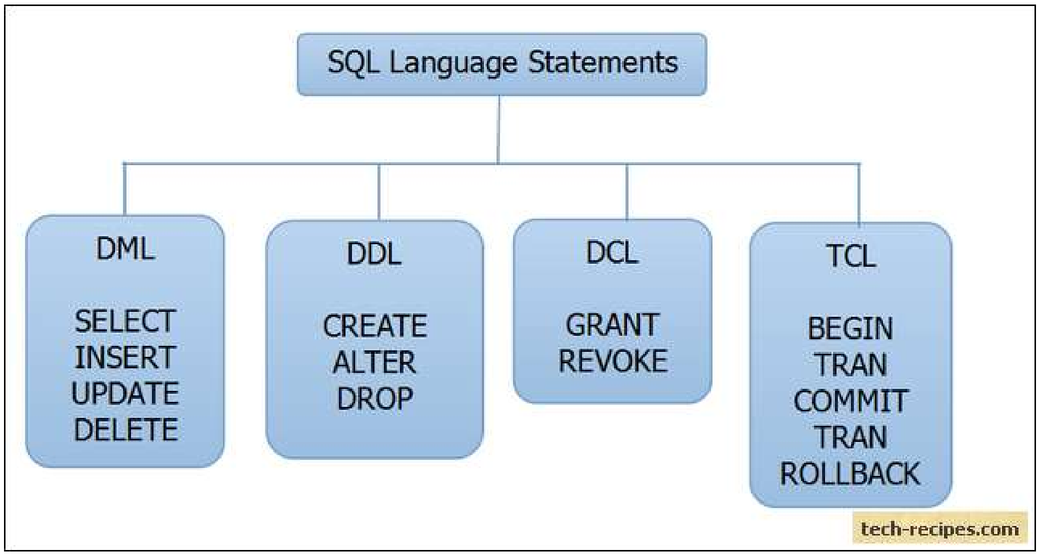
SHANIMZ ACADEMY

SQL - Syllabus

Sql introduction,Sql Datatypes, Sql basics-DDL,DML,DCL,Comparison Operators,Constraints,Aggregate Functions,Practice questions,Joins,Stored Procedures,Practice Question,Views,Group By,Having ,Order By Clauses, Practice Question.

Day 4-SQL



DCL (Data Control Language):

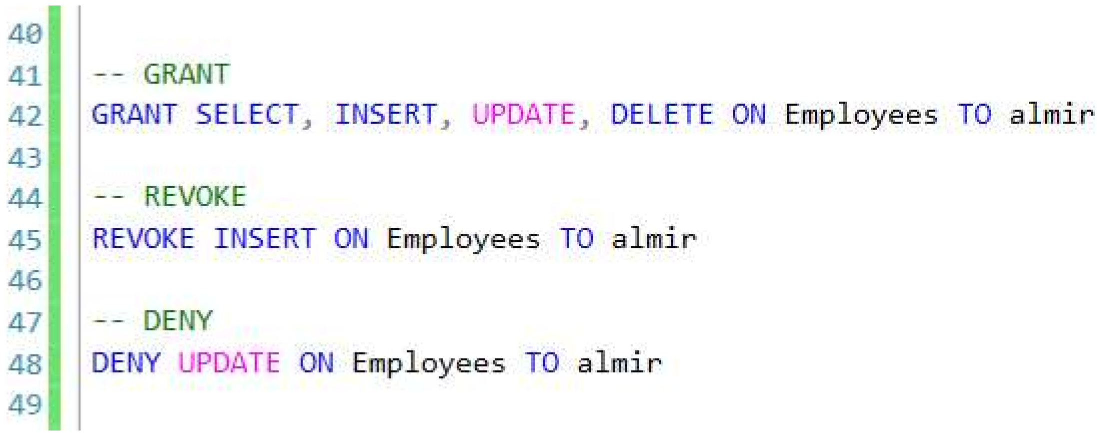
DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

GRANT: This command gives users access privileges to the database.

REVOKE: This command withdraws the user's access privileges given by using the GRANT command.

DENY is a specific command. We can conclude that every user has a list of privilege which is denied or granted so command DENY is there to explicitly ban you some privileges on the database objects.



```

-- GRANT
GRANT SELECT, INSERT, UPDATE, DELETE ON Employees TO almir

-- REVOKE
REVOKE INSERT ON Employees TO almir

-- DENY
DENY UPDATE ON Employees TO almir

```

TCL(Transaction Control Language):

Though many resources claim there to be another category of SQL clauses TCL – Transaction Control Language.

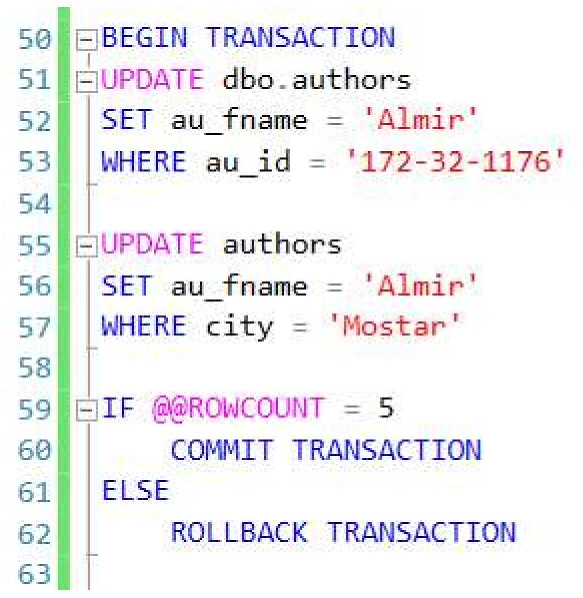
With TCL commands we can mange and control T-SQL transactions so we can be sure that our transaction is successfully done and that integrity of our database is not violated.

TCL commands are:

BEGIN TRAN - begin of transaction

COMMIT TRAN - commit for completed transaction

ROLLBACK - go back to beginning if something was not right in transaction.



```
BEGIN TRANSACTION
UPDATE dbo.authors
SET au_fname = 'Almir'
WHERE au_id = '172-32-1176'

UPDATE authors
SET au_fname = 'Almir'
WHERE city = 'Mostar'

IF @@ROWCOUNT = 5
    COMMIT TRANSACTION
ELSE
    ROLLBACK TRANSACTION

```

```
------------Rename Column name -----------
sp_rename 'Register1.email','emailid','Column';
```

Comparison Operators in SQL Server

=	Equal to
<>	Not equal to
!=	Not equal to(non-ISO standard
>	Greater than
>=	Greater than or equal to
!>	Not greater than(non-ISO standard)
<	Less than
<=	Less than or equal to
!<	Not less than(non-ISO standard)

SQL Server Constraints

Constraints are the predefined set of rules and restrictions applied on the tables or columns for restricting unauthorised values to be inserted into the tables. They are responsible for ensuring the column's data accuracy, integrity, and reliability inside the table. Constraints also tell that the data will be inserted into the table when the inserted data satisfies the constraint rule. Otherwise, the insert operation will be terminated if the inserted data violates the defined constraint.

SQL Server categorises the constraints into two types:

Table level constraints: These constraints apply to the entire table that limit the types of data that can be entered into the table. Its definitions are specified after creating the table using the ALTER statement.

Column level constraints: These constraints apply to the single or multiple columns to limit the types of data that can be entered into the column. Its definition is specified while creating the tables.

Constraints used in SQL Server

The following are the most common constraints used in the SQL Server that we will describe deeply with examples:

NOT NULL

UNIQUE

PRIMARY KEY : This constraint consists of one or more columns with values and identifies each record in a table uniquely.

FOREIGN KEY :A foreign key is a database key that links two tables together

CHECK : This constraint is used to limit the range of values in a column. It ensures that all the inserted values in a column must follow the specific rule.

Ex:

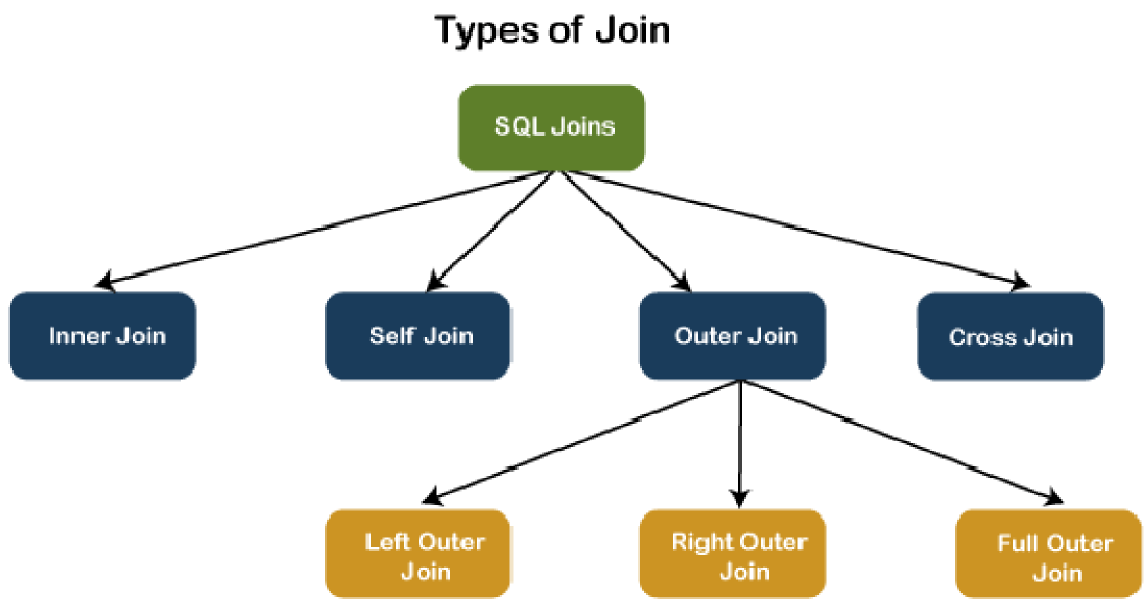
```
CREATE TABLE Sales ( Id int NOT NULL, Amount int NOT NULL, Vendor_Name varchar(255) );

ALTER TABLE Sales ADD CONSTRAINT UQ__Constrai UNIQUE (Vendor_Name);

ALTER TABLE Sales DROP CONSTRAINT UQ__Constrai;
```

```
CREATE TABLE salary_info ( ID INT PRIMARY KEY, Name VARCHAR(250) NOT NULL, Salary INT
    CHECK (Salary>10000) )
```

Joins in SQL Server

Joins are the types of a single concept, which allows the joining of two or more tables using a defined syntax in SQL programming. Joining of the tables being facilitated through a common field which is present in each of the tables, either by same or different names, and the joins being characterized into various types, based on the number and the nature of records extracted from the tables by the SQL query, such as inner join, left outer join, right outer join, full outer join, and self-outer join, etc., are termed as types of joins in SQL Server.

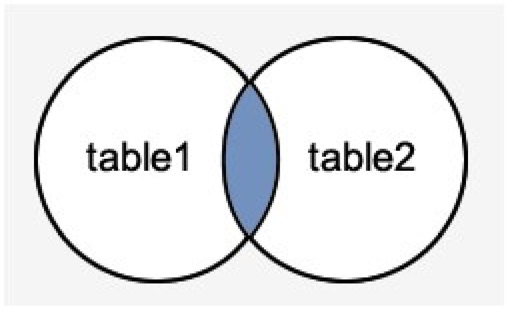
There are different types of Joins:



INNER JOINS

This JOIN returns all records from multiple tables that satisfy the specified join condition. It is the simple and most popular form of join and assumes as a **default join**. If we omit the INNER keyword with the JOIN query, we will get the same output.

The following visual representation explains how INNER JOIN returns the matching records from **table1** and **table2:**



INNER JOIN Syntax

The following syntax illustrates the use of INNER JOIN in SQL Server:

```
SELECT columns
FROM table1
INNER JOIN table2 ON condition1
INNER JOIN table3 ON condition2
```

INNER JOIN Example

Let us first create two tables "**Student**" and "**Fee**" using the following statement:

```
CREATE TABLE Student (
  id int PRIMARY KEY IDENTITY,
  admission_no varchar(45) NOT NULL,
  first_name varchar(45) NOT NULL,
  last_name varchar(45) NOT NULL,
  age int,
  city varchar(25) NOT NULL
);
  ----------------------------------------------
```

```
CREATE TABLE Fee (
  admission_no varchar(45) NOT NULL,
  course varchar(45) NOT NULL,
  amount_paid int,
);
```

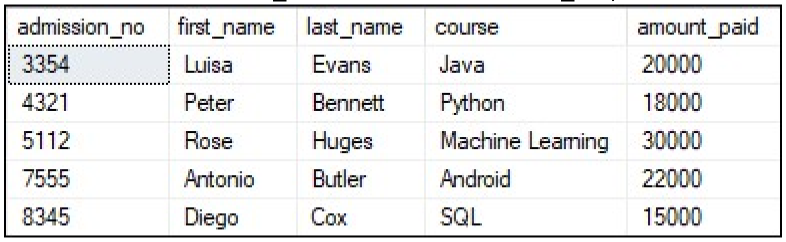
Insert values to the table

```
INSERT INTO Student (admission_no, first_name, last_name, age, city)
VALUES (3354,'Luisa', 'Evans', 13, 'Texas'),
(2135, 'Paul', 'Ward', 15, 'Alaska'),
(4321, 'Peter', 'Bennett', 14, 'California'),
(4213,'Carlos', 'Patterson', 17, 'New York'),
(5112, 'Rose', 'Huges', 16, 'Florida'),
(6113, 'Marielia', 'Simmons', 15, 'Arizona'),
(7555,'Antonio', 'Butler', 14, 'New York'),
(8345, 'Diego', 'Cox', 13, 'California');
  ----------------------------------------------
INSERT INTO Fee (admission_no, course, amount_paid)
VALUES (3354,'Java', 20000),
(7555, 'Android', 22000),
(4321, 'Python', 18000),
(8345,'SQL', 15000),
(5112, 'Machine Learning', 30000);
```

Execute the **SELECT** statement to verify the records:

We can demonstrate the INNER JOIN using the following command:

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fe
e.amount_paid
FROM Student
INNER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```



admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
4321	Peter	Bennett	Python	18000
5112	Rose	Huges	Machine Learning	30000
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

In this example, we have used the **admission_no column** as a join condition to get the data from both tables. Depending on this table, we can see the information of the students who have paid their fee.

SELF JOIN

A table is joined to itself using the SELF JOIN. It means that **each table row is combined with itself** and with every other table row. The SELF JOIN can be thought of as a JOIN of two copies of the same tables. We can do this with the help of **table name aliases** to assign a specific name to each table's instance. The table aliases enable us to use the **table's temporary** name that we are going to use in the query. It's a useful way to extract hierarchical data and comparing rows inside a single table.

SELF JOIN Syntax

The following expression illustrates the syntax of SELF JOIN in SQL Server. It works the same as the syntax of joining two different tables. Here, we use aliases names for tables because both the table name are the same.

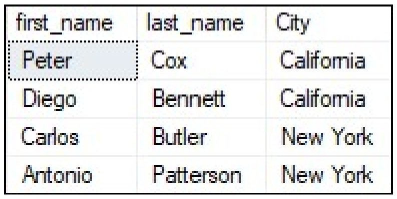
```
SELECT T1.col_name, T2.col_name...
FROM table1 T1, table1 T2
WHERE join_condition;
```

Example

We can demonstrate the SELF JOIN using the following command:

```
SELECT S1.first_name, S2.last_name, S2.city
FROM Student S1, Student S2
WHERE S1.id <> S2.iD AND S1.city = S2.city
ORDER BY S2.city;
```

This command gives the below result:



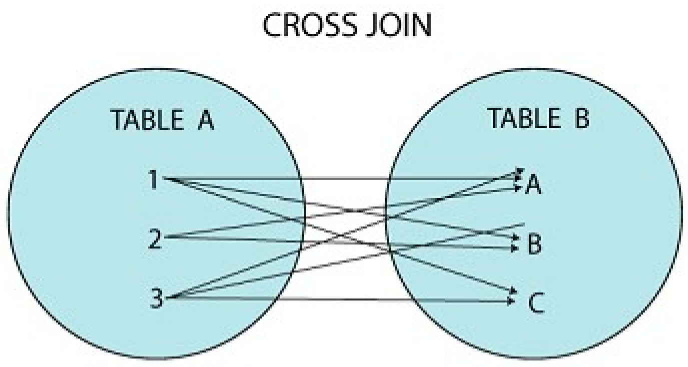
first_name	last_name	City
Peter	Cox	California
Diego	Bennett	California
Carlos	Butler	New York
Antonio	Patterson	New York

In this example, we have used the **id and city column** as a join condition to get the data from both tables.

CROSS JOIN

CROSS JOIN in SQL Server combines all of the possibilities of two or more tables and returns a result that includes every row from all contributing tables. It's also known as **CARTESIAN JOIN** because it produces the **Cartesian product** of all linked tables. The Cartesian product represents all rows present in the first table multiplied by all rows present in the second table.

The below visual representation illustrates the CROSS JOIN. It will give all the records from **table1** and **table2** where each row is the combination of rows of both tables:



CROSS JOIN Syntax

The following syntax illustrates the use of CROSS JOIN in SQL Server:

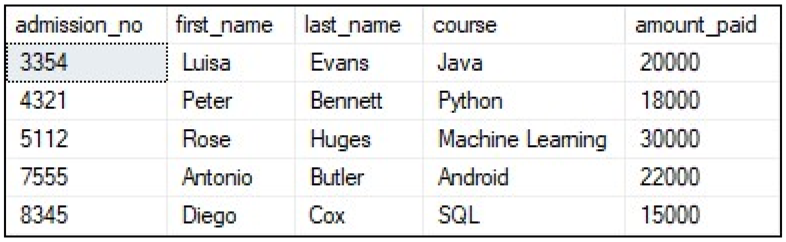
```
SELECT column_lists
FROM table1
CROSS JOIN table2;
```

Example

We can demonstrate the CROSS JOIN using the following command:

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fe
e.amount_paid
FROM Student
CROSS JOIN Fee
WHERE Student.admission_no = Fee.admission_no;
```

This command gives the below result:



admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
4321	Peter	Bennett	Python	18000
5112	Rose	Huges	Machine Learning	30000
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

OUTER JOIN

OUTER JOIN in SQL Server **returns all records from both tables** that satisfy the join condition. In other words, this join will not return only the matching record but also return all unmatched rows from one or both tables.

We can categories the OUTER JOIN further into three types:

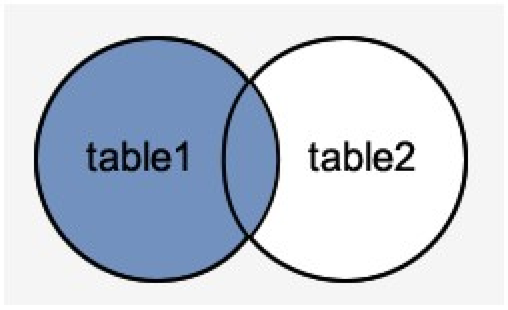
- LEFT OUTER JOIN

- RIGHT OUTER JOIN
- FULL OUTER JOIN

LEFT OUTER JOIN

The LEFT OUTER JOIN **retrieves all the records from the left table and matching rows from the right table**. It will return **NULL** when no matching record is found in the right side table. Since OUTER is an optional keyword, it is also known as LEFT JOIN.

The below visual representation illustrates the LEFT OUTER JOIN:



LEFT OUTER JOIN Syntax

The following syntax illustrates the use of LEFT OUTER JOIN in SQL Server:

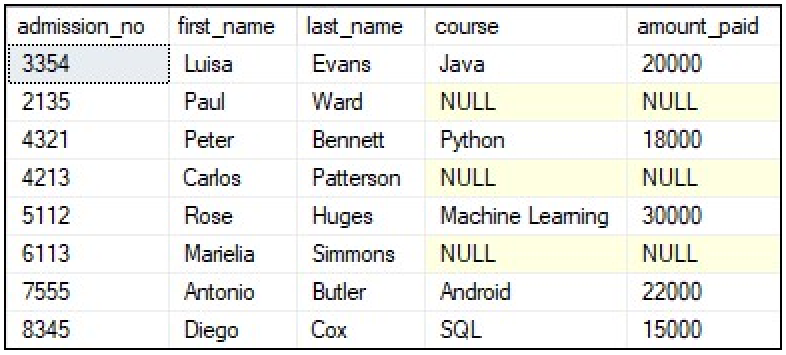
```
SELECT column_lists
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

Example

We can demonstrate the LEFT OUTER JOIN using the following command:

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fe
e.amount_paid
FROM Student
LEFT OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

This command gives the below result:



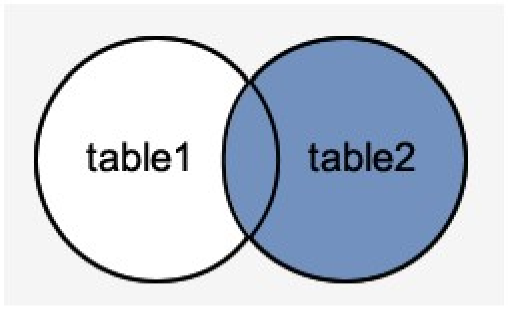
admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
2135	Paul	Ward	NULL	NULL
4321	Peter	Bennett	Python	18000
4213	Carlos	Patterson	NULL	NULL
5112	Rose	Huges	Machine Learning	30000
6113	Marielia	Simmons	NULL	NULL
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

This output shows that the unmatched row's values are replaced with NULLs in the respective columns.

RIGHT OUTER JOIN

The RIGHT OUTER JOIN **retrieves all the records from the right-hand table and matched rows from the left-hand table**. It will return **NULL** when no matching record is found in the left-hand table. Since OUTER is an optional keyword, it is also known as RIGHT JOIN.

The below visual representation illustrates the RIGHT OUTER JOIN:



RIGHT OUTER JOIN Syntax

The following syntax illustrates the use of RIGHT OUTER JOIN in SQL Server:

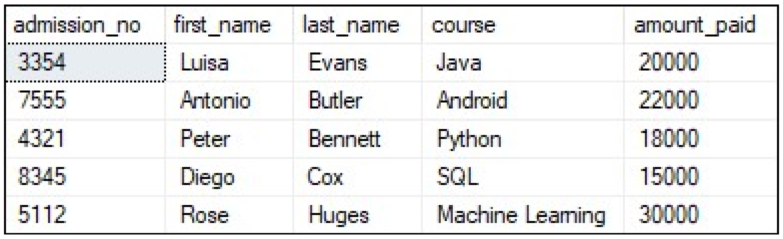
```
SELECT column_lists
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

Example

The following example explains how to use the RIGHT OUTER JOIN to get records from both tables:

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fe
e.amount_paid
FROM Student
RIGHT OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

This command gives the below result:



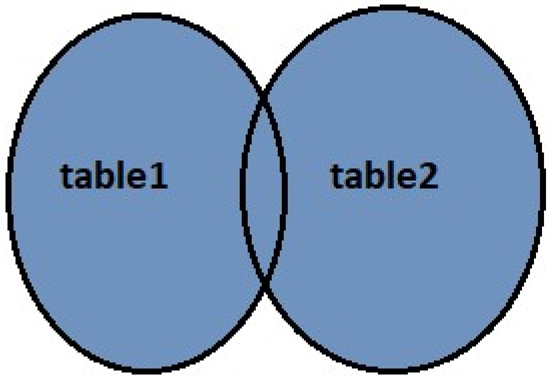
admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
7555	Antonio	Butler	Android	22000
4321	Peter	Bennett	Python	18000
8345	Diego	Cox	SQL	15000
5112	Rose	Huges	Machine Learning	30000

In this output, we can see that no column has NULL values because all rows in the Fee table are available in the Student table based on the specified condition.

FULL OUTER JOIN

The FULL OUTER JOIN in SQL Server **returns a result that includes all rows from both tables**. The columns of the right-hand table return NULL when no matching records are found in the left-hand table. And if no matching records are found in the right-hand table, the left-hand table column returns NULL.

The below visual representation illustrates the FULL OUTER JOIN:



FULL OUTER JOIN Syntax

The following syntax illustrates the use of FULL OUTER JOIN in SQL Server:

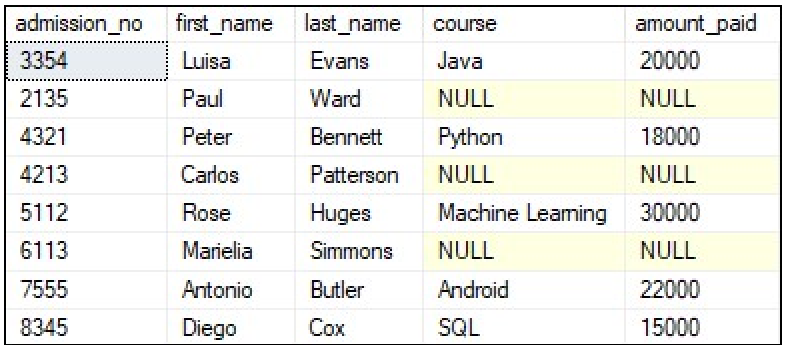
```
SELECT column_lists
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

Example

The following example explains how to use the FULL OUTER JOIN to get records from both tables:

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fe
e.amount_paid
FROM Student
FULL OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

This command gives the below result:



admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
2135	Paul	Ward	NULL	NULL
4321	Peter	Bennett	Python	18000
4213	Carlos	Patterson	NULL	NULL
5112	Rose	Huges	Machine Learning	30000
6113	Marielia	Simmons	NULL	NULL
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

In this output, we can see that the column has NULL values when no matching records are found in the left-hand and right-hand table based on the specified condition.

SQL Stored Procedures for SQL Server

What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Execute a Stored Procedure

```
EXEC procedure_name;
```

Example

```
CREATE PROCEDURE SelectAllStudents
AS
SELECT * FROM Student
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllStudents;
```

Stored Procedure With One Parameter

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

Example

```
USE Shanimz
GO

CREATE PROCEDURE StudPro @fname nvarchar(30)
AS
SELECT * FROM Student WHERE first_name=@fname;

EXEC StudPro @fname = 'Rose';
```

Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode
nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the `CREATE VIEW` statement.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

Example

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

We can query the view above as follows:

Example

```
SELECT * FROM [Products Above Average Price];
```

SQL Updating a View

A view can be updated with the `CREATE OR REPLACE VIEW` statement.

SQL CREATE OR REPLACE VIEW Syntax

```
CREATE OR ALTER VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR ALTER VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

SQL Dropping a View

A view is deleted with the `DROP VIEW` statement.

SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

The SQL GROUP BY Statement

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.

GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```