Chapter 6: SELECT

The SELECT statement is at the heart of most SQL queries. It defines what result set should be returned by the query, and is almost always used in conjunction with the FROM clause, which defines what part(s) of the database should be queried.

Section 6.1: Using the wildcard character to select all columns in a query

Consider a database with the following two tables.

Employees table:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Departments table:

Id	Name
1	Sales
2	Marketing
3	Finance
4	IT

Simple select statement

`*` is the **wildcard character** used to select all available columns in a table.

When used as a substitute for explicit column names, it returns all columns in all tables that a query is selecting `FROM`. This effect applies to **all tables** the query accesses through its `JOIN` clauses.

Consider the following query:

```
SELECT * FROM Employees
```

It will return all fields of all rows of the `Employees` table:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Dot notation

To select all values from a specific table, the wildcard character can be applied to the table with *dot notation*.

Consider the following query:

```
SELECT
    Employees.*,
    Departments.Name
FROM
    Employees
JOIN
```

```
    Departments
    ON Departments.Id = Employees.DeptId
```

This will return a data set with all fields on the `Employee` table, followed by just the `Name` field in the `Departments` table:

Id	FName	LName	DeptId	Name
1	James	Smith	3	Finance
2	John	Johnson	4	IT

Warnings Against Use

It is generally advised that using `*` is avoided in production code where possible, as it can cause a number of potential problems including:

1. Excess IO, network load, memory use, and so on, due to the database engine reading data that is not needed and transmitting it to the front-end code. This is particularly a concern where there might be large fields such as those used to store long notes or attached files.
2. Further excess IO load if the database needs to spool internal results to disk as part of the processing for a query more complex than `SELECT <columns> FROM <table>`.
3. Extra processing (and/or even more IO) if some of the unneeded columns are:
    - computed columns in databases that support them
    - in the case of selecting from a view, columns from a table/view that the query optimiser could otherwise optimise out
4. The potential for unexpected errors if columns are added to tables and views later that results ambiguous column names. For example `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - if a column column called `displayname` is added to the orders table to allow users to give their orders meaningful names for future reference then the column name will appear twice in the output so the `ORDER BY` clause will be ambiguous which may cause errors ("ambiguous column name" in recent MS SQL Server versions), and if not in this example your application code might start displaying the order name where the person name is intended because the new column is the first of that name returned, and so on.

When Can You Use `*`, Bearing The Above Warning In Mind?

While best avoided in production code, using `*` is fine as a shorthand when performing manual queries against the database for investigation or prototype work.

Sometimes design decisions in your application make it unavoidable (in such circumstances, prefer `tablealias.*` over just `*` where possible).

When using `EXISTS`, such as `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)`, we are not returning any data from B. Thus a join is unnecessary, and the engine knows no values from B are to be returned, thus no performance hit for using `*`. Similarly `COUNT(*)` is fine as it also doesn't actually return any of the columns, so only needs to read and process those that are used for filtering purposes.

Section 6.2: SELECT Using Column Aliases

Column aliases are used mainly to shorten code and make column names more readable.

Code becomes shorter as long table names and unnecessary identification of columns *(e.g., there may be 2 IDs in the table, but only one is used in the statement)* can be avoided. Along with table aliases this allows you to use longer descriptive names in your database structure while keeping queries upon that structure concise.

Furthermore they are sometimes *required*, for instance in views, in order to name computed outputs.

All versions of SQL

Aliases can be created in all versions of SQL using double quotes (").

```
SELECT
    FName AS "First Name",
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Different Versions of SQL

You can use single quotes ('), double quotes (") and square brackets ([ ]) to create an alias in Microsoft SQL Server.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Both will result in:

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

This statement will return `FName` and `LName` columns with a given name (an alias). This is achieved using the `AS` operator followed by the alias, or simply writing alias directly after the column name. This means that the following query has the same outcome as the above.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

However, the explicit version (i.e., using the `AS` operator) is more readable.

If the alias has a single word that is not a reserved word, we can write it without single quotes, double quotes or brackets:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

FirstName	LastName
James	Smith
John	Johnson
Michael	Williams

A further variation available in MS SQL Server amongst others is **<alias> = <column-or-calculation>**, for instance:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

which is equivalent to:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Both will result in:

FullName	Addr1	Addr2
James Smith	123 AnyStreet	TownVille
John Johnson	668 MyRoad	Anytown
Michael Williams	999 High End Dr	Williamsburgh

Some find using = instead of `As` easier to read, though many recommend against this format, mainly because it is not standard so not widely supported by all databases. It may cause confusion with other uses of the = character.

All Versions of SQL

Also, if you *need* to use reserved words, you can use brackets or quotes to escape:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
FROM Employees
```

Different Versions of SQL

Likewise, you can escape keywords in MSSQL with all different approaches:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

SELECT	FROM	WHERE
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

Also, a column alias may be used any of the final clauses of the same query, such as an `ORDER BY`:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
```

```
    Employees
ORDER BY
    LastName DESC
```

However, you may *not* use

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

To create an alias from these reserved words (`SELECT` and `FROM`).

This will cause numerous errors on execution.

Section 6.3: Select Individual Columns

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

This statement will return the columns `PhoneNumber`, `Email`, and `PreferredContact` from all rows of the `Customers` table. Also the columns will be returned in the sequence in which they appear in the `SELECT` clause.

The result will be:

PhoneNumber	Email	PreferredContact
3347927472	william.jones@example.com	PHONE
2137921892	dmiller@example.net	EMAIL
NULL	richard0123@example.com	EMAIL

If multiple tables are joined together, you can select columns from specific tables by specifying the table name before the column name: `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

*`AS OrderId` means that the `Id` field of `Orders` table will be returned as a column named `OrderId`. See selecting with column alias for further information.

To avoid using long table names, you can use table aliases. This mitigates the pain of writing long table names for each field that you select in the joins. If you are performing a self join (a join between two instances of the *same* table), then you must use table aliases to distinguish your tables. We can write a table alias like `Customers c` or `Customers AS c`. Here `c` works as an alias for `Customers` and we can select let's say `Email` like this: `c.Email`.

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

Section 6.4: Selecting specified number of records

The SQL 2008 standard defines the `FETCH FIRST` clause to limit the number of records returned.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

This standard is only supported in recent versions of some RDMSs. Vendor-specific non-standard syntax is provided in other systems. Progress OpenEdge 11.x also supports the `FETCH FIRST <n> ROWS ONLY` syntax.

Additionally, `OFFSET <m> ROWS` before `FETCH FIRST <n> ROWS ONLY` allows skipping rows before fetching rows.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

The following query is supported in SQL Server and MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
```

To do the same in MySQL or PostgreSQL the `LIMIT` keyword must be used:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

In Oracle the same can be done with `ROWNUM`:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Results: **10** records.

```
Id    ProductName              UnitPrice            Package
38    Côte de Blaye            263.50               12 - 75 cl bottles
29    Thüringer Rostbratwurst  123.79               50 bags x 30 sausgs.
9    Mishi Kobe Niku           97.00                18 - 500 g pkgs.
```

```
20    Sir Rodney's Marmalade    81.00                30 gift boxes
18    Carnarvon Tigers          62.50                16 kg pkg.
59    Raclette Courdavault      55.00                5 kg pkg.
51    Manjimup Dried Apples     53.00                50 - 300 g pkgs.
62    Tarte au sucre            49.30                48 pies
43    Ipoh Coffee               46.00                16 - 500 g tins
28    Rössle Sauerkraut         45.60                25 - 825 g cans
```

Vendor Nuances:

It is important to note that the `TOP` in Microsoft SQL operates after the `WHERE` clause and will return the specified number of results if they exist anywhere in the table, while `ROWNUM` works as part of the `WHERE` clause so if other conditions do not exist in the specified number of rows at the beginning of the table, you will get zero results when there could be others to be found.

Section 6.5: Selecting with Condition

The basic syntax of SELECT with WHERE clause is:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

The *[condition]* can be any SQL expression, specified using comparison or logical operators like >, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN etc.

The following statement returns all columns from the table 'Cars' where the status column is 'READY':

```
SELECT * FROM Cars WHERE status = 'READY'
```

See WHERE and HAVING for more examples.

Section 6.6: Selecting with CASE

When results need to have some logic applied 'on the fly' one can use CASE statement to implement it.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

also can be chained

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

one also can have `CASE` inside another `CASE` statement

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
```

```
FROM TableName
```

Section 6.7: Select columns which are named after reserved keywords

When a column name matches a reserved keyword, standard SQL requires that you enclose it in double quotation marks:

```
SELECT
    "ORDER",
    ID
FROM ORDERS
```

Note that it makes the column name case-sensitive.

Some DBMSes have proprietary ways of quoting names. For example, SQL Server uses square brackets for this purpose:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

while MySQL (and MariaDB) by default use backticks:

```
SELECT
    `Order`,
    id
FROM orders
```

Section 6.8: Selecting with table alias

```
SELECT e.Fname, e.LName
FROM Employees e
```

The Employees table is given the alias 'e' directly after the table name. This helps remove ambiguity in scenarios where multiple tables have the same field name and you need to be specific as to which table you want to return data from.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Note that once you define an alias, you can't use the canonical table name anymore. i.e.,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

would throw an error.

It is worth noting table aliases -- more formally 'range variables' -- were introduced into the SQL language to solve the problem of duplicate columns caused by `INNER JOIN`. The 1992 SQL standard corrected this earlier design flaw by introducing `NATURAL JOIN` (implemented in mySQL, PostgreSQL and Oracle but not yet in SQL Server), the result of which never has duplicate column names. The above example is interesting in that the tables are joined on

columns with different names (`Id` and `ManagerId`) but are not supposed to be joined on the columns with the same name (`LName`, `FName`), requiring the renaming of the columns to be performed *before* the join:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Note that although an alias/range variable must be declared for the dervied table (otherwise SQL will throw an error), it never makes sense to actually use it in the query.

Section 6.9: Selecting with more than 1 condition

The `AND` keyword is used to add more conditions to the query.

Name	Age	Gender
Sam	18	M
John	21	M
Bob	22	M
Mary	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

This will return:

Name
John
Bob

using `OR` keyword

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

This will return:

name
Sam
John
Bob

These keywords can be combined to allow for more complex criteria combinations:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

This will return:

name
Sam
Mary

Section 6.10: Selecting without Locking the table

Sometimes when tables are used mostly (or only) for reads, indexing does not help anymore and every little bit counts, one might use selects without LOCK to improve performance.

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

where `UR` stands for "uncommitted read".

If used on table that has record modifications going on might have unpredictable results.

Section 6.11: Selecting with Aggregate functions

Average

The `AVG()` aggregate function will return the average of values selected.

```
SELECT AVG(Salary) FROM Employees
```

Aggregate functions can also be combined with the where clause.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Aggregate functions can also be combined with group by clause.

If employee is categorized with multiple department and we want to find avg salary for every department then we can use following query.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Minimum

The `MIN()` aggregate function will return the minimum of values selected.

```
SELECT MIN(Salary) FROM Employees
```

Maximum

The `MAX()` aggregate function will return the maximum of values selected.

```
SELECT MAX(Salary) FROM Employees
```

Count

The `COUNT()` aggregate function will return the count of values selected.

```
SELECT Count(*) FROM Employees
```

It can also be combined with where conditions to get the count of rows that satisfy specific conditions.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Specific columns can also be specified to get the number of values in the column. Note that `NULL` values are not counted.

```
Select Count(ManagerId) from Employees
```

Count can also be combined with the distinct keyword for a distinct count.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Sum

The `SUM()` aggregate function returns the sum of the values selected for all rows.

```
SELECT SUM(Salary) FROM Employees
```

Section 6.12: Select with condition of multiple values from column

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

This is semantically equivalent to

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

i.e. `value IN ( <value list> )` is a shorthand for disjunction (logical `OR`).

Section 6.13: Get aggregated result for row groups

Counting rows based on a specific column value:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Getting average income by department:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

The important thing is to select only columns specified in the `GROUP BY` clause or used with aggregate functions.

There `WHERE` clause can also be used with `GROUP BY`, but `WHERE` filters out records *before* any grouping is done:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

If you need to filter the results after the grouping has been done, e.g, to see only departments whose average income is larger than 1000, you need to use the `HAVING` clause:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Section 6.14: Selection with sorted Results

```
SELECT * FROM Employees ORDER BY LName
```

This statement will return all the columns from the table `Employees`.

Id	FName	LName	PhoneNumber
2	John	Johnson	2468101214
1	James	Smith	1234567890
3	Michael	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Or

```
 SELECT * FROM Employees ORDER BY LName ASC
```

This statement changes the sorting direction.

One may also specify multiple sorting columns. For example:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

This example will sort the results first by `LName` and then, for records that have the same `LName`, sort by `FName`. This will give you a result similar to what you would find in a telephone book.

In order to save retyping the column name in the `ORDER BY` clause, it is possible to use instead the column's number. Note that column numbers start from 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

You may also embed a `CASE` statement in the `ORDER BY` clause.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones' THEN 0 ELSE 1
END ASC
```

This will sort your results to have all records with the `LName` of "Jones" at the top.

Section 6.15: Selecting with null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

Selection with nulls take a different syntax. Don't use `=`, use `IS NULL` or `IS NOT NULL` instead.

Section 6.16: Select distinct (unique values only)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

This query will return all `DISTINCT` (unique, different) values from `ContinentCode` column from `Countries` table

ContinentCode
OC
EU

AS
NA
AF

SQLFiddle Demo

Section 6.17: Select rows from multiple tables

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

This is called cross product in SQL it is same as cross product in sets

These statements return the selected columns from multiple tables in one query.

There is no specific relationship between the columns returned from each table.