Fundamentals of Relational Databases and SQL

Learning Outcomes

After completing this unit, students will be able to:

- Explain the fundamental concepts of relational databases and their role in managing structured data.
- Describe key components of database systems: tables, relationships, keys, constraints.
- Understand normalization principles and apply them to improve database designs.
- Write SQL queries using SELECT, WHERE, GROUP BY, HAVING, JOIN, ORDER BY, INSERT, UPDATE, DELETE.
- Design basic relational schemas for environmental datasets (e.g., air quality, noise monitoring).
- Analyze environmental datasets using SQL (aggregation, filtering, joins, subqueries).
- Interpret query results and prepare short analytical reports.

Lecture Content

Introduction

In digital systems, data is a core asset. Environmental agencies, research groups, and governmental institutions collect large volumes of observational data: atmospheric measurements, meteorological readings, water quality indicators, emissions inventories, etc. Without a structured approach, such data becomes fragmented, inconsistent, and difficult to analyze.

Relational databases organize information into structured tables, where each table represents a specific type of entity (e.g., *Stations*, *Measurements*, *Users*).

The data model is based on the **relational model** introduced by E. F. Codd, which ensures consistency and mathematical soundness.

Key characteristics:

1. Table-based Structure

- Data is stored in tables (relations) with rows (tuples) and columns (attributes).
- Every table has a defined schema, specifying attribute names, data types, and constraints.

2. Strict Schema

- All data must conform to the predefined structure.
- Each column has a defined domain (*integer*, *text*, *date*, etc.).
- Constraints such as PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, and CHECK guarantee data integrity.

3. Structured and Consistent Data

- Records must follow the schema exactly. Example: A table *AirQuality(station_id, datetime, pollutant, value)* requires every row to contain these attributes with the correct data types.

4. Relationships Between Tables

- Tables can reference each other using keys, enabling normalized design and preventing redundancy.

Because of this structure, relational databases work best for well-defined, structured data where consistency is critical - such as financial systems, scientific datasets, inventory management, and environmental monitoring.

Technology Overview

Core Concepts of Relational Databases

Tables and Schemas

In the relational model, all data is organized into logical structures called **tables**, also known formally as *relations (Table.1)*. A table represents a single kind of entity-such as a monitoring station, a pollution measurement, or a weather observation-and provides a uniform, structured way to store multiple instances of that entity. Each table is composed of **rows** and **columns**. The rows, or *tuples*, capture individual records, while the columns, or *attributes*, describe the specific properties of each record.

Table.1 Elements of the Relation Model and Their Representation

Element of the Relational Model	Form of Representation
Relation	Table
Tuple	Row of a table

Attribute	Column header of a table
Key	A set of attributes that uniquely identifies each row in a table, or is used to connect tables, or improves the efficiency of operations on tables
Domain	The set of all possible values an attribute may take
Relation Schema	The list of attribute names (table column headers) that define the structure of the relation

For example, consider a table used in an environmental monitoring system:

AirQuality(station_id, datetime, pollutant, value, unit).

Here, every row in *AirQuality* represents a single measurement taken at a particular moment, at a particular station, and for a particular pollutant. The table enforces a consistent structure: all measurements must include a station identifier, a timestamp, a pollutant name, a numerical value, and its corresponding unit.

The structure of a table is governed by a **schema**. A schema defines the attributes of each table, their data types, and the rules they must follow. It serves as a blueprint for the database, ensuring that stored data is both meaningful and interpretable. In a broader sense, a schema can describe the entire database, specifying all tables as well as the relationships and constraints that tie them together. This explicit structure is one of the primary strengths of relational databases: it ensures predictability, reduces ambiguity, and supports robust querying and analysis.

For example, a database schema for environmental monitoring may contain:

- *Stations (station_id, name, region, latitude, longitude)*
- *AirQuality (station_id, datetime, pollutant, value, unit)*
- *Weather (station_id, datetime, temperature, humidity)*

Keys

The relational model relies heavily on the concept of keys to identify records uniquely and to maintain logical connections between tables.

A **Primary Key** (PK) is an attribute-or a combination of attributes-that uniquely identifies each row in a table. No two rows may share the same primary key value, and primary keys cannot

contain NULLs. For instance, a Stations table may use station_id as its primary key, guaranteeing that every station in the dataset has a unique identifier. In more complex tables such as AirQuality, uniqueness may be achieved through a composite key, such as the combination of (station_id, datetime, pollutant).

A **Foreign Key** (FK) creates a logical relationship between two tables. It is an attribute in one table that refers to the primary key of another table. This ensures that data remains consistent across the database. For example, every station_id in AirQuality must correspond to an existing station_id in Stations. Without foreign keys, the database could accumulate invalid references-measurements from stations that do not exist or that have been deleted.

Candidate Keys are attributes that could potentially serve as a primary key because they are capable of uniquely identifying rows. A table may have several candidate keys, but only one is chosen to be the primary key. Finally, Composite Keys are keys formed by multiple attributes, often used when no single attribute can guarantee uniqueness on its own.

Keys are essential for efficiency, reliability, and meaningful interpretation of data. They support indexing, prevent duplication, and enable linking across tables-capabilities that are indispensable in large-scale environmental databases.

Relationships

Relationships describe how data in one table connects to data in another. The relational model defines several kinds of binary relationships, each reflecting different real-world associations.

- **One-to-One (1:1)**
  In a one-to-one relationship, each record in one table is linked to at most one record in another table. This pattern is relatively rare but useful when separating optional, sensitive, or infrequently accessed information into a separate table.
- **One-to-Many (1:N)**
  One-to-many is the most common and practical relationship in relational databases. In this structure, a single row in one table can be associated with multiple rows in another. For example, one environmental monitoring station may collect hundreds of air quality measurements per day. In this case:
  Stations is on the “*one*” side and *AirQualityMeasurments* is on the “*many*” side

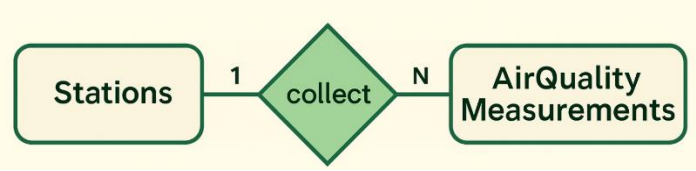


Fig.1 Chen notation Entity-Relationship Diagram example for relation 1:N

Such relationships allow the database to avoid redundancy-station metadata is stored once but referenced many times.

- **Many-to-Many (M:N)**

  In a many-to-many relationship, multiple rows in one table can relate to multiple rows in another. Relational databases do not support this relationship directly; instead, it is implemented through a junction table that stores pairings of foreign keys.

  For example, if stations can monitor multiple pollutants and pollutants can be measured at multiple stations, the relationship is many-to-many (*Fig.2*). A junction table like *StationPollutant(station_id, pollutant)* captures these associations.

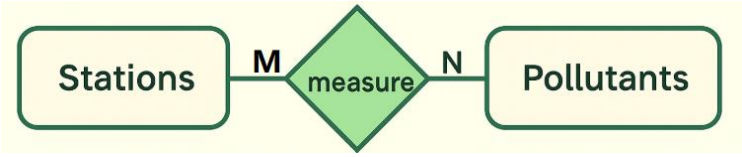


Fig.2 Chen notation Entity-Relationship Diagram example for relation M:N

Understanding relationships is crucial for designing normalized schemas and writing correct SQL queries, especially joins.

Cardinality of a relationship reflects the number of instances of one entity that can be associated with a single instance of another related entity.
In the Chen notation, cardinality is represented by specifying corresponding numbers next to the entities. Cardinality therefore shows how many instances of the related entity may participate in the relationship.

Attributes

An attribute represents a property or characteristic of an entity. Each attribute describes some aspect of the entity and holds values that belong to a predefined set. The set of all possible values an attribute may take is called its **domain**.

Different types of attributes are used in conceptual database design depending on their structure, meaning, and behavior. The main attribute types are shown at Fig. 3 and described in Table 2

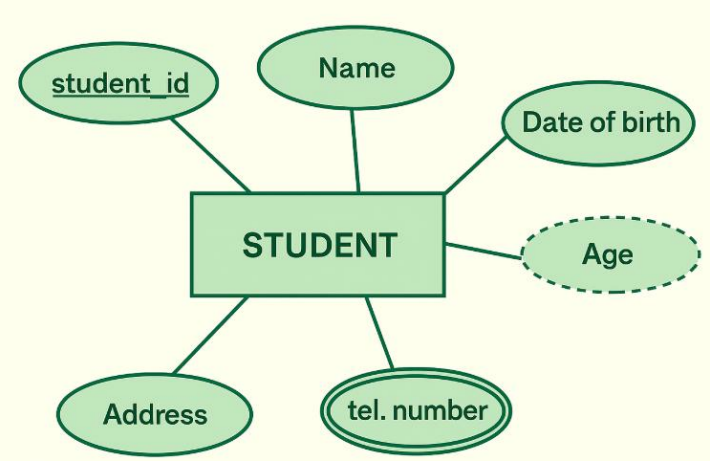


Fig.3 Chen notation Entity-Relationship Diagram example for Student entity

Table 2 Types of Attributes

Attribute Type	Description	Examples
Simple (Atomic) Attribute	An attribute that cannot be subdivided into smaller meaningful components.	*student_id, Last name*
Composite Attribute	An attribute that can be broken down into smaller sub-attributes, each with its own meaning.	*Address → street, building, apartment;* *Full name → last name, first name*
Single-Valued Attribute	An attribute that holds only one value for each entity instance.	*student_id, Passport number*
Multi-Valued Attribute	An attribute that may contain multiple values for the same entity instance.	*Phone numbers (mobile, home), Email addresses*
Derived Attribute	An attribute whose value is calculated from other stored data and is not kept directly in the database.	*Age (from Date of Birth), Number of credits (from list of courses)*

Key Attribute	An attribute or group of attributes that uniquely identifies an entity instance.	*Primary Key: Student ID; Candidate Keys: Email, Record book number*

Integrity Constraints

Integrity constraints are rules that the database enforces automatically to preserve accuracy, consistency, and reliability. They prevent incorrect data from entering the system and ensure that relationships between tables remain valid. Several common types of constraints include:

- **NOT NULL**
  A NOT NULL constraint ensures that a column must always contain a value. This is essential for attributes that are required for analysis or identification. For example, environmental measurement timestamps must never be NULL.
- **UNIQUE**
  The UNIQUE constraint ensures that no two rows contain the same value in a given attribute. It is useful for identifiers such as serial numbers, usernames, or pollutant codes.
- **CHECK**
  CHECK constraints ensure that attribute values satisfy a logical condition.
  For example, *CHECK* (*value* ≥ *0)* ensures that pollutant concentrations cannot be recorded as negative-a scenario that would be scientifically invalid.
- **Referential Integrity** (via Foreign Keys)
  This constraint ensures that every foreign key value matches an existing primary key value in the referenced table. It also prevents deletion of referenced rows unless dependent records are handled appropriately. Referential integrity is vital for keeping complex datasets coherent, especially when tables are interconnected through keys and relationships.

Stages of Database Design

Database design is a structured, multi-step process that transforms real-world information needs into an efficient and reliable data management system. Although different methodologies may present variations, the classical approach includes four fundamental stages:

1. System Analysis of the Application Domain

2. Conceptual Design
3. Logical Design
4. Physical Design

Each stage addresses a different level of abstraction and contributes to the creation of a robust, scalable, and well-structured database system.

The process of designing a database begins with **system analysis of the application domain**, which serves as the foundation for all subsequent design stages. During system analysis, the designer develops a deep understanding of the real-world environment in which the future database will operate. This requires studying the business processes, scientific domain, or organizational context in detail in order to determine what information the system must store and how that information will be used. System analysis also involves identifying stakeholders – individuals, groups, or departments that will rely on the database – and clarifying their goals, data needs, and expectations. To gather these insights, designers examine existing documents, reports, forms, workflows, and technological tools. They analyze how data is currently created, processed, transmitted, and consumed. The outcome of this stage is a comprehensive and structured description of the problem domain, including the key objects that exist within it, the properties they possess, the relationships between them, and the rules that govern their behavior. This understanding forms the conceptual groundwork upon which the entire database design process is built.

Once the domain has been thoroughly analyzed, the process proceeds to **conceptual design**, which focuses on representing the structure of the data at a high level of abstraction, independent of any database technology. The goal of conceptual design is to construct a conceptual data model – most commonly expressed using Entity–Relationship (ER) diagrams or UML class diagrams – that accurately reflects the semantics of the domain. During this stage, the designer identifies the entities that appear in the domain, defines the attributes that describe them, and establishes the relationships that link these entities together, including their cardinalities and optionalities. Conceptual design also incorporates the business rules discovered during system analysis so that the future database mirrors real-world constraints and processes. At this point, the central question is what information the database should represent, without concern for how this information will eventually be implemented or stored.

The next phase, **logical design**, transforms the conceptual model into a structure appropriate for implementation in a specific type of database system, typically a relational DBMS. Logical design is dependent on the chosen data model but remains independent of physical storage

considerations. Here, the abstract conceptual model is translated into relational structures: tables, primary keys, candidate keys, foreign keys, and well-defined attributes with their corresponding data types. Logical design also includes the process of normalization, which systematically eliminates redundancy and prevents update, insertion, and deletion anomalies. Through restrictions and integrity constraints, the logical schema ensures that the data will remain accurate and consistent. Ultimately, logical design answers the question of how the conceptual representation of the domain will be realized within the data model of the selected DBMS.

The final stage of the design process is **physical design**, which concerns the actual implementation of the logical model within a specific hardware and software environment. Physical design focuses on performance, efficiency, and data management strategies. During this stage, the designer determines how the data will be stored and accessed, selecting storage structures, file organizations, indexing strategies, and partitioning or clustering methods that support expected workloads. Physical considerations also include configuring memory allocation, optimizing access paths and queries, and planning for backup, archiving, and recovery procedures to ensure reliability and fault tolerance. Security measures such as access control, encryption, and audit mechanisms may also be incorporated. Physical design thus adapts the logical schema to the operational constraints and performance characteristics of the DBMS, addressing the practical question of how the database will be stored and accessed efficiently on the physical system.

Table 3 Database design stages summary

Stage	Key Question	Result
System Analysis	*What is the domain? What do users need?*	Requirements, business rules, domain understanding
Conceptual Design	*An attribute that can be broken down into smaller sub-attributes, each with its own meaning.*	ER diagram or conceptual schema
Logical Design	*How to represent this data in a DBMS data model?*	Relational schema: tables, keys, constraints
Physical Design	*How to store and access the data efficiently?*	Indexes, partitions, storage configuration

Normalization

Normalization is a systematic method used in relational database design to organize data into well-structured tables that minimize redundancy and prevent inconsistencies. The technique ensures that each fact is stored once and only once, and that every attribute in the database depends on the key, the whole key, and nothing but the key. Normalization brings clarity, reduces storage waste, and protects the database from update, insertion, and deletion anomalies that can distort analytical results.

Although normalization is a theoretical construct rooted in set theory and relational algebra, its value becomes especially evident in practical domains such as environmental data management. Environmental datasets – including air quality measurements, meteorological readings, pollutant categories, and station metadata – are often large, repetitive, and interdependent. Without normalization, duplication of station information, pollutant definitions, or unit descriptions can quickly lead to inconsistencies across datasets and reporting pipelines.

To systematically eliminate redundancy, normalization is performed step by step using a sequence of normal forms, each imposing stricter rules on how data can be organized. The most important of these, and the ones used in practical database design, are the First, Second, and Third Normal Forms.

First Normal Form (1NF)

First Normal Form requires that all data in a table be **atomic**, meaning that each cell contains a single, indivisible value. Repeating groups, nested tables, or multiple values in a single field are not permitted.

For example, a poorly designed environmental dataset might store multiple pollutants in one column:

station_id	pollutants_measured
101	NO2, PM2.5, O3

Such representation makes querying, filtering, and indexing extremely difficult. In 1NF, each pollutant must be separated into its own row or moved to a related table. After normalization, the structure becomes clear, queryable, and consistent:

station_id	pollutants_measured

101	NO2
101	PM2.5
101	O3

By ensuring data atomicity, 1NF lays the foundation for all higher forms.

Second Normal Form (2NF)

Second Normal Form applies only to tables with **composite primary keys** (keys formed by more than one attribute). It requires that every non-key attribute depend on the *entire* primary key, not just part of it. In other words, partial dependencies are not allowed.

Consider a table that stores air-quality measurements, keyed by *(station_id, datetime)*:

applies only to tables with **composite primary keys** (keys formed by more than one attribute). It requires that every non-key attribute depend on the *entire* primary key, not just part of it. In other words, **partial dependencies** are not allowed.

Consider a table that stores air-quality measurements, keyed by *(station_id, datetime)*:

station_id	**datetime**	**pollutant**	**value**	**station_name**
101	2025-01-15 09:00	NO2	40	Zagreb Center

2NF requires moving such attributes into a separate table – in this case, *Stations*. After normalization:

AirQualityMeasurements(station_id, datetime, pollutant, value)

Stations(station_id, station_name)

This structure eliminates redundancy and ensures that each attribute appears in the correct place.

Third Normal Form (3NF)

Third Normal Form eliminates **transitive dependencies**, meaning no non-key attribute may depend on another non-key attribute. Attributes must depend only on the primary key and not on other non-key attributes.

To illustrate Third Normal Form using an environmental dataset, consider a table that stores information about weather stations and the regions they belong to:

station_id	station_name	region_name	region_manager
201	Riverbank West	Central Basin	Dr. Alicia Moreno
202	Hilltop South	Mountain Highlands	Eng. Robert Davies

The primary key is: station_id. The attributes *station_name, region_name*, and *region_manager* are all non-key attributes. *region_manager* does not depend directly on *station_id*. Instead, it depends on *region_name* – another non-key attribute. This creates a **transitive dependency**:

station_id → region_name

region_name → region_manager.

This violates Third Normal Form.

To resolve the transitive dependency and achieve Third Normal Form, we split the table into two related tables:

station_id (PK)	station_name	region_name
201	Riverbank West	Central Basin
202	Hilltop South	Mountain Highlands

region_name (PK)	region_manager
Central Basin	Dr. Alicia Moreno
Mountain Highlands	Eng. Robert Davies

The database is now in Third Normal Form:

- *station_id* uniquely identifies each station
- *region_name* uniquely identifies each region
- *region_manager* depends only on its key (*region_name*)
- No non-key attribute depends on another non-key attribute

Normalization plays a crucial role in preserving the integrity and reliability of a database by preventing the logical anomalies that often arise when data is stored in an unstructured or

redundant manner. These anomalies occur because a single fact may appear repeatedly across multiple rows, or because different types of information are improperly combined within a single table. In environmental information systems-where measurements, station metadata, regional classifications, and regulatory attributes interact continuously – such inconsistencies can quickly undermine the accuracy of analyses and reports.

One common problem is the **update anomaly**, which occurs when a change must be made to repeated information. If the name of a monitoring station or the designation of a region appears in numerous rows, every instance must be updated consistently. A single forgotten update can produce conflicting records, causing a system to refer to the same station by two different names or to assign a region multiple managers. These inconsistencies propagate into reports, dashboards, and downstream modelling tasks.

A second issue is the **insertion anomaly**, which arises when certain information cannot be entered into the database because other, unrelated information is not yet available. For example, it may be impossible to add a new pollutant or a new environmental region unless at least one measurement already exists for it. This artificially restricts the order in which data may be created and forces the system to violate natural workflows.

The **deletion anomaly** represents the opposite problem: removing the last occurrence of some operational data can inadvertently remove important descriptive information. If station metadata or region details are stored alongside measurement data, deleting the final measurement for a station might eliminate the station's description entirely, even though the station still exists in the real world.

By systematically organizing information into a set of normalized tables, these anomalies are eliminated. Each fact is stored exactly once, in the table where it logically belongs, and all other tables refer to it through keys rather than by duplication. As a result, the database becomes internally consistent, easier to maintain, and more robust against erroneous updates or deletions. For environmental applications – such as long-term air quality monitoring, watershed assessment, or ecological modelling – normalization ensures that datasets remain reliable, traceable, and analytically sound. It safeguards the continuity of scientific observations across years, supports regulatory compliance, and provides a stable foundation for large-scale environmental modeling and decision-making.

Structured Query Language (SQL)

Structured Query Language, is the standard language used to interact with relational database management systems. It enables users to define the structure of data, manipulate stored information, regulate access to database objects, and manage transactions. Although SQL is widely known for its ability to retrieve data, it is, in fact, a rich language composed of several specialized subsets, each serving a distinct purpose within the lifecycle of database operations. These subsets are traditionally classified into four main categories: **DDL, DML, DCL, and TCL**.

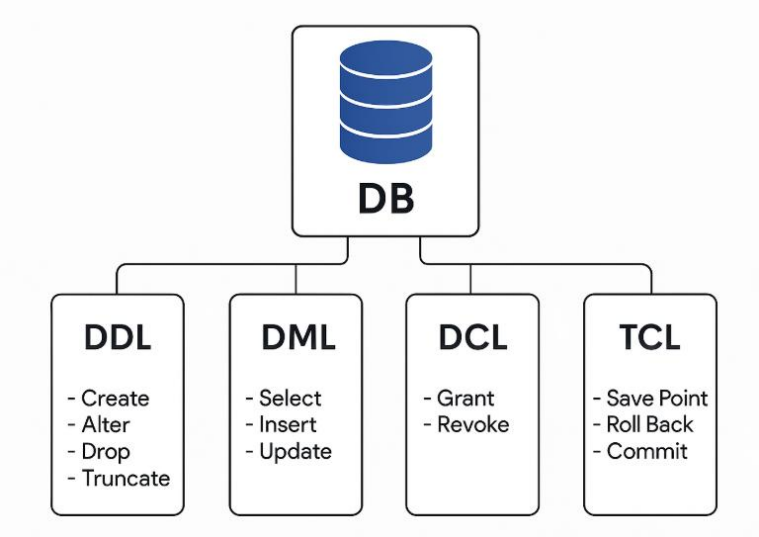


Fig.4 SQL Structure

1. Data Definition Language (DDL)

The Data Definition Language is responsible for describing and modifying the structure of database objects. DDL statements define the schema – the blueprint – of the database. They specify what tables exist, what attributes they contain, and how these objects relate to one another.

- **CREATE** is used to construct a new database object, such as a table, a view, an index, or even an entire database. When a designer defines a new table for storing environmental measurements or adds a view summarizing pollutant trends, the CREATE statement is involved.

```
CREATE TABLE Stations (
    station_id INT PRIMARY KEY,
    name VARCHAR(100),
```

```
    region VARCHAR(50)
);
```

- **ALTER** modifies an existing database object. This may include adding a new column to a table when additional environmental attributes are required, adjusting a data type to accommodate more precise measurements, or altering constraints to enforce updated data rules.

```
ALTER TABLE AirQuality ADD COLUMN unit VARCHAR(20);
```

- **DROP** removes a database object entirely. When a table, an index, or a view is no longer needed-perhaps after a redesign or data consolidation-the DROP command permanently deletes it from the schema.

```
DROP TABLE TempData;
```

DDL statements shape the structural framework of the database and determine the foundation upon which data will be stored and accessed.

2. Data Manipulation Language (DML)

While DDL builds the structure, the Data Manipulation Language handles the day-to-day operations performed on the data itself. These commands allow users to insert new information, retrieve existing records, make modifications, and remove data that is outdated or incorrect.

- **INSERT** adds new records to a table. As new sensor readings arrive from air quality stations or new metadata is entered for additional monitoring sites, INSERT statements populate the database.

```
INSERT INTO Stations VALUES (101, 'Zagreb-Center', 'Zagreb');
```

- **UPDATE** modifies existing records. If a station's metadata changes-such as an updated altitude, corrected coordinates, or a change in operator-UPDATE ensures that the stored information remains accurate.

```
UPDATE AirQuality
SET value = NULL
WHERE value < 0;
```

- **DELETE** removes records from a table. It is used when erroneous entries must be eliminated or when obsolete data is no longer needed.

```
DELETE FROM Stations WHERE region = 'Unknown';
```

- **SELECT** retrieves data from one or more tables, often applying filters, sorting, grouping, and aggregation. Environmental analysts might use SELECT to extract measurements that exceed regulatory thresholds or to compare pollutant levels across monitoring stations.

```
SELECT station_id, pollutant, value
FROM AirQuality
WHERE pollutant = 'PM2.5';
```

  - ***Ordering***

```
ORDER BY value DESC;
```

  - ***Filtering with logical operators***

```
WHERE value > 50 AND pollutant = 'NO2';
```

  - ***Aggregation and GROUP BY (Used for environmental reporting)***

```
SELECT station_id, pollutant, AVG(value) AS avg_value
FROM AirQuality
GROUP BY station_id, pollutant;
```

  - ***HAVING - filtering aggregated data***

```
HAVING AVG(value) > 40;
```

  - ***INNER JOIN***

```
SELECT s.name, a.pollutant, a.value
```

```
FROM AirQuality a
JOIN Stations s ON a.station_id = s.station_id;
```

- ***LEFT JOIN***

  Scenario: we want to list all monitoring stations, including those that do not yet have any measurements recorded. Table 1: *Stations(station_id, station_name).* Table 2: *AirQualityMeasurements(station_id, pollutant, value)*

```
SELECT s.station_id, s.station_name, m.pollutant, m.value
FROM Stations AS s
LEFT JOIN AirQualityMeasurements AS m
ON s.station_id = m.station_id;
```

- ***RIGHT JOIN***

  Scenario: we want to list all pollutants measured, even if some pollutants are not assigned to any station yet. Table 1: *AirQualityMeasurements(station_id, pollutant, value).* Table2: *Pollutants(pollutant, description)*

```
SELECT m.station_id, p.pollutant, p.description, m.value
FROM AirQualityMeasurements AS m
RIGHT JOIN Pollutants AS p
ON m.pollutant = p.pollutant;
```

- **Subqueries**

  Iterative and nested analyses, e.g., exceeding EU limits:

```
SELECT *
FROM AirQuality
WHERE value > (
      SELECT threshold
      FROM EU_Standards
      WHERE pollutant = AirQuality.pollutant
);
```

DML forms the operational core of SQL, enabling users to work directly with the stored information.

3. Data Control Language (DCL)

The Data Control Language regulates access to database objects and governs permissions. In multi-user database environments-particularly in scientific and governmental systems where

environmental data must be shared but also protected-DCL ensures that only authorized individuals can perform specific actions.

- **GRANT** assigns permissions to users or roles, allowing them to perform defined operations on certain objects. For instance, researchers may be granted permission to read pollutant data but not to modify station metadata.

```
GRANT SELECT
ON AirQualityMeasurements
TO researcher_1;
```

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON Stations
TO env_admin_role;
```

- **REVOKE** removes previously granted permissions. This is essential when responsibilities change or when data access must be restricted for security or compliance reasons.

```
REVOKE INSERT
ON AirQualityMeasurements
FROM researcher_1;
```

- **DENY** specifically forbids certain actions, overriding any permissions that might otherwise allow them. It is a strict measure that enforces security policies even if multiple permission layers are in place.

```
DENY DELETE
ON Pollutants
TO researcher_2;
```

DCL statements thus help maintain confidentiality, data integrity, and controlled collaboration across teams.

4. Transaction Control Language (TCL)

The Transaction Control Language ensures that groups of related DML operations behave atomically-that is, they either succeed entirely or fail without leaving partial changes. Transactions are especially important when handling interconnected environmental datasets where inconsistencies may compromise analytical outcomes.

- **COMMIT** finalizes a transaction, permanently applying all changes made since the last commit. Once committed, inserted measurements, updated metadata, or deleted records become part of the official dataset.

```
BEGIN TRANSACTION;

INSERT INTO AirQualityMeasurements (station_id, pollutant, value)
VALUES (101, 'PM2.5', 23.5);

COMMIT;
```

- **ROLLBACK** undoes all changes made during the current transaction. If an error occurs while updating multiple tables-for example, when linking a new monitoring station to several reference tables-ROLLBACK ensures the database returns to a stable state.

```
BEGIN TRANSACTION;

UPDATE AirQualityMeasurements
SET value = -10      -- invalid value
WHERE measurement_id = 455;

-- Error detected → revert everything
ROLLBACK;
```

- **SAVEPOINT** creates named markers within a transaction, allowing partial rollbacks. This is useful in complex operations such as importing large batches of sensor data, where the system may need to revert to an intermediate state rather than restart from the beginning.

```
BEGIN TRANSACTION;

UPDATE Stations
SET region = 'Central Basin'
WHERE station_id = 101;

SAVEPOINT sp1;

UPDATE Stations
SET region = 'Mountain Highlands'
WHERE station_id = 102;

-- Decide to undo only the second update
ROLLBACK TO sp1;

COMMIT;
```

Through TCL, SQL guarantees data consistency and reliability, even in complex, multi-step operations.

ACID Properties of Transactions

Environmental datasets – air quality measurements, meteorological observations, sensor logs, and regional metadata – are often interconnected and updated continuously. If even one part of

this data is corrupted or only partially updated, analytical results and long-term monitoring records may be compromised.

To ensure that multi-step database operations behave predictably and consistently, modern database systems use **transactions**-groups of operations that must be executed as a single, unified unit. The correctness of these transactions is guaranteed by four foundational principles known collectively as the **ACID properties**: Atomicity, Consistency, Isolation, and Durability.

1. Atomicity

Atomicity requires that a transaction be treated as an indivisible unit: either *all* of its operations succeed, or *none* of them take effect.

A transaction may include many steps-such as inserting a new measurement, updating sensor calibration records, and logging an event. Atomicity ensures that the database never ends up in a state where only some of these steps were completed. If an error occurs, or if the system fails mid-process, the entire transaction is rolled back to its previous stable state.

Example. An air quality station transmits a batch of measurements. The system must insert all measurements. If even one fails due to a format error, atomicity guarantees that *none of the records* are inserted, preventing partial and misleading data in the database.

Atomicity protects the integrity of complex operations and prevents partial updates that could distort environmental analyses.

2. Consistency

Consistency ensures that every transaction transforms the database from one valid state into another, preserving all defined rules, constraints, and relationships.

These rules include data types, primary and foreign keys, check constraints, and scientific validation logic. A consistent database always respects these rules before and after every transaction.

Example. If the database enforces a rule that pollutant concentrations must be non-negative, a transaction attempting to insert a value of -15 must fail. The database rejects invalid data to maintain scientific accuracy.

Consistency guarantees that environmental data never violates physical, regulatory, or logical constraints.

3. Isolation

In real-world systems, many users and processes access the database simultaneously. Isolation ensures that the execution of one transaction does not interfere with another. Until a transaction is complete, its intermediate results are hidden from others.

Without isolation, two transactions updating the same station metadata or inserting overlapping measurement sets could influence each other, leading to unpredictable results.

Example. Two researchers update different attributes of the *Stations* table at the same time. Isolation ensures that their updates do not conflict, overwrite each other, or produce inconsistent metadata.

Most database systems support several **isolation levels**, such as:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

Higher levels offer stronger guarantees but may reduce concurrency. In environmental systems that ingest large batches of sensor data, finding the right balance is essential.

4. Durability

Durability guarantees that once a transaction is committed, its results become permanent-even in the case of hardware failures, power loss, or system crashes.

Database systems implement durability through:

- write-ahead logs,
- checkpointing,
- redundant storage,
- automatic recovery mechanisms.

Example. After successfully inserting a day's worth of meteorological readings and issuing a COMMIT, the data must remain safely stored even if a failure occurs moments later. Durability ensures that environmental monitoring records remain intact over years of analysis and reporting.

The ACID properties – Atomicity, Consistency, Isolation, and Durability – ensure that every operation carried out on data is complete, valid, interference-free, and permanent.

Applied Use Case

For learning SQL, we will use the online tool **Oracle FreeSQL** (https://freesql.com/). Oracle FreeSQL provides students with free access to a real Oracle cloud database, allowing them to practice SQL and relational database concepts in a professional environment. To work in Oracle FreeSQL, you need to sign in to your account or, if you don't have one, sign up.

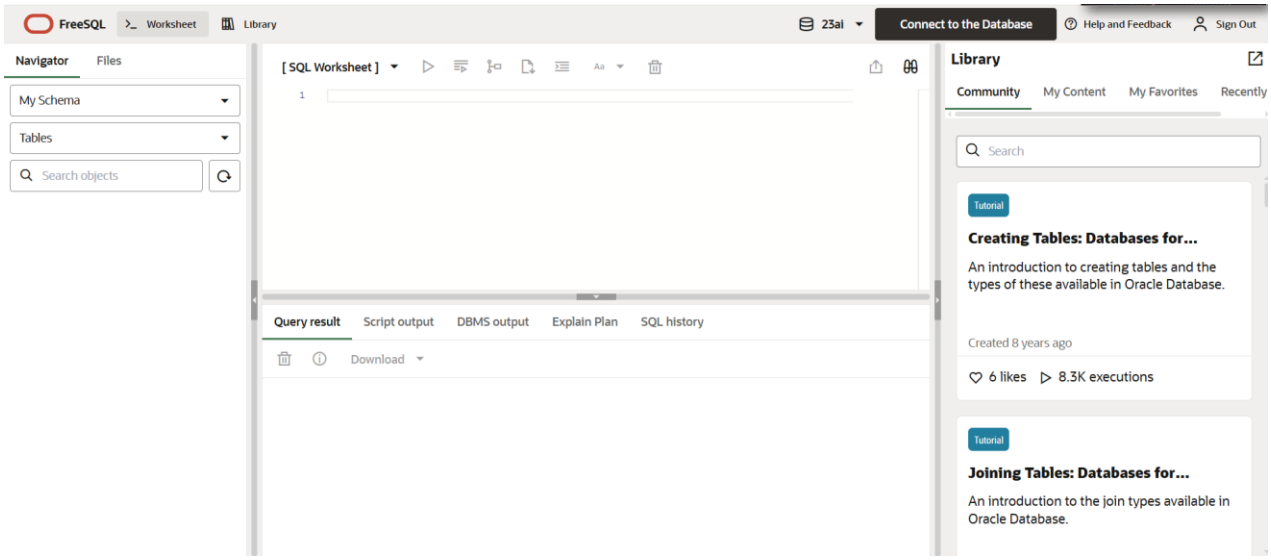


Fig.5 Oracle FreeSQL environment

Before practicing SQL, it is important to understand what kind of database we are going to build and why.

In this course, we construct a relational training database that models air quality monitoring in the city of Warsaw, Poland. The database is designed for educational purposes and is based on the logic of real-world environmental information systems used by public authorities and research institutions. While the data values are simplified for learning, the structure reflects realistic relationships and constraints.

The database represents how air pollution measurements are collected by monitoring stations, how these stations are described by metadata, and how pollutant types are defined and regulated. Each of these concepts is stored in a separate table, following the principles of the relational model.

At the core of the database are **monitoring stations**, which represent physical locations in Warsaw where air quality sensors are installed. Each station has an identifier, a name, a

geographic location, and descriptive attributes such as region and operator. Stations are stored independently so that their metadata can be maintained even if no measurements are available yet.

Another key component of the database is the **pollutant dictionary**. Pollutants such as nitrogen dioxide (NO_2), particulate matter (PM10, PM2.5), ozone (O_3), and sulfur dioxide (SO_2) are defined once and referenced consistently throughout the database. This approach avoids duplication, ensures data integrity, and enables meaningful comparisons across measurements.

The **measurement data** form the main factual part of the database. Each measurement records the concentration of a specific pollutant at a specific station and time. Measurements are linked to both stations and pollutants through relational keys, allowing queries that combine location, time, and pollutant type in a single result set.

In addition, the database contains a table with **regulatory threshold values**, representing simplified European air quality limits. These values allow analytical queries that detect exceedances and support comparisons between measured concentrations and accepted standards.

The overall structure follows the principles of **normalization** and **referential integrity**. Each real-world concept is represented by its own table, and relationships between tables are explicitly defined. This design ensures that the database is consistent, extensible, and suitable for analytical SQL queries involving joins, aggregation, and subqueries.

1. **CREATE SQL operations**

Let's create a database by creating tables step by step.

1.1 Create stations table writing SQL code into SQL Worksheet field and then press **Run Statement** button:

```
CREATE TABLE stations (
  station_id     NUMBER PRIMARY KEY,
  station_name   VARCHAR2(120) NOT NULL,
  country_code   CHAR(2) DEFAULT 'PL' NOT NULL,
  city           VARCHAR2(60) DEFAULT 'Warsaw' NOT NULL,
  region         VARCHAR2(60),
  latitude       NUMBER(9,6),
  longitude      NUMBER(9,6)
);
```

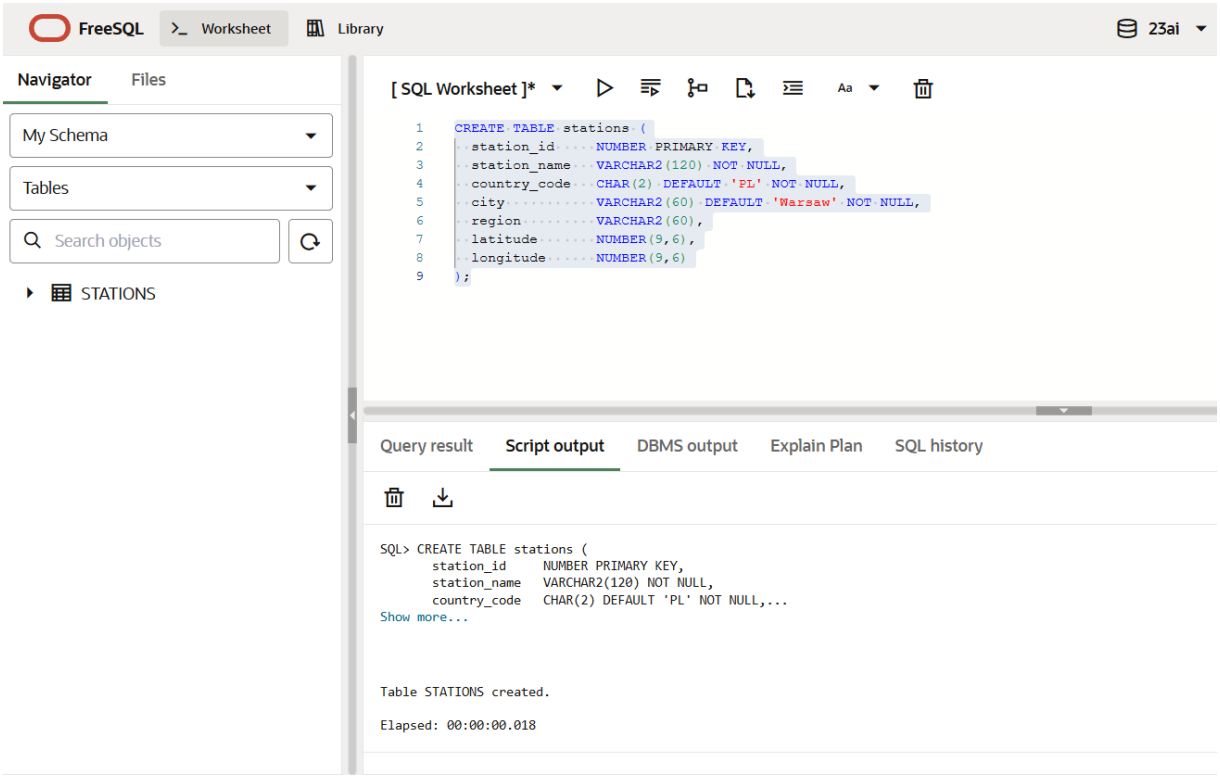


Fig.6 Creating stations table

1.2 Creating pollutants table

```
CREATE TABLE pollutants (
  pollutant_code VARCHAR2(20) PRIMARY KEY,
  description    VARCHAR2(200) NOT NULL,
  default_unit   VARCHAR2(20) NOT NULL
);
```

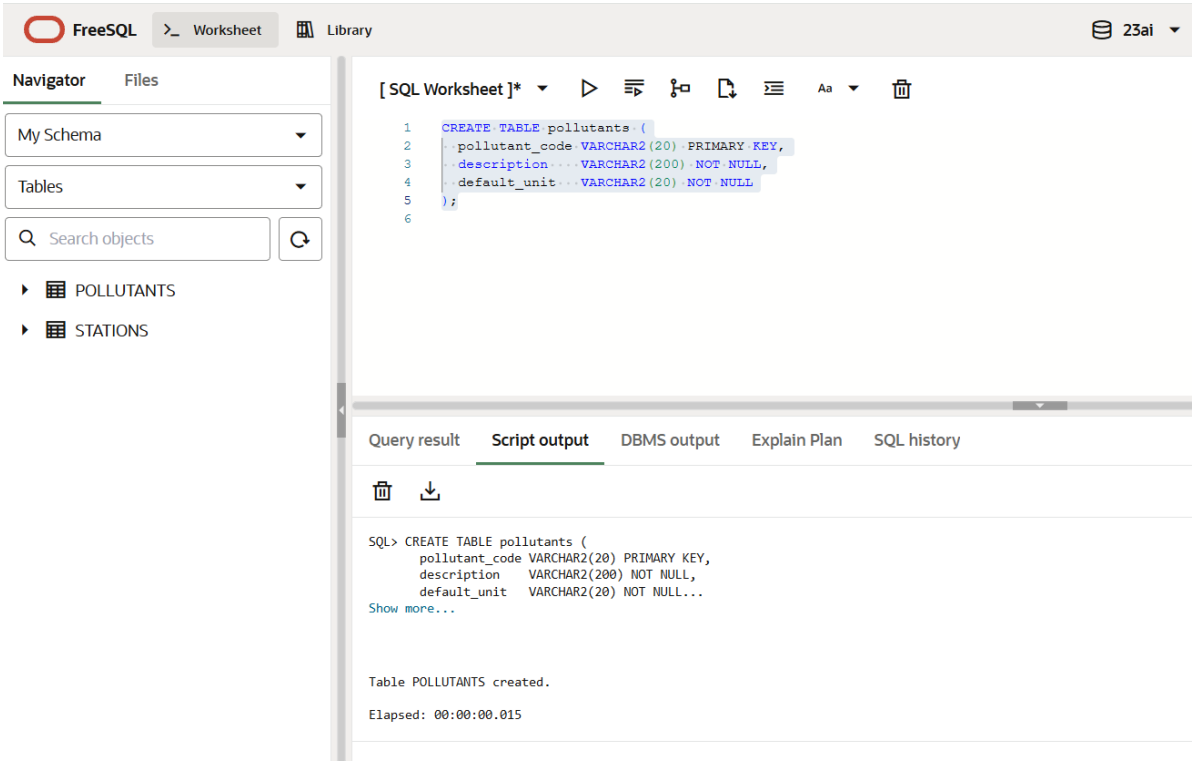


Fig.7 Creating pollutants table

1.3 Creating air quality measurements table:

```
CREATE TABLE air_quality_measurements (
  measurement_id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  station_id     NUMBER NOT NULL,
  pollutant_code VARCHAR2(20) NOT NULL,
  measurement_ts TIMESTAMP NOT NULL,
  value          NUMBER,
  unit           VARCHAR2(20),

  CONSTRAINT fk_station
    FOREIGN KEY (station_id)
    REFERENCES stations(station_id),

  CONSTRAINT fk_pollutant
    FOREIGN KEY (pollutant_code)
    REFERENCES pollutants(pollutant_code)
);
```

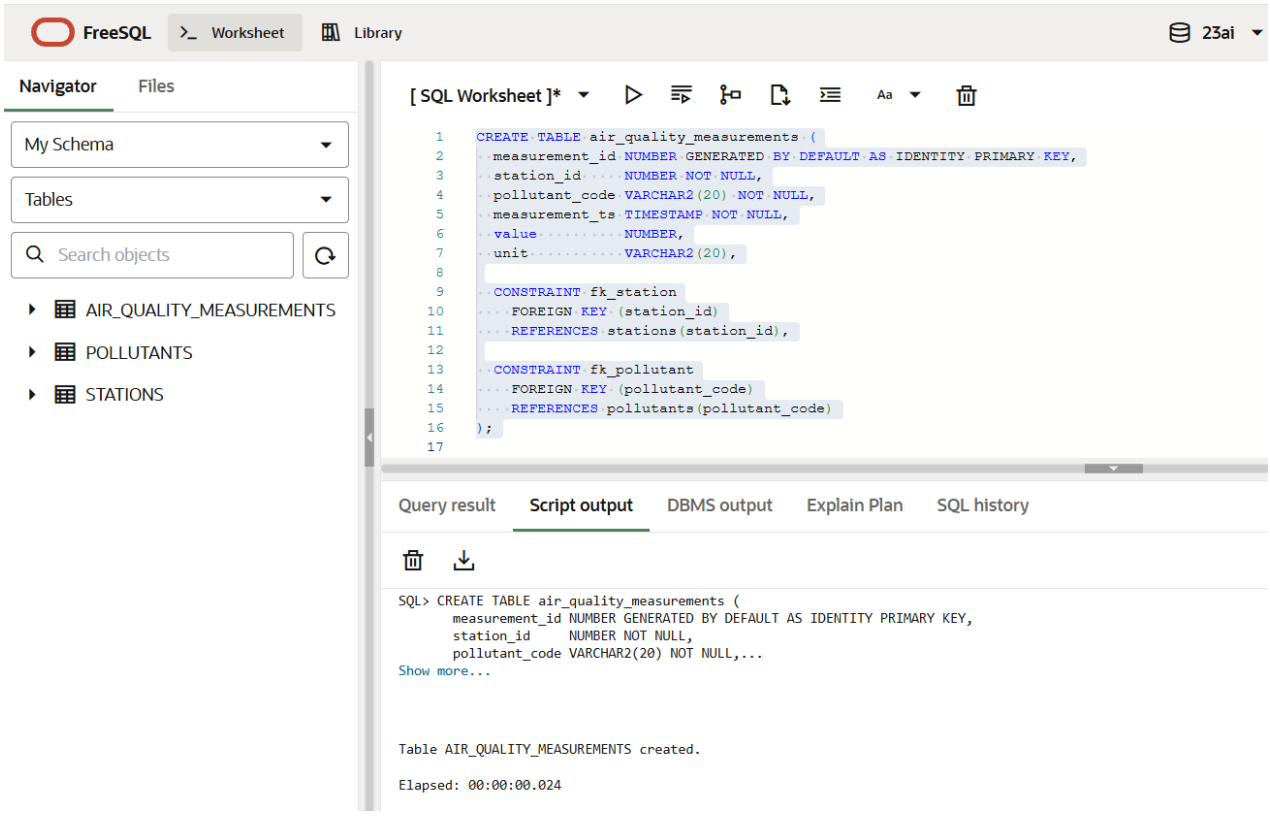


Fig.8 Creating pollutants table

At this stage, the database has been created. You can visually verify that the tables have appeared by seeing the table names in the Navigator field on the left: STATIONS, POLLUTANTS, AIR_QUALITY_MEASUREMENTS.

2. INSERT operations

Now let's fill in the tables with data.

2.1 Now we will insert data to the stations table:

```
INSERT INTO stations VALUES
(101, 'Warsaw-Center', 'PL', 'Warsaw', 'Mazowieckie', 52.229676, 21.012229);


INSERT INTO stations VALUES
(102, 'Warsaw-Traffic', 'PL', 'Warsaw', 'Mazowieckie', 52.239000, 21.005000);
```

```
INSERT INTO stations VALUES
(103, 'Warsaw-Suburb', 'PL', 'Warsaw', 'Mazowieckie', 52.300000, 20.900000);

INSERT INTO stations VALUES
(104, 'Warsaw-Park', 'PL', 'Warsaw', 'Mazowieckie', 52.260000, 21.020000);

-- Station without measurements (for LEFT JOIN testing)
INSERT INTO stations VALUES
(105, 'Warsaw-NewStation', 'PL', 'Warsaw', 'Mazowieckie', 52.215000, 21.040000);
```

After executing the command, right-clicking on the STATIONS table in the left field, we can open the table and see that the table has been filled in with data (fig.9).

STATIONS

Columns | Data | Constraints | Grants | Statistics | Triggers | Dependencies | Details | Partitions | Indexes

	STATION_ID	STATION_NAME	COUNTRY_CODE	CITY	REGION	LATITUDE	LONGITU
1	101	Warsaw-Center	PL	Warsaw	Mazowieckie	52.229676	2
2	102	Warsaw-Traffic	PL	Warsaw	Mazowieckie	52.239	
3	103	Warsaw-Suburb	PL	Warsaw	Mazowieckie	52.3	
4	104	Warsaw-Park	PL	Warsaw	Mazowieckie	52.26	
5	105	Warsaw-NewStation	PL	Warsaw	Mazowieckie	52.215	21.04

Fig.9 Result of filling in the stations table with data

2.2 Let's insert data about the pollutants:

```
INSERT INTO pollutants VALUES
```

```
('NO2', 'Nitrogen dioxide', 'ug/m3');

INSERT INTO pollutants VALUES
('PM25', 'Particulate matter < 2.5 µm', 'ug/m3');

INSERT INTO pollutants VALUES
('PM10', 'Particulate matter < 10 µm', 'ug/m3');

INSERT INTO pollutants VALUES
('O3', 'Ozone', 'ug/m3');

-- Pollutant without measurements (for testing RIGHT JOIN)
INSERT INTO pollutants VALUES
('SO2', 'Sulfur dioxide', 'ug/m3');
```

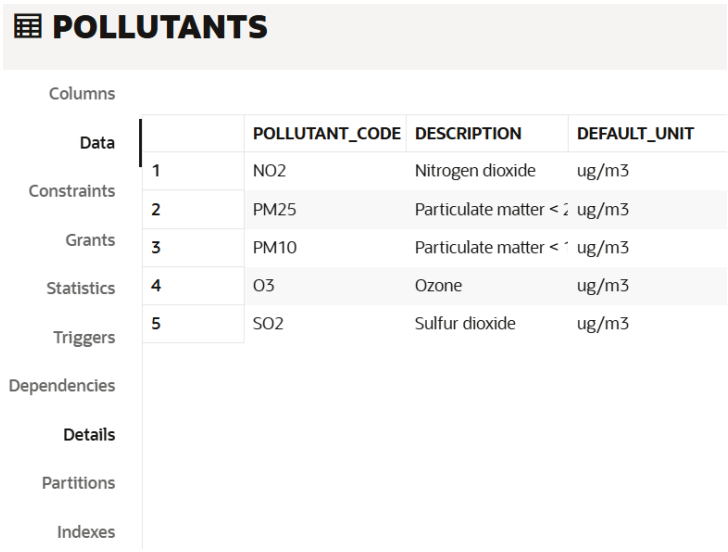


Fig.10 Result of filling in the pollutants data

2.3 Now let's insert data about measurements:

```
INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(101, 'NO2', TIMESTAMP '2025-03-01 08:00:00', 45, 'ug/m3');
```

```
INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(102, 'PM25', TIMESTAMP '2025-03-01 08:00:00', 28, 'ug/m3');

INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(101, 'NO2', TIMESTAMP '2025-03-01 09:00:00', 78, 'ug/m3');

INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(102, 'NO2', TIMESTAMP '2025-03-01 08:00:00', 210, 'ug/m3');

INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(103, 'PM25', TIMESTAMP '2025-03-01 08:00:00', 18, 'ug/m3');

INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(104, 'PM10', TIMESTAMP '2025-03-01 08:00:00', 62, 'ug/m3');

INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
VALUES
(104, 'O3', TIMESTAMP '2025-03-01 13:00:00', 130, 'ug/m3');

-- Incorrect measurement (for UPDATE and DELETE testing)
INSERT INTO air_quality_measurements
(station_id, pollutant_code, measurement_ts, value, unit)
```

```
VALUES
(101, 'PM25', TIMESTAMP '2025-03-01 10:00:00', -7, 'ug/m3');
```

AIR_QUALITY_MEASUREMENTS

Columns | Data | Constraints | Grants | Statistics | Triggers | Dependencies | Details | Partitions | Indexes

	MEASUREMENT_ID	STATION_ID	POLLUTANT_CODE	MEASUREMENT_TS	VALUE	UNIT
1	1	101	NO2	2025-03-01T08:00:(	45	ug/m3
2	2	102	PM25	2025-03-01T08:00:(	28	ug/m3
3	3	101	NO2	2025-03-01T09:00:(	78	ug/m3
4	4	102	NO2	2025-03-01T08:00:(	210	ug/m3
5	5	103	PM25	2025-03-01T08:00:(	18	ug/m3
6	6	104	PM10	2025-03-01T08:00:(	62	ug/m3
7	7	104	O3	2025-03-01T13:00:(	130	ug/m3
8	8	101	PM25	2025-03-01T10:00:(	-7	ug/m3

Fig.11 Result of filling in the measurements data

3. UPDATE AND DELETE operations

3.2 Using the UPDATE command, we can correct incorrect measurement values, replacing negative values with NULL and then use DELETE to remove all false entries:

```
UPDATE air_quality_measurements
SET value = NULL
WHERE value < 0;
DELETE FROM air_quality_measurements
WHERE value IS NULL;
```

AIR_QUALITY_MEASUREMENTS

Columns | Data | Constraints | Grants | Statistics | Triggers | Dependencies | Details

	MEASUREMENT_ID	STATION_ID	POLLUTANT_CODE	MEASUREMENT_TS	VALUE	UNIT
1	1	101	NO2	2025-03-01T08:00:(	45	ug/m3
2	2	102	PM25	2025-03-01T08:00:(	28	ug/m3
3	3	101	NO2	2025-03-01T09:00:(	78	ug/m3
4	4	102	NO2	2025-03-01T08:00:(	210	ug/m3
5	5	103	PM25	2025-03-01T08:00:(	18	ug/m3
6	6	104	PM10	2025-03-01T08:00:(	62	ug/m3
7	7	104	O3	2025-03-01T13:00:(	130	ug/m3

Fig.12 Result after removing incorrect measurement values

3.2 Write similar command for removing stations with NULL instead of a region.

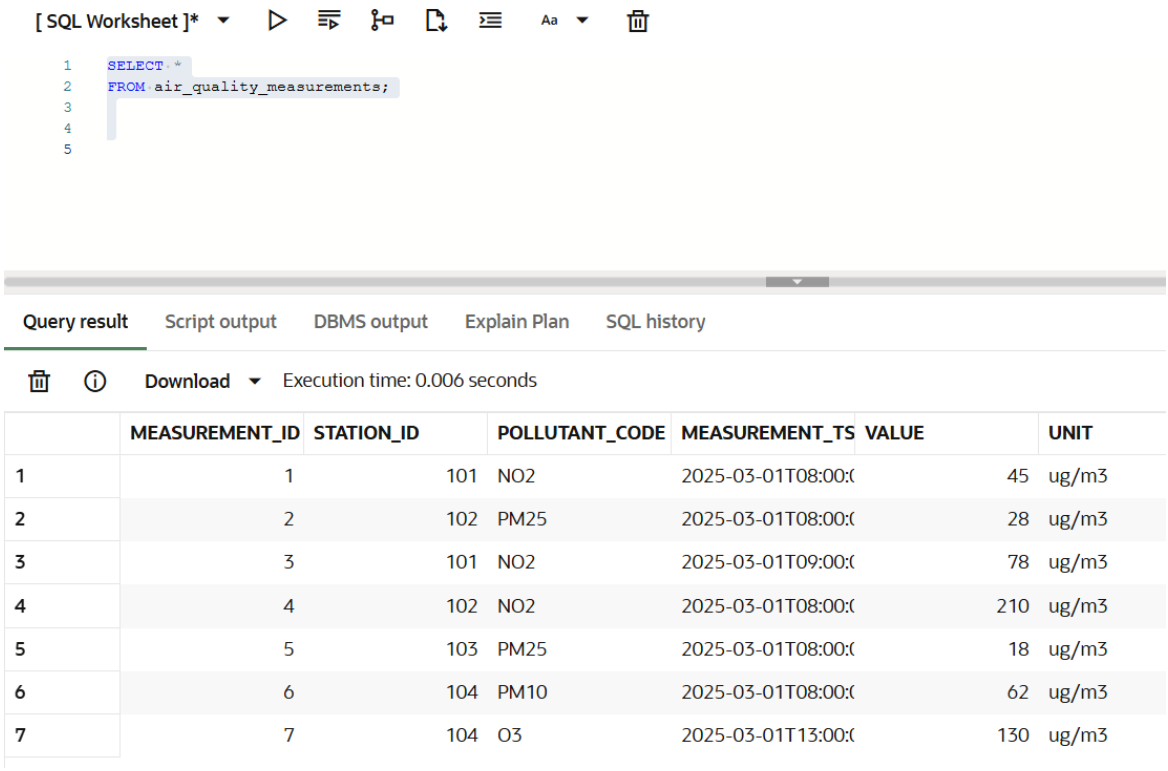
4. SELECT queries

We have learned about DML commands, so now let's try the SELECT query command.

4.1 Simple SELECT:

```
SELECT *

FROM air_quality_measurements;
```



	MEASUREMENT_ID	STATION_ID	POLLUTANT_CODE	MEASUREMENT_TS	VALUE	UNIT
1	1	101	NO2	2025-03-01T08:00:(	45	ug/m3
2	2	102	PM25	2025-03-01T08:00:(	28	ug/m3
3	3	101	NO2	2025-03-01T09:00:(	78	ug/m3
4	4	102	NO2	2025-03-01T08:00:(	210	ug/m3
5	5	103	PM25	2025-03-01T08:00:(	18	ug/m3
6	6	104	PM10	2025-03-01T08:00:(	62	ug/m3
7	7	104	O3	2025-03-01T13:00:(	130	ug/m3

Fig.13 Query execution result

4.2 Now check in practice how the result will change when executing the following query with sorting using ORDER BY:

```
SELECT station_id, pollutant_code, value
FROM air_quality_measurements
ORDER BY value DESC;
```

4.3 Now let's use WHERE filtering with logical operators and we will obtain data only for the specified pollutant:

```
SELECT station_id, pollutant_code, value
FROM air_quality_measurements
WHERE value > 50
  AND pollutant_code = 'NO2';
```

Query result | Script output | DBMS output | Explain Plan | SQL history

Download | Execution time: 0.007 seconds

	STATION_ID	POLLUTANT_CODE	VALUE
1	101	NO2	78
2	102	NO2	210

Fig.14 Query execution result

4.4 Let's try applying aggregation functions with grouping. How will the output change?

```
SELECT station_id, pollutant_code, AVG(value) AS avg_value
FROM air_quality_measurements
WHERE value IS NOT NULL
GROUP BY station_id, pollutant_code;
```

4.5 For filtering pollutants with average values higher than 40 let's try aggregated function with HAVING:

```
SELECT station_id, pollutant_code, AVG(value) AS avg_value
FROM air_quality_measurements
WHERE value IS NOT NULL
GROUP BY station_id, pollutant_code
HAVING AVG(value) > 40;
```

Query result | Script output | DBMS output | Explain Plan | SQL history

Download | Execution time: 0.073 seconds

	STATION_ID	POLLUTANT_CODE	AVG_VALUE
1	101	NO2	61.5
2	102	NO2	210
3	104	PM10	62
4	104	O3	130

Fig.15 Query execution result

5. JOIN operations

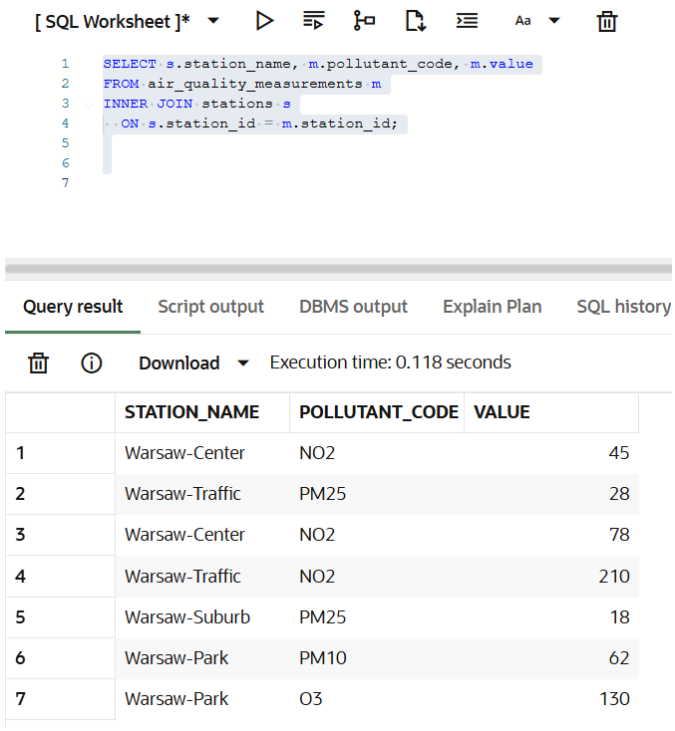
In a relational database, information is rarely stored in a single table. Instead, data is distributed across multiple tables, each representing a specific real-world concept. To analyze such data effectively, it is often necessary to combine rows from different tables into a single result set. This is achieved using the **JOIN** operation in SQL.

A JOIN defines how rows from two tables are related based on a logical condition, usually involving a primary key in one table and a foreign key in another. In the training database used in this course, air quality measurements are linked to monitoring stations through the station_id attribute. This relationship allows queries to retrieve both measurement values and descriptive station information at the same time.

5.1 An **INNER JOIN** returns only those rows for which **matching values exist in both tables**. Rows without a corresponding match in either table are excluded from the result.

In the context of the air quality database, an INNER JOIN between the measurement table and the station table returns only measurements that are associated with a known monitoring station. This is the most commonly used type of join and is suitable when incomplete or unrelated data should be ignored.

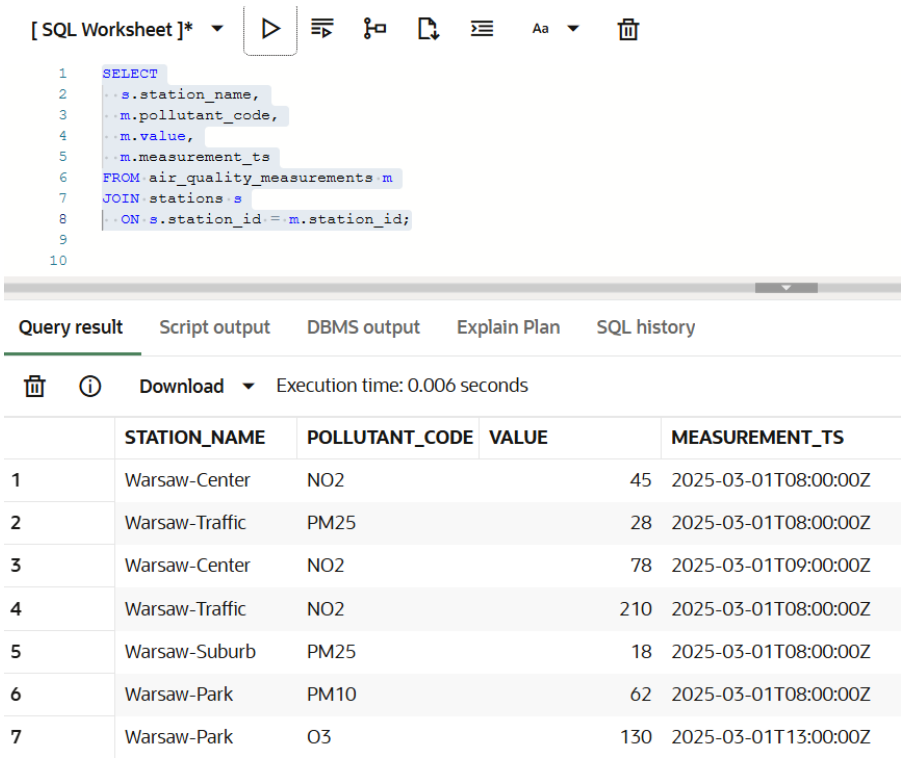
Conceptually, an INNER JOIN answers the question: “Which records exist in both tables at the same time?”



	STATION_NAME	POLLUTANT_CODE	VALUE
1	Warsaw-Center	NO2	45
2	Warsaw-Traffic	PM25	28
3	Warsaw-Center	NO2	78
4	Warsaw-Traffic	NO2	210
5	Warsaw-Suburb	PM25	18
6	Warsaw-Park	PM10	62
7	Warsaw-Park	O3	130

Fig.16 INNER JOIN query result

Also, if you see only word JOIN in the query, it also refers to INNER JOIN (fig. 17).



	STATION_NAME	POLLUTANT_CODE	VALUE	MEASUREMENT_TS
1	Warsaw-Center	NO2	45	2025-03-01T08:00:00Z
2	Warsaw-Traffic	PM25	28	2025-03-01T08:00:00Z
3	Warsaw-Center	NO2	78	2025-03-01T09:00:00Z
4	Warsaw-Traffic	NO2	210	2025-03-01T08:00:00Z
5	Warsaw-Suburb	PM25	18	2025-03-01T08:00:00Z
6	Warsaw-Park	PM10	62	2025-03-01T08:00:00Z
7	Warsaw-Park	O3	130	2025-03-01T13:00:00Z

Fig.17 INNER JOIN query result

5.2 A **LEFT JOIN** returns all rows from the left table, even if there is no matching row in the right table. When no match exists, the columns from the right table contain NULL values.

This type of join is particularly useful for identifying missing or incomplete data. For example, when joining monitoring stations with air quality measurements, a LEFT JOIN allows the query to list all stations, including those that do not yet have any recorded measurements.

A LEFT JOIN answers the question: “Which records exist in the left table, and what related data is available (if any) in the right table?”

[SQL Worksheet]*

```
SELECT s.station_name, m.pollutant_code, m.value
FROM stations s
LEFT JOIN air_quality_measurements m
  ON s.station_id = m.station_id
ORDER BY s.station_name;
```

Query result | Script output | DBMS output | Explain Plan | SQL history

Download | Execution time: 0.073 seconds

	STATION_NAME	POLLUTANT_CODE	VALUE
1	Warsaw-Center	NO2	45
2	Warsaw-Center	NO2	78
3	Warsaw-NewStation	*(null)*	*(null)*
4	Warsaw-Park	O3	130
5	Warsaw-Park	PM10	62
6	Warsaw-Suburb	PM25	18
7	Warsaw-Traffic	PM25	28
8	Warsaw-Traffic	NO2	210

Fig.18 LEFT JOIN query result

As you can see, the Warsaw-NewStation station has no connected measurements, but is still displayed in the results.

5.3 A **RIGHT JOIN** works in the opposite direction. It returns all rows from the right table, even if there is no matching row in the left table. If no match is found, the columns from the left table are filled with NULL values.

In the training database, a RIGHT JOIN can be used to list all defined pollutants, including those for which no measurements have been recorded yet. This is useful when verifying reference data completeness or identifying unused categories.

A RIGHT JOIN answers the question: “Which records exist in the right table, and are they currently referenced in the left table?”

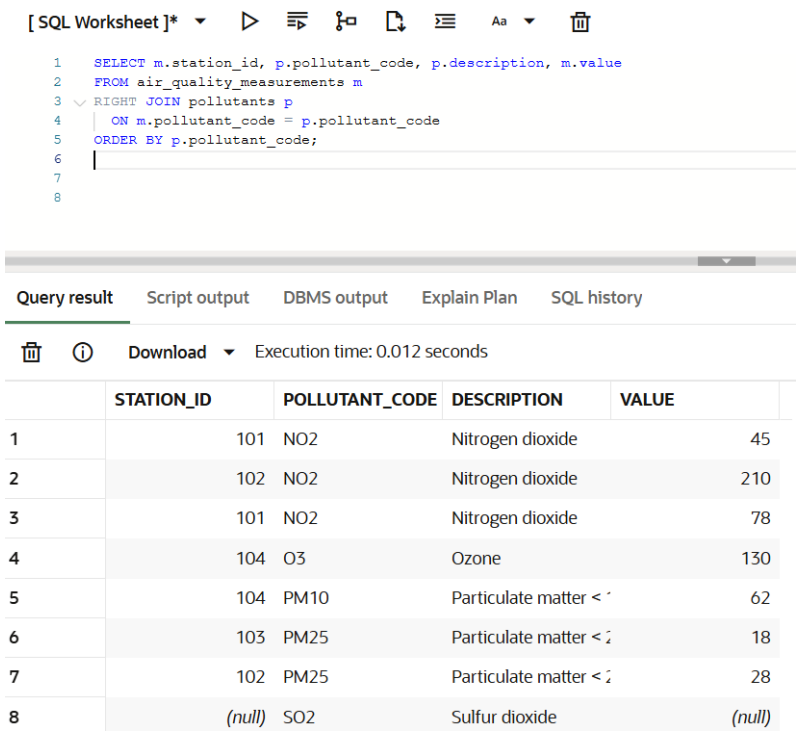


Fig.19 RIGHT JOIN query result

As you can see, the RIGHT JOIN results in the last row show the SO_2 measurement result, which is not linked to any station.

6. Subqueries in SQL

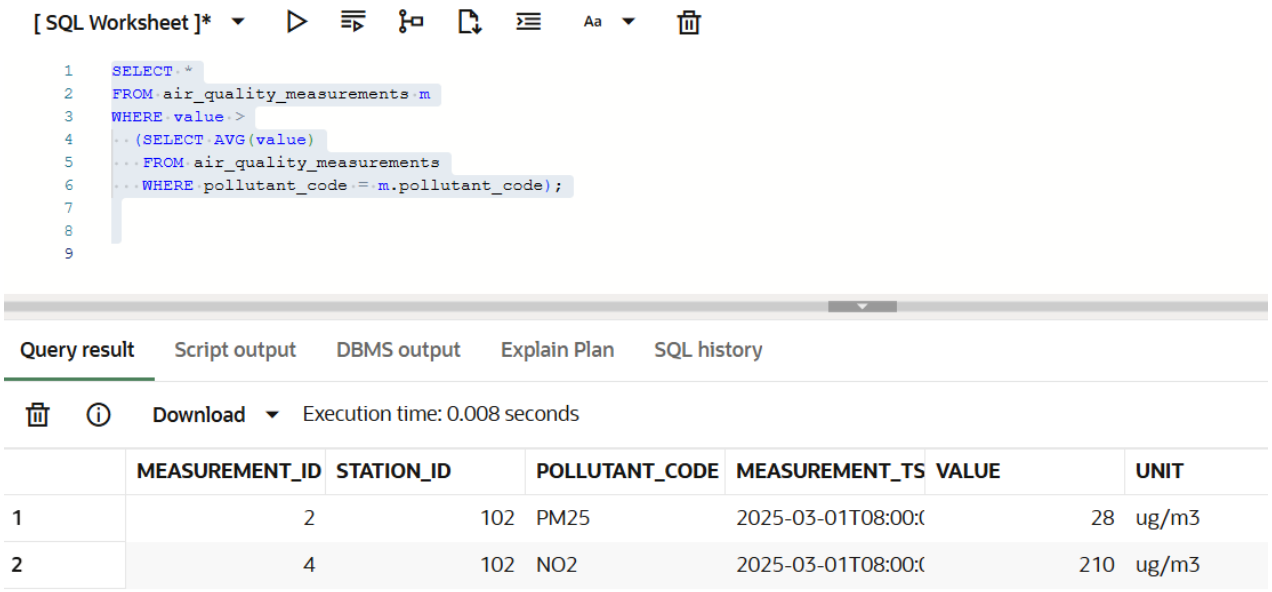
A **subquery** is a query that is **nested inside another SQL statement**. It is used when the result of one query is required to determine the outcome of another. Subqueries allow SQL statements to express complex logic in a clear and structured way.

Typically, a subquery is executed first, and its result is then used by the outer query. Subqueries are especially useful for analytical tasks involving calculated values such as averages, maximums, minimums, or dynamically defined thresholds.

Subqueries can return a **single value**, a **single row**, or a **set of rows**, depending on their purpose. A common use case is comparing individual records with aggregated values derived from the same table.

Subqueries may be **non-correlated**, meaning they run independently of the outer query, or **correlated**, meaning they depend on values from the outer query and are evaluated repeatedly for each row. While correlated subqueries are more advanced, both types play an important role in relational data analysis.

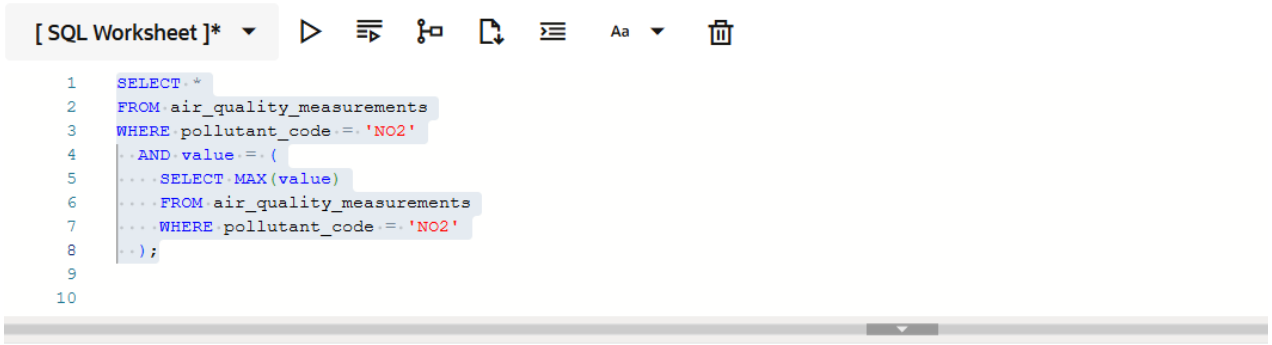
6.1 Let's create a subquery that will display measurements that are higher than the average value for a specific pollutant (fig. 20).



	MEASUREMENT_ID	STATION_ID	POLLUTANT_CODE	MEASUREMENT_TS	VALUE	UNIT
1	2	102	PM25	2025-03-01T08:00:(	28	ug/m3
2	4	102	NO2	2025-03-01T08:00:(	210	ug/m3

Fig.20 Subquery execution result

6.2 A subquery can also help identify stations with maximum NO_2 values (fig. 21).



	MEASUREMENT_ID	STATION_ID	POLLUTANT_CODE	MEASUREMENT_TS	VALUE	UNIT
1	4	102	NO2	2025-03-01T08:00:(	210	ug/m3

Fig.21 Subquery execution result

7. Transaction Control Language (TCL)

Transaction Control Language (TCL) ensures that groups of SQL operations are executed reliably. Transactions in Oracle are automatically started, but they must be explicitly finalized or cancelled using **COMMIT** or **ROLLBACK**. The **SAVEPOINT** command allows partial rollback within a transaction.

7.1 Insert a new air quality measurement and permanently save the change.

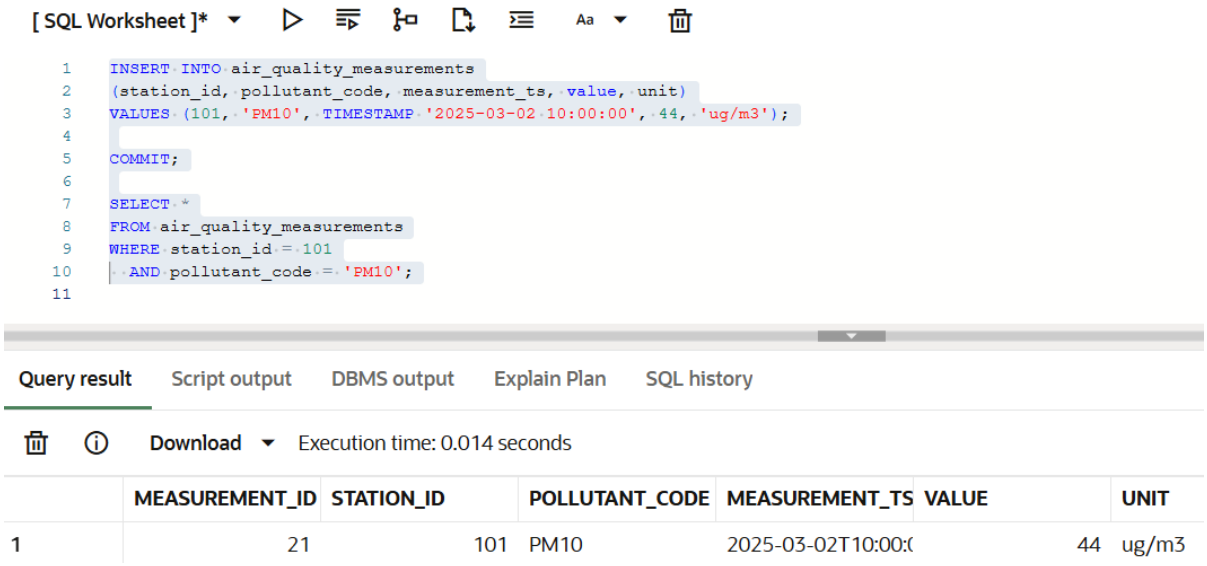


Fig.22 Committed transaction verification

7.2 Insert a measurement and then cancel the operation using ROLLBACK.

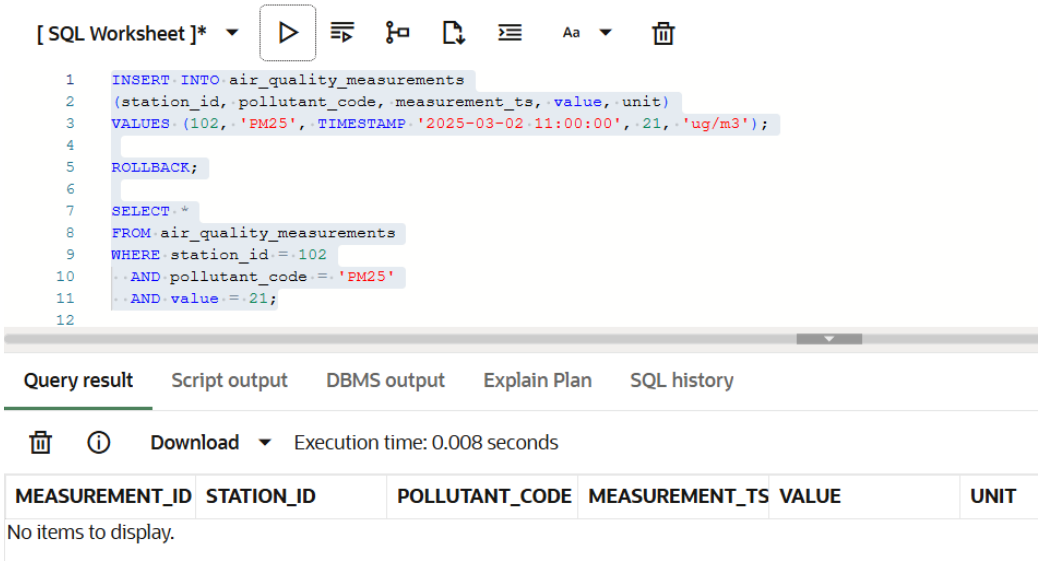


Fig.23 Transaction rollback verification

7.3 Perform multiple updates and roll back only part of the transaction.

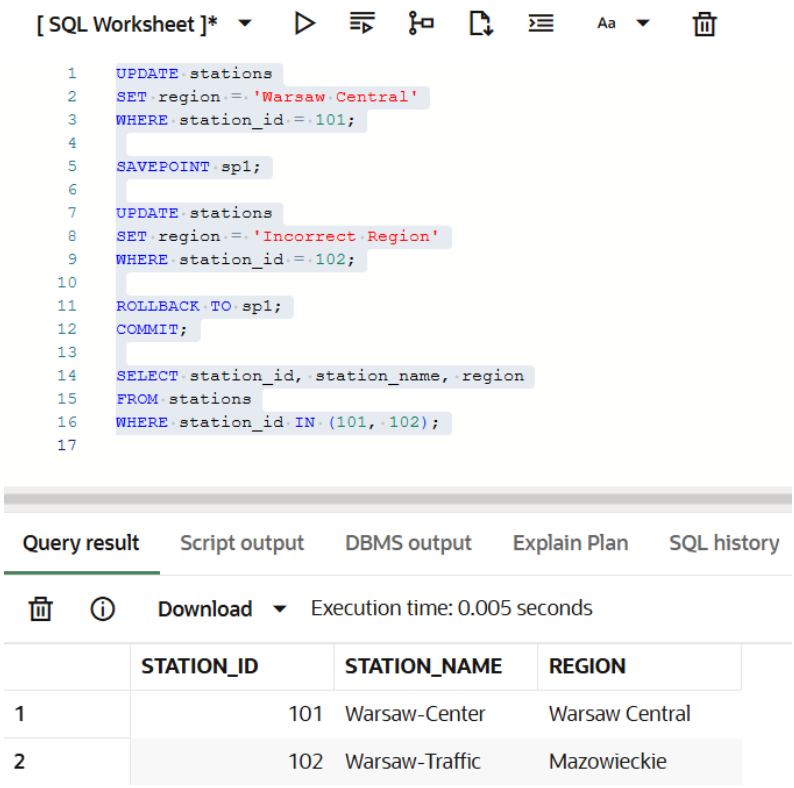


Fig.24 Partial transaction rollback verification

Analytical SQL Tasks:

1. Display all air quality measurements, including station identifier, pollutant code, measured value, and timestamp. Sort the results by measurement time.
2. List all measurements where the recorded value is greater than 50. Display the station identifier, pollutant, value, and time.
3. Retrieve all measurements of NO_2, sorted from the highest value to the lowest.
4. For each pollutant, find the station where the maximum measured value was recorded. Display the station name, pollutant, and measured value.
5. Display only those station-pollutant combinations where the average value exceeds 40.
6. For each pollutant, find the maximum recorded measurement value.

Practical Exercise(s)

Exercise 1 - Create new Database Schema

To analyze the India Urban Noise Monitoring dataset (available at https://www.kaggle.com/datasets/rohanrao/noise-monitoring-data-in-india), students must implement the database schema using DDL statements. Create two relational tables: STATIONS (Station, Name, City, State, Type) and STATION_MONTH (Station, Year, Month, Day, Night, DayLimit, NightLimit). Define Station as the primary key of the STATIONS table. Then, define a foreign key in STATION_MONTH so that STATION_MONTH.Station references STATIONS(Station), ensuring referential integrity between measurement records and station metadata.

Exercise 2 - Insert Sample Data

Because Oracle FreeSQL does not support direct CSV import and the primary data-loading approach in this environment is the INSERT statement, the dataset has been converted into SQL scripts: *stations_insert.sql* and *station_month_insert.sql*. Execute these scripts to populate the corresponding tables. Insert station metadata first (STATIONS), and then load the monthly measurements (STATION_MONTH).

Exercise 3 - Basic Queries

Write basic SELECT queries to explore the data. First, list all stations located in a selected city (for example, Delhi). Next, display monthly noise records for a chosen station and a selected year. These queries should help students understand the structure and meaning of both tables before moving to analytical tasks.

Exercise 4 - Filtering with Logical Operators

Retrieve and compare two categories of exceedance cases. First, return all records where both limits are exceeded, meaning Day > DayLimit AND Night > NightLimit. Then, return all records where at least one limit is exceeded, meaning Day > DayLimit OR Night > NightLimit. Interpret how the choice of logical operator changes the meaning and size of the result set.

Exercise 5 - Aggregations

Compute a station-level summary by calculating the average daytime noise level and average nighttime noise level for each station. Group the results by Station. This exercise reinforces the use of aggregate functions and GROUP BY for analytical reporting.

Exercise 5 - JOIN Queries

Produce a consolidated report that combines station metadata with monthly measurement data. Using an INNER JOIN between STATIONS and STATION_MONTH, output a result set that includes station name, city, station type, year, month, daytime noise, and nighttime noise. This exercise demonstrates how relational links enable meaningful reports that combine descriptive and measured data.

Assessment

Quiz (5 questions): short check of concepts before practical work.

1. Which statement best describes a LEFT JOIN?

   A. Returns only rows that match in both tables

   B. Returns all rows from the left table and matching rows from the right table

   C. Returns all rows from the right table and matching rows from the left table

   D. Combines every row from both tables without conditions

2. What type of anomaly occurs when deleting the last measurement for a station causes the station's metadata to be lost?

   A. Update anomaly

   B. Insertion anomaly

   C. Deletion anomaly

   D. Join anomaly

3. Which SQL command is part of the Data Definition Language (DDL)?

   A. SELECT

   B. INSERT

   C. DROP

   D. UPDATE

4. Which of the following is a valid use case for a subquery in SQL?

   A. Calculating the average pollutant value inside a WHERE clause

   B. Dropping a database

   C. Granting permissions

   D. Creating new tables

5. Which clause is used to combine rows from two tables based on a matching condition?

A. GROUP BY

B. ORDER BY

C. JOIN … ON

D. HAVING

Evaluation criteria: accuracy of SQL queries, completeness of results, clarity of interpretation, and quiz score.

Component	Weight	Criteria
Quiz	20%	Understanding of concepts
SQL Tasks	50%	Correctness, efficiency, coverage
Report	30%	Interpretation, clarity, visualization

References / Further Reading

Oracle Academy Cloud resources

EU Air Quality Standards documentation

Open-Meteo API reference

Selected open-access books on data warehousing

Selected open-access research papers on data warehousing in environmental science

C. J. Date. Introduction to Database Systems. Addison-Wesley Longman, Inc., 2004. 983 p.

H. Garcia-Molina, J. Ullman, J. Widom. Database Systems: The Complete Book. Pearson, 2008. 1248 p.

P. Rob, C. Coronel. Database Systems: Design, Implementation, and Management. Course Technology, 2011. 728 p.