Accessing Databases from Applications

Choosing the appropriate DBMS

Understanding Requirements:

Selecting a suitable database system, such as MySQL, PostgreSQL, MongoDB, or SQLite, depends on the specific requirements of the application. Factors like scalability, data structure, and performance play a crucial role in this decision-making process.

Comparing DBMS Options:

Each DBMS has its strengths and weaknesses. It's essential to compare the features, licensing, and community support of different systems to make an informed choice.

Scalability and Flexibility:

The chosen DBMS should align with the scalability needs of the application and provide the flexibility to adapt to evolving data requirements.

Installation and Configuration

Setting Up User Accounts:

Installing and configuring the DBMS involves creating user accounts with appropriate permissions. This step ensures that the database is secure and accessible only to authorized users.

Configuring Server Settings:

Configuring server settings, such as memory allocation, connection limits, and storage configurations, is essential for optimizing the performance of the DBMS.

Testing the Installation:

After installation, thorough testing of the DBMS setup is necessary to ensure that it functions as expected and meets the application's requirements.

Establishing Database Connection

Database Drivers and Libraries:

Exploring the available database drivers and libraries in different programming languages, such as SQLAlchemy in Python or JDBC in Java, that facilitate establishing connections to the database.

Connection String Configuration:

Understanding the process of configuring connection strings, including details such as the database host, port, username, password, and database name, to establish a successful connection.

Handling Connection Errors:

Addressing common connection errors and troubleshooting tips to ensure a seamless connection to the database from the application.

Performing CRUD Operations

Create (Insert):

Exploring the process of inserting new records into the database, including considerations for data validation and maintaining data integrity.

Read (Retrieve):

Understanding the retrieval of data from the database based on specific criteria, including querying techniques and best practices for efficient data retrieval.

Update and Delete Operations:

Explaining the methods for modifying existing records and removing records from the database, emphasizing the importance of error handling and transactional integrity.

Server-Side vs. Client-Side Database Access

Server-Side Scripting:

Accessing databases from web applications is primarily done through server-side scripting languages like PHP, Ruby, or Python. These languages facilitate communication between the web application and the database, enabling data retrieval and manipulation.

Client-Side Limitations:

While client-side scripting can interact with databases indirectly through server requests, direct database access from the client-side is restricted due to security and performance considerations.

Data Integrity and Security:

Server-side scripting ensures that database access is controlled and secure, preventing unauthorized access and maintaining data integrity.

Database Access Best Practices

Connection Pooling:

Efficient database access involves connection pooling, which allows reusing established database connections, reducing the overhead of connection creation and disposal.

Parameterized Queries:

Utilizing parameterized queries helps prevent SQL injection attacks and enhances the security of database interactions within web applications.

Error Handling:

Implementing robust error handling mechanisms is crucial for gracefully managing database access errors, ensuring the stability and reliability of the web application.

Database Connection Pooling

Database connection pooling is a technique to manage a pool of database connections that can be reused by the web application.

Benefits:

- It improves application performance by reducing the overhead of creating new database connections for each user request.

Considerations:

- Connection pooling requires careful management to avoid resource leaks and bottlenecks.

Data Access Methods

JDBC API

Java Database Connectivity (JDBC) API provides a standard interface for accessing databases from Java applications.

ORM Frameworks

Object-Relational Mapping (ORM) frameworks like Hibernate and Sequelize simplify database interactions by mapping database tables to application objects.

RESTful APIs

Web applications use RESTful APIs to perform CRUD (Create, Read, Update, Delete) operations on the database.

Real-life Applications of Databases

eCommerce Platforms

Data Management: eCommerce platforms use databases to store product information, customer details, and transaction records.

Scalability: Databases enable eCommerce platforms to handle a large number of products and users while maintaining performance.

Example: Amazon's product catalog and order management system.

Healthcare Systems

Patient Records:

Healthcare systems utilize databases to store and manage patient records, medical history, and treatment plans.

Data Security:

Databases in healthcare applications adhere to strict security standards to protect sensitive patient information.

Example:

Electronic Health Record (EHR) systems used by hospitals and clinics.

Online Banking Systems

Transaction Processing:

Online banking systems use databases to manage financial transactions, account balances, and customer information.

Data Consistency:

Databases ensure the consistency and accuracy of financial data across banking applications.

Example:

Database systems supporting online banking portals and mobile banking apps.

Data Access Methods

JDBC API for Database Access

Java Database Connectivity (JDBC): JDBC provides a standard Java API for accessing relational databases, enabling seamless integration with Java applications.

DataSource Objects: Applications access databases through DataSource objects, which provide properties to identify and describe the data source.

Example: Using JDBC to connect a Java web application to a MySQL database.

ORM Frameworks for Simplified Database Interactions

Object-Relational Mapping (ORM):

ORM frameworks like Hibernate and Sequelize map database tables to application objects, simplifying database interactions.

Entity Relationships:

ORM frameworks handle complex entity relationships and database queries using object-oriented paradigms.

Example:

Using Hibernate to perform CRUD operations on database entities in a web application.

SWOT Analysis of Database Integration

Strengths:

Database integration provides data consistency, scalability, and security for web applications.

Weaknesses:

Over-reliance on databases can lead to performance bottlenecks and complex data migration challenges.

Opportunities:

Emerging database technologies offer improved performance, real-time analytics, and cloud-based database solutions.

Threats:

Security vulnerabilities, data breaches, and compliance issues pose significant threats to database-integrated web applications.

Embedded SQL

Understanding Embedded SQL

Purpose:

Embedded SQL is a powerful method that facilitates the integration of high-level programming languages with database management systems (DBMS). It allows the seamless execution of SQL statements within application code, enabling efficient data manipulation and retrieval.

Significance:

Embedded SQL plays a crucial role in modern software development and database management, making it an essential concept for students to grasp as they prepare for careers in technology and programming.

Benefits of Embedded SQL

Seamless Database Interaction:

Embedded SQL enables developers to embed SQL statements directly within their application code, streamlining database interactions and enhancing data processing capabilities.

Enhanced Performance and Security:

By leveraging the power of SQL within programming languages, embedded SQL ensures optimized database access and robust security measures, contributing to efficient and secure data management practices.

Embedded SQL

- Embedded SQL is typically used in conjunction with programming languages like C/C++, Java, Python, and others. SQL statements are inserted directly into the source code of these languages.
- A language to which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- EXEC SQL statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement > END-EXEC

  Note: this varies by language. E.g. the Java embedding uses # SQL { .... } ;

Example

Specify the query in SQL and declare a *cursor* for it

EXEC SQL

declare *c* **cursor for**
select *customer-name, customer-city*

from *depositor, customer, account*

where *depositor.customer-name = customer.customer-name*

and *depositor account-number = account.account-number*

and *account.balance > :amount*

END-EXEC

Embedded SQL

- The **open** statement causes the query to be evaluated
- EXEC SQL **open** *c* END-EXEC
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL **fetch** *c* **into** :*cn, :cc* END-EXEC

Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* END-EXEC

Note: above details vary with language. E.g. the Java embedding defines Java iterators to step through result tuples.

Benefits of Embedded SQL

Seamless Integration:

Embedded SQL allows developers to seamlessly incorporate database operations into their application code, eliminating the need for separate SQL scripts or procedures.

Improved Performance:

By embedding SQL directly into the application code, developers can optimize database access and minimize overhead, resulting in improved performance compared to standalone SQL scripts.

Enhanced Security:

Parameterized queries and other security features provided by embedded SQL help mitigate the risk of SQL injection attacks, making applications more secure.

Maintainability:

With SQL statements embedded within the application code, developers can maintain and update database-related logic more easily, as all code is contained within a single codebase.

Variable Binding:

Embedded SQL allows for variables from the programming language to be bound to SQL queries. This enables dynamic query generation based on runtime values, enhancing flexibility and adaptability.

Error Handling:

Embedded SQL provides mechanisms for error detection and handling within the application code. Developers can implement error checking routines to manage exceptions and ensure the robustness of their applications.

Declaring Variables and Exception

Declaring Variables

Data Types:

Embedded SQL supports various data types similar to the host programming language. These include integers, floats, strings, and more complex types like structures or objects, depending on the capabilities of the host language.

Host Variables:

These are special types of variables used to interface between the host program and the SQL engine. When you declare a host variable, you're essentially telling the SQL engine where to store or retrieve data during SQL operations.

Size Consideration:

It's essential to consider the size of host variables, especially when fetching data from the database. Ensure that the size of the host variable matches the size of the data being retrieved to avoid truncation or buffer overflow issues.

Scope:

Host variables are typically declared in the host program's scope where they are needed. They can be local to a function or global, depending on the requirements of your application.

Declaring Variables-Example

```
// Declare variables
    int emp_id = 1001;
    String emp_name = null;
    float emp_salary = 0.0f;
// JDBC Connection variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
```

Note:

- *emp_id, emp_name, and emp_salary are variables declared in Java.*

Connection, Statement, and ResultSet are JDBC objects used for database interaction

Exception Handling

SQLCODE and SQLSTATE:

In addition to SQLCODE, embedded SQL also provides a mechanism to retrieve more detailed error information through SQLSTATE. SQLSTATE is a five-character string that provides standardized error information, while SQLCODE is a numeric value specific to the SQL implementation.

Error Classes:

SQL errors can be classified into different categories, such as syntax errors, constraint violations, connection errors, etc. Understanding these error classes can help you determine the appropriate action to take when handling exceptions.

Exception Propagation:

Depending on the host language and the framework you're using, exceptions raised by SQL operations may propagate differently. Some environments automatically handle SQL exceptions, while others require explicit handling in your code.

Logging and Debugging:

Proper exception handling includes logging error information for troubleshooting and debugging purposes. You can log SQL error codes, messages, and context information to aid in diagnosing issues with your application.

Transaction Management:

- Exception handling is closely tied to transaction management in database applications.
- Rollback transactions in the event of an error to maintain data integrity and consistency.

Error Handling:

SQLCODE can be checked after executing an SQL statement to determine whether it executed successfully or encountered an error. If SQLCODE is non-zero, it indicates an error, and you can handle it accordingly in your host program.

Handling Errors:

Depending on the error, you might take different actions. For example, if a SELECT statement fails to find a record, you might handle it by displaying a message to the user or taking some alternative action.

Exception Handling - Example

```
EXEC SQL SELECT emp_name INTO :emp_name FROM employees WHERE emp_id =
:emp_id;

if (SQLCODE != 0) {

   printf("Error retrieving employee data\n");

   // Handle error

}
```

Note:

By properly declaring variables and handling exceptions, you can ensure the reliability and robustness of your embedded SQL code, making it capable of handling various scenarios that may arise during database interactions.

Embedding SQL statements

- Embedding SQL statements refers to incorporating SQL (Structured Query Language) queries directly within another programming language or application.
- This practice is common in scenarios where dynamic data retrieval or manipulation is required within the context of a larger application.
- This integration allows developers to seamlessly interact with a database management system (DBMS) from within their codebase, enabling the retrieval, manipulation, and management of data stored in a relational database.

Working and Reason for embedding SQL statements:

- Integration with Programming Languages
- Dynamic Query Generation
- Parameterization
- Data Retrieval and Manipulation
- Transaction Management

- Error Handling
- Performance Optimization

Embedding SQL statements

Integration with Programming Languages:

- SQL statements can be embedded within programming languages like Python, Java, C#, PHP, etc.
- This integration allows developers to leverage the power of SQL for database operations while staying within the syntax and structure of the host language.

Dynamic Query Generation:

- Embedding SQL enables the dynamic generation of queries based on variables or conditions within the host application.
- This flexibility is crucial for scenarios where the structure or content of the query needs to change based on user input, system state, or other runtime factors.

Parameterization:

- Many programming languages provide mechanisms for parameterizing SQL queries embedded within them.
- Parameterization involves substituting placeholders (parameters) in the SQL query with actual values during execution.
- This helps prevent SQL injection attacks and improves query performance by allowing the database to cache query execution plans.

Data Retrieval and Manipulation:

- Embedded SQL is commonly used for tasks such as retrieving data from a database (SELECT statements), inserting, updating, or deleting records (INSERT, UPDATE, DELETE statements), and executing database-specific operations (e.g., creating tables, indexes, etc.).

Transaction Management:

- Embedded SQL allows developers to manage transactions within the context of their application.
- Transactions ensure data integrity by grouping database operations into atomic units that either succeed entirely or fail entirely.

Error Handling:

- Applications can handle errors raised by SQL queries directly within the host programming language, allowing for more robust error detection and recovery mechanisms.

Performance Optimization:

- By embedding SQL directly within the application code, developers can optimize database access patterns and minimize unnecessary round-trips between the application and the database.

Drawbacks of embedding SQL statements

Code Maintainability:

- Mixing SQL with application code can sometimes lead to less maintainable code, especially in larger applications where SQL statements are scattered throughout the codebase.

Security Risks:

- Improperly constructed SQL queries can lead to security vulnerabilities such as SQL injection attacks.
- Parameterization and other security best practices should be followed to mitigate these risks.

Database Coupling:

- Embedding SQL statements tightly couples the application code with the database schema and query language.
- This can make it more challenging to switch to a different database system or make significant changes to the database structure.

Cursors

- Cursor is a private SQL workgroup area allocated temporarily
- The required amount of memory space will be allocated in cursor name
- A cursor holds the records written by select statement
- There are two types of cursors
    - Implicit Cursors
    - Explicit Cursors

Implicit Cursors

- Oracle create implicit cursor automatically whenever the DML statements (INSERT, UPDATE and DELETE) are executed.
- The implicit cursors are SQL cursors
- The SQL cursors have four attributes

Attribute	Description
SQL%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
SQL%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
SQL%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

SQL%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Example for Implicit Cursors (Consider EMP table)

```
DECLARE
    total_rows number(2);
  BEGIN
    UPDATE emp SET sal =  sal + 500 where comm is null ;
    IF sql%notfound THEN
      dbms_output.put_line('No Employee selected');
    ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' Employees selected ');
    END IF;
 END;
```

Output

10 Employees selected

PL/SQL procedure successfully completed.

Note : Sal updated in EMP table for 10 employees , those commission is null

Explicit Cursors

- ✓ Explicit cursors are user-defined cursors
- ✓ It should be defined in the declaration section of the PL/SQL Block.

Syntax

CURSOR cursor_name IS select_statement;

The following steps to be followed for explicit cursors

- ✓ Declare the cursor for initialize the memory
- ✓ Open the cursor for allocating memory
- ✓ Fetch the cursor values into local variables
- ✓ Close the cursor for release the memory

Example for Explicit Cursors

```
DECLARE
emp_no emp.empno%type;
emp_name emp.ename%type;
emp_sal emp.sal%type;
CURSOR emp_cur is SELECT empno,ename,sal FROM emp;
BEGIN
OPEN emp_cur;
dbms_output.put_line('emp_no' || ' ' || 'emp_name' || ' ' ||'emp_sal');
LOOP
FETCH emp_cur into emp_no,emp_name,emp_sal;
EXIT WHEN emp_cur%notfound;
dbms_output.put_line(emp_no  || ' ' || emp_name || ' ' || emp_sal);
END LOOP;
CLOSE emp_cur;
END;
```

Example for Explicit Cursors-Output

Example for Explicit Cursors-Output

Output

emp_no	emp_name	emp_sal
7369	SMITH	800
7499	ALLEN	1600
7521	WARD	1250
7566	JONES	2975
7654	MARTIN	1250
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7844	TURNER	1500

Cursor based Records

```
DECLARE
CURSOR emp_currec is SELECT empno, ename FROM emp;
emp_rec emp_currec%rowtype;
BEGIN
OPEN emp_currec;
DBMS_OUTPUT.put_line('Employee Number' || ' ' || 'Name');
LOOP
FETCH emp_currec into emp_rec;
 EXIT WHEN emp_currec%notfound;
```

```
DBMS_OUTPUT.put_line(emp_rec.empno || ' ' || emp_rec.ename);

END LOOP;

END;
```

```
Output
Employee Number Name
7369            SMITH
7499            ALLEN
7521            WARD
7566             JONES
7654            MARTIN
7698            BLAKE
7782            CLARK
7788             SCOTT
7839            KING
7844            TURNER
7876             ADAMS
7900            JAMES
7902            FORD
7934            MILLER
PL/SQL procedure successfully completed.
```

Usage of Cursor

Complex Data Manipulation:

- When you need to perform complex data manipulation or calculations on a row-by-row basis, cursors can be helpful.
- For example, you might need to calculate running totals, perform custom validations, or update values based on certain conditions.

Iterating Over Result Sets:

- If you need to iterate over a result set returned by a query and perform specific actions for each row, cursors can be used.
- This is useful when you want to process each row individually, perhaps for printing reports or generating complex output.

Procedural Logic:

- Cursors can be used within stored procedures or functions to implement procedural logic.
- You might use a cursor to loop through a set of records and perform different actions based on certain conditions.

Batch Processing:

- In some cases, you might need to process data in batches rather than all at once.
- Cursors can be used to fetch a subset of rows at a time and process them in smaller chunks, which can be more efficient and manageable for large datasets.

Properties of Cursors and Dynamic SQL

Traversal:

- Cursors allow sequential traversal of the result set returned by a SQL query, enabling access to each row individually.

Scoped:

- Cursors are scoped to the connection or session in which they are declared.
- This means that they are accessible only within the context of the connection or session in which they are defined.

Read-Only or Updatable:

- Cursors can be either read-only or updatable, depending on how they are defined.
- Read-only cursors allow fetching rows from the result set but do not allow modifications to the underlying data.
- Updatable cursors, on the other hand, allow modifications such as inserts, updates, and deletes to the rows in the result set.

Positioning:

- Cursors maintain a current position within the result set, which can be moved forward, backward, or to a specific position within the result set using appropriate cursor operations.

Scrollability:

- Cursors can be either forward-only or scrollable. Forward-only cursors allow traversal only in one direction, typically from the beginning to the end of the result set.
- Scrollable cursors, on the other hand, allow navigation in both forward and backward directions, as well as random access to any position within the result set.

Fetch Size:

- Cursors can fetch one or more rows at a time, depending on the fetch size specified during cursor declaration.
- Fetch size determines how many rows are retrieved from the result set in each fetch operation.

Resource Management:

- Cursors consume system resources such as memory and processing power.
- It's important to properly manage cursor resources by closing them when they are no longer needed to free up system resources and prevent potential memory leaks.

Transaction Scope:

- Cursors are typically associated with transactions, and their behavior may be influenced by the transaction properties such as isolation level and transaction boundaries.

Cursor Variables:

- Some DBMSs support cursor variables, which are variables that can hold references to cursors.
- Cursor variables provide a way to pass cursors as parameters to stored procedures or functions, enabling more flexible cursor usage.

Dynamic SQL

- Dynamic SQL refers to the generation and execution of SQL statements at runtime within a program or stored procedure.
- Instead of writing static SQL queries with fixed table and column names, dynamic SQL allows you to construct SQL statements dynamically based on varying conditions, user inputs, or other runtime factors.

Properties of Dynamic SQL:

Flexibility:

- Dynamic SQL offers flexibility by allowing you to generate SQL statements dynamically based on runtime conditions.
- This can be particularly useful when dealing with varying search criteria, dynamic column selections, or changing table structures.

String Manipulation:

- Dynamic SQL involves manipulating strings to construct SQL statements. You can concatenate strings, variables, and expressions to build SQL queries that meet specific requirements at runtime.

Parameterization:

- Dynamic SQL supports parameterization, enabling you to pass parameters to dynamically generated SQL statements.
- This helps prevent SQL injection attacks and promotes better performance by allowing the database to reuse query execution plans.

Dynamic Table and Column Names:

- With dynamic SQL, you can generate SQL statements with dynamic table and column names.
- This is useful when the table or column names are not known at compile time or when you need to construct queries based on user inputs.

Dynamic Conditions:

- Dynamic SQL allows you to dynamically generate WHERE clauses and other conditional statements based on runtime conditions.
- This flexibility enables the creation of dynamic search queries that adapt to user inputs or changing business requirements.

Dynamic SQL in Stored Procedures:

- Stored procedures can leverage dynamic SQL to dynamically generate and execute SQL statements based on input parameters or other runtime factors.
- This allows stored procedures to be more versatile and adaptable to different scenarios.

Dynamic SQL and Security:

- While dynamic SQL offers flexibility, it also introduces security considerations, particularly regarding SQL injection vulnerabilities.
- It's important to properly sanitize user inputs and validate dynamic SQL statements to mitigate the risk of SQL injection attacks.

Performance Implications:

- Dynamic SQL may have performance implications compared to static SQL due to the need for SQL statement parsing and compilation at runtime.
- However, parameterized dynamic SQL can help mitigate some performance overhead by allowing the database to cache and reuse execution plans.

Dynamic SQL

Allows programs to construct and submit SQL queries at run time.

Example of the use of dynamic SQL from within a C program.

```
char *  sqlprog = "update account
                   set balance = balance * 1.05
                   where account-number = ?"
EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.