Homework 8

[**1. Describe different DB models.**](#_qs6g8bsp5db9) **2**

[Relational model](#_5834wd9on843) 2

[Hierarchical model](#_hotmnnfh6d73) 2

[Network model](#_jei305phr9z5) 2

[Object-oriented database model](#_y7otqmucyukn) 2

[Object-relational model](#_obv2dk4d2ca1) 3

[Entity-relationship model](#_baxt1dofuxoz) 3

[Other database models](#_pyi4eh77pwo3) 3

[Inverted file model](#_pmo0pjla8fsr) 3

[Flat model](#_g3icc5am428e) 3

[Multidimensional model](#_krkq31umwrfu) 3

[Semi Structured model](#_m77qdlpmbcx7) 4

[Context model](#_261513n0zbvm) 4

[Associative model](#_6vy4hn67l16w) 4

[NoSQL database models](#_nwxrgn9870yo) 4

[Databases on the Web](#_l8m7djile9bd) 5

[**2. Describe DB normalization, connection between tables.**](#_ozmbiuv8y53q) **5**

[Data normalization](#_jssrt9tvqj0) 5

[Data Normalization Forms](#_rg76wh3zs013) 6

[1NF - First Normal Form](#_svkzybgd4vdb) 7

[2NF - Second Normal Form](#_5vt24ebmz4wn) 8

[3NF - Third Normal Form](#_8p14ypfv6fgl) 9

[BCNF - Boyce and Codd Normal Form](#_lgss39nyug8i) 11

[One-to-many relationships](#_sc8ngjzeb6lm) 12

[Many-to-many relationships](#_yjtapxpl1efd) 12

[One-to-one relationships](#_30kzgtfb2fn1) 13

[**3. Provide sql queries to create/drop/delete tables, join tables with different kinds of join.**](#_l1c8r8v0gnqt) **13**

[**4. Provide sql queries to check presence of duplicates in a table, queries with aggregating and window functions.**](#_1eqo5c7ip58f) **18**

[How to Find Duplicate Values in SQL](#_oar5jbaajl42) 18

[Duplicate Values in One Column](#_watlayewydcq) 18

[Duplicate Values in Multiple Columns](#_wwaqy9u743im) 20

[Unique Values](#_3yxkw6ep4i82) 22

[Grouping queries](#_xpxycko6qlcm) 22

[Aggregate calculations](#_c7hxnxr2z40o) 22

[Ranking Functions](#_10vjx3av58yz) 24

[Window Aggregates](#_ht1lb98tg1z1) 29

[Window Aggregate Enhancements in 2012](#_y52an0j6xp1g) 30

[Offset Functions](#_ii4y5vkrhjb8) 32

[Statistical Functions](#_y0o38k7alsw0) 35

# **1. Describe different DB models.**

## Relational model

The most common model, the relational model sorts data into tables, also known as relations, each of which consists of columns and rows. Each column lists an attribute of the entity in question, such as price, zip code, or birth date. Together, the attributes in a relation are called a domain. A particular attribute or combination of attributes is chosen as a primary key that can be referred to in other tables, when it’s called a foreign key.

Each row, also called a tuple, includes data about a specific instance of the entity in question, such as a particular employee.

The model also accounts for the types of relationships between those tables, including one-to-one, one-to-many, and many-to-many relationships.

## Hierarchical model

The hierarchical model organizes data into a tree-like structure, where each record has a single parent or root. Sibling records are sorted in a particular order. That order is used as the physical order for storing the database. This model is good for describing many real-world relationships.

## Network model

The network model builds on the hierarchical model by allowing many-to-many relationships between linked records, implying multiple parent records. Based on mathematical set theory, the model is constructed with sets of related records. Each set consists of one owner or parent record and one or more member or child records. A record can be a member or child in multiple sets, allowing this model to convey complex relationships.

## Object-oriented database model

This model defines a database as a collection of objects, or reusable software elements, with associated features and methods. There are several kinds of object-oriented databases:

A multimedia database incorporates media, such as images, that could not be stored in a relational database.

A hypertext database allows any object to link to any other object. It’s useful for organizing lots of disparate data, but it’s not ideal for numerical analysis.

The object-oriented database model is the best known post-relational database model, since it incorporates tables, but isn’t limited to tables. Such models are also known as hybrid database models.

## Object-relational model

This hybrid database model combines the simplicity of the relational model with some of the advanced functionality of the object-oriented database model. In essence, it allows designers to incorporate objects into the familiar table structure.

Languages and call interfaces include SQL3, vendor languages, ODBC, JDBC, and proprietary call interfaces that are extensions of the languages and interfaces used by the relational model.

## Entity-relationship model

This model captures the relationships between real-world entities much like the network model, but it isn’t as directly tied to the physical structure of the database. Instead, it’s often used for designing a database conceptually.

## Other database models

A variety of other database models have been or are still used today.

### Inverted file model

A database built with the inverted file structure is designed to facilitate fast full text searches. In this model, data content is indexed as a series of keys in a lookup table, with the values pointing to the location of the associated files. This structure can provide nearly instantaneous reporting in big data and analytics, for instance.

This model has been used by the ADABAS database management system of Software AG since 1970, and it is still supported today.

### Flat model

The flat model is the earliest, simplest data model. It simply lists all the data in a single table, consisting of columns and rows. In order to access or manipulate the data, the computer has to read the entire flat file into memory, which makes this model inefficient for all but the smallest data sets.

### Multidimensional model

This is a variation of the relational model designed to facilitate improved analytical processing. While the relational model is optimized for online transaction processing (OLTP), this model is designed for online analytical processing (OLAP).

Each cell in a dimensional database contains data about the dimensions tracked by the database. Visually, it’s like a collection of cubes, rather than two-dimensional tables.

### Semi Structured model

In this model, the structural data usually contained in the database schema is embedded with the data itself. Here the distinction between data and schema is vague at best. This model is useful for describing systems, such as certain Web-based data sources, which we treat as databases but cannot constrain with a schema. It’s also useful for describing interactions between databases that don’t adhere to the same schema.

### Context model

This model can incorporate elements from other database models as needed. It cobbles together elements from object-oriented, semistructured, and network models.

### Associative model

This model divides all the data points based on whether they describe an entity or an association. In this model, an entity is anything that exists independently, whereas an association is something that only exists in relation to something else.

The associative model structures the data into two sets:

* A set of items, each with a unique identifier, a name, and a type
* A set of links, each with a unique identifier and the unique identifiers of a source, verb, and target. The stored fact has to do with the source, and each of the three identifiers may refer either to a link or an item.

Other, less common database models include:

* Semantic model, which includes information about how the stored data relates to the real world
* XML database, which allows data to be specified and even stored in XML format
* Named graph
* Triplestore

## NoSQL database models

In addition to the object database model, other noSQL models have emerged in contrast to the relational model:

The graph database model, which is even more flexible than a network model, allows any node to connect with any other.

The multi value model, which breaks from the relational model by allowing attributes to contain a list of data rather than a single data point.

The document model, which is designed for storing and managing documents or semi-structured data, rather than atomic data.

## Databases on the Web

Most websites rely on some kind of database to organize and present data to users. Whenever someone uses the search functions on these sites, their search terms are converted into queries for a database server to process. Typically, middleware connects the web server with the database.The broad presence of databases allows them to be used in almost any field, from online shopping to micro-targeting a voter segment as part of a political campaign. Various industries have developed their own norms for database design, from air transport to vehicle manufacturing.

# **2. Describe DB normalization, connection between tables.**

## Data normalization

Data normalization is the process of reorganizing data within a database so that users can utilize it for further queries and analysis. Simply put, it is the process of developing clean data. This includes eliminating redundant and unstructured data and making the data appear similar across all records and fields.

How Does Data Normalization Work?

Data organization in a database is done by normalization. This entails building tables and linking those tables together in accordance with principles intended to safeguard the data and increase the database's adaptability by removing duplication and inconsistent reliance.

Disk space is wasted by redundant data, and maintenance issues result. If data that already exists in multiple locations needs to be modified, it must be updated in the same manner everywhere. If the information is kept solely in the Customers table and not elsewhere in the database, changing a customer's address is significantly simpler to do.

Whereas it makes perfect sense for a user to search in the Customers database for a specific customer's address, it might not sound right to do so for the worker who phones on that customer's behalf. The wage of the employee must be transferred to the Employees table because it is connected to or dependent upon the employee. Data might become difficult to access as a result of inconsistent dependencies because the path to finding the data may be incomplete or damaged.

Before moving on to the different forms of data normalization, you need to first understand the concept of keys in SQL. A key can be a single column or a combination of columns that uniquely identify the rows (or tuples) in the table. It also helps to identify duplicate information and establish relationships between different tables.

Here are the most common type of keys:

* [Primary key](https://www.simplilearn.com/tutorials/sql-tutorial/primary-key-in-sql) - A single column used to uniquely identify a table
* [Composite key](https://www.simplilearn.com/tutorials/sql-tutorial/composite-key-in-sql) - A set of columns used to uniquely identify the rows in a table
* [Foreign key](https://www.simplilearn.com/tutorials/sql-tutorial/foreign-key) - A key that references the primary key of another table

## Data Normalization Forms

Data normalization can be divided into different types of normal forms. The most popular ones are 1NF, 2NF, 3NF, and BCNF. Let us dive into all these normal forms with the help of an example. Assume that a company has a database of all their employees and their key skills as shown in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| Salutation | Full Name | Address | Skills |
| Mr. | John Denver | 12, Bates Brothers Road | Content writing, Social media marketing |
| Ms. | Mary Ann | 34, Shadowman Drive | Deep Learning, Data science |
| Ms. | Nancy Drew | 4, First Plot Street | DBMS |

### 1NF - First Normal Form

The most basic form of data normalization is 1NF which ensures there are no two same entries in a group. For a table to be in the first normal form, it should satisfy the following rules:

* Each cell should contain a single value
* Each record should be unique

The table in 1NF will look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| Salutation | Full Name | Address | Skills |
| Mr. | John Denver | 12, Bates Brothers Road | Content writing |
| Mr. | John Denver | 12, Bates Brothers Road | Social media marketing |
| Ms. | Mary Ann | 34, Shadowman Drive | Machine Learning |
| Ms. | Mary Ann | 34, Shadowman Drive | Data science |
| Ms. | Nancy Drew | 4, First Plot Street | DBMS |

### 2NF - Second Normal Form

In a 2NF table, all the subsets of data that can be placed in multiple rows are placed in separate tables. For a table to be in the second normal form, it should satisfy the following rules:

* It should be in 1F
* The primary key should not be functionally dependant on any subset of candidate key

Let’s divide the 1NF table into two tables - Table 1 and Table 2. Table 1 contains all the employee information. Table 2 contains information on their key skills.

Table 1

|  |  |  |  |
| --- | --- | --- | --- |
| Employee ID | Salutation | Full Name | Address |
| 1 | Mr. | John Denver | 12, Bates Brothers Road |
| 2 | Ms. | Mary Ann | 34, Shadowman Drive |
| 3 | Ms. | Nancy Drew | 4, First Plot Street |

Table 2

|  |  |
| --- | --- |
| Employee ID | Key skills |
| 1 | Content marketing |
| 1 | Social media marketing |
| 2 | Machine learning |
| 2 | Data science |
| 3 | DBMS |

We have introduced a new column called Employee ID which is the primary key for Table 1. The records can be uniquely identified using this primary key.

In Table 2, Employee ID is the foreign key.

### 3NF - Third Normal Form

For a table to be in the third normal form, it should satisfy the following rules:

* It should be in 2F
* It should not have any transitive functional dependencies

A transitive functional dependency is when a change in a column (which is not a primary key) may cause any of the other columns to change.

In our example, if there is a name change (male to female), there may be a change in the salutation (Mr., Ms., Mrs., etc.). Hence we will introduce a new table that stores the salutations

Table 1

|  |  |  |  |
| --- | --- | --- | --- |
| Employee ID | Full Name | Address | Salutation |
| 1 | John Denver | 12, Bates Brothers Road | 1 |
| 2 | Mary Ann | 34, Shadowman Drive | 2 |
| 3 | Nancy Drew | 4, First Plot Street | 2 |

Table 2

|  |  |
| --- | --- |
| Employee ID | Key skills |
| 1 | Content marketing |
| 1 | Social media marketing |
| 2 | Machine learning |
| 2 | Data science |
| 3 | DBMS |

Table 3

|  |  |
| --- | --- |
| Salutation ID | Salutation |
| 1 | Mr. |
| 2 | Ms. |
| 3 | Mrs. |

Now, there are no transitive functional dependencies and our table is now in 3F. Salutation ID is the primary key in Table 3. Salutation ID in Table 1 is foreign to the primary key in Table 3.

### BCNF - Boyce and Codd Normal Form

Boyce and Codd Normal Form is a higher version of 3NF and is also known as 3.5NF. A BCNF is a 3NF table that does not have multiple overlapping candidate keys. For a table to be in BCNF, it should satisfy the following rules:

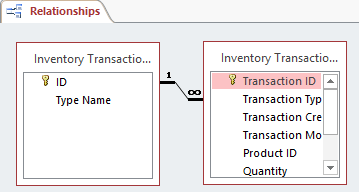
* It should be in 3F
* For each functional dependency ( X → Y ), X should be a super key

## One-to-many relationships

A one-to-many relationship is the most common kind of relationship. In this kind of relationship, a row in table A can have many matching rows in table B. But a row in table B can have only one matching row in table A. For example, the "Publishers" and "Titles" tables have a one-to-many relationship. That is, each publisher produces many titles. But each title comes from only one publisher.

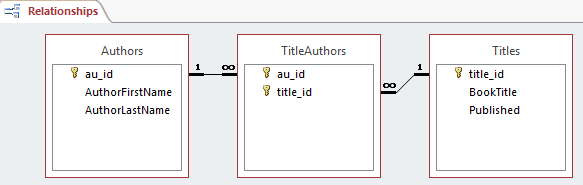
A one-to-many relationship is created if only one of the related columns is a primary key or has a unique constraint.

In the relationship window in Access, the primary key side of a one-to-many relationship is denoted by a number 1. The foreign key side of a relationship is denoted by an infinity symbol.



## Many-to-many relationships

In a many-to-many relationship, a row in table A can have many matching rows in table B, and vice versa. You create such a relationship by defining a third table that is called a junction table. The primary key of the junction table consists of the foreign keys from both table A and table B. For example, the "Authors" table and the "Titles" table have a many-to-many relationship that is defined by a one-to-many relationship from each of these tables to the "TitleAuthors" table. The primary key of the "TitleAuthors" table is the combination of the au\_ID column (the "Authors" table's primary key) and the title\_ID column (the "Titles" table's primary key).



## One-to-one relationships

In a one-to-one relationship, a row in table A can have no more than one matching row in table B, and vice versa. A one-to-one relationship is created if both of the related columns are primary keys or have unique constraints.

This kind of relationship is not common, because most information that is related in this manner would be in one table. You might use a one-to-one relationship to take the following actions:

* Divide a table with many columns.
* Isolate part of a table for security reasons.
* Store data that is short-lived and could be easily deleted by deleting the table.
* Store information that applies only to a subset of the main table.

In Access, the primary key side of a one-to-one relationship is denoted by a key symbol. The foreign key side is also denoted by a key symbol.

# **3. Provide sql queries to create/drop/delete tables, join tables with different kinds of join.**

The CREATE TABLE command creates a new table in the database.

The following SQL creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

CREATE TABLE Persons (

PersonID int,

LastName varchar(255),

FirstName varchar(255),

Address varchar(255),

City varchar(255)

);

The following SQL creates a new table called "TestTables" (which is a copy of two columns of the "Customers" table):

CREATE TABLE TestTable AS

SELECT customername, contactname

FROM customers;

The DROP TABLE command deletes a table in the database. The following SQL deletes the table "Shippers":

DROP TABLE Shippers;

The TRUNCATE TABLE command deletes the data inside a table, but not the table itself. The following SQL truncates the table "Categories":

TRUNCATE TABLE Categories;

The DELETE command is used to delete existing records in a table. The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

DELETE FROM Customers;

## 

The INNER JOIN keyword selects records that have matching values in both tables.

SELECT *column\_name(s)*

FROM *table1*

INNER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;



The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

SELECT *column\_name(s)*

FROM *table1*

LEFT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;



The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

SELECT *column\_name(s)*

FROM *table1*

RIGHT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;



The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

SELECT *column\_name(s)*

FROM *table1*

FULL OUTER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*

WHERE *condition*;



A SELF JOIN is a regular join, but the table is joined with itself.

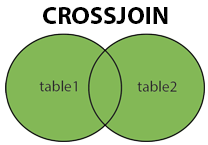
SELECT *column\_name(s)*

FROM *table1 T1, table1 T2*

WHERE *condition*;

*T1* and *T2* are different table aliases for the same table.

The CROSS JOIN keyword returns all records from both tables (table1 and table2).



SELECT *column\_name(s)*

FROM *table1*

CROSS JOIN *table2*;

# **4. Provide sql queries to check presence of duplicates in a table, queries with aggregating and window functions.**

## How to Find Duplicate Values in SQL

First, you will need to define the criteria for detecting duplicate rows. Is it a combination of two or more columns where you want to detect duplicate values, or are you simply searching for duplicates within a single column?

In the examples below, we will be exploring both these scenarios using a simple customer order database.

In terms of the general approach for either scenario, finding duplicates values in SQL comprises two key steps:

1. Using the GROUP BY clause to group all rows by the target column(s) – i.e. the column(s) you want to check for duplicate values on.
2. Using the COUNT function in the HAVING clause to check if any of the groups have more than 1 entry; those would be the duplicate values.

### Duplicate Values in One Column

Here, we will be demonstrating how you can find duplicate values in a single column. For this example, we will be using the Orders table, a modified version of the table we used in my previous article on [using GROUP BY in SQL](https://learnsql.com/blog/what-is-group-by-in-sql/). A sample of the table is shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |
| 10251 | 84 | 3 | 1996-07-08 | 1 |
| 10251 | 84 | 3 | 1996-07-08 | 1 |
| 10252 | 76 | 4 | 1996-07-09 | 2 |
| … | … | … | … | … |
| 10443 | 66 | 8 | 1997-02-12 | 1 |

In this example, there are a few duplicates in the OrderID column. Ideally, each row should have a unique value for OrderID, since each individual order is assigned its own value. For some reason, that wasn’t implemented here. To find the duplicates, we can use the following query:

SELECT OrderID, COUNT(OrderID)

FROM Orders

GROUP BY OrderID

HAVING COUNT(OrderID) > 1

RESULT

Number of Records: 2

|  |  |
| --- | --- |
| OrderID | COUNT(OrderID) |
| 10251 | 2 |
| 10276 | 2 |

As we can see, OrderID 10251 (which we saw in the table sample above) and OrderID 10276 have duplicates.

Using the GROUP BY and HAVING clauses can neatly show the duplicates in your data. Once you have validated that the rows are the same, you may choose to remove the duplicate(s) using the [DELETE](https://learnsql.com/blog/sql-insert-sql-update-sql-delete-oh-my/) statement.

### Duplicate Values in Multiple Columns

Often, you’re interested in finding rows where a combination of a few columns match. For this example, we will be using the OrderDetails table, a sample of which is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| OrderDetailID | OrderID | ProductID | Quantity |
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 14 | 2 |
| 6 | 10249 | 51 | 40 |
| … | … | … | … |
| 520 | 10443 | 28 | 12 |

We want to find entries where the OrderID and ProductID columns are identical. This type of duplicate likely means there is a bug in the ordering system, since each order will process each product in that order only once in the cart. If multiple quantities of that product are ordered, the Quantity value would simply be increased; separate (duplicate) rows should not be created. A glitch of this type may impact business operations negatively if the orders are being fulfilled, packaged, and shipped automatically.

To find duplicates in multiple column values, we can use the following query. It’s very similar to the one for a single column:

SELECT OrderID, ProductID, COUNT(\*)

FROM OrderDetails

GROUP BY OrderID, ProductID

HAVING COUNT(\*) > 1

RESULT

Number of Records: 2

Above, we can confirm that the ordering system does indeed have a bug. Like the first example using a single column, this second example similarly allows us to find errors in the ordering system. In this case, products are being registered as a new order even though they were added to the same cart by the same customer. Now you, as the business owner, can take proper corrective actions to rectify this bug in your order management system.

Note that above, we used COUNT(\*) and not a column-specific counter such as COUNT(OrderID). COUNT(\*) counts all rows, whereas COUNT (Column) only counts non-null values in the specified column. However, in this example, it will not have made a difference either way – there were no null values in either of the two columns being grouped.

## Unique Values

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT *column1*, *column2, ...*

FROM *table\_name*;

## Grouping queries

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

GROUP BY *column\_name(s)*

ORDER BY *column\_name(s);*

## Aggregate calculations

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

SELECT MIN(*column\_name*)

FROM *table\_name*

WHERE *condition*;

SELECT MAX(*column\_name*)

FROM *table\_name*

WHERE *condition*;

The COUNT() function returns the number of rows that matches a specified criterion.

SELECT COUNT(*column\_name*)

FROM *table\_name*

WHERE *condition*;

The AVG() function returns the average value of a numeric column.

SELECT AVG(*column\_name*)

FROM *table\_name*

WHERE *condition*;

The SUM() function returns the total sum of a numeric column.

SELECT SUM(*column\_name*)

FROM *table\_name*

WHERE *condition*;

Window (also, windowing or windowed) functions perform a calculation over a set of rows. I like to think of “looking through the window” at the rows that are being returned and having one last chance to perform a calculation. The window is defined by the OVER clause which determines if the rows are partitioned into smaller sets and if they are ordered. In fact, if you use a window function you will always use an OVER clause. The OVER clause is also part of the NEXT VALUE FOR syntax required for the sequence object, but, otherwise it’s used with window functions.

The OVER clause may contain a PARTITION BY option. This breaks the rows into smaller sets. You might think that this is the same as GROUP BY, but it’s not. When grouping, one row per unique group is returned. When using PARTITION BY, all of the detail rows are returned along with the calculations. If you have a window in your home that is divided into panes, each pane is a window. When thinking about window functions, the entire set of results is a partition, but when using PARTITION BY, each partition can also be considered a window. PARTITION BY is supported – and optional – for all windowing functions.

The OVER clause may also contain an ORDER BY option. This is independent of the ORDER BY clause of the query. Some of the functions require ORDER BY, and it’s not supported by the others. When the order of the rows is important when applying the calculation, the ORDER BY is required.

Window functions may be used only in the SELECT and ORDER BY clauses of a query. They are applied after any joining, filtering, or grouping.

## Ranking Functions

The most commonly used window functions, ranking functions, have been available since 2005. That’s when Microsoft introduced ROW\_NUMBER, RANK, DENSE\_RANK, and NTILE. ROW\_NUMBER is used very frequently, to add unique row numbers to a partition or to the entire result set. Adding a row number, or one of the other ranking functions, is not usually the goal, but it is a step along the way to the solution.

ORDER BY is required in the OVER clause when using ROW\_NUMBER and the other functions in this group. This tells the database engine the order in which the numbers should be applied. If the values of the columns or expressions used in the ORDER BY are not unique, then RANK and DENSE\_RANK will deal with the ties, while ROW\_NUMBER doesn’t care about ties. NTILE is used to divide the rows into buckets based on the ORDER BY.

One benefit of ROW\_NUMBER is the ability to turn non-unique rows into unique rows. This could be used to eliminate duplicate rows, for example.

To show how this works, start with a temp table containing duplicate rows. The first step is to create the table and populate it.

CREATE TABLE #Duplicates(Col1 INT, Col2 CHAR(1));

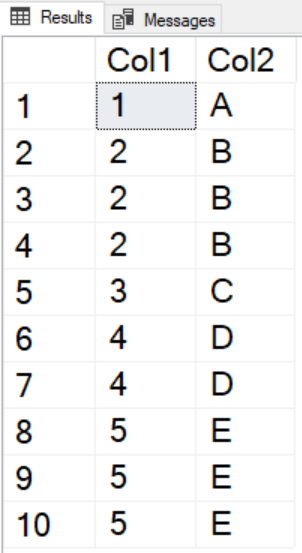
INSERT INTO #Duplicates(Col1, Col2)

VALUES(1,'A'),(2,'B'),(2,'B'),(2,'B'),

(3,'C'),(4,'D'),(4,'D'),(5,'E'),

(5,'E'),(5,'E');

SELECT \* FROM #Duplicates;

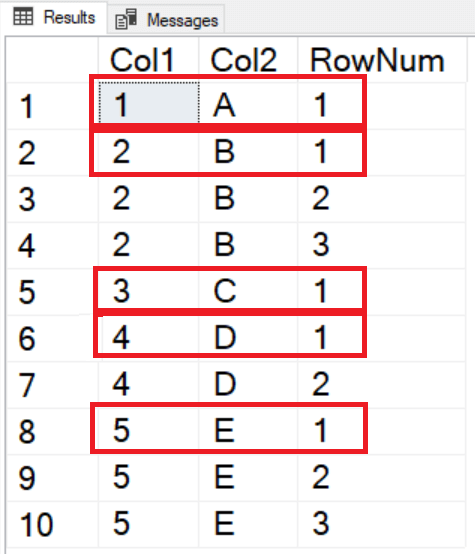


Adding ROW\_NUMBER and partitioning by each column will restart the row numbers for each unique set of rows. You can identify the unique rows by finding those with a row number equal to one.

SELECT Col1, Col2,

ROW\_NUMBER() OVER(PARTITION BY Col1, Col2 ORDER BY Col1) AS RowNum

FROM #Duplicates;

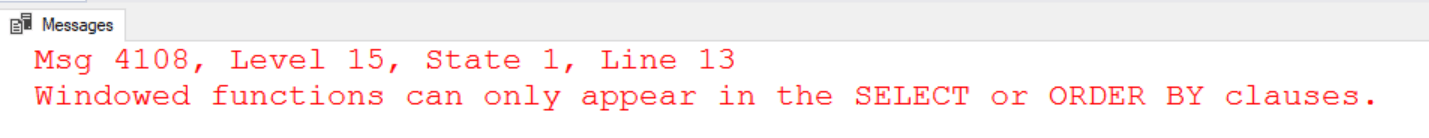


Now, all you have to do is to delete any rows that have a row number greater than one. The problem is that you cannot add window functions to the WHERE clause.

DELETE #Duplicates

WHERE ROW\_NUMBER() OVER(PARTITION BY Col1, Col2 ORDER BY Col1) <> 1;

You’ll see this error message:



The way around this problem is to separate the logic using a common table expression (CTE). You can then delete the rows right from the CTE.

WITH Dupes AS (

SELECT Col1, Col2,

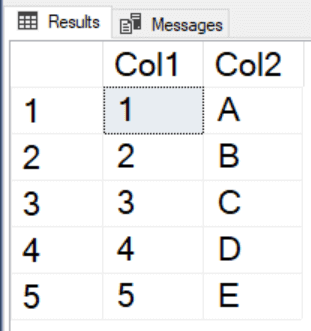
ROW\_NUMBER() OVER(PARTITION BY Col1, Col2 ORDER BY Col1) AS RowNum

FROM #Duplicates)

DELETE Dupes

WHERE RowNum <> 1;

SELECT \* FROM #Duplicates;



Success! The extra rows were deleted, and a unique set of rows remains.

To see the difference between ROW\_NUMBER, RANK, and DENSE\_RANK, run this query:

USE Adventureworks2017; --Or whichever version you have

GO

SELECT SalesOrderID, OrderDate, CustomerID,

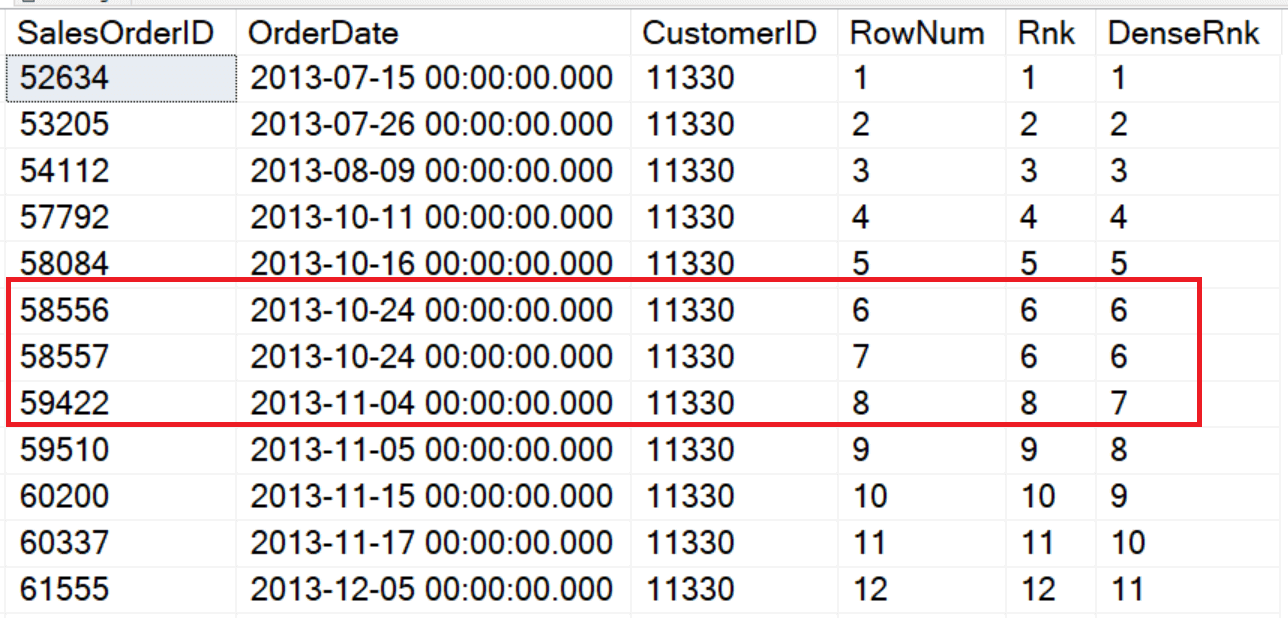
ROW\_NUMBER() OVER(ORDER BY OrderDate) As RowNum,

RANK() OVER(ORDER BY OrderDate) As Rnk,

DENSE\_RANK() OVER(ORDER BY OrderDate) As DenseRnk

FROM Sales.SalesOrderHeader

WHERE CustomerID = 11330;



The ORDER BY for each OVER clause is OrderDate which is not unique. This customer placed two orders on 2013-10-24. ROW\_NUMBER just continued assigning numbers and didn’t do anything different even though there is a duplicate date. RANK assigned 6 to both rows and then caught up to ROW\_NUMBER with an 8 on the next row. DENSE\_RANK also assigned 6 to the two rows but assigned 7 to the following row.

Two explain the difference, think of ROW\_NUMBER as *positional*. RANK is both *positional* and *logical*. Those two rows are ranked logically the same, but the next row is ranked by the position in the set. DENSE\_RANK ranks them *logically*. Order 2013-11-04 is the 7th unique date.

The final function in this group is called NTILE. It assigns bucket numbers to the rows instead of row numbers or ranks. Here is an example:

SELECT SP.FirstName, SP.LastName,

SUM(SOH.TotalDue) AS TotalSales,

NTILE(4) OVER(ORDER BY SUM(SOH.TotalDue)) \* 1000 AS Bonus

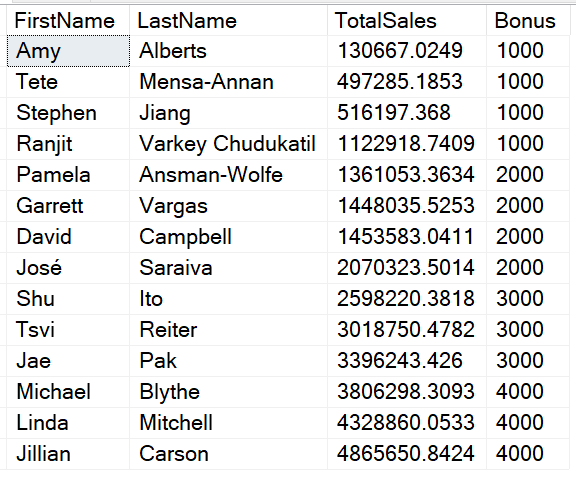
FROM [Sales].[vSalesPerson] SP

JOIN Sales.SalesOrderHeader SOH

ON SP.BusinessEntityID = SOH.SalesPersonID

WHERE SOH.OrderDate >= '2012-01-01' AND SOH.OrderDate < '2013-01-01'

GROUP BY FirstName, LastName;



NTILE has a parameter, in this case 4, which is the number of buckets you want to see in the results. The ORDER BY is applied to the sum of the sales. The rows with the lowest 25% are assigned 1, the rows with the highest 25% are assigned 4. Finally, the results of NTILE are multiplied by 1000 to come up with the bonus amount. Since 14 cannot be evenly divided by 4, an extra row goes into each of the first two buckets.

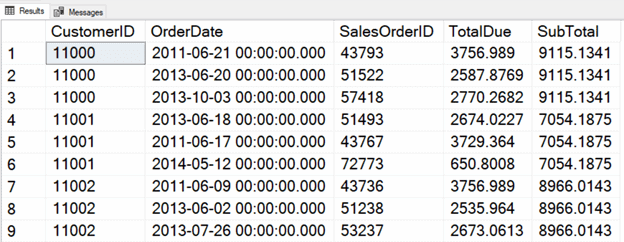
## Window Aggregates

Window aggregates were also introduced with SQL Server 2005. These make writing some tricky queries easy but will often perform worse than older techniques. They allow you to add your favorite aggregate function to a non-aggregate query. Say, for example you would like to display all the customer orders along with the subtotal for each customer. By adding a SUM using the OVER clause, you can accomplish this very easily:

SELECT CustomerID, OrderDate, SalesOrderID, TotalDue,

SUM(TotalDue) OVER(PARTITION BY CustomerID) AS SubTotal

FROM Sales.SalesOrderHeader;



By adding the PARTITION BY, a subtotal is calculated for each customer. Any aggregate function can be used, and ORDER BY in the OVER clause is not supported.

## Window Aggregate Enhancements in 2012

Beginning with 2012, you can add an ORDER BY to the OVER clause to window aggregates to produce running totals and moving averages, for example. At the same time, Microsoft introduced the concept of framing. Adding a PARTITION BY is like dividing a window into panes. Adding framing is like creating a stained-glass window. Each row has an individual window where the expression will be applied.

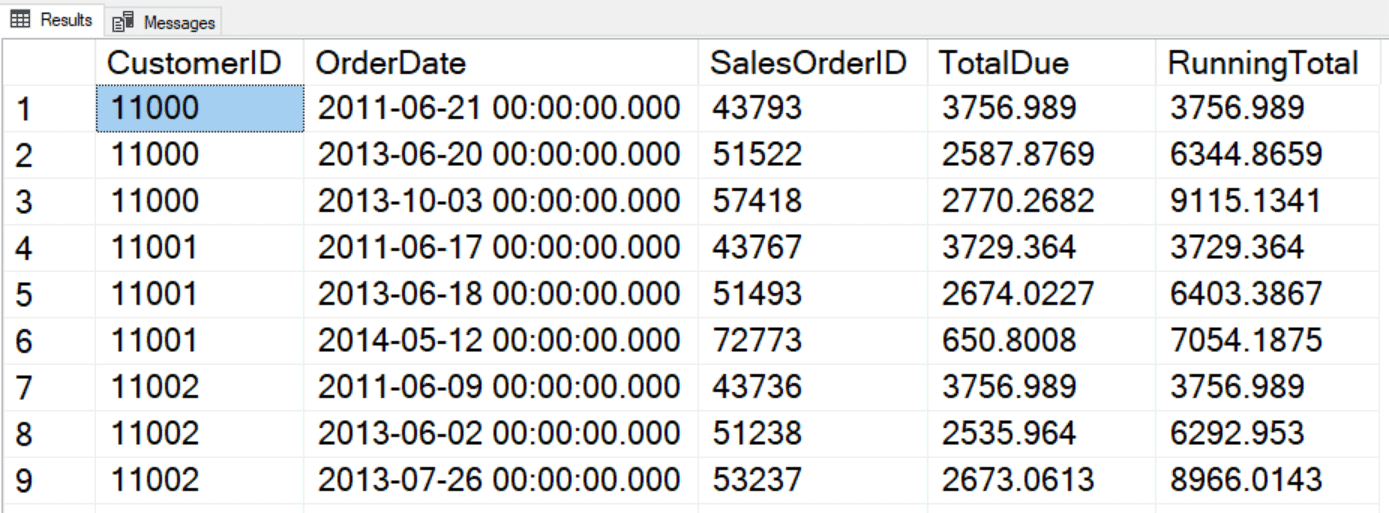
With this enhancement, you can create running totals even without adding the framing syntax. Here is an example that returns a running total by customer:

SELECT CustomerID, OrderDate, SalesOrderID, TotalDue,

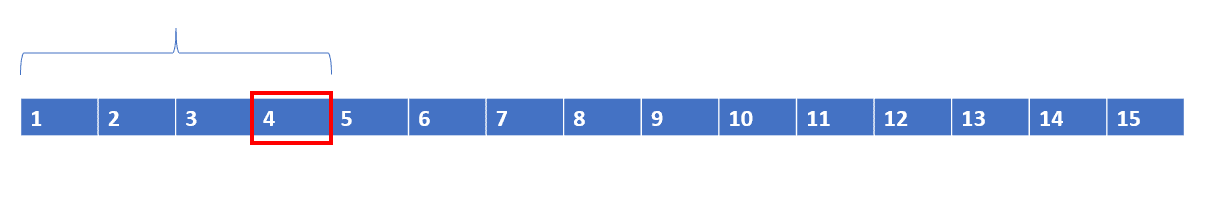
SUM(TotalDue) OVER(PARTITION BY CustomerID ORDER BY SalesOrderID)

AS RunningTotal

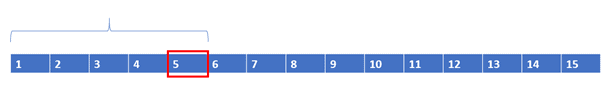
FROM Sales.SalesOrderHeader;



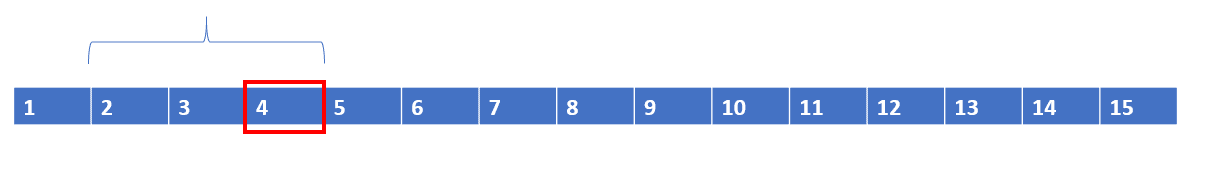
The default frame, which is used if a frame is not specified, is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Unfortunately, this will not perform as well as if you specify this frame instead: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. The difference is the word ROWS. RANGE is only partially implemented at this time, and it’s meant for working with periods of time, while ROWS is positional. The frame, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, means that the window consists of the first row of the partition and all the rows up to the current row. Each calculation is done over a different set of rows. For example, when performing the calculation for row 4, the rows 1 to 4 are used.



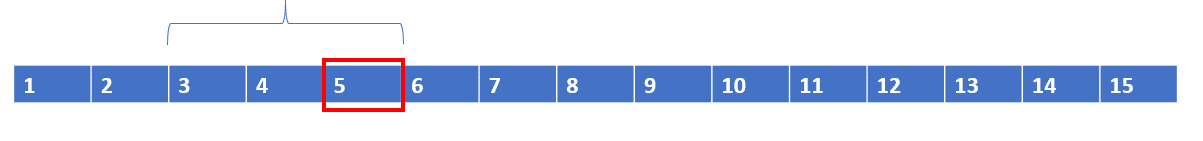
When performing the calculation for row 5, the rows are 1 to 5. The window grows larger as you move from one row to the next.



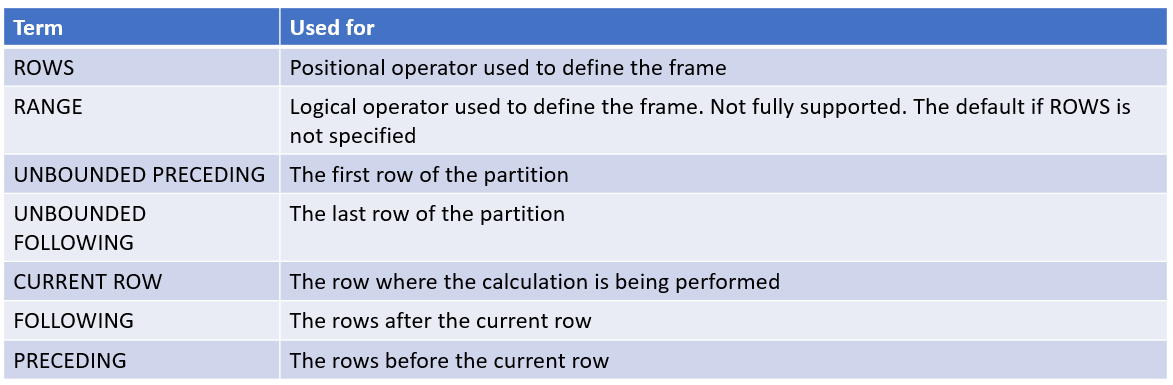
You can also use the syntax ROWS BETWEEN N PRECEEDING AND CURRENT ROW or ROWS BETWEEN CURRENT ROW AND N FOLLOWING. This could be useful for calculating a three-month moving average, for example. The following figure represents ROWS BETWEEN 2 PRECEDING AND CURRENT ROW.



When 5 is the current row, the window moves; it doesn’t change size.



Here is the list of terms you need to know when writing the framing option:



I admit that this syntax is a bit confusing but using [SQL Prompt](https://www.red-gate.com/products/sql-development/sql-prompt/) helps makes writing the framing option easier!

## Offset Functions

Also included with the release of SQL Server 2012 are four functions that allow you to include values from other rows – without doing a self-join. Microsoft calls these ‘analytic functions’, but I always refer to them as ‘offset functions’ when presenting on this topic. Two of the functions allow you to pull columns or expressions from a row before (LAG) or after (LEAD) the current row. The other two functions allow you to return values from the first row of the partition (FIRST\_VALUE) or last row of the partition (LAST\_VALUE). FIRST\_VALUE and LAST\_VALUE also require framing, so be sure to include the frame when using these functions. All four of the functions require the ORDER BY option of the OVER clause. That makes sense, because the database engine must know the order of the rows to figure out which row contains the value to return.

Some people have a favourite band; some people have a favourite movie. I have a favourite function – LAG. It’s easy to use (no frame!) and performs great. Here is an example:

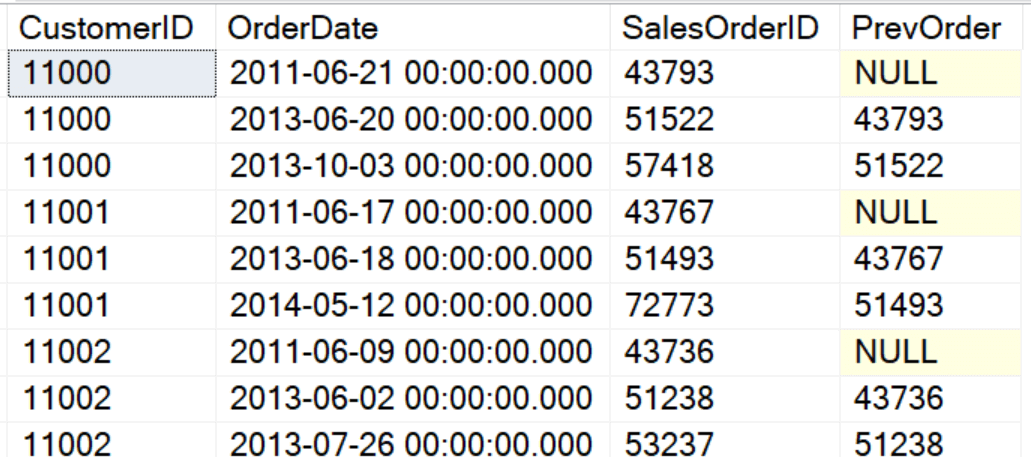
SELECT CustomerID, OrderDate, SalesOrderID,

LAG(SalesOrderID) OVER(PARTITION BY CustomerID ORDER BY SalesOrderID

) AS PrevOrder

FROM Sales.SalesOrderHeader

ORDER BY CustomerID;



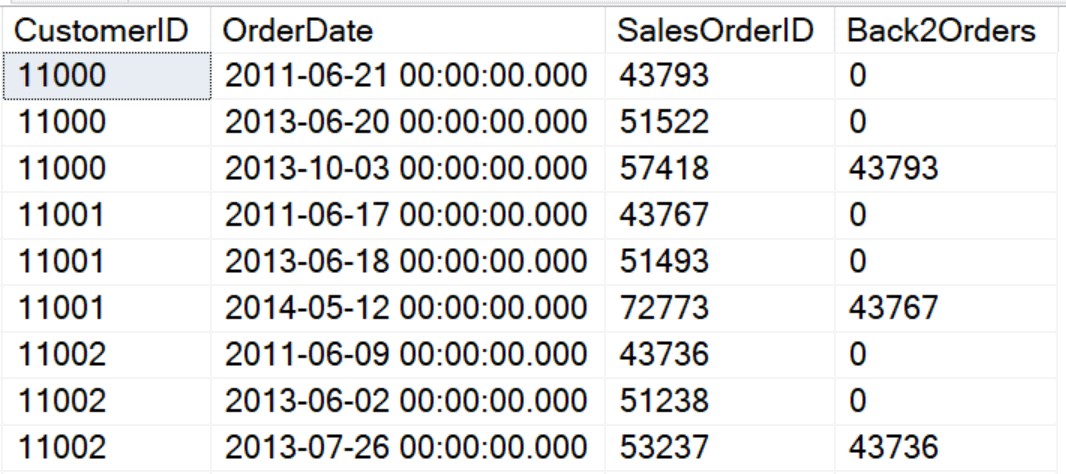
LAG and LEAD require an argument – the column or expression you want to return. By default, LAG returns the value from the previous row, and LEAD returns the value from the following row. You can modify that by supplying a value for the OFFSET parameter, which is 1 by default. Notice that the first row of the partition returns NULL. If you wish to override the NULLs, you can supply a DEFAULT value. Here is a similar query that goes back two rows and has a default value:

SELECT CustomerID, OrderDate, SalesOrderID,

LAG(SalesOrderID,2,0) OVER(PARTITION BY CustomerID

ORDER BY SalesOrderID) AS Back2Orders

FROM Sales.SalesOrderHeader;



FIRST\_VALUE and LAST\_VALUE can be used to find a value from the very first row or very last row of the partition. Be sure to specify the frame, not only for performance reasons, but because the default frame doesn’t work as you would expect with LAST\_VALUE. The default frame, RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, only goes up to the current row. The last row of the partition is not included. To get the expected results, be sure to specify ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING when using LAST\_VALUE. Here is an example using FIRST\_VALUE

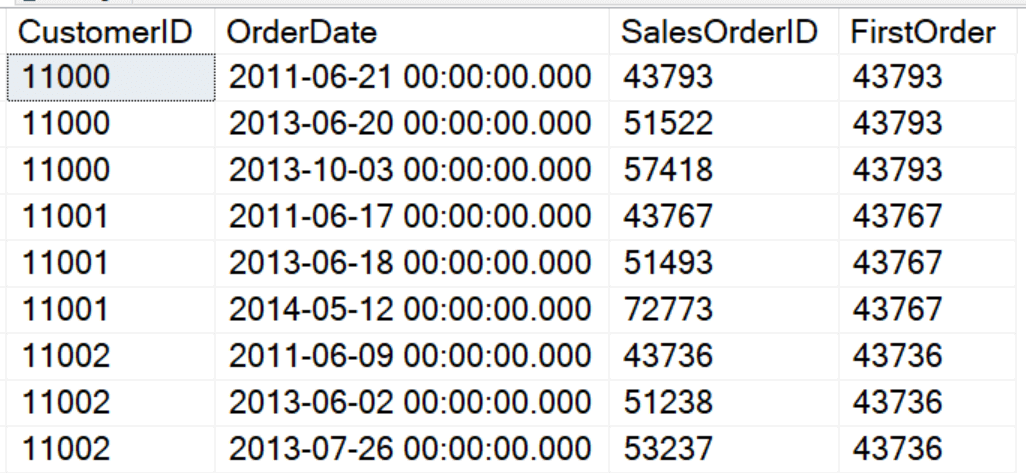
SELECT CustomerID, OrderDate, SalesOrderID,

FIRST\_VALUE(SalesOrderID) OVER(PARTITION BY CustomerID

ORDER BY SalesOrderID

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS FirstOrder

FROM Sales.SalesOrderHeader;



## Statistical Functions

Microsoft groups these four functions – PERCENT\_RANK, CUME\_DIST, PERCENTILE\_DISC, PERCENTILE\_CONT – along with the offset functions calling all eight the analytic functions. Since I like to distinguish these from the offset functions, I call these statistical.

PERCENT\_RANK and CUME\_DIST provide a ranking for each row over a partition. They differ slightly. PERCENT\_RANK returns the percentage of rows that rank lower than the current row. “My score is higher than 90% of the scores.” CUME\_DIST, or cumulative distribution, returns the exact rank. “My score is at 90% of the scores.” Here is an example using the average high temperature in St. Louis for each month. Note that the ranks were determined by the Fahrenheit temperature.

CREATE TABLE #MonthlyTempsStl(MName varchar(15), AvgHighTempF INT,

AvgHighTempC INT)

INSERT INTO #MonthlyTempsStl(MName, AvgHighTempF, AvgHighTempC)

VALUES('Jan',40,4),('Feb',45, 7),('Mar',56, 13),('Apr',67, 20),

('May',76,25),('Jun',85,30),('Jul',89,32),('Aug',88,31),

('Sep',80,27),('Oct',69,20),('Nov',56,13),('Dec',43,6);

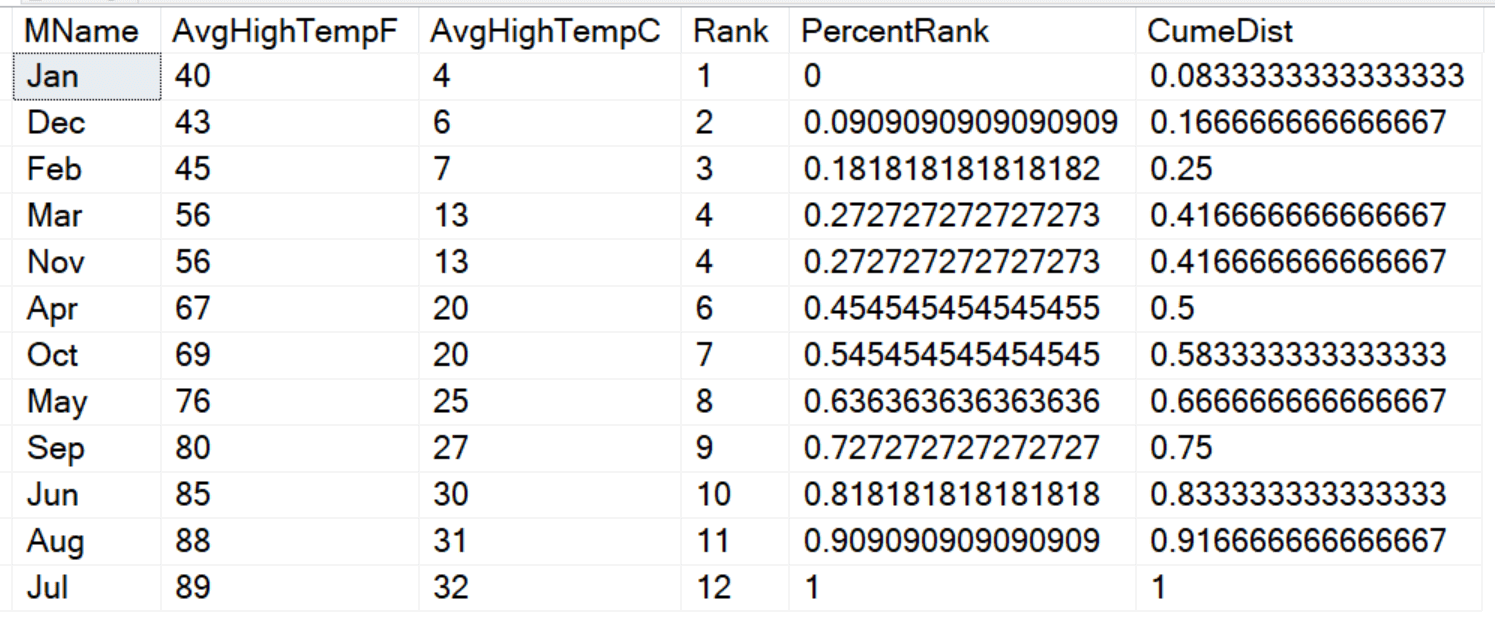
SELECT MName, AvgHighTempF,AvgHighTempC,

RANK() OVER(ORDER BY AvgHighTempF) AS Rank,

PERCENT\_RANK() OVER(ORDER BY AvgHighTempF) AS PercentRank,

CUME\_DIST() OVER(ORDER BY AvgHighTempF) AS CumeDist

FROM #MonthlyTempsStl;



The ranks are not determined by the relative values, but by the positions of the rows. Notice that March and November have the same average high temp, so they were ranked the same.

You may be wondering how to calculate PERCENT\_RANK and CUME\_DIST. Here are the formulas:

PERCENT\_RANK = (Rank -1)/(Row count -1)

CUME\_DIST = (Rank)/(Row count)

PERCENTILE\_DISC and PERCENTILE\_CONT work in the opposite way. Given a percent rank, find the value at that rank. They differ in that PERCENTILE\_DISC will return a value that exists in the set while PERCENTILE\_CONT will calculate an exact value if none of the values in the set falls precisely at that rank. You can use PERCENTILE\_CONT to calculate a median by supplying 0.5 as the percent rank. For example, which temperature ranks at 50% in St. Louis?

SELECT MName, AvgHighTempF,AvgHighTempC,

RANK() OVER(ORDER BY AvgHighTempF) AS Rank,

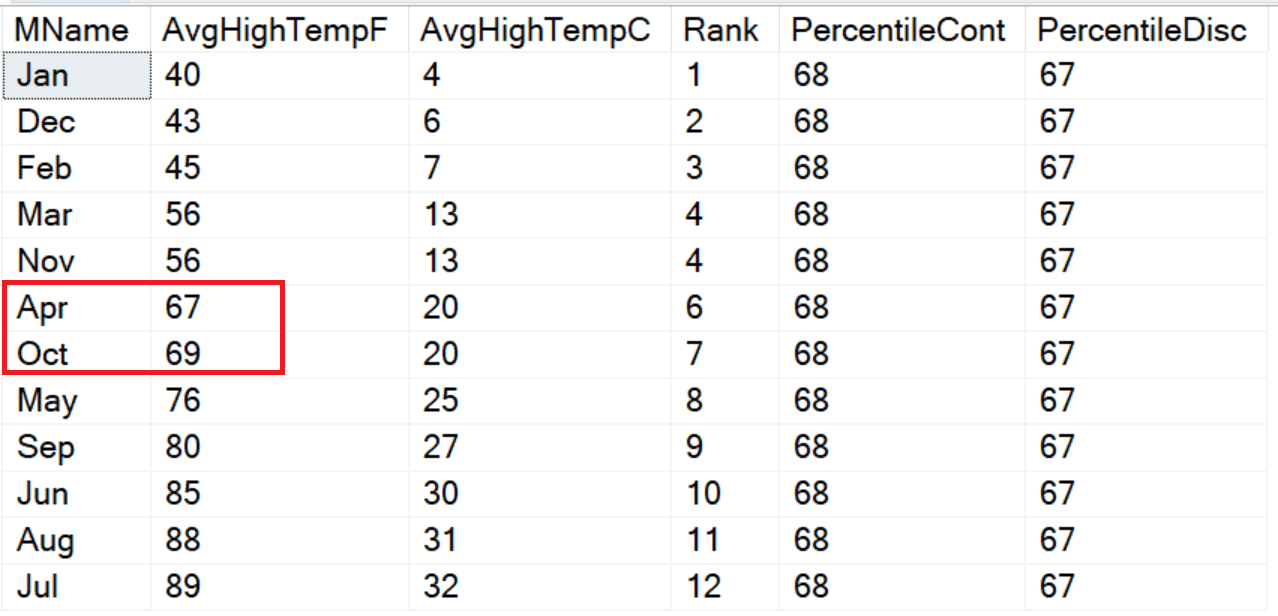
PERCENTILE\_CONT(0.5) WITHIN GROUP(ORDER BY AvgHighTempF)

OVER() AS PercentileCont,

PERCENTILE\_DISC(0.5) WITHIN GROUP(ORDER BY AvgHighTempF)

OVER() AS PercentileDisc

FROM #MonthlyTempsStl;



The PERCENTILE\_CONT function takes the average of the two values closest to the middle, 67 and 69, and averages them. PERCENTILE\_DISC returns an exact value, 67. Also notice that these two functions have an extra clause not seen in the other functions, WITHIN GROUP, that contains the ORDER BY instead of within the OVER clause.