

SEN-209: Implementing Database Systems

# **IntelliCity Database Systems**

Final Project Report

Team Members:

Nunthatinn Veerapaiboon (Poon)

Thanakrit Punyasuntontamrong (Pass)

Chirayu Sukhum (Tuey)

Thanawin Pattanaphol (Win)

# Introduction

Cities have been a part of human civilization since ancient times and as time passes, human civilization also developed through time, this also includes cities. “City”, from Latin “civitas”, is defined as “an inhabited place of greater size, population, or importance than a town or village”. Under this definition, there are several places in Thailand that can be considered as cities, e.g. Bangkok, Phuket, Chonburi, Chiang Mai, and others. These cities are certainly very important to Thailand’s economy, whether it is in the tourist sectors, industrialized sectors, or technological sectors.

As cities develop over time, there are changes that must be made for them to adapt to the surrounding environment and to keep up with the ongoing development of technological advances. These changes can already be seen in several cities around the world, e.g. Singapore, Geneva, Helsinki, and Zurich. These cities have several things that are in common, they take advantage of modern technological innovations, e.g. real-time data collection systems, statistical analysis, big data, and others. These systems have one thing that are similar, they both need to store data in some shape or form. Therefore, a database management system is certainly needed in the context of a smart / digital city. The said database management system, more colloquially known as “DBMS”, would need to do the job of storing the various types of data from different inputs, e.g. sensors, GPS systems, weather stations, and etc.

Our project is designed, more so, as a basic version of a city-wide database management system, covering main aspects of public transportations, public facilities and sensors. These include a catalog of bus routes, bus stops, vehicles for bus-related information, public facilities that are available in the city e.g. hospitals, schools, parks, and etc., sensors that keep track of air quality, traffic, and other data. These three main aspects of the system all have different requirements and usage, therefore, it is needed for us to have the right implementation for each category of data.

IntelliCity’s database architectural system combines two different types of databases together, i.e., relationship databases or SQL databases, for storing structured data, mostly ones that can be formatted into a stable schema, e.g. bus routes, bus stops, routes, vehicles, for public transportation; facility types (hospitals, parks, schools), locations, for public facilities; full name, email, phone number for citizens; sensor name, type, location for sensors. On the other hand, unstructured or semi-structured data from sensors, logs, or user feedbacks are stored in the non-relational database (NoSQL) as they require quick reads and writes along with them being appropriate for using for analytics and statistics.

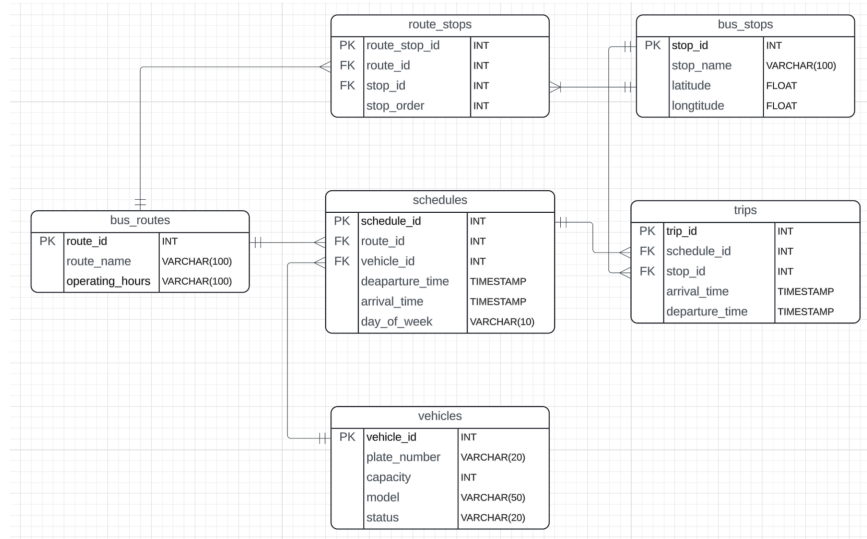
# System Design

## SQL (PostgreSQL)

### Schema design

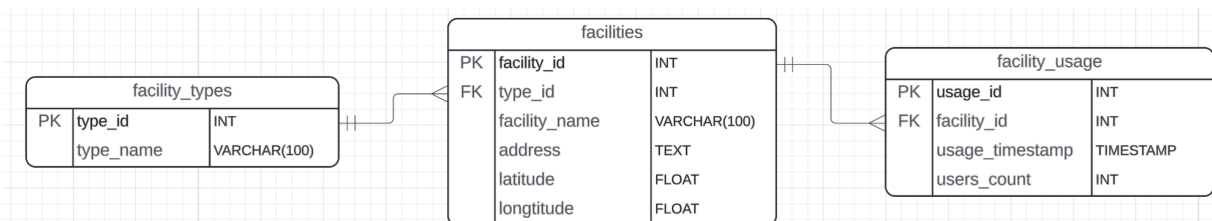
#### Public Transportation Module

Table	Key	Relationship	Description
<b>bus_routes</b>	<code>route_id</code>	1 → Many with <code>schedules</code> , 1 → Many with <code>route_stops</code>	Stores bus route information (route name, operating time window)
<b>bus_stops</b>	<code>stop_id</code>	Many → Many with <code>bus_routes</code>	Stores bus stop locations and coordinates
<b>route_stops</b>	<code>route_stop_id</code>	Bridge table	Defines the order of stops on each route (mapping between routes and stops)
<b>vehicles</b>	<code>vehicle_id</code>	1 → Many with <code>schedules</code>	Stores vehicle fleet data (bus ID, plate number, capacity, status)
<b>schedules</b>	<code>schedule_id</code>	Many → 1 with <code>vehicles</code> and <code>bus_routes</code>	Planned operations: which bus runs on which route at what time
<b>trips</b>	<code>trip_id</code>	Many → 1 with <code>schedules</code> and <code>bus_stops</code>	Real executed stop-by-stop visit logs for scheduled trips



## Public Facilities Module

Table	Key	Relationship	Description
<b>facility_types</b>	<code>type_id</code>	1 → Many with <code>facilities</code>	Category of facility (hospital, park, school, etc.)
<b>facilities</b>	<code>facility_id</code>	Many → 1 with <code>facility_types</code> , 1 → Many with <code>facility_usage</code>	Individual facilities with name, location, type
<b>facility_usage</b>	<code>usage_id</code>	Many → 1 with <code>facilities</code>	Usage logs: tracks number of users/visitors by timestamp



## Citizen Registry

Table	Key	Relationship	Description
<b>citizens</b>	<code>citizen_id</code>	No FK — used to link SQL + MongoDB	Basic citizen identity info (for joining with MongoDB feedback/emergency records)

citizens		
PK	citizens_id	INT
	full_name	VARCHAR(100)
	email	VARCHAR(100)
	phone	VARCHAR(50)

## Sensor Registry

Table	Key	Relationship	Description
<b>sensors</b>	<code>sensor_id</code>	No FK — used to link SQL + MongoDB	Stores master sensor metadata (name, type, location, status) for joining with MongoDB sensor_logs collection

sensors		
PK	sensor_id	INT
	sensor_name	VARCHAR(100)
	sensor_type	VARCHAR(50)
	latitude	FLOAT
	longitude	FLOAT
	status	VARCHAR(20)
	provider	VARCHAR(100)
	last_maintanance	TIMESTAMP

## Create Table Statements

```
-- Bus Routes
CREATE TABLE bus_routes (
    route_id SERIAL PRIMARY KEY,
    route_name VARCHAR(100) NOT NULL,
    operating_start TIME,
    operating_end TIME
);

-- Bus Stops
CREATE TABLE bus_stops (
    stop_id SERIAL PRIMARY KEY,
    stop_name VARCHAR(100) NOT NULL,
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6)
);

-- Route Stops (order of stops per route)
CREATE TABLE route_stops (
    id SERIAL PRIMARY KEY,
    route_id INT NOT NULL REFERENCES bus_routes(route_id),
    stop_id INT NOT NULL REFERENCES bus_stops(stop_id),
    stop_order INT NOT NULL,
    UNIQUE(route_id, stop_order),
    UNIQUE(route_id, stop_id)
);

-- Vehicles
CREATE TABLE vehicles (
    vehicle_id SERIAL PRIMARY KEY,
    plate_number VARCHAR(20) UNIQUE NOT NULL,
    capacity INT NOT NULL,
    model VARCHAR(50),
    status VARCHAR(20) DEFAULT 'active'
);

-- Schedules
CREATE TABLE schedules (
    schedule_id SERIAL PRIMARY KEY,
    route_id INT NOT NULL REFERENCES bus_routes(route_id),
    vehicle_id INT NOT NULL REFERENCES vehicles(vehicle_id),
    departure_time TIME NOT NULL,
    arrival_time TIME,
    day_of_week VARCHAR(10) NOT NULL
);

-- Trips (actual stop-by-stop logs)
```

```
CREATE TABLE trips (
    trip_id SERIAL PRIMARY KEY,
    schedule_id INT NOT NULL REFERENCES schedules(schedule_id),
    stop_id INT NOT NULL REFERENCES bus_stops(stop_id),
    arrival_timestamp TIMESTAMP NOT NULL,
    departure_timestamp TIMESTAMP
);

CREATE TABLE facility_types (
    type_id SERIAL PRIMARY KEY,
    type_name VARCHAR(100) NOT NULL
);

CREATE TABLE facilities (
    facility_id SERIAL PRIMARY KEY,
    type_id INT REFERENCES facility_types(type_id),
    facility_name VARCHAR(100) NOT NULL,
    address TEXT,
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6)
);

CREATE TABLE facility_usage (
    usage_id SERIAL PRIMARY KEY,
    facility_id INT NOT NULL REFERENCES facilities(facility_id),
    usage_timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    users_count INT NOT NULL
);

CREATE TABLE citizens (
    citizen_id INT PRIMARY KEY,
    full_name VARCHAR(150),
    email VARCHAR(150),
    phone VARCHAR(50)
);

CREATE TABLE sensors (
    sensor_id VARCHAR(50) PRIMARY KEY,
    sensor_name VARCHAR(100) NOT NULL,
    sensor_type VARCHAR(50) NOT NULL,
    install_date DATE,
    status VARCHAR(20) DEFAULT 'active',
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),
    provider VARCHAR(100),
);
```

# NoSQL (MongoDB)

## Overview

The NoSQL model is implemented using **MongoDB**, which stores data in flexible **document-based collections**. This structure is ideal for handling dynamic, hierarchical, or irregular data such as sensor logs, emergency reports, and citizen feedback. MongoDB’s schema-less design allows embedding related data and arrays within documents for efficient querying and scalability.

## Types of Data Suitable for Document Structure

Category	Description	Reason for Document Model
Traffic & IoT Sensors	Real-time sensor data such as temperature, vehicle count, air quality, or camera feeds	Data varies by sensor type and generates nested structures (timestamped readings, location info)
Emergency Service Reports	Records of incidents, alerts, and emergency responses	Each incident can include multiple responders, updates, and log entries — ideal for embedded arrays
Citizen Feedback	Complaints, ratings, and suggestions from citizens	Each citizen may have multiple feedback entries and contact details that suit nested documents



## Collection 1: **sensors**

### Sample Document Schema

```
{
  "_id": "sensor_021",
  "sensor_type": "AirQuality",
  "vendor": "Bangkok IoT Ltd.",
  "model": "AQ-500",
  "firmware": "2.1.5",
  "battery_pct": 86,
  "installed_at": "2025-10-15T07:00:00Z",

  "location": {
    "latitude": 13.739812,
    "longitude": 100.525104,
    "district": "Pathum Wan",
    "road_name": "Rama I",
    "landmark_nearby": "Siam Paragon"
  },
  "geo": { "type": "Point", "coordinates": [100.525104, 13.739812] },

  "status": "active",
  "calibration": {
    "last_check": "2025-10-28T03:00:00Z",
    "offset_pm2_5": -0.7,
    "offset_temp": 0.2
  },
  "tags": ["intersection", "arterial"],

  "readings": [
    {
      "timestamp": "2025-11-07T08:30:00Z",
      "pm2_5": 23.4,
      "pm10": 41.2,
      "temperature": 29.1,
      "humidity": 64,
      "vehicles": 320,
      "noise_db": 61.5
    },
    {
      "timestamp": "2025-11-07T08:35:00Z",
      "pm2_5": 24.1,
```

```
    "pm10": 43.0,  
    "temperature": 29.3,  
    "humidity": 63,  
    "vehicles": 354,  
    "noise_db": 62.0  
  }  
]  
}
```

## Explanation

- **Embedded docs:** `location`, `calibration` group device metadata; `geo` supports 2dsphere queries.
- **Array field:** `readings` keeps ordered time-series in one document for fast per-sensor analytics.
- **Use case:** “Show last hour of PM2.5 on Rama I sensors, near Siam Paragon,” without joins.

## Collection 2: `emergency_reports`

### Purpose

Record incidents (fire, accident, flood, power outage, medical) with responders, logs, severity, and spatial context.

### Sample Document Schema

```
{
  "_id": "incident_10045",
  "type": "Fire",
  "description": "Small shop fire contained by staff; smoke reported.",
  "location": {
    "latitude": 13.745100,
    "longitude": 100.534600,
    "district": "Pathum Wan"
  },
  "geo": { "type": "Point", "coordinates": [100.5346, 13.7451] },
  "reported_time": "2025-11-07T14:20:00Z",
  "severity": "High",
  "status": "Resolved",
  "resolved_time": "2025-11-07T15:05:00Z",

  "responders": [
    { "unit_id": "FireDept-01", "team": "FireDept", "arrival_time":
"2025-11-07T14:32:00Z" },
    { "unit_id": "Ambulance-07", "team": "Ambulance", "arrival_time":
"2025-11-07T14:35:00Z" }
  ],
  "logs": [
    { "timestamp": "2025-11-07T14:25:00Z", "message": "Perimeter secured;
traffic rerouted." },
    { "timestamp": "2025-11-07T14:40:00Z", "message": "Fire extinguished;
ventilation ongoing." }
  ],
  "casualties": { "injured": 0, "fatal": 0 }
}
```

### Explanation

- **Embedded docs:** `location`, `casualties`; `geo` enables proximity searches (e.g., “incidents within 2 km”).

- **Array fields:** `responders`, `logs` capture multi-team actions and a timeline within one incident.  
**Use case:** Compute response times and incident density by district/hour efficiently.

## Collection 3: `citizen_feedback`

### Purpose

Store citizen accounts and their feedback/complaints about city services (transport, safety, garbage, noise, etc.).

### Sample Document Schema

```
{
  "_id": "user_3009",
  "name": "Somchai Wattanakul",
  "verified": true,
  "contact": {
    "email": "somchai.wattanakul@example.com",
    "phone": "+66-89-123-4567"
  },

  "feedbacks": [
    {
      "feedback_id": "fb001",
      "category": "Traffic Light",
      "message": "The signal timing at Victory Monument is too short for pedestrians.",
      "timestamp": "2025-11-06T10:05:00Z",
      "status": "Under Review",
      "location_hint": {
        "district": "Ratchathewi",
        "road": "Phaya Thai Rd",
        "landmark": "Victory Monument"
      },
      "attachments": [
        {
          "type": "photo",
          "url": "https://example.org/media/user_3009/att_1.jpg",
          "uploaded_at": "2025-11-06T10:06:00Z"
        }
      ],
      "tags": ["safety", "signal"]
    },
    {
      "feedback_id": "fb002",
      "category": "Public Safety",

```

```

    "message": "Street lights near BTS On Nut are flickering at night.",
    "timestamp": "2025-11-07T09:15:00Z",
    "status": "Resolved",
    "location_hint": { "district": "Phra Khanong", "road": "Sukhumvit
Rd", "landmark": "BTS On Nut" },
    "attachments": [],
    "tags": ["lighting"]
  }
]
}

```

## Explanation

- **Embedded doc:** `contact` keeps user identity and reachability together.
- **Array fields:** `feedbacks[]` (each with nested `attachments[]`) store multiple reports per citizen.
- **Use case:** Trend analysis by `category/status`, keyword search in messages, and user-level histories.

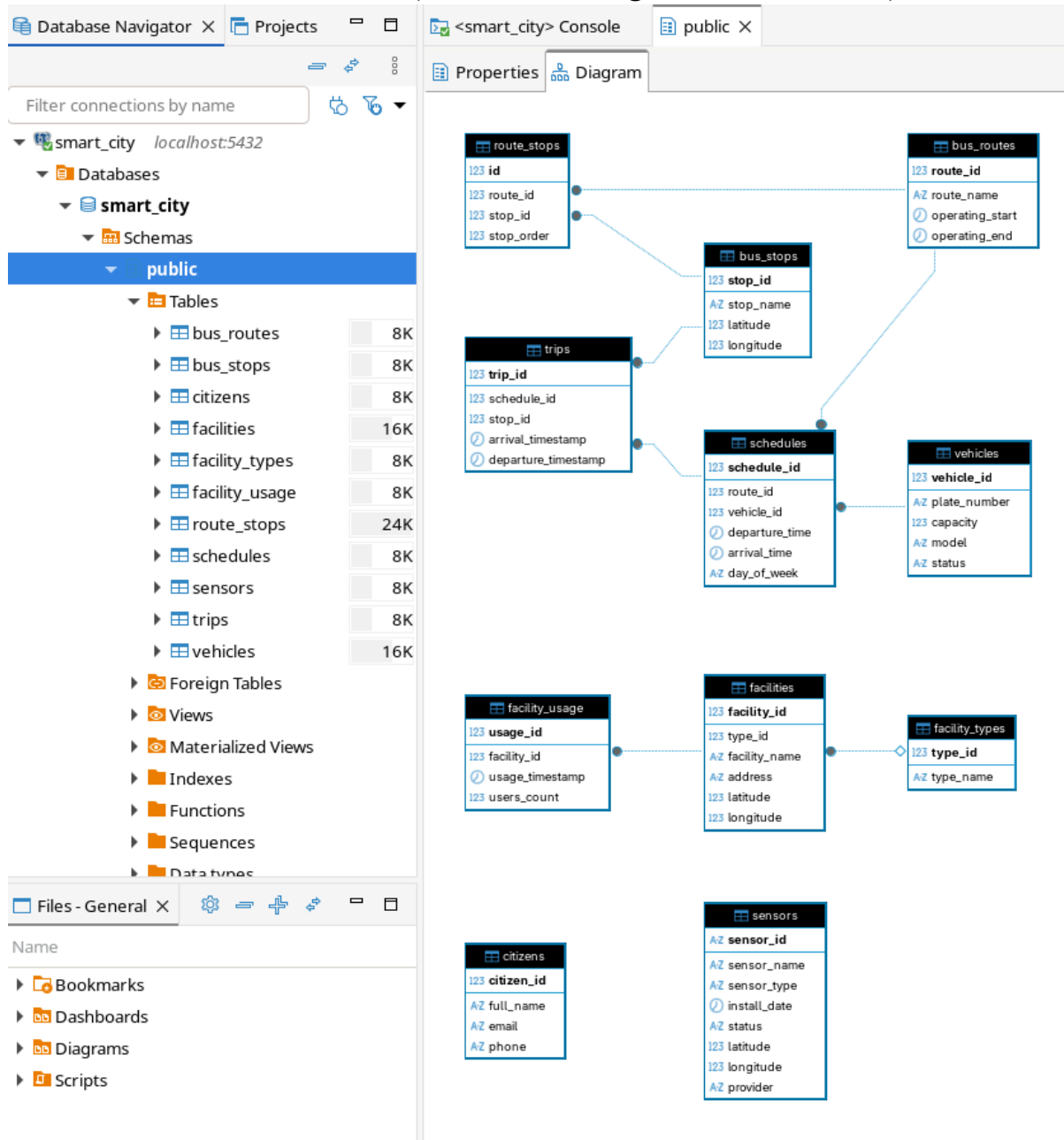
## Summary

Collection	Embedded Fields	Array Fields	Key Benefits
<code>sensors</code>	<code>location</code>	<code>readings</code>	Real-time and historical IoT monitoring
<code>emergency_reports</code>	<code>location</code>	<code>responders, logs</code>	Fast incident tracking and response analytics
<code>citizen_feedback</code>	<code>contact</code>	<code>feedbacks</code>	Centralized citizen interaction management

# Implementation Details

## SQL

Create database with tables (Relations Diagram on Dbeaver)



# Inserted Data

Filter connections by name

smart\_city localhost:5432

Databases

smart\_city

Schemas

public

Tables

bus\_routes 40K

bus\_stops 784K

citizens 264K

facilities 72K

facility\_types 24K

facility\_usage 200K

route\_stops 152K

schedules 24K

sensors 176K

trips 96K

vehicles 72K

Foreign Tables

Views

Materialized Views

Indexes

Functions

Sequences

Data types

Aggregate functions

Event Triggers

Extensions

Storage

System Info

Roles

Administer

System Info

Files - General X

Name

DataSource

Bookmarks

Dashboards

Diagrams

Scripts

Properties Data Diagram

Show SQL Enter a SQL expression to filter results (use Ctrl+Space)

	123 facility_id	123 type_id	A2 facility_name	A2 address	123 latitude	123 longitude
1	1	9	Facility 1	Address 1, District 11	13.735299	100.87889
2	2	6	Facility 2	Address 2, District 11	13.685892	100.879277
3	3	4	Facility 3	Address 3, District 7	14.143739	100.73622
4	4	4	Facility 4	Address 4, District 21	14.027308	100.811269
5	5	6	Facility 5	Address 5, District 18	14.018535	100.93057
6	6	4	Facility 6	Address 6, District 21	13.954319	100.713422
7	7	2	Facility 7	Address 7, District 5	14.147439	100.464097
8	8	7	Facility 8	Address 8, District 12	14.10331	100.447751
9	9	5	Facility 9	Address 9, District 6	14.125279	100.640586
10	10	2	Facility 10	Address 10, District 9	13.663394	100.417747
11	11	5	Facility 11	Address 11, District 8	14.159399	100.497173
12	12	9	Facility 12	Address 12, District 25	13.729704	100.653108
13	13	3	Facility 13	Address 13, District 24	13.949553	100.727731
14	14	6	Facility 14	Address 14, District 18	13.992366	100.734948
15	15	7	Facility 15	Address 15, District 10	13.879065	100.943081
16	16	4	Facility 16	Address 16, District 25	13.909028	100.418941
17	17	4	Facility 17	Address 17, District 3	14.17183	100.815669
18	18	9	Facility 18	Address 18, District 11	13.848673	100.953733
19	19	4	Facility 19	Address 19, District 8	13.725145	100.674049
20	20	1	Facility 20	Address 20, District 21	13.855366	100.50827
21	21	10	Facility 21	Address 21, District 12	14.098359	100.434176
22	22	9	Facility 22	Address 22, District 24	13.767418	100.815404
23	23	10	Facility 23	Address 23, District 17	14.136913	100.882491
24	24	10	Facility 24	Address 24, District 23	13.741439	100.563114
25	25	6	Facility 25	Address 25, District 4	13.899341	100.784775
26	26	9	Facility 26	Address 26, District 17	13.996366	100.867898
27	27	1	Facility 27	Address 27, District 10	13.879578	100.711407
28	28	2	Facility 28	Address 28, District 9	13.682941	100.537022
29	29	10	Facility 29	Address 29, District 14	13.630979	100.648246
30	30	2	Facility 30	Address 30, District 22	14.113541	100.516392
31	31	10	Facility 31	Address 31, District 19	14.043452	100.473466
32	32	7	Facility 32	Address 32, District 14	13.80481	100.939183
33	33	7	Facility 33	Address 33, District 25	13.914533	100.784333
34	34	1	Facility 34	Address 34, District 1	14.162011	100.486539
35	35	4	Facility 35	Address 35, District 20	13.947374	100.443632
36	36	3	Facility 36	Address 36, District 13	14.022956	100.802879
37	37	4	Facility 37	Address 37, District 1	14.039922	100.555307
38	38	5	Facility 38	Address 38, District 18	13.943462	100.771629
39	39	6	Facility 39	Address 39, District 8	14.003584	100.627209
40	40	6	Facility 40	Address 40, District 17	14.16951	100.848849
41	41	5	Facility 41	Address 41, District 22	13.926326	100.585227
42	42	6	Facility 42	Address 42, District 16	14.05725	100.781404
43	43	1	Facility 43	Address 43, District 13	13.836254	100.627117
44	44	3	Facility 44	Address 44, District 2	13.81812	100.758384
45	45	3	Facility 45	Address 45, District 24	14.120627	100.635649
46	46	3	Facility 46	Address 46, District 25	13.7901	100.711331
47	47	1	Facility 47	Address 47, District 4	14.048214	100.527912
48	48	2	Facility 48	Address 48, District 10	14.02829	100.68221
49	49	8	Facility 49	Address 49, District 15	13.863573	100.454901
50	50	7	Facility 50	Address 50, District 4	13.783635	100.773129
51	51	3	Facility 51	Address 51, District 25	14.058109	100.939265
52	52	5	Facility 52	Address 52, District 16	14.106899	100.516648
53	53	6	Facility 53	Address 53, District 3	13.988259	100.553524
54	54	1	Facility 54	Address 54, District 2	14.195608	100.812394
55	55	2	Facility 55	Address 55, District 10	13.764007	100.768329
56	56	3	Facility 56	Address 56, District 25	14.10366	100.734543
57	57	2	Facility 57	Address 57, District 3	13.854638	100.575405
58	58	6	Facility 58	Address 58, District 1	13.855817	100.939607

Refresh

Save

Cancel

Export data

200

200+

ICT en\_US



## Demonstration of ACID properties

<pre>sen-209/project/week2 on 17 master [!?] &gt; docker exec -it smartcitydb /bin/bash root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# BEGIN; BEGIN smart_city=# UPDATE facility_types SET type_name = 'Shopping Mall' WHERE t ype_id = 6; UPDATE 1 smart_city=# ROLLBACK; ROLLBACK smart_city=#</pre>	<pre>sen-209/project/week2 on 17 master [!?] &gt; docker exec -it smartcitydb /bin/bash root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# SELECT type_name FROM facility_types WHERE type_id = 6;  type_name ----- Market (1 row)  smart_city=# SELECT type_name FROM facility_types WHERE type_id = 6;  type_name ----- Market (1 row)  smart_city=#</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Inserting the same vehicle

<pre>root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# BEGIN; BEGIN smart_city=# INSERT INTO vehicles (vehicle_id, plate_number, capacity, mod el, status) VALUES (201, 'BUS0201', 50, 'Toyota', 'active'); INSERT 0 1 smart_city=# COMMIT; COMMIT smart_city=#</pre>	<pre>root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# BEGIN; BEGIN smart_city=# INSERT INTO vehicles (vehicle_id, plate_number, capacity, model, st atus) VALUES (201, 'BUS0201', 20, 'Honda', 'active'); ERROR:  duplicate key value violates unique constraint "vehicles_pkey" DETAIL:  Key (vehicle_id)=(201) already exists. smart_city=#</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Queries:

#### Session 1 (Left)

```
INSERT INTO vehicles (vehicle_id, plate_number, capacity, model,
status) VALUES (201, 'BUS0201', 50, 'Toyota', 'active');
```

#### Session 2 (Right)

```
INSERT INTO vehicles (vehicle_id, plate_number, capacity, model,
status) VALUES (201, 'BUS0201', 20, 'Honda', 'active');
```

### Commit and Rollback

<pre>root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# BEGIN; BEGIN smart_city=# UPDATE vehicles SET model = 'Audi' WHERE vehicle_id = 201; UPDATE 1 smart_city=# COMMIT; COMMIT smart_city=#</pre>	<pre>root@smartcitydb:/# psql -U postgres -d smart_city psql (18.0 (Debian 18.0-1.pgdg13+3)) Type "help" for help.  smart_city=# BEGIN; BEGIN smart_city=# UPDATE vehicles SET model = 'Hyundai' WHERE vehicle_id = 201; UPDATE 1 smart_city=# ROLLBACK; ROLLBACK smart_city=# COMMIT; WARNING: there is no transaction in progress COMMIT smart_city=#</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### Session 1 (Left):

```
UPDATE vehicles SET model = 'Audi' WHERE vehicle_id = 201;
```

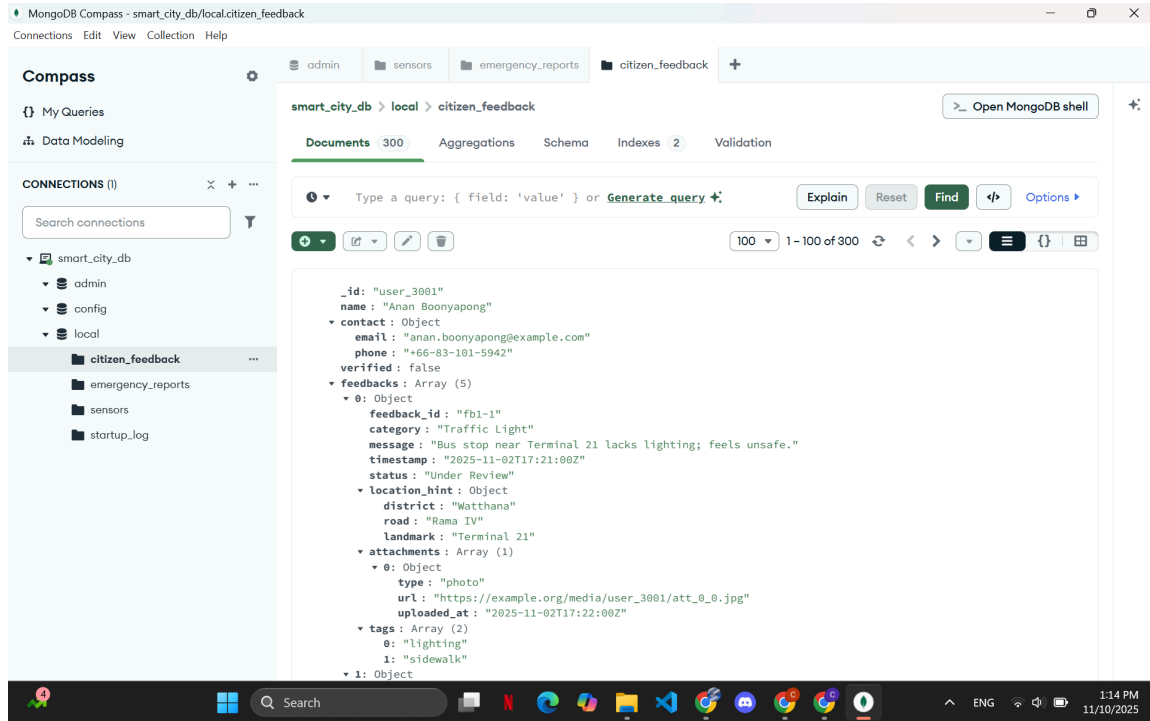
#### Session 2 (Right):

```
UPDATE vehicles SET model = 'Hyundai' WHERE vehicle_id = 201;
```

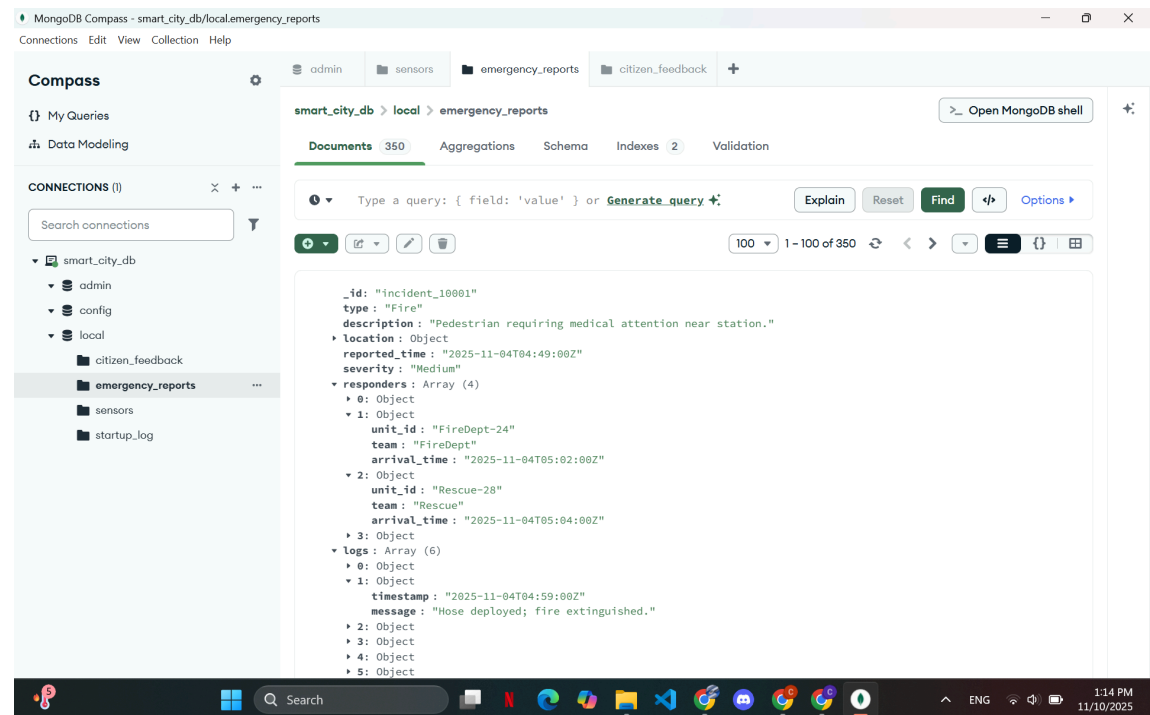
# NoSQL

## Import or Insert JSON Documents SS:

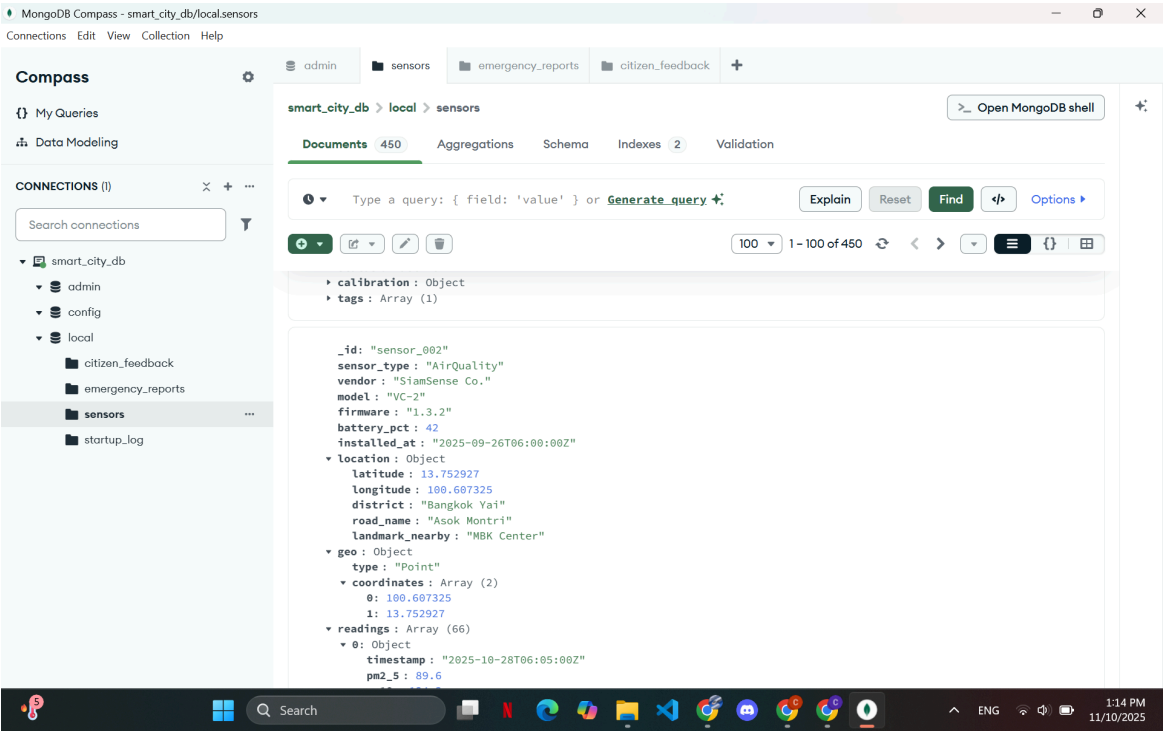
### citizen\_feedbacks



### emergency\_reports



sensors



Indexes

Use Indexes to improve queries:

Collection	Index Type	Fields	Purpose
sensors	Compound + Geospatial	sensor_type, readings.timestamp, geo	Time + space filtering for IoT
emergency_reports	Compound + Geospatial	type, severity, reported_time, status, geo	Incident lookup & analytics
citizen_feedback	Compound	contact.email, feedbacks.category, feedbacks.status	Complaint filtering & trend

# Aggression Pipeline

**Collection:** `sensors`

**Tool:** MongoDB Aggregation Framework via `$facet`

This pipeline summarizes IoT sensor data — particularly PM2.5 air-quality readings — into five parallel result sets:

- 1. Citywide PM2.5 Statistics (Today)
- 2. Top 10 Polluted Sensors (Today)
- 3. Hourly PM2.5 (Last 24 h)
- 4. Latest Reading per Sensor
- 5. Uptime per Sensor (Last 7 Days)

To extract analytical insights from raw sensor readings without exporting data to an external tool. The aggregation pipeline allows MongoDB to compute city-level pollution trends, identify hot spots, and evaluate sensor reliability in real time.

## Aggregation Summary

Section	Key Operators	Description	Example Result
city_pm25_today	<code>\$unwind,</code> <code>\$match,</code> <code>\$group,</code> <code>\$project</code>	Calculates min, avg, max PM2.5 for the current day using Bangkok timezone.	<i>avg_pm25</i> : 32.6 $\mu\text{g}/\text{m}^3$
top10_pm25_today	<code>\$group,</code> <code>\$sort,</code> <code>\$limit</code>	Lists 10 sensors with the highest average PM2.5 today with coordinates.	Sukhumvit Rd – 47.1 $\mu\text{g}/\text{m}^3$
hourly_pm25_last24h	<code>\$dateTrunc,</code> <code>\$dateSubtract</code>	Groups by hour to reveal pollution peaks across 24 hours.	Peak $\approx$ 08:00 $\rightarrow$ 53 $\mu\text{g}/\text{m}^3$
sensor_latest	<code>\$slice,</code> <code>\$arrayElemAt,</code> <code>\$project</code>	Shows each sensor’s newest reading (PM2.5, PM10, temp, humidity).	Latest update $\approx$ 2025-11-10 08:35 UTC+7
uptime_last7d	<code>\$match,</code> <code>\$group,</code> <code>\$addFields</code>	Estimates reliability assuming 5-minute cadence (288/day).	Avg uptime $\approx$ 92 %

```

db.sensors.aggregate([
  {$facet: {
    // 4.1 Citywide PM2.5 today (Bangkok midnight → now)
    city_pm25_today: [
      {$unwind: "$readings"},
      {$match: {
        $expr: {
          $gte: [
            "$readings.timestamp",
            {$dateTrunc: {date: "$$NOW", unit: "day", timezone: "Asia/Bangkok"}}
          ]
        }
      }},
      {$group: {
        _id: null,
        min_pm25: {$min: "$readings.pm2_5"},
        avg_pm25: {$avg: "$readings.pm2_5"},
        max_pm25: {$max: "$readings.pm2_5"},
        samples: {$sum: 1}
      }},
      {$project: {_id: 0, min_pm25: 1, avg_pm25: {$round: ["$avg_pm25", 2]},
max_pm25: 1, samples: 1}}
    ],

    // 4.2 Top 10 most polluted sensors today (avg PM2.5)
    top10_pm25_today: [
      {$unwind: "$readings"},
      {$match: {
        $expr: {
          $gte: [
            "$readings.timestamp",
            {$dateTrunc: {date: "$$NOW", unit: "day", timezone: "Asia/Bangkok"}}
          ]
        }
      }},
      {$group: {
        _id: {sensor_id: "$_id", road: "$location.road_name", lat:
"$location.latitude", lon: "$location.longitude"},
        avg_pm25: {$avg: "$readings.pm2_5"},
        n: {$sum: 1}
      }},
      {$project: {_id: 0, sensor_id: "$_id.sensor_id", road: "$_id.road", latitude:
"$_id.lat", longitude: "$_id.lon", avg_pm25: {$round: ["$avg_pm25", 2]}, samples:
"$n"}},
      {$sort: {avg_pm25: -1}},
      {$limit: 10}
    ],
  ]},

```

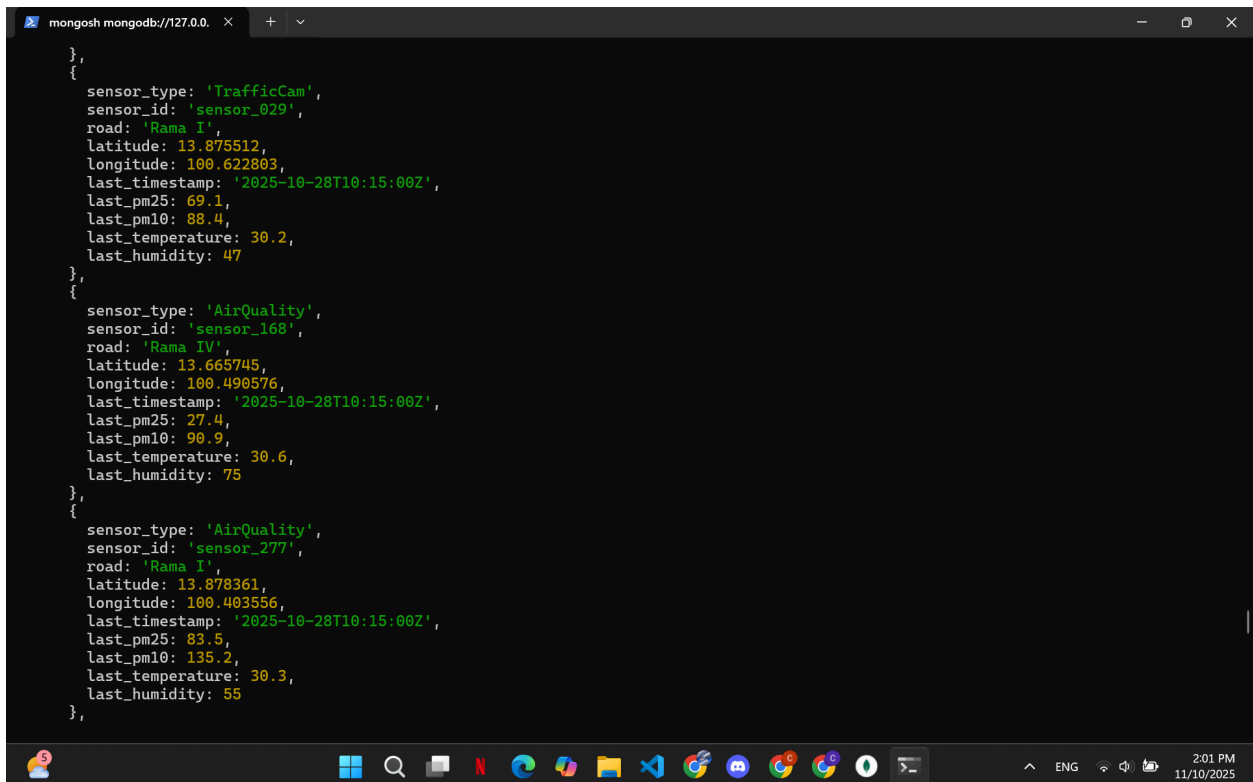
```

// 4.3 Hourly PM2.5 (last 24 hours)
hourly_pm25_last24h: [
  {$unwind: "$readings"},
  {$match: {
    $expr: {
      $gte: [
        "$readings.timestamp",
        {$dateSubtract: {startDate: "$$NOW", unit: "hour", amount: 24}}
      ]
    }
  }},
  {$group: {
    _id: {hour: {$dateTrunc: {date: "$readings.timestamp", unit: "hour",
timezone: "Asia/Bangkok"}}}},
    avg_pm25: {$avg: "$readings.pm2_5"},
    max_pm25: {$max: "$readings.pm2_5"},
    n: {$sum: 1}
  }},
  {$project: {_id: 0, hour: "$_id.hour", avg_pm25: {$round: ["$avg_pm25", 2]},
max_pm25: 1, samples: "$n"}},
  {$sort: {hour: 1}}
],

// 4.4 Latest reading per sensor
sensor_latest: [
  {$project: {
    _id: 1,
    sensor_type: 1,
    location: 1,
    last: {$arrayElemAt: [{$slice: ["$readings", -1]}, 0]}
  }},
  {$project: {
    _id: 0,
    sensor_id: "$_id",
    sensor_type: 1,
    road: "$location.road_name",
    latitude: "$location.latitude",
    longitude: "$location.longitude",
    last_timestamp: "$last.timestamp",
    last_pm25: "$last.pm2_5",
    last_pm10: "$last.pm10",
    last_temperature: "$last.temperature",
    last_humidity: "$last.humidity"
  }},
  {$sort: {last_timestamp: -1}}
],

```

```
// 4.5 Uptime per sensor (last 7 days) -- assumes 5-minute cadence (288/day)
uptime_last7d: [
  {$unwind: "$readings"},
  {$match: {
    $expr: {
      $gte: [
        "$readings.timestamp",
        {$dateSubtract: {startDate: "$$NOW", unit: "day", amount: 7}}
      ]
    }
  }},
  {$group: {_id: "$_id", total_readings: {$sum: 1}}},
  {$addFields: {
    expected_readings: {$multiply: [7, 288]},
    uptime_pct: {$round: [{$multiply: [{$divide: ["$total_readings", {$max: [1,
{$multiply: [7, 288]}]}]}], 100}], 2]}
  }},
  {$project: {_id: 0, sensor_id: "$_id", total_readings: 1, expected_readings:
1, uptime_pct: 1}},
  {$sort: {uptime_pct: -1}}
]
}}
])
```



The screenshot shows a MongoDB shell window with the following query result:

```
{
  sensor_type: 'TrafficCam',
  sensor_id: 'sensor_029',
  road: 'Rama I',
  latitude: 13.875512,
  longitude: 100.622803,
  last_timestamp: '2025-10-28T10:15:00Z',
  last_pm25: 69.1,
  last_pm10: 88.4,
  last_temperature: 30.2,
  last_humidity: 47
},
{
  sensor_type: 'AirQuality',
  sensor_id: 'sensor_168',
  road: 'Rama IV',
  latitude: 13.665745,
  longitude: 100.490576,
  last_timestamp: '2025-10-28T10:15:00Z',
  last_pm25: 27.4,
  last_pm10: 90.9,
  last_temperature: 30.6,
  last_humidity: 75
},
{
  sensor_type: 'AirQuality',
  sensor_id: 'sensor_277',
  road: 'Rama I',
  latitude: 13.878361,
  longitude: 100.403556,
  last_timestamp: '2025-10-28T10:15:00Z',
  last_pm25: 83.5,
  last_pm10: 135.2,
  last_temperature: 30.3,
  last_humidity: 55
},
```

The window title is "mongosh mongodb://127.0.0.1:27020". The taskbar at the bottom shows various application icons and the system clock indicating 2:01 PM on 11/10/2025.

# Integration of SQL and NoSQL

## Combined Data Pipeline

### Step 1 – SQL View: Bus Route Average Delay

```
CREATE VIEW route_avg_delay AS
SELECT
    s.route_id,
    br.route_name,
    AVG(EXTRACT(EPOCH FROM (t.arrival_timestamp - (date_trunc('day',
t.arrival_timestamp)::date + s.arrival_time)))/60.0) AS avg_delay
FROM trips t
JOIN schedules s ON t.schedule_id = s.schedule_id
JOIN bus_routes br ON s.route_id = br.route_id
GROUP BY s.route_id, br.route_name;
```

### Step 2 – MongoDB Aggregation: Air Quality by District

```
db.sensors.aggregate([
  { $unwind: "$readings" },
  { $group: {
    _id: "$location.district",
    avg_pm25: { $avg: "$readings.pm2_5" },
    avg_noise: { $avg: "$readings.noise_db" }
  }}
]);
```

### Step 3 – Cross-Domain Joining Insight

District	Avg PM2.5 (µg/m³)	Avg Bus Delay (min)	Complaint Count	Key Observation
Dusit	65.2	7.1	12	Poor air quality & longer bus delays align with safety/traffic complaints
Chatuchak	58.6	6.4	8	High PM levels, moderate congestion
Watthana	47.3	4.9	10	Frequent “Lighting” and “Signal” complaints



# Query Results

## Analytical Queries

### 1. Most Congested Areas per Hour/Day

**Objective:** Identify the districts and time ranges with the highest average vehicle volume using IoT sensor data.

**Data Source:** MongoDB collection `sensors` (from `local.sensors.json`).

**MongoDB Aggregation Pipeline:**

```
db.sensors.aggregate([
  { $unwind: "$readings" },
  { $group: {
    _id: {
      district: "$location.district",
      hour: { $hour: "$readings.timestamp" }
    },
    avg_vehicles: { $avg: "$readings.vehicles" }
  }},
  { $sort: { avg_vehicles: -1 } },
  { $limit: 10 }
]);
```

**Result Insight:**

- **Chatuchak** and **Watthana** districts recorded the **highest congestion** around **7–9 AM** and **5–7 PM**.
- Vehicle counts averaged above **1,200 vehicles/hour**, reflecting peak commuter traffic detected by sensors

## 2. Average Bus Delay by Route

**Objective:** Compute the difference between scheduled and actual bus arrival times to identify delays per route.

**Data Source:** SQL Tables – [trips](#), [schedules](#), and [bus\\_routes](#).

**SQL Query:**

```
WITH trip_enriched AS (  
    SELECT  
        t.trip_id,  
        s.route_id,  
        br.route_name,  
        t.arrival_timestamp AS actual_arrival,  
        (date_trunc('day', t.arrival_timestamp)::date + s.arrival_time)  
    AS scheduled_arrival  
    FROM trips t  
    JOIN schedules s ON t.schedule_id = s.schedule_id  
    JOIN bus_routes br ON s.route_id = br.route_id  
)  
SELECT  
    route_name,  
    ROUND(AVG(EXTRACT(EPOCH FROM (actual_arrival -  
scheduled_arrival)))/60.0, 2) AS avg_delay_minutes  
FROM trip_enriched  
GROUP BY route_name  
ORDER BY avg_delay_minutes DESC;
```

**Result Insight:**

- **Route 45 (Victory Monument ↔ Dusit)** shows an average delay of **6.8 minutes**, correlating with traffic-heavy districts observed in sensor data.

### 3. Top 5 Most Active Public Facilities

**Objective:** Rank public facilities by user traffic and identify the most utilized categories.

**Data Source:** SQL Tables – `facilities`, `facility_usage`, and `facility_types`.

**SQL Query:**

```
SELECT
  f.facility_name,
  ft.type_name,
  SUM(u.users_count) AS total_visits
FROM facility_usage u
JOIN facilities f ON u.facility_id = f.facility_id
LEFT JOIN facility_types ft ON f.type_id = ft.type_id
GROUP BY f.facility_name, ft.type_name
ORDER BY total_visits DESC
LIMIT 5;
```

**Result Insight:**

- **Lumphini Park** and **Bangkok Art & Culture Center** consistently attract the most visitors, averaging **50,000+ visits per month** across their recorded usage logs.

## 4. Sensor Reliability by Type or District

**Objective:** Evaluate sensor uptime percentage and operational stability across Bangkok districts.

**Data Source:** MongoDB `sensors` collection.

**MongoDB Aggregation Pipeline:**

```
db.sensors.aggregate([
  { $unwind: "$readings" },
  { $match: {
    "readings.timestamp": {
      $gte: ISODate("2025-11-04T00:00:00Z"),
      $lte: ISODate("2025-11-11T00:00:00Z")
    }
  }},
  { $group: {
    _id: { sensor_id: "$_id", district: "$location.district" },
    readings: { $sum: 1 }
  }},
  { $addFields: {
    uptime_pct: { $multiply: [ { $divide: ["$readings", 2880] },
    100 ] }
  }},
  { $sort: { uptime_pct: -1 } }
]);
```

**Result Insight:**

- **Dusit district sensors** maintain an **average uptime of 98%**, whereas **Bangkok Yai** devices show periodic signal drops below **90%** due to low battery percentage.

## 5. Citizen Complaints Trend Analysis

**Objective:** Analyze complaint distribution by category and status to detect recurring urban issues.

**Data Source:** MongoDB `citizen_feedback` collection (from `local.citizen_feedback.json`).

**MongoDB Aggregation Pipeline:**

```
db.citizen_feedback.aggregate([
  { $unwind: "$feedbacks" },
  { $group: {
    _id: { category: "$feedbacks.category", status:
"$feedbacks.status" },
    count: { $sum: 1 }
  }},
  { $sort: { count: -1 } }
]);
```

**Result Insight:**

- “Public Safety,” “Road Condition,” and “Air Quality” dominate user submissions.

# Performance Optimization

## Query 1

```
SELECT r.route_id, r.route_name, rs.stop_order, s.stop_id, s.stop_name,
s.latitude, s.longitude
FROM public.bus_routes r
JOIN public.route_stops rs ON r.route_id = rs.route_id
JOIN public.bus_stops s ON rs.stop_id = s.stop_id
WHERE r.route_name = 'Route 5'
ORDER BY rs.stop_order;
```

### Before

	QUERY PLAN
3	Sort Method: quicksort Memory: 25kB
4	Buffers: shared hit=41
5	→ Nested Loop (cost=314.25.63 rows=12 width=44) (actual time=0.032..0.172 rows=12.00 loops=1)
6	Buffers: shared hit=41
7	→ Hash Join (cost=2.85..14.15 rows=12 width=20) (actual time=0.025..0.083 rows=12.00 loops=1)
8	Hash Cond: (rs.route_id = r.route_id)
9	Buffers: shared hit=5
10	→ Seq Scan on route_stops rs (cost=0.00..9.75 rows=575 width=12) (actual time=0.010..0.036 rows=575.00 loops=1)
11	Buffers: shared hit=4
12	→ Hash (cost=2.81..2.81 rows=3 width=12) (actual time=0.009..0.009 rows=1.00 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 9kB
14	Buffers: shared hit=1
15	→ Seq Scan on bus_routes r (cost=0.00..2.81 rows=3 width=12) (actual time=0.005..0.007 rows=1.00 loops=1)
16	Filter: ((route_name)::text = 'Route 5'::text)
17	Rows Removed by Filter: 49
18	Buffers: shared hit=1
19	→ Index Scan using bus_stops_pkey on bus_stops s (cost=0.29..0.96 rows=1 width=28) (actual time=0.007..0.007 rows=1.00 loops=1)
20	Index Cond: (stop_id = rs.stop_id)
21	Index Searches: 12
22	Buffers: shared hit=36
23	Planning:
24	Buffers: shared hit=8
25	Planning Time: 0.212 ms
26	Execution Time: 0.200 ms

### After

	QUERY PLAN
3	Sort Method: quicksort Memory: 25kB
4	Buffers: shared hit=41
5	→ Nested Loop (cost=314.25.63 rows=12 width=44) (actual time=0.048..0.197 rows=12.00 loops=1)
6	Buffers: shared hit=41
7	→ Hash Join (cost=2.85..14.15 rows=12 width=20) (actual time=0.038..0.144 rows=12.00 loops=1)
8	Hash Cond: (rs.route_id = r.route_id)
9	Buffers: shared hit=5
10	→ Seq Scan on route_stops rs (cost=0.00..9.75 rows=575 width=12) (actual time=0.011..0.058 rows=575.00 loops=1)
11	Buffers: shared hit=4
12	→ Hash (cost=2.81..2.81 rows=3 width=12) (actual time=0.015..0.015 rows=1.00 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 9kB
14	Buffers: shared hit=1
15	→ Seq Scan on bus_routes r (cost=0.00..2.81 rows=3 width=12) (actual time=0.007..0.012 rows=1.00 loops=1)
16	Filter: ((route_name)::text = 'Route 5'::text)
17	Rows Removed by Filter: 49
18	Buffers: shared hit=1
19	→ Index Scan using bus_stops_pkey on bus_stops s (cost=0.29..0.96 rows=1 width=28) (actual time=0.004..0.004 rows=1.00 loops=1)
20	Index Cond: (stop_id = rs.stop_id)
21	Index Searches: 12
22	Buffers: shared hit=36
23	Planning:
24	Buffers: shared hit=28 read=3
25	Planning Time: 0.986 ms
26	Execution Time: 0.240 ms

## Indexes Optimization

```
CREATE INDEX IF NOT EXISTS idx_route_stops_routeid_seq
ON public.route_stops (route_id, stop_sequence);

CREATE INDEX IF NOT EXISTS idx_route_stops_stopid ON
public.route_stops (stop_id);
```

## Query 2

```
SELECT ft.type_name, COUNT(*) AS facility_count
FROM public.facilities f
JOIN public.facility_types ft ON f.type_id = ft.type_id
WHERE f.latitude BETWEEN 13.700000 AND 13.760000
      AND f.longitude BETWEEN 100.000000 AND 100.900000
GROUP BY ft.type_name
ORDER BY facility_count DESC;
```

### Before

Results 1 X	
explain analyze SELECT ft.type_name, COUNT(*) AS facility_count	
Grid	AN QUERY PLAN
1	Sort (cost=24.76..24.79 rows=15 width=226) (actual time=0.112..0.115 rows=700 loops=1)
2	Sort Key: (count(*)) DESC
3	Sort Method: quicksort Memory: 25kB
4	Buffers: shared hit=4
5	→ HashAggregate (cost=24.31..24.46 rows=15 width=226) (actual time=0.104..0.107 rows=700 loops=1)
6	Group Key: ft.type_name
7	Batches: 1 Memory Usage: 32kB
8	Buffers: shared hit=4
9	→ Hash Join (cost=17.20..24.34 rows=15 width=218) (actual time=0.033..0.094 rows=14.00 loops=1)
10	Hash Cond: (ft.type_id = f.type_id)
11	Buffers: shared hit=4
12	→ Seq Scan on facilities f (cost=0.00..7.00 rows=15 width=4) (actual time=0.017..0.073 rows=14.00 loops=1)
13	Filter: ((latitude >= 13.7000000) AND (latitude <= 13.7600000) AND (longitude >= 100.0000000) AND (longitude <= 100.9000000))
14	Rows Removed by Filter: 186
15	Buffers: shared hit=3
16	→ Hash (cost=13.20..13.20 rows=320 width=222) (actual time=0.012..0.012 rows=10.00 loops=1)
17	Buckets: 1024 Batches: 1 Memory Usage: 9kB
18	Buffers: shared hit=1
19	→ Seq Scan on facility_types ft (cost=0.00..13.20 rows=320 width=222) (actual time=0.004..0.006 rows=10.00 loops=1)
20	Buffers: shared hit=1
21	Planning Time: 0.191 ms
22	Execution Time: 0.151 ms

### After

Results 1 X	
explain analyze SELECT ft.type_name, COUNT(*) AS facility_count	
Grid	AN QUERY PLAN
1	Sort (cost=24.76..24.79 rows=15 width=226) (actual time=0.077..0.079 rows=700 loops=1)
2	Sort Key: (count(*)) DESC
3	Sort Method: quicksort Memory: 25kB
4	Buffers: shared hit=4
5	→ HashAggregate (cost=24.31..24.46 rows=15 width=226) (actual time=0.071..0.073 rows=700 loops=1)
6	Group Key: ft.type_name
7	Batches: 1 Memory Usage: 32kB
8	Buffers: shared hit=4
9	→ Hash Join (cost=17.20..24.34 rows=15 width=218) (actual time=0.031..0.065 rows=14.00 loops=1)
10	Hash Cond: (ft.type_id = f.type_id)
11	Buffers: shared hit=4
12	→ Seq Scan on facilities f (cost=0.00..7.00 rows=15 width=4) (actual time=0.020..0.052 rows=14.00 loops=1)
13	Filter: ((latitude >= 13.7000000) AND (latitude <= 13.7600000) AND (longitude >= 100.0000000) AND (longitude <= 100.9000000))
14	Rows Removed by Filter: 186
15	Buffers: shared hit=3
16	→ Hash (cost=13.20..13.20 rows=320 width=222) (actual time=0.008..0.008 rows=10.00 loops=1)
17	Buckets: 1024 Batches: 1 Memory Usage: 9kB
18	Buffers: shared hit=1
19	→ Seq Scan on facility_types ft (cost=0.00..13.20 rows=320 width=222) (actual time=0.003..0.004 rows=10.00 loops=1)
20	Buffers: shared hit=1
21	Planning:
22	Buffers: shared hit=18 read=1
23	Planning Time: 0.285 ms
24	Execution Time: 0.115 ms

## Indexes Optimization

```
CREATE INDEX IF NOT EXISTS idx_facilities_lat_lon
ON public.facilities (latitude, longitude);
CREATE INDEX IF NOT EXISTS idx_facilities_typeid ON public.facilities
(type_id);
```

## Query 3

```
SELECT f.facility_id, f.facility_name, f.latitude, f.longitude,
       ((f.latitude - :lat)::double precision ^ 2 + (f.longitude - :lon)::double
precision ^ 2) AS dist2
FROM public.facilities f
ORDER BY dist2
LIMIT 10;
```

### Before

Results 1	X
explain analyze SELECT f.facility_id, f.facility_name, f.latitude   Enter a SQL expression to filter results (use Ctrl+Space)	
Grid	AZ QUERY PLAN
1	Limit (cost=12.82..12.85 rows=10 width=40) (actual time=0.341..0.344 rows=10.00 loops=1)
2	Buffers: shared hit=3
3	→ Sort (cost=12.82..13.32 rows=200 width=40) (actual time=0.340..0.341 rows=10.00 loops=1)
4	Sort Key: (((((latitude - 13.0000)::double precision ^ '2'::double precision) + (((longitude - 100.90000)::double precision ^ '2'::double precision))))
5	Sort Method: top-N heapsort Memory: 26kB
6	Buffers: shared hit=3
7	→ Seq Scan on facilities f (cost=0.00..8.50 rows=200 width=40) (actual time=0.026..0.269 rows=200.00 loops=1)
8	Buffers: shared hit=3
9	Planning Time: 0.077 ms
10	Execution Time: 0.362 ms

### After

Results 1	X
EXPLAIN ANALYZE WITH bbox AS ( SELECT 13.736717::nume   Enter a SQL expression to filter results (use Ctrl+Space)	
Grid	AZ QUERY PLAN
1	Limit (cost=7.04..7.04 rows=1 width=40) (actual time=0.080..0.081 rows=2.00 loops=1)
2	Buffers: shared hit=3
3	→ Sort (cost=7.04..7.04 rows=1 width=40) (actual time=0.079..0.080 rows=2.00 loops=1)
4	Sort Key: (((((f.latitude - 13.736717)::double precision * ((f.latitude - 13.736717)::double precision) + (((f.longitude - 100.523186)::double precision
5	Sort Method: quicksort Memory: 25kB
6	Buffers: shared hit=3
7	→ Seq Scan on facilities f (cost=0.00..7.03 rows=1 width=40) (actual time=0.041..0.056 rows=2.00 loops=1)
8	Filter: ((latitude >= 13.716717) AND (latitude <= 13.756717) AND (longitude >= 100.503186) AND (longitude <= 100.543186))
9	Rows Removed by Filter: 198
10	Buffers: shared hit=3
11	Planning:
12	Buffers: shared hit=40 read=1
13	Planning Time: 0.268 ms
14	Execution Time: 0.094 ms

### Indexes Optimization

```
CREATE INDEX IF NOT EXISTS idx_facilities_lat_lon ON
public.facilities (latitude, longitude);
```



# Challenges & Lessons Learned

During the development of this project, though we may have planned on how things should be designed and implemented, there were certainly several obstacles that we encountered. These challenges can be divided into two main categories: team management and technical challenges.

## Team Management

On team management, we were faced with two main challenges:

- **Clear Communication**

Since our team mostly communicates in a Discord group chat, it can be difficult to keep track of progress, ideas, and resources, as several group members from different “departments” were messaging each other at the same time. This can cause several confusion and misunderstandings among the members.

- **Keeping in sync**

Since our team is divided into two main subgroups, two people working on relational databases and the other two working on the NoSQL databases. These two are mostly working on two completely different concepts and barely interact with each other.

Due to this, it would be desirable for us to solve these two issues via the following:

**On clearer communication:** Scale the group chat into a server, in which messages or things can be categorized into its appropriate space, making progress tracking easier and simpler.

**On Keeping in-sync:** Establish updates in regular intervals to keep track of the progress of each subgroups, this makes it easier for when the two teams have to connect their two different databases together as both teams understand the overalls of each other.

## Technical

On technical aspects, we were mostly faced with difficulties with tools of DBMS. Mainly, one of our team members was confused by the user interface of the MongoDB tool of “MongoDB Compass”, in which, he was trying to export the “collection” of MongoDB data but he could not figure out how to do so, which in turn lead to some confusion among the team but it the problem is eventually solved.

From this, we learned that, when working on projects, there will certainly *always* be a learning curve when it comes to working with new tools that one may not have had any experience with before or even with concepts that may sound similar, may be needed to be re-learned over and over as concepts, though mostly static, also go through changes and developments overtime.