SQL Project Documentation

By: Bassel Elbahnasy

Table of Contents:

1st Query

Retrieve the list of all products with their respective categories and suppliers.

```
USE Northwind
-- 1-Retrieve the list of all products with their respective categories and suppliers.
SELECT p.ProductName, c.CategoryName, s.CompanyName AS SupplierName
FROM Products AS p
JOIN Categories AS c ON p.CategoryID = c.CategoryID
JOIN Suppliers AS s ON p.SupplierID = s.SupplierID;
```

Explanation:

This query joins the Products table with Categories and Suppliers tables to retrieve product names along with their corresponding category names and supplier company names.

SELECT p.ProductName, c.CategoryName, s.CompanyName AS SupplierName: This part specifies the columns to be retrieved: the product name, category name, and supplier name (aliased as SupplierName).

FROM Products AS p: This indicates the main table, Products, and assigns it the alias p.

JOIN Categories AS c ON p.CategoryID = c.CategoryID: This joins the Categories table (aliased as c) to the Products table based on the CategoryID column.

JOIN Suppliers AS s ON p.SupplierID = s.SupplierID: This joins the Suppliers table (aliased as s) to the Products table based on the SupplierID column.

2nd Query

List all orders along with customer and employee details.

```
USE Northwind
-- 2-List all orders along with customer and employee details.
SELECT o.OrderID, c.CompanyName , c.ContactName,
       e.FirstName + ' ' + e.LastName AS EmployeeName, e.EmployeeID
FROM Orders AS o
JOIN Customers AS c ON o.CustomerID = c.CustomerID
JOIN Employees AS e ON o.EmployeeID = e.EmployeeID;
```

Explanation:

This query joins the Orders table with Customers and Employees tables to list all orders with customer contact names, company names and employee full names.

SELECT:
o.OrderID: The ID of the order.
c.CompanyName: The name of the company that placed the order.
c.ContactName: The contact person at the company.
e.FirstName + ' ' + **e.LastName** AS **EmployeeName**: The full name of the employee handling the order, concatenated from their first and last names.
e.EmployeeID: The ID of the employee.
FROM:
Orders (aliased as **o**).
JOIN:
JOIN **Customers** AS **c** ON **o.CustomerID** = **c.CustomerID**: Links the Orders table with the Customers table based on the **CustomerID.**

JOIN **Employees** AS **e** ON **o.EmployeeID** = **e.EmployeeID**: Links the Orders table with the Employees table based on the **EmployeeID**.

3rd Query

List the total number of orders for each customer, sorted by the number of orders in descending order. Include only customers with more than 10 orders.

```
USE Northwind
--3-List the total number of orders for each customer,

SELECT c.CompanyName, COUNT(o.OrderID) AS TotalOrders
FROM Customers AS c
JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.CompanyName
HAVING COUNT(o.OrderID) > 10
ORDER BY TotalOrders DESC;
```

Explanation:

This query counts the number of orders for each customer, filters for those with more than 10 orders using HAVING, and sorts the results in descending order.

SELECT:
c.CompanyName: The name of the company.
COUNT(o.OrderID) AS TotalOrders: The total number of orders placed by the company.
FROM Clause: Indicates the primary table to query from, which is Customers (aliased as c).
JOIN Clause: Combines data from the Orders table (aliased as o) with the Customers table based on the CustomerID.
GROUP BY Clause: Groups the results by c.CompanyName to aggregate the order counts for each company.
HAVING Clause: Filters the groups to include only those companies with more than 10 orders.
ORDER BY Clause: Sorts the results in descending order based on the total number of orders (TotalOrders).

4th Query

Create a new table to store customer reviews with appropriate data types. Insert sample data into this table.

```sql
USE Northwind
--4-Create a new table to store customer reviews with appropriate data types. Insert sample data into this table.
CREATE TABLE CustomerReviews (
    CustomerID NCHAR(5) FOREIGN KEY REFERENCES Customers(CustomerID),
    ProductID INT FOREIGN KEY REFERENCES Products(ProductID),
    Rating INT CHECK (Rating BETWEEN 1 AND 5),
    Comments NVARCHAR(MAX),
);

INSERT INTO CustomerReviews (CustomerID, ProductID, Rating, Comments)
VALUES
('ALFKI', 1, 4, 'Great product!'),
('ANATR', 2, 5, 'Excellent quality and fast delivery.'),
('ANTON', 3, 3, 'Good product but a bit pricey.');

select * from CustomerReviews
```

Explanation:

This creates a new table for customer reviews having 4 columns (CustomerID, ProductID, Rating, Comments) then inserts some data.

```
CREATE TABLE CustomerReviews (
  CustomerID NCHAR(5) FOREIGN KEY REFERENCES Customers(CustomerID),
  ProductID INT FOREIGN KEY REFERENCES Products(ProductID),
  Rating INT CHECK (Rating BETWEEN 1 AND 5),
  Comments NVARCHAR(MAX),
);
```

This code creates a new table named CustomerReviews with the following columns:

- **CustomerID**: A 5-character code that references the CustomerID in the Customers table (ensuring data integrity).
- **ProductID**: An integer value that references the ProductID in the Products table (ensuring data integrity).
- **Rating**: An integer value between 1 and 5, representing the customer's rating of the product.
- **Comments**: Textual comments from the customer about the product.

```
INSERT INTO CustomerReviews (CustomerID, ProductID, Rating, Comments)
VALUES
('ALFKI', 1, 4, 'Great product!'),
('ANATR', 2, 5, 'Excellent quality and fast delivery.'),
('ANTON', 3, 3, 'Good product but a bit pricey.');
```

This code inserts three rows of data into the CustomerReviews table, providing sample customer reviews for different products.

5th Query

Update the unit price of all products in the 'Beverages' category by 10%.

```
UPDATE Products
SET UnitPrice = UnitPrice * 1.10
WHERE CategoryID = (SELECT CategoryID FROM Categories WHERE CategoryName = 'Beverages');

select * from Products where CategoryID = 1
```

Explanation:

- **UPDATE Products**: Specifies that the Products table will be modified.
- **SET UnitPrice = UnitPrice * 1.10**: Increases the UnitPrice of each product by 10% (multiplying it by 1.10).
- **WHERE CategoryID = (SELECT CategoryID FROM Categories WHERE CategoryName = 'Beverages')**: This condition applies to the price increase only to products that belong to the 'Beverages' category. It does this by finding the CategoryID for 'Beverages' in the Categories table and using that value to filter the products in the Products table.

This query increases the unit price of all products in the 'Beverages' category by 10%.

6th Query

Delete all orders placed before January 1, 1997.

```
DELETE FROM [Order Details]
WHERE OrderID IN (SELECT OrderID FROM Orders WHERE OrderDate < '1997-01-01');

DELETE FROM Orders
WHERE OrderDate < '1997-01-01';

--show results
Select OrderID from Orders where OrderDate < '1997-01-01'
```

Explanation:

This query first deletes the related records in the Order Details table, then deletes the orders from the Orders table. This two-step process is necessary due to foreign key constraints.

7th Query

Calculate the average unit price of products in each category.

```sql
SELECT c.CategoryName, AVG(p.UnitPrice) AS AveragePrice
FROM Products AS p
JOIN Categories AS c ON p.CategoryID = c.CategoryID
GROUP BY c.CategoryName;
```

Explanation:

This query calculates the average price of products for each category. It joins the Products table with the Categories table on the CategoryID field and then groups the results by the CategoryName. The AVG function is used to find the average unit price of products within each category.

8th Query

Format the order date to 'DD-MM-YYYY' format for all orders.

```sql
SELECT OrderID,
       CONVERT(VARCHAR(10), OrderDate, 105) AS FormattedOrderDate
FROM Orders;
```

Explanation:

This query retrieves the OrderID and the order date from the Orders table. It uses the CONVERT function to format the OrderDate as a string in the format DD-MM-YYYY (style 105). The formatted date is given an alias FormattedOrderDate.

9th Query

Find the total sales amount (Quantity * Unit Price) for each employee, grouped by employee.

```sql
SELECT e.EmployeeID, e.FirstName + ' ' + e.LastName AS EmployeeName,
       SUM(od.Quantity * od.UnitPrice) AS TotalSalesAmount
FROM Employees AS e
JOIN Orders AS o ON e.EmployeeID = o.EmployeeID
JOIN [Order Details] AS od ON o.OrderID = od.OrderID
GROUP BY e.EmployeeID, e.FirstName, e.LastName
ORDER BY TotalSalesAmount;
```

Explanation:

This query calculates the total sales amount for each employee. It joins the Employees table with the Orders table on EmployeeID, and then joins with the Order Details table on OrderID. It sums the product of Quantity and UnitPrice from the Order Details table to get the total sales amount for each employee. The results are grouped by EmployeeID, FirstName, and LastName, and are ordered by TotalSalesAmount in ascending order.

10th Query

List products that have never been ordered.

```
SELECT p.ProductID, p.ProductName
FROM Products AS p
LEFT JOIN [Order Details] AS od ON p.ProductID = od.ProductID
WHERE od.OrderID IS NULL;
```

Explanation:

This query retrieves the ProductID and ProductName of products that have never been ordered. It performs a left join between the Products table and the [Order Details] table on ProductID, and filters the results to include only those products where OrderID is null, indicating no matching records in [Order Details].

11th Query

Create a Common Table Expression (CTE) that lists the top 5 products by sales amount.

```
WITH ProductSales AS (
    SELECT p.ProductID, p.ProductName,
           SUM(od.Quantity * od.UnitPrice) AS TotalSalesAmount,
           ROW_NUMBER() OVER (ORDER BY SUM(od.Quantity * od.UnitPrice) DESC) AS SalesRank
    FROM Products p
    JOIN [Order Details] od ON p.ProductID = od.ProductID
    GROUP BY p.ProductID, p.ProductName
)
SELECT ProductID, ProductName, TotalSalesAmount
FROM ProductSales
WHERE SalesRank <= 5;
```

Explanation:

This query calculates the top 5 products by total sales amount. It uses a Common Table Expression (CTE) named ProductSales to compute the total sales amount for each product and assigns a sales rank based on the descending order of the total sales. The outer query then selects the ProductID, ProductName, and TotalSalesAmount for the top 5 products based on their sales rank.

12th Query

Combine the list of customers and suppliers into a single list of companies.

```sql
SELECT CompanyName, 'Customer' AS CompanyType
FROM Customers
UNION ALL
SELECT CompanyName, 'Supplier' AS CompanyType
FROM Suppliers;
```

Explanation:

This query combines the names of companies from the Customers and Suppliers tables into a single result set, distinguishing them by a CompanyType column. The UNION ALL operator is used to include all rows from both tables, even if there are duplicates. Each company name is labeled as either 'Customer' or 'Supplier'.

13th Query

List the top 3 employees with the highest total sales.

```sql
SELECT TOP 3 e.EmployeeID, e.FirstName + ' ' + e.LastName AS EmployeeName,
       SUM(od.Quantity * od.UnitPrice) AS TotalSales
FROM Employees AS e
JOIN Orders AS o ON e.EmployeeID = o.EmployeeID
JOIN [Order Details] AS od ON o.OrderID = od.OrderID
GROUP BY e.EmployeeID, e.FirstName, e.LastName
ORDER BY TotalSales DESC;
```

Explanation:

This query retrieves the top 3 employees with the highest total sales. It joins the Employees, Orders, and [Order Details] tables and groups the results by employee ID, first name, and last name. The total sales are calculated as the sum of Quantity times UnitPrice for each employee, and the results are ordered in descending order of total sales, selecting only the top 3.

14th Query

Create a pivot table that shows the total sales amount for each category per year.

```sql
SELECT *
FROM (
    SELECT
        c.CategoryName,
        YEAR(o.OrderDate) AS OrderYear,
        od.Quantity * od.UnitPrice AS SalesAmount
    FROM Categories AS c
    JOIN Products AS p ON c.CategoryID = p.CategoryID
    JOIN [Order Details] AS od ON p.ProductID = od.ProductID
    JOIN Orders AS o ON od.OrderID = o.OrderID
) AS SourceTable
PIVOT (
    SUM(SalesAmount)
    FOR OrderYear IN ([1996], [1997], [1998])
) AS PivotTable;
```

Explanation:

This query calculates the sales amount for each category by year and displays it in a pivot table format. The inner query retrieves the category name, the year of the order, and the sales amount (calculated as Quantity * UnitPrice). The outer query uses the PIVOT operator to summarize the sales amount by category for each specified year (1996, 1997, 1998), creating a pivot table with years as columns and categories as rows.

15th Query

Write a stored procedure to retrieve the sales report for a given date range.

```sql
CREATE PROCEDURE GetSalesReport
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    SELECT
        o.OrderID,
        o.OrderDate,
        c.CompanyName AS CustomerName,
        p.ProductName,
        od.Quantity,
        od.UnitPrice,
        od.Quantity * od.UnitPrice AS TotalAmount
    FROM Orders AS o
    JOIN Customers AS c ON o.CustomerID = c.CustomerID
    JOIN [Order Details] AS od ON o.OrderID = od.OrderID
    JOIN Products AS p ON od.ProductID = p.ProductID
    WHERE o.OrderDate BETWEEN @StartDate AND @EndDate
    ORDER BY o.OrderDate;
END;



DECLARE @StartDate DATE = '1997-01-01';
DECLARE @EndDate DATE = '1998-01-01';

EXEC GetSalesReport @StartDate, @EndDate;
```

Explanation:

This code defines a stored procedure GetSalesReport that takes two date parameters (@StartDate and @EndDate). It retrieves sales data (including order ID, order date, customer name, product name, quantity, unit price, and total amount) for orders placed between the specified dates. The data is ordered by order date. The stored procedure is then executed with the start date of January 1, 1997, and the end date of January 1, 1998.

16th Query

Write a T-SQL script to generate a monthly sales report and store it in a new table.

```sql
CREATE TABLE MonthlySalesReport (
    ReportMonth DATE,
    TotalOrders INT,
    TotalSales DECIMAL(18,2)
);

-- Insert monthly sales data
INSERT INTO MonthlySalesReport (ReportMonth, TotalOrders, TotalSales)
SELECT
    DATEFROMPARTS(YEAR(o.OrderDate), MONTH(o.OrderDate), 1) AS ReportMonth,
    COUNT(DISTINCT o.OrderID) AS TotalOrders,
    SUM(od.Quantity * od.UnitPrice) AS TotalSales
FROM Orders o
JOIN [Order Details] od ON o.OrderID = od.OrderID
GROUP BY DATEFROMPARTS(YEAR(o.OrderDate), MONTH(o.OrderDate), 1)
ORDER BY ReportMonth;


SELECT * FROM MonthlySalesReport;
```

Explanation:

This code creates a table named MonthlySalesReport to store monthly sales data, including the report month, total number of orders, and total sales amount. It then inserts data into this table by selecting and grouping order data by month, calculating the total number of orders and total sales amount for each month. Finally, it selects and displays all data from the MonthlySalesReport table.

17th Query

Modify the stored procedure from task 15 to include error handling for invalid date ranges.

```sql
CREATE PROCEDURE GetSalesReport
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    BEGIN TRY
        IF @StartDate > @EndDate
            THROW 50000, 'Start date must be earlier than or equal to end date.', 1;
        SELECT
            o.OrderID,
            o.OrderDate,
            c.CompanyName AS CustomerName,
            p.ProductName,
            od.Quantity,
            od.UnitPrice,
            od.Quantity * od.UnitPrice AS TotalAmount
        FROM Orders AS o
        JOIN Customers AS c ON o.CustomerID = c.CustomerID
        JOIN [Order Details] AS od ON o.OrderID = od.OrderID
        JOIN Products AS p ON od.ProductID = p.ProductID
        WHERE o.OrderDate BETWEEN @StartDate AND @EndDate
        ORDER BY o.OrderDate;
    END TRY
    BEGIN CATCH
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
        DECLARE @ErrorState INT = ERROR_STATE();
        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
    END CATCH;
END;

Declare @StartDate Date = '01-01-1998';
Declare @EndDate Date = '01-01-1997';

exec GetSalesReport @startDate, @EndDate;
```

Explanation:

This code defines a stored procedure GetSalesReport that includes error handling. It takes two date parameters (@StartDate and @EndDate) and checks if the start date is later than the end date, throwing an error if true. If the dates are valid, it retrieves sales data for orders placed between the specified dates, ordered by order date. If an error occurs, the error message, severity, and state are captured and re-raised. The stored procedure is then executed with the start date of January 1, 1998, and the end date of January 1, 1997, which triggers the error handling logic.

18th Query

Retrieve the details of the five most recent orders, including order date and customer details.

```
SELECT TOP 5
    o.OrderID,
    o.OrderDate,
    c.CompanyName,
    c.ContactName,
    c.Country
FROM Orders AS o
JOIN Customers AS c ON o.CustomerID = c.CustomerID
ORDER BY o.OrderDate DESC;
```

Explanation:

This query retrieves the top 5 most recent orders, including order ID, order date, and the customer's company name, contact name, and country. The results are ordered by order date in descending order.

19th Query

Write a query to find all customers who have placed more than five orders in the last year.

```
SELECT c.CustomerID, c.CompanyName, COUNT(o.OrderID) AS OrderCount
FROM Customers AS c
JOIN Orders AS o ON c.CustomerID = o.CustomerID
WHERE o.OrderDate > YEAR(1998)
GROUP BY c.CustomerID, c.CompanyName
HAVING COUNT(o.OrderID) > 5;
```

Explanation:

This query counts the number of orders for each customer in the last year (1998) and filters for those with more than five orders.

20th Query

Write a query to find the average time (in days) between orders for each customer.

```sql
WITH OrderDates AS (
    SELECT
        CustomerID,
        OrderDate,
        LAG(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS PrevOrderDate
    FROM Orders
)
SELECT
    CustomerID,
    AVG(DATEDIFF(DAY, PrevOrderDate, OrderDate)) AS AvgDaysBetweenOrders
FROM OrderDates
WHERE PrevOrderDate IS NOT NULL
GROUP BY CustomerID;
```

Explanation:

This query calculates the average number of days between orders for each customer. It uses a Common Table Expression (CTE) to get the previous order date for each order and then computes the average difference in days between consecutive orders.

21st Query

Create a view that shows the total quantity of each product sold per year.

```sql
CREATE VIEW ProductSalesPerYear AS
SELECT
    p.ProductID,
    p.ProductName,
    YEAR(o.OrderDate) AS SalesYear,
    SUM(od.Quantity) AS TotalQuantity
FROM Products AS p
JOIN [Order Details] AS od ON p.ProductID = od.ProductID
JOIN Orders AS o ON od.OrderID = o.OrderID
GROUP BY p.ProductID, p.ProductName, YEAR(o.OrderDate);

--See the results
select * from ProductSalesPerYear
```

Explanation:

This query creates a view named ProductSalesPerYear that shows the total quantity of each product sold per year. The view includes product ID, product name, sales year, and total quantity sold. The second query retrieves data from this view.

22nd Query

Write a query to find the employees who have handled orders totaling more than $100,000 in sales.

```
SELECT
    e.EmployeeID,
    e.FirstName + ' ' + e.LastName AS EmployeeName,
    SUM(od.Quantity * od.UnitPrice) AS TotalSales
FROM Employees AS e
JOIN Orders AS o ON e.EmployeeID = o.EmployeeID
JOIN [Order Details] AS od ON o.OrderID = od.OrderID
GROUP BY e.EmployeeID, e.FirstName, e.LastName
HAVING SUM(od.Quantity * od.UnitPrice) > 100000
ORDER BY TotalSales DESC;
```

Explanation:

This query retrieves employees who have generated more than $100,000 in sales. It shows the employee ID, full name, and total sales amount, ordering the results by total sales in descending order.

23rd Query

Write a query to find the top three products with the highest total sales amount each month.

```
WITH MonthlySales AS (
    SELECT
        DATEFROMPARTS(YEAR(o.OrderDate), MONTH(o.OrderDate), 1) AS SalesMonth,
        p.ProductID,
        p.ProductName,
        SUM(od.Quantity * od.UnitPrice) AS TotalSales,
        ROW_NUMBER() OVER (PARTITION BY DATEFROMPARTS(YEAR(o.OrderDate), MONTH(o.OrderDate), 1)
                           ORDER BY SUM(od.Quantity * od.UnitPrice) DESC) AS SalesRank
    FROM Products AS p
    JOIN [Order Details] AS od ON p.ProductID = od.ProductID
    JOIN Orders AS o ON od.OrderID = o.OrderID
    GROUP BY DATEFROMPARTS(YEAR(o.OrderDate), MONTH(o.OrderDate), 1), p.ProductID, p.ProductName
)
SELECT SalesMonth, ProductID, ProductName, TotalSales
FROM MonthlySales
WHERE SalesRank <= 3
ORDER BY SalesMonth, SalesRank;
```

Explanation:

This query finds the top 3 products by total sales for each month. It uses a CTE to calculate monthly sales and assigns a sales rank to each product, then retrieves the top 3 products per month.

24th Query

Create a function that returns the total sales amount for a given product.

```
CREATE FUNCTION dbo.GetTotalSalesForProduct
(
    @ProductID INT
)
RETURNS DECIMAL(18,2)
AS
BEGIN
    DECLARE @TotalSales DECIMAL(18,2);

    SELECT @TotalSales = SUM(Quantity * UnitPrice)
    FROM [Order Details]
    WHERE ProductID = @ProductID;

    RETURN ISNULL(@TotalSales, 0);
END;

--See result
SELECT dbo.GetTotalSalesForProduct(5) AS TotalSales;
```

Explanation:

This query defines a user-defined function GetTotalSalesForProduct that calculates the total sales for a given product ID. It sums the quantity times unit price for the specified product and returns the result. The second query tests this function for product ID 5.

25th Query

Write a query to find all orders where the total order amount exceeds the average order amount.

```sql
WITH OrderTotals AS (
    SELECT
        OrderID,
        SUM(UnitPrice * Quantity) AS TotalAmount
    FROM [Order Details]
    GROUP BY OrderID
)
SELECT
    o.OrderID,
    o.OrderDate,
    o.CustomerID,
    ot.TotalAmount
FROM Orders o
JOIN OrderTotals ot ON o.OrderID = ot.OrderID
CROSS JOIN (
    SELECT AVG(TotalAmount) AS AvgOrderAmount
    FROM OrderTotals
) AS AvgOrder
WHERE ot.TotalAmount > AvgOrder.AvgOrderAmount
ORDER BY ot.TotalAmount DESC;
```

Explanation:

We use a Common Table Expression (CTE) named OrderTotals to calculate the total amount for each order, considering the unit price and quantity.
We then join this CTE with the Orders table to get additional order details.
We use a subquery to calculate the average order amount across all orders.
Finally, we filter for orders where the total amount exceeds this average.
The results are ordered by total amount in descending order.

26th Query

Create an index to optimize the query performance for retrieving order details based on order date.

```sql
CREATE NONCLUSTERED INDEX INDEX_Orders_OrderDate
ON Orders (OrderDate);
```

Explanation:

This creates a non-clustered index on the OrderDate column of the Orders table.
A non-clustered index is chosen because it allows for fast retrieval of data based on the OrderDate without requiring a complete reorganization of the table data.
This index will significantly improve the performance of queries that filter or sort by OrderDate.

27th Query

Write a query to find the percentage increase in total sales for each category from the previous year.

```sql
WITH YearlyCategorySales AS (
    SELECT
        c.CategoryID,
        c.CategoryName,
        YEAR(o.OrderDate) AS SalesYear,
        SUM(od.UnitPrice * od.Quantity) AS TotalSales
    FROM Categories c
    JOIN Products p ON c.CategoryID = p.CategoryID
    JOIN [Order Details] od ON p.ProductID = od.ProductID
    JOIN Orders o ON od.OrderID = o.OrderID
    GROUP BY c.CategoryID, c.CategoryName, YEAR(o.OrderDate)
)
SELECT
    y1.CategoryID,
    y1.CategoryName,
    y1.SalesYear,
    y1.TotalSales,
    y0.TotalSales AS PreviousYearSales,
    CASE
        WHEN y0.TotalSales IS NULL OR y0.TotalSales = 0 THEN NULL
        ELSE (y1.TotalSales - y0.TotalSales) / y0.TotalSales * 100
    END AS PercentageIncrease
FROM YearlyCategorySales y1
LEFT JOIN YearlyCategorySales y0 ON y1.CategoryID = y0.CategoryID
                                AND y1.SalesYear = y0.SalesYear + 1
ORDER BY y1.CategoryID, y1.SalesYear;
```

Explanation:

We start with a CTE named YearlyCategorySales that calculates the total sales for each category per year.
In the main query, we join this CTE with itself, offsetting the year by 1 to get the previous year's sales.
We calculate the percentage increase using the formula: (Current Year Sales - Previous Year Sales) / Previous Year Sales * 100.
We use a CASE statement to handle scenarios where there might be no sales in the previous year (to avoid division by zero).
The results are ordered by CategoryID and SalesYear for easy reading.