SMALLDATETIMEFROMPARTS()	SMALLDATETIMEFROMPARTS(year, month, day, hour, minute)	smalldatetime
TIMEFROMPARTS()	TIMEFROMPARTS(hour, minute, seconds, fractions, precision)	time

```
SELECT DATEFROMPARTS(2012,2,12);
SELECT DATETIME2FROMPARTS(2012,2,12,8,30,0,0,0);
```

Functions that modify date and time values:

Function	Syntax	Remarks
DATEADD()	DATEADD(datepart, interval, date)	Adds interval to date, returns same datatype as date
EOMONTH()	EOMONTH(start_date, interval)	Returns last day of month as start date, with optional offset
SWITCHOFFSET()	SWITCHOFFSET(datetimeoffset, time_zone)	Changes time zone offset
TODATETIMEOFFSET()	TODATETIMEOFFSET(expression, time_zone)	Converts datetime2 into datetimeoffset

```
SELECT DATEADD(day,1,'20120212');
SELECT EOMONTH('20120212');
```

Functions that operate on date and time values:

Function	Syntax	Remarks
DATEDIFF()	DATEDIFF(datepart, start_date, end_date)	Returns the number of boundaries crossed for the specified datepart
ISDATE()	ISDATE(expression)	Determines whether a datetime or smalldate time is a valid value

Converting with cast:

Converts a value from one data type to another

- Can be used in SELECT and WHERE clauses
- ANSI standard
- Truncation can occur if converting to smaller data type

CAST Example:

```
SELECT CAST(SYSDATETIME() AS date) AS 'TodaysDate';
```

Returns an error if data types are incompatible:

```
--attempt to convert datetime2 to int
SELECT CAST(SYSDATETIME() AS int);
```

Converting with convert:

Converts a value from one data type to another

- Can be used in SELECT and WHERE clauses
- CONVERT is specific to SQL Server, not standards-based

Style specifies how input value is converted:

- Date, time, numeric, XML, etc.

Example:

```
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP,112) AS ISO_style;
```

SQL Server 2012 built-in function types:

Function Category	Description
Scalar	Operate on a single row, return a single value
Grouped Aggregate	Take one or more values but 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 T-SQL statement

— year()
— MAX()

So is ROWSET a inline table valued function (table expression) ??

Scalar functions:

Scalar function categories

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

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 expressionin queries
May be deterministic or non-deterministic
Collation depends on input value or default collation of database

```
SELECT SalesOrderID, YEAR(OrderDate) AS OrderYear
FROM Sales.SalesOrderHeader;
SELECT ABS(-1.0), ABS(0.0), ABS(1.0);
SELECT CAST(SYSDATETIME() AS date);
SELECT DB_NAME() AS current_database;
```

Window functions:

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

→ No use of PARTITION means the complete ROWSET is a partition (or a group)

```
SELECT TOP(5) ProductID, Name, ListPrice, RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM Production.Product
ORDER BY RankByPrice;
```

749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1

and 'RANK'

What is the difference using ROW_NUMBER with / order by & with / partition by

```
select ROW_NUMBER() over (order by SalesOrderDetailID) as rownum1
 ,*
from Sales.SalesOrderDetail as t
order by SalesOrderDetailID

select ROW_NUMBER() over (order by SalesOrderDetailID) as rownum1
 ,      ROW_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1
 ,*
from Sales.SalesOrderDetail as t
order by SalesOrderDetailID
```

```
select ROW_NUMBER() over (order by SalesOrderDetailID) as rownum1
 ,      ROW_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1
 , SUM(UnitPrice) over (partition by salesorderid ) as totalorderprice
 ,*
from Sales.SalesOrderDetail as t
order by SalesOrderDetailID

select ROW_NUMBER() over (order by SalesOrderDetailID) as rownum1
 ,      ROW_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1
 , SUM(UnitPrice) over (partition by salesorderid ) as totalorderprice
 , SUM(UnitPrice) over () as totaltotalorderprice
 ,*
from Sales.SalesOrderDetail as t
order by SalesOrderDetailID

SELECT TOP(5) ProductID, Name, ListPrice, RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM Production.Product
ORDER BY RankByPrice;
```

Writing logical tests with functions:

ISNUMERIC tests whether an input expression is a valid numeric data type

Returns a 1 when the input evaluates to any valid numeric type, including FLOAT and MONEY, otherwise returns 0

```
SELECT ISNUMERIC('SQL') AS isnmumeric_result;
SELECT ISNUMERIC('101.99') AS isnmumeric_result;
```

Performing conditional tests with IIF:

IIF returns one of two values, depending on a logical test

Shorthand for a two-outcome CASE expression

IIF Element	Comments
Boolean_expression	Logical test evaluating to TRUE, FALSE, or UNKNOWN
True_value	Value returned if expression evaluates to TRUE
False_value	Value returned if expression evaluates to FALSE or UNKNOWN

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

Selecting items from a list with CHOOSE:

CHOOSE returns an item from a list as specified by an index value

CHOOSE example:

```
SELECT CHOOSE (3, 'Beverages', 'Condiments', 'Confections') AS choose_result;
```

Aggregate Functions:

- SUM
- MIN
- MAX
- AVG
- COUNT
- COUNT_BIG
- STDEV
- STDEVP
- VAR
- VARP
- CHECKSUM_AGG
- GROUPING
- GROUPING_ID

Return a scalar value (with no column name)

Ignore NULLs except in COUNT(*)

Can be used in

SELECT, HAVING, and ORDER BY clauses

not in 'WHERE'

Frequently used with GROUP BY clause

```
SELECT COUNT (DISTINCT SalesOrderID) AS UniqueOrders,
AVG(UnitPrice) AS Avg_UnitPrice, MIN(OrderQty)AS Min_OrderQty, MAX(LineTotal) AS Max_LineTotal
FROM Sales.SalesOrderDetail;
```

Using DISTINCT with aggregate functions:
Use DISTINCT with aggregate functions to summarize only unique values
DISTINCT aggregates eliminate duplicate values, not rows (unlike SELECT DISTINCT)
Compare (with partial results):

```
SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear,
COUNT(CustomerID) AS All_Custs,
COUNT(DISTINCT CustomerID) AS Unique_Custs
FROM Sales.SalesOrderHeader
GROUP BY SalesPersonID, YEAR(OrderDate);

--all orders
select * from Sales.SalesOrderHeader
--now i want to get maximum of 1 order per customer??
--first how many customers do i have
select COUNT(distinct CustomerID) from Sales.SalesOrderHeader--returns 19119
--so i should have 19119 orders returned as a result of actual query to be answered
--group by would not work..
--select * from Sales.SalesOrderHeader
--group by CustomerID
--we need all the order details...not only the customer id
select CustomerID from Sales.SalesOrderHeader
group by CustomerID
--i can't use distinct as it will use ditinct over the entrier row. But i can use this row_number thing
for this problem.
select
ROW_NUMBER() over (partition by customerID order by OrderDate desc) as rn
,*
from Sales.SalesOrderHeader

--i cant use this as rn column name is not avaiable at the time of execution of where clause
--select
--ROW_NUMBER() over (partition by customerID order by OrderDate desc) as rn
--,*
--from Sales.SalesOrderHeader
--where rn =1

--now i cant do this as all the ranking functions are calculated at select time
--select
--ROW_NUMBER() over (partition by customerID order by OrderDate desc) as rn
--,*
--from Sales.SalesOrderHeader
--where ROW_NUMBER() over (partition by customerID order by OrderDate desc) =1

--but this would work...u can compare the count of rows returned and it would be 19119
select * from(
	select
	ROW_NUMBER() over (partition by customerID order by OrderDate desc) as rn
	,*
	from Sales.SalesOrderHeader
) as a
where rn=1;

--now lets say i want to get all orders that are made in the same day. now if they are made in the same
day, they will get different row_numbers
--but if i use rank, then all the order with the same order date will come back with same rank..this
will return more that 19119 rows
select * from(
	select
	rank() over (partition by customerID order by OrderDate desc) as rn
	,*
	from Sales.SalesOrderHeader
) as a
where rn=1;
```

'GROUP BY' does not preserve the row level details but PARTITION' does

They are used for similar problems

derived table expression.

Group by clause:

GROUP BY creates groups for output rows, according to unique combination of values specified in the GROUP BY clause

```
SELECT <select_list>
FROM <table_source>
WHERE <search_condition>
GROUP BY <group_by_list>;
```

GROUP BY calculates a summary value for aggregate functions in subsequent phases

```
SELECT SalesPersonID, COUNT(*) AS Cnt
FROM Sales.SalesOrderHeader
GROUP BY SalesPersonID;
```

Detail rows are "lost" after GROUP BY clause is processed

HAVING, SELECT, and ORDER BY must return a single value per group

All columns in SELECT, HAVING, and ORDER BY must appear in GROUP BY clause or be inputs to aggregate expressions

If a query uses GROUP BY, all subsequent phases operate on the groups, not source rows

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

If a query uses group by, then all subsequent phases operate on the group, not on the source rows

Using GROUP BY with aggregate functions:

Aggregate functions are commonly used in SELECT clause, summarize per group:

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

Aggregate functions may refer to any columns, not just those in GROUP BY clause

```
SELECT productid, MAX(OrderQty) AS largest_order
FROM Sales.SalesOrderDetail
GROUP BY productid;
```

Filtering grouped data using HAVING Clause:

HAVING clause provides a search condition that each group must satisfy

HAVING clause is processed after GROUP BY

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

Compare HAVING to WHERE clauses:

WHERE filters rows before groups created

Controls which rows are placed into groups

HAVING filters groups

Controls which groups are passed to next logical phase

Using a COUNT(*) expression in HAVING clause is useful to solve common business problems:

Show only customers that have placed more than one order:

```
SELECT Cust.Customerid, COUNT(*) AS cnt
FROM Sales.Customer AS Cust
JOIN Sales.SalesOrderHeader AS Ord ON Cust.CustomerID = ORD.CustomerID
GROUP BY Cust.CustomerID
HAVING COUNT(*) > 1;
```

Show only products that appear on 10 or more orders:

```
SELECT Prod.ProductID, COUNT(*) AS cnt
FROM Production.Product AS Prod
JOIN Sales.SalesOrderDetail AS Ord ON Prod.ProductID = Ord.ProductID
```

```
GROUP BY Prod.ProductID
HAVING COUNT(*) >= 10;
```

Working with subqueries:

Subqueries are nested queries or queries within queries
Results from inner query are passed to outer query
    Inner query acts like an expression from perspective of outer query
Subqueries can be self-contained or correlated
    Self-contained subqueries have no dependency on outer query
    Correlated subqueries depend on values from outer query
Subqueries can be scalar, multi-valued, or table-valued

Subquery is different from derived table expressions. Derived table expressions go in the 'FROM' clause

Scalar subquery returns single value to outer query
Can be used anywhere single-valued expression can be used: SELECT, WHERE, etc.

```
SELECT SalesOrderID, ProductID, UnitPrice, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID =
(SELECT MAX(SalesOrderID) AS LastOrder
FROM Sales.SalesOrderHeader);
```

If inner query returns an empty set, result is converted to NULL
Construction of outer query determines whether inner query must return a single value

Rowset & Column set

Multi-valued subquery returns multiple values as a single column set to the outer query
Used with IN predicate
    If any value in the subquery result matches IN predicate expression, the predicate returns TRUE

```
SELECT CustomerID, SalesOrderId,TerritoryID
FROM Sales.SalesorderHeader
WHERE CustomerID IN (
SELECT CustomerID
FROM Sales.Customer
WHERE TerritoryID = 10);
```

May also be expressed as a JOIN (test both for performance)

The keyword EXISTS does not follow a column name or other expression.
The SELECT list of a subquery introduced by EXISTS typically only uses an asterisk (*).

```
SELECT CustomerID, PersonID
FROM Sales.Customer AS Cust
WHERE EXISTS (
SELECT *
FROM Sales.SalesOrderHeader AS Ord
WHERE Cust.CustomerID = Ord.CustomerID);

SELECT CustomerID, PersonID
FROM Sales.Customer AS Cust
WHERE NOT EXISTS (
SELECT *
FROM Sales.SalesOrderHeader AS Ord
WHERE Cust.CustomerID = Ord.CustomerID);


--in a complex query, the derived table expressions can become unreadable. so then use common table
expressions.
--it is just declaring the expression before hand
with a as(
        select
        rank() over (partition by customerID order by OrderDate desc) as rn
        ,*
        from Sales.SalesOrderHeader
)

select * from  a
where rn=1
```

```sql
--find number of orders per customer
select
CustomerID, COUNT(*) as ordercnt
from Sales.SalesOrderHeader
group by CustomerID

--here as well we have repeat the expression
select
	case
		when RevisionNumber = 3 then 'ok'
		else 'not ok'
	end as status
	, count(*) as ordercnt
from Sales.SalesOrderHeader
group by case
		when RevisionNumber = 3 then 'ok'
		else 'not ok'
	end;

with a as (
select
	case
		when RevisionNumber = 3 then 'ok'
		else 'not ok'
	end as status
from Sales.SalesOrderHeader
)

select status,count(*)  from a
group by status
```

Views:

Views are saved queries created in a database by administrators and developers

Views are defined with a single SELECT statement

ORDER BY is not permitted in a view definition without the use of TOP, OFFSET/FETCH, or FOR XML

To sort the output, use ORDER BY ~~in the outer query~~

View creation supports additional options beyond the scope of this class

```sql
CREATE VIEW HumanResources.EmployeeList
AS
	SELECT BusinessEntityID, JobTitle, HireDate, VacationHours
		FROM HumanResources.Employee;
go

SELECT * FROM HumanResources.EmployeeList
go
```

Creating simple inline table-valued functions:

Table-valued functions are created by administrators and developers

Create and name function and optional parameters with CREATE FUNCTION

Declare return type as TABLE

Define inline SELECT statement following RETURN

```sql
CREATE FUNCTION Sales.fn_LineTotal (@SalesOrderID INT)
RETURNS TABLE
AS
RETURN
    SELECT SalesOrderID,
    CAST((OrderQty * UnitPrice * (1 - SpecialOfferID))
    AS DECIMAL(8, 2)) AS LineTotal
    FROM    Sales.SalesOrderDetail
    WHERE   SalesOrderID = @SalesOrderID ;
```

Writing queries with derived tables:

Derived tables are named query expressions created within an outer SELECT statement

Not stored in database – represents a virtual relational table

When processed, unpacked into query against underlying referenced objects

Allow you to write more modular queries

even a 'select' view can be used to make inserts or updates from an outer query !!

↗ and inline table-value function.

use 'SCHEMABINDING' with view & 'with check option' as another option & 'encryption'

→ a type of table expressions

① Derived table expr.
② CTE
③ Views
④ Inline table valued function

Think of them as views with parameters.

Derived table expr. is actually a ~~derived~~ table subquery.

```
SELECT <column_list>
FROM   (
       <derived_table_definition>
       ) AS <derived_table_alias>;
```

Scope of a derived table is the query in which it is defined

Derived Tables Must	Derived Tables May
• Have an alias • Have names for all columns • 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	• Use internal or external aliases for columns • Refer to parameters and/or variables • Be nested within other derived tables

Passing arguments to derived tables:
Derived tables may refer to arguments
Arguments may be:
- Variables declared in the same batch as the SELECT statement
- Parameters passed into a table-valued function or stored procedure

Same as views

Why would you prefer table-valued function over a stored procedure?

That is because they are used in place of tables & tables are sets (un-ordered)

```
DECLARE @emp_id INT = 9;
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
       SELECT YEAR(orderdate) AS orderyear, custid
       FROM Sales.Orders
       WHERE empid=@emp_id
) AS derived_year
GROUP BY orderyear;
```

Creating queries with common table expressions:
Use WITH clause to create a CTE:
- Define the table expression in WITH clause
- Reference the CTE in the outer query
- Assign column aliases (inline or external)
- Pass arguments if desired

```
WITH CTE_year AS
(
SELECT YEAR(OrderDate) AS OrderYear, customerID
FROM Sales.SalesOrderHeader
)
SELECT orderyear, COUNT(DISTINCT CustomerID) AS CustCount
FROM CTE_year
GROUP BY OrderYear;
```

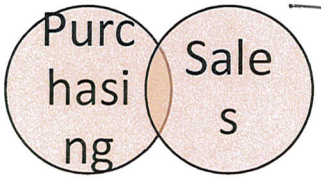
Why would you use a view vs a table valued function?? Table valued function is just a view that takes parameters

SET operators (UNION, INTERSECT, EXCEPT, APPLY):
The results of two input queries may be combined, compared, or operated against each other
Both sets must have the same number of compatible columns
ORDER BY not allowed in input queries, but may be used for result of set operation
NULLs considered equal when comparing sets
SET operators include UNION, INTERSECT, EXCEPT, and APPLY

```
<SELECT query_1>
<set_operator>
<SELECT query_2>
[ORDER BY <sort_list>]
```

Apply is set operator!!
A table is also a set!
So Apply is also called a table operator just like 'join'.

UNION returns a result set of distinct rows combined from both sides



or unique rows

Duplicates removed during query processing (affects performance)

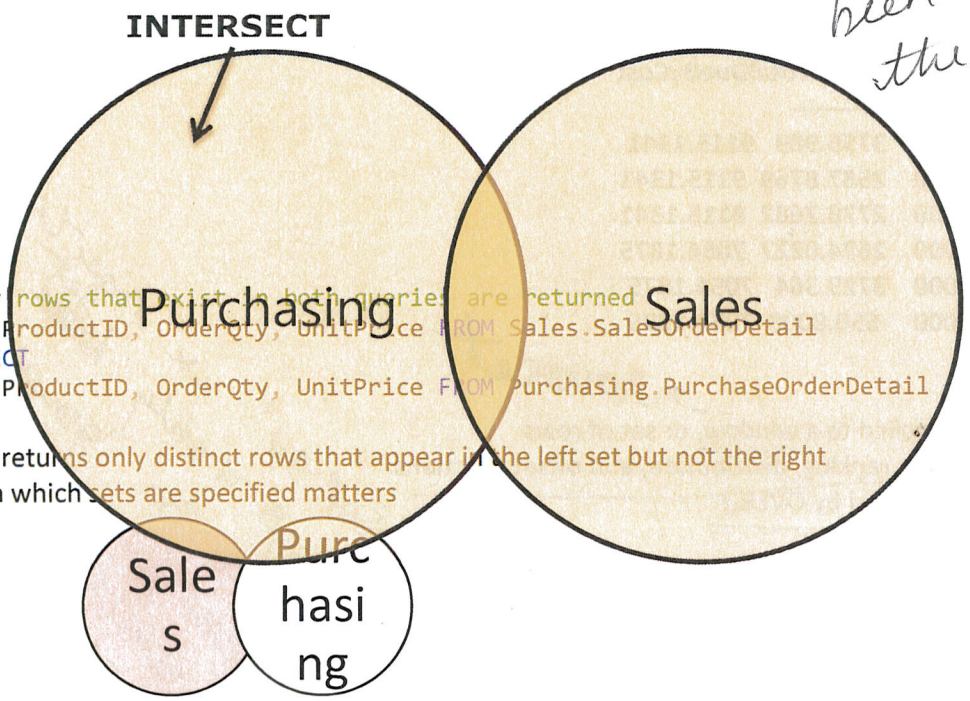
```
-- only distinct rows from both queries are returned
SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail
UNION
SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail
```

UNION ALL returns a result set with all rows from both sets

To avoid performance penalty, use UNION ALL even if you know there are no duplicates

```
-- all rows from both queries are returned
SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail
UNION ALL
SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail
```

INTERSECT returns only distinct rows that appear in both result sets → should it not have been that it returns the common rows




```
-- only rows that exist in both queries are returned
SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail
INTERSECT
SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail
```

EXCEPT returns only distinct rows that appear in the left set but not the right
Order in which sets are specified matters


```
-- only rows from Sales are returned
SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail
EXCEPT
SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail
```

→ just like JOIN

APPLY is a table operator used in the FROM clause and can be either a CROSS APPLY or OUTER APPLY
Operates on two input tables, left and right
Right table is often a derived table or a table-valued function → table expression
OUTER APPLY is similar to LEFT OUTER JOIN between two tables

1. OUTER APPLY applies the right table expression to each row in left table → cross apply does not do this
2. OUTER APPLY adds rows for those with NULL in columns for right table

CROSS APPLY is like CROSS JOIN
INNER JOIN

```
SELECT <column_list>
FROM   <left_table> AS <alias>
CROSS/OUTER APPLY
   <derived_table_expression or inline_TVF> AS <alias>
```

```
--show all customers and last 5 order for each customer..we can use row_number or apply...lets use apply
--apply would execute once for each row in the outer set
select
c.CustomerID,
c.AccountNumber,
```

OUTER APPLY is like LEFT OUTER JOIN

```
o.*
from Sales.Customer as c
outer apply(
select top(5) * from Sales.SalesOrderHeader as sho
where sho.CustomerID = c.CustomerID
order by sho.OrderDate desc
)as o
where c.TerritoryID =3
```

SQL windowing:
Windows extend T-SQL's set-based approach
Windows allow you to specify an order as part of a calculation, without regard to order of input or final output order
Windows allow partitioning and framing of rows to support functions
Window functions can simplify queries that need to find running totals, moving averages, or gaps in data
Partitioning windows:
Partitioning limits a set to rows with same value in the partitioning column
Use PARTITION BY in the OVER() clause
Without a PARTITION BY clause defined, OVER() creates a single partition of all rows

```
SELECT  CustomerID, OrderDate, TotalDue,
     SUM(TotalDue) OVER(PARTITION BY CustomerID)
AS TotalDueByCust
FROM Sales.SalesOrderHeader;
```

CustomerID	OrderDate	TotalDue	TotalDueByCust
11000	2007-08-01 00:00:00.000	3756.989	9115.1341
11000	2007-10-01 00:00:00.000	2587.8769	9115.1341
11000	2006-09-01 00:00:00.000	2770.2682	9115.1341
11001	2007-08-01 00:00:00.000	2674.0227	7054.1875
11001	2006-11-01 00:00:00.000	3729.364	7054.1875
11001	2007-04-01 00:00:00.000	650.8008	7054.1875

Defining window functions:
A windows function is a function applied to a window, or set of rows
Window functions include aggregate, ranking, distribution, and offset functions
Window functions depend on set created by OVER()
Windows aggregate functions:
- Similar to grouped aggregate functions such as SUM, MIN, MAX, etc.
- Applied to windows defined by OVER clause
- Support partitioning, ordering, and framing

Window ranking functions:
Ranking functions require a windows order clause
Partitioning is optional
To display results in sorted order still requires ORDER BY!

Function	Description
RANK	Returns the rank of each row within the partition of a result set. May include ties and gaps.
DENSE_RANK	Returns the rank of each row within the partition of a result set. May include ties but will not include gaps.
ROW_NUMBER	Returns a unique sequential row number within partition based on current order.
NTILE	Distributes the rows in an ordered partition into a specified number of groups. Returns the number of the group to which the current row belongs.

Window offset functions:
Window offset functions allow comparisons between rows in a set without the need for a self-join
Offset functions operate on an position relative to the current row, or to the start or end of the window frame

Function	Description
LAG	Returns an expression from a previous row that is a defined offset from the current row. Returns NULL if no row at specified position.