Project Report:

Water & Waste Management System

Author:

Raul Lopes a86079

&

Ali Ahammad a81317

Course name:

Data Modelling and Integration

Table of Contents

1. Introduction

In the water and waste management sectors, being abreast of resource management while balancing sustainability and operational effectiveness is necessary. Making new systems for two non-existent sectors will serve to accomplish our goal in the project, i.e., a water management system in one sector and a solid waste disposal system in another. These organizations will try to solve their problems with advanced data modelling and integration techniques. Water constitutes being a fundamental, but one of a finite, natural recourse. It is a critical utility for industrial processes, agriculture, and general human use. Industries, with food production et al., need water as an extensive resource to deploy in cleaning, cooling and processing. Poor water management will result in added costs, environmental impacts and regulatory difficulties. In this context, water management companies need to utilize data-driven technology for monitoring, optimization, and control of water consumption and quality.

The project's aim is to design an intelligent water management system integrating data from structured (MySQL), semi-structured (MongoDB), and unstructured (XML) databases. The system namely allows to:

- Real-time tracking of water usage and quality
- Optimization of water treatment and resource utilization
- Compliance monitoring with safety and environmental regulations
- Advanced reporting for sustainability initiatives

All databases that are not of the same kind are integrated to provide a unified platform for querying and analysing for complex data which allows for decision-making in a timely and proactive manner.

The efficient management of waste is important for environmental sustainability and operational transparency. As more and more businesses adapt to the widespread use of data modelling, the centralized systems operate effectively in relation to the waste generated, the collection and recycling processes that need to be optimized as well as the regulatory compliance, which is very important to a waste management company. This project is on the development of a scalable waste management framework using a combination of SQL (PostgreSQL), NoSQL (Apache Cassandra), and XML databases. These technologies are able to support various functions such as customer management, waste classification, compliance tracking, and performance analysis. The system adheres to all sectors in-use-residential, commercial, and industrial-and can handle the various and sometimes complex waste streams which include hazardous and e-materials.

Advanced query techniques aimed at producing actionable insights through appropriate industry approaches.

- Delineating the regions where noncompliance highlights occur.
- Identifying patterns related to recycling waste.
- Helping in resource allocation for waste removal.

Data integration in the waste management project serve various organizations with the objectives of optimizing their operations, directing-in more insight into decision-making processes, and attaining sustainability goals.The project integrates the two systems together by joining five distinct databases (MySQL, MongoDB, XML, PostgreSQL, Apache Cassandra) using the Global-As-View approach

(GAV), Local-As-View approach (LAV), or Global-Local-As-View approach. Attribute matching algorithms like Jaccard and Levenshtein assure proper schema mapping among heterogeneous data sources. Name queries are used to further evaluate the integration process by confirming data consistency and usability.

This project paper will describe the complete workflow for the project achievement, including the system requirements, database models, stages of integration, results with querying. The proposed integrated framework exemplifies the utility of data-driven solutions in tackling actual requirements in resource management, fostering an economic boon and environmental sustainability.

2. Project Outline

The task concentrates on creating a single, real-world virtual data integration system that links datasets involving two make-believe companies, namely Water Management and Waste Management. The purpose herein is to endow equal access to all types of autonomous and heterogeneous data sources commingled in the database with competent storage, processing, and query support, which can be deployed on multiple platforms.

Water Management enjoys features from three sundry data sources: MySQL holds data about structured water-related data (e.g., information regarding sources of water, billing), as to also quality tests; and MongoDB is useful for the management of semi-structured data like customer feedback and maintenance records. The Waste Management's world relies upon PostgreSQL by designating its structured data for waste collection unofficial ties and information on waste disposal and XML Data Storage for its various wastes through licensing, certification, and compliance. It also uses Apache Cassandra for the archiving of enormous time series data out of sensor readings from waste management operations.

Through schema mapping and a matching application within the standard data sources, the systems successfully integrate by GAV (Global-As-View), LAV (Local-As-View), and GLAV (Global-Local-As-View) approaches. This is done to provide singular, integrated views of the operations involving both companies. The project ended with testing for the integration through performance in a series of queries to gauge the system's performance, reliability, ultimately to perceive relevance and dependable insights making anybody willing to action water and waste management operations.

2.1 Policy Implementation and Customer-Based Business Model

The integration of water and waste management data into a unified system enables the implementation of data-driven policies and a customer-focused business model. Through the virtual integration of heterogeneous data sources, the system supports efficient policy enforcement by ensuring real-time monitoring and analytics for regulatory compliance, water conservation, and waste reduction initiatives. The **Water Management Company** aims to create an intricate system where combined billing, water quality testing, and consumption data assist in the promotion of transparent pricing practices and tailor-made conservation plans. In contrast, the **Waste Management Company** aims to provide tailor-made waste management and flexible solutions as a part of its customers-based services that include use of the "**pay-as-you-go**" principle tailored to a given household's pattern of use. Such a system borrowed from the American practice is most appropriate for the **Greater Faro Region**, delivering payments and

service conditions that are crystal-clear, enhancing therefore the clients' acceptance and accessibility while doggedly adhering to scheduled and optimized waste collection and recycling programs. The system enhances forecasting or a lot of predictive analysis opportunities, expenditure of superb cost-savings, because the prediction on waste or water demand provides businesses with ample mantle space to plan for the satisfaction of their own service needs and environmental concerns. Directing our actions toward customer satisfaction and interaction and adding value for enhanced contracting models was envisaged as a key factor for sustenance.

3. Tools Used to Generate and Populate Databases

We had Docker as the primary containerization mechanism for data sources. As an emerging technology with a different use case, Docker helps in the efficient and scalable management of containers and guarantees consistency and portability throughout development and production environments. Having Docker configurations facilitated spinning up many databases' systems-Cassandra, PostgreSQL, MySQL, MongoDB, BaseX-with the Docker containers and with relative less effort in setup and manual installations. Much needed flexibility for the project running with disparate database systems was provided, thus offering the means of integrating data from autonomous sources or sources of different types. Docker also provides rapid temporal reproducibility to the environment, thereby making dependency- and configuration-management easier.

There is a multitude of database ecosystem products available to perform various forms of data persistence:

- **Cassandra** is suitable for large-scale, distributed NoSQL storage (ideal for time-series data and scalable for high-throughput data insertion),
- **PostgreSQL** and **MySQL** are deemed apt due to the provision of relational SQL environment (which is good for structured data and can process complex queries),
- **MongoDB** enables NoSQL storage of documents (useful on account of schema-free or hierarchical data),
- **BaseX** is considered for storing XML data (XML-integration and XML-based querying).

We established these to run as Docker services, each with its specific execution script and environment variables. Persistent data in the Docker network is facilitated by Docker volumes, ensuring backups are available when instances are terminated, therefore, keeping data synchronize for the rapid recovery of databases.

Data Fetching and Integration for Website

To render the fetched data, the web server-serving using Nginx in a Docker container-serves data to the end user. On the website-built front: HTML and JavaScript source code call the respective backend services and ready AJAX to receive responses. Pseudo code implementation shall consist of

Pseudo-Code:

```
Function LoadData(Type):
   -> Define urls as a dictionary with the following key-value pairs:
      sensors -> 'sensors.json'
      reports -> 'reports.json'
      temperature -> 'temperature.json'
```

```
      ph -> 'ph.json'
      turbidity -> 'turbidity.json'

   -> Fetch the data from the URL corresponding to the selected type (urls[Type])
   -> Parse the fetched data as JSON
   -> Get the table header and body elements (tableHead, tableBody)
   -> Clear the contents of the table header and body (tableHead.innerHTML = '',
tableBody.innerHTML = '')

   -> IF the data is not empty THEN
      -> Extract the keys from the first item in the data (headers)
      -> Create a header row by looping through the headers and generating table header cells
      -> Insert the header row into the table header (tableHead.innerHTML = headerRow)

      -> FOR each row in the data:
         -> FOR each header in the headers, generate a table cell for that header with the
corresponding value in the row
         -> Add the generated row to the table body (tableBody.innerHTML += rowHtml)

   -> ELSE
      -> IF no data is available THEN
         -> Insert a row into the table body indicating that no data is available (tableBody.innerHTML
= '<tr><td colspan="100%">No Data Available</td></tr>')
```

This pseudocode shows how the data flow works: introduction of the JSON file, data is parsed and rendered in a table on the frontend. It deals with cases where data is available and where none is found. It makes sure the user gets a clear indication about the status of the data. The script is listening to button-click events on the Internet, each button representing a specific type of query: sensors, temperature, reports, etc. It carries out a fetch request to load JSON data, which is then parsed and dynamically rendered in an HTML table. This system provides easy management for the display of various data types originating from different databases, whereby the user would not need to know where the certain data was extracted from.

The backend handles the databases interactions and database pooling. Depending on the frontend's requests, the backend uses the necessary queries to merge and format the data in JSON format so that the frontend can consume it quickly.

Data Integration and Server-Side Rendering

The project had to factor in scalability and easy configuration to make sure they got a gold star. This Docker help me quickly deploy and manage a good deal of database systems with nary a worry about conflicts or bad customization, which is especially helpful for deployments across various databasing systems. Docker's containerization gave me an implementation where We could isolate the environment for every database, guaranteeing that the dependencies and configurations of each system would not break the others.

By being such a solid fit for serving the front end, a highly scalable and efficient lightweight proxy-server, Apache's performance benefits were unprecedented. Indeed, it was deemed extremely easy to

configure within Docker containers and could have been employed to serve static Web files rapidly on the Internet. It was clear that with AJAX calls and dynamic rendering on the front end, creating an elegant and interactive user interface while easing the page-reload problem was to be our next step.

When Nginx, Docker, and a multiplicity of database systems came together, it became possible to install a seamless data integration mechanism. The backend side allowed for queries to sift through many databases, while some AJAX client-side real-time data treatment dispensed with any unnecessary page reloads. In terms of the goal, such an approach was the most efficient one towards integrating heterogeneous data sources and making them presentable in an unchanged manner on the accessible end.

4. Water Management Company

4.1 Domain Problem:

Water management encompasses activities that manage and optimize water resources for sustainable use in a series of regions while collecting and processing data about water sources, water usage, and water quality. Real-time monitoring of water quality required by the Water Organising Company includes parameters like pH, turbidity, and temperature, and aims to ensure that the water usage is followed up from various departments all the way through to the clients. The information from sensors—found in everything from reservoirs to conservation program subscriptions, to sample reports—helps to find useful patterns in the usage of water to ensure proper water conservation.

4.2 User Requirements:

The user in this case is a water management company who are responsible for implementing water sources, conservation programs, and the use of water by the consumers. Here are some of the materials which the company is expected to include in their storage:

- Different water sources (reservoirs, treatment plants).
- Water conservation programs and customer participation.
- Sensor data monitoring pH, temperature, turbidity, etc.
- Regular water quality reports.
- Customer, department water usage data for water-conservation efforts.

Water management experts suggest an integrated system that starts with real-time sensor data being collated in reports to suggest immediate actions once there are water-quality indicators. Audit of water conservation program compliance is a must, and its impact must be evaluated in respect to the consumer.

4.3 Domain Solution

Glossary of Domain Concepts:

- **WaterSource:** epresents any physical water sources such as ponds, reservoirs, or streams that are monitored for quality.
- **Sensors:** Any devices capable of taking measurements of water quality parameters like pH, turbidity, and temperature.

- **WaterUsageRecords:** The amount of water used per department for various customers.
- **ConservationPrograms:** Plans aimed squarely at reducing water consumption and promoting sustainable practices.
- **Reports:** Documents produced from sensor readings summarizing water quality at different moments.

System Requirements:

- **Sensors Monitoring:** The system records and optimizes water usage across departments and for customers.
- **Water Usage Optimization:** The system also manages and keeps track of the campaign of customer water conservation.
- **Conservation Programs:** Manage and track customer participation in water conservation efforts.
- **Report Generation:** The system generates and saves reports based on sensor data and water quality metrics.
- **User Interface:** A front-end application capable of showcasing real-time data and reports would be meant.

Constraints:

- **Data Accuracy:** Ensuring the sensor data is accurate requires a track to flag discrepancies.
- **Scalability:** The system must be able to scale itself to the future extension of water sources, sensors, and customers.
- **Compliance with Regulations:** The system must comply with regulatory and environmental standards for water quality monitoring.

4.4 MySQL Schema

A comprehensive water management system is supported by many interconnected tables that constitute a relational database. Each table has its attributes and has relations to other tables, ensuring logical consistency and quality with data management. We have used MySQL app instances to create.

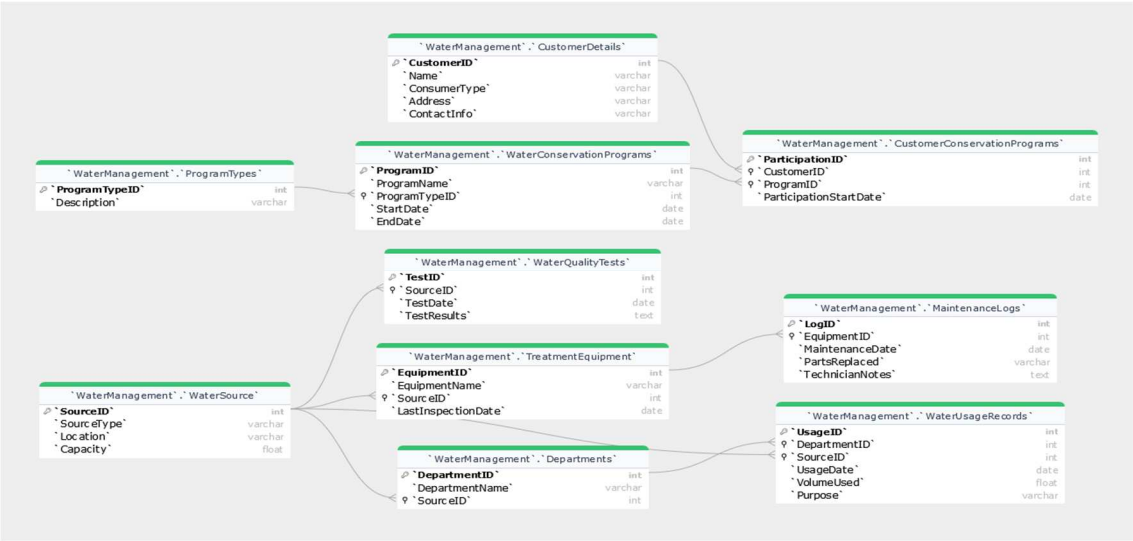

Figure 1: MySQL ER diagram

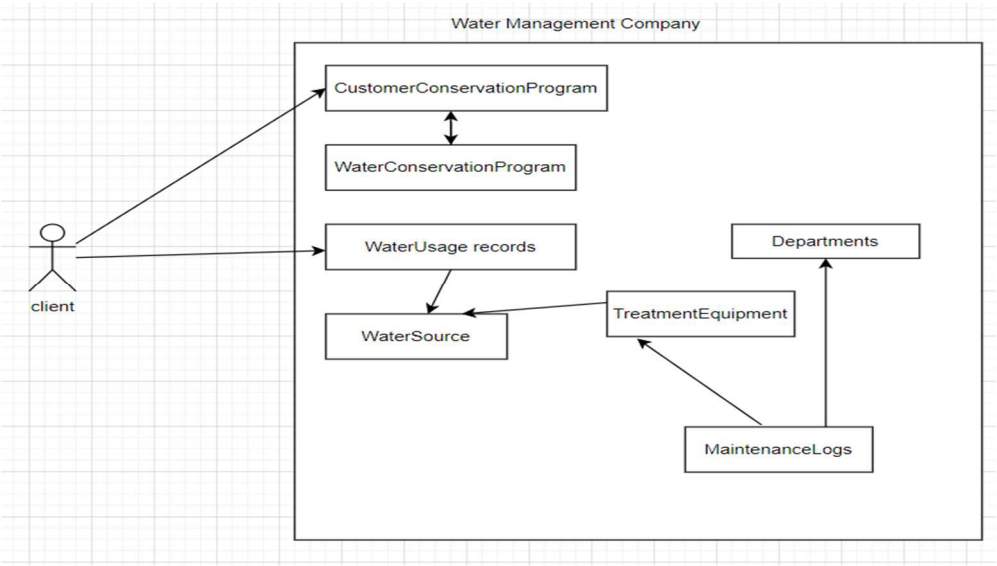

Figure 2: Use Case Diagram of MySQL schema

4.4.1 Problem Domain Addressed:

The restriction of the schema pointed to the Water Management domain and therefore we have given attention to the efficient management and integration of water-related data being an essential recent trend for water management companies. The import is to design an intelligent system that can manage multiple data sources that will grant comprehensive insights (integration) across various dimensions, such as water conservation programs, customers, departments, relationships, and water resources.

Data Integration and Management falls under the problems. Here the challenge is to enable homogeneous access to heterogeneous data sources (SQL databases, NoSQL databases, XML databases, web-based data forms), but keeping each data source to remain autonomous. In this scenario, it represents a schema and an imaginary water resources environment by organizing and linkage of data into various sources in such a way that the queries cancer across them can be processed without distinction.

1. **Multiple Data Sources & Heterogeneity:** The database schema that you have created supports various aspects of water-management systems, for example:

- Various water sources (e.g., rivers, reservoirs, or wells) with details like the source type, location, and capacity are being housed.
- Allows water-conservation programs to record different programs depending on the type and date of the programs.
- Water-usage records, which record the volume of water used by the department, are linked with the department for which it is used and water sources.
- The treatment equipment and maintenance log track the condition and operation of the treatment infrastructure.

These bodies form independent and disparate building blocks of the data. Their pattern of structure so intended is they are designed for connections between each other to bring about data integration. For instance, a **WaterConservationProgram** could be connected to specific Departments and

WaterUsageRecords linked to **WaterSource** and **Departments**. This pattern is quite suitable for the integration of these troubled data sources.

2. **Uniform Access & Querying:** Centralizing this information into a relational database (MySQL) would make this schema a consistent schema for the retrieval of diverse water management data. Then, a query like:

- Extracting water usage for a few departments across time
- Identifying maintenance needs based on inspection dates
- Identifying customers participating in conservation programs
- Viewing treatment equipment linked to certain water sources

can be executed easily, regardless of the source's origin.

3. **Semantic Heterogeneity:** The design really encompasses this clear set of relations (tables), using foreign keys to maintain integrity and consistency over different types of data (i.e., programs, usage, sources, maintenance). Such an arrangement guarantees semantic consistency through data integration with heterogeneous sources, which in turn could facilitate handling data from different formats around systems. We also provide the source's authority, wherein each source, such as **WaterSource** or **Departments**, can evolve alone without harming the relationships

4. **Schema Mappings and Manipulation:** The relational schema is the global representation that can be consistently queried irrespective of the sources of most input data. For an example, when this project is employed to import data from NoSQL databases or web forms, the data fields (e.g., source types or customer data) will be mapped towards those entities in the relational schema, effectuating simple transition and manipulation of the data.

5. **Scalability for Further Data Integration:** The design of this project is successful in handling the current requirements but leaves a provision for all future expansion options. For example, the concluding portion of the project is about integrating different types of data sources, hence any infrastructure may at that time build into the overarching schema while taking nodes in this database structure into a running list of prospective data sources being consolidated for optimum data effusion.

Key Components of this Database Solution:

1. **Glossary of Domain Concepts:**
    - **ProgramTypes**: Groups of programs related to different kinds of water conservation initiatives.
    - **WaterConservationPrograms**: Additional actions related to the ProgramTypes.
    - **WaterSource**: Water from places situated in either bodies of water such as reservoirs or natural sources like rivers or springs.
    - **Departments**: Independent departments within the corporation, using water.
    - **WaterUsageRecords**: Numbers of water consumed within every department, which are grouped across water sources.
    - **TreatmentEquipment**: The equipment used for the treatment of water from different sources.

- **MaintenanceLogs**: Everything that relates to maintenance and all transactions during the repair launches for returning it to normal working conditions.
- **WaterQualityTests**: A vigilant aspect to the source of water for testing requirements to keep the purity of water safe.
- **CustomerDetails**: Important data about customers, their addresses, and points of contact.
- **CustomerConservationPrograms**: Track customer involvement in conservation programs.

2. **System Requirements & Use Cases:** This system will need to:
   - Track and manage water usage across departments.
   - Record and schedule maintenance for treatment equipment.
   - Maintain customer records and link them to conservation programs.
   - Ensure smooth querying across relational data for reports and insights.
   - Integrate data from multiple platforms, enabling uniform access.
3. **Constraints:**
   - The schema must support referential integrity with foreign key constraints.
   - Ensure data can be queried across different platforms (e.g., NoSQL, SQL).
   - The data model should allow for adding additional sources like XML or web-based forms.

4.4.2 Data Population

All data were structured to insert in Water Management. Everything has been structured logically with so much thoughtfulness right from the ordering of the data in the database to inserting it into the relational schema to see that each data was correctly aligned. At ever point during data population, the moderately less-normal tenet of populating the database was taken strictly in series, with a sharp eye on the constraints of foreign keys and the relationship between tables with consideration on supporting referential integrity. It started with **WaterSource** table, which included all kinds of water: reservoirs, rivers, aquifers, natural springs-all with their locations and capacities. Soon, the nature of each **sourceID** values was pulled to be used for inserting in a variety of tables. Departments were then created, one for each water source in this file. Different departments (leak distributions, irrigation, and others) are tied to a water source via **SourceID** so that each department is linked to a water source in the design schema and background practices followed. After this, 1,000 random customer records were generated from **CustomerDetails**. Some details concerning the customer, such as the name, type (Residential, Industrial, Business), address details, and contact details, were created using a method that ensures reasonability in accuracy. To mirror a true-life distribution of **customerIDs** in the data in ways that will permit the company to learn about water usage patterns for further correlation with other studies, conservation marketing, or prompting an official variance, we compiled this into logical blocks: the first block of 400 **customerIDs** for residential, the next 400 for industrial, and all the remaining IDs for businesses.

Next came the **WaterUsageRecords**, with some 5,000 records on water usage across five years in five categories, with a linkage to each record to the respective department and water source. With the usage dates evenly distributed within these five years, various patterns of volume were applied to the design of use so that they were unlike types from both other logically (but then totally consistent) records set. **DepartmentID** and **SourceID** were chosen from filled earlier tables so that the established relationship was maintained. Further comments suggest that **TreatmentEquipment** and **MaintenanceLogs** tables were filled. Records about equipment linked to water sources were created, where every piece of equipment was related to an installed maintenance log via **EquipmentID**. This maintenance log rehearses signature work of an engineer (in various notes and parts changed) and was mainly set by dates and assignments of equipment. **WaterQualityTests**, like **EquipmentMaintenance** or **PaymentOfUtilityBillLogs**, was composed equitably with 1,000 tests, each related to a water source. Obviously, following the test, the test results wisely and precisely are reflecting different possible cases.

So it was, for the tables **WaterConservationPrograms** and **CustomerConservationPrograms**, we created a catalogue of conservation programs culminating in money types such as "*Education*," "*Sustainability*," and "*Research*." Some 1,000 customers took part in the various conservation initiatives to ensure that their participation records were well-integrated about **ProgramID**. In all this, the links between the tables were highly respected. This extended to the use of **SourceID**, **DepartmentID**, **CustomerID**, and **ProgramID** data consistently across the corresponding tables, which in turn ensured the integrity of the relational database. The customer and usage data reached real-world patterns, thus giving rise in the attributes to support real data and grant meaningful collections or analyses in the future.

The pseudocode below holds a sketch of how the system developed in Python populated its main attributes in terms of an algorithm. It focuses on retaining logical consistency across the tables respecting their relationships and ensuring relational integrity.

```
connect_to_mongodb()

-> for each lookup_table in ['OptimizationTypes', 'ContractStatuses', 'CustomerRelevances',
'ProgramTypes'] -> populate_lookup_table(lookup_table)

-> for customer in range(1, 201) -> generate_customer_data(customer) ->
insert_into_db('CustomerDetails', customer_data)

-> for policy in range(1, 51) -> generate_policy_data(policy) -> insert_into_db('PoliciesDetails',
policy_data)

-> for optimization in range(1, 201) -> generate_optimization_data(optimization) ->
insert_into_db('WaterUsageOptimization', optimization_data)

-> for status in range(1, 201) -> generate_contract_status_data(status) ->
insert_into_db('Customer_ContractStatus', status_data)

-> for program in range(1, 21) -> generate_program_data(program) ->
insert_into_db('WaterConservationPrograms', program_data)

-> for participant in range(1, 201) -> generate_participant_data(participant) ->
insert_into_db('ConservationParticipants', participant_data)

-> for policy in range(1, 201) -> generate_contract_policy_data(policy) ->
insert_into_db('ContractPolicies', contract_policy_data)
```

-> print("Database populated successfully!")

CONNECT to MySQL server with credentials and database 'WaterManagement'

-> IF connection is successful THEN

-> CREATE water_sources list with sample data (SourceType, Location, Capacity)

-> for each source in water_sources -> INSERT SourceType, Location, Capacity into WaterSource table

-> COMMIT changes

-> EXECUTE "SELECT SourceID FROM WaterSource" -> STORE SourceIDs in list

-> CREATE departments list (DepartmentName, SourceID)

-> for each department in departments -> INSERT DepartmentName, SourceID into Departments table

-> COMMIT changes

-> EXECUTE "SELECT DepartmentID FROM Departments" -> STORE DepartmentIDs in list

-> CREATE customers list with 1000 customers

-> for i from 1 to 1000

-> SET consumer_type = "Residential" if i <= 400 ELSE if i <= 800 "Industrial" ELSE "Business"

-> SET full_address = "Rua i, CityZone"

-> SET contact_info = "+35191000i"

-> INSERT customer details (Name, ConsumerType, Address, ContactInfo) into CustomerDetails table

-> COMMIT changes

-> EXECUTE "SELECT CustomerID FROM CustomerDetails" -> STORE CustomerIDs in list

-> CREATE usage_records list with 5000 records

-> SET start_date as 5 years ago (Date.today - timedelta(5 years))

-> for i from 1 to 5000

-> SET department_id from DepartmentIDs based on i % number_of_departments

-> SET source_id from SourceIDs based on i % number_of_sources

-> SET usage_date as start_date + (i % 1825) days (spread over 5 years)

-> SET volume_used = 150 + (i % 100) (logical pattern)

-> SET purpose = "Purpose_(i % 5 + 1)"

-> INSERT department_id, source_id, usage_date, volume_used, purpose into WaterUsageRecords table

-> COMMIT changes

-> CREATE treatment_equipment list with 50 pieces of equipment

-> for i from 1 to 50

-> SET equipment_name = "Treatment Equipment i"

-> SET source_id from SourceIDs based on i % number_of_sources

-> SET last_inspection_date as start_date + (i % 365) days

-> INSERT equipment_name, source_id, last_inspection_date into TreatmentEquipment table

-> COMMIT changes

-> EXECUTE "SELECT EquipmentID FROM TreatmentEquipment" -> STORE EquipmentIDs in list

-> CREATE maintenance_logs list with 1000 records

-> for i from 1 to 1000

-> SET equipment_id from EquipmentIDs based on i % number_of_equipment

-> SET maintenance_date as start_date + (i * 3) % 365 days

-> SET parts_replaced = "Part_(i % 5 + 1)"

-> SET technician_notes = "Technician note i"

-> INSERT equipment_id, maintenance_date, parts_replaced, technician_notes into MaintenanceLogs table

-> COMMIT changes

-> CREATE water_quality_tests list with 1000 test records

-> for i from 1 to 1000

-> SET source_id from SourceIDs based on i % number_of_sources

-> SET test_date as start_date + (i % 365) days

-> SET test_results = "Test result (i % 5 + 1)"

```
        -> INSERT source_id, test_date, test_results into WaterQualityTests table
    -> COMMIT changes


    -> CREATE program_types list with 'Education', 'Sustainability', 'Research'
    -> for each program in program_types -> INSERT program description into ProgramTypes table
    -> COMMIT changes
    -> EXECUTE "SELECT ProgramTypeID FROM ProgramTypes" -> STORE ProgramTypeIDs in
list


    -> CREATE water_conservation_programs list with 500 programs
    -> for i from 1 to 500
        -> SET program_name = "Program i"
        -> SET program_type_id from ProgramTypeIDs based on (i - 1) % number_of_program_types
        -> SET start_date = "2024-01-01"
        -> SET end_date = "2025-01-01"
        -> INSERT program_name, program_type_id, start_date, end_date into
WaterConservationPrograms table
    -> COMMIT changes


    -> CREATE conservation_participants list with 1000 participants
    -> for i from 1 to 1000
        -> SET customer_id from CustomerIDs based on (i - 1) % number_of_customers
        -> SET program_id from ProgramIDs based on (i - 1) % number_of_programs
        -> SET participation_start_date = "2024-01-01"
        -> INSERT customer_id, program_id, participation_start_date into
CustomerConservationPrograms table
    -> COMMIT changes
    -> PRINT "Database populated successfully."


-> EXCEPT error -> PRINT error_message
-> FINALLY -> CLOSE database connection
END
```

4.5 MongoDB Schema

The architecture diagram describes the implementation of a Data Integration System for a Water Management Company that uses MongoDB for managing customer data, water conservation programs, and rules. The database solution proposed here is aimed at providing both consistency and referential integrity-unhindered by multiple predefined applications forked over multiple random coherences-in a way to maintain coherent updates across many tables.

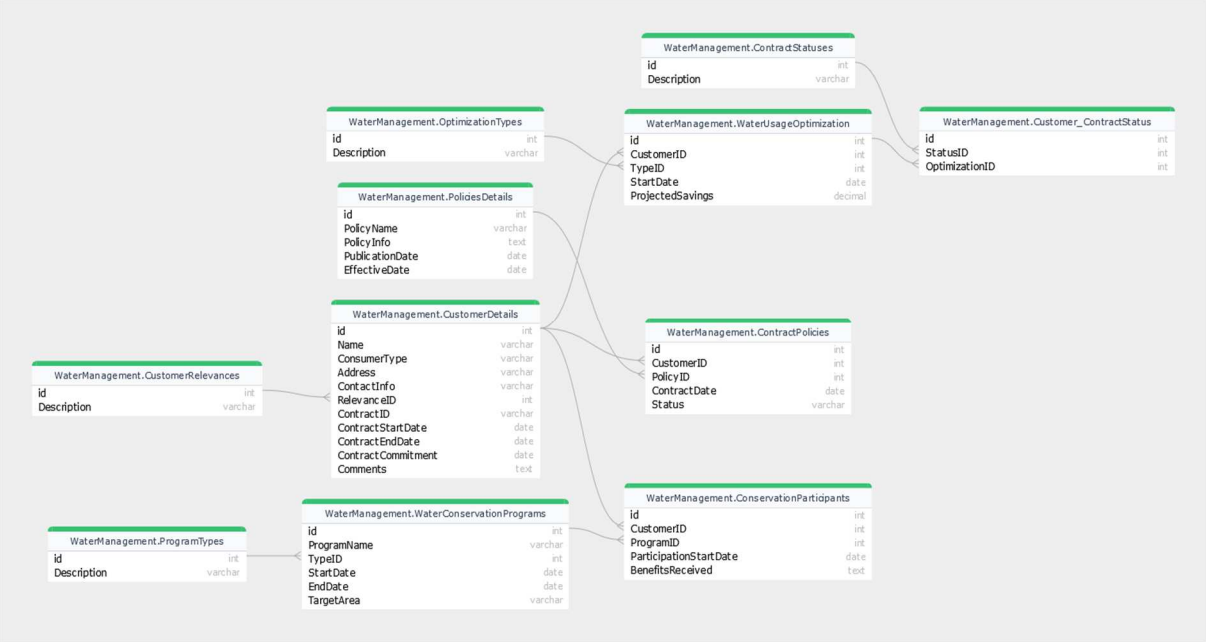

Figure 3: ER diagram of MongoDB database (NoSQL)

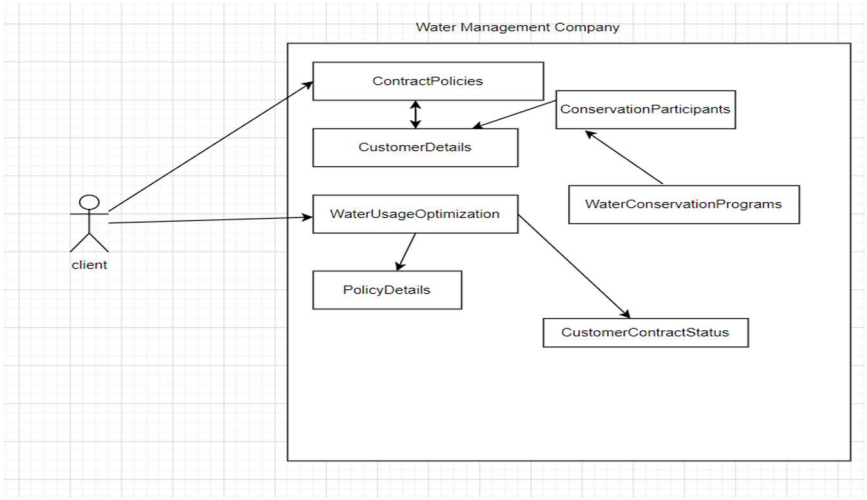

Figure 4: Use case Diagram of MongoDB schema

4.5.1 Problem Domain Addressed

The project's problem domain pertains to the Water Management Industry, with critical junctures around the management of water conservation programs, customer contracts and their related policies. The database solution is proposed to broker the bridging of heterogeneous data that contribute to the different processes of water management.

Glossary of Domain Concepts

- **Types of Optimizations:** Refers to any strategy in favour of water conservation. Included in this are some common strategies to conserve water such as efficient toilets, energy efficiency, and leak detection or some less common strategies such as water recycling.
- **Contract Status:** This status of the different states the customer contracts can be in including active, pending, and cancelled.
- **Customer Relevance:** Gives more meaning to a customer classification with priority ranges, i.e. a customer might have high, medium, or low relevance.
- **Program Types:** The classification of different types of conservation programs for water, e.g., residential, industrial, or business.
- **Policy:** The set of rules, explicitly known as regulations that govern how the customer interacts with the water conservation programs.

System Requirements and Use Cases

The system should support Management of the following:

1. **Customer Data**: Storing detailed information about a customer, including contract status, participation in programs, and program pertinence.
2. **Water Usage Optimization:** Recording and monitoring water-saving efforts and associated cost savings for each customer.
3. **Water Conservation Programs:** The system should be able to handle schemes aimed at various regions and customer segments such as *residential*, *industry*, and *commercial*.
4. **Policies:** Despite the various policies being monitored by the system for customers, management is especially considered regarding the customer's link to water conservation initiatives.

Use Cases:

1. **Query Customer Information:** Customer database should further reveal to all specifics belonging to a peculiar customer, such as their contract methods, optimization efforts in place, or suggestions, along with the categories that they might have joined for conservation campaign.
2. **Track Water Optimization Progress:** Learn of the water optimization efforts-will include figures on water savings and how these are achieved.
3. **Program Participation:** Observing the graph of customer participation within any conservation initiative and programs and the benefits that have accrued as a result.

Constraints

- **Consistency**: Data must always be consistent across tables (that is, for example, if the relevance of a customer is matched qualified and consistent in the **CustomerRelevances** table).
- **Referential Integrity:** Foreign keys must always aim to valid rows within their corresponding tables (e.g., for the relevance of the customer, they would refer to a certain record in the **CustomerRelevances** table).
- **Data Integrity:** Especially with customer contracts, policies, and optimization records, the system must ensure all data integrity is preserved.

4.5.2 Data Population

The population of the database was completed using pre-defined data instead of random data, which helps to ensure consistency across all related tables. The main techniques used were as follows:

1. **Referential Integrity:** In case of inserting data into some tables such as **CustomerDetails**, the foreign key values like **RelevanceID** and **ContractID** were drawn from their corresponding lookup tables (e.g., **CustomerRelevances**, **ContractStatuses**) to sustain the relational consistency.
2. **Date Ranges and Progression:** A strategy consisting of *datetime* and *timedelta* was put into practice to simulate realistic contract starting dates and ending dates, beginning of optimization and participation timelines to ensure a cohesive data progression of time periods.
3. **Patterns Instead of Randomness:** We avoided to use any *random*() to populate any tables. Thus, the data followed structured patterns all along, adding quality and value to this generated data.

Given here is the pseudocode for populating the MongoDB:

```
connect_to_mongodb()


-> Create and populate lookup tables
   -> FOR each lookup_table in ['OptimizationTypes', 'ContractStatuses', 'CustomerRelevances',
'ProgramTypes']:
      -> populate_lookup_table(lookup_table)


-> Populate CustomerDetails table
   -> FOR customer in range(1, 201):
      -> customer_data = generate_customer_data(customer)
      -> insert_into_db('CustomerDetails', customer_data)


-> Populate PoliciesDetails table
   -> FOR policy in range(1, 51):
      -> policy_data = generate_policy_data(policy)
```

```
        -> insert_into_db('PoliciesDetails', policy_data)


-> Populate WaterUsageOptimization table
    -> FOR optimization in range(1, 201):
        -> optimization_data = generate_optimization_data(optimization)
        -> insert_into_db('WaterUsageOptimization', optimization_data)


-> Populate ContractStatuses table
    -> FOR status in range(1, 201):
        -> status_data = generate_contract_status_data(status)
        -> insert_into_db('Customer_ContractStatus', status_data)


-> Populate WaterConservationPrograms table
    -> FOR program in range(1, 21):
        -> program_data = generate_program_data(program)
        -> insert_into_db('WaterConservationPrograms', program_data)


-> Populate ConservationParticipants table
    -> FOR participant in range(1, 201):
        -> participant_data = generate_participant_data(participant)
        -> insert_into_db('ConservationParticipants', participant_data)


-> Populate ContractPolicies table
    -> FOR policy in range(1, 201):
        -> contract_policy_data = generate_contract_policy_data(policy)
        -> insert_into_db('ContractPolicies', contract_policy_data)


-> Final message
    -> print("Database populated successfully!")
```

4.6 Web-based Data Source

This schema is intended for a system that monitors and gathers environmental parameters (e.g., pH, temperature, turbidity) with sensor devices in reservoirs. Data is set up to give details of tracking,

clearance, and reporting of such parameters. The ER Diagram equivalent of this is displayed in Figure 5 and use case diagram in Figure6 given below:

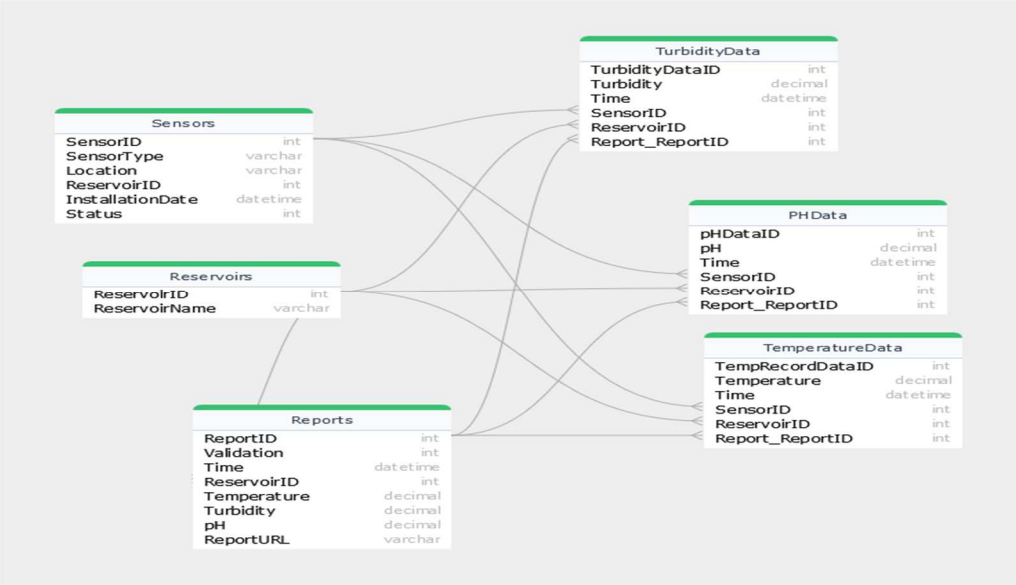

Figure 5: ER model diagram of website Data source

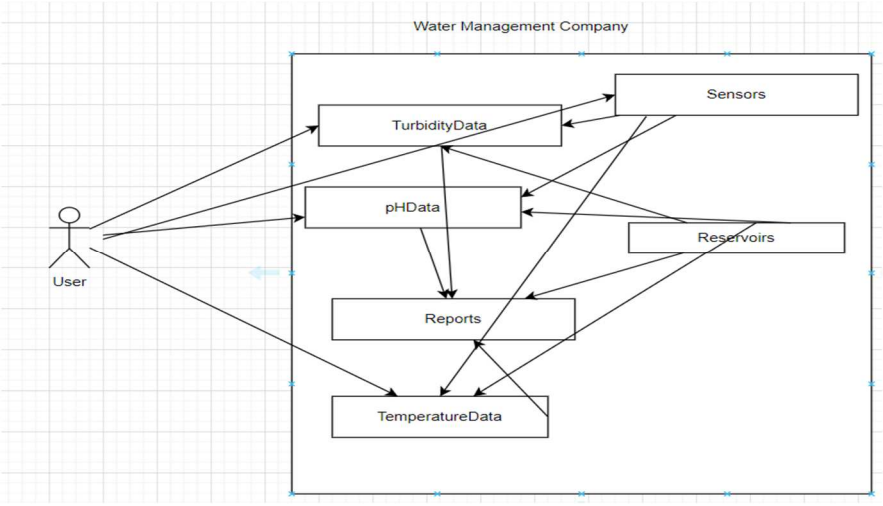

Figure 6: Use Case diagram

4.6.1 Problem Domain Addressing

Predominant problem areas in this project are concerned with gathering and integrating environmental data from various reservoirs. The data on parameters of water quality management in the reservoirs, namely temperature, pH levels, turbidity, and sensors readings, are also essential. It proposes a system to collect and integrate this data in heterogeneous formats (like JSON) and from various sources such as sensor readings and reports to present these to the users in a consumable manner.

Key Components of the Database Solution:

1. **Glossary of Domain Concepts:**
    - **Sensors:** Devices capable of measuring certain environmental parameters (temperature, turbidity, pH) within the reservoirs.

- **Reservoir:** Water storage privilege to house several installation sensors.
- **Reports:** Documents giving the status and findings of various environmental parameters measured within either reservoir. This comprises aspects like validation and related metrics.
- **pH, Temperature, and Turbidity Data:** Sensor readings regarding water acidity, temperature and turbidity within the reservoir.

2. **System Requirements & Use Cases:**
   - **Data Collection & Integration:** The System Should fetch and view data from JSON files (from sensors, reports, and reservoir parameters).
   - **Data Querying:** In the UI, a user would be able to see his sensor and report data in an easily readable format in tabular form covering pH, temperature, and turbidity.
   - **Reporting & Validation:** Reports involving data validation checks, including the temperature, turbidity, and pH levels, would witness the effectiveness of the water quality.
   - **User Interface (UI):** A very user-friendly web interface making everything around integrated buttons for fetching data from different categories for the user to work on Go.

Use Cases:

- A User can view sensor data across different reservoirs.
- A User can view water quality reports such as *pH* value, *temperature*, *turbidity, etc* and may opt for different dates.
- A user can filter and retrieve the data based on any specific reservoir and sensor type.

Constraints:

- **Data Consistency:** The data collected from various sensors and various reports must maintain consistency, and the relationship between sensors, reports, and reservoirs must be preserved correctly.
- **Heterogeneity of Sources:** The system will take in data from various external sources of data, including a series of JSON files with different structured forms which requires careful mapping of the different formats (relational vs nonrelational data).
- **Data Accessibility:** It is important to ensure that all data points (*pH* value, *turbidity*, *temperature*, *sensor* details, and *reports*) are available and linked up properly within the system.

4.6.2 Data Population

Data from several JSON files, each containing records about various environmental items, is used to fill the database, with particular data distributed into an HTML table. These JSON files are structured in the manner of records for the tables in the database-sensors, reports, temperature, pH, and turbidity-

that hold the most important fields each record requires. The JSON files are acted upon as AJAX calls that load the data asynchronously and thereby provide a pretty responsive user interface.

Whenever the random data is concerned, it is not employed-simply because all values are referred back from the particular JSON files. For example, pH levels for a container must refer to the respective sensor ID, container ID, and report ID, which allows scanning for a possible amount of integrity for each data point right from its source. This relational and consistent mode of working is key to data integrity and the availability of meaningful data.

The previous section already provided an outline of the technology of the website, if you will, including a pseudocode.

5. Waste Management company

5.1 Domain Problem

On its back, this company is losing sleep over the management of data emanating from diverse sources as it pertains to both water and waste management in Greater Algarve. In fact, companies like *Waste Management, Inc., Veolia, Suez, Recology* etc employ such waste management model that is "Pay-As-You-Go," which implies that the customers are charged based on waste collection activities. This means that diverse types of data accrue for management purposes, such as client information, contract details, waste collection schedules, billing details, and governing regulations among others taken together. In their storage and management come different systems including an SQL database-PostgreSQL and a NoSQL database-Apache CASSANDRA and an XML database-BaseX-all having their patterns and schemas of access. All good and fine-one takes them as they are without any transformation-access, query.

Access is near impossible to integrate data from these heterogeneous systems without changing their autonomy. The integration system will bring a unified view up for the customer relationship, service contracts, billing, waste collection scheduling, and complaints. This means that a significant part of the goal for which this project is being carried out is to tackle the defended data heterogeneity-format, schema, semantics-and autonomy-the independence of the data sources. The model will also have to cater for data in an efficient and correct manner to guarantee its transformation and query despite all these heterogeneities in the sources.

5.2 User Requirements

The beneficiaries of this system would include all interested parties such as customers and the administrators and management teams. The specific ones for functional groups are as follows:

1. **Customers:** For complete access to the waste collection schedule, waste disposal quantities, billing details, module for complaining, or generating complaint tickets with respect to logged time, and masking their personal information in 'My Account' settings. Real-time billing based on the exact volumetric disposal needs and therefore demanding the system to always serve the most correct and newest data of that customer.

2. **Administrators**: Track contracts with thousands of customers with cleaning as per the agreement, update service plans, and monitor billing cycles. The key task within this hierarch is to guarantee that the system accommodates the sharing of data properly among various locations, ensuring that the information is not held in mini dynamics but synchronized. They will also have to ensure that the system automatically supports all legal and regulatory policies anywhere, for either prominently managing complaints or breeched contracts.

3. **Management**: Management needs to monitor operation efficiency and generate reports, which, again, must comprise waste collection efficiency, billing trends, and customer complaints. Nearly in parallel lies the task of verifying that the already recommended legal instruments are strictly adhered to by all departments, relying on absolute integration and reporting from the very many data sources.

5.3 Domain Solution

A solution would be to centralize access to data held on the three different platforms (PostgreSQL-SQL database, Apache Cassandra-NoSQL database, BaseX-XML database) by creating a virtual data integration system. This system therefore takes the schema mappings and transformations path to convert the data from all sources into a single global representation, as a result allowing any single user to log in and make a query for any information from such.

Glossary of Domain Concepts

- **Client:** Any customers of the organization in the business of waste management and identified as someone to whom waste cleanup service is provided.
- **Contract:** Protocols articulated in the agreement between the client and the company regarding the terms of their service, including specifics such as the frequency of garbage collection, the type of waste, and how billing should be carried out.
- **Billing:** A complete record prepared for a waste customer based on the activities in waste disposal.
- **Waste Categories:** Categories for review vary by waste kinds, including but not limited to residential and industrial.
- **Collection Frequency:** Frequency at which waste is picked up from a client-labour-saving collection service (e.g., daily, weekly, bi-weekly).
- **Regulatory Policies:** Statutory limits governing the waste management process in the field.
- **Complaint:** Any intimation from customers to the office for billing discrepancies or any other issues about an on-going service.
- **Contract Breach:** An issue in which a client contravenes the terms' covenants contained within their waste management procedures.

System Requirements

Specific cardinal pre-requisites are essential for this system to be viable:

- **Data Integration**: The desired system should effectively incorporate data from both PostgreSQL, Apache Cassandra, and BaseX databases, thereby bridging seamless search between various data sources.

- **Real-Time Data Access**: The system should enable real-time data processing of waste disposal and billing data so that customers are charged fairly on a basis that reflects authentic utilization.
- **User-Friendly Interface**: The system, modified explicitly for a customer-friendly interface, would have been easy to use both on behalf of administrators, to get the first view of data and manage it accordingly.
- **Scalability**: The system needed to support a vast influx of customers and data records in the wake of business expansion.
- **Compliance with Regulations**: The system followed regulatory practices to regulate company and law bounds throughout the land.

5.4 Constraints

- **Semantics Heterogeneity**: SQL, NOSQL, and XML Database have diverse ways of organizing & representing data, creating the biggest challenges in merging these diverse databases. System should incorporate differences in structure and semantics.
- **Data Volume and Performance:** Data, with hundreds of clients, contracts, and waste collections, must perform with high efficiency during the communication process across three sources. Consistency and performance of data validity are thus very important to design.
- **Schema Mapping and Transformation:** The data that is to be integrated (PostgreSQL, Apache Cassandra, and BaseX) need to go through schema mapping and conversion rules for the data to become a consolidated format. Doing this is of utmost importance for data consistency and correctness
- **Security and Privacy:** Sensitive client data-level details and billing information need to be secured in storage, transmission, and access that do not violate data privacy regulations and demand that access control to be implemented.
- **Data Consistency:** Data consistency is important when the system integrates data from various sources. Differing information or stale information might bring about wrong bills or complaints from customers.

5.5 PostgreSQL Schema

Efficient management of a region's waste is a complex task that requires careful monitoring of waste production, collection, disposal, and billing processes. The aim is to set up a centre located database system that can cater to diverse client demands, industrial to commercial or residential, and conforming to environmental laws, while ensuring the entire operation to be operational. Figures 7 and 8 show the ER diagram and use case diagram, respectively, of PostgreSQL database for a waste management business.

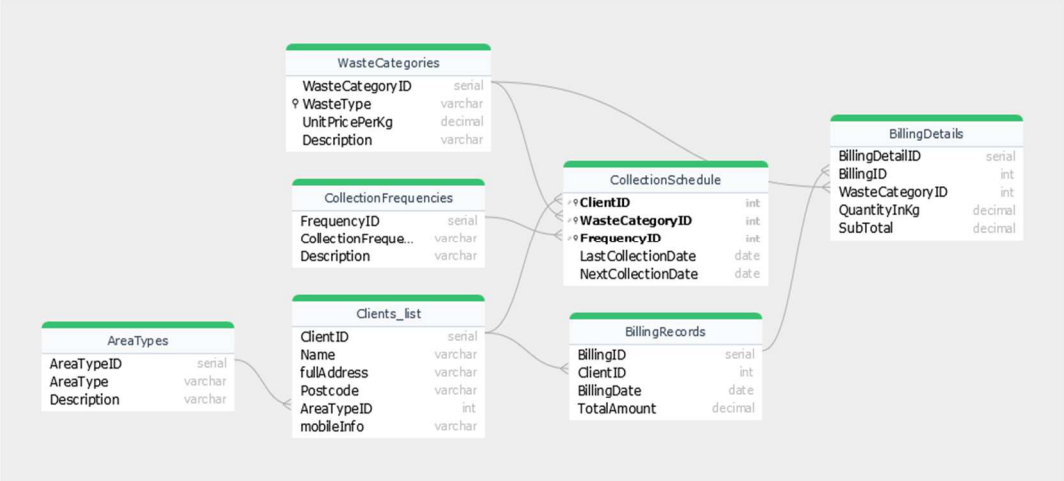

Figure 7: ER diagram for PostgreSQL database

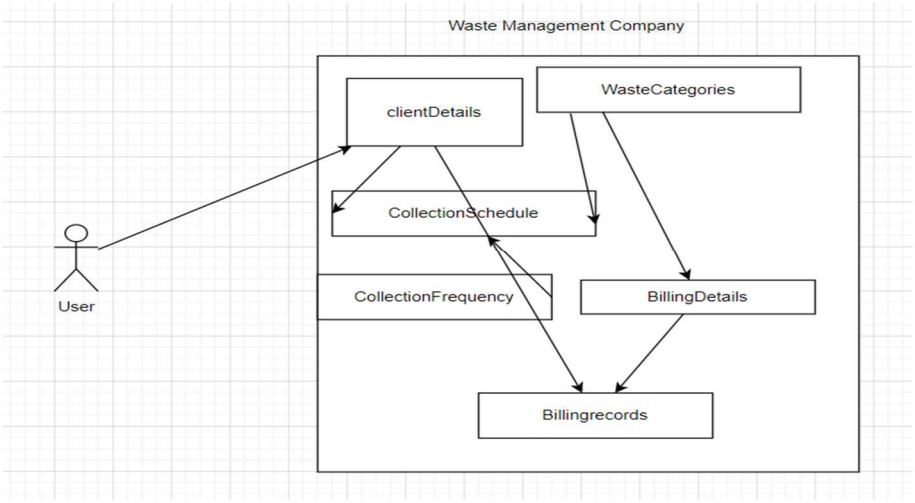

Figure 8: Use case Diagram

5.5.1 Problem Domain Addressed

The problem domain addressed in this project is to manage and integrate multiple data sources for a waste management company that operates through giving its services to the Greater Algarve area in Portugal, under the "Pay-As-You-Go" billing model. The company is dealing with diverse data sources, such as SQL databases (PostgreSQL), NoSQL databases (Apache Cassandra), and XML databases (BaseX). These data sources are maintained and designed differently, thereby causing problems for the integration. The purpose is also to have a uniform interface that integrates these multiple heterogeneous sources truly, thereby allowing various users, including customers, admins, and management teams, to seek knowledge in a uniform manner regardless of the data source.

Key Issues:

1. **Multiple Data Sources & Heterogeneity:** The main impediment arises out of the differences in data sources and bases in the types of structure in which data are stored. Each distinct data store with separate database repetitions (SQL, NoSQL, and XML, in this case) is created. For example, PostgreSQL saves its data in a relational table with a reference schema, Apache Cassandra saves it in a wide-column format, and BaseX saves its data format to the XML

standard. Since the data is stored differently, with the data formats indirectly pointing to inquiry or insertion, their interoperation proves to be a greater ordeal.

2. **Uniform Access & Querying:** It is essential for the implementation to generate uniform access to the different data sources through a single-facing tool where users can express on-the-fly queries and retrieve data from any source without bothering about technical specifications. This means translating data formats into a common query schema for easy querying from different sources.

3. **Semantic Heterogeneity:** Different databases understand data in different ways, i.e., what already defines data and how it is understood while the same attribute name in PostgreSQL may mean something completely different than in Cassandra. In a nutshell, it would now be necessary for the system to come up with ways to impose a unique interpretation across all the data sources; this task is necessary even if they use different terminologies. The task is, in essence, significant for the establishment of one common code.

4. **Schema Mappings and Manipulation:** For managing data heterogeneity and semantics, the system uses methods such as schema mapping and data transformation to transform the data of all sources to a global schema, which would enable users mostly to query the data as if it were coming from one single unified source, e.g., both schedules of waste collection in PostgreSQL and in XML need to be transformed to accommodate comparison and combining.

Key Components of this Database Solution

1. **Glossary of Domain Concepts:**
    - **Client:** The customer receiving environmental services, tracked by the company.
    - **Contract:** It is the deal made between the company and a client setting down terms of waste disposal service.
    - **Waste Categories:** Various categories of waste that may be disposed of, e.g., domestic waste, industrial waste, etc.
    - **Billing Record:** Detailing the note of amounts due from a client on grounds of waste disposal.
    - **Collection Schedule:** Timetable specifying when waste is to be collected and how frequently it is to be collected for a particular client.
    - **Waste Disposal:** Waste disposal is tracked with respect to quantity and type of waste.
    - **Regulatory Policy:** Set of legal rules governing the disposal and recycling of waste systems
2. **System Requirements & Use Cases:** A system designer should fulfil the following requirements:
    - **Data Integration:** The three data ports of the database systems need to seamlessly integrate data in real-time so that the data is consistent and updated.
    - **Real-Time Queries:** The integrated data concern a simple query made available to an individual while the data from a source goes through a customer piece and one with BaseX in service logs.

- **Security:** Sensitive information concerning customers and billing details such as the authentication and role access for, e.g. customers, administrators, is to make the authenticated data more highly secure, which dictates the requirement of restricting access to itself.
- **Scalability:** The system should be able to extend itself while new sources of information are added on without stressing it to be reconstructed from scratch.

Use Cases:

- **Customer View:** Once observed by the customer who views his/her waste pickup schedule or checks their billing history and complains about any problem of complaints.
- **Administrator View:** From this particular view, the administrator is allowed to modify contract terms, amend billing details, and then work towards resolving customer complaints.
- **Management View:** Now management is basically checking the view on how efficiently the waste collection system is operating, billing behaviour, and regulatory restrictions in detail.

Constraints:

There are many technical issues which we can address hopefully after implementation:

- **Semantics Heterogeneity:** Data in different databases may have different meanings for similar terms – e.g., "waste category" or "client's status"-- thus making it quite difficult to directly compare and/or join them. The system must, therefore, comprehend how to map such terms to ensure the consistency across the data sources.
- **Performance:** The system is going to work with hundreds or even thousands of clients and billings. To process information in big chunks and maintain overall query response time quickly in an elaborated fashion, you'll need this kind of system rather than some other means to source in that volume of data.
- **Data Integrity:** It may be useful, but the system must guarantee the accuracy and consistency of data collected from different sources. Inconsistencies amid databases (such as missing data or out-of-date information) must be immediately flagged for attention.
- **Complexity of Queries:** Users might pose more complex questions that would span across all data sources (e.g., "Show Me the billing records for clients in certain areas who have exceeded waste disposal limit"). The system should be able to handle such sophisticated queries well.

5.5.2 Data Population

In positing data for the PostgreSQL database on the waste management project, we resisted random data generation techniques (e.g., random ()) instead bounding the data logistically to ensure the relational integrity of the database, satisfying all constraints from the system, and assuring logical integrity for all the data across the tables.

The purpose of the data population was to sustain the relationships between different tables, incorporating possible actions such as tying clients to their related area types, assigning the right waste categories to various schedule types, and maintaining the right connection between waste-disposal entries and these schedules. An ordered system took the responsibility of clearly annotating and performing anything pertinent to data in the database to reference them to other related actions. Thereby, this controlled setting ensured clients could only be given one specific **clientID** in the **Clients_list** table with some knowledge of their area allowed to be inserted in the AreaTypeid field. This much-needed uniformity was ensured by this condition, and using a random index number was not an option.

The generation of any random data should never happen due to the presence of constraints that have been defined as part of a normal system. With any random data, these constraints could have easily been compromised, thus compromising the database relationship aspects of data integrity and consistency. This may have compromised the application of database insertion due to incorrect foreign key relationships between all tables like **Client_list**, **CollectionSchedule**, and **WasteDisposal**. One of the rare examples of HOLV bypass and risk can be seen at random data insertion time as the **Members_list** and _list_of_posts might be grouped with the wrong information. The tables were subsequently populated logically which ensured that each **ClientID** in the **Clients_list** had a valid **AreaTypeID**, while each waste disposal record had a corresponding valid **ClientID**, **WasteCategoryID**, and **FrequencyID**.

Additionally, the serialized approach made it easier for me to manage and debug the data population process. Customer records were logically split into three categories, namely Residential, Industrial, and Business, each to be accommodated with the relevant attributes like **Postcode**, **MobileInfo**, and **AreaTypeID**. Doing so ensures that the relations in the software are simple and understandable, strong, and systematic. This ensures meaningful connections to the random data (relationships supported by correct data), enabling the production of error messages or incorrect results of an unsuccessful query.

Choosing the correct order for data population preserves the quality and integrity of the relational database. It ensured that data followed system constraints and kept the relationships between different tables intact. The method is essential to avoid inconsistencies and inaccuracies in the database, which is crucial for the waste management system's integrity since precise customer, waste category, and collection schedule tracking is required.

Here is the pseudo code of the data population:

```
1. Start the database population process
2. Retrieve valid AreaTypeIDs from the database
   -> EXECUTE "SELECT AreaTypeID FROM AreaTypes"
3. Retrieve valid FrequencyIDs from the database
   -> EXECUTE "SELECT FrequencyID FROM CollectionFrequencies"
4. Populate the WasteCategories table with predefined categories
   -> INSERT predefined WasteCategories into WasteCategories table
5. Retrieve valid WasteCategoryIDs from the database
   -> EXECUTE "SELECT WasteCategoryID FROM WasteCategories"
6. Populate the Clients_list table with customers based on logical criteria:
```

-> FOR each customer in range (1, 201):

a. Assign customers to Residential, Industrial, or Business area types

b. Generate consistent Postcodes and MobileInfo based on customer type

-> INSERT customer data (Name, fullAddress, Postcode, AreaTypeID, mobileInfo) into Clients_list table

7. Populate the CollectionSchedule table with collection schedules for each client:

-> FOR each client_id in valid ClientIDs:

a. Assign collection frequencies based on client types (Daily, Weekly, Bi-Weekly)

b. Calculate Last and Next Collection Dates based on current date and client ID

-> INSERT collection schedule (ClientID, WasteCategoryID, FrequencyID, LastCollectionDate, NextCollectionDate) into CollectionSchedule table

8. Populate the WasteDisposal table with disposal records for each collection schedule:

-> FOR each schedule in valid CollectionSchedules:

a. Generate waste disposal records linked to CollectionSchedule entries

b. Ensure quantity and disposal dates are logically consistent

-> INSERT disposal records (ClientID, WasteCategoryID, FrequencyID, DisposalDate, QuantityInKg) into WasteDisposal table

9. Populate the BillingRecords table with billing information for each client:

-> FOR each client_id in valid ClientIDs:

a. Calculate total amounts based on waste categories and quantities

-> INSERT billing record (ClientID, BillingDate, TotalAmount) into BillingRecords table

10. Populate the BillingDetails table with detailed billing information:

-> FOR each billing_id in valid BillingIDs:

a. Link each BillingRecord to its corresponding waste categories and quantities

-> INSERT billing details (BillingID, WasteCategoryID, QuantityInKg, SubTotal) into BillingDetails table

11. Commit the data to the database

-> COMMIT all changes to the database

12. End the population process

5.6 Apache Cassandra Schema

Supporting multiple clients, contracts, and compliance with loads of regulations in the waste-management industry, we believed this to exist as a preconceived condition. The following ER diagram-based issues are handled worthily in-cassandra-schema-supported deployment.

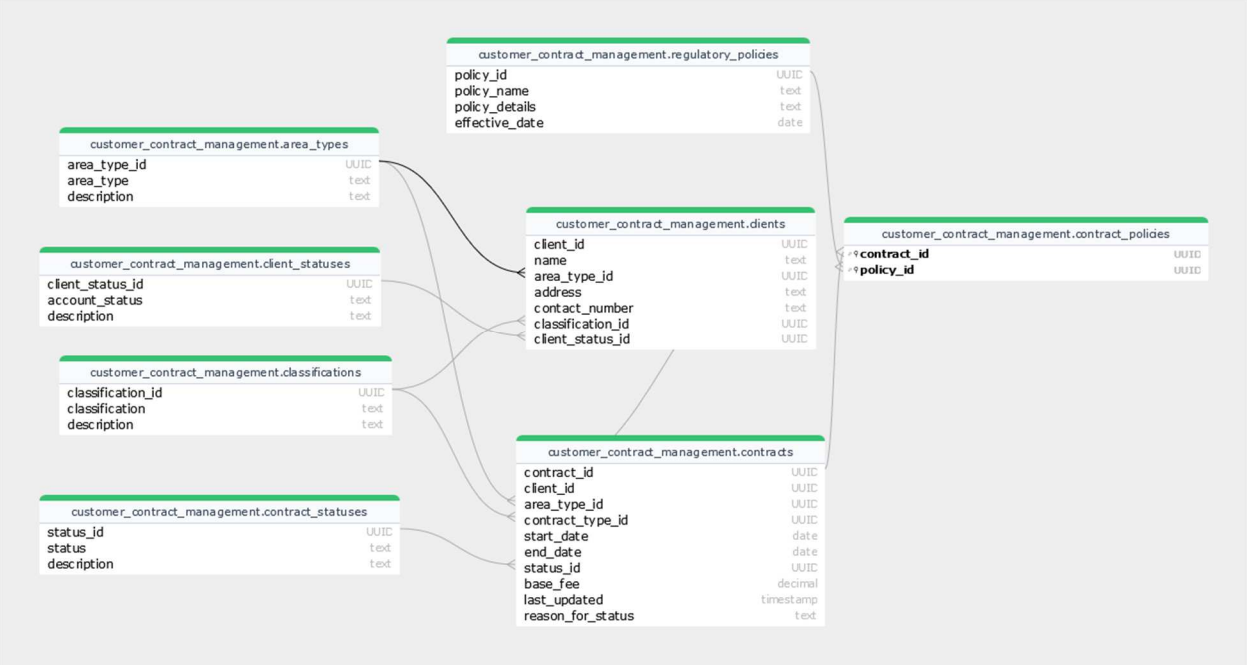

Figure 9: ER diagram of Apache Cassandra database

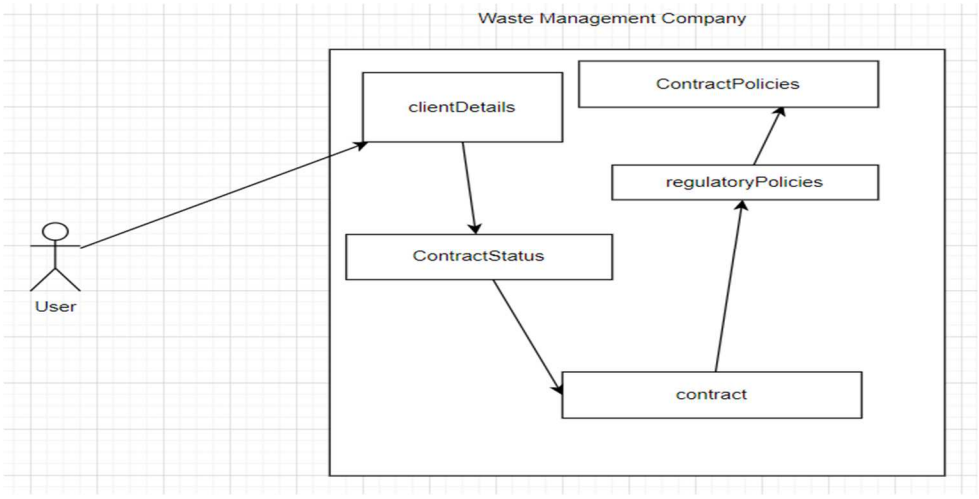

Figure 10: Use Case diagram

5.6.1 Problem Domain Addressed

There are several opportunities to manage and integrate data from myriad sources in a waste management business-as outline in this project, here concentrating on "Pay-As-You-Go," with managing the data seamlessly across the plethora of platforms such as SQL databases, NoSQL systems, XML databases, and web forms. The major issue in this case is concerning diverse nature of sources of data. Different sources often hold and represent data in different formats and hence, obtaining them in uniformity for querying has been a bit of a challenge. Heterogeneous semantics are one other major requirement to facilitate data integration. For instance, while data could be accessed from numerous sources, its meaning or representation could vary (e.g., the same entity might have different names in

different databases). The integration system is expected to be able to transparently accesses and queries these heterogeneous sources, thereby presenting a consolidated view of data. The system has the native capabilities to manage schema mappings and manipulations so that the data from different schemas could be converted into one unified global representation. Lastly, scalability is also of prime requirement so that the system would be easy to add more and more data sources as the organization continues to grow.

Key Components of this Database Solution

1. **Glossary of Domain Concepts:** Various concepts under waste management needing clear definitions out there for integration and querying. Basic terminologies comprise the likes of Clients (the customers within the system); Waste Categories, which group the various waste types such as paper, plastics, organic waste; Collection Frequencies which define how often waste has been collected from clients (daily, weekly, and bi-weekly); Billing Records which keep track of clients' payments based on their waste-disposal activities. The **AreaTypes** table further classifies different regions like Residential, Industrial, and Business for categorizing clients.

2. **System Requirements & Use Cases**: There are many systems that need to meet certain functional and non-functional requirements. Functional requirements like managing client information, tracking waste disposal schedules, categorizing types of wastes, calculating customer bills as per weight, and generating reports for clients and internal governance are included. Non-functional requirements include system performance, ensuring that queries and data retrieval across multiple platforms (SQL, NoSQL, and XML) can be processed efficiently. A use case might involve a client interacting with the system to request waste collection services, view past disposal schedules, or receive a bill for services rendered. Another use case could be for the system administrator to add new waste categories or adjust collection frequencies.

3. **Constraints:** Quite a few constraints need to be managed within the system. One amongst them is semantic heterogeneity amidst given sources—different sources may maintain waste categories or client data in various formats, hence, requiring mapping and transformation according to the standard format. Another constraint would be the schema mappings and manipulation for the conversion of source schemas into a unified global representation that would make it easier for querying. There is also a constraint of scalability, as the business model in consideration is continuously growing its services (e.g., more regions or client additions), which demands the Data Integration system to scale easily. The Integration system should accommodate new data sources without any agitation to pre-existing data flows or to existing queries.

Addressing Multiple Data Sources & Heterogeneity

This involves integrating several data sources that may be placed in a handful of formats and databases (SQL for structured data, NoSQL for semi-structured data, and XML for configuration or external data). Data sources would normally spread out in many places such as PostgreSQL relational database for structured data, NoSQL data store in MongoDB, web forms in which input is fed by end-users. The handling of the heterogeneity of the system needs a mediator or integrating platform for access to these disparate sources of data.

Schema mappings are the cornerstone of uniform presentation of the data source in the F out of the combined schema. This allows the system to put data readings together from different sources. For instance, the individual entities that are the client ID in the SQL database need to be associated via a mapping to the corresponding client ID residing in the NoSQL database. Hence, while the very same name, in cases where different systems represent the values in different kinds of formats, it is critical to convert the data into the relational form used by the company.

Schema Mappings & Data Conversion

Data conversion is ultimately about transforming the schemas of each of our data sources (SQL, NoSQL, XML) into a unified representation to enable querying. For example, the waste category information is stored in PostgreSQL in the **WasteCategories** table while they are stored, more or less, in an unstructured or semi-structured format in NoSQL data stores like MongoDB. The data integration system should be able to convert one of these types of schemes, enabling waste data from any source to be queried following a uniform format across various platforms. This kind of transformation may include converting something like unit prices or waste category names to the same format, which is vital to ensuring that queries from different databases could eventually return the same information.

Query Processing in Data Integration

Once the integration and mapping of the data sources occur, the database system must have considerations for query processing that will be used to recover, mash up, and combine data from those sources. Query containment involves ensuring that queries across the integrated database system should return the correct and complete results regarding the data that may be stored in different formats for different sources. For instance, in order to retrieve the total waste disposal amount for a client from PostgreSQL, based on the **CollectionSchedule** and **WasteDisposal** tables, data could be further needed from some customer-related dataset in MongoDB. With data views, one can consider isolating different data with unique properties from different sources to appear as a single database being queried.

Scalability for Further Data Integration

The waste management system needs to seamlessly adapt to the addition of further datasets. Growth in terms of business will translate to more clients, waste categories, or regions and waste types besides the currently envisaged data sources. The integration system should be extensible, allowing for the integration of data from further sources. By masterminding such a flexible system, the waste management company desires to have the data from many sources plus maintain, via efficient processing and queries, a well-designed and group of efficient databases. Flexibility of this kind offers scalability that is able to support the growth of the business by itself without losing its analytical functions as well as good response to the queries.

5.6.2 Data Population

During the data population phase for the waste management system, we purposely omitted the practice of using random data generation (as done by, for instance, using the random () function) when writing test data. Rather, we allowed a more logical order for overseeing the relational database across all databases. Choice in question is vital for a number of reasons, the main being:

1. **Ensuring Relational Integrity:** As one of the main principles in relational database design, it presupposes that some serious measures ought to be taken to make sure that data is consistent and logically interlinked in all databases. By using a logical order for populating the data, we thus ensured that value of foreign key relationships between various tables (such as clients,

contracts, and client_status) was still intact and correctly mapped. For example, when populating the table clients, Area_Type containing a distinction of Residential, Industrial, Business was assigned logically to clients by predefined ranges (i.e., if the first 67 clients are residential, 67 industrials, and so forth). This is done to support consistency and ensure that each client_id always points to reliable and consistent data in tables closely related to clients, say, contracts, client_statuses, etc.

2. **Following Business Logic:** In a real-world scenario, the data is not random but follows certain business rules. For instance, clients are assigned classifications (*Premium*, *General*, *Discount*) based on their area type or client type. By following this logical approach, the generated data mimics the actual structure and logic that would be used in a real system. This makes the data much more meaningful and usable for the purpose of testing or simulating real operations.

3. **Maintaining System Constraints:** Organizing the data with a specified logical order while enabling the replication of system constraints that have existed during database design also facilitates proper data management; basically, constraints like an exclusive primary key for the client client_id and contract contract_id-referencing foreign keys should be better managed into the database through suitable data insertions. Apparently, this order is necessary because for a group of data to be saved to the table one by one, its "conforms" to the "correct" logical order of insertion into the table; affecting UserId and ContractId with independent data would have created conflicts and produce null or erroneous entry violating constraints and causing insertions to fail or integrity to substrates to be breached.

4. **Simplifying Data Management:** More than having a data management approach, this thinking will greatly facilitate data troubleshooting. It always remains easier to locate anguish in the logical and ascending order of stored data than in randomly chosen data, because the former reveals a sequence of information in a case log that can be used, among others, to diagnose the root of the trouble. Should there be trouble with contract data, such as unresolved client_id or contract_id mismatches, by knowing the order of data insertion, the investigation can be slightly less painful and shorter.

Generalizing the argument, if all randomness and insertion were done in an orderly manner, we would ensure data integrity in a relational database, maintain impossible system constraints, and simulate real-world behaviour for achievement of a significantly higher quality, consistency desired on all platforms involved in the integration, and for any data contained areas.

The following pseudo-code should provide an idea on database population:

```
PopulateDatabase()
{
  // Connect to Cassandra cluster
  ConnectToCassandra()

  // Clear existing data
  Truncate(area_types)
  Truncate(classifications)
  Truncate(client_statuses)
  Truncate(contract_statuses)
  Truncate(clients)
  Truncate(contracts)
```

```
    Truncate(regulatory_policies)
    Truncate(contract_policies)

    // Insert area types logically
    For each area_type in area_types
        InsertIntoAreaTypes(area_type_id, area_type, description)

    // Create area type mapping
    CreateMapping(area_type_map)

    // Insert classifications logically
    For each classification in classifications
        InsertIntoClassifications(classification_id, classification, description)

    // Create classification mapping
    CreateMapping(classification_map)

    // Insert client statuses logically
    For each client_status in client_statuses
        InsertIntoClientStatuses(client_status_id, account_status, description)

    // Create client status mapping
    CreateMapping(client_status_map)

    // Insert contract statuses logically
    For each contract_status in contract_statuses
        InsertIntoContractStatuses(status_id, status, description)

    // Create contract status mapping
    CreateMapping(contract_status_map)

    // Insert clients logically based on area types and classifications
    For each client in clients
        AssignLogicalValues(client)
        InsertIntoClients(client_id, name, area_type_id, address, contact_number, classification_id,
client_status_id)

    // Insert contracts logically based on client IDs
    For each contract in contracts
        InsertIntoContracts(contract_id, client_id, area_type_id, classification_id, start_date, end_date,
status_id)

    // Insert regulatory policies logically
    For each policy in policies
        InsertIntoRegulatoryPolicies(policy_id, policy_name, policy_details, effective_date)

    // Insert contract policies logically
```

```
    For each contract_policy in contract_policies
        InsertIntoContractPolicies(contract_id, policy_id)

    // Print success message
    Print("Database populated successfully!")
}
```

5.7 XML Schema

The following figures mention ER diagram and use case diagram respectively:

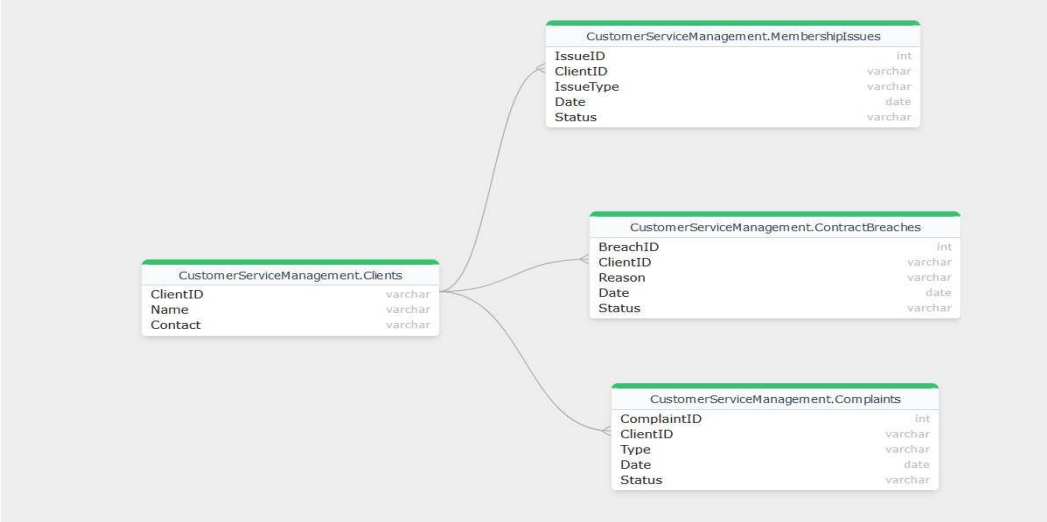

Figure 11: ER diagram of XML database

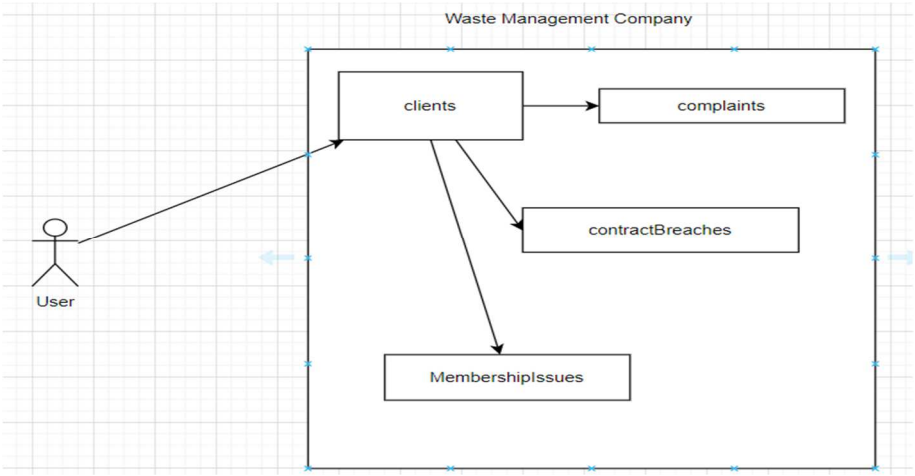

Figure 12: Use Case Diagram

5.7.1 Problem Domain Addressed

Now, the primary challenge lies in integrating data from several data sources from multiples platforms, among them implementation of SQL databases, NoSQL databases (such as Apache Cassandra), and XML databases. Each data storage represents a distinct data structure and technology. In the subsequent vision of the project, it is ment the conflict by making information from all sources accessibly in an identical mode despite many trivial subtleties. The key fact is that access and queries across all data sources will be unified. Finally, the challenge facing the project is to provide a data integration layer.

Another item of consideration, which proves difficult in its own ways, is semantic heterogeneity, wherein the various data sources may differently implement the same notion of a schema. As an example, while on the SQL side one might see a concept like a customer represented with such fields as "*Name*," "*Address*," and "*Contact*," the same concept could be displayed by such tags as <**ClientID**>, <**Address**>, and <**PhoneNumber**> on the XML side. Appropriate schema mappings and schema merging techniques must be implemented to enable the translation of data across source-specific schema into a consistent, global representation guaranteeing that whatever queries are issued across multiple sources, shall still translate well into data processing.

Scalability is another important function. The solution should be scalable in the context of future data sources and query needs. When additional data sources are added, the integration layer must be extensible to support new formats or platforms. This is necessary because the solution is meant to grow without many fundamental structural changes being implemented.

Key Components of this Database Solution

Glossary of Domain Concepts

The domain concept map glossary includes terms like Clients, Complaints, Membership Issues, Contract Breach, and Policies, each term having a certain meaning from the various data sources. Within the Waste Management Company, Clients might be associated with the attributes **ClientID**, **Address**, and **AreaType**, while the Complaints might fall under categories of Billing, Contract Breaches, and Renewal issues. Thus, the understanding of these terms and how they relate to the various data sources is critical in the development of a consistent approach to integrate the data.

System Requirements & Use Cases

System requirements would mean that three different types of data sources—SQL, NoSQL, and XML—can provide data integrability and query performance. Use cases may include querying the Complaints for a specific client or getting some Contract Breach data related to certain Membership Issues. On execution of these queries, the system should also return findings from the data sources uniformly. Say a user makes a query-for all the unresolved complaints concerning Billing of his clients in the Industrial Area-and this query will query all the data sources to integrate the results.

Constraints

To maintain the query capability and data integrity, the system should be well-founded on prerequisite restrictions. The previous problem of semantic heterogeneity must henceforth be faced. The system must follow data integrity rules, like you see that foreign key relationships between various data sources (*clients*, *complaints*, *contracts*) are maintained. Finally, the system must be able to accommodate scalability of any new data sources without breaking the integration logic or requiring a complete redesign.

5.7.2 Data Population

Here is the **pseudo-Code** for XML database creation and for population:

```
PopulateDatabase()
{
   // Connect to Database
   ConnectToDatabase()
```

```
    // Clear existing data
    Truncate(area_types)
    Truncate(classifications)
    Truncate(client_statuses)
    Truncate(complaints)
    Truncate(membership_issues)
    Truncate(contract_breaches)

    // Insert Area Types logically
    For each area_type in area_types
        InsertIntoAreaTypes(area_type_id, area_type, description)

    // Insert Classifications logically
    For each classification in classifications
        InsertIntoClassifications(classification_id, classification, description)

    // Insert Clients logically with valid relationships
    For each client in clients
        AssignClientAttributes(client)
        InsertIntoClients(client_id, name, contact, address, area_type_id, classification_id)

    // Insert Complaints logically
    For each complaint in complaints
        AssignComplaintAttributes(complaint)
        InsertIntoComplaints(complaint_id, client_id, complaint_type, complaint_date,
complaint_status)

    // Insert Membership Issues logically
    For each membership_issue in membership_issues
        AssignIssueAttributes(membership_issue)
        InsertIntoMembershipIssues(issue_id, client_id, issue_type, issue_date, issue_status)

    // Insert Contract Breaches logically
    For each contract_breach in contract_breaches
        AssignBreachAttributes(contract_breach)
        InsertIntoContractBreaches(breach_id, client_id, breach_reason, breach_date, breach_status)

    // Print success message
    Print("Database populated successfully!")
}
```

6. Integration Process:

For the Group Project, we are supposed to extract and analyze data from all the databases on waste and water management systems effectively. Targeted with sourcing, merging, and integrating data from multiple sources and afterward analyzing the same, this report varies in the way the various cases about sourcing the data, merging and the logic governing certain decisions like row limitation, treatment of large data sets, and saving of data in CSV, are explained. Further, with the programs LAV (Local as

View), GLAV (Global as View) and GAV (Global As View), these strategies for the integration are described along with the actual steps taken as yet not even mentioned. Thereafter, as the project progresses, our report will come out in detail in this prospect regarding these integration models.

6.1 Data Extraction Algorithms and Wrappers

The actual process for data extraction is done using a few Python scripts, where each one has custom-built or existing wrappers that interact between the databases or APIs.:

1. **PostgreSQL:** *psycopg2* works as the wrapper that communicates with PostgreSQL databases. Connect to the database and execute SQL queries to collect results and save them as CSV using the CSV module.
2. **MySQL:** *mysql.connector* was used to communicate with the MySQL database. The library executes your queries, fetches the data, and saves that into your disk in CSV format.
3. **MongoDB:** *pymongo* communicates with MongoDB, retrieves collections, transforms them into *DataFrames* directly, and saves them as CVS Format.
4. **Cassandra:** *cassandra.cluster.Cluster* connects with Apache Cassandra. The queries run on Cassandra are to pull data out from several tables, while data saved as CSV files.
5. **XML Data:** The XML data is fetched using the requests library with the API of BaseX. This is parsed using the *xml.etree.ElementTree*, and the CSVs are written extracting the correct information wherever necessary.
6. **Web API**: The POST request module is used to make a RESTful API call. It engages with another API where sensor data is provided to compute the water quality index, for which the results are stored in a CSV format using the *csv.DictWriter* class.

Why CSV Format and Limiting Rows

After huge volumes of data were pulled from so many sources, the brainchild was to store data in formats like CSV and specify the number of 'row counts' to handle with the following reasons behind it:

1. **Memory Constraints**: Too much memory is used when processing huge datasets and the system risks crashing when resources are low with the CPU and it also does not have access to enough space to operate on; hence, saving data as CSV discharges the memory load while extracting the data. In due course, such bulky datasets can just be read into memory in bits and pieces whenever they are required.

2. **Ease of Use**: CSV files exist in human-readable format which further binds them well with other utilities of Python, like Pandas. CSVs play a good role in other areas of data handling or analysis.

3. **Memory Errors**: It helps keep memory errors at bay if any rows are to be chosen-for instance, one may easily cap the rows to keep the memory capacity in check, used in laying out a helpline to explain to the system that a lot of data is beyond the memory available. This warranted stability in terms of the system.

4. **Laptop Limitations**: Describing further, secondary devices like laptops are hindered chiefly due to their CPU and RAM capabilities; if they crash or become slow, they are unable to process very huge data sets. Substantially our recommendation would be to save these datasets into CSV and, in fact, cap the number of rows to be processed. Then proceed to integrate them without that risk to enable smooth and efficient data integration. If such datasets cannot be

loaded into memory entirely, there might be a need to separate the actual processing workload into several mini-processes which can be carried out on a per-call basis.

Data Merging Process

After extraction and storage of data from different sources into CSV files, diverse merging operations were carried out with the application of different algorithms to construct datasets. The merging of water management datasets from MySQL and MongoDB was performed in the following steps:

1. **Data Cleaning**: Column names were cleaned on each occasion (removed any additional spaces and converted to lowercase) to promote settling schema formatting across all datasets. This is an essential step during pre-processing to wipe-off potential attacks from column name discrepancies.
2. **Limiting Rows**: For example, limited queries were used in MySQL and MongoDB based on the number of records in each dataset until we could fetch data and load them into the framework. If the dataset would contain more than 100 rows, fetching will be done for the first 50 rows, but when less than 50 and greater than 20 rows, fetching will be done for the first 20 rows. This step helps in managing the memory usage effectively while making certain that the merging activities do not crash the system.
3. **Merging *DataFrames*:** The merging sequence of *DataFrames* was executed using Pandas. Data from various sources were fitted using *pd.merge*() on common columns. The restored operation was an "inner" join type, such that it ensures that only those records with identical values in both datasets survive in the end resulting merge output: thus, the possibility of any un-matching records within the datasets forming between two data sources unloaded out from such merged data.

Example:

- Merging water conservation programs (**water_conservation_programs_df**) with program types (**program_types_df**) based on the **programtypeid**.
- Merging with other datasets like **water_usage_records_df, departments_df**, etc., to combine relevant information.

6.2 Heterogeneity:

The concept of heterogeneity rears its head when dealing with multiple data sources, each having their structures, formats, and schemas—consolidating databases relating to Web-based water management data, Cassandra-based waste management information, billing data from PostgreSQL, and XML information from customer service presents the challenge of integrating heterogeneous data.

- **Water management (Web)**: The data consists of environmental monitoring parameters like pH, turbidity, and temperature collected from an API that stores those values in JSON format. This data mainly involves sensor and report tables connected by common identifying attributes like **ReservoirID**.
- **Water management (MongoDB)**: Data lying inside different collections in MongoDB (**CustomerDetails**, **WaterUsageOptimization**, **PoliciesDetails**) are merged together by matching their documents' common attributes. The merged data is stored as merged_*mongo.csv*.

- **Water management (MySQL)**: In merging, **WaterConservationPrograms**, **ProgramTypes**, **Departments**, **WaterSource**, and others including related tables will be considered appropriately. The dataset with inner joints performed on the common columns provides *mysql_merged.csv* at the end.
- **Waste management (Cassandra)**: This is NoSQL database (Cassandra) data consisting of data regarding clients, contracts, policies, and waste categories. Cassandra requires structured data to be populated for denormalization.
- **Billing (PostgreSQL)**: Structured tables pertaining to clients, collection histories, and billing are stored in PostgreSQL. Fetching the data using SQL schemas returned in form of plain table with aligned column names from different sources imply that the merged data will receive similar alignment.
- **Customer service (XML)**: XML data involves clients, complaints, membership issues, and contract breaches. The challenge here is to convert the XML data into a structured format that can be related to other tables.

Heterogeneity between these data sources is solved by determining common attributes, for example, **ClientID**, **ReservoirID**, and **ContractID**. These attributes are also linked identification markers which ensure proper merging of data at the right record level despite the differences in data storage formats for different systems.

6.3 Schema Mappings/Manipulation:

The next critical steps in the integration are mapping and data manipulation, especially in mapping relational/non-relational databases. Here, fields across various sources get aligned to enable the data to be merged effectively.

- **Water management schema**: Each time-series dataset for environmental parameter (from *sensor*, *report*, *pH*, *turbidity*, *temperature*) needs to be aligned on **ReservoirID** for reconciliation into a single dataset. Mapping ensures that each report contains environmental conditions sourced from the relevant sensor operations.
    - Example: The water report data might have columns such as **ReportID**, **ReservoirID**, and Date, and this can be extended by adding columns for sensor readings (**SensorID**, **Temperature**, etc.). In the below code we can see in the mappings for Web dataset that certain environmental parameters like Temperature and Turbidity are linked with sensor operations:

```
schema_mapping_web = {
    'ReportID': 'ClientID',
    'Name': 'SensorType',  # Sensor operation
    'Address': 'Location',
    'ContactInfo': 'ReportURL',
    'ProgramName': 'Temperature',  # Environmental parameter (temperature)
    'StartDate': 'Time',
    'EndDate': 'Time',
    'Status': 'Validation',
    'PolicyName': 'pH',  # Another environmental parameter (pH)
    'BillingAmount': 'Turbidity',  # Environmental parameter (turbidity)
```

```
  'LastInspectionDate': 'Time'
}
```

    - In this case, the **sensor data** such as temperature, turbidity, and pH from the Web dataset are mapped to the global schema.
    - These mappings can be extended as needed, and you might need to adjust or add ReservoirID or other related identifiers depending on the data structure.

- **Waste management schema (Cassandra)**: Cassandra's data is more flexible and requires creating lookup dictionaries for each data table (e.g., clients, contracts, area_types). During merging, we associate related records based on keys like client_id, area_type_id, and contract_id. Data from these tables is brought together using these key relationships.

    - Example: For each client_id, the associated contract_id is retrieved, and additional information like contract_status and policy_name are attached. in the case of Cassandra, the schema mapping aligns specific columns from the local dataset (merged_cassandra) to the global schema (lav_cassandra):

```
schema_mapping_cassandra = {
  'ClientID': 'ClientID',
  'Name': 'ClientName',
  'Address': 'Address',
  'ContactInfo': 'ContactNumber',  # Linking contact info
  'ProgramName': 'PolicyName',
  'StartDate': 'StartDate',
  'EndDate': 'EndDate',
  'Status': 'ContractStatus',  # Contract status
  'PolicyName': 'PolicyName',
  'BillingAmount': 'BaseFee',
  'LastInspectionDate': 'LastUpdated'
}
```

    - In this case, **client records** in Cassandra are linked to their contract_id, and **contract-related information** (such as contract_status, policy_name) is brought over.
    - If needed, you can extend this mapping by adding new fields from other Cassandra tables and ensuring each record is correctly associated by client_id or contract_id.

- **PostgreSQL schema**: Tables in PostgreSQL are structured in a relational model with clear foreign key relationships. The data fetched from Clients_list, AreaTypes, CollectionSchedule, and other tables is merged by their foreign keys (AreaTypeID, ClientID, etc.) to create a unified record for each client.

    - Example: For each client, attributes like AreaType, CollectionFrequency, and WasteType are added to generate a comprehensive profile.

```
schema_mapping_postgres = {
  'ClientID': 'ClientID',
  'Name': 'Name',
  'Address': 'FullAddress',
```

```
    'ContactInfo': 'MobileInfo',  # Contact info
    'ProgramName': 'WasteType',  # Waste type for each client
    'StartDate': 'LastCollectionDate',  # Mapping dates
    'EndDate': 'NextCollectionDate',
    'Status': 'Status',
    'PolicyName': 'WasteType',
    'BillingAmount': 'TotalAmount',  # Total amount for the client
    'LastInspectionDate': 'BillingDate'
}
```

- In this example, **PostgreSQL tables** (e.g., Clients_list, AreaTypes, CollectionSchedule) are merged by foreign keys like ClientID, AreaTypeID, and so on.
- We can see that information like **waste type**, **collection schedule**, and **billing amount** are added to the client profile, creating a unified record for each client.

- **MySQL Schema:** The MySQL data consists of various tables such as ProgramTypes, WaterConservationPrograms, WaterSource, etc. The key step here is to clean the data by standardizing column names (converting them to lowercase and stripping whitespaces). For example, after loading data into DataFrames, the ProgramTypes and WaterConservationPrograms tables can be merged based on a shared column, programtypeid. Similarly, data from other tables such as WaterUsageRecords, Departments, and TreatmentEquipment is merged on common columns, ensuring each entity is connected.
  - Example:

```
# Merge WaterConservationPrograms with ProgramTypes in MySQL
merged_df = pd.merge(water_conservation_programs_df, program_types_df, on='programtypeid', how='inner')
```

- **MongoDB Schema:** MongoDB requires a slightly different approach due to its document-based nature. The data is fetched from collections like CustomerDetails, PoliciesDetails, and WaterUsageOptimization. As MongoDB doesn't use relational keys, the merging process typically involves joining collections based on common attributes such as customerid or programid. The merging of MongoDB data is done by merging documents on index-based keys, while ensuring that column names are aligned properly.
  - Example:

```
# Fetch data from MongoDB collections and merge them based on row limits
merged_data = merge_collections(row_limit)
```

The key challenge in schema mapping here is aligning the varying structures (relational tables, JSON objects, XML elements) into a unified format. This is achieved through careful manipulation of each dataset, ensuring that data fields across sources are mapped to a consistent schema in the merged output.

Schema mappings serve as the foundation for data conversion between local source schemas and a global schema, enabling data integration across different data sources. In this report, we employ different schema mapping strategies: **LAV (Local-As-View)**, **GLAV (Global-As-View)**, and **GAV**

(Global-As-View) to achieve integration across multiple data sources, including MySQL, MongoDB, PostgreSQL, Cassandra, Web, and XML databases.

LAV (Local-As-View) Transformation:

Purpose: The LAV approach transforms local schema columns to match a global schema. This enables local datasets to be viewed as a projection or simplified version of the global schema, maintaining consistency across heterogeneous sources.

Transformation Logic:

- Each dataset is mapped to a global schema where local columns (from the source data) are mapped to the corresponding global column names.
- For example, in Cassandra, the **ClientName** column is mapped to **ClientID** in the global schema. Similarly, **ContractStatus** in Cassandra is mapped to Status in the global schema.
- **Code Explanation for LAV Transformation:**

```
def lav_transform(df, schema_mapping):
    transformed_data = {}
    for global_col, local_col in schema_mapping.items():
        if local_col in df.columns:
            transformed_data[global_col] = df[local_col]
        else:
            transformed_data[global_col] = None
    return pd.DataFrame([transformed_data])
```

- The function lav_transform takes a local dataframe (df) and a schema mapping (schema_mapping). It returns a dataframe where columns from the local schema are transformed into the global schema.
- A schema mapping for Cassandra might look like this:

```
schema_mapping_cassandra = {
    'ClientID': 'ClientName',
    'ProgramName': 'PolicyName',
    'StartDate': 'StartDate',
    'EndDate': 'EndDate',
    'BillingAmount': 'BaseFee',
    'LastInspectionDate': 'LastUpdated'
}
```

  This mapping ensures that **ClientName** from Cassandra is renamed as ClientID in the global schema.

- **Example Schema Transformation**:
  - **Before LAV:** Local column **ClientName** in Cassandra.
  - **After LAV:** Global column ClientID which now corresponds to **ClientName** from Cassandra.

GLAV (Global-As-View) Transformation:

Purpose: The GLAV approach is the reverse of LAV. It takes a global schema and merges it with the local datasets. This provides a hybrid schema where global data is extended with local data.

Transformation Logic:

- The GAV transformation generates a global schema view by combining data from multiple local sources into a unified view. We concatenate local dataframes into a single, global dataset.
- The GLAV transformation involves merging the global schema data with the local data. The global schema acts as the base schema, and local data is appended to it.
- Code Explanation for GLAV Transformation:

```
def glav_transform(global_df, local_dfs):
    merged_data = pd.concat([global_df] + local_dfs, ignore_index=True, sort=False)
    return merged_data
```

- This function takes the global *dataframe* (*global*_df) and a list of local *dataframes* (*local_dfs*), merging them together to create a global view.
- GAV & GLAV Combined Workflow:
    - **GAV:** Combines the data from all sources into a unified global schema.
    - **GLAV:** Merges this global schema with local data for richer context, adding details such as **BillingAmount**, **StartDate**, etc.

GAV (Global-As-View) Transformation:

Purpose: The GAV transformation creates a comprehensive view of the data by aligning all local datasets according to the global schema. This is critical for cross-platform integration.

- Code Explanation for GAV Transformation:

```
def gav_transform(local_dfs):
    local_dfs_reset = [reset_index_with_check(df) for df in local_dfs]
    combined_data = pd.concat(local_dfs_reset, ignore_index=True, sort=False)
    return combined_data.fillna(np.nan)
```

- This function ensures that local data sources are integrated into a global schema by resetting indices and concatenating them into a unified *dataframe*.

6.4 Merged Data Across All Databases:

Once heterogeneity is managed and schema mappings are defined, we merge data from the different sources. Here's how the process looks for each dataset:

- **Web-based water management**: After limiting the number of rows to avoid memory overload, the relevant environmental data (e.g., *ph, temperature, turbidity*) is merged based on the common **ReservoirID**. The merged data is then written to a CSV file with a structured header and data rows.
- **Cassandra-based waste management**: Data is fetched from multiple tables (e.g., **clients**, **contracts**, **contract**_policies) and merged based on the client_id and contract_id. The merged rows are then written to a CSV file, ensuring that each client's information is presented with relevant contract and policy details.
- **PostgreSQL-based billing**: Data is pulled from multiple billing-related tables using a SQL query with appropriate joins. The fetched data is then merged by ensuring that all related information (client, area type, collection frequency, etc.) is combined into a single record. The result is written to a CSV file.
- **XML-based customer service management**: Data from Clients, Complaints, **MembershipIssues**, and **ContractBreaches** is merged by matching the **ClientID**. After merging, the data is saved into a CSV, ensuring all necessary fields from all tables are included.
- **MySQL Merged Data:** The merging process includes combining **WaterConservationPrograms**, **ProgramTypes**, **Departments**, **WaterSource**, and other related tables. After performing inner joins on the shared columns, the dataset is saved as mysql_merged.csv.
- **MongoDB Merged Data:** Data from various MongoDB collections (e.g., **CustomerDetails**, **WaterUsageOptimization**, **PoliciesDetails**) is merged by matching documents on common attributes. The merged result is saved as *merged_mongo.csv*.
    - Example of merging logic:

```
# Merge WaterUsageOptimization with CustomerDetails
merged_data = pd.merge(merged_data,
customer_details_df, left_on='customerid',
right_on='customerid', how='left')
```

6.5 Query Processing in Data Integration:

Query processing in data integration involves two key techniques: query containment and queries using views. These helps ensure data retrieval is efficient across integrated schemas and sources.

Query Containment:

- Query containment ensures that queries on integrated data can be executed efficiently across various data sources without redundancy. When querying integrated datasets, we ensure that each query is covered by the available views and transformations, minimizing data access errors.

Queries Using Views:

- **LAV View Queries**: In the LAV approach, queries are based on transformed local views. For example:

```
lav_cassandra[['ClientID', 'Name', 'Address', 'ContactInfo']].head()
```

- This query retrieves information about clients, based on the transformed Cassandra dataset into the global schema.
- **GAV View Queries:** Queries using the GAV view involve retrieving information from the merged global schema view, which includes data from all sources.

```
gav_view[['ClientID', 'Name', 'Address', 'ContactInfo']].head()
```

  - This query retrieves client information from the integrated global dataset.

- **GLAV View Queries:** GLAV queries involve using the final global view that combines both global and local schemas.

```
glav_view[['ClientID', 'BillingAmount', 'ProgramName']].head()
```

- This query retrieves client information along with billing and program details, integrating both local and global perspectives.

6.6 Data Normalization:

Before performing schema mappings and transformations, we normalize the data to ensure consistency across datasets:

- **Dates Normalization:** Ensures all date columns across datasets are converted to a standard format (datetime).
  - Example: The dates in Cassandra and MongoDB datasets are normalized as shown below:

```
def normalize_dates(df, date_columns):
    for col in date_columns:
        if col in df.columns:
            df[col] = pd.to_datetime(df[col], errors='coerce', dayfirst=True)
    return df
```

- **Contacts Normalization:** Standardizes contact details across datasets to treat them as string data types.
  - Example: Contact columns like **ContactNumber** in Cassandra and **MobileInfo** in PostgreSQL are normalized.
- **Numerical Data Normalization:** Converts numerical columns to a uniform data type to enable accurate computations.

```
def normalize_numerical(df, numerical_columns):
    for col in numerical_columns:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors='coerce')
    return df
```

6.7 Similarity Measures:

The Jaccard and Levenshtein similarity algorithms are often employed to evaluate the similarity or distance between two sets or strings. Both algorithms have applications in various fields such as data mining, natural language processing, and record linkage. In this case, you are applying both algorithms to evaluate the similarity of column names between different datasets.

6.7.1 Jaccard Similarity

The Jaccard Similarity measures the similarity between two sets by comparing the size of their intersection to the size of their union. The formula is:

$$Jaccard\ Similiarity = \frac{|A \cap B|}{|A \cap B|}$$

Where AAA and BBB are two sets.

- **Result Interpretation**: The higher the Jaccard score, the more similar the two datasets are in terms of column names. For example:
    - **Cassandra vs Mongo**: The Jaccard similarity is 0.1224, which means that approximately 12% of the column names in Cassandra and Mongo are common.
    - **Cassandra vs Postgres**: The similarity score is even lower at 0.0571, suggesting minimal overlap in column names.
    - **Cassandra vs Web** and **Cassandra vs MySQL** both show scores of 0.0, indicating that there are no common column names between Cassandra and these datasets.

In general, a score of 0 means no shared elements, and a score of 1 means the two sets are identical. These scores provide a sense of the overlap of column names between the different datasets, with the higher the score, the more closely the datasets resemble each other.

6.7.2 Levenshtein Similarity

The Levenshtein Similarity measures the similarity between two strings by calculating the minimum number of single-character edits (insertions, deletions, substitutions) required to convert one string into the other. The formula for similarity is:

$$Lavenshtein\ Similiarity = 1 - \frac{lavenshtein\ Distance}{Max\ Length\ of\ Strings}$$

This method is useful when the strings involved are similar but not identical, especially when small variations exist in the column names.

- **Result Interpretation**: The output provides a list of similarity scores between each column name in Cassandra and Mongo, for example:
    - The Levenshtein similarity scores for some column pairs range from 0.0 (completely different) to 1.0 (identical). For instance, columns that match closely will have a higher score like 0.5, while completely different names will have a score near 0.
    - **Cassandra vs Mongo**: You see several values indicating varying degrees of similarity. For example, the score between some columns is 0.125, meaning there is minimal similarity between those columns, while some scores reach 0.4, indicating a stronger similarity.

6.7.3 Combining Results

When combining the results of the Jaccard and Levenshtein algorithms, you can create a more comprehensive view of how the datasets relate in terms of their column names:

- For **Cassandra vs Mongo**, the Jaccard score is low (0.122), while the Levenshtein scores show a variety of values indicating some degree of similarity for certain columns.
- For **Cassandra vs Postgres**, the Jaccard score is even lower (0.0571), and the Levenshtein scores also reflect some moderate differences.

This combination of Jaccard and Levenshtein scores helps identify the best matches and areas where column names are similar or distinct, and it provides insight into how datasets may be integrated or aligned based on their schema.

7. Conclusion

This project acts as a rough guide towards the way water and waste data can be integrated by showcasing some practical steps to extract, merge, and analyze data from various sources. Python libraries including *psycopg2*, *mysql.connector*, and *pymongo* helped with easy extraction from PostgreSQL, MySQL, MongoDB, and other databases. Data was stored in CSV format to work with huge datasets, with row limits imposed on rows to prevent memory errors. Row capping for specific thresholds such as that with 50 rows or 100 rows was considered to prevent hogging the memory, where the memory problems would abate in systems utilizing limited RAM. Such an approach may pose impediments to the maintenance of exhibitable data for particularly elaborate analyses on the much more elaborate dataset sizes. In the same vein, these methodologies, stress stability, may at times lead to information loss if handled with the wrong level of sampling.

It was expected that the data integration would be aimed at confronting the heterogeneous data by applying schema mapping techniques ingrained in the LAV, GLAV, and GAV undertakings. The presence of a great variety of data structures such as relational tables, NoSQL documents, and XML elements posed great threats to the uniformity of bringing the data together, but the schema mappings addressed this problem. These schema mappings permitted a high degree of consistency in data acquisition, be it from algorithms like Jaccard and Levenshtein similarity, providing the actual external insights into the overlap of the data while aligning the column names across sources.

These row limits could help contain the problems generated by memory issues while also raising a slight question as to whether the restitution of the data if a representative view did not make it to the capped rows of data-this was the case when selecting a specific sample of the data to fill the gap of a few capped rows. On the contrary, by working with smaller chunks, the integration process became more efficient and manageable. It follows that working experience of such methodologies will see these guide the integration process further, securing a broader and intelligible platform for discussions. Such integration of distinct data sets addresses technical challenges and can also provide a more holistic approach acting upon which the systems are viewed for efficient managing and analysis.