# Query Execution Order

|  |  |
| --- | --- |
| **5** | select \* |
| **1** | from table |
| **2** | where [some predicate] |
| **3** | group by [some columns] |
| **4** | having [some group requirement] |
| **6** | order by [some columns] |

# Inner Join

From Table1 t1

Join Table2 t2

on t1.col = t2.col

Inner join grabs only matches from both tables. If we say Customers and their orders, this will NOT get customers who have no orders

# Outer Join

from Table1 t1

left Join Table2 t2

on t1.col = t2.col

All rows from t1 will be returned – all matching rows from t2 will also return. Any non-matching rows will have t2 columns replaced with NULL

from Table2 t2

right Join Table1 t1

on t1.col = t2.col

from Table2 t2

full Join Table1 t1

on t1.col = t2.col

All rows from t1 are returned. All rows from t2 are returned. Any non-matching for each table will be replaced with null

# Cross join

from table1 cross join table2

Every row from table1 combined with every row in table2

# Order By

Can use columns, expressions or case statements

Case use columns not even listed in select clause

Can use column names or aliases – AND IT IS THE ONLY STEP THAT CAN

Can use ordinal position

Specify ASC for ascending, DESC for descending. ASC is optional as it is default

Treats nulls as equal (all at the top for ASC, all at the bottom for DESC)

# Top

select top n

select top n with ties

select top n percent

select top n percent with ties

NOT ANSI STANDARD

Uses the order by clause to limit the number of records returned.

[With Ties] looks at value of the [n]th record and if more records have that same value, they are also retrieved.

# Offset Fetch

Offset n rows

Fetch next n rows only

Used to page through a result set. ANSI STANDARD. Works with an order by.

# DML

## Insert

insert into table1 (*column\_destination\_list*)

values (*column\_values\_list*)

If you are supplying a value for every column in the table except for Identity column, the destination list can be omitted

insert into table1

values (*column\_values\_list*)

For multiple inserts at once

insert into table1 (*column\_destination\_list*)

values (*column\_values\_list*), (column\_values\_list), (column\_values\_list)

or if you are supplying a value for every column except for identity column

insert into table1

values (*column\_values\_list*), (*column\_values\_list*), (*column\_values\_list*)

For 1 or more inserts at once when the records are in another location

insert into table1 (*column\_destination\_list*)

*select statement*

or if you are supplying a value for every column except for identity column

insert into table1

*any select statement*

For 1 or more inserts at once when the records are supplied as the output of a stored procedure

insert into table1

*exec stored\_proc\_name*

## Update

Update table

Set column\_name1 = value, column\_name2 = value

Where {condition}

ALWAYS START YOUR UPDATE AS A SELECT. MAKE SURE YOU HAVE THE EXACT RECORDS YOU WANT TO UPDATE

THEN COMMENT OUT THE SELECT AND ADD THE UPDATE AND SET

Update t1

Set column\_name1 = value, column\_name2 = value

-- select blah blah

From table1 t1

Join table t2

On {join condition}

Where {filter condition}

## Merge

MERGE

INTO Prodtable AS Destination

USING ImportData AS StagingTable

ON (join\_condition)

WHEN NOT MATCHED THEN

INSERT (column\_destination\_list)

VALUES (column\_values\_list)

WHEN MATCHED THEN

update\_statement

## Delete

Delete table

Where\_clause

Delete

-- select

From table1 t1

Join table t2

On {join condition}

Where {filter condition}

# Grouping and Aggregating Data

## Aggregate Functions

Aggregate functions usually occur with a group by clause – if they do not, the entire table is considered a group and the functions are executed against that one group

Distinct works within the aggregate function to apply to only the unique values in the column

SELECT empid, YEAR(orderdate) AS orderyear,

COUNT(custid) AS count\_of\_all\_customers,

COUNT(DISTINCT custid) AS count\_of\_only\_unique\_customers

FROM Sales.Orders

GROUP BY empid, YEAR(orderdate);

## Group by

Is placed after the where clause and operates on data gathered by the where clause to create groups of data – one for each time a difference is detected amongst the columns listed in the group by clause.

When there are aggregate functions in a query – ALL COLUMNS must be aggregate functions unless that column is listed in the group by clause.

To use a group by:

1. Figure out what you want and build it in the select clause
2. Figure out the tables and joins needed to get to that data
3. Any columns in the select clause that are not aggregate functions must be added to the group by clause

## Having Clause

Having clause is placed just after the group by and simply removes all groups that don’t meet the condition laid out in the having clause

SELECT p.productid, COUNT(\*) AS cnt

FROM Production.Products AS p JOIN Sales.OrderDetails AS od ON p.productid = od.productid

GROUP BY p.productid

HAVING COUNT(\*) >= 10;

This query brings back the count of all products that appear on 10 or more orders

# Subqueries

## Self-contained Subqueries

Select \*

From table1

Where field in (select field

from table2)

Self-contained subqueries can be ran all on their own because they do no reference containing query

## Correlated Subqueries

Select \*

From table1 outer

Where column = (select field

from table2 inner

where inner.column = outer.column)

Inner query acts like it is running once per each outer row. Inner query accepts a value from the outer query. Inner query cannot

## Exists Predicate

Select \*

From table1 outer

Where not exists (select \*

From table2 inner

where inner.column = outer.column)

Exists functions as an existence test (or lookup). The inner query doesn’t actually return anything, it merely return whether or not it WOULD HAVE returned something.

# Table Expressions

## Views

create view {schema}.{viewname}

as

{some select statement}

Views are select queries that are stored in the database and act just like a table to those using them.

To use them you simply create a query and references the view as if it was a table.

Select \*

From {scheme}.{viewname}

Order by {order\_column}

Order by is not allowed in view unless you are using Top or Offset Fetch OR FOR XML

Views can’t take parameters

By default, views are readonly

## Inline Table-valued Functions

Create|alter function {schema}.{fn\_functionname} [(@parameter1 datatype, @parameter2…)]

returns table

as

return

{some select statement}

TVF’s are considered to be parameterized views. They are virtual the same, with the same rules, but can have parameters which are set as local to the function when they are supplied.

## Derived Tables

Select \*

From table1 t1

Join (select {column\_list}

From table2 t2

Left join table3 t3

On {join\_condition}) as alias

On t1.column = alias.column

Derived tables are useful when the data you need to retrieve comes from data retrieved by different query which must execute first.

The inner query is placed in parenthesis.

All columns must have aliases

All columns must have unique aliases

## Common Table Expressions

With alias as (

select {column\_list}

From table2 t2

Left join table3 t3

On {join\_condition}

)

Select \*

From table1 t1

Join alias t2

On t1.column = t2.column

CTE’s are basically the same as derived tables, except they make the experience a little cleaner. Whereas with derived tables, you declare you sub table within the original query, CTE’s are declared above the query making a clean separation of the two steps required to be executed.

# Set Operators

## Union

Union takes two data compatible result sets and combines them together.

SELECT country, region, city FROM HR.Employees

UNION

SELECT country, region, city FROM Sales.Customers;

Two queries must have the same number of columns and they must be of compatible data types (int vs decimal is okay, varchar(50) vs varchar(100) is okay)

Union will ALWAYS filter out duplicates after combining the two result sets. If you know you don’t have duplicates, this is not an optimized way to perform this task.

## Union All

Union All is the same as Union, but it does NOT perform the task of removing duplicates.

SELECT country, region, city FROM HR.Employees

UNION

SELECT country, region, city FROM Sales.Customers;

## Intersect

Intersect basically says keep all those from the first result set, but only if they are ALSO found in the second result set

SELECT country, region, city FROM HR.Employees

Intersect

SELECT country, region, city FROM Sales.Customers;

## Except

Except says the opposite. Keep all those from the first result set, but only if they are NOT found in the second result set

SELECT country, region, city FROM HR.Employees

Except

SELECT country, region, city FROM Sales.Customers;

# Windowing

## Over Clause

Over clause is the command to examine the table just for the purposes of calculating one column.

Within the over you can have:

Order by – sets up a sort order just for the purposes of calculating this column without respect to the final sort order

Partition by – set up a group by. A condition for when the aggregate function listed is reset

Rows – allows you to specify how the aggregate function is reset and what value ends up being placed in each row

### Examples

SELECT Category, Qty, Orderyear,

SUM(Qty) OVER (

PARTITION BY category

ORDER BY orderyear

ROWS BETWEEN UNBOUNDED PRECEDING

AND CURRENT ROW) AS RunningQty

FROM Sales.CategoryQtyYear;

Orders the table by orderyear and then sums the qty as a running total, resetting at each category

SELECT custid, ordermonth, qty,

SUM(qty) OVER(PARTITION BY custid)

AS totalpercust

FROM Sales.CustOrders;

Sums the qty for each customer

SELECT productid, productname, unitprice,

RANK() OVER(ORDER BY unitprice DESC) AS pricerank

FROM Production.Products

ORDER BY pricerank;

Ranks the prices from high to low. Any ties will have equal ranking, but the count will continue so you will have gaps. For instance, if you have 4 rows that are tied for 4th place, they will all receive a rank of 4 and the next row will be 8.

SELECT productid, productname, unitprice,

DENSE\_RANK() OVER(ORDER BY unitprice DESC) AS pricerank

FROM Production.Products

ORDER BY pricerank;

Ranks the prices from high to low. Any ties will have equal ranking, but the count will not continue so you will not have gaps. For instance, if you have 4 rows that are tied for 4th place, they will all receive a rank of 4 and the next row will be 5.

SELECT productid, productname, unitprice,

ROWNUMBER() OVER(ORDER BY unitprice DESC) AS pricerank

FROM Production.Products

ORDER BY pricerank;

Ranks the prices from high to low. No ties and no gaps.

SELECT productid, productname, unitprice,

NTILE(7) OVER(ORDER BY unitprice DESC) AS pricerank

FROM Production.Products

ORDER BY pricerank;

Splits the given records into n groups (in this example, seven) as evenly as it can

SELECT employee, orderyear ,totalsales AS currsales,

LEAD (totalsales, 1,0) OVER (PARTITION BY employee

ORDER BY orderyear) AS nextsales

FROM Sales.OrdersByEmployeeYear

ORDER BY employee, orderyear;

Shows a value from the NEXT partition within the current partition – potentially disrupting the space/time continuum.

SELECT employee, orderyear ,totalsales AS currsales,

LAG (totalsales, 1,0) OVER (PARTITION BY employee

ORDER BY orderyear) AS nextsales

FROM Sales.OrdersByEmployeeYear

ORDER BY employee, orderyear;

Goes back in time and grabs a value from the PREVIOUS partition and places it in the current position.

# Pivot

SELECT Category, [2006],[2007],[2008]

FROM (SELECT Category, Qty, Orderyear

FROM Sales.CategoryQtyYear) AS D

PIVOT(SUM(QTY) FOR orderyear

IN ([2006],[2007],[2008])) AS pvt

ORDER BY Category;

Pivot allows you to take data that is vertically oriented and spread the distinct values of a specific column across multiple columns making it horizontally oriented and more concise.

In this example, we take the qty column and add them all up for each year listed. We add that year as a column and then place the summed value into that column. Pivot will always result in LESS ROWS, but MORE COLUMNS.

# Grouping Sets

SELECT Category, Cust, SUM(Qty) AS TotalQty

FROM Sales.CategorySales

GROUP BY

GROUPING SETS((Category),(Cust),());

Grouping sets allow you to run an aggregate function with more than one group by. They will each run in turn and each result will be appended to the final results. The above code groups by Category, by Customer and by nothing (which sums the entire table)

## Cube

SELECT Category, Cust, SUM(Qty) AS TotalQty

FROM Sales.CategorySales

GROUP BY CUBE(Category,Cust)

ORDER BY Category, Cust;

Cube accomplishes the same thing as Grouping Sets and could be looked at as a shortcut. However, Cube computes all combinations of the listed values including a grand total: Category, Cust, Category+Cust, Grand Total

## Rollup

SELECT Category, Cust, SUM(Qty) AS TotalQty

FROM Sales.CategorySales

GROUP BY ROLLUP(Category,Cust)

ORDER BY Category, Cust;

Rollup is the same as Cube except it assumes that there is a hierarchical relationship between Category and Cust so it will not have all the same sums

# Stored Procs

Create proc {schema}.{procname} (parameter\_list}

As

…

Stored proc can:

* Insert
* Update
* Delete
* All three
* Return multiple result sets
* Return output parameters
* Perform administrative functions

## Parameters

Parameters can be input (no need to specify) or output…must be specified

(@myvar int ouput)

When you call the stored Proc

Exec|Execute {schema}.{procname} {any required parateters}

If you are sending output parameters, you MUST use the word output as you send it.

Exec {schema}.{procname} @myoutputvar ***output***

## Optional parameters

Simply give your parameter a value.

If passed, the value will be your value. If not passed, the value will be the default

(@myvar int = 5000)

# Delicious Links

My repository for all things demo… There are many repositories there, but most of them are programming/Cloud(Azure) related

<https://github.com/CentriqTraining/TSQL>

A learning platform which will eventually have TSQL content available in video form to reinforce what we talked about in this class

<https://docs.microsoft.com/en-us/learn/browse/>

Thousands of IT related videos

<https://channel9.msdn.com/>

Download Sql Server Express edition

<https://www.microsoft.com/en-us/sql-server/sql-server-2017?&OCID=AID739534_SEM_fkhHI4O5&MarinID=sfkhHI4O5_258104145771_sql%20server_e_c__57166984510_aud-411628295034:kwd-14998960_>

Download SSMS Express

<https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-2017&OCID=AID739534_SEM_eBg0sKPb&MarinID=seBg0sKPb_340829462784_%2Bssms_b_c__68566309356_kwd-305276309241_&viewFallbackFrom=sql-server-2017%3F>

Your courseware

<https://skillpipe.courseware-marketplace.com/>

Yours course content – (D Drive)

<https://www.microsoft.com/en-us/learning/companion-moc.aspx>