# Order of execution: FROM, JOIN > WHERE > GROUP BY, HAVING >SELECT, WINDOW, DISTINCT, ORDER BY

**1. FROM and JOINs**

The **FROM** clause, and subsequent **JOIN**s are first executed to determine the total working set of data that is being queried. This includes subqueries in this clause, and can cause temporary tables to be created under the hood containing all the columns and rows of the tables being joined.

**2. WHERE**

Once we have the total working set of data, the first-pass **WHERE** constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. *Each of the constraints can only access columns directly from the tables requested in the****FROM****clause.*

*Aliases in the****SELECT****part of the query are not accessible in most databases since they may include expressions dependent on parts of the query that have not yet executed.*

**3. GROUP BY**

The remaining rows after the **WHERE** constraints are applied are then grouped based on common values in the column specified in the **GROUP BY** clause. As a result of the grouping, there will only be as many rows as there are unique values in that column. Implicitly, this means that you should only need to use this when you have aggregate functions in your query.

*Like the****WHERE****clause, aliases are not yet computed so they are not accessible from this step.*

**4. HAVING**

If the query has a **GROUP BY** clause, then the constraints in the **HAVING** clause are then applied to the grouped rows, discard the grouped rows that don't satisfy the constraint.

*Like the****WHERE****clause, aliases are not yet computed so they are not accessible from this step.*

**5. SELECT**

Any expressions in the **SELECT** part of the query are finally computed.

Windows functions are computed at this step.

**6. DISTINCT**

Of the remaining rows, rows with duplicate values in the column marked as **DISTINCT** will be discarded.

**7. ORDER BY**

If an order is specified by the **ORDER BY** clause, the rows are then sorted by the specified data in either ascending or descending order. Since all the expressions in the **SELECT** part of the query have been computed, you can reference aliases in this clause.

**8. LIMIT / OFFSET**

Finally, the rows that fall outside the range specified by the **LIMIT** and **OFFSET** are discarded, leaving the final set of rows to be returned from the query.

# Alias in Select cannot appear in Where, Group By, Having

Alias in Select (e.g. select substring(name, 1, 4) as ALIAS) are generated/computed almost the last in the execution order while WHERE, GROUP BY, HAVING are computed beforehand so the alias cannot appear in Where, Group by, Having.

Exp: The two followings are meant to do the same thing but (1) doesn’t compile.

1. ~~select substring(name, 1, 4) as Alias from employee where Alias = ‘Mike’~~
2. select substring(name, 1, 4) as Alias from employee where substring(name, 1, 4) = ‘Mike’

# GROUP BY

1. GROUP BY can have more than 1 field.  
   Group By X: put all rows with same X value in a group

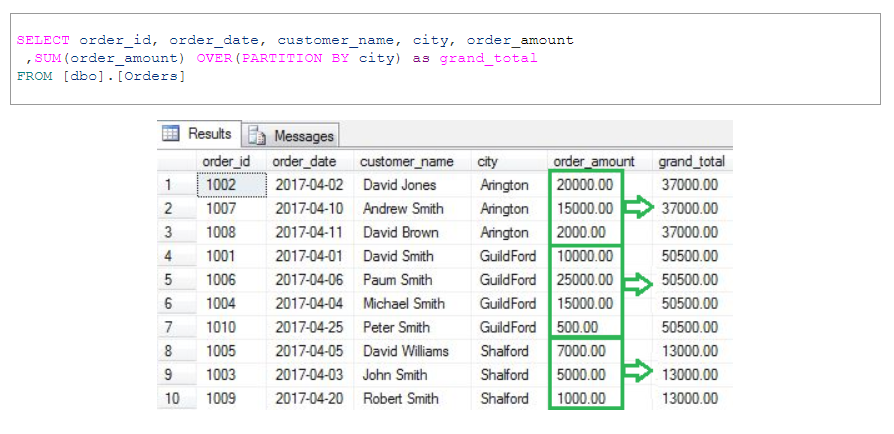
Group By X, Y: put all rows with same both X, Y value in a group

1. **If you use GROUP BY, then fields in SELECT, HAVING must be either in GROUP BY or in an AGGREGATE function.**
2. Having is used to limit the number of groups.

# Having

1. **WHERE doesn’t accept aggregate functions. Conditions with Aggregate functions can be used only in HAVING.**
2. Having doesn’t need to go with GROUP BY. Having without Group by is done by viewing all the rows as a single group. Having requires aggregate functions that are often applied on files in Group by, this is why Having often goes with Group by.
3. The fields in HAVING must be either in GROUP BY or in AGGREGATE FUNCTIONS.

# Why is it called WINDOW functions?



# OVER PARTITION BY vs GROUP BY

The syntax of Analytic functions is similar to the syntax of Aggregate functions:

Analytic functions(fields) over (partition by …)

Aggregate functions(fields) group by …

(The difference is: both Analytic functions(…) and over(partition by …) are in same Select while Aggregate functions(…) and group by are in two separated clauses.)

In terms of rows, the difference is: for Analytic functions, all rows of a group are fully shown while for Aggregate functions, only one row representing the group is shown.

Example: Consider the following “Employee” table

|  |  |  |  |
| --- | --- | --- | --- |
| Employee | | | |
| ID | EmployeeName | EmployeeAge | DepartmentName |
| 1 | Michael | 50 | Accounting |
| 2 | Frances | 60 | Accounting |
| 3 | Daisy | 20 | Accounting |
| 4 | Dennis | 40 | Software |
| 5 | Andrei | 25 | Software |
| 6 | Carol | 25 | Software |
| 7 | Peter | 40 | Human resource |
| 8 | Larry | 35 | Human resource |
| 9 | Robert | 35 | Human resource |
| 10 | Leonard | 22 | Marketing |
| 11 | Andrew | 24 | Marketing |

**Group by:**

**Exp1**: count the employees of each department that has more than 2 employees

Select DepartmentName, count(EmployeeName) as Count

From Employee

Group by DeparmentName

Having count(EmployeeName) > 2

|  |  |
| --- | --- |
| Group by | |
| DepartmentName | Count |
| Accounting | 3 |
| Software | 3 |
| Human resource | 3 |

(Note: it doesn’t compile if use “where” instead of “having”. **Aggregate functions or Alias are not allowed in Where**

~~Select DepartmentName, count(EmployeeName) as tempCount~~

~~From Employee~~

**~~Where tempCount > 2~~**

~~Group by DeparmentName~~)

Exp2: Counting number of employees older than 25 for each department

Select DepartmentName, count(EmployeeAge) as X

From Employee

where EmployeeAge > 25

Group by DepartmentName

|  |  |
| --- | --- |
| Group by | |
| DepartmentName | Count |
| Accounting | 2 |
| Software | 1 |
| Human resource | 3 |

(Note: it doesn’t compile if use “Having” instead of “Where” since EmployeeAge must be in Group By

~~Select DepartmentName, count(EmployeeAge) as X~~

~~From Employee~~

~~Group by DepartmentName~~

~~Having EmployeeAge > 25~~)

**Over partition by:**

**Exp1**: count the employees of each department that has more than 2 employees

The following doesn’t compile since X is an alias that cannot be in where

~~Select EmployeeName, DepartmentName, count(EmployeeName) over (partition by DepartmentName) as x~~

~~from Employee~~

~~where X > 2~~

Exp2: Counting number of employees older than 25 for each department

Select EmployeeName, DepartmentName, count(EmployeeName) over (partition by DepartmentName) as Count

From Employee

Where EmployeeAge > 25

|  |  |  |  |
| --- | --- | --- | --- |
| Over partition by | | | |
| EmployeeName | EmployeeAge | DepartmentName | Count |
| Michael | 50 | Accounting | 2 |
| Frances | 60 | Accounting | 2 |
| Dennis | 40 | Software | 1 |
| Peter | 40 | Human resource | 3 |
| Larry | 35 | Human resource | 3 |
| Robert | 35 | Human resource | 3 |

The most important difference is “Aggregate vs Analytic”.

In the above example, for Group By, only groups plus some properties of the groups are viewed. For Over partition by, individual elements of groups and properties of groups are displayed so there must be repetition.

Viewing group is called aggregate while viewing elements is called analytic.

# For every group, list only some elements

Exp: still with Employee table above, for each department, listing the oldest employee

Subquery:

Select DepartmentName, EmployeeName

From Employee

Where EmployeeAge = (Select MAX(EmployeeAge)

From Employee as tempEmployee

Where tempEmployee.DepartmentName = Employee.DepartmentName)

Over partition by

select \*

from (select DepartmentName, EmployeeName, EmployeeAge,

row\_number() over (partition by DepartmentName order by EmployeeAge DESC) as rank

from Employee) as extendedEmployee

where extendedEmployee.rank = 1

# with tableName as (select from where)

“with” must be followed by Select/Insert/Update command and it’s used only ONCE.

“with” cannot be followed by “with”. If want so, put multiple “with” in one command, separated by comma.

With name1 as (…), name2 as (…)

# In FROM clause, user-defined tables must have name

# JOIN, be it left join, outer join, inner join can multiply the number of records