[edX Home Page](https://courses.edx.org/dashboard)

Microsoft: DAT201x

**Querying Data with Transact-SQL**

Module 1: Introduction to Transact-SQL

* The fully qualified naming syntax for an object is *server\_name*.*database\_name*.*schema\_name*.*object\_name*, but in most cases you can abbreviate this to *schema\_name.object\_name.*
* SELECT statements are written with the following clauses: **SELECT**, **FROM**, **WHERE**, **GROUP** **BY**, **HAVING**, **ORDER** **BY**. However, the query engine processes the clauses in this order: **FROM**, **WHERE**, **GROUP BY**, **HAVING**, **SELECT**, **ORDER BY**.
* You can specify expressions in the SELECT clause to return the results of calculations.
* You can use the AS keyword to specify aliases for columns in the rowset returned by the SELECT statement.
* Transact-SQL supports a wide range of data types, which can be broadly categorized as **exact numeric**, **approximate numeric**, **character**, **date/time**, **binary**, and **other** (which includes specialized data types for handling data such as XML and spatial data).
* Some data types are compatible, and values can be implicitly converted between them. Conversion between other data types requires the use of explicit conversion functions.
* NULL is used to indicate an unknown or missing value. NULL is **not** equivalent to zero or an empty string.
* Arithmetic or string concatenation operations involving one or more NULL operands return NULL. For example, 12 + NULL = NULL.
* If you need to compare a value to NULL, use the **IS** operator instead of the **=** operator.
* The **ISNULL** function returns a specified alternative value for NULL columns and variables.
* The **NULLIF** function returns NULL when a column or variable contains a specified value.
* The **COALESCE** function returns the first non-NULL value in a specified list of columns or variables).

Module 1: EXAMPLES:

SELECT Title, FirstName, MiddleName, LastName, Suffix

FROM SalesLT.Customer

SELECT SalesPerson, Title + ' ' + LastName AS CustomerName, Phone

FROM SalesLT.Customer

SELECT CAST(CustomerID AS nvarchar) + ': ' + CompanyName AS CustomerCompanies

FROM SalesLT.Customer

SELECT SalesOrderNumber + ' ('+ CAST(RevisionNumber AS nvarchar) + ')',

CONVERT(nvarchar, OrderDate, 102)

FROM SalesLT.SalesOrderHeader

SELECT FirstName + ISNULL(' ' + MiddleName,'') + ' ' + LastName AS [List of Customers]

FROM SalesLT.Customer

SELECT CustomerID, ISNULL(EmailAddress, Phone) AS PrimaryContact

FROM SalesLT.Customer

SELECT CustomerID, COALESCE(EmailAddress, Phone) AS PrimaryContact

FROM SalesLT.Customer

SELECT SalesOrderID,

CASE

WHEN ShipDate IS NULL THEN 'Awaiting Shipment'

ELSE 'Shipped'

END AS ShippingStatus

FROM SalesLT.SalesOrderHeader

Module 2: Querying Tables with SELECT

* By default, the **SELECT** statement returns all rows. If multiple rows contain the same values for every column, they are duplicated in the results. Using the **DISTINCT** keyword eliminates duplicates, ensuring that only one row for each distinct combination of column values is returned.
* The order of rows in the result of a SELECT statement is not guaranteed unless you explicitly specify one or more columns in an **ORDER BY** clause. You can specify sort direction as **ASC** (the default) or **DESC**.
* You can combine the **ORDER BY** clause with the **TOP** keyword to restrict the results so that they include only the top *n* rows (where n is the number or percentage of rows you want to return).
* You can implement a query to retrieve a specified "page" of results by using the **OFFSET** and **FETCH** keywords with the ORDER BY clause.
* Use the WHERE clause to filter the results returned by a SELECT query based on a search condition.
* A search condition is composed of one or more predicates.
* Predicates include conditional operators (such as =, >, and <), **IN**, **LIKE**, and **NOT**.
* You can use **AND** and **OR** to combine predicates based on Boolean logic.

Module 2: EXAMPLES:

SELECT DISTINCT City, StateProvince

FROM SalesLT.Address

SELECT TOP 10 PERCENT Name

FROM SalesLT.Product

ORDER BY Weight DESC

SELECT Name

FROM SalesLT.Product

ORDER BY Weight DESC

OFFSET 10 ROWS

FETCH NEXT 100 ROWS ONLY

SELECT Name, Color,Size

FROM SalesLT.Product

WHERE ProductModelID = 1

SELECT ProductNumber, Name

FROM SalesLT.Product

WHERE (Color='black' OR Color='red' OR Color='white') AND (Size='S' OR Size='M')

SELECT ProductNumber, Name

FROM SalesLT.Product

WHERE Color IN ('black', 'red', 'white') AND Size IN ('S', 'M')

SELECT ProductNumber, Name, ListPrice

FROM SalesLT.Product

WHERE ProductNumber LIKE 'BK-%'

SELECT ProductNumber, Name, ListPrice

FROM SalesLT.Product

WHERE ProductNumber LIKE 'BK-[^R]%-[0-9][0-9]'

Module 3: Querying Multiple Tables with Joins

* **JOIN**s are used to match rows in one table to rows in another table.
* The query engine supports two ways to define **JOIN**s:
  + ANSI SQL-92 syntax (in which the join is specified in the **FROM** clause) is the preferred approach.
  + ANSI SQL-89 syntax (in which the join is specified in the **WHERE** clause)
* **INNER JOIN**s return only rows where a match can be found in both tables.
* **INNER JOIN**s that match rows based on columns containing the same value in both tables are sometimes referred to as equi-joins.
* Use a **LEFT OUTER JOIN** to include all rows from the first table and values from matched rows in the second table. Columns in the second table for which no matching rows exist are populated with NULLs.
* Use a **RIGHT OUTER JOIN** to include all rows from the second table and values from matched rows in the first table. Columns in the first table for which no matching rows exist are populated with NULLs.
* Use a **FULL OUTER JOIN** to include all rows from the first and second tables. Columns in the either table for which no matching rows exist are populated with NULLs.
* A **CROSS JOIN** returns a Cartesian product that includes every combination of the selected columns from both tables.
* While not commonly used in typical application processing, **CROSS JOIN**s can be useful in some specialized scenarios - such as generating test data.
* A **self-join** is an inner, outer, or cross join that matches rows in a table to other rows in the same table.
* When defining a **self-join**, you must specify an alias for at least one instance of the table being joined.

Module 3: EXAMPLES:

SELECT b.CompanyName, a.SalesOrderID, a.TotalDue

FROM SalesLT.SalesOrderHeader AS a

LEFT OUTER JOIN SalesLT.Customer AS b

ON a.CustomerID = b.CustomerID

SELECT b.CompanyName, a.SalesOrderID, a.TotalDue, c.AddressType, d.AddressLine1, d.AddressLine2, d.City, d.StateProvince, d.PostalCode, d.CountryRegion

FROM SalesLT.SalesOrderHeader AS a

LEFT OUTER JOIN SalesLT.Customer AS b

ON a.CustomerID = b.CustomerID

LEFT OUTER JOIN SalesLT.CustomerAddress AS c

ON b.CustomerID = c.CustomerID

LEFT OUTER JOIN SalesLT.Address AS d

ON c.AddressID = d.AddressID

WHERE c.AddressType = 'Main Office'

SELECT a.CompanyName, a.FirstName, a.LastName, b.SalesOrderID, b.TotalDue

FROM SalesLT.Customer As a

FULL OUTER JOIN SalesLT.SalesOrderHeader AS b

ON a.CustomerID = b.CustomerID

ORDER BY b.TotalDue DESC

SELECT b.CustomerID, b.CompanyName, b.FirstName, b.LastName, b.Phone, c.AddressLine1, c.AddressLine2

FROM SalesLT.CustomerAddress AS a

LEFT OUTER JOIN SalesLT.Customer AS b

ON a.CustomerID = b.CustomerID

LEFT OUTER JOIN SalesLT.Address AS c

ON a.AddressID = c.AddressID

WHERE c.AddressLine2 IS NULL

SELECT b.CustomerID, b.CompanyName, b.FirstName, b.LastName, b.Phone, c.AddressLine1, c.AddressLine2

FROM SalesLT.Customer AS b

LEFT OUTER JOIN SalesLT.CustomerAddress AS a

ON a.CustomerID = b.CustomerID

LEFT OUTER JOIN SalesLT.Address AS c

ON a.AddressID = c.AddressID

WHERE c.AddressID IS NULL

SELECT c.CustomerID, p.ProductID

FROM SalesLT.Customer AS c

FULL OUTER JOIN SalesLT.SalesOrderHeader AS h

ON h.CustomerID = c.CustomerID --joined SalesOrderHeader with Customer ON customerID

FULL OUTER JOIN SalesLT.SalesOrderDetail AS o

ON o.SalesOrderID = h.SalesOrderID --joined SalesOrderDatail with SalesOrderHeader ON SalesOrderID

FULL OUTER JOIN SalesLT.Product AS p

ON p.ProductID = o.ProductID --joined Product with SalesOrderDatail ON ProductID

WHERE o.SalesOrderID IS NULL

Module 4: Using Set Operators

* Use **UNION** to combine the rowsets returned by multiple queries.
* Column **aliases** – must be expressed in first query
* Each **UNION** query must return **the same number of columns** with **compatible data types**.
* By default, **UNION** eliminates duplicate rows. Specify the **UNION** **ALL** option to include duplicates (or to avoid the overhead of checking for duplicates when you know in advance that there are none).
* Use **INTERSECT** to return only rows that are returned by both queries.
* Use **EXCEPT** to return rows from the first query that are not returned by the second query.

Module 4: EXAMPLES:

SELECT customer.CompanyName, address.AddressLine1, address.City, 'Billing' AS AddressType

FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS addressSpareTable

ON customer.CustomerID = addressSpareTable.CustomerID

JOIN SalesLT.Address AS address

ON addressSpareTable.AddressID = address.AddressID

WHERE addressSpareTable.AddressType = 'Main Office';

SELECT customer.CompanyName, address.AddressLine1, address.City, 'Shipping' AS AddressType

FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS addressSpareTable

ON customer.CustomerID = addressSpareTable.CustomerID

JOIN SalesLT.Address AS address

ON addressSpareTable.AddressID = address.AddressID

WHERE addressSpareTable.AddressType = 'Shipping';

SELECT customer.CompanyName, address.AddressLine1, address.City, 'Billing' AS AddressType

FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS addressSpareTable

ON customer.CustomerID = addressSpareTable.CustomerID

JOIN SalesLT.Address AS address

ON addressSpareTable.AddressID = address.AddressID

WHERE addressSpareTable.AddressType = 'Main Office'

UNION

SELECT customer.CompanyName, address.AddressLine1, address.City, 'Shipping' AS AddressType

FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS addressSpareTable

ON customer.CustomerID = addressSpareTable.CustomerID

JOIN SalesLT.Address AS address

ON addressSpareTable.AddressID = address.AddressID

WHERE addressSpareTable.AddressType = 'Shipping'

ORDER BY customer.CompanyName, AddressType;

SELECT customer.CompanyName FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS customerAddress

ON customer.CustomerID = customerAddress.CustomerID

WHERE customerAddress.AddressType = 'Main Office'

EXCEPT

SELECT customer.CompanyName FROM SalesLT.Customer AS customer

JOIN SalesLT.CustomerAddress AS customerAddress

ON customer.CustomerID = customerAddress.CustomerID

WHERE customerAddress.AddressType = 'Shipping';

SELECT customer.CompanyName FROM SalesLT.Customer as customer

JOIN SalesLT.CustomerAddress AS customerAddress

ON customer.CustomerID = customerAddress.CustomerID

WHERE customerAddress.AddressType = 'Main Office'

INTERSECT

SELECT customer.CompanyName FROM SalesLT.Customer as customer

JOIN SalesLT.CustomerAddress AS customerAddress

ON customer.CustomerID = customerAddress.CustomerID

WHERE customerAddress.AddressType = 'Shipping'

Module 5: Using Functions and Aggregating Data

* **Scalar functions** return a single value based on zero or more input parameters.
* **Logical functions** return Boolean values (true or false) based on an expression or column value.
* **Window functions** are used to rank rows across partitions or "windows". Window functions include **RANK**, **DENSE\_RANK**, **NTILE**, and **ROW\_NUMBER**.
* **Aggregate functions** are used to provide summary values for multiple rows - for example, the total cost of products or the maximum number of items in an order. Commonly used aggregate functions include **SUM**, **COUNT**, **MIN**, **MAX**, and **AVG**.
* You can use **GROUP BY** with aggregate functions to return aggregations grouped by one or more columns or expressions.
* All columns in the **SELECT** clause that are not aggregate function expressions must be included in a **GROUP BY** clause.
* The order in which columns or expressions are listed in the **GROUP BY** clause determines the grouping hierarchy.
* You can filter the groups that are included in the query results by specifying a **HAVING** clause.

Module 5: EXAMPLES:

SELECT p.ProductID,

UPPER(p.Name) AS ProductName,

ROUND(p.Weight, 0) AS ApproxWeight,

YEAR(p.SellStartDate) AS SellStartYear,

DATENAME(mm, p.SellStartDate) AS SellStartMonth

FROM SalesLT.Product AS p

SELECT p.ProductID,

UPPER(p.Name) AS ProductName,

ROUND(p.Weight, 0) AS ApproxWeight,

YEAR(p.SellStartDate) AS SellStartYear,

DATENAME(mm, p.SellStartDate) AS SellStartMonth,

LEFT(p.ProductNumber, 2) AS ProductType

FROM SalesLT.Product AS p

SELECT p.ProductID,

UPPER(p.Name) AS ProductName,

ROUND(p.Weight, 0) AS ApproxWeight,

YEAR(p.SellStartDate) AS SellStartYear,

DATENAME(mm, p.SellStartDate) AS SellStartMonth,

LEFT(p.ProductNumber, 2) AS ProductType

FROM SalesLT.Product AS p

WHERE ISNUMERIC(Size) = 1 -- 1=true 0=false

SELECT c.CompanyName, o.TotalDue,

RANK()OVER(ORDER BY o.TotalDue DESC) AS Ranking

FROM SalesLT.Customer AS c

LEFT JOIN SalesLT.SalesOrderHeader AS o

ON c.CustomerID = o.CustomerID

WHERE ISNUMERIC(o.TotalDue) = 1

SELECT p.Name AS ProductName, SUM(o.LineTotal) AS TotalRevenue

FROM SalesLT.SalesOrderDetail AS o

LEFT OUTER JOIN SalesLT.Product AS p

ON p.ProductID = o.ProductID

GROUP BY p.Name

ORDER BY SUM(o.LineTotal) DESC;

SELECT p.Name AS ProductName, SUM(o.LineTotal) AS TotalRevenue, SUM(p.ListPrice) AS SalesTotal

FROM SalesLT.SalesOrderDetail AS o

LEFT OUTER JOIN SalesLT.Product AS p

ON p.ProductID = o.ProductID

WHERE p.ListPrice > 1000

GROUP BY p.Name

ORDER BY SUM(o.LineTotal) DESC;

SELECT p.Name AS ProductName, SUM(o.LineTotal) AS TotalRevenue, SUM(p.ListPrice) AS SalesTotal

FROM SalesLT.SalesOrderDetail AS o

FULL OUTER JOIN SalesLT.Product AS p

ON p.ProductID = o.ProductID

WHERE p.ListPrice > 1000

GROUP BY p.Name

HAVING SUM(o.LineTotal) > 20000

ORDER BY SUM(o.LineTotal) DESC;

Module 6: Using Subqueries and APPLY

* **Subqueries** are Transact-SQL queries nested within an outer query.
* **Scalar subqueries** return a single value.
* Multi-valued subqueries return a single-column rowset.
* Correlated subqueries reference objects in the outer query.
* The **APPLY** operator enables you to execute a table-valued function for each row in a rowset returned by a **SELECT** statement. Conceptually, this approach is similar to a correlated subquery.
* **CROSS APPLY** returns matching rows, similar to an inner join. **OUTER APPLY** returns all rows in the original **SELECT** query results with NULL values for rows where no match was found.

Module 6: EXAMPLES:

Module 7: Using Table Expressions

* Views are database objects that encapsulate **SELECT** queries.
* You can query a view in the same way as a table, however there are some considerations for updating them.
* Temporary tables are prefixed with a # symbol (You can also create global temporary tables that can be accessed by other processes by prefixing the name with ##)
* Local temporary tables are automatically deleted when the session in which they were created ends. Global temporary tables are deleted when the last user sessions referencing them is closed.
* Table variables are prefixed with a @ symbol.
* Table variables are scoped to the batch in which they are created.
* Table-Valued Functions (TVFs) are functions that return a rowset.
* TVFs can be parameterized.
* A derived table is a subquery that generates a multicolumn rowset. You must use the **AS** clause to define an alias for a derived query.
* Common Table Expressions (CTEs) provide a more intuitive syntax or defining rowsets than derived tables, and can be used multiple times in the same query.
* You can use CTEs to define recursive queries.

Module 7: EXAMPLES:

Module 8: Grouping Sets and Pivoting Data

* Use **GROUPING SETS** to define custom groupings.
* Use **ROLLUP** to include subtotals and a grand total for hierarchical groupings.
* Use **CUBE** to include all possible groupings.
* Use **PIVOT** to re-orient a rowset by generating multiple columns from values in a single column.
* Use **UNPIVOT** to re-orient multiple columns in a an existing rowset into a single column.

Module 8: EXAMPLES:

Module 9: Modifying Data

* Use the **INSERT** statement to insert one or more rows into a table.
* When inserting explicit values, you can omit identity columns, columns that allow **NULL**s, and columns on which a default constraint is defined.
* Identity columns generate a unique integer identifier for each row. You can also use a *sequence* to generate unique values that can be used in multiple tables.
* Use the **UPDATE** statement to modify the values of one or more columns in specified rows of a table.
* Use the **DELETE** statement to delete specified rows in a table.
* Use the **MERGE** statement to insert, update, and delete rows in a target table based on data in a source table.

Module 9: EXAMPLES:

Module 10: Programming with Transact-SQL

* A batch defines a group of Transact-SQL command submitted by a client application for execution. Some commands can only be executed at the start of a new batch, and variable values cannot span batches.
* Use comments to document your Transact-SQL code. Inline comments are prefixed by --, and multi-line comment blocks are enclosed in /\* and \*/.
* Declare variables by using the **DECLARE** keyword, specifying a name (prefixed with @) and a data type. You can optionally assign an initial value.
* Assign values to variables by using the **SET** keyword or in a **SELECT** statement.
* Use the IF keyword to execute a task based on the results of a conditional test.
* Use an **ELSE** clause if you need to execute an alternative task if the conditional test returns false.
* Enclose multiple statements in an **IF** or **ELSE** clause between **BEGIN** and **END** keywords.
* Use a **WHILE** loop if you need to repeat one or more statements until a specified condition is true.
* Use **BREAK** and **CONTINUE** to exit or restart the loop.
* Avoid using loops to iteratively update or retrieve single records - in most cases, you should use set-based operations to retrieve and modify data.
* Use stored procedures to encapsulate Transact-SQL code in a reusable database objects.
* You can define parameters for a stored procedure, and use them as variables in the Transact-SQL code it contains.
* Stored procedures can return rowsets (usually the results of a **SELECT** statement). They can also return output parameters, and they always return a *return value*, which is used to indicate status.

Module 10: EXAMPLES:

Module 11: Error Handling and Transactions

* System errors have pre-defined numbers, messages, severity levels, and other characteristics that you can use to troubleshoot issues.
* You can use **RAISERROR** and **THROW** to raise custom errors.
* Use **TRY**...**CATCH** blocks in your Transact-SQL code to catch and handle exceptions.
* A common exception handling pattern is to log the error, and then if the operation cannot be completed successfully, throw it (or a new custom error) to the calling application.
* Transactions are used to protect data integrity by ensuring that all data changes within a transaction succeed or fail as a unit.
* Individual Transact-SQL statements are inherently treated as transactions, and you can define explicit transactions that encompass multiple statements.
* Use the **BEGIN TRANSACTION**, **COMMIT TRANSACTION**, and **ROLLBACK TRANSACTION** statements to manage transactions.
* Enable the **XACT\_ABORT** option to automatically rollback all transactions if an exception occurs.
* Use the **@@TRANCOUNT** system variable and **XACT\_STATE** system function to determine transaction status.

Module 11: EXAMPLES: