Database Management System Practical
B. Sc. (Information Technology) Semester – III

By munotes.in

Updated 2023

Mumbai University

List of Practical
1. SQL Statements – 1
a. Writing Basic SQL SELECT Statements

Solutions

Writing basic SQL SELECT statements is a practical skill that is essential for querying and retrieving data from a database. It allows you to specify the data you want to retrieve and filter it based on specific conditions. Here's an example of a basic SQL SELECT statement:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Let's break down the components:

- `SELECT`: This keyword is used to specify the columns you want to retrieve in the result set.
- `column1, column2, ...`: These are the columns you want to select. You can specify multiple columns separated by commas or use the asterisk (*) to select all columns.
- `FROM`: This keyword is used to specify the table from which you want to retrieve the data.
- `table_name`: This is the name of the table containing the data you want to retrieve.
- `WHERE`: This keyword is used to specify the conditions that the retrieved data must meet.

- `condition`: This is the condition used to filter the data based on specific criteria. For example, `column_name = value` specifies that the value in the column must be equal to the specified value.

Here's an example that illustrates a basic SQL SELECT statement in action:

```sql
SELECT first_name, last_name, age
FROM employees
WHERE department = 'IT';
```

This statement selects the `first_name`, `last_name`, and `age` columns from the `employees` table, but only for employees who belong to the 'IT' department. The result will be a list of employees' first names, last names, and ages that meet the specified condition.

Mastering SQL SELECT statements is crucial for working with databases as it forms the foundation for more advanced queries and data manipulation operations.

b. Restricting and Sorting Data

Solutions

Restricting and sorting data are important aspects of SQL that allow you to refine and organize the results of your queries. Here's an overview of how to restrict and sort data using SQL statements:

1. Restricting Data with the WHERE Clause:

The WHERE clause is used to specify conditions that filter the data based on specific criteria. It allows you to retrieve only the records that meet the specified conditions. Here's an example:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

For instance, to retrieve all employees from the "employees" table who are in the 'IT' department and have a salary greater than $50,000, you can use the following query:

```sql
SELECT *
FROM employees
WHERE department = 'IT' AND salary > 50000;
```

2. Sorting Data with the ORDER BY Clause:
The ORDER BY clause is used to sort the result set in ascending or descending order based on one or more columns. By default, it sorts the data in ascending order. Here's an example:

```sql
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 ASC, column2 DESC;
```

For instance, to retrieve all employees from the "employees" table and sort them by their last names in ascending order and their salaries in descending order, you can use the following query:

```sql
SELECT *
FROM employees
ORDER BY last_name ASC, salary DESC;
```

This will return the employees' records sorted alphabetically by last name and, for employees with the same last name, sorted by their salary in descending order.

By combining the WHERE and ORDER BY clauses, you can further refine your queries and retrieve specific data in a sorted manner. These techniques are fundamental to SQL and are used extensively in database applications to extract and manipulate data effectively.

c. Single-Row Functions

Solutions

Single-row functions in SQL are used to operate on individual rows of data and return a single result for each row. These functions can be applied to columns or expressions within the SELECT statement. Here are some commonly used single-row functions:

1. String Functions:
   - `LOWER(str)`: Converts a string to lowercase.
   - `UPPER(str)`: Converts a string to uppercase.

- `SUBSTR(str, start_position, length)`: Extracts a substring from a string starting at a specified position with a specified length.
- `LENGTH(str)`: Returns the length of a string.
- `CONCAT(str1, str2)`: Concatenates two strings.

2. Numeric Functions:
- `ROUND(num, decimal_places)`: Rounds a number to the specified decimal places.
- `TRUNC(num, decimal_places)`: Truncates a number to the specified decimal places.
- `ABS(num)`: Returns the absolute value of a number.
- `MOD(num, divisor)`: Returns the remainder of the division of a number by another number.

3. Date Functions:
- `SYSDATE`: Returns the current system date and time.
- `MONTH(date)`: Returns the month of a specified date.
- `YEAR(date)`: Returns the year of a specified date.
- `TO_CHAR(date, format)`: Converts a date to a specified format as a string.

4. Conversion Functions:
- `TO_NUMBER(str)`: Converts a string to a number.
- `TO_CHAR(num)`: Converts a number to a string.
- `TO_DATE(str, format)`: Converts a string to a date.

Here's an example that demonstrates the usage of single-row functions:

```sql
SELECT first_name, UPPER(last_name), LENGTH(email), ROUND(salary, 2)
FROM employees;
```

```

In this example, we select the `first_name`, convert the `last_name` to uppercase, calculate the length of the `email` address, and round the `salary` to two decimal places for each row in the "employees" table.

Single-row functions enhance the capabilities of SQL by allowing you to transform, manipulate, and derive new values from existing data. They are widely used in SQL queries to perform calculations, manipulate strings, and work with dates and numbers.
```

2. SQL Statements – 2
a. Displaying Data from Multiple Tables

Solutions

Displaying data from multiple tables is a fundamental aspect of SQL that allows you to combine information from different tables into a single result set. This is typically achieved using joins, which establish relationships between tables based on common columns. Here are the common types of joins used to display data from multiple tables:

1. INNER JOIN:
   An inner join returns only the rows that have matching values in both tables being joined. It combines rows from two tables based on a specified condition. The syntax for an inner join is as follows:

   ```sql
   SELECT column1, column2, ...
   FROM table1
   INNER JOIN table2
   ON table1.column = table2.column;
   ```

   Here's an example that demonstrates an inner join:

   ```sql
   SELECT customers.customer_id, customers.customer_name, orders.order_date
   FROM customers
   INNER JOIN orders
   ON customers.customer_id = orders.customer_id;
   ```

This query selects the customer ID, customer name from the "customers" table, and the order date from the "orders" table. The join condition is based on the matching customer IDs in both tables.

2. LEFT JOIN:
A left join returns all the rows from the left table and the matching rows from the right table. If there is no match, it returns NULL values for the right table columns. The syntax for a left join is as follows:

```sql
SELECT column1, column2, ...
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Here's an example of a left join:

```sql
SELECT customers.customer_id, customers.customer_name, orders.order_date
FROM customers
LEFT JOIN orders
ON customers.customer_id = orders.customer_id;
```

This query selects the customer ID, customer name from the "customers" table, and the order date from the "orders" table. It returns all customers, including those who have not placed any orders, with NULL values for the order date.

3. RIGHT JOIN:
A right join returns all the rows from the right table and the matching rows from the left table. If there is no match, it returns NULL values for the left table columns. The syntax for a right join is as follows:

```sql
SELECT column1, column2, ...
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Here's an example of a right join:

```sql
SELECT customers.customer_id, customers.customer_name,
orders.order_date
FROM customers
RIGHT JOIN orders
ON customers.customer_id = orders.customer_id;
```

This query selects the customer ID, customer name from the "customers" table, and the order date from the "orders" table. It returns all orders, including those without matching customer records, with NULL values for the customer ID and customer name.

These join types enable you to combine data from multiple tables in SQL queries, providing a powerful way to retrieve and analyze related information.

b. Aggregating Data Using Group Functions

Solutions

Aggregating data using group functions is a crucial aspect of SQL that allows you to perform calculations on a set of rows and return a single result for each group. Group functions, also known as aggregate functions, operate on a group of rows rather than individual rows. Here are some commonly used group functions:

1. COUNT:
   The COUNT function returns the number of rows in a group, including NULL values. It can be used with the `*` wildcard to count all rows or with a specific column to count the non-null values in that column.

   Example:

   ```sql
   SELECT COUNT(*) AS total_records
   FROM employees;
   ```

   This query returns the total number of records in the "employees" table.

2. SUM:
   The SUM function calculates the sum of a numeric column within a group.

   Example:

   ```sql
   SELECT SUM(salary) AS total_salary
   FROM employees;
   ```

```

This query returns the total sum of the "salary" column in the "employees" table.

3. AVG:
The AVG function calculates the average value of a numeric column within a group.

Example:

```sql
SELECT AVG(salary) AS average_salary
FROM employees;
```

This query returns the average salary of all employees in the "employees" table.

4. MAX:
The MAX function returns the maximum value of a column within a group.

Example:

```sql
SELECT MAX(salary) AS max_salary
FROM employees;
```

This query returns the highest salary among all employees in the "employees" table.
```

5. MIN:
   The MIN function returns the minimum value of a column within a group.

   Example:

   ```sql
   SELECT MIN(salary) AS min_salary
   FROM employees;
   ```

   This query returns the lowest salary among all employees in the "employees" table.

Group functions are often used in conjunction with the GROUP BY clause, which groups the data based on one or more columns. This allows you to calculate aggregated results for each group individually.

Example:

```sql
SELECT department, AVG(salary) AS average_salary
FROM employees
GROUP BY department;
```

This query calculates the average salary for each department by grouping the data based on the "department" column.

Aggregating data using group functions is essential for summarizing and analyzing large datasets in SQL. It allows you to derive meaningful insights and make data-driven decisions based on calculated results.

c. Subqueries

Solutions

Subqueries, also known as nested queries or inner queries, are SQL queries that are embedded within another query. They allow you to use the results of one query as input for another query. Subqueries can be used in various parts of a SQL statement, such as the SELECT, FROM, WHERE, or HAVING clauses. Here's an overview of how subqueries work and their common uses:

1. Subqueries in the WHERE Clause:
   Subqueries can be used in the WHERE clause to filter data based on the results of a nested query. The subquery is enclosed within parentheses and is typically written within the condition of the WHERE clause. Here's an example:

   ```sql
   SELECT column1, column2, ...
   FROM table_name
   WHERE column1 IN (SELECT column
                     FROM another_table
                     WHERE condition);
   ```

   For instance, to retrieve all employees from the "employees" table who belong to departments with a budget greater than $100,000, you can use the following query:

   ```sql
   SELECT employee_name, department
   ```

```
FROM employees
WHERE department IN (SELECT department
                     FROM departments
                     WHERE budget > 100000);
```

This query uses a subquery to fetch the departments with a budget greater than $100,000, and then selects the employees from those departments.

2. Subqueries in the SELECT Clause:

Subqueries can also be used in the SELECT clause to retrieve values or perform calculations based on the results of a nested query. The subquery is typically written within parentheses and used as an expression in the SELECT clause. Here's an example:

```sql
SELECT column1, column2, (SELECT expression
                          FROM another_table
                          WHERE condition) AS calculated_value
FROM table_name;
```

For example, to retrieve the total number of orders for each customer from the "customers" table, you can use the following query:

```sql
SELECT customer_id, customer_name, (SELECT COUNT(*)
                                    FROM orders
                                    WHERE orders.customer_id =
customers.customer_id) AS total_orders
FROM customers;
```

```

This query uses a subquery within the SELECT clause to count the number of orders for each customer and returns the result as "total_orders" for each customer record.

Subqueries provide a powerful way to perform complex queries, make comparisons, and retrieve data based on specific conditions. They allow you to break down complex problems into smaller, more manageable parts and leverage the results of one query within another. Subqueries are widely used in SQL to solve various data retrieval and manipulation challenges.
```

3. Manipulating Data
a. Using INSERT statement

Solutions

The INSERT statement in SQL is used to insert new records into a table. It allows you to add data to a table by specifying the values for each column or by selecting values from another table. Here's an overview of how to use the INSERT statement:

1. Inserting Values into Specific Columns:
   To insert values into specific columns, you need to specify the column names in the INSERT statement. The values should be provided in the same order as the column names. Here's the syntax:

   ```sql
   INSERT INTO table_name (column1, column2, ...)
   VALUES (value1, value2, ...);
   ```

   For example, to insert a new employee into the "employees" table with specific values for the "first_name", "last_name", and "salary" columns, you can use the following query:

   ```sql
   INSERT INTO employees (first_name, last_name, salary)
   VALUES ('John', 'Doe', 50000);
   ```

   This query inserts a new row into the "employees" table with the specified values for the "first_name", "last_name", and "salary" columns.

2. Inserting Values from Another Table:

You can also insert values into a table by selecting them from another table. In this case, the column names in the INSERT statement should match the column names in the SELECT statement. Here's the syntax:

```sql
INSERT INTO table_name (column1, column2, ...)
SELECT column1, column2, ...
FROM another_table
WHERE condition;
```

For example, to insert all employees from the "new_employees" table into the "employees" table who have a salary greater than $50000, you can use the following query:

```sql
INSERT INTO employees (first_name, last_name, salary)
SELECT first_name, last_name, salary
FROM new_employees
WHERE salary > 50000;
```

This query selects the values for the "first_name", "last_name", and "salary" columns from the "new_employees" table and inserts them into the "employees" table for the employees who meet the specified condition.

The INSERT statement allows you to add new data to a table, whether by providing specific values or by selecting values from another table. It is a fundamental SQL statement used for data manipulation and plays a crucial role in populating databases with information.

b. Using DELETE statement

Solutions

The DELETE statement in SQL is used to delete existing records from a table. It allows you to remove one or more rows that match specific conditions. Here's an overview of how to use the DELETE statement:

1. Deleting All Rows from a Table:
   To delete all rows from a table, you can use the following syntax:

   ```sql
   DELETE FROM table_name;
   ```

   For example, to delete all records from the "employees" table, you can use the following query:

   ```sql
   DELETE FROM employees;
   ```

   This query will remove all rows from the "employees" table, effectively emptying the table.

2. Deleting Specific Rows:
   To delete specific rows from a table based on certain conditions, you can use the WHERE clause in the DELETE statement. This allows you to specify the criteria for selecting the rows to be deleted. Here's the syntax:

   ```sql
   DELETE FROM table_name
   ```

```
WHERE condition;
```

For example, to delete all employees from the "employees" table who have a salary less than $50000, you can use the following query:

```sql
DELETE FROM employees
WHERE salary < 50000;
```

This query will remove all rows from the "employees" table where the salary is less than $50000.

It's important to exercise caution when using the DELETE statement because it permanently removes data from a table. Ensure that you have appropriate backups or confirm that the records you're deleting are indeed no longer needed. Additionally, be careful when using the DELETE statement without a WHERE clause, as it can delete all rows in a table, resulting in data loss.

The DELETE statement is a powerful tool for removing data from a table, providing control over the data present in a database.

c. Using UPDATE statement

Solutions

The UPDATE statement in SQL is used to modify existing records in a table. It allows you to change the values of one or more columns in specific rows based on specified conditions. Here's an overview of how to use the UPDATE statement:

1. Updating Specific Rows:
To update specific rows in a table, you can use the WHERE clause in the UPDATE statement to specify the conditions that determine which rows to modify. Here's the syntax:

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

For example, to update the salary of an employee with the employee ID 1234 in the "employees" table, you can use the following query:

```sql
UPDATE employees
SET salary = 60000
WHERE employee_id = 1234;
```

This query will change the value of the "salary" column to 60000 for the employee with the employee ID 1234.

2. Updating All Rows:
To update all rows in a table, you can omit the WHERE clause in the UPDATE statement. However, be cautious when doing so, as it will update all records in the table. Here's the syntax:

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...;
```

```

For example, to update the department of all employees in the "employees" table to 'HR', you can use the following query:

```sql
UPDATE employees
SET department = 'HR';
```

This query will set the value of the "department" column to 'HR' for all employees in the table.

The UPDATE statement allows you to modify the values of existing records in a table, either for specific rows based on conditions or for all rows. Be cautious when using the UPDATE statement, as it directly modifies data in the table. It's a good practice to review and double-check the conditions and values before executing an UPDATE statement to avoid unintended changes.
```

4. Creating and Managing Tables
a. Creating and Managing Tables

Solutions

Creating and managing tables is an essential part of working with relational databases. In SQL, you can create tables to store structured data and define the structure of the table using columns and their data types. Here's an overview of how to create and manage tables in SQL:

1. Creating a Table:
   To create a table, you need to use the CREATE TABLE statement. This statement specifies the table name and defines the columns and their data types. Here's the syntax:

   ```sql
   CREATE TABLE table_name (
       column1 data_type,
       column2 data_type,
       ...
   );
   ```

   For example, to create a "customers" table with columns for customer ID, name, and email, you can use the following query:

   ```sql
   CREATE TABLE customers (
       customer_id INT,
       customer_name VARCHAR(50),
       email VARCHAR(100)
   );
   ```

```

This query creates a "customers" table with three columns: "customer_id" of type INT, "customer_name" of type VARCHAR with a maximum length of 50 characters, and "email" of type VARCHAR with a maximum length of 100 characters.

2. Modifying a Table:
Once a table is created, you may need to modify its structure or properties. SQL provides several statements to alter tables, such as adding or dropping columns, changing column data types, or adding constraints. Here are a few examples:

- Adding a Column:
  ```sql
  ALTER TABLE table_name
  ADD column_name data_type;
  ```

- Modifying a Column Data Type:
  ```sql
  ALTER TABLE table_name
  ALTER COLUMN column_name new_data_type;
  ```

- Dropping a Column:
  ```sql
  ALTER TABLE table_name
  DROP COLUMN column_name;
  ```

3. Managing Table Data:
  ```

To manage data within tables, you can use various SQL statements such as INSERT, UPDATE, and DELETE, as discussed in the previous sections. These statements allow you to add, modify, and remove records in a table.

4. Dropping a Table:
To delete a table and remove it from the database, you can use the DROP TABLE statement. Here's the syntax:

```sql
DROP TABLE table_name;
```

For example, to drop the "customers" table, you can use the following query:

```sql
DROP TABLE customers;
```

This query will permanently delete the "customers" table and all its data.

Creating and managing tables is a fundamental aspect of database management. It involves defining the structure of the table and manipulating the table's data as needed. SQL provides a range of statements to create, modify, and delete tables, allowing you to organize and store data efficiently.

b. Including Constraints

Solutions

When creating tables in SQL, you can include constraints to enforce rules and maintain data integrity. Constraints define specific conditions or rules that the data in a table must satisfy. Here are some commonly used constraints in SQL:

1. PRIMARY KEY Constraint:
   The PRIMARY KEY constraint uniquely identifies each record in a table. It ensures that the column or combination of columns specified as the primary key have unique values and cannot contain NULL values. Here's an example of defining a primary key constraint:

   ```sql
   CREATE TABLE table_name (
       column1 data_type,
       column2 data_type,
       ...
       PRIMARY KEY (column1)
   );
   ```

   This query creates a table with a primary key constraint on "column1".

2. FOREIGN KEY Constraint:
   The FOREIGN KEY constraint establishes a relationship between two tables based on a column or columns. It ensures referential integrity by enforcing that the values in the foreign key column(s) match values in the primary key column(s) of the referenced table. Here's an example:

   ```sql
   CREATE TABLE table_name1 (
       column1 data_type PRIMARY KEY,
       ...
   ```

```
);

CREATE TABLE table_name2 (
    column1 data_type,
    ...
    FOREIGN KEY (column1) REFERENCES table_name1(column1)
);
```

This query creates two tables, with "table_name2" referencing the primary key of "table_name1" using a foreign key constraint.

3. UNIQUE Constraint:
The UNIQUE constraint ensures that the values in a column or combination of columns are unique, meaning no duplicate values are allowed. Unlike the primary key constraint, a unique constraint can have NULL values. Here's an example:

```sql
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
    UNIQUE (column1)
);
```

This query creates a table with a unique constraint on "column1".

4. CHECK Constraint:

The CHECK constraint allows you to specify a condition that must be satisfied for the data in a column. It ensures that only values meeting the specified condition are allowed. Here's an example:

```sql
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
    CHECK (column1 > 0)
);
```

This query creates a table with a check constraint on "column1", ensuring that only positive values are allowed.

Including constraints in your table definitions helps maintain data integrity and enforce rules for data consistency. Constraints provide a way to control the data entered into the tables, ensuring that it adheres to the defined rules and relationships.

5. Creating and Managing other database objects
a. Creating Views

Solutions

Creating views is a way to create virtual tables in SQL. A view is a saved SQL query that behaves like a table but does not store data directly. It provides a way to present data from one or more tables in a customized and convenient way. Here's an overview of how to create views in SQL:

1. Creating a View:
   To create a view, you need to use the CREATE VIEW statement. This statement defines the view name, the columns to include, and the query that retrieves the data. Here's the syntax:

   ```sql
   CREATE VIEW view_name AS
   SELECT column1, column2, ...
   FROM table_name
   WHERE condition;
   ```

   For example, to create a view called "customer_orders" that includes the customer name and the number of orders for each customer from the "customers" and "orders" tables, you can use the following query:

   ```sql
   CREATE VIEW customer_orders AS
   SELECT customers.customer_name, COUNT(orders.order_id) AS order_count
   FROM customers
   JOIN orders ON customers.customer_id = orders.customer_id
   ```

```
GROUP BY customers.customer_name;
```

This query creates a view that combines data from the "customers" and "orders" tables and presents it as "customer_orders" with columns for customer name and order count.

2. Using Views:
Once a view is created, you can use it like a regular table in subsequent SQL queries. You can query the view, join it with other tables, or apply filters on it, just like you would with a physical table. For example:

```sql
SELECT customer_name, order_count
FROM customer_orders
WHERE order_count > 10;
```

This query retrieves the customer names and order counts from the "customer_orders" view, filtering for customers with more than 10 orders.

3. Modifying and Dropping Views:
Views can be modified or dropped using the ALTER VIEW and DROP VIEW statements, respectively. Here are the basic syntaxes:

- Modifying a View:
  ```sql
  ALTER VIEW view_name AS
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition;
  ```

- Dropping a View:
  ```sql
  DROP VIEW view_name;
  ```

Modifying a view allows you to redefine the underlying query or columns of the view, while dropping a view permanently removes it from the database.

Views provide a convenient way to present data from multiple tables or complex queries in a simplified manner. They can be used to improve query performance, provide data security by restricting access, and encapsulate complex logic. Views are a powerful tool for data modeling and data presentation in SQL databases.

b. Other Database Objects

Solutions

Certainly! Here's an example of SQL code that creates a table, a view, an index, and a stored procedure:

```sql
-- Creating a table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

```
-- Creating a view
CREATE VIEW high_salary_employees AS
SELECT employee_id, first_name, last_name
FROM employees
WHERE salary > 50000;

-- Creating an index
CREATE INDEX idx_employee_last_name ON employees(last_name);

-- Creating a stored procedure
CREATE PROCEDURE get_employee_count()
BEGIN
    SELECT COUNT(*) AS total_employees FROM employees;
END;
```

In the above code, we create a table called "employees" with columns for employee ID, first name, last name, and salary. Then, we create a view named "high_salary_employees" that selects employees with salaries greater than 50000. Next, we create an index on the "last_name" column of the "employees" table to improve query performance. Finally, we create a stored procedure called "get_employee_count" that retrieves the total count of employees from the "employees" table.

c. Controlling User Access

Solutions

Controlling user access is a crucial aspect of database security and involves managing user privileges and permissions. SQL provides mechanisms to grant or revoke permissions to users and control their

access to database objects. Here's an overview of how to control user access in SQL:

1. Creating Users:
To control user access, you first need to create user accounts. User accounts are typically created at the database level and are associated with specific login credentials. The process of creating users may vary depending on the database system you are using. Here's an example of creating a user:

```sql
CREATE USER username IDENTIFIED BY 'password';
```

Replace "username" with the desired username and 'password' with the user's password.

2. Granting Privileges:
After creating users, you can grant them specific privileges to perform operations on database objects. Privileges control what actions a user can perform, such as reading, modifying, or deleting data. The specific privileges and their syntax may vary depending on the database system. Here's an example of granting privileges:

```sql
GRANT SELECT, INSERT, UPDATE ON table_name TO username;
```

This query grants the user specified by "username" the privileges to select, insert, and update data on the "table_name" table. You can grant different privileges on different objects as needed.

3. Revoking Privileges:
   If you need to revoke privileges from a user, you can use the REVOKE statement. This removes previously granted privileges from a user account. Here's an example of revoking privileges:

   ```sql
   REVOKE INSERT, UPDATE ON table_name FROM username;
   ```

   This query revokes the INSERT and UPDATE privileges from the user specified by "username" on the "table_name" table.

4. Managing Roles:
   Roles are named groups of privileges that can be assigned to users. By assigning roles to users, you can simplify user management and maintain consistent access control across multiple users. The process of creating and managing roles may vary depending on the database system.

5. Using Views and Stored Procedures:
   Views and stored procedures can be used to control user access indirectly. By granting or revoking privileges on views and stored procedures, you can control what data and operations users have access to. Users can interact with the views and execute the stored procedures without directly accessing the underlying tables.

Controlling user access is essential for maintaining the security and integrity of a database. It allows you to restrict unauthorized access, provide appropriate permissions to users, and ensure that data is accessed and modified only by authorized individuals or applications. The specific syntax and methods for controlling user access may vary depending on the database system you are using.

6. Using SET operators, Date/Time Functions, GROUP BY clause (advanced features) and advanced subqueries

a. Using SET Operators

Solutions

SET operators in SQL allow you to combine the results of multiple queries into a single result set. There are three primary SET operators: UNION, INTERSECT, and EXCEPT. Here's an overview of each SET operator and its usage:

1. UNION Operator:
   The UNION operator combines the result sets of two or more SELECT statements into a single result set, removing duplicate rows. The columns in the SELECT statements must have compatible data types and be in the same order. Here's an example:

   ```sql
   SELECT column1, column2, ...
   FROM table1
   UNION
   SELECT column1, column2, ...
   FROM table2;
   ```

   The UNION operator returns the combined result set of the two SELECT statements.

2. INTERSECT Operator:
   The INTERSECT operator returns the common rows between the result sets of two or more SELECT statements. It returns only the rows that

appear in all SELECT statements and removes duplicate rows. The columns in the SELECT statements must have compatible data types and be in the same order. Here's an example:

```sql
SELECT column1, column2, ...
FROM table1
INTERSECT
SELECT column1, column2, ...
FROM table2;
```

The INTERSECT operator returns the common rows between the two SELECT statements.

3. EXCEPT Operator:
The EXCEPT operator returns the rows from the first SELECT statement that do not appear in the result set of the second SELECT statement. It removes duplicate rows. The columns in the SELECT statements must have compatible data types and be in the same order. Here's an example:

```sql
SELECT column1, column2, ...
FROM table1
EXCEPT
SELECT column1, column2, ...
FROM table2;
```

The EXCEPT operator returns the rows from the first SELECT statement that do not appear in the second SELECT statement.

SET operators provide a way to combine and compare the results of multiple queries, allowing for more complex and flexible data retrieval and analysis. They can be used to consolidate data from different tables, find common elements, or retrieve unique records.

b. Datetime Functions

Solutions

Datetime functions in SQL allow you to perform various operations and manipulations on date and time values. These functions help you extract specific components from datetime values, perform calculations, and format dates and times. Here are some commonly used datetime functions in SQL:

1. CURRENT_DATE:
   The CURRENT_DATE function returns the current date in the system's default date format.

   Example:
   ```sql
   SELECT CURRENT_DATE;
   ```

   This query returns the current date.

2. CURRENT_TIME:
   The CURRENT_TIME function returns the current time in the system's default time format.

   Example:
   ```sql
   ```

```
SELECT CURRENT_TIME;
```

This query returns the current time.

3. CURRENT_TIMESTAMP:
The CURRENT_TIMESTAMP function returns the current date and time in the system's default timestamp format.

Example:
```sql
SELECT CURRENT_TIMESTAMP;
```

This query returns the current timestamp.

4. DATE Functions:
SQL provides various functions to extract specific components from date values, such as year, month, day, and more. Here are some examples:

- EXTRACT:
  ```sql
  SELECT EXTRACT(YEAR FROM date_column) AS year
  FROM table_name;
  ```

- YEAR:
  ```sql
  SELECT YEAR(date_column) AS year
  FROM table_name;
  ```

- MONTH, DAY, HOUR, MINUTE, SECOND, etc.

5. DATE_ADD and DATE_SUB:
These functions allow you to add or subtract a specific interval from a date or timestamp.

Example:
```sql
SELECT DATE_ADD(date_column, INTERVAL 1 MONTH) AS new_date
FROM table_name;
```

This query adds one month to the date in the "date_column" and returns the updated date.

6. DATE_FORMAT:
The DATE_FORMAT function allows you to format date and time values into a specified format.

Example:
```sql
SELECT DATE_FORMAT(date_column, 'YYYY-MM-DD') AS formatted_date
FROM table_name;
```

This query formats the "date_column" into the format 'YYYY-MM-DD'.

c. Enhancements to the GROUP BY Clause

Solutions

The GROUP BY clause in SQL is used to group rows based on one or more columns and perform aggregate functions on each group. It allows you to summarize and analyze data at a higher level. In addition to the basic usage, there are some enhancements and additional features that can be used with the GROUP BY clause. Here are a few examples:

1. GROUP BY with Multiple Columns:
   The GROUP BY clause can be used with multiple columns to create more granular groups. This allows you to group data based on multiple criteria and obtain aggregated results for each combination of columns. Here's an example:

   ```sql
   SELECT column1, column2, SUM(quantity)
   FROM table_name
   GROUP BY column1, column2;
   ```

   This query groups the data based on both "column1" and "column2" and calculates the sum of the "quantity" for each combination of values.

2. GROUP BY with Rollup:
   The ROLLUP modifier extends the functionality of the GROUP BY clause by adding extra rows that represent subtotals and grand totals. It generates a result set that includes not only the individual groups but also rows with aggregated values for each level of grouping. Here's an example:

   ```sql
   SELECT column1, column2, SUM(quantity)
   FROM table_name
```

```
GROUP BY ROLLUP (column1, column2);
```

This query produces a result set that includes subtotals and grand totals for each level of grouping based on "column1" and "column2".

3. GROUP BY with Cube:
The CUBE modifier is similar to ROLLUP but generates a more comprehensive result set. It creates subtotals and grand totals for all possible combinations of grouping columns. Here's an example:

```sql
SELECT column1, column2, SUM(quantity)
FROM table_name
GROUP BY CUBE (column1, column2);
```

This query generates a result set that includes subtotals and grand totals for all possible combinations of "column1" and "column2".

These enhancements to the GROUP BY clause provide more flexibility and options for analyzing and summarizing data in SQL. They allow you to group data at different levels, generate subtotals and grand totals, and obtain a more comprehensive view of the data. The specific syntax and availability of these enhancements may vary depending on the database system you are using.

d. Advanced Subqueries

Solutions

Advanced subqueries in SQL allow you to create more complex queries by nesting one or more queries within another query. Subqueries can be used in various parts of a SQL statement, such as the SELECT, FROM, WHERE, or HAVING clauses. Here are some examples of advanced subqueries:

1. Subqueries in the SELECT Clause:
   Subqueries can be used within the SELECT clause to retrieve a single value or result for each row returned by the outer query. The subquery is enclosed within parentheses and can be used as an expression. Here's an example:

   ```sql
   SELECT column1, column2, (SELECT COUNT(*) FROM table2 WHERE table2.column = table1.column) AS count
   FROM table1;
   ```

   This query retrieves values from "table1" and includes a subquery within the SELECT clause that calculates the count from "table2" based on a condition in "table1".

2. Subqueries in the FROM Clause:
   Subqueries can be used in the FROM clause to treat the result of a subquery as a temporary table. This allows you to perform additional operations on the subquery result. Here's an example:

   ```sql
   SELECT t1.column1, t2.column2
   FROM (SELECT column1 FROM table1 WHERE condition) AS t1
   JOIN table2 AS t2 ON t1.column1 = t2.column1;
   ```

This query uses a subquery in the FROM clause to create a temporary table "t1" and then performs a join operation with "table2" based on a column in the subquery result.

3. Subqueries in the WHERE Clause:

Subqueries can be used within the WHERE clause to filter data based on the results of a nested query. The subquery is enclosed within parentheses and is typically written within the condition of the WHERE clause. Here's an example:

```sql
SELECT column1, column2
FROM table1
WHERE column1 IN (SELECT column FROM table2 WHERE condition);
```

This query retrieves data from "table1" and filters it based on the result of a subquery that selects a column from "table2" based on a condition.

Advanced subqueries provide a powerful way to solve complex data retrieval and analysis challenges in SQL. They allow you to break down complex problems into smaller, more manageable parts and leverage the results of one query within another. Subqueries can be used creatively to achieve a wide range of operations and allow for more sophisticated SQL queries.

7. PL/SQL Basics
a. Declaring Variables

Solutions

In PL/SQL, variables are used to store and manipulate data within a program. To declare variables in PL/SQL, you can use the DECLARE keyword. Here's an example of declaring variables in PL/SQL:

```plsql
DECLARE
  variable1 datatype;
  variable2 datatype := default_value;
BEGIN
  -- PL/SQL code goes here
END;
```

In the above code, "datatype" represents the data type of the variable, such as VARCHAR2, NUMBER, DATE, etc. You can declare multiple variables in the DECLARE section, each on a separate line.

You can also assign an initial value to a variable using the assignment operator (:=). This is optional, and if you don't specify a default value, the variable will be initialized to NULL.

Here's an example that declares and initializes variables in PL/SQL:

```plsql
DECLARE
  name VARCHAR2(50) := 'John';
  age NUMBER := 30;
```

```
  isMarried BOOLEAN := TRUE;
BEGIN
  -- PL/SQL code goes here
END;
```

In this example, we declare three variables: "name" of type VARCHAR2, "age" of type NUMBER, and "isMarried" of type BOOLEAN. We also assign initial values to each variable.

Once variables are declared, you can use them within the PL/SQL block to store, manipulate, or retrieve data as needed.

b. Writing Executable Statements

Solutions

In PL/SQL, executable statements are used to perform actions, manipulate data, control flow, and interact with the database. Executable statements are written within the BEGIN and END block of a PL/SQL program. Here are some examples of executable statements in PL/SQL:

1. Assignment Statement:
   An assignment statement is used to assign a value to a variable. It follows the syntax: variable := expression. Here's an example:

   ```plsql
   DECLARE
     name VARCHAR2(50);
   BEGIN
     name := 'John';
   END;
   ```

```

This statement assigns the value 'John' to the variable "name".

2. Conditional Statements:
Conditional statements allow you to perform different actions based on a condition. PL/SQL provides IF-THEN-ELSE and CASE statements for conditional branching. Here's an example using IF-THEN-ELSE:

```plsql
DECLARE
  age NUMBER := 30;
BEGIN
  IF age >= 18 THEN
    DBMS_OUTPUT.PUT_LINE('Adult');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Minor');
  END IF;
END;
```

This statement checks the value of the variable "age" and displays 'Adult' if the condition is true, otherwise 'Minor' is displayed.

3. Loop Statements:
Loop statements allow you to repeat a block of code until a condition is met. PL/SQL provides various loop statements such as LOOP, WHILE, and FOR loops. Here's an example using a WHILE loop:

```plsql
DECLARE
  counter NUMBER := 1;
```
```

```
BEGIN
  WHILE counter <= 5 LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || counter);
    counter := counter + 1;
  END LOOP;
END;
```

This statement initializes a counter variable and uses a WHILE loop to repeatedly display the counter value until it reaches 5.

4. SQL Statements:

PL/SQL allows you to execute SQL statements within the block to interact with the database. For example, you can use SELECT, INSERT, UPDATE, and DELETE statements to retrieve, modify, or delete data. Here's an example using a SELECT statement:

```plsql
DECLARE
  name VARCHAR2(50);
BEGIN
  SELECT first_name INTO name FROM employees WHERE employee_id = 123;
  DBMS_OUTPUT.PUT_LINE('Name: ' || name);
END;
```

This statement retrieves the first name of an employee with the employee ID 123 and displays it using the DBMS_OUTPUT.PUT_LINE procedure.

c. Interacting with the Oracle Server

Solutions

In PL/SQL, you can interact with the Oracle Server using various built-in packages and procedures. These provide functionality to perform database operations, handle exceptions, retrieve information, and more. Here are some ways to interact with the Oracle Server in PL/SQL:

1. Data Manipulation Language (DML) Statements:
   PL/SQL allows you to execute SQL statements, including Data Manipulation Language (DML) statements like INSERT, UPDATE, DELETE, and SELECT. You can use these statements to manipulate data in the database. Here's an example of executing an INSERT statement:

   ```plsql
   DECLARE
     v_id NUMBER := 1;
     v_name VARCHAR2(50) := 'John';
   BEGIN
     INSERT INTO employees (employee_id, employee_name) VALUES (v_id, v_name);
   END;
   ```

   This PL/SQL block inserts a new row into the "employees" table using the INSERT statement.

2. Built-in Packages:
   Oracle provides several built-in packages that offer a wide range of functionality. For example, the DBMS_OUTPUT package allows you to display output from PL/SQL programs, and the DBMS_SQL package

allows you to execute dynamic SQL statements. Here's an example using the DBMS_OUTPUT package:

```plsql
DECLARE
  v_name VARCHAR2(50) := 'John';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
END;
```

This PL/SQL block uses the DBMS_OUTPUT.PUT_LINE procedure to display the value of the "v_name" variable.

3. Exception Handling:
PL/SQL provides mechanisms for handling exceptions that may occur during program execution. You can use the EXCEPTION block to catch and handle specific exceptions. Here's an example:

```plsql
DECLARE
  v_id NUMBER := 1;
BEGIN
  SELECT employee_name INTO v_name FROM employees WHERE employee_id = v_id;
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found.');
END;
```

This PL/SQL block attempts to retrieve the name of an employee with the ID stored in the "v_id" variable. If no data is found, the NO_DATA_FOUND exception is raised and the code within the EXCEPTION block is executed.

d. Writing Control Structures

Solutions

Control structures in PL/SQL allow you to control the flow of execution within your program. They provide the ability to make decisions, perform repetitive tasks, and handle exceptional situations. Here are some commonly used control structures in PL/SQL:

1. IF-THEN-ELSE:
   The IF-THEN-ELSE statement allows you to perform different actions based on a condition. It follows the syntax:

   ```plsql
   IF condition THEN
     -- Code to execute when the condition is true
   ELSE
     -- Code to execute when the condition is false
   END IF;
   ```

   Here's an example:

   ```plsql
   DECLARE
     num NUMBER := 10;
```

```
BEGIN
  IF num > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Positive');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Non-positive');
  END IF;
END;
```

This code checks if the value of "num" is greater than 0 and displays 'Positive' or 'Non-positive' accordingly.

2. CASE:
The CASE statement allows you to perform different actions based on multiple conditions. It follows the syntax:

```plsql
CASE expression
  WHEN value1 THEN
    -- Code to execute when expression equals value1
  WHEN value2 THEN
    -- Code to execute when expression equals value2
  ...
  ELSE
    -- Code to execute when none of the above conditions match
END CASE;
```

Here's an example:

```plsql
DECLARE
```

```
  day VARCHAR2(20) := 'Tuesday';
BEGIN
  CASE day
    WHEN 'Monday' THEN
      DBMS_OUTPUT.PUT_LINE('Start of the week');
    WHEN 'Tuesday' THEN
      DBMS_OUTPUT.PUT_LINE('Second day');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Other day');
  END CASE;
END;
```

This code checks the value of "day" and displays a corresponding message based on the day.

3. LOOP:

The LOOP statement allows you to perform repetitive tasks until a certain condition is met. It can be used with various loop control structures such as EXIT, WHILE, and FOR. Here's an example using a basic loop:

```plsql
DECLARE
  counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || counter);
    counter := counter + 1;
    EXIT WHEN counter > 5;
  END LOOP;
END;
```

8. Composite data types, cursors and exceptions.
a. Working with Composite Data Types

Solutions

In PL/SQL, composite data types allow you to create structured variables that can hold multiple related values. These composite data types include records, collections (nested tables, varrays, and associative arrays), and object types. Here's an overview of working with composite data types in PL/SQL:

1. Records:
   Records are composite data types that can hold multiple values of different types. They are similar to rows in a database table. You can define a record type using the %ROWTYPE attribute, which automatically matches the structure of a database table or cursor. Here's an example:

```plsql
DECLARE
  TYPE EmployeeRecord IS RECORD (
    id NUMBER,
    name VARCHAR2(50),
    salary NUMBER
  );

  employee EmployeeRecord;
BEGIN
  employee.id := 1;
  employee.name := 'John Doe';
  employee.salary := 5000;
END;
```

```

In this example, we define a record type called "EmployeeRecord" with three fields: id, name, and salary. Then, we declare a variable "employee" of the "EmployeeRecord" type and assign values to its fields.

2. Collections:
Collections are composite data types that can hold multiple values of the same type. There are three types of collections in PL/SQL: nested tables, varrays, and associative arrays.

- Nested Tables:
Nested tables are one-dimensional arrays that can be dynamically resized. They are useful when the number of elements may vary. Here's an example:

```plsql
DECLARE
  TYPE NumberList IS TABLE OF NUMBER;
  numbers NumberList := NumberList(1, 2, 3, 4, 5);
BEGIN
  -- Accessing elements
  DBMS_OUTPUT.PUT_LINE(numbers(2)); -- Output: 2

  -- Adding elements
  numbers.EXTEND;
  numbers(numbers.LAST) := 6;

  -- Iterating through elements
  FOR i IN numbers.FIRST..numbers.LAST LOOP
    DBMS_OUTPUT.PUT_LINE(numbers(i));
  END LOOP;
```
```

```
END;
```

- Varrays:

Varrays (variable-size arrays) have a fixed size that you specify during declaration. They are useful when the number of elements is known in advance. Here's an example:

```plsql
DECLARE
  TYPE NumberArray IS VARRAY(5) OF NUMBER;
  numbers NumberArray := NumberArray(1, 2, 3, 4, 5);
BEGIN
  -- Accessing elements
  DBMS_OUTPUT.PUT_LINE(numbers(3)); -- Output: 3

  -- Adding elements
  numbers.EXTEND;
  numbers(numbers.LAST) := 6;

  -- Iterating through elements
  FOR i IN 1..numbers.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(numbers(i));
  END LOOP;
END;
```

- Associative Arrays:

Associative arrays (also known as index-by tables) are unordered collections that use arbitrary keys to access elements. They are useful when you want to associate values with specific keys. Here's an example:

```plsql
DECLARE
  TYPE SalaryByEmployeeID IS TABLE OF NUMBER INDEX BY
VARCHAR2(10);
  salaries SalaryByEmployeeID;
BEGIN
  -- Adding elements
  salaries('E001') := 5000;
  salaries('E002') := 6000;

  -- Accessing elements
  DBMS_OUTPUT.PUT_LINE(salaries('E001')); -- Output: 5000

  -- Iterating through elements
  FOR key IN salaries.FIRST..salaries.LAST LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || key || ', Salary: ' ||
salaries(key));
  END LOOP;
END;
```

3. Object Types:
Object types allow you to define your own custom data types with multiple attributes. They are similar to records but can have additional behaviors and methods. Defining and working with object types goes beyond the scope of this response but involves creating object types using the CREATE TYPE statement, instantiating objects, and accessing their attributes and methods.

b. Writing Explicit Cursors

Solutions

Explicit cursors in PL/SQL allow you to retrieve and process query results row by row. They provide more control and flexibility compared to implicit cursors. Here's how you can write and use explicit cursors in PL/SQL:

1. Cursor Declaration:
   To declare an explicit cursor, you need to define a cursor variable of a specific type that matches the result set structure. The cursor variable serves as a pointer to the result set. Here's an example of declaring an explicit cursor:

   ```plsql
   DECLARE
     CURSOR cursor_name IS
       SELECT column1, column2, ...
       FROM table_name
       WHERE condition;
   BEGIN
     -- PL/SQL code goes here
   END;
   ```

   In this example, "cursor_name" is the name of the cursor variable, and the SELECT statement defines the result set structure and conditions.

2. Opening the Cursor:
   Once the cursor is declared, you need to open it before fetching the rows. The OPEN statement is used to open the cursor and associate it with the query results. Here's an example:

   ```plsql
   ```

```
DECLARE
  CURSOR cursor_name IS
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;
BEGIN
  OPEN cursor_name;
  -- PL/SQL code goes here
END;
```

3. Fetching Rows:
   After opening the cursor, you can fetch the rows one by one using the FETCH statement. The FETCH statement retrieves the next row from the cursor and assigns the column values to variables. Here's an example:

```plsql
DECLARE
  CURSOR cursor_name IS
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;
  variable1 datatype;
  variable2 datatype;
BEGIN
  OPEN cursor_name;
  LOOP
    FETCH cursor_name INTO variable1, variable2;
    EXIT WHEN cursor_name%NOTFOUND;
    -- Process the fetched row
  END LOOP;
  CLOSE cursor_name;
```

```
END;
```

In this example, "variable1" and "variable2" are variables that store the column values retrieved from each fetched row. The LOOP iterates through the rows, and the EXIT WHEN statement exits the loop when there are no more rows to fetch.

4. Closing the Cursor:
After fetching all the required rows, it's good practice to close the cursor using the CLOSE statement. This releases the resources associated with the cursor. Here's an example:

```plsql
DECLARE
  CURSOR cursor_name IS
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;
BEGIN
  OPEN cursor_name;
  -- PL/SQL code goes here
  CLOSE cursor_name;
END;
```

Explicit cursors allow you to fetch and process query results in a controlled manner. They are particularly useful when you need to perform row-by-row processing, apply conditions, or manipulate the fetched data within your PL/SQL code.

c. Handling Exceptions

Solutions

Exception handling in PL/SQL allows you to handle and manage exceptional situations that may occur during the execution of your program. By using exception handlers, you can gracefully handle errors, perform error-specific actions, and ensure the proper flow of your code. Here's how you can handle exceptions in PL/SQL:

1. Exception Declaration:
   Before handling exceptions, you can declare custom exceptions using the DECLARE keyword. This allows you to define specific exceptions for different scenarios. Here's an example of declaring a custom exception:

   ```plsql
   DECLARE
     custom_exception EXCEPTION;
   BEGIN
     -- PL/SQL code goes here
   EXCEPTION
     WHEN custom_exception THEN
       -- Exception handling code for custom_exception
   END;
   ```

   In this example, "custom_exception" is a user-defined exception that can be raised and handled within the PL/SQL block.

2. Exception Handling:
   To handle exceptions, you use the EXCEPTION block. The EXCEPTION block contains the code that is executed when an exception is raised. You can specify multiple WHEN clauses to handle different types of exceptions. Here's an example:

```plsql
BEGIN
  -- PL/SQL code goes here
EXCEPTION
  WHEN exception_type1 THEN
    -- Exception handling code for exception_type1
  WHEN exception_type2 THEN
    -- Exception handling code for exception_type2
  ...
  WHEN OTHERS THEN
    -- Exception handling code for all other exceptions
END;
```

In this example, "exception_type1" and "exception_type2" represent specific exceptions that you want to handle individually. The WHEN OTHERS clause is used to handle all other exceptions that are not explicitly specified.

3. Raising Exceptions:

To raise an exception explicitly, you can use the RAISE statement. This allows you to trigger an exception at a specific point in your code. Here's an example:

```plsql
BEGIN
  IF condition THEN
    RAISE custom_exception;
  END IF;
EXCEPTION
  WHEN custom_exception THEN
```

```
    -- Exception handling code for custom_exception
  END;
  ```

  In this example, the "custom_exception" is raised if a certain condition is met, and the exception is then handled in the corresponding WHEN clause.

4. Exception Propagation:
  Exceptions can be propagated to the calling program or block using the RAISE statement without an exception name. This allows you to handle exceptions at a higher level in the program. Here's an example:

  ```plsql
  DECLARE
    custom_exception EXCEPTION;
  BEGIN
    raise_application_error(-20001, 'Custom error message');
  EXCEPTION
    WHEN OTHERS THEN
      -- Exception handling code
  END;
  ```

  In this example, the RAISE_APPLICATION_ERROR procedure raises a predefined exception and propagates it to the calling program or block for handling.

Exception handling in PL/SQL enables you to manage errors and exceptional situations in your code. By anticipating potential issues and implementing appropriate exception handlers, you can ensure that your program handles errors gracefully, provides meaningful feedback, and maintains the expected flow of execution.

9. Procedures and Functions
a. Creating Procedures

Solutions

Procedures in PL/SQL are named blocks of code that can be invoked and executed. They allow you to encapsulate a sequence of actions into a reusable unit. Here's how you can create and use procedures in PL/SQL:

1. Procedure Declaration:
   To create a procedure, you need to define its name, parameters (if any), and the code block within the DECLARE and BEGIN-END keywords. Here's an example of creating a procedure without parameters:

   ```plsql
   CREATE OR REPLACE PROCEDURE procedure_name IS
   BEGIN
     -- PL/SQL code goes here
   END procedure_name;
   ```

   In this example, "procedure_name" is the name of the procedure, and the PL/SQL code is written between the BEGIN and END keywords.

2. Procedure Parameters:
   Procedures can have parameters to accept input values or variables. Parameters can be of different types, such as IN (input), OUT (output), or IN OUT (both input and output). Here's an example of creating a procedure with parameters:

```plsql
CREATE OR REPLACE PROCEDURE procedure_name (parameter1 IN datatype1, parameter2 OUT datatype2) IS
BEGIN
  -- PL/SQL code goes here
END procedure_name;
```

In this example, "parameter1" is an IN parameter of type "datatype1", and "parameter2" is an OUT parameter of type "datatype2". You can access the values of the OUT parameters outside the procedure.

3. Calling a Procedure:
To invoke a procedure, you use the CALL or EXECUTE statement followed by the procedure name and any required arguments. Here's an example:

```plsql
BEGIN
  procedure_name(argument1, argument2);
END;
```

In this example, "procedure_name" is the name of the procedure, and "argument1" and "argument2" are the values or variables passed to the procedure.

4. Executing a Procedure:
Once the procedure is called, the code within the procedure block is executed. You can include any valid PL/SQL code, such as SQL statements, control structures, variable declarations, and more, to perform specific actions.

5. Example:
   Here's an example of a simple procedure that displays a message:

   ```plsql
   CREATE OR REPLACE PROCEDURE display_message IS
   BEGIN
     DBMS_OUTPUT.PUT_LINE('Hello, world!');
   END display_message;
   ```

   In this example, the procedure "display_message" is created, and when called, it prints 'Hello, world!' using the DBMS_OUTPUT.PUT_LINE procedure.

Procedures provide a way to encapsulate and reuse code in PL/SQL. They allow you to modularize your code, improve code readability, and promote code reuse by invoking the procedure multiple times from different parts of your program.

b. Creating Functions

Solutions

Functions in PL/SQL are named blocks of code that return a single value. They are similar to procedures but have a RETURN statement to provide a result. Here's how you can create and use functions in PL/SQL:

1. Function Declaration:
   To create a function, you need to define its name, parameters (if any), return type, and the code block within the DECLARE and BEGIN-END keywords. Here's an example of creating a function with parameters:

```plsql
CREATE OR REPLACE FUNCTION function_name (parameter1 datatype1, parameter2 datatype2) RETURN return_type IS
BEGIN
  -- PL/SQL code goes here
  RETURN result;
END function_name;
```

In this example, "function_name" is the name of the function, and "parameter1" and "parameter2" are the input parameters. "return_type" represents the data type of the return value, and "result" is the value that is returned by the function.

2. Function Parameters:
Functions can have parameters to accept input values or variables. Parameters can be of different types, such as IN (input), OUT (output), or IN OUT (both input and output). Here's an example of creating a function with parameters:

```plsql
CREATE OR REPLACE FUNCTION function_name (parameter1 IN datatype1, parameter2 OUT datatype2) RETURN return_type IS
BEGIN
  -- PL/SQL code goes here
  RETURN result;
END function_name;
```

In this example, "parameter1" is an IN parameter of type "datatype1", and "parameter2" is an OUT parameter of type "datatype2". The OUT

parameter can be assigned a value inside the function and will be accessible outside the function.

3. Calling a Function:
To invoke a function, you can use it in expressions or assignments like any other PL/SQL expression. Here's an example:

```plsql
DECLARE
  variable return_type;
BEGIN
  variable := function_name(argument1, argument2);
END;
```

In this example, "function_name" is the name of the function, and "argument1" and "argument2" are the values or variables passed to the function. The return value of the function is assigned to the variable "variable".

4. Executing a Function:
When a function is called, the code within the function block is executed. You can include any valid PL/SQL code, such as SQL statements, control structures, variable declarations, and more, to perform specific actions. The RETURN statement is used to return the result from the function.

5. Example:
Here's an example of a simple function that calculates the square of a number:

```plsql
```

```
CREATE OR REPLACE FUNCTION square (num NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN num * num;
  END square;
```

In this example, the function "square" takes a parameter "num" of type NUMBER and returns the square of that number.

Functions provide a way to encapsulate reusable code and perform calculations or transformations on data. They can be called from other PL/SQL code or used in SQL statements as part of expressions. Functions enhance code modularity and promote code reuse by encapsulating specific operations that can be used multiple times within a program.

c. Managing Subprograms

Solutions

Managing subprograms in PL/SQL involves aspects such as overloading, forward declaration, and visibility within a program. Let's explore these concepts:

1. Overloading Subprograms:
   Overloading allows you to define multiple subprograms with the same name but different parameter lists. This enables you to create subprograms that perform similar operations but with different inputs. Overloaded subprograms are distinguished by their parameter types or number of parameters. Here's an example:

```plsql
```

```
CREATE OR REPLACE FUNCTION calculate_area(radius NUMBER)
RETURN NUMBER IS
BEGIN
  -- Calculate area of a circle
END calculate_area;

CREATE OR REPLACE FUNCTION calculate_area(length NUMBER,
width NUMBER) RETURN NUMBER IS
BEGIN
  -- Calculate area of a rectangle
END calculate_area;
```

In this example, two functions named "calculate_area" are defined. The first calculates the area of a circle based on the radius, while the second calculates the area of a rectangle based on its length and width. The appropriate function is invoked based on the number and types of arguments passed.

2. Forward Declaration:

In PL/SQL, you can use forward declaration to declare a subprogram before its implementation. This is useful when you have mutually recursive subprograms or when you want to define subprograms in a specific order. Here's an example:

```plsql
DECLARE
  FUNCTION function1 RETURN datatype;
  PROCEDURE procedure1;
BEGIN
  -- Subprogram implementation
END;
```

```
FUNCTION function1 RETURN datatype IS
BEGIN
  -- Function implementation
END;

PROCEDURE procedure1 IS
BEGIN
  -- Procedure implementation
END;
```

In this example, the subprograms "function1" and "procedure1" are declared first, and their implementation follows. This allows you to define the subprograms in any order you prefer.

3. Visibility within a Program:

Subprograms in PL/SQL have their own scope, and they can be declared within a block or at the schema level. Subprograms declared within a block are only visible within that block, while subprograms declared at the schema level are visible throughout the program. Here's an example:

```plsql
DECLARE
  PROCEDURE inner_procedure IS
  BEGIN
    -- Inner procedure implementation
  END;

  PROCEDURE outer_procedure IS
    PROCEDURE nested_procedure IS
    BEGIN
```

```
    -- Nested procedure implementation
  END;
 BEGIN
  -- Outer procedure implementation
 END;
BEGIN
 -- Main block implementation
END;
```

In this example, the subprogram "inner_procedure" is only visible within the main block. The subprogram "nested_procedure" is only visible within the "outer_procedure". Each subprogram has its own scope and can be accessed within its respective scope.

Managing subprograms in PL/SQL allows you to organize your code, define multiple subprograms with the same name but different parameters, handle mutual recursion, and control the visibility of subprograms within a program. These features enhance code modularity, reusability, and maintainability in PL/SQL programs.

d. Creating Packages

Solutions

Creating packages in PL/SQL is a way to organize related procedures, functions, variables, and other PL/SQL constructs into a cohesive unit. Packages provide modularity, encapsulation, and easier code management. Here's how you can create packages in PL/SQL:

1. Package Specification:

The package specification defines the public interface of the package. It declares the procedures, functions, variables, constants, and types that are accessible outside the package. Here's an example of creating a package specification:

```plsql
CREATE OR REPLACE PACKAGE package_name IS
  -- Declarations of public items
  PROCEDURE procedure_name;
  FUNCTION function_name RETURN datatype;
END package_name;
```

In this example, "package_name" is the name of the package, and the package specification includes the declarations of the public items, such as procedures and functions.

2. Package Body:
The package body contains the implementation of the package. It includes the actual code for the procedures and functions declared in the package specification, as well as any private items that are not accessible outside the package. Here's an example of creating a package body:

```plsql
CREATE OR REPLACE PACKAGE BODY package_name IS
  -- Declarations of private items
  VARIABLE variable_name datatype;

  -- Implementation of procedures and functions
  PROCEDURE procedure_name IS
  BEGIN
    -- Procedure implementation
```

```
  END procedure_name;

  FUNCTION function_name RETURN datatype IS
  BEGIN
    -- Function implementation
  END function_name;
END package_name;
```

In this example, "package_name" is the name of the package, and the package body includes the declarations and implementation of private items and the implementation of the procedures and functions.

3. Using the Package:
Once the package is created, you can use its procedures and functions in other PL/SQL code. Here's an example of using a package:

```plsql
BEGIN
  package_name.procedure_name;
  variable := package_name.function_name;
END;
```

In this example, "package_name.procedure_name" calls the procedure defined in the package, and "variable := package_name.function_name" calls the function and assigns its return value to a variable.

Packages provide a way to organize and modularize your PL/SQL code. They encapsulate related functionality, promote code reuse, and improve code maintainability. By separating the package specification from the package body, you can control the visibility of items and create a clear

separation between the public interface and the implementation details of the package.

10. Creating Database Triggers

Solution

Database triggers in PL/SQL are stored programs that are automatically executed in response to specific events or actions that occur on a table or view. Triggers allow you to enforce business rules, maintain data integrity, perform data validation, and automate tasks. Here's how you can create database triggers in PL/SQL:

1. Trigger Creation:
   To create a database trigger, you use the CREATE TRIGGER statement followed by the trigger name, the table or view name, and the trigger timing (BEFORE or AFTER) and event (INSERT, UPDATE, or DELETE). Here's an example:

   ```plsql
   CREATE OR REPLACE TRIGGER trigger_name
   BEFORE INSERT OR UPDATE OR DELETE ON table_name
   FOR EACH ROW
   BEGIN
     -- PL/SQL code goes here
   END;
   ```

   In this example, "trigger_name" is the name of the trigger, "table_name" is the name of the table or view, and the trigger is specified to fire before an INSERT, UPDATE, or DELETE operation on each row.

2. Trigger Timing and Event:
   The timing and event of a trigger define when the trigger fires and on what action. The timing can be BEFORE or AFTER, and the event can be

INSERT, UPDATE, or DELETE. You can specify multiple events using the OR keyword. Here's an example:

```plsql
CREATE OR REPLACE TRIGGER trigger_name
AFTER INSERT OR UPDATE ON table_name
FOR EACH ROW
BEGIN
  -- PL/SQL code goes here
END;
```

In this example, the trigger fires after an INSERT or UPDATE operation on each row of the specified table.

3. Trigger Body:
The body of the trigger contains the PL/SQL code that is executed when the trigger fires. The code can include SQL statements, PL/SQL logic, variable declarations, and more. You can refer to the OLD and NEW qualifiers to access the old and new values of the affected row(s). Here's an example:

```plsql
CREATE OR REPLACE TRIGGER trigger_name
BEFORE INSERT ON table_name
FOR EACH ROW
BEGIN
  IF :NEW.column_name IS NULL THEN
    :NEW.column_name := 'Default Value';
  END IF;
END;
```

In this example, the trigger is fired before an INSERT operation on the specified table. If the value of the "column_name" in the new row is NULL, it is assigned a default value.

4. Using Triggers:
Once a trigger is created, it is automatically activated when the specified event occurs on the associated table or view. You don't need to explicitly call or invoke the trigger. It runs as part of the database operation. Here's an example of using a trigger:

```plsql
INSERT INTO table_name (column1, column2) VALUES ('Value1', 'Value2');
```

In this example, when an INSERT operation is performed on the specified table, the trigger associated with that event and table fires and executes its code.

Database triggers are powerful tools for automating actions and enforcing business rules within the database. They allow you to react to data changes and perform custom logic before or after the changes occur. It's important to carefully design and test triggers to ensure they meet your requirements and don't introduce any unintended side effects.