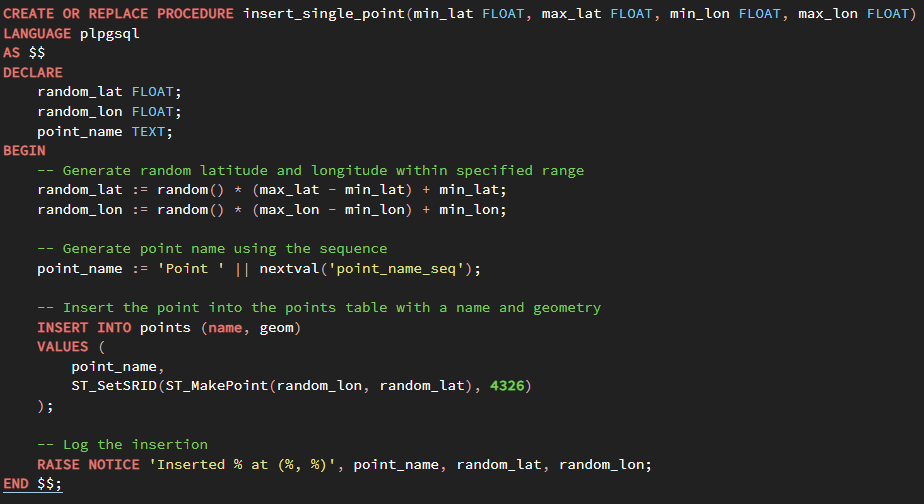


### Stored Procedure

#### Explanation

The **insert\_single\_point** stored procedure generates and inserts random points within specified latitude and longitude boundaries.

* **Functionality**: The procedure generates a random point, assigns a unique name using a sequence, and inserts it into the *points* table.
* **Procedure Call:** The stored procedure is called by a **PL/SQL block**, which sets the bounding box and specifies the number of points (**num\_points**) to insert.
* **Loop**: The block runs a loop to call the procedure repeatedly, inserting one point per iteration.



A computer screen shot of a program

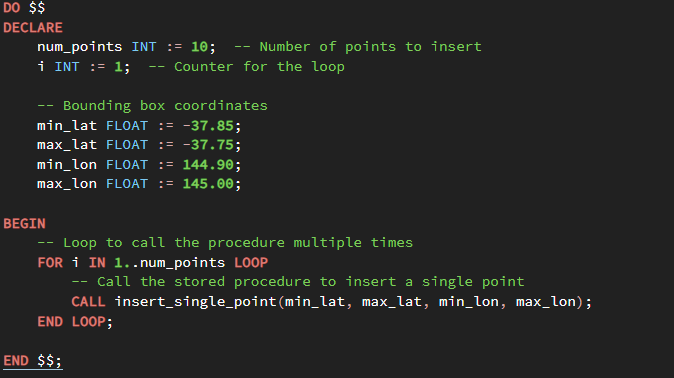
Description automatically generated

### Trigger

#### Explanation:

This **trigger function** automatically checks whether newly inserted points fall within a specified bounding box and logs or raises a notice based on their location.

* A **trigger** **function** (***check\_point\_in\_bounding\_box****)*:
  + Extracts the latitude and longitude from the **geom** column of the newly inserted point.
  + Checks if the point’s coordinates fall within the predefined bounding box.
  + If the point is outside the bounding box, the trigger raises a notice with the point’s coordinates. If inside, it logs the coordinates for future reference.
* **Trigger** (***after\_insert\_check\_bounding\_box***):
  + This **trigger** is attached to the *points* table and fires after a new row is inserted.
  + For each new point, the **trigger function** checks its location, ensuring all points adhere to the geographical boundaries.



A black screen with white text

Description automatically generated

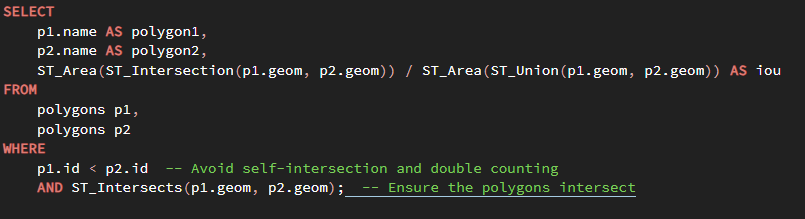
## Intersection Over Union and KML Extraction

### IoU calculations

#### Explanation:

The **IoU** **(Intersection over Union)** calculation is used to determine the overlap between two polygons comparing their areas of intersection and union.

* **ST\_Intersection**: This **PostGIS** function calculates the geometric intersection between two polygons, producing a new polygon that represents the overlapping area.
* **ST\_Union**: This function calculates the geometric union of two polygons, combining them into a single polygon that covers both areas.
* **ST\_Area**: This function computes the area of a given geometry, whether it’s the intersection or the union. The area is used in the **IoU** calculation.
* **IoU Calculation**: **IoU** is determined by dividing the area of the intersection by the area of the union. This ratio provides a measure of how much two polygons overlap relative to their combined area, which can be useful for identifying overlapping critical zones.



### KML Code and Results

#### Explanation:

The following steps outline the process for calculating polygon intersections and generating KML data

* **Intersection Calculation**: This query calculates the intersection between polygon pairs, identifies overlapping areas and inserts these intersections as a new polygon into the database with appropriate names.
* **KML Data Generation**: A subsequent query is run to generate KML data from the polygons, which can be visualized in **Google Earth**. This allows the intersections to be viewed geographically within the context of Melbourne.

A computer screen with text

Description automatically generated

A black background with white text

Description automatically generated

### Visualization in Google Earth

#### Explanation:

The polygons and their intersections are visualized in Google Earth to provide a spatial overview of the data

* **Visualization**: The polygons gathered across Melbourne are displayed in white, with intersections highlighted in red. This color coding makes it easier to identify areas where critical zones overlap, helping in the identification of high-risk areas for key loggers or other security concerns.

A map of a city

Description automatically generated

## Theory and Context

### Additional Spatial Queries

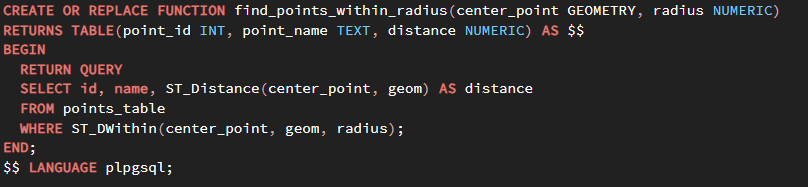
From the scenario that was conducted there are three improvements that could further enhance the spatial data analysis and improve the intelligence gathering process, these include:

1. **Distance Calculation between Points**
   1. Calculating the distance between locations of points and polygons can be critical for assessment based on the needs of the work.
   2. This function takes two points as input and returns the calculated distance between them using the **ST\_Distance** function.

A computer screen with text

Description automatically generated

1. **Finding Points within a Radius**
   1. This function finds all points within a specific radius of a given point. It could help identify clusters of points near polygons.



1. **Polygon Intersection with a Buffer Zone**
   1. Creating buffer zones around points and checking whether they intersect with polygons representing important locations.
   2. This function creates a buffer zone around a point with a specific radius and checks if this zone intersects with any polygon.

A computer screen shot of a program code

Description automatically generated

### Reflection on SQL and Spatial Functionality

When spatial or procedural extensions are unavailable, implementing spatial operations like distance calculations, identifying points within a radius, or checking polygon intersections becomes more challenging. However, you can achieve similar functionality by leveraging basic SQL operations and custom formulas based on geometry.

1. **Calculating Distance Between Points Using Basic Geometry**
   1. Without spatial extensions like **ST\_Distance**, a manual calculation of the Euclidean distance between two points using the basic Pythagorean theorem is used. This requires the points to have a **x** and **y** coordinates stored in separate columns.

A black background with red and green text

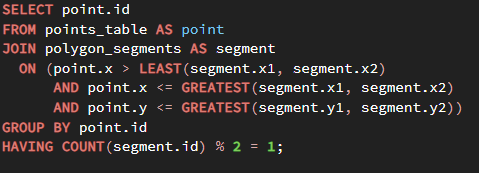
Description automatically generated

1. **Identifying Points within a Radius Using Standard SQL**
   1. Using basic conditions to check Euclidean distance between two points is less than or equal to specified radius.

A black screen with text and numbers

Description automatically generated

1. **Checking for Polygon Intersections Without Spatial Extensions**
   1. To check if a point falls inside a polygon without spatial extensions requires breaking down the polygon into a series of line segments and applying a point-in-polygon algorithm. One such algorithm is the **ray-casting** algorithm.



## Conclusion

In this report, a spatial data management system was developed in a scenario to assist the ASIO in identifying and analyzing key logger placements across Melbourne. Using PostgresSQL with PostGIS extensions, polygons were created to represent critical zones, while points were gathered to simulate potential key logger locations. Various SQL and PL/SQL techniques were implemented to assess spatial relationships, including point-in-polygon checks, distance calculations and proximity analysis.

This project demonstrated the powerful capabilities of spatial SQL in solving real-world problems, particularly in scenarios involving geographic data analysis. The calculations of **Intersection over Union (IoU)** and the generation of **Keyhole Makeup Language (KML)** files provided a means of visualizing overlapping zones and their impact on intelligence gathering. By leveraging Google Earth for visualization, ASIO can more effectively prioritize areas of interest based on the spatial data collected.

In addition to the use of advanced spatial functions, the report also explored alternative methods for performing spatial analysis without the use of procedural or spatial extensions. Basic geometry and SQL operations were shown to be viable alternatives in situations where such extensions are unavailable, though less efficient

Overall, the spatial data system designed in this report provides a robust framework for monitoring and analyzing spatial relationships in high-risk areas. The system’s flexibility and effectiveness can be expanded with further enhancements, such as dynamic radius-based point clustering and more advanced polygon intersection techniques.

## Appendix

Zone 1:

-37.789890400738464, 144.93988387832408

-37.794125061974356, 144.93715349351203

-37.79817801040263, 144.93743418727774

-37.80178716535356, 144.9373831520476

-37.80507356086497, 144.9415170056883

-37.806263079129444, 144.94674811711997

-37.807452578127126, 144.95419926071924

-37.801020990886066, 144.9543778840247

-37.7968472159845, 144.9502185127303

-37.79273370000408, 144.9450129192568

Zone 2:

-37.823006185309765, 144.94902898567088

-37.82255701941874, 144.9513678718137

-37.822158700209606, 144.95293428179923

-37.82159935471262, 144.9546616243175

-37.823785863007096, 144.95578815204684

-37.8253028210104, 144.95634605149377

-37.82676890162839, 144.95715071415756

-37.82697228623435, 144.952805535773

-37.825845189488945, 144.94926502005228

-37.82435366658141, 144.94862128992122

Zone 3

-37.81301314066648, 144.94401415604995

-37.80965675953339, 144.95190404640175

-37.80869776547184, 144.95578830011345

-37.806060467583364, 144.95675936354138

-37.80797851175228, 144.97090047471042

-37.81502689611017, 144.97429919670813

-37.821307542381575, 144.95269303543697

-37.822170490187155, 144.94808048415433

-37.818191257972195, 144.94656319754824

-37.81358850496955, 144.94492452801362

Zone 4:

-37.826853442954594, 144.9597359281583

-37.833580136415826, 144.95212149576676

-37.83520471195878, 144.94634751859786

-37.842871085655666, 144.93956309542438

-37.84839950001652, 144.950966700333

-37.841360673570854, 144.95908635572678

-37.8389952490994, 144.96255074202816

-37.83560372504069, 144.96987647556122

-37.83377964761995, 144.96716992376327

-37.8284781666135, 144.96190116959664

Zone 5:

-37.810544688935714, 144.99324060170164

-37.812568909246956, 145.0138825700805

-37.81465009198595, 145.0138103953659

-37.819924059480456, 145.01445996779745

-37.824827085837086, 145.0155425885166

-37.82730698093828, 145.01539823908738

-37.82573924085064, 145.0017933053831

-37.82422847798232, 144.98966795332836

-37.81809959443499, 144.99053404990372

-37.81105787680101, 144.99165275798018

Zone 6:

-37.80473097556101, 144.90376085779866

-37.800918646248725, 144.9055512558329

-37.80034318322175, 144.90266841154047

-37.79516381423065, 144.90360912915168

-37.79463626553066, 144.89981591297737

-37.791542835985894, 144.88977147654782

-37.80130228577595, 144.88834522726629

-37.80293273153639, 144.89247224646394

-37.80391577702852, 144.89869312098978

-37.803508174387986, 144.89996764162433

Zone 7:

-37.797044593313494, 144.96433879823275

-37.79785986768389, 144.9701044868177

-37.80279927881332, 144.96940653504163

-37.80697114544594, 144.96846581743037

-37.80565247744306, 144.956661328696

-37.80193610456981, 144.95781446641294

-37.79047411261281, 144.95893725840054

-37.79193693442663, 144.96075800216423

-37.79282420567852, 144.9634891178097

-37.79431096064672, 144.96476363844425

Zone 8

-37.808853005044405, 144.9525100898748

-37.80804658243119, 144.9551128868455

-37.80550629363672, 144.95557220395798

-37.80623209935186, 144.96251299587993

-37.79719934250194, 144.96435026432982

-37.79292452787789, 144.95802189522456

-37.79647344802006, 144.9517955965887

-37.79719934250194, 144.95092799759848

-37.804498218309675, 144.9481720949236

-37.80877236317939, 144.9528673365178

Zone 9:

-37.81018514921073, 144.93490540530345

-37.81320587747928, 144.942643566299

-37.81267845763934, 144.9450408789212

-37.821356432299886, 144.95168659365856

-37.82279467208456, 144.94792372321365

-37.8254074026545, 144.9472257714376

-37.82488006998998, 144.9405800567002

-37.82528755465253, 144.93235636403432

-37.81948667868062, 144.93290258716343

-37.81519565477242, 144.93445021936253

Zone 10:

-37.81001378913555, 144.99085959332643

-37.817454717577284, 144.98938001154386

-37.82834391974111, 144.98782825504023

-37.82637714085972, 144.98526605244118

-37.825978077878105, 144.98126035542012

-37.82002038108056, 144.97490897996332

-37.81534510522521, 144.975017242045

-37.80807503329349, 144.97137241862944

-37.80898739532479, 144.97858989073944

-37.809871235292775, 144.98638476061822

Point 1: Latitude = -37.86822151937941, Longitude = 144.86054138177983

Point 2: Latitude = -37.88429438765142, Longitude = 144.95155901167982

Point 3: Latitude = -37.787575578990285, Longitude = 144.87731007422084

Point 4: Latitude = -37.78834202663459, Longitude = 144.94042076081456

Point 5: Latitude = -37.804353856149255, Longitude = 144.81360193035277

Point 6: Latitude = -37.843414452847036, Longitude = 144.93442933493117

Point 7: Latitude = -37.84908171244042, Longitude = 144.85968565013158

Point 8: Latitude = -37.908320761436926, Longitude = 144.8137452693266

Point 9: Latitude = -37.90529174138746, Longitude = 144.81245725954022

Point 10: Latitude = -37.83059066267458, Longitude = 144.9857972637114

Point 11: Latitude = -37.809471123249686, Longitude = 144.99772338557253

Point 12: Latitude = -37.78460339059595, Longitude = 144.94111357022433

Point 13: Latitude = -37.86902391806779, Longitude = 144.91142886424024

Point 14: Latitude = -37.8738248204058, Longitude = 144.85011596596536

Point 15: Latitude = -37.90787523844147, Longitude = 144.81846325694258

Point 16: Latitude = -37.82228008118728, Longitude = 144.93226732821165

Point 17: Latitude = -37.74773316851138, Longitude = 144.99053219533118

Point 18: Latitude = -37.73284966186407, Longitude = 144.9488421013984

Point 19: Latitude = -37.752400669011585, Longitude = 144.93715956269617

Point 20: Latitude = -37.73609174545991, Longitude = 144.95042642075728

Point 21: Latitude = -37.77058777453074, Longitude = 144.86578275296895

Point 22: Latitude = -37.73557151746919, Longitude = 144.98680598987247

Point 23: Latitude = -37.714883282210515, Longitude = 144.90269328264034

Point 24: Latitude = -37.87130551738631, Longitude = 144.96348942383813

Point 25: Latitude = -37.816007985306676, Longitude = 144.87983851553003

Point 26: Latitude = -37.85639843900725, Longitude = 144.81399175283204

Point 27: Latitude = -37.83101211902902, Longitude = 144.9842765083187

Point 28: Latitude = -37.90917625353248, Longitude = 144.92336561732438

Point 29: Latitude = -37.774722176330634, Longitude = 144.99533356547593

Point 30: Latitude = -37.86359244809355, Longitude = 144.89324599443046

Point 31: Latitude = -37.81004364630364, Longitude = 144.95449544122476

Point 32: Latitude = -37.82993430770243, Longitude = 144.8470066393247

Point 33: Latitude = -37.811058316210875, Longitude = 144.8903491199592

Point 34: Latitude = -37.71328009223021, Longitude = 144.80013627938155

Point 35: Latitude = -37.82325512243714, Longitude = 144.8014213540658

Point 36: Latitude = -37.763226943942755, Longitude = 144.97591321843848

Point 37: Latitude = -37.774335800157935, Longitude = 144.88573363216585

Point 38: Latitude = -37.72510884442415, Longitude = 144.8055032755356

Point 39: Latitude = -37.76952403237349, Longitude = 144.94903109729182

Point 40: Latitude = -37.88432966172666, Longitude = 144.95964675836805

Point 41: Latitude = -37.71721407456278, Longitude = 144.8358554886757

Point 42: Latitude = -37.8722785625472, Longitude = 144.8146078382802

Point 43: Latitude = -37.82374364699469, Longitude = 144.99373496338134

Point 44: Latitude = -37.81372918493235, Longitude = 144.91654089911262

Point 45: Latitude = -37.86585945330761, Longitude = 144.80292342040966

Point 46: Latitude = -37.878369896496466, Longitude = 144.98442731753724

Point 47: Latitude = -37.77107557557946, Longitude = 144.91569441981898

Point 48: Latitude = -37.720820518513364, Longitude = 144.83527826142335

Point 49: Latitude = -37.820464326909715, Longitude = 144.95097603624288

Point 50: Latitude = -37.8369797851189, Longitude = 144.92983523107338

Point 51: Latitude = -37.764726392157485, Longitude = 144.91199373209452

Point 52: Latitude = -37.72841680851226, Longitude = 144.90990519077656

Point 53: Latitude = -37.88733089770599, Longitude = 144.8270061806136

Point 54: Latitude = -37.726700504633236, Longitude = 144.96531503540257

Point 55: Latitude = -37.88509870650703, Longitude = 144.88454225451113

Point 56: Latitude = -37.878218766326825, Longitude = 144.90720479655948

Point 57: Latitude = -37.836566765049895, Longitude = 144.84191798648692

Point 58: Latitude = -37.84138425282093, Longitude = 144.91787202784843

Point 59: Latitude = -37.71557103510373, Longitude = 144.91737865304958

Point 60: Latitude = -37.9006196354996, Longitude = 144.9240427849979

Point 61: Latitude = -37.898901990921665, Longitude = 144.99346088544993

Point 62: Latitude = -37.87912174169822, Longitude = 144.83631918495536

Point 63: Latitude = -37.887721945121655, Longitude = 144.9311249016046

Point 64: Latitude = -37.71109525909659, Longitude = 144.9624304540866

Point 65: Latitude = -37.82232890140187, Longitude = 144.96902324685757

Point 66: Latitude = -37.782050299355845, Longitude = 144.94646746752645

Point 67: Latitude = -37.77518053807138, Longitude = 144.86206711178886

Point 68: Latitude = -37.72394257764382, Longitude = 144.84174775158226

Point 69: Latitude = -37.90317670175381, Longitude = 144.99653696645572

Point 70: Latitude = -37.809910509375214, Longitude = 144.8020036754865

Point 71: Latitude = -37.84013315857362, Longitude = 144.80204076769488

Point 72: Latitude = -37.7656619066885, Longitude = 144.84211188149447

Point 73: Latitude = -37.81601232343898, Longitude = 144.80588682268956

Point 74: Latitude = -37.83125704117628, Longitude = 144.8099382891211

Point 75: Latitude = -37.82084105020243, Longitude = 144.87563862269423

Point 76: Latitude = -37.87211330081295, Longitude = 144.86692745165942

Point 77: Latitude = -37.85175388989753, Longitude = 144.8034702867001

Point 78: Latitude = -37.8392067953412, Longitude = 144.88237073368202

Point 79: Latitude = -37.801577203241855, Longitude = 144.90940735716265

Point 80: Latitude = -37.83806529931494, Longitude = 144.9747309206214

Point 81: Latitude = -37.82637490096281, Longitude = 144.888331291435

Point 82: Latitude = -37.907663988385686, Longitude = 144.89220812680244

Point 83: Latitude = -37.87119055863688, Longitude = 144.8866756357806

Point 84: Latitude = -37.806953370801445, Longitude = 144.98493898336258

Point 85: Latitude = -37.78659355367425, Longitude = 144.94230699506744

Point 86: Latitude = -37.89293387559967, Longitude = 144.81446411888464

Point 87: Latitude = -37.83031424355149, Longitude = 144.85214309205875

Point 88: Latitude = -37.822007000225994, Longitude = 144.91382316662197

Point 89: Latitude = -37.77724052586013, Longitude = 144.9861152187426

Point 90: Latitude = -37.82846072400645, Longitude = 144.92681015285774

Point 91: Latitude = -37.82725521586855, Longitude = 144.97415158053977

Point 92: Latitude = -37.732620088326094, Longitude = 144.85210653644253

Point 93: Latitude = -37.88701133283212, Longitude = 144.8148995588466

Point 94: Latitude = -37.796408736863896, Longitude = 144.86329970756458

Point 95: Latitude = -37.78875276903695, Longitude = 144.9149401157082

Point 96: Latitude = -37.88022730261108, Longitude = 144.80905579900127

Point 97: Latitude = -37.73120187571452, Longitude = 144.94182630710733

Point 98: Latitude = -37.84008197832278, Longitude = 144.8978630956321

Point 99: Latitude = -37.8869163281284, Longitude = 144.8325099917932

Point 100: Latitude = -37.807096279842895, Longitude = 144.97533003430635