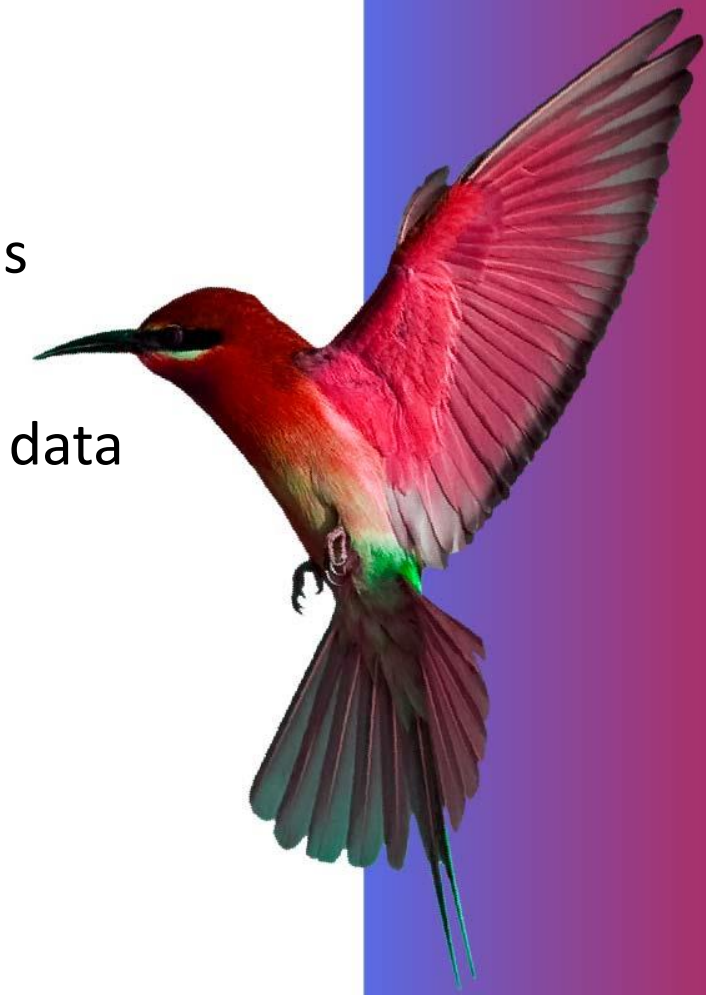
T-SQL Fundamentals

NashTech Viet Nam

09th Apr 2020

Agenda

- Module 1: Introduction to T-SQL
- Module 2: Query table with SELECT
- Module 3: Query multiple tables with JOINs
- Module 4: Using SET Operators
- Module 5: Using Function and aggregating data
- Module 6: Using Subqueries and APPLY
- Module 7: Using Table expression



Module 1: Introduction to T-SQL

Module Overview

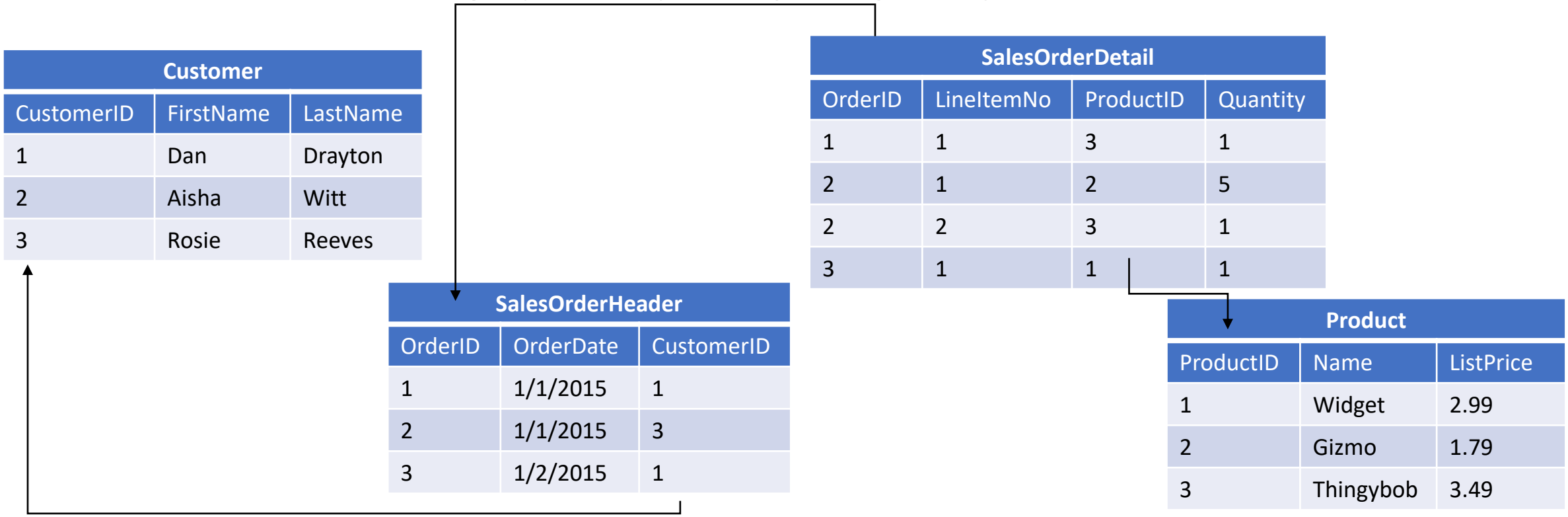
- What is Transact-SQL?
- Relational Databases
- Schemas and Object Names
- SQL Statement Types
- The SELECT Statement
- Working with Data Types
- Working with NULLs

What is Transact-SQL?

- Structured Query Language (SQL)
  - Developed by IBM in 1970s
  - Adopted as a standard by ANSI and ISO standards bodies
  - Widely used in industry
- Microsoft’s implementation is Transact-SQL
  - Referred to as T-SQL
  - Query language for SQL Server and Azure SQL Database
- SQL is declarative, not procedural
  - Describe what you want, don’t specify steps

Relational Databases

- Entities are represented as *relations* (tables), in which their attributes are represented as *domains* (columns)
- Most relational databases are *normalized*, with relationships defined between tables through *primary* and *foreign* keys



Customer

CustomerID	FirstName	LastName
1	Dan	Drayton
2	Aisha	Witt
3	Rosie	Reeves

SalesOrderHeader

OrderID	OrderDate	CustomerID
1	1/1/2015	1
2	1/1/2015	3
3	1/2/2015	1

SalesOrderDetail

OrderID	LineItemNo	ProductID	Quantity
1	1	3	1
2	1	2	5
2	2	3	1
3	1	1	1

Product

ProductID	Name	ListPrice
1	Widget	2.99
2	Gizmo	1.79
3	Thingybob	3.49

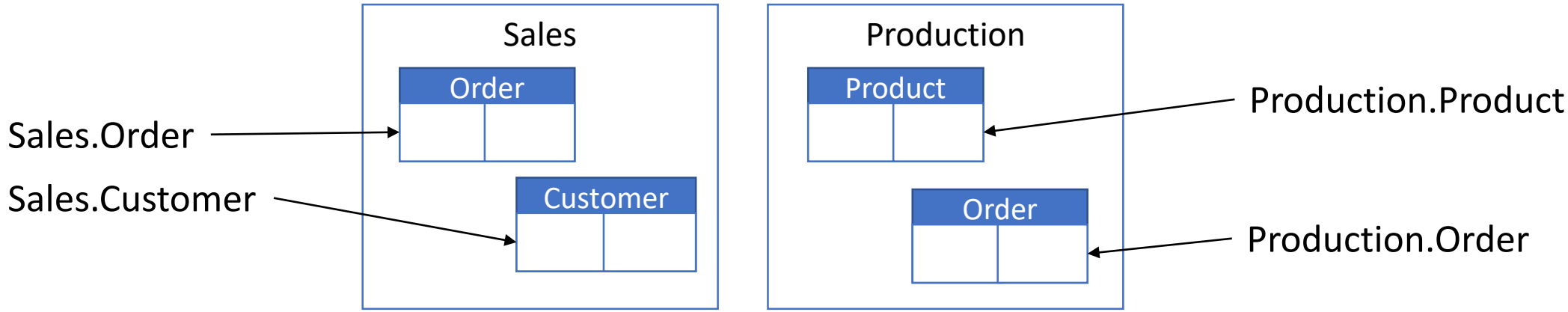
Schemas and Object Names

- Schemas are namespaces for database objects
- Fully-qualified names:

  [*server_name*.][*database_name*.][*schema_name*.]*object_name*

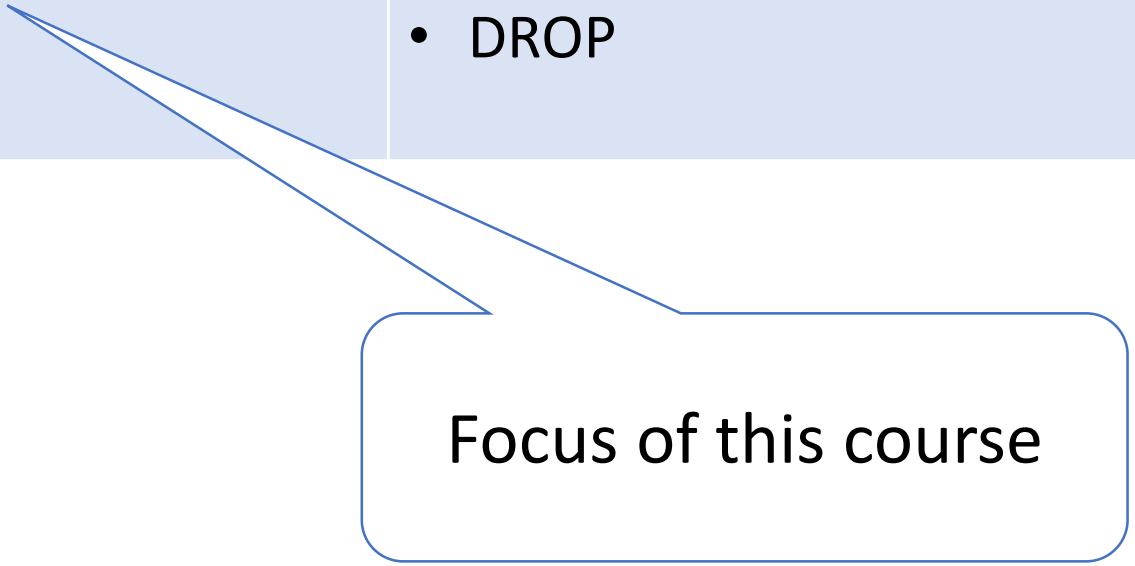
- Within database context, best practice is to include schema name:

  schema_name.object_name



SQL Statement Types

Data Manipulation Language (DML)	Data Definition Language (DDL)	Data Control Language (DCL)
Statements for querying and modifying data: • SELECT • INSERT • UPDATE • DELETE	Statements for defining database objects: • CREATE • ALTER • DROP	Statements for assigning security permissions: • GRANT • REVOKE • DENY



The SELECT Statement

	Element	Expression	Role
5	SELECT	<select list>	Defines which columns to return
1	FROM	<table source>	Defines table(s) to query
2	WHERE	<search condition>	Filters rows using a predicate
3	GROUP BY	<group by list>	Arranges rows by groups
4	HAVING	<search condition>	Filters groups using a predicate
6	ORDER BY	<order by list>	Sorts the output

```
SELECT OrderDate, COUNT(OrderID)
FROM Sales.SalesOrder
WHERE Status = 'Shipped'
GROUP BY OrderDate
HAVING COUNT(OrderID) > 1
ORDER BY OrderDate DESC;
```

Basic SELECT Query Examples

- All columns

```
SELECT * FROM Production.Product;
```

- Specific columns

```
SELECT Name, ListPrice
FROM Production.Product;
```

- Expressions and Aliases

```
SELECT Name AS Product, ListPrice * 0.9 AS SalePrice
FROM Production.Product;
```

Basic SELECT Queries

```
SELECT Name, StandardCost, ListPrice
FROM SalesLT.Product;


SELECT Name, ListPrice - StandardCost
FROM SalesLT.Product;


SELECT Name, ListPrice - StandardCost AS Markup
FROM SalesLT.Product;


SELECT ProductNumber, Color, Size, Color + ', ' + Size AS ProductDetails
FROM SalesLT.Product;


SELECT ProductID + ': ' + Name
FROM SalesLT.Product;
```

Working with Data Types

Exact Numeric	Approximate Numeric	Character	Date/Time	Binary	Other
tinyint	float	char	date	binary	cursor
smallint	real	varchar	time	varbinary	hierarchyid
int		text	datetime	image	sql_variant
bigint		nchar	datetime2		table
bit		nvarchar	smalldatetime		timestamp
decimal		ntext	datetimeoffset		uniqueidentifier
numeric					xml
money					geography
smallmoney					geometry

Data Type Conversion

- Implicit Conversion
  - Compatible data types can be automatically converted
- Explicit Conversion
  - Requires an explicit conversion function
    - CAST / TRY_CAST
    - CONVERT / TRY_CONVERT
    - PARSE / TRY_PARSE
    - STR

Converting Data Types

```
SELECT CAST(ProductID AS varchar(5)) + ': ' + Name AS ProductName
FROM SalesLT.Product;

SELECT CONVERT(varchar(5), ProductID) + ': ' + Name AS ProductName
FROM SalesLT.Product;

SELECT SellStartDate,
       CONVERT(nvarchar(30), SellStartDate) AS ConvertedDate,
       CONVERT(nvarchar(30), SellStartDate, 126) AS ISO8601FormatDate
FROM SalesLT.Product;

SELECT Name, CAST (Size AS Integer) AS NumericSize
FROM SalesLT.Product; --(note error - some sizes are incompatible)

SELECT Name, TRY_CAST (Size AS Integer) AS NumericSize
FROM SalesLT.Product; --(note incompatible sizes are returned as NULL)
```

Working with NULLs – NULL values

- NULL represents a missing or unknown value
- ANSI behaviour for NULL values:
  - The result of any expression containing a NULL value is NULL
    - 2 + NULL = NULL
    - ‘MyString: ‘ + NULL = NULL
  - Equality comparisons always return false for NULL values
    - NULL = NULL returns *false*
    - NULL **IS NULL** returns *true*

Working with NULLs – NULL Functions

- ISNULL(*column/variable, value*)
  - Returns *value* if the column or variable is NULL
- *NULLIF(column/variable, value*)
  - Returns NULL if the column or variable is *value*
- COALESCE (*column/variable1, column/variable2,…*)
  - Returns the value of the first non-NULL column or variable in the list

Working with NULLs

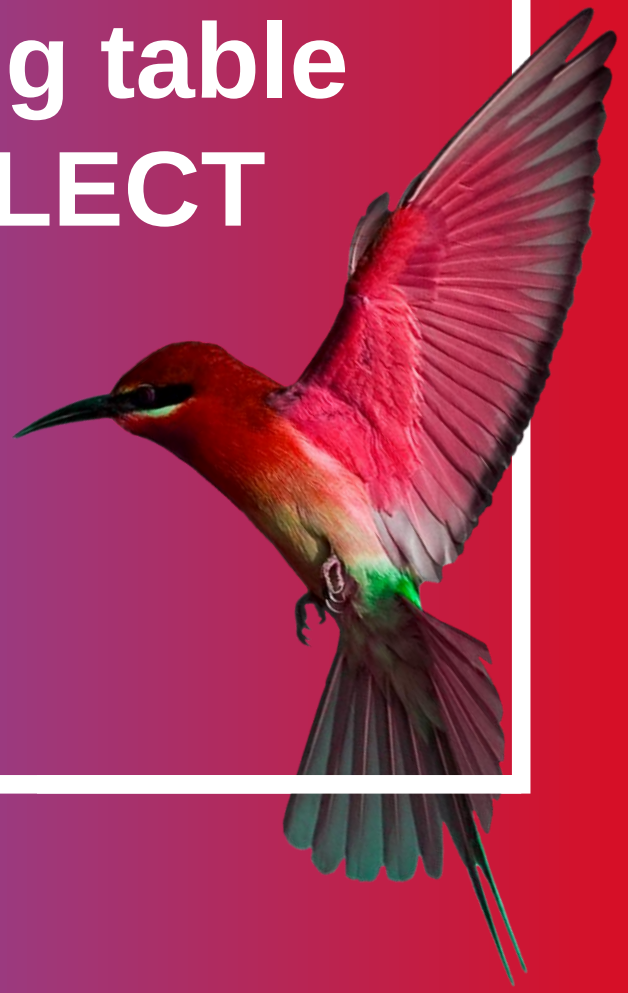
```
SELECT Name, ISNULL(TRY_CAST(Size AS Integer),0) AS NumericSize
FROM SalesLT.Product;

SELECT ProductNumber, ISNULL(Color, '') + ', ' + ISNULL(Size, '') AS ProductDetails
FROM SalesLT.Product;

SELECT Name, NULLIF(Color, 'Multi') AS SingleColor
FROM SalesLT.Product;

SELECT Name, COALESCE(DiscontinuedDate, SellEndDate, SellStartDate) AS FirstNonNullDate
FROM SalesLT.Product;

--Searched case
SELECT Name,
        CASE
            WHEN SellEndDate IS NULL THEN 'On Sale'
            ELSE 'Discontinued'
        END AS SalesStatus
FROM SalesLT.Product;

--Simple case
SELECT Name,
        CASE Size
            WHEN 'S' THEN 'Small'
            WHEN 'M' THEN 'Medium'
            WHEN 'L' THEN 'Large'
            WHEN 'XL' THEN 'Extra-Large'
            ELSE ISNULL(Size, 'n/a')
        END AS ProductSize
FROM SalesLT.Product;
```

Module Recap

- What is Transact-SQL?
- Relational Databases
- Schemas and Object Names
- SQL Statement Types
- The SELECT Statement
- Working with Data Types
- Working with NULLs

- Lab: Introduction to Transact-SQL

Module 2: Querying table with SELECT



Nash Tech.

Module Overview

- Removing Duplicates
- Sorting Results
- Paging Sorted Results
- Filtering and Using Predicates

Removing Duplicates

- SELECT ALL
  - Default behavior includes duplicates

```
SELECT Color
FROM Production.Product;
```

- SELECT DISTINCT
  - Removes duplicates

```
SELECT DISTINCT Color
FROM Production.Product;
```

Color
Blue
Red
Yellow
Blue
Yellow
Black

Color
Blue
Red
Yellow
Black

Sorting Results

- Use ORDER BY to sort results by one or more columns
  - Aliases created in SELECT clause are visible to ORDER BY
  - You can order by columns in the source that are not included in the SELECT clause
  - You can specify ASC or DESC (ASC is the default)

```
SELECT ProductCategory AS Category, ProductName
FROM Production.Product
ORDER BY Category, Price DESC;
```

Limiting Sorted Results

- TOP allows you to limit the number or percentage of rows returned by a query
- Works with ORDER BY clause to limit rows by sort order
- Added to SELECT clause:
  - SELECT TOP (N) | TOP (N) Percent
    - With percent, number of rows rounded up
  - SELECT TOP (N) WITH TIES
    - Retrieve duplicates where applicable (nondeterministic)

```
SELECT TOP 10 ProductName, ListPrice
FROM Production.Product
ORDER BY ListPrice DESC;
```

Paging Through Results

OFFSET-FETCH is an extension to the ORDER BY clause:

- Allows filtering a requested range of rows
  - Dependent on ORDER BY clause
- Provides a mechanism for paging through results
- Specify number of rows to skip, number of rows to retrieve:

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW(S)
FETCH FIRST|NEXT <fetch_value> ROW(S) ONLY
```

Eliminating Duplicates and Sorting Results

```
--Display a list of product colors
SELECT Color FROM SalesLT.Product;

--Display a list of product colors with the word 'None' if the value is null
SELECT DISTINCT ISNULL(Color, 'None') AS Color FROM SalesLT.Product;

--Display a list of product colors with the word 'None' if the value is null sorted by color
SELECT DISTINCT ISNULL(Color, 'None') AS Color FROM SalesLT.Product ORDER BY Color;

--Display a list of product colors with the word 'None' if the value is null and a dash if the size is null sorted by color
SELECT DISTINCT ISNULL(Color, 'None') AS Color, ISNULL(Size, '-') AS Size FROM SalesLT.Product ORDER BY Color;


--Display the top 100 products by list price
SELECT TOP 100 Name, ListPrice FROM SalesLT.Product ORDER BY ListPrice DESC;

--Display the first ten products by product number
SELECT Name, ListPrice FROM SalesLT.Product ORDER BY ProductNumber OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;

--Display the next ten products by product number
SELECT Name, ListPrice FROM SalesLT.Product ORDER BY ProductNumber OFFSET 10 ROWS FETCH FIRST 10 ROW ONLY;
```

Filtering and Using Predicates

- Specify predicates in the WHERE clause

Predicates and Operators	Description
= < >	Compares values for equality / non-equality.
IN	Determines whether a specified value matches any value in a subquery or a list.
BETWEEN	Specifies an inclusive range to test.
LIKE	Determines whether a specific character string matches a specified pattern, which can include wildcards.
AND	Combines two Boolean expressions and returns TRUE only when both are TRUE.
OR	Combines two Boolean expressions and returns TRUE if either is TRUE.
NOT	Reverses the result of a search condition.

Filtering with Predicates

```sql
--List information about product model 6
SELECT Name, Color, Size FROM SalesLT.Product WHERE ProductModelID = 6;

--List information about products that have a product number beginning FR
SELECT productnumber,Name, ListPrice FROM SalesLT.Product WHERE ProductNumber LIKE 'FR%';

--Filter the previous query to ensure that the product number contains two sets of two didgets
SELECT Name, ListPrice FROM SalesLT.Product WHERE ProductNumber LIKE 'FR-_[0-9][0-9]_-[0-9][0-9]';

--Find products that have no sell end date
SELECT Name FROM SalesLT.Product WHERE SellEndDate IS NOT NULL;

--Find products that have a sell end date in 2006
SELECT Name FROM SalesLT.Product WHERE SellEndDate BETWEEN '2006/1/1' AND '2006/12/31';

--Find products that have a category ID of 5, 6, or 7.
SELECT ProductCategoryID, Name, ListPrice FROM SalesLT.Product WHERE ProductCategoryID IN (5, 6, 7);

--Find products that have a category ID of 5, 6, or 7 and have a sell end date
SELECT ProductCategoryID, Name, ListPrice, SellEndDate FROM SalesLT.Product WHERE ProductCategoryID IN (5, 6, 7) AND SellEndDate IS NULL;

--Select products that have a category ID of 5, 6, or 7 and a product number that begins FR
SELECT Name, ProductCategoryID, ProductNumber FROM SalesLT.Product WHERE ProductNumber LIKE 'FR%' OR ProductCategoryID IN (5,6,7);
```

Module Recap

- Removing Duplicates
- Sorting Results
- Paging Sorted Results
- Filtering and Using Predicates

- Lab: Querying Tables with SELECT

Module 3: Query multiple tables with JOINs

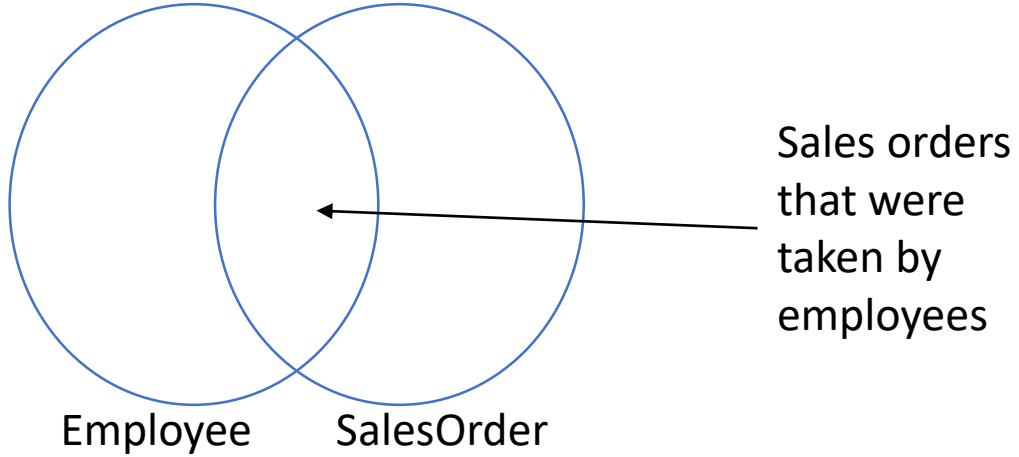
Nash Tech.

Module Overview

- Join Concepts
- Join Syntax
- Inner Joins
- Outer Joins
- Cross Joins
- Self Joins

Join Concepts

- Combine rows from multiple tables by specifying matching criteria
  - Usually based on primary key – foreign key relationships
  - For example, return rows that combine data from the **Employee** and **SalesOrder** tables by matching the **Employee.EmployeeID** primary key to the **SalesOrder.EmployeeID** foreign key
- It helps to think of the tables as sets in a Venn diagram



Join Syntax

- ANSI SQL-92
  - Tables joined by JOIN operator in FROM Clause
    - Preferred syntax

```
SELECT ...
FROM   Table1 JOIN Table2
       ON <on_predicate>;
```

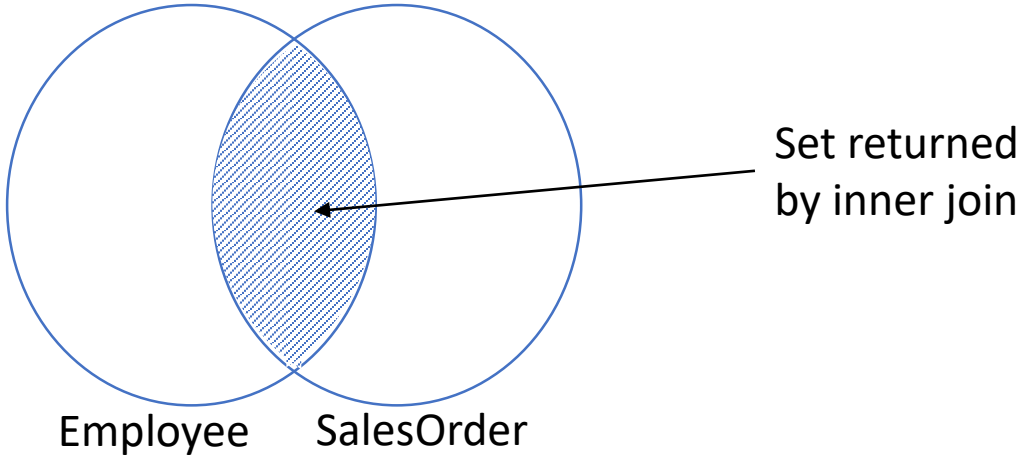
- ANSI SQL-89
  - Tables joined by commas in FROM Clause
    - Not recommended: Accidental Cartesian products!

```
SELECT ...
FROM   Table1, Table2
WHERE  <where_predicate>;
```

Inner Joins

- Return only rows where a match is found in both input tables
- Match rows based on attributes supplied in predicate
- If join predicate operator is =, also known as equi-join

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
[INNER] JOIN Sales.SalesOrder AS ord
ON emp.EmployeeID = ord.EmployeeID
```



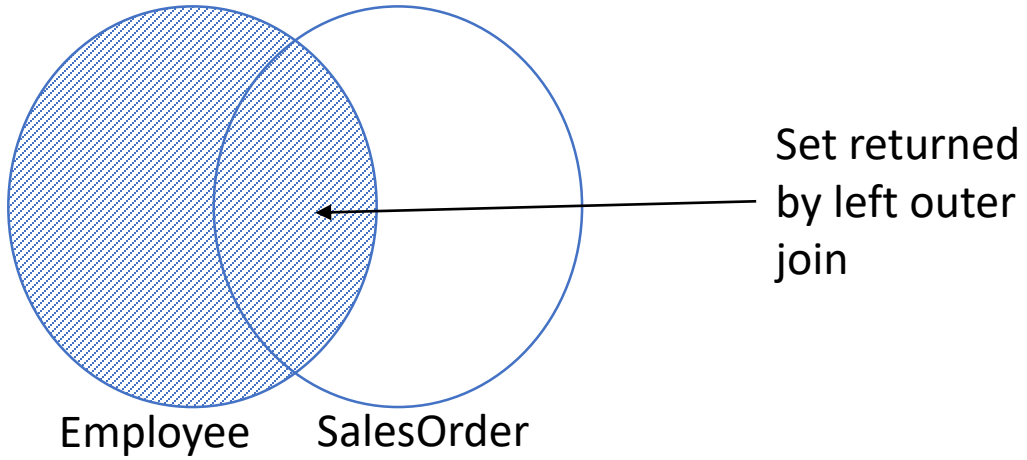
Using Inner Joins

```
-- Table aliases
SELECT p.Name As ProductName, c.Name AS Category
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory As c
ON p.ProductCategoryID = c.ProductCategoryID;

-- Joining more than 2 tables
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
INNER JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
INNER JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;

-- Multiple join predicates
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
INNER JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
INNER JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID AND od.UnitPrice = p.ListPrice --Note multiple predicates
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

Outer Joins

- Return all rows from one table and any matching rows from second table
- One table's rows are "preserved"
  - Designated with LEFT, RIGHT, FULL keyword
  - All rows from preserved table output to result set
- Matches from other table retrieved
- Additional rows added to results for non-matched rows
  - NULLs added in places where attributes do not match
- Example: Return all employees and for those who have taken orders, return the order amount. Employees without matching orders will display NULL for order amount.

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
LEFT [OUTER] JOIN Sales.SalesOrder AS ord
ON emp.EmployeeID = ord.EmployeeID;
```



Using Outer Joins

```
--Get all customers, with sales orders for those who've bought anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
ORDER BY c.CustomerID;

--Return only customers who haven't purchased anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
WHERE oh.SalesOrderNumber IS NULL
ORDER BY c.CustomerID;


--More than 2 tables
SELECT p.Name As ProductName, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT JOIN SalesLT.SalesOrderDetail AS od
ON p.ProductID = od.ProductID
LEFT JOIN SalesLT.SalesOrderHeader AS oh --Additional tables added to the right must also use a left join
ON od.SalesOrderID = oh.SalesOrderID
ORDER BY p.ProductID;
```

Cross Joins

- Combine each row from first table with each row from second table
- All possible combinations output
- Logical foundation for inner and outer joins
  - Inner join starts with Cartesian product, adds filter
  - Outer join takes Cartesian output, filtered, adds back non-matching rows (with NULL placeholders)
- Due to Cartesian product output, not typically a desired form of join
  - Some useful exceptions:
    - Table of numbers, generating data for testing

Employee	
EmployeeID	FirstName
1	Dan
2	Aisha

Product	
ProductID	Name
1	Widget
2	Gizmo

```
SELECT emp.FirstName, prd.Name
FROM HR.Employee AS emp
CROSS JOIN Production.Product AS prd;
```

Result	
FirstName	Name
Dan	Widget
Dan	Gizmo
Aisha	Widget
Aisha	Gizmo

Using Cross Joins

```
--Call each customer once per product
SELECT p.Name, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Product as p
CROSS JOIN SalesLT.Customer as c;
```

Self Joins

- Compare rows in same table to each other
- Create two instances of same table in FROM clause
  - At least one alias required
- Example: Return all employees and the name of the employee's manager

Employee		
EmployeeID	FirstName	ManagerID
1	Dan	NULL
2	Aisha	1
3	Rosie	1
4	Naomi	3

```
SELECT emp.FirstName AS Employee,
       man.FirstName AS Manager
FROM HR.Employee AS emp
LEFT JOIN HR.Employee AS man
ON emp.ManagerID = man.EmployeeID;
```

Result	
Employee	Manager
Dan	*NULL*
Aisha	Dan
Rosie	Dan
Naomi	Rosie

Using Self Joins

```
--note there's no employee table, so we'll create one for this example
CREATE TABLE SalesLT.Employee
(EmployeeID int IDENTITY PRIMARY KEY,
EmployeeName nvarchar(256),
ManagerID int);
GO
-- Get salesperson from Customer table and generate managers
INSERT INTO SalesLT.Employee (EmployeeName, ManagerID)
SELECT DISTINCT Salesperson, NULLIF(CAST(RIGHT(SalesPerson, 1) as INT), 0)
FROM SalesLT.Customer;
GO
UPDATE SalesLT.Employee
SET ManagerID = (SELECT MIN(EmployeeID) FROM SalesLT.Employee WHERE ManagerID IS NULL)
WHERE ManagerID IS NULL
AND EmployeeID > (SELECT MIN(EmployeeID) FROM SalesLT.Employee WHERE ManagerID IS NULL);
GO

-- Here's the actual self-join demo
SELECT e.EmployeeName, m.EmployeeName AS ManagerName
FROM SalesLT.Employee AS e
LEFT JOIN SalesLT.Employee AS m
ON e.ManagerID = m.EmployeeID
ORDER BY e.ManagerID;
```

Module Recap

- Join Concepts
- Join Syntax
- Inner Joins
- Outer Joins
- Cross Joins
- Self Joins

- Lab: Querying Multiple Tables with Joins

Module 4: Using Set Operators

Nash Tech.

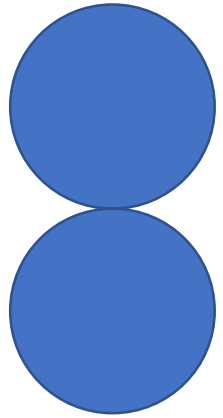
Module Overview

- What are UNION Queries?
- What are INTERSECT Queries?
- What are EXCEPT Queries?

What are UNION Queries?

- UNION returns a result set of distinct rows combined from all statements
- UNION removes duplicates during query processing (affects performance)
- UNION ALL retains duplicates during query processing

```
-- only distinct rows from both queries are returned
SELECT countryregion, city FROM HR.Employees
UNION
SELECT countryregion, city FROM Sales.Customers;
```



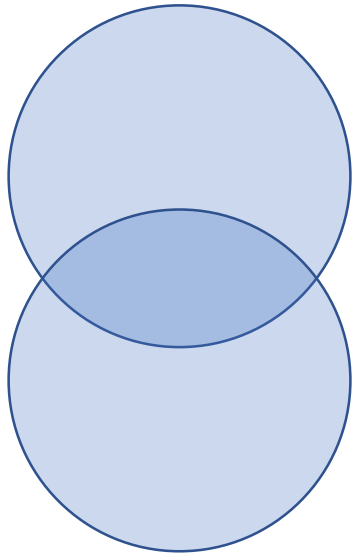
UNION Guidelines

- Column aliases
  - Must be expressed in first query
- Number of columns
  - Must be the same
- Data types
  - Must be compatible for implicit conversion (or converted explicitly)

What are INTERSECT Queries?

- INTERSECT returns only distinct rows that appear in both result sets

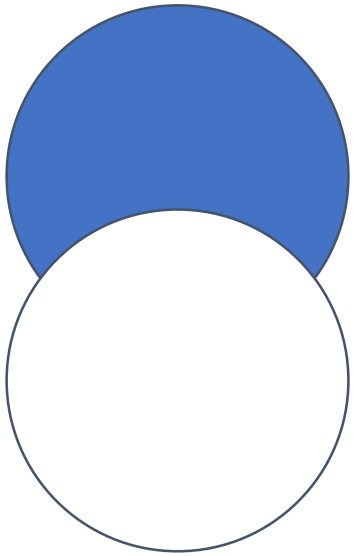
```
-- only rows that exist in both queries will be returned
SELECT countryregion, city FROM HR.Employees
INTERSECT
SELECT countryregion, city FROM Sales.Customers;
```



What are EXCEPT Queries?

- EXCEPT returns only distinct rows that appear in the first set but not the second
  - Order in which sets are specified matters

```
-- only rows from Employees will be returned
SELECT countryregion, city FROM HR.Employees
EXCEPT
SELECT countryregion, city FROM Sales.Customers;
```



Creating UNION, INTERSECT, and EXCEPT Queries

```sql
-- Union example
SELECT FirstName, LastName
FROM SalesLT.Employees
UNION
SELECT FirstName, LastName
FROM SalesLT.Customers
ORDER BY LastName;
```

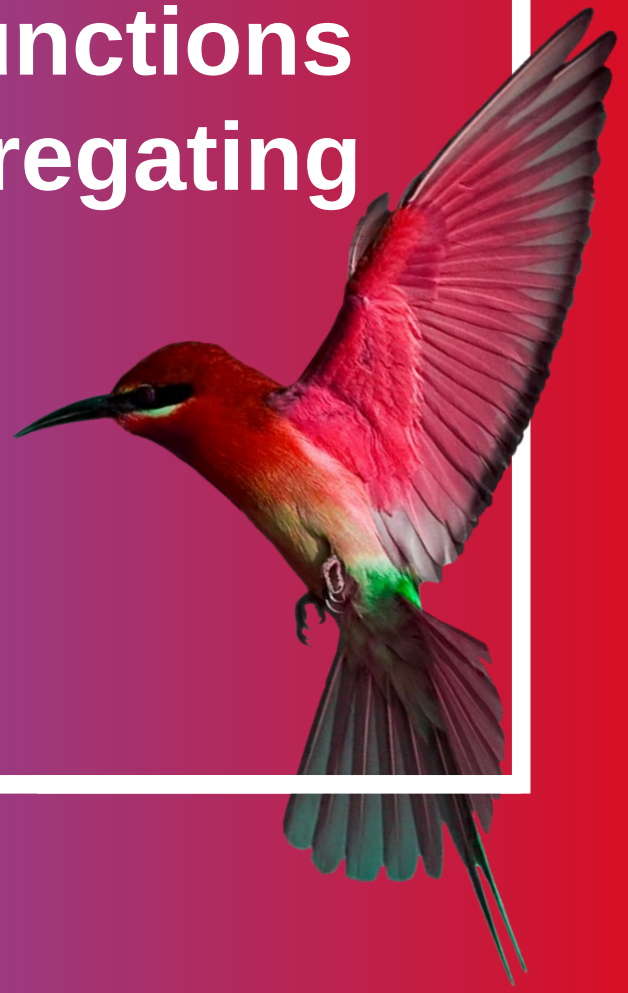
```sql
--Intersect
SELECT FirstName, LastName
FROM SalesLT.Customers
INTERSECT
SELECT FirstName, LastName
FROM SalesLT.Employees;

--Except
SELECT FirstName, LastName
FROM SalesLT.Customers
EXCEPT
SELECT FirstName, LastName
FROM SalesLT.Employees;
```

Module Recap

- What are UNION Queries?
- What are INTERSECT Queries?
- What are EXCEPT Queries?

- Lab: Using Set Operators

Module 5: Using Functions and Aggregating Data



Nash Tech.

Module Overview

- Introduction to Built-In Functions
- Scalar Functions
- Aggregate Functions
- Logical Functions
- Window Functions
- Grouping with GROUP BY
- Filtering with HAVING

Introduction to Built-In Functions

Function Category	Description
Scalar	Operate on a single row, return a single value
Logical	Scalar functions that compare multiple values to determine a single output
Aggregate	Take one or more input values, return a single summarizing value
Window	Operate on a window (set) of rows
Rowset	Return a virtual table that can be used subsequently in a Transact-SQL statement

Scalar Functions

- Operate on elements from a single row as inputs, return a single value as output
- Return a single (scalar) value
- Can be used like an expression in queries
- May be deterministic or non-deterministic

Scalar Function Categories

- Configuration
- Conversion
- Cursor
- Date and Time
- Mathematical
- Metadata
- Security
- String
- System
- System Statistical
- Text and Image

Using Scalar Functions

```
-- Scalar functions
SELECT YEAR(SellStartDate) SellStartYear, ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;

SELECT YEAR(SellStartDate) SellStartYear, DATENAME(mm,SellStartDate) SellStartMonth,
       DAY(SellStartDate) SellStartDay, DATENAME(dw, SellStartDate) SellStartWeekday,
       ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;

SELECT DATEDIFF(yy,SellStartDate, GETDATE()) YearsSold, ProductID, Name
FROM SalesLT.Product
ORDER BY ProductID;

SELECT CONCAT(FirstName + ' ', LastName) AS FullName
FROM SalesLT.Customer;

SELECT Name, ProductNumber, LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product;
```

Logical Functions

Output is determined by comparative logic

- ISNUMERIC

```
SELECT ISNUMERIC('101.99') AS Is_a_Number;
```

- IIF

```
SELECT productid, listprice, IIF(listprice > 50, 'high','low') AS PricePoint
FROM Production.Product;
```

- CHOOSE

```
SELECT ProductName, Color, Size,
        CHOOSE (ProductCategoryID, 'Bikes','Components','Clothing','Accessories') AS
Category
FROM Production.Product;
```

Using Logical Functions

```
-- Logical functions
    SELECT Name, ISNUMERIC(Size) AS NumericSize
    FROM SalesLT.Product


    SELECT Name, IIF(ProductCategoryID IN (5,6,7), 'Bike', 'Other') ProductType
    FROM SalesLT.Product;

    SELECT Name, IIF(ISNUMERIC(Size) = 1, 'Numeric', 'Non-Numeric') SizeType
    FROM SalesLT.Product;

    SELECT prd.Name AS ProductName, cat.Name AS Category,
          CHOOSE (cat.ParentProductCategoryID, 'Bikes','Components','Clothing','Accessories') AS ProductType
    FROM SalesLT.Product AS prd
    JOIN SalesLT.ProductCategory AS cat
    ON prd.ProductCategoryID = cat.ProductCategoryID;
```

Window Functions

- Functions applied to a window, or set of rows
- Include ranking, offset, aggregate and distribution functions

```
SELECT TOP(3) ProductID, Name, ListPrice,
     RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM Production.Product
ORDER BY RankByPrice;
```

ProductID	Name	ListPrice	RankByPrice
8	Gizmo	263.50	1
29	Widget	123.79	2
9	Thingybob	97.00	3

Using Window Functions

```
-- Window functions
SELECT TOP(100) ProductID, Name, ListPrice,
    RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
ORDER BY RankByPrice;

SELECT c.Name AS Category, p.Name AS Product, ListPrice,
    RANK() OVER(PARTITION BY c.Name ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductcategoryID
ORDER BY Category, RankByPrice;
```

Aggregate Functions

- Functions that operate on sets, or rows of data
- Summarize input rows
- Without GROUP BY clause, all rows are arranged as one group

```
SELECT COUNT(*) AS OrderLines,
          SUM(OrderQty*UnitPrice) AS TotalSales
FROM   Sales.OrderDetail;
```

OrderLines	TotalSales
542	714002.9136

Using Aggregate Functions

```
-- Aggregate Functions
SELECT COUNT(*) AS Products,
       COUNT(DISTINCT ProductCategoryID) AS Categories,
       AVG(ListPrice) AS AveragePrice
FROM SalesLT.Product;

SELECT COUNT(p.ProductID) BikeModels,
       AVG(p.ListPrice) AveragePrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductCategoryID
WHERE c.Name LIKE '%Bikes';
```

Grouping with GROUP BY

- GROUP BY creates groups for output rows, according to a unique combination of values specified in the GROUP BY clause
- GROUP BY calculates a summary value for aggregate functions in subsequent phases
- Detail rows are “lost” after GROUP BY clause is processed

```
SELECT CustomerID, COUNT(*) AS Orders
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

Grouping with GROUP BY

```
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
ORDER BY Salesperson;

SELECT c.Salesperson, ISNULL(SUM(oh.SubTotal), 0.00) SalesRevenue
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;

SELECT c.Salesperson, CONCAT(c.FirstName +' ', c.LastName) AS Customer,
       ISNULL(SUM(oh.SubTotal), 0.00) SalesRevenue
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson, CONCAT(c.FirstName +' ', c.LastName)
ORDER BY SalesRevenue DESC, Customer;
```

Filtering with HAVING

- HAVING clause provides a search condition that each group must satisfy
- WHERE clause is processed before GROUP BY, HAVING clause is processed after GROUP BY

```
SELECT CustomerID, COUNT(*) AS Orders
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
HAVING COUNT(*) > 10;
```

Filtering with HAVING

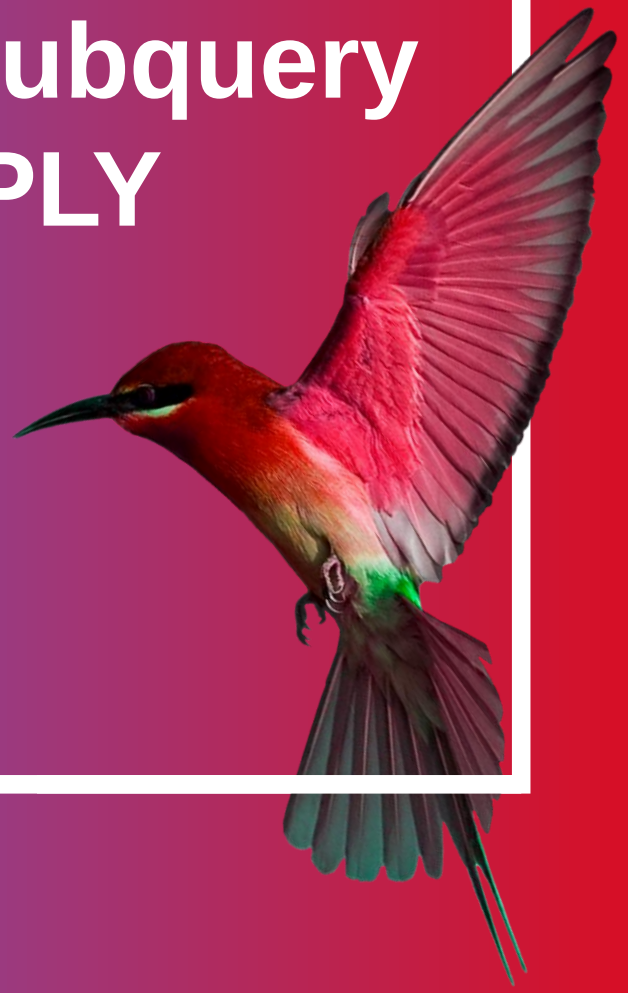
```
-- Try to find salespeople with over 150 customers (fails with error)
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
WHERE COUNT(CustomerID) > 100
GROUP BY Salesperson
ORDER BY Salesperson;

--Need to use HAVING clause to filter based on aggregate
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
HAVING COUNT(CustomerID) > 100
ORDER BY Salesperson;
```

Module Recap

- Introduction to Built-In Functions
- Scalar Functions
- Aggregate Functions
- Logical Functions
- Window Functions
- Grouping with GROUP BY
- Filtering with HAVING

- Lab: Using Functions and Aggregating Data

Module 6: Using Subquery and APPLY



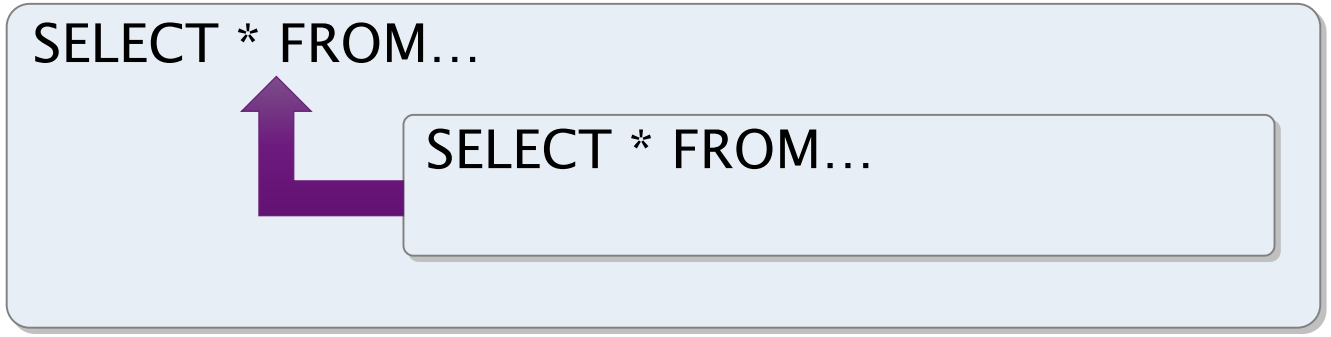
Nash Tech.

Module Overview

- Introduction to Subqueries
- Scalar or Multi-Valued?
- Self-Contained or Correlated?
- Using APPLY with Table-Valued Functions

Introduction to Subqueries

- Subqueries are nested queries: queries within queries
- Results of inner query passed to outer query
  - Inner query acts like an expression from perspective of outer query



Scalar or Multi-Valued?

- Scalar subquery returns single value to outer query
  - Can be used anywhere single-valued expression is used: SELECT, WHERE, and so on

```
SELECT orderid, productid, unitprice, qty
FROM Sales.OrderDetails
WHERE orderid =
      (SELECT MAX(orderid) AS lastorder
      FROM Sales.Orders);
```

- Multi-valued subquery returns multiple values as a single column set to the outer query
  - Used with IN predicate

```
SELECT custid, orderid
FROM Sales.orders
WHERE custid IN (
      SELECT custid
      FROM Sales.Customers
      WHERE countryregion = N'Mexico');
```

Using Subqueries

```
--Display a list of products whose list price is higher than
--the highest unit price of items that have sold
SELECT MAX(UnitPrice) FROM SalesLT.SalesOrderDetail

SELECT * from SalesLT.Product
WHERE ListPrice >

SELECT * from SalesLT.Product
WHERE ListPrice >
(SELECT MAX(UnitPrice) FROM SalesLT.SalesOrderDetail)
```

```
--List products that have an order quantity greater than 20
SELECT Name FROM SalesLT.Product
WHERE ProductID IN
(SELECT ProductID from SalesLT.SalesOrderDetail
WHERE OrderQty>20)

SELECT Name
FROM SalesLT.Product P
JOIN SalesLT.SalesOrderDetail SOD
ON P.ProductID=SOD.ProductID
WHERE OrderQty>20
```

Self-Contained or Correlated?

- Most subqueries are self-contained and have no connection with the outer query other than passing it results
- Correlated subqueries refer to elements of tables used in outer query
  - Dependent on outer query, cannot be executed separately
  - Behaves as if inner query is executed once per outer row
  - May return scalar value or multiple values

```
SELECT orderid, empid, orderdate
FROM Sales.Orders AS O1
WHERE orderdate = (SELECT MAX(orderdate)
                   FROM Sales.Orders AS O2
                   WHERE O2.empid = O1.empid)
ORDER BY empid, orderdate;
```

Creating a Correlated Subquery

```
--For each customer list all sales on the last day that they made a sale

SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
ORDER BY CustomerID,OrderDate

SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
WHERE orderdate =
(SELECT MAX(orderdate)
FROM SalesLT.SalesOrderHeader)


SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
WHERE orderdate =
(SELECT MAX(orderdate)
FROM SalesLT.SalesOrderHeader AS SO2
WHERE SO2.CustomerID = SO1.CustomerID)
ORDER BY CustomerID
```

Using APPLY with Table-Valued Functions

- CROSS APPLY applies the right table expression to each row in left table
  - Conceptually similar to CROSS JOIN between two tables but can correlate data between sources

```
SELECT S.supplierid, s.companyname, P.productid,  P.productname, P.unitprice
FROM Production.Suppliers AS S
CROSS APPLY dbo.fn_TopProductsByShipper(S.supplierid) AS P
```

- OUTER APPLY adds rows for those with NULL in columns for right table
  - Conceptually similar to LEFT OUTER JOIN between two tables

Using APPLY with Table-Valued Functions

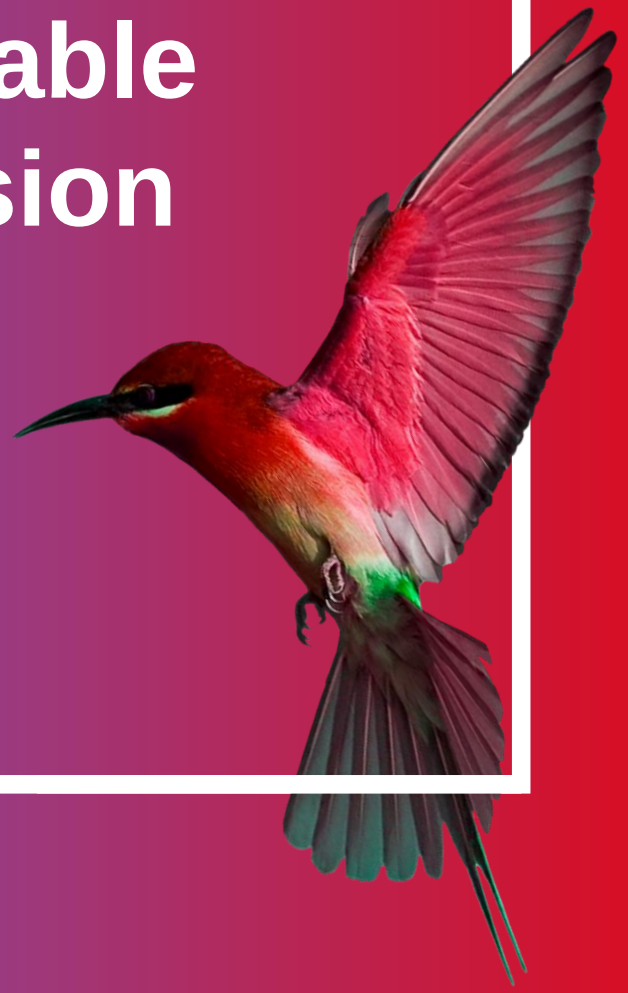
```
-- Setup
CREATE FUNCTION SalesLT.udfMaxUnitPrice (@SalesOrderID int)
RETURNS TABLE
AS
RETURN
SELECT SalesOrderID,Max(UnitPrice) as MaxUnitPrice FROM
SalesLT.SalesOrderDetail
WHERE SalesOrderID=@SalesOrderID
GROUP BY SalesOrderID;

--Display the sales order details for items that are equal to
-- the maximum unit price for that sales order
SELECT * FROM SalesLT.SalesOrderDetail AS SOH
CROSS APPLY SalesLT.udfMaxUnitPrice(SOH.SalesOrderID) AS MUP
WHERE SOH.UnitPrice=MUP.MaxUnitPrice
ORDER BY SOH.SalesOrderID;
```

Module Recap

- Introduction to Subqueries
- Scalar or Multi-Valued?
- Self-Contained or Correlated?
- Using APPLY with Table-Valued Functions

- Lab: Using Subqueries and APPLY

Module 7: Using Table Expression



Nash Tech.

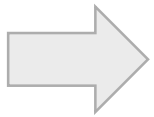
Module Overview

- Views
- Temporary Tables
- Table Variables
- Table-Valued Functions
- Derived Tables
- Common Table Expressions

Querying Views

- Views are named queries with definitions stored in a database
  - Views can provide abstraction, encapsulation and simplification
  - From an administrative perspective, views can provide a security layer to a database
- Views may be referenced in a SELECT statement just like a table

```
CREATE VIEW Sales.vSalesOrders
AS
SELECT  oh.OrderID, oh.Orderdate, oh.CustomerID,
            od.LineItemNo, od.ProductID, od.Quantity
FROM Sales.OrderHeaders AS oh
JOIN Sales.OrderDetails AS od
ON od.OrderID = oh.OrderID;
```



```
SELECT OrderID, CustomerID, ProductID
FROM Sales.vSalesOrder
ORDER BY OrderID;
```

Temporary Tables

- Temporary tables are used to hold temporary result sets within a user's session
  - Created in tempdb and deleted automatically
  - Created with a # prefix
  - Global temporary tables are created with ## prefix

```
CREATE TABLE #tmpProducts
(ProductID INTEGER,
 ProductName varchar(50));
GO
…
SELECT * FROM #tmpProducts;
```

Table Variables

- Introduced because temporary tables can cause recompilations
- Used similarly to temporary tables but scoped to the batch
- Use only on very small datasets

```
DECLARE @varProducts table
(ProductID INTEGER,
 ProductName varchar(50));
…
SELECT * FROM @varProducts
```

Temporary Tables and Table Variables

```
-- Temporary table
CREATE TABLE #Colors
(Color varchar(15));

INSERT INTO #Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM #Colors;

-- Table variable
DECLARE @Colors AS TABLE (Color varchar(15));

INSERT INTO @Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM @Colors;

-- New batch
SELECT * FROM #Colors;

SELECT * FROM @Colors; -- now out of scope
```

Table-Valued Functions

- TVFs are named objects with definitions stored in a database
- TVFs return a virtual table to the calling query
- Unlike views, TVFs support input parameters
  - TVFs may be thought of as parameterized views

```
CREATE FUNCTION Sales.fn_GetOrderItems (@OrderID AS Integer)
RETURNS TABLE
AS
RETURN
(SELECT ProductID, UnitPrice, Quantity
FROM Sales.OrderDetails
WHERE OrderID = @OrderID);
…
SELECT * FROM Sales.fn_GetOrderItems (1025) AS LineItems;
```

Using Table-Valued Functions

```
CREATE FUNCTION SalesLT.udfCustomersByCity
(@City AS VARCHAR(20))
RETURNS TABLE
AS
RETURN
(SELECT C.CustomerID, FirstName, LastName, AddressLine1, City, StateProvince
 FROM SalesLT.Customer C JOIN SalesLT.CustomerAddress CA
 ON C.CustomerID=CA.CustomerID
 JOIN SalesLT.Address A ON CA.AddressID=A.AddressID
 WHERE City=@City);


SELECT * FROM SalesLT.udfCustomersByCity('Bellevue')
```

Derived Tables - Introduction

- Derived tables are named query expressions created within an outer SELECT statement
- Not stored in database – represents a virtual relational table
- Scope of a derived table is the query in which it is defined

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM
        (SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

Derived Tables - Guidelines

- Derived tables **must**:
  - Have an alias
  - Have unique names for all columns
  - Not use an ORDER BY clause (without TOP or OFFSET/FETCH)
  - Not be referred to multiple times in the same query
- Derived tables **may**:
  - Use internal or external aliases for columns
  - Refer to parameters and/or variables
  - Be nested within other derived tables

Derived Tables – Specifying column aliases

- Column aliases may be defined inline:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (  SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

- Or externally:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (  SELECT YEAR(orderdate), custid
        FROM Sales.Orders) AS
        derived_year(orderyear, custid)
GROUP BY orderyear;
```

Using Derived Tables

```
SELECT Category, COUNT(ProductID) AS Products
FROM
    (SELECT p.ProductID, p.Name AS Product, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID) AS ProdCats
GROUP BY Category
ORDER BY Category;
```

Common Table Expressions (CTEs)

- CTEs are named table expressions defined in a query
- CTEs are similar to derived tables in scope and naming requirements
- Unlike derived tables, CTEs support multiple references and recursion

```
WITH CTE_year (OrderYear, CustID)
AS
(
        SELECT YEAR(orderdate), custid
        FROM Sales.Orders
)
SELECT OrderYear, COUNT(DISTINCT CustID) AS Cust_Count
FROM CTE_year
GROUP BY orderyear;
```

Common Table Expressions - Recursion

- Specify a query for the anchor (root) level
- Use UNION ALL to add a recursive query for other levels
- Query the CTE, with optional MAXRECURSION option

```
WITH OrgReport (ManagerID, EmployeeID, EmployeeName, Level)
AS
(
        SELECT e.ManagerID, e.EmployeeID, EmployeeName, 0
        FROM HR.Employee AS e
        WHERE ManagerID IS NULL
        UNION ALL
        SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
        FROM HR.Employee AS e
        INNER JOIN OrgReport AS o ON e.ManagerID = o.EmployeeID
)
SELECT * FROM OrgReport
OPTION (MAXRECURSION 3);
```

Using Common Table Expressions(CTE)

```
--Using a CTE
WITH ProductsByCategory (ProductID, ProductName, Category)
AS
(
    SELECT p.ProductID, p.Name, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID
)

SELECT Category, COUNT(ProductID) AS Products
FROM ProductsByCategory
GROUP BY Category
ORDER BY Category;
```

Using Recursive CTE

```
-- Recursive CTE
SELECT * FROM SalesLT.Employee

-- Using the CTE to perform recursion
WITH OrgReport (ManagerID, EmployeeID, EmployeeName, Level)
AS
(
    -- Anchor query
    SELECT e.ManagerID, e.EmployeeID, EmployeeName, 0
    FROM SalesLT.Employee AS e
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive query
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM SalesLT.Employee AS e
    INNER JOIN OrgReport AS o ON e.ManagerID = o.EmployeeID
)

SELECT * FROM OrgReport
OPTION (MAXRECURSION 3);
```

Module Recap

- Views
- Temporary Tables
- Table Variables
- Table-Valued Functions
- Derived Tables
- Common Table Expressions

- Lab: Using Table Expressions

Reference

- Querying with T-SQL, Graeme Malcolm, Microsoft course

Thank you