Sarajevo School of Science and Technology

Database Systems Project Report: Air Quality Monitoring

Students:
Din Bečirbašić
Darin Anić
Džani Eterle

Professors:
Amer Hadžikadić
Bakir Husović

Table of Contents

Introduction

This comprehensive report details the development lifecycle of a database system designed to manage air quality measurements from various sources. The database not only accommodates the storage and management of these measurements but also stores information about the users inputting and viewing this data. This functionality is distributed across multiple entities, leading to a well-structured, efficient database.

Project Scope and Functionality

The primary aim of the database is to store and manage air quality data. This includes measurements of different air quality parameters like Sulphur dioxide (SO2), Particulate Matter (PM10), and Ozone (O3) among others. These measurements are taken from different sources in various locations and are inputted by administrator users.

Administrators have the capacity to manage users and their privileges, while all users can view measurements. The data can be analyzed for identifying air quality trends, and to understand which areas have the best and worst air quality, as well as the sources of different pollutants.

Database Development Lifecycle Steps

Step 1: Database Initial Study (Initial Planning and Feasibility Analysis)

In the case of our project, the requirement was for an Air Quality Monitoring System. The need was driven by increasing environmental and health concerns around air quality and the desire for a more systematic, data-driven approach to managing and monitoring these aspects. The objective was to create a robust and scalable system capable of capturing a variety of measurements and data points, effectively storing and managing them, and facilitating easy retrieval and analysis.

Once the need and objectives were clearly defined, we proceeded to a feasibility analysis. This included a technical feasibility assessment to ascertain whether current technological resources were sufficient to support the new system.

Identifying the main entities of the database was another crucial aspect of the initial study. In our Air Quality Monitoring System, the essential entities included `AirQualityMeasurement`, `AirQualityParameter`, `AirQualitySource`, `City`, `Location`, and `User`. Each of these entities represented distinct data elements that would be tracked and managed within the system. For instance, `AirQualityMeasurement` captures each specific measurement event, `AirQualityParameter` records the types of parameters being measured, and `User` represents the individuals entering and managing data within the system.

The relationships among these entities were also preliminarily defined in this step. It was important to establish these relationships to ensure data integrity and consistency, and facilitate efficient data retrieval. For example, we determined that an `AirQualityMeasurement` would be associated with a specific `Location`, `AirQualitySource`, `AirQualityParameter`, and `User`.

In conclusion, the initial study phase provided a blueprint for the entire project, laying out the objectives, requirements, key entities, and expected benefits. It ensured the project started on solid footing, and set the stage for the subsequent design phase. Through careful planning and thorough feasibility analysis, we were able to set clear expectations to proceed with the development of the Air Quality Monitoring System.

Step 2: Database Design (Conceptual, Logical, and Physical)

The second phase of the Database Development Life Cycle (DBLC) is the Database Design phase. This stage is arguably the most critical as it translates the insights gathered during the initial study into a blueprint for the future database system. Design is performed in three sub-stages: conceptual, logical, and physical.

Conceptual design forms the high-level description of the database and doesn't concern itself with the physical intricacies of data storage. It involves creating an Entity-Relationship Diagram (ERD) to define the relationships between entities, while keeping in mind the system's requirements. This phase abstracts the database design, ensuring it is understandable even to non-technical stakeholders. For the Air Quality Monitoring System, entities such as `AirQualityMeasurement`, `AirQualityParameter`, `AirQualitySource`, `City`, `Location`, and `User` were identified, and the relationships between them were established using an ERD.

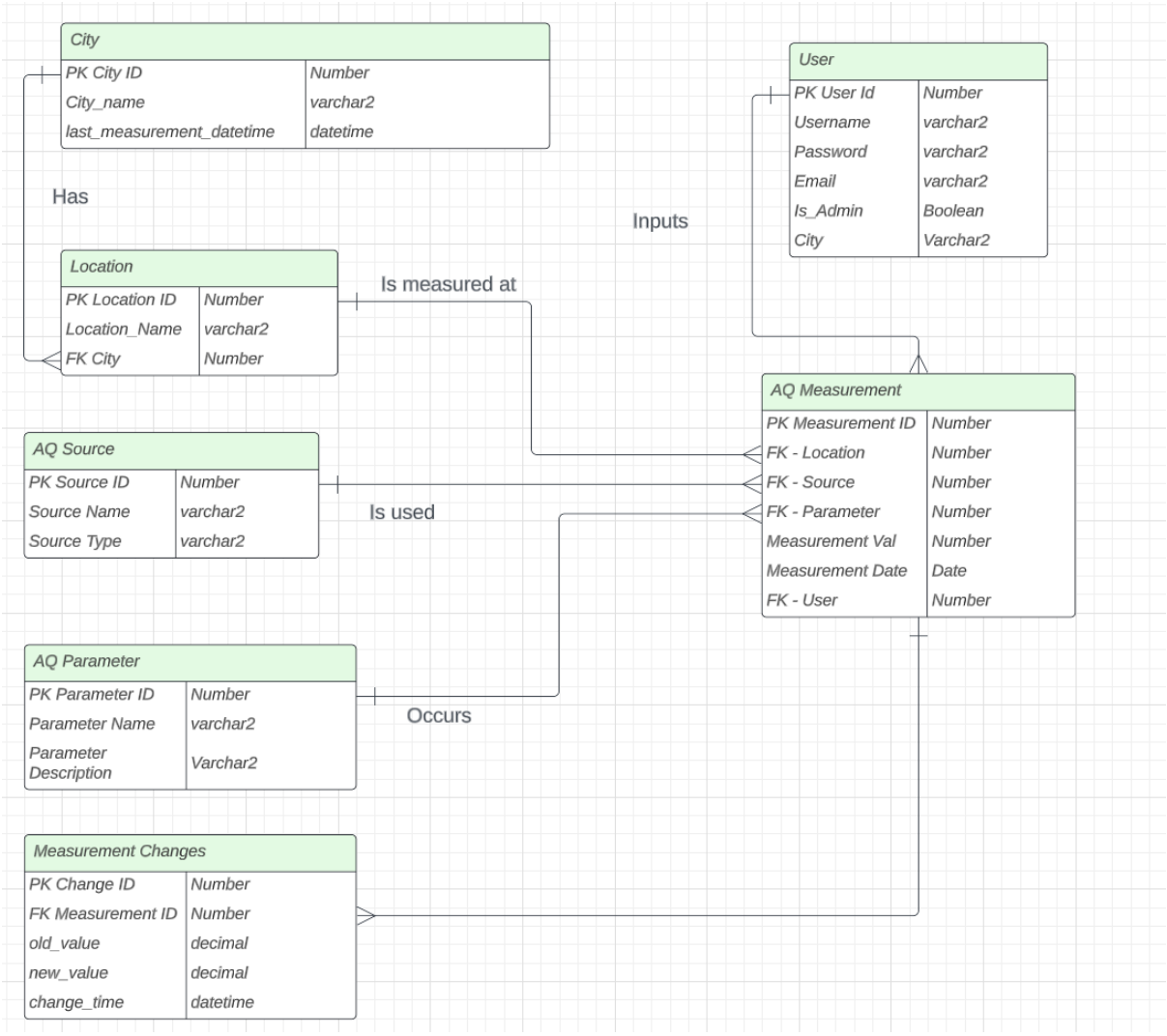

Figure 1 - Entity Relationship Diagram

The logical design phase serves as a bridge between the high-level conceptual design and the detailed physical design. During this phase, the ERD from the conceptual design is translated into a data model. This logical data model outlines the database's structure and details how the entities connect to each other, without any technical specifications. The model includes tables representing each entity, and these tables consist of rows and columns corresponding to records and attributes, respectively. The primary and foreign keys that form the links between the tables are also defined at this stage.

In our Air Quality Monitoring System, tables were defined to match each entity identified during the conceptual design. For example, the `AirQualityMeasurement` table was designed to hold each measurement record, with the primary key `measurement_id`. The `AirQualityParameter` table was created to store different air quality parameters, with `parameter_id` as the primary key. Relationships were established using foreign keys, such as `location_id` in `AirQualityMeasurement`, linking it to the `Location` table.

The physical design phase is where the rubber meets the road. In this stage, the logical data model is translated into physical storage structures, considering factors such as storage space, processing efficiency, and access speed. Decisions are made about physical data storage, index design, and optimizing for performance. For example, for our `AirQualityMeasurement` table, the field `measurement_value` was defined as a decimal type to allow precise air quality readings. Additionally, indexes were created on keys like `location_id`, `source_id`, `parameter_id`, and `user_id` to speed up data retrieval.

The design phase is a significant stage in the DBLC, laying out a plan for the database that both captures the system's requirements and ensures efficient operation. The Air Quality Monitoring System benefited from a thorough design phase, ensuring the database could effectively store and manage the critical data needed for air quality tracking and analysis.

Step 3: Implementation and Loading

Having established a clear database design, the third step in the Database Development Life Cycle (DBLC) is Implementation and Loading. It's a two-pronged phase that involves physically setting up the database based on the physical design and populating it with the initial data set. This process is vital as it turns the theoretical plans and structures into a functional system that users can interact with.

In the implementation stage, the database is set up in a DBMS (Database Management System). The DBMS creates a storage structure based on the specifications given in the database design and ensures that data is appropriately stored and retrieved. During this phase, data types for each attribute are defined, constraints are set up, and relationships between entities are established based on the physical design.

For our Air Quality Monitoring System, the database was implemented in MySQL, a popular relational DBMS. The database tables were defined using SQL `CREATE TABLE` statements with the precise structure and constraints as outlined in the design. For example, the `AirQualityMeasurement` table was created with an `int` data type for the `measurement_id` attribute and a `decimal(10,5)` data type for the `measurement_value`. Moreover, foreign key constraints were set up to ensure the consistency and integrity of data across tables. For instance, the `location_id` in the `AirQualityMeasurement` table was linked to the `Location` table using a foreign key constraint.

In the loading stage, data is inserted into the database tables. This initial dataset could come from multiple sources, such as data files, existing databases, or data created for testing purposes. The loading process must be carefully controlled to ensure data integrity and quality, as incorrect or low-quality data can significantly reduce the utility of the database.

In the case of our Air Quality Monitoring System, the initial data was loaded into the database using SQL `INSERT INTO` statements. Data for each table was inserted according to the table structure and constraints. For example, in the `AirQualityMeasurement` table, values for `measurement_id`, `location_id`, `source_id`, `parameter_id`, `measurement_value`, `measurement_datetime`, and `user_id` were provided. This ensured the correct population of the database while also maintaining the integrity of the data.

Through the process of implementation and loading, the theoretical and logical design of the database takes tangible form. After this step, the Air Quality Monitoring System is no longer just a design on paper; it's a functioning database, capable of

storing, retrieving, and manipulating data as needed. However, the DBLC is not done yet. With a working database in hand, we now need to verify that everything works as intended, setting the stage for the next step in the cycle: Testing.

Step 4: Testing

Testing is the fourth stage in the Database Development Life Cycle (DBLC) and arguably one of the most critical. At this stage, the database and its elements undergo rigorous examinations to ensure its correct functioning, reliability, and performance under various conditions.

For the Air Quality Monitoring System, the testing process would evaluate several critical areas. Firstly, it would ensure that the database has been implemented according to the design specifications. This would include validating the creation of all tables, attributes, and relationships, as specified in the design stage. It would also involve checking that all constraints are enforced correctly.

In our MySQL-based system, we would run queries to fetch the table schema and verify the data types, keys, constraints, and relationships. For instance, we could run a `DESCRIBE AirQualityMeasurement;` command to verify that all the attributes in the `AirQualityMeasurement` table are correctly defined. Similarly, we could examine the enforcement of relationships by inserting data that violates the foreign key constraints and expecting the system to reject such data.

Next, we would test the data's integrity and consistency. The database is expected to maintain its consistency even in situations of concurrent access, system failures, or bugs in the application. This ensures the reliability of the data, which is crucial for an air quality monitoring system where inaccurate or inconsistent data could lead to incorrect analyses and decisions.

Further, we would conduct security testing to verify that the database adequately protects sensitive data and prevents unauthorized access or manipulation. In our MySQL database, we would evaluate the security protocols in place, such as user authentication and authorization. In addition to this, features were implemented to test data integrity if malicious or ignorant actors accessed administration privileges, this made it impossible for users to delete data that had constraints implemented.

Lastly, the system would be tested for compatibility with the interfaces it needs to interact with, such as the applications or systems that will use the database.

Each of these tests could be performed manually or using automated testing tools. They would identify any bugs, inefficiencies, or vulnerabilities in the system, allowing these issues to be corrected before the system goes live.

Once the database has passed all tests, it can be deemed ready for use. However, the work doesn't stop there. The next step in the DBLC is Operation, which includes ongoing maintenance and regular evaluation of the system to ensure it continues to meet the needs of its users.

Step 5: Operation

Once the database system has been tested thoroughly and deemed ready, it moves to the operation phase. This is the longest stage in the Database Development Life Cycle (DBLC) and involves running the database to support day-to-day activities.

The first part of the operation phase is the actual implementation of the database in a live environment. The database would then start serving the application or system it was built to support. For instance, it would begin to accept and store air quality measurements from the various sources outlined in the `AirQualitySource` table.

During the operation phase, the performance of the database system is closely monitored. The focus is on ensuring that the system performs optimally and continues to meet user requirements. Performance tuning, such as optimizing queries and indices, can be done during this phase to ensure the database runs smoothly. Given the potential volume of data, our air quality monitoring database would need ongoing performance tuning to manage the load effectively.

Another critical aspect of this phase is the security and integrity of the database. The MySQL database for our Air Quality Monitoring System would store critical environmental data, so it would be vital to ensure only authorized personnel can access and manipulate this data.

Ultimately, the operation phase is about maintaining the health and efficiency of the database system while serving the needs of its users effectively. Once the system is operational and functioning as expected, we move on to the final phase of the DBLC, which is the Maintenance phase. This phase aims to keep the system up-to-date and adapt to changing user needs or technological advancements.

Step 6: Maintenance & Evolution (Growth, New Requirements, and Database Redesign)

The maintenance phase involves the ongoing upkeep and optimization of the database. Given the critical nature of air quality data, it's important that data queries return timely and accurate results. Regularly checking the database's performance and making the necessary adjustments can help ensure it remains up to the task.

Given that the air quality monitoring system holds potentially sensitive and valuable information, it’s important to ensure the data is regularly backed up and that there are procedures in place for data recovery in case of a system failure or other unforeseen issues. In our SQL dump, we've made sure to include both the structure of the database (via the CREATE TABLE statements) and the current data (via the INSERT statements) for a comprehensive backup.

While our current database design may be sufficient for the present data volume, it may need to evolve to handle larger amounts of data in the future. This might involve partitioning large tables, adding more indexes, or even changing the storage engine.

Advanced Queries and Trigger Implementation

To further enhance the interactivity of the database, we've implemented advanced SQL queries and triggers:

1. Average Measurement Value by Parameter: Calculates the average value of all measurements for each air quality parameter.

2. Measurements for a Specific User: Retrieves all the measurements recorded for a specific user's city. This queries the logged in user's information and checks the measurements to correlate the logged in user's city to information from that city. It then dynamically queries the measurements to list a few measurements on the main page.

3. Trigger for High Measurement Values: A trigger has been implemented to send a notification when a new air quality measurement exceeding a certain threshold is added.

4. Trigger for Logging Measurement Changes: A trigger has been implemented to save information about any changes made to previously existing values in the air quality measurements, ensuring that historical accuracy can be maintained.

5. Trigger for Saving Last Measurement Time: A trigger has been implemented to update a city table attribute with the date of the latest measurement, ensuring administrators can verify the relevance of the measurements.

6. Trigger for Validation of Measurement Values: A trigger has been implemented to automatically verify whether newly input measurements are within normal ranges, if an impossible value is detected the database would issue an error.

7. View for Better Admin Database interaction: A view has been implemented to simplify the output of information from the measurements table. This allows administrators to access location specific measurements, removing the need for a city to be linked with the location thus ensuring better readability of information.

8. Procedure for Getting User City: A procedure has been implemented to allow administrators to easily run a function that would take the user id and find the city in which they are registered.

Conclusion and Possible Improvements

The successful implementation of the database development lifecycle resulted in an efficient and reliable air quality monitoring database system. This database offers an effective solution for storing, managing, and analyzing air quality data. It allows for the easy manipulation and retrieval of data, thereby serving its intended purpose effectively. Advanced SQL queries have further enriched data analysis, and the trigger implementation ensures timely notifications for potential air quality hazards.

Looking forward, there are several areas of potential improvement and expansion that could further augment the system's efficiency and user-friendliness. One such improvement would be the introduction of automatic data capture through APIs, eliminating the need for manual data input and providing real-time air quality updates. Implementing advanced charting and graphing functions would provide users with visually compelling and comprehensible data representations, enhancing data analysis and understanding. In terms of geographical interactivity, integrating advanced map interactions would enable users to navigate and explore air quality data more intuitively and contextually. We could also expand the scope of the platform by integrating advanced weather radar readings, thereby providing users with comprehensive environmental data that could enhance their understanding of the correlation between weather patterns and air quality. These proposed enhancements aim to make the platform more versatile, powerful, and instrumental in monitoring and analyzing air quality.

Appendix

Database: `Projekat`

```
--
-- Table structure for table `AirQualityMeasurement`
--

CREATE TABLE `AirQualityMeasurement` (
  `measurement_id` int(11) NOT NULL,
  `location_id` int(11) DEFAULT NULL,
  `source_id` int(11) DEFAULT NULL,
  `parameter_id` int(11) DEFAULT NULL,
  `measurement_value` decimal(10,5) DEFAULT NULL,
  `measurement_datetime` datetime DEFAULT NULL,
  `user_id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;



--
-- Triggers `AirQualityMeasurement`
--

DELIMITER $$
CREATE TRIGGER `log_measurement_changes` AFTER UPDATE ON
`AirQualityMeasurement` FOR EACH ROW BEGIN
    INSERT INTO MeasurementChanges (measurement_id, old_value, new_value,
change_time)
    VALUES (NEW.measurement_id, OLD.measurement_value,
NEW.measurement_value, NOW());
END
$$
DELIMITER ;
DELIMITER $$
CREATE TRIGGER `update_last_measurement_time` AFTER INSERT ON
`AirQualityMeasurement` FOR EACH ROW BEGIN
    UPDATE City
    SET last_measurement_time = NEW.measurement_datetime
    WHERE city_id = (SELECT location_id FROM Location WHERE location_id =
NEW.location_id);
END
```

```
$$
DELIMITER ;
DELIMITER $$
CREATE TRIGGER `validate_measurement_value` BEFORE INSERT ON
`AirQualityMeasurement` FOR EACH ROW BEGIN
    IF NEW.measurement_value < 0 OR NEW.measurement_value > 100000 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Invalid measurement value.
The value must be between 0 and 100000.';
    END IF;
END
$$
DELIMITER ;



--
-- Table structure for table `AirQualityParameter`
--

CREATE TABLE `AirQualityParameter` (
  `parameter_id` int(11) NOT NULL,
  `parameter_name` varchar(255) DEFAULT NULL,
  `parameter_description` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;



--
-- Table structure for table `AirQualitySource`
--

CREATE TABLE `AirQualitySource` (
  `source_id` int(11) NOT NULL,
  `source_name` varchar(255) DEFAULT NULL,
  `source_type` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;



--
-- Table structure for table `City`
--

CREATE TABLE `City` (
  `city_id` int(11) NOT NULL,
```

```
  `city_name` varchar(255) DEFAULT NULL,
  `last_measurement_time` datetime DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--


--
-- Table structure for table `Location`
--

CREATE TABLE `Location` (
  `location_id` int(11) NOT NULL,
  `location_name` varchar(255) DEFAULT NULL,
  `city_id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;



--
-- Table structure for table `MeasurementChanges`
--

CREATE TABLE `MeasurementChanges` (
  `change_id` int(11) NOT NULL,
  `measurement_id` int(11) DEFAULT NULL,
  `old_value` decimal(10,2) DEFAULT NULL,
  `new_value` decimal(10,2) DEFAULT NULL,
  `change_time` datetime DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;



--
-- Stand-in structure for view `measurementdetails`
--
CREATE TABLE `measurementdetails` (
`measurement_id` int(11)
,`measurement_value` decimal(10,5)
,`parameter_name` varchar(255)
,`location_name` varchar(255)
);



--
```

```
-- Table structure for table `User`
--

CREATE TABLE `User` (
  `user_id` int(11) NOT NULL,
  `username` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `is_admin` tinyint(1) DEFAULT NULL,
  `city` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;




--
-- Structure for view `measurementdetails`
--
DROP TABLE IF EXISTS `measurementdetails`;

CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY
DEFINER VIEW `projekat`.`measurementdetails`  AS SELECT `m`.`measurement_id`
AS `measurement_id`, `m`.`measurement_value` AS `measurement_value`,
`p`.`parameter_name` AS `parameter_name`, `l`.`location_name` AS `location_name`
FROM ((`projekat`.`airqualitymeasurement` `m` join `projekat`.`airqualityparameter` `p`
on(`m`.`parameter_id` = `p`.`parameter_id`)) join `projekat`.`location` `l`
on(`m`.`location_id` = `l`.`location_id`))  ;



--
-- Indexes for dumped tables
--


--
-- Indexes for table `AirQualityMeasurement`
--
ALTER TABLE `AirQualityMeasurement`
  ADD PRIMARY KEY (`measurement_id`),
  ADD KEY `location_id` (`location_id`),
  ADD KEY `source_id` (`source_id`),
  ADD KEY `parameter_id` (`parameter_id`),
  ADD KEY `user_id` (`user_id`);
```

```
--
-- Indexes for table `AirQualityParameter`
--
ALTER TABLE `AirQualityParameter`
  ADD PRIMARY KEY (`parameter_id`);

--
-- Indexes for table `AirQualitySource`
--
ALTER TABLE `AirQualitySource`
  ADD PRIMARY KEY (`source_id`);

--
-- Indexes for table `City`
--
ALTER TABLE `City`
  ADD PRIMARY KEY (`city_id`);

--
-- Indexes for table `Location`
--
ALTER TABLE `Location`
  ADD PRIMARY KEY (`location_id`),
  ADD KEY `city_id` (`city_id`);

--
-- Indexes for table `MeasurementChanges`
--
ALTER TABLE `MeasurementChanges`
  ADD PRIMARY KEY (`change_id`),
  ADD KEY `measurement_id` (`measurement_id`);

--
-- Indexes for table `User`
--
ALTER TABLE `User`
  ADD PRIMARY KEY (`user_id`);

--
-- AUTO_INCREMENT for dumped tables
--
```

```
--
-- AUTO_INCREMENT for table `AirQualityMeasurement`
--
ALTER TABLE `AirQualityMeasurement`
  MODIFY `measurement_id` int(11) NOT NULL AUTO_INCREMENT,
AUTO_INCREMENT=6562;

--
-- AUTO_INCREMENT for table `AirQualityParameter`
--
ALTER TABLE `AirQualityParameter`
  MODIFY `parameter_id` int(11) NOT NULL AUTO_INCREMENT,
AUTO_INCREMENT=10;

--
-- AUTO_INCREMENT for table `AirQualitySource`
--
ALTER TABLE `AirQualitySource`
  MODIFY `source_id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=10;

--
-- AUTO_INCREMENT for table `City`
--
ALTER TABLE `City`
  MODIFY `city_id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=21;

--
-- AUTO_INCREMENT for table `Location`
--
ALTER TABLE `Location`
  MODIFY `location_id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=33;

--
-- AUTO_INCREMENT for table `MeasurementChanges`
--
ALTER TABLE `MeasurementChanges`
  MODIFY `change_id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=3;

--
-- AUTO_INCREMENT for table `User`
--
```

```
ALTER TABLE `User`
  MODIFY `user_id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=15;

--
-- Constraints for dumped tables
--

--
-- Constraints for table `AirQualityMeasurement`
--
ALTER TABLE `AirQualityMeasurement`
  ADD CONSTRAINT `airqualitymeasurement_ibfk_1` FOREIGN KEY (`location_id`)
REFERENCES `Location` (`location_id`),
  ADD CONSTRAINT `airqualitymeasurement_ibfk_2` FOREIGN KEY (`source_id`)
REFERENCES `AirQualitySource` (`source_id`),
  ADD CONSTRAINT `airqualitymeasurement_ibfk_3` FOREIGN KEY (`parameter_id`)
REFERENCES `AirQualityParameter` (`parameter_id`),
  ADD CONSTRAINT `airqualitymeasurement_ibfk_4` FOREIGN KEY (`user_id`)
REFERENCES `User` (`user_id`);

--
-- Constraints for table `Location`
--
ALTER TABLE `Location`
  ADD CONSTRAINT `location_ibfk_1` FOREIGN KEY (`city_id`) REFERENCES `City`
(`city_id`);

--
-- Constraints for table `MeasurementChanges`
--
ALTER TABLE `MeasurementChanges`
  ADD CONSTRAINT `measurementchanges_ibfk_1` FOREIGN KEY
(`measurement_id`) REFERENCES `AirQualityMeasurement` (`measurement_id`);
COMMIT;



--
-- Some elements cannot be exported from MySQL, these will be copied below
--
```

```
DELIMITER //

-- Create procedure: CalculateAverageMeasurement
CREATE PROCEDURE CalculateAverageMeasurement(IN parameterId INT, OUT
average DECIMAL(10,2))
BEGIN
    SELECT AVG(measurement_value) INTO average
    FROM AirQualityMeasurement
    WHERE parameter_id = parameterId;
END //

-- Create function: GetUserCity
CREATE FUNCTION GetUserCity(userId INT) RETURNS VARCHAR(255)
BEGIN
    DECLARE userCity VARCHAR(255);
    SELECT city INTO userCity
    FROM User
    WHERE user_id = userId;
    RETURN userCity;
END //

-- Create view: MeasurementDetails
CREATE VIEW MeasurementDetails AS
SELECT m.measurement_id, m.measurement_value, p.parameter_name,
l.location_name
FROM AirQualityMeasurement AS m
JOIN AirQualityParameter AS p ON m.parameter_id = p.parameter_id
JOIN Location AS l ON m.location_id = l.location_id //

DELIMITER ;
```