|  |  |
| --- | --- |
|  | **Recursive functions examples** |
| 1 | **Finding the managers - employee org chart.** |
|  | use hr2;  with recursive manager as (  select employee\_id, manager\_id  from employees  where employee\_id = 110  UNION all  select e.employee\_id,e.manager\_id  from employees e  inner join manager m  on e.employee\_id = m.manager\_id  )  select employee\_id,manager\_id from manager |
| 2 | **Counting numbers** |
|  | WITH RECURSIVE counting\_numbers AS (  SELECT 5 AS number\_i  UNION ALL  SELECT number\_i-1 FROM counting\_numbers WHERE number\_i >0  )  SELECT number\_i FROM counting\_numbers; |
| 3 | **Powers of 2** |
|  | WITH RECURSIVE powers\_of\_2 AS (  SELECT 1 AS power  UNION ALL  SELECT power \* 2 FROM powers\_of\_2 WHERE power < 16  )  SELECT power FROM powers\_of\_2; |
| 4 | **Fibonacci series** |
| A | **First approach** - WITH RECURSIVE fibonacci AS (  SELECT 0 AS n, 1 AS next  UNION ALL  SELECT next, n + next FROM fibonacci WHERE n < 6  )  SELECT n FROM fibonacci; |
| B | **Second approach** –  WITH RECURSIVE Fibonacci (n, fib\_n\_minus\_2, fib\_n\_minus\_1, fib\_n) AS (  -- Base case: The first two numbers in the Fibonacci sequence  SELECT 1, 0, 1, 1 -- n, fib\_n\_minus\_2, fib\_n\_minus\_1, fib\_n  UNION ALL  -- Recursive case: Calculate the next Fibonacci number  SELECT n + 1, fib\_n\_minus\_1, fib\_n, fib\_n\_minus\_1 + fib\_n  FROM Fibonacci  WHERE n < 10 -- Generate the first 10 Fibonacci numbers (you can adjust this value)  )  SELECT n, fib\_n  FROM Fibonacci; |
|  | Key differences in both the approaches ( both are correct , I prefer the first method as it is easier to understand  1. Structure of Base Case and Recursive Case:   * In first version:   + The WITH RECURSIVE starts with a base case that only includes two columns: n and next.   + The base case initializes n = 0 and next = 1, which represents the first two Fibonacci numbers.   + The recursive case produces the next Fibonacci number by swapping n and next values and adding them (n + next) to generate the next Fibonacci number. * In the second version:   + The WITH RECURSIVE includes four columns: n, fib\_n\_minus\_2, fib\_n\_minus\_1, and fib\_n.   + The base case explicitly represents the first Fibonacci numbers (fib\_n\_minus\_2 = 0 and fib\_n\_minus\_1 = 1).   + The recursive case generates the next Fibonacci number by adding the last two numbers (fib\_n\_minus\_2 + fib\_n\_minus\_1).   2. Columns Used:   * In first version:   + Only two columns (n and next) are used to compute the Fibonacci numbers.   + The result selects just the n column from the output (which represents the Fibonacci number itself). * In the second version:   + Four columns are used to maintain both the current Fibonacci number (fib\_n) and the previous two numbers (fib\_n\_minus\_2 and fib\_n\_minus\_1) to generate the sequence.   + The result selects both n (the index) and fib\_n (the Fibonacci number).   3. Recursive Logic:   * In the first version:   + The recursive part swaps the n and next values and computes the new next as n + next.   + It's a more minimalistic approach, relying only on these two columns and performing the addition in the recursive step itself.   + This approach effectively captures the Fibonacci recurrence relation using fewer columns. * In the second version:   + The recursive logic is more explicit, where the Fibonacci sequence is directly calculated using the previous two values (fib\_n\_minus\_2 and fib\_n\_minus\_1).   + It’s more verbose but makes the flow of how each Fibonacci number is computed clearer.   4. Stopping Condition:   * In the first version version:   + The recursion stops when n < 6, meaning it generates Fibonacci numbers up to n = 6 (which results in the first 7 Fibonacci numbers). * In the second version:   + The stopping condition is based on the value of n reaching 10 (or whatever value is set), generating the first 10 Fibonacci numbers. |
| 5 | **Family tree example ( first to last)** |
|  | --- create table  CREATE TABLE CUSTOMER\_HOUSEHOLD  ( Cust\_Id INT, NAME VARCHAR(20), PARENT\_NAME  VARCHAR(20));  INSERT INTO CUSTOMER\_HOUSEHOLD VALUES  (123000, 'GEFF', NULL ),  (123001, 'MARK', 'GEFF' ),  (123002, 'CHARLIE' , 'MARK' ) ,  (123003, 'CRISTY' , 'MARK' ) ,  (123004, 'SARAH', 'GEFF' ),  (123005, 'ROBERT' , 'SARAH') ,  (123006, 'ANDY' , 'SARAH') ;  select \* from CUSTOMER\_HOUSEHOLD;  --- generate family tree starting from first to last child  WITH RECURSIVE Family ( NAME, PARENT\_NAME, Hierarchy) AS  (SELECT Name, Parent\_name, CAST(Name AS CHAR(200))  FROM CUSTOMER\_HOUSEHOLD  WHERE Parent\_Name IS NULL  UNION ALL  SELECT ch.Name, ch.Parent\_Name, CONCAT(cf.Hierarchy, ",", ch.name)  FROM Family cf  JOIN CUSTOMER\_HOUSEHOLD ch  ON cf.name = ch.parent\_name )  SELECT \* FROM Family ORDER BY Hierarchy; |
| 6 | **Factorial of a number** |
| A | **Approach 1 –**  WITH RECURSIVE Factorial (n, prev\_fact, fact) AS (  -- Base case: n = 1, factorial(1) = 1  SELECT 1, 1, 1  UNION ALL  -- Recursive case: fact = prev\_fact \* n  SELECT n + 1, fact, fact \* (n + 1)  FROM Factorial  WHERE n < 6 -- Change this to any number to find the factorial up to that number  )  SELECT n, fact  FROM Factorial; |
| B | WITH RECURSIVE Factorial AS (  -- Base case: n = 1, factorial(1) = 1  SELECT 1 AS n, 1 AS fact  UNION ALL  -- Recursive case: fact = fact \* (n + 1)  SELECT n + 1, fact \* (n + 1)  FROM Factorial  WHERE n < 6 -- Change this to any number to find the factorial up to that number  )  SELECT n, fact  FROM Factorial; |
| 7 | **Question - Write a query to find all the subcategories for “Electronics” for the below data** |
|  | Data - CREATE TABLE Categories (  CategoryID INT PRIMARY KEY,  CategoryName VARCHAR(100) NOT NULL,  ParentCategoryID INT NULL,  FOREIGN KEY (ParentCategoryID) REFERENCES Categories(CategoryID)  );  INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES  (1, 'Electronics', NULL), -- Top-level category  (2, 'Computers', 1), -- Subcategory of Electronics  (3, 'Laptops', 2), -- Subcategory of Computers  (4, 'Desktops', 2), -- Subcategory of Computers  (5, 'Smartphones', 1); -- Subcategory of Electronics |
| 8 | **Bill of Materials**  **In manufacturing, a Bill of Materials (BOM) lists the components and sub-components required to build a product. Recursion can help in traversing the BOM to determine all parts needed for assembly.**  CREATE TABLE BOM (  PartID INT PRIMARY KEY,  PartName VARCHAR(100) NOT NULL,  ParentPartID INT NULL,  FOREIGN KEY (ParentPartID) REFERENCES BOM(PartID)  );  INSERT INTO BOM (PartID, PartName, ParentPartID) VALUES  (1, 'Bicycle', NULL), -- Final product  (2, 'Frame', 1), -- Part of Bicycle  (3, 'Wheel', 1), -- Part of Bicycle  (4, 'Spoke', 3), -- Part of Wheel  (5, 'Hub', 3); -- Part of Wheel  **Write a query to retrieve all components required to build a Bicycle using recursion** |
|  | **Case statements in sql** |
|  | **Table 1: Orders**  CREATE TABLE Orders (  OrderID INT PRIMARY KEY,  CustomerName VARCHAR(100),  OrderDate DATE,  Status VARCHAR(20)  );  INSERT INTO Orders (OrderID, CustomerName, OrderDate, Status) VALUES  (1, 'Rajesh Kumar', '2024-09-15', 'Pending'),  (2, 'Asha Patel', '2024-09-17', 'Shipped'),  (3, 'Vikram Singh', '2024-09-18', 'Cancelled'),  (4, 'Priya Sharma', '2024-09-20', 'Delivered'),  (5, 'Sunita Rao', '2024-09-22', 'Pending');  **Table 2: Products**  CREATE TABLE Products (  ProductID INT PRIMARY KEY,  ProductName VARCHAR(100),  Price DECIMAL(10, 2),  Category VARCHAR(50)  );  INSERT INTO Products (ProductID, ProductName, Price, Category) VALUES  (1, 'Laptop', 50000.00, 'Electronics'),  (2, 'Mobile Phone', 20000.00, 'Electronics'),  (3, 'Chair', 2500.00, 'Furniture'),  (4, 'Table', 5000.00, 'Furniture'),  (5, 'Washing Machine', 15000.00, 'Appliances');  **Table 3: Payments**  CREATE TABLE Payments (  PaymentID INT PRIMARY KEY,  OrderID INT,  PaymentMethod VARCHAR(50),  Amount DECIMAL(10, 2),  PaymentStatus VARCHAR(20),  FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)  );  INSERT INTO Payments (PaymentID, OrderID, PaymentMethod, Amount, PaymentStatus) VALUES  (1, 1, 'Credit Card', 50000.00, 'Completed'),  (2, 2, 'Debit Card', 20000.00, 'Completed'),  (3, 3, 'Cash', 0.00, 'Refunded'),  (4, 4, 'Credit Card', 15000.00, 'Completed'),  (5, 5, 'UPI', 0.00, 'Pending'); |
| 1 | **Scenario 1: Categorize Order Statuses**  **Use Case:**  You want to categorize orders based on their status (Pending, Shipped, Cancelled, Delivered) into different types like "Open", "Closed", and "Cancelled" using a CASE statement. |
| 2 | **Scenario 2: Categorize Products Based on Price Range**  **Use Case:**  You want to classify products into different price categories such as "Budget", "Mid-range", and "Premium" using a CASE statement. |
| 3 | **Scenario 3: Calculate Payment Status**  **Use Case:**  You want to display the payment status of each order, adding logic that considers the amount and payment status. If the payment status is "Pending", show "Incomplete"; if the amount is 0.00, show "No Payment Made"; otherwise, show "Payment Complete". |
|  | **Correlated queries – use exists and non exists for these** |
| 1 | **Find Customers with Pending Payments Exceeding a Specific Amount**  ***Question:*** Retrieve the names of customers who have at least one order with a pending payment (PaymentStatus = 'Pending') where the pending amount is greater than ₹10,000. |
| 2 | **Identify Orders Without Any Completed Payments**  *Question:* Find all orders that do not have any associated payments with a status of 'Completed'. |
| 3 | **Retrieve Customers Who Have Ordered All Available Product Categories**  *Question:* List the names of customers who have placed orders that include products from every category available in the Products table**.** |
| 4 | **List Products That Have Never Been Ordered**  *Question:* Identify all products that have never been included in any order. |
|  | **Use HR schema to answer following questions ( includes all scenarios)** |
| 1 | **Find employees who earn a salary greater than all salaries in the IT department** |
| 2 | **Find employees in the HR department with a salary greater than the average salary of all employees** |
|  | **-- Create dependents table**  CREATE TABLE dependents (  dependent\_id INT PRIMARY KEY,  employee\_id INT,  first\_name VARCHAR(50),  last\_name VARCHAR(50),  relationship VARCHAR(50)  );  INSERT INTO dependents (dependent\_id, employee\_id, first\_name, last\_name, relationship)  VALUES  (1, 101, 'John', 'Doe', 'Child'),  (2, 102, 'Jane', 'Smith', 'Spouse'),  (3, 103, 'Bob', 'Johnson', 'Child')  ; |
| 3 | **Find employees who have at least one dependent**   * + - 1. **Using exists**       2. **Using joins and aggregation** |
|  | -- Create training\_history table  CREATE TABLE training\_history (  training\_id INT PRIMARY KEY,  employee\_id INT,  training\_date DATE,  training\_title VARCHAR(100),  trainer\_name VARCHAR(100)  -- CONSTRAINT fk\_training\_history\_employee FOREIGN KEY (employee\_id) REFERENCES employees(employee\_id)  );  desc employees;  -- Insert sample data into training\_history table  INSERT INTO training\_history (training\_id, employee\_id, training\_date, training\_title, trainer\_name)  VALUES  (1, 101, '2023-01-15', 'SQL Basics', 'John Trainer'),  (2, 102, '2023-02-20', 'Advanced Excel', 'Jane Trainer'),  (3, 103, '2023-03-10', 'Presentation Skills', 'Bob Trainer')  ;  -- Check the training\_history table  SELECT \* FROM training\_history; |
| 4 | **Find employees who have attended any training sessions**   * + - 1. **Using exists**       2. **Using joins and aggregation** |
| 5 | **Find employees whose salary is above the average salary in their department.** |
| 6 | **Find employees whose job title matches that of their manager** |
| 7 | **Find employees who have a higher commission percentage than the average commission percentage in their department** |