



SQL PRIMER



OBJECTIVES

- Relational Database Overview
- Creating Databases and Tables
- Insert, Update, Delete Data
- Retrieving Data
- Joining Multiple Tables
- Grouping and Summarizing Data
- Subqueries
- Views

SQL OVERVIEW

- Structured Query Language
- SQL standard
 - Database extensions

RELATIONAL BASICS AND JOINING TABLES

- Employees

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |

RELATIONSHIPS

- Employees and Departments

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |




| Dept_ID | Name |
|---------|-----------------|
| 10 | Human Resources |
| 20 | Sales |

RELATIONSHIPS

- Employees and Job Histories

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |



| Employee_ID | Position_ID | Start_Date | End_Date |
|-------------|-------------|------------|----------|
| 1005 | 2011 | 20180824 | 20200105 |
| 1005 | 2015 | 20200106 | NULL |

NORTHWIND SCHEMA



CREATE TABLES

- SQL Syntax

```
CREATE TABLE table_name  
(  
    column_name1 data_type(size),  
    column_name2 data_type(size),  
    column_name3 data_type(size),  
    ....  
);
```

- SQL - Example

```
CREATE TABLE Employees  
(  
    Name varchar(20),  
    Position varchar(25),  
    Salary int  
);
```


CONSTRAINTS

- Primary Key
- Foreign Key
- Unique

PRIMARY KEY CONSTRAINT

```
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT PRIMARY KEY,
    LastName varchar(50) NOT NULL,
    FirstName varchar(50),
    Address varchar(50),
    City varchar(50)
);
OR
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT NOT NULL,
    LastName varchar(50) NOT NULL,
    FirstName varchar(50),
    Address varchar(50),
    City varchar(50),
    CONSTRAINT pk_contact PRIMARY KEY (ContactID)
);
```

PRIMARY KEY CONSTRAINT WITH ALTER TABLE

```
ALTER TABLE Contact
```

```
ADD PRIMARY KEY (ContactID);
```

OR

```
ALTER TABLE Contact
```

```
ADD CONSTRAINT pk_contact PRIMARY KEY (ContactID);
```

TO DELETE

```
ALTER TABLE Contact DROP PRIMARY KEY;
```

FOREIGN KEY CONSTRAINT

```
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT PRIMARY KEY,
    LastName varchar(50) NOT NULL,
    FirstName varchar(50),
    Address varchar(50),
    City varchar(50),
    CountryID int,
    FOREIGN KEY (CountryID) REFERENCES Country(CountryID)
);
```

OR

```
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT NOT NULL,
    LastName varchar(50) NOT NULL,
    FirstName varchar(50),
    Address varchar(50),
    City varchar(50),
    CountryID int NOT NULL,
    CONSTRAINT fk_contact_country FOREIGN KEY (CountryID)
    REFERENCES Country(CountryID)
);
```

FOREIGN KEY CONSTRAINT WITH ALTER TABLE

```
ALTER TABLE Contact  
ADD FOREIGN KEY (CountryID)  
REFERENCES Country (CountryID);
```

OR

```
ALTER TABLE Contact  
ADD CONSTRAINT fk_country_contact  
FOREIGN KEY (CountryID)  
REFERENCES Country (CountryID);
```

UNIQUE CONSTRAINT

```
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT PRIMARY KEY,
    Name varchar(50) NOT NULL,
    EmailAddress varchar(50) UNIQUE NOT NULL,
    City varchar(50),
    CountryID int FOREIGN KEY REFERENCES Country(CountryID)
);
```

OR

```
CREATE TABLE Contact
(
    ContactID int AUTO_INCREMENT NOT NULL,
    Name varchar(50) NOT NULL,
    EmailAddress varchar(50),
    City varchar(50),
    CountryID int NOT NULL,
    CONSTRAINT uq_email UNIQUE (EmailAddress)
);
```

UNIQUE CONSTRAINT WITH ALTER TABLE

```
ALTER TABLE Contact  
ADD UNIQUE (EmailAddress) ;
```

OR

```
ALTER TABLE Contact  
ADD CONSTRAINT uq_Email  
UNIQUE (EmailAddress) ;
```

EXERCISE

- Using SQL Statements
 - Create a database CourseDB
 - In the CourseDB database create a table suppliers:
 - Primary key supplierid int
 - Other fields:
 - name varchar(40)
 - country varchar(40)
 - city varchar(40)
 - Create a table products:
 - Primary key productid int
 - Other fields:
 - name varchar(50)
 - price decimal(6,2)
 - supplierid int (foreign key references the suppliers table)

SQL CRUD

- Create/Insert data
- Retrieve data
- Update data
- Delete data

INSERTING A ROW OF DATA

It is possible to write the INSERT INTO statement in two forms.

The first form does not specify the column names where the data will be inserted, only their values:

```
INSERT [INTO] table_name  
VALUES (value1,value2,value3,...);
```

The second form specifies both the column names and the values to be inserted:

```
INSERT [INTO] table_name  
(column1,column2,column3,...)  
VALUES (value1,value2,value3,...);
```

INSERTING A ROW OF DATA EXAMPLE

- Must Adhere to Destination Constraints or the INSERT Transaction Fails
- Use a Column List to Specify Destination Columns
- Specify a Corresponding List of Values

```
USE northwind;
INSERT customers
    (customerid, companyname, contactname, contacttitle
    ,address, city, region, postalcode, country, phone
    ,fax)

VALUES ('PECOF', 'Pecos Coffee Company', 'Michael Dunn'
    , 'Owner', '1900 Oak Street', 'Vancouver', 'BC'
    , 'V3F 2K1', 'Canada', '(604) 555-3392'
    , '(604) 555-7293');
```

USING THE INSERT...SELECT STATEMENT

- All Rows That Satisfy the SELECT Statement Are Inserted
- Verify That the Table That Receives New Row Exists
- Ensure That Data Types Are Compatible
- Determine Whether Default Values Exist or Whether Null Values Are Allowed

```
USE northwind;  
INSERT Shippers (CompanyName, Phone)  
  SELECT CompanyName, Phone FROM Customers;
```

INSERTING DATA FROM A FILE (LOAD DATA INFILE)

```
USE Northwind;  
LOAD DATA INFILE 'data.txt' INTO TABLE tbl_name  
  FIELDS TERMINATED BY ','  
  LINES TERMINATED BY '\r\n'  
  IGNORE 1 LINES;
```

USING THE DELETE STATEMENT

- The DELETE statement removes one or more rows in a table according to the WHERE clause condition, if specified

```
USE northwind
DELETE FROM orders
WHERE YEAR(OrderDate) = 2018;
```

UPDATING DATA

- Updating Rows
- Syntax:

```
UPDATE table_name  
SET column1=value1, column2=value2, ...  
WHERE some_column=some_value;
```

UPDATING ROWS BASED ON DATA IN THE TABLE

- WHERE Clause Specifies Rows to Change
- SET Keyword Specifies the New Data
- Input values must have compatible data types with the columns
- Updates Do Not Occur in Rows That Violate Any Integrity Constraints

```
USE northwind;  
UPDATE products  
    SET unitprice = (unitprice * 1.1) WHERE  
    CategoryID = 5;
```


RETRIEVING DATA

- Retrieving Data by Using the SELECT Statement
- Filtering Data
- Formatting Result Sets

USING THE SELECT STATEMENT

- Select List Specifies the Columns
- FROM Clause Specifies the Table
- WHERE Clause Specifies the Condition Restricting the Query

Partial Syntax

```
SELECT [ALL | DISTINCT] <select_list>  
FROM {<table_source>} [,...n]  
WHERE <search_condition>;
```

SELECT STATEMENT

`SELECT column1 [, column2 ...] FROM tablename`

For example

`SELECT * FROM Employees;`

`SELECT First_Name, Last_Name FROM Employees;`

USING THE WHERE CLAUSE TO SPECIFY ROWS

```
USE northwind;  
SELECT CompanyName, ContactName, City,  
Country FROM  
Suppliers WHERE Country = 'France';
```

FILTERING DATA

- Using Comparison Operators
- Using String Comparisons
- Using Logical Operators
- Retrieving a Range of Values
- Using a List of Values as Search Criteria
- Retrieving Unknown Values

COMPARISON OPERATORS

| Operator | Description |
|----------|-----------------------|
| = | Equal |
| <> Or != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

USING COMPARISON OPERATORS

```
USE Northwind;  
SELECT ProductName, UnitPrice FROM  
Products WHERE UnitPrice >= 20;  
  
SELECT OrderID, OrderDate, CustomerID  
FROM Orders WHERE  
OrderDate <= '1996-12-31';
```

USING STRING COMPARISONS

```
USE northwind;  
SELECT CompanyName, ContactName, ContactTitle,  
City, Country  
FROM Suppliers  
WHERE ContactTitle LIKE '%Marketing%';
```

Use the LIKE operator and wild Cards:

- % for zero or more characters
- _ for a single character

USING LOGICAL OPERATORS AND, OR AND NOT

```
USE northwind;  
SELECT ProductName, UnitPrice, CategoryID  
FROM Products  
WHERE CategoryID = 1 AND UnitPrice >= 50;
```

```
USE northwind;  
SELECT ProductName, UnitPrice, CategoryID  
FROM Products  
WHERE CategoryID = 1 OR UnitPrice >= 50;
```

```
USE northwind;  
SELECT ProductName, UnitPrice, CategoryID  
FROM Products  
WHERE CategoryID = 1 AND NOT UnitPrice >= 50;
```

RETRIEVING A RANGE OF VALUES USING BETWEEN

```
USE northwind;  
SELECT ProductName, UnitPrice  
FROM Products  
WHERE unitprice BETWEEN 10 AND 20;
```

USING A LIST OF VALUES AS SEARCH CRITERIA AND THE IN OPERATOR

```
USE northwind;  
SELECT CompanyName, City, Country  
FROM Customers  
WHERE Country IN ('Germany', 'France');
```

RETRIEVING UNKNOWN VALUES USING IS NULL

```
USE northwind;  
SELECT CompanyName, Region  
FROM suppliers  
WHERE Region IS NULL;
```

FORMATTING RESULT SETS

- Sorting Data
- Eliminating Duplicate Rows
- Changing Column Names
- Case Expressions

SORTING DATA USING ORDER BY

```
USE northwind;  
SELECT ProductName, CategoryId, UnitPrice  
FROM Products  
ORDER BY CategoryId, UnitPrice DESC;
```

ASC is the default and can be omitted

ELIMINATING DUPLICATE ROWS USING DISTINCT

```
USE northwind;  
SELECT DISTINCT country  
FROM suppliers  
ORDER BY country;
```

CHANGING COLUMN NAMES

Both options shown here can be used (AS or omit AS)

```
USE northwind;  
SELECT CompanyName AS Supplier,  
ContactName AS Contact  
FROM Suppliers;
```

```
USE northwind;  
SELECT CompanyName Supplier,  
ContactName Contact  
FROM Suppliers;
```


MYSQL SIMPLE CASE EXPRESSION

CASE value

WHEN value1 THEN result1

WHEN value2 THEN result2

...

[ELSE else_result]

END

```
SELECT CategoryName,  
       CASE CategoryName  
         WHEN 'Beverages' THEN 'Drinks'  
         WHEN 'Condiments' THEN 'Seasonings'  
         WHEN 'Confections' THEN 'Sweets'  
         ELSE 'Something Else'  
       END Comment  
FROM Categories;
```

MYSQL SEARCHED CASE EXPRESSION

CASE

WHEN expression1 THEN result1

WHEN expression2 THEN result2

...

[ELSE else_result]

END

```
SELECT ProductName, UnitPrice,  
CASE  
    WHEN UnitPrice > 40 THEN 'Expensive'  
    WHEN UnitPrice > 20 THEN 'Reasonable'  
    ELSE 'Cheap' END Comment  
FROM Products;
```

RELATIONAL BASICS AND JOINING TABLES

- Employees

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |

RELATIONSHIPS

- Employees and Departments

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |

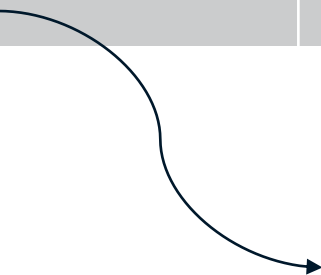


| Dept_ID | Name |
|---------|-----------------|
| 10 | Human Resources |
| 20 | Sales |

RELATIONSHIPS

- Employees and JobHistories

| Employee_ID | First_Name | Last_Name | Dept_ID | Location_ID |
|-------------|------------|-----------|---------|-------------|
| 1001 | John | Jones | 10 | 100 |
| 1002 | Susan | Smith | 20 | 100 |
| 1003 | Jackson | Black | 10 | 200 |
| 1004 | Thom | Thomas | 20 | 300 |
| 1005 | Robert | Reid | 10 | 400 |



| Employee_ID | Position_ID | Start_Date | End_Date |
|-------------|-------------|------------|----------|
| 1005 | 2011 | 20180824 | 20200105 |
| 1005 | 2015 | 20200106 | NULL |

JOINING TABLES

```
SELECT First_Name, Last_Name, Name  
FROM Employees JOIN Departments  
ON Employees.Dept_ID = Departments.Dept_ID;
```

```
SELECT First_Name, Last_Name, Name  
FROM Employees LEFT JOIN Departments  
ON Employees.Dept_ID = Departments.Dept_ID;
```

OVERVIEW

- Using Aliases for Table Names
- Combining Data from Multiple Tables
- Combining Multiple Result Sets

USING ALIASES FOR TABLE NAMES

Example 1 (without an alias name)

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories INNER JOIN Products  
ON Categories.CategoryID = Products.CategoryID;
```

Example 2 (with an alias name)

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c INNER JOIN Products AS p  
ON c.CategoryID = p.CategoryID;
```


COMBINING DATA FROM MULTIPLE TABLES

- Introduction to Joins
- Using Inner Joins
- Using Outer Joins
- Using Cross Joins
- Joining More Than Two Tables
- Joining a Table to Itself

INTRODUCTION TO JOINS

- Selects Specific Columns from Multiple Tables
 - JOIN keyword specifies that tables are joined and how to join them
 - ON keyword specifies join condition
- Queries Two or More Tables to Produce a Result Set
 - Use primary and foreign keys as join conditions
 - Use columns common to specified tables to join tables

USING INNER JOINS WITH JOIN

With inner joins rows are returned from the joined tables where there is a match on the join field(s)

Syntax:

```
SELECT columns  
FROM tbl1  
[INNER] JOIN tbl2  
ON tbl1.column = tbl2.column;
```

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c INNER JOIN Products AS p  
ON c.CategoryID = p.CategoryID;
```

USING INNER JOINS WITH THE WHERE CLAUSE

With inner joins rows are returned from the joined tables where there is a match on the join field(s)

Syntax:

```
SELECT columns  
FROM tbl1, tbl2  
WHERE tbl1.column = tbl2.column;
```

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c, Products AS p  
WHERE c.CategoryID = p.CategoryID;
```

USING LEFT OUTER JOINS

With left outer joins all rows are returned from the table on the left side of the join operator and only rows on the right side where there is a match on the join field(s)

Syntax:

```
SELECT columns  
FROM tbl1  
LEFT [OUTER] JOIN tbl2  
ON tbl1.column = tbl2.column;
```

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c LEFT OUTER JOIN  
Products AS p  
ON c.CategoryID = p.CategoryID;
```

USING RIGHT OUTER JOINS

With right outer joins all rows are returned from the table on the right side of the join operator and only rows on the left side where there is a match on the join field(s)

Syntax:

```
SELECT columns  
FROM tbl1  
RIGHT [OUTER] JOIN tbl2  
ON tbl1.column = tbl2.column;
```

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c RIGHT OUTER JOIN  
Products AS p  
ON c.CategoryID = p.CategoryID;
```

CROSS JOINS

A cross join is a cartesian product, the number of rows returned is the tbl1 row count times the tbl2 row count.

Syntax:

```
SELECT columns  
FROM tbl1  
CROSS JOIN tbl2;
```

```
USE Northwind;  
SELECT CategoryName, ProductName  
FROM Categories AS c  
CROSS JOIN Products AS p;
```

JOINING MORE THAN TWO TABLES - EXAMPLE

The result of any join can then be joined to another table

```
USE Northwind;  
SELECT CategoryName, ProductName, Quantity  
FROM Categories AS c  
JOIN Products AS p  
ON c.CategoryID = p.CategoryID  
JOIN Order_Details AS od  
ON p.ProductID = od.ProductID
```


EXERCISE - JOINS

Write a SQL Statement that displays a list showing the Category (CategoryName), Supplier (CompanyName), Product (ProductName) and Price (UnitPrice) of all products. To do this you will need to Join the Categories, Products and Suppliers tables

JOINING A TABLE TO ITSELF

Sometimes it is necessary to join a table to itself. The example here is typical, you need to show the manager of each employee in an employee table.

```
USE Northwind;  
SELECT e.LastName Employee, m.LastName  
Manager FROM  
Employees e INNER JOIN Employees m  
ON e.EmployeeID = m.ReportsTo;
```

COMBINING MULTIPLE RESULT SETS

- Use the UNION Operator to Create a Single Result Set from Multiple Queries
- Each Query Must Have:
 - Similar data types
 - Same number of columns
 - Same column order in select list

```
USE northwind;  
SELECT CONCAT(FirstName, ' ', LastName) AS name  
        ,City, PostalCode  
FROM employees  
UNION  
SELECT CompanyName, City, PostalCode  
FROM Customers;
```

GROUPING AND AGGREGATING DATA

- Listing the TOP n Values
- Using Aggregate Functions
- GROUP BY Fundamentals
- Generating Aggregate Values Within Result Sets

LISTING THE TOP *N* VALUES

- Lists Only the First *n* Rows of a Result Set
- Specify the Range of Values in the ORDER BY Clause

```
USE northwind;  
SELECT OrderID, ProductID, Quantity  
FROM Order_Details  
ORDER BY Quantity DESC LIMIT 5;
```

EXERCISE – TOP N

Write a SQL Statement that displays the 10 most expensive products, show the product name and its price

USEFUL AGGREGATE FUNCTIONS

AVG() - Returns the average value

COUNT() - Returns the number of rows

FIRST() - Returns the first value

LAST() - Returns the last value

MAX() - Returns the largest value

MIN() - Returns the smallest value

SUM() - Returns the sum

USING AGGREGATE FUNCTIONS WITH NULL VALUES

- Most Aggregate Functions Ignore Null Values
- COUNT(*) Function Counts Rows with Null Values

```
USE Northwind;  
SELECT COUNT(*)  
FROM Customers;
```

```
USE Northwind;  
SELECT COUNT(Region)  
FROM Customers;
```


GROUP BY FUNDAMENTALS

- Using the GROUP BY Clause
- Using the GROUP BY Clause with the HAVING Clause

USING THE GROUP BY CLAUSE

Aggregate functions are particularly useful when used with the **GROUP BY** clause. This statement lets us see the total sales for each product.

```
USE Northwind;  
SELECT ProductName, SUM(Quantity) AS Sales  
FROM Products p JOIN Order_Details od  
ON p.ProductID = od.ProductID  
GROUP BY ProductName;
```

EXERCISE – AGGREGATES

Write a SQL Statement that displays the revenue for each category. The revenue can be calculated using the expression $(\text{unitprice} * \text{quantity}) * (1 - \text{discount})$. You will need to join the Categories, Products and Order_Details tables to do this.

USING THE GROUP BY CLAUSE WITH THE HAVING CLAUSE

If you need to filter on an aggregate in a SELECT that uses GROUP BY you cannot use WHERE, you must use HAVING. The HAVING clause comes after GROUP BY. This statement only shows the total sales for products that have 1000 or more sales.

```
USE Northwind;
SELECT ProductName, SUM(Quantity) AS Sales
FROM Products p JOIN Order_Details od
ON p.ProductID = od.ProductID
GROUP BY ProductName
HAVING SUM(Quantity) >= 1000;
```

USING THE GROUP BY CLAUSE WITH THE ROLLUP OPERATOR

The ROLLUP Operator allows you to generate sub-totals and a grand total when there are more than one non-aggregate columns in the GROUP BY statement. Here we can generate category sub-totals for product sales.

```
USE Northwind;  
SELECT CategoryName, ProductName,  
SUM(Quantity) AS Sales  
FROM Products p JOIN Order_Details od  
ON p.ProductID = od.ProductID JOIN Categories c  
ON c.CategoryID = p.CategoryID  
GROUP BY CategoryName, ProductName  
WITH ROLLUP;
```

SUBQUERIES

- Introduction to Subqueries
- Using a Subquery as an Expression
- Using a Subquery to Correlate Data
- Using the EXISTS and NOT EXISTS Clauses

INTRODUCTION TO SUBQUERIES

- Why use Subqueries
 - To break down a complex query into a series of logical steps
 - To answer a query that relies on the results of another query
- How to Use Subqueries

USING A SUBQUERY AS AN EXPRESSION

- Is Evaluated and Treated as an Expression
- Is Executed Once for the Query
- Here Products with a UnitPrice less than the average UnitPrice are returned

```
SELECT ProductName, (SELECT AVG(UnitPrice) FROM Products)  
Average, UnitPrice Price FROM Products WHERE UnitPrice  
<=(SELECT AVG(UnitPrice) FROM Products);
```


EVALUATING A CORRELATED SUBQUERY

With a correlated subquery the inner query is evaluated for every row in the outer query. Here we show the highest order for each Category along with the Product Name.

```
SELECT CategoryName, ProductName, Quantity FROM
Categories c JOIN Products p
ON c.CategoryId = p.CategoryId
JOIN Order_Details od
ON p.ProductID = od.ProductID
WHERE Quantity =
  (SELECT MAX(Quantity) FROM Categories c1 JOIN
Products p1
ON c1.CategoryId = p1.CategoryId
JOIN Order_Details od1
ON p1.ProductID = od1.ProductID
WHERE c.CategoryID = c1.CategoryID);
```

EXERCISE - SUBQUERIES

Execute this SQL Statement

```
SELECT ProductName, DATE_FORMAT(OrderDate, "%D %b %Y")  
`Order Date`, Quantity FROM Products p JOIN  
Order_Details od ON p.ProductID = od.ProductID JOIN  
Orders o ON o.OrderID = od.OrderID ORDER BY  
ProductName, Quantity DESC;
```

Can you modify it so that we only see the row or rows for the best sale for each product

USING THE EXISTS AND NOT EXISTS CLAUSES

- Use with Correlated Subqueries
- Determine Whether Data Exists in a List of Values
- SQL Server Process
 - Outer query tests for the existence of rows
 - Inner query returns TRUE or FALSE
 - No data is produced
- Here Customers are listed that are in the same city as a Supplier

```
USE Northwind;  
SELECT CompanyName, Country, City FROM Customers c  
WHERE EXISTS  
(SELECT City FROM Suppliers s WHERE c.City = s.City);
```

INTRODUCTION TO VIEWS

- What Is a View?
- Types of Views
- Advantages of Views

WHAT IS A VIEW?

In SQL, a view is a virtual table based on the result of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

Views can be used to mask complexity, prepare data for reports and client applications. They can also be used as a security mechanism.

CREATING AND MANAGING VIEWS

- Syntax for Creating Views
- Syntax for Altering and Dropping Views

SYNTAX FOR CREATING VIEWS

- Use CREATEVIEW SQL statement:

```
CREATE VIEW view_name AS  
SELECT column_name(s)  
FROM table_name  
WHERE condition
```

SYNTAX FOR ALTERING AND DROPPING VIEWS

Change by using the ALTER VIEW Transact-SQL statement:

```
ALTER VIEW view_name AS  
SELECT column_name(s)  
FROM table_name  
WHERE condition;
```

Delete by using the DROPVIEW SQL statement:

```
DROP VIEW view_name
```


THANK YOU
