SQL

Structured Query Language

A note by
Shah Md. Arshad Rahman Ziban

Chapter 00: Basic Info

What is database?

A database is a collection of information that is organized so that it can be easily accessed, managed, and updated. In more technical terms, a database is an electronic system that allows data to be stored, retrieved, updated, and managed in an efficient and secure manner. Databases are crucial for handling vast amounts of information systematically, where data can be quickly searched, sorted, and analyzed.

Example:

Category	Database	Description
Relational Databases	MySQL	Popular open-source relational database.
	PostgreSQL	Advanced open-source relational database.
	Oracle	Enterprise-level relational database.
	Microsoft SQL Server	Relational database by Microsoft.
NoSQL Databases	MongoDB	Document-based NoSQL database.
	Cassandra	Scalable, distributed NoSQL database.
	Redis	In-memory key-value store.
Cloud Databases	Amazon RDS	Managed relational database service.
	Google Cloud Firestore	Flexible cloud database for mobile and web.
	Azure Cosmos DB	Globally distributed multi-model database.
NewSQL Databases	CockroachDB	Distributed SQL database.
	Google Spanner	Globally distributed SQL database.

What is DBMS?

A Database Management System (DBMS) is system software that enables the creation, management, and manipulation of databases. It provides an interface between the database and the user, allowing for the organization, storage, retrieval, and management of data.

Key functions of a DBMS include:

Data Storage	Efficiently stores data in a structured format.
Data Retrieval	Allows users to query and retrieve data using languages like SQL.
Data Manipulation	Supports operations such as inserting, updating, and deleting data.
Data Security	Implements access controls to protect data from unauthorized access.
Data Integrity	Ensures accuracy and consistency of data through constraints and validation rules.
Backup and Recovery	Provides mechanisms for data backup and recovery in case of failures.

Common types of DBMS include relational databases (like MySQL and Oracle), NoSQL databases (like MongoDB), and object-oriented databases.

Here's a table that highlights the differences between a Database and a Database Management

Aspect	Database	DBMS
Definition	A collection of organized data.	Software to manage and interact with the database.
Purpose	To store data efficiently.	To provide tools and features for data management and access.
Components	Tables, records, fields, etc.	Query processor, database engine, transaction management, etc.
Interaction	Direct interaction with stored data.	Provides an interface to interact with the database.
Data Management	Data needs to be managed manually or with simple tools.	Automates data management, including backup, security, and multi-user access.

What is SQL?

SQL stands for Structured Query Language. It is a powerful and versatile database language used for storing, manipulating, and retrieving data in databases. SQL allows users to perform various operations, including:

1. **DML** (Data Manipulation Language)
Select, insert, update, delete etc.

2. **DDL** (Data Definition Language)
create, alter, drop, truncate, rename etc.

3. **DCL** (Data Control Language)
comment etc

Chapter 01: Queries

Create Database

Syntax:

```
CREATE DATABASE database_name;
```

Example:

```
CREATE DATABASE sampleDB;
```

Use Database

Syntax:

```
USE database_name;
```

Example:

```
USE sampleDB;
```

Create Table

Syntax:

```
CREATE TABLE table_name (
   column1 datatype ,
   column2 datatype ,
   column3 datatype,
   ...
);
```

Example:

```
CREATE TABLE Employee (
  id INT,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50),
  email VARCHAR(100),
  hire_date DATE
);
```

The NOT NULL constraint in SQL is used to ensure that a column cannot hold a NULL value.

Insert Data

Syntax:

```
NSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

Example:

```
INSERT INTO Employee (id, first_name, last_name, email, hire_date)
VALUES
  (1, 'John', 'Doe', 'john.doe@.com', '2023-01-15'),
  (2, 'Jane', 'Smith', 'jane.smith@.com', '2022-09-20'),
```

Show a Table

Syntax:

```
SELECT * FROM Employee
```

Example:

id	first_name	last_name	email	hire_date
1	John	Doe	john.doe@.com	2023-01-15
2	John	Smith	jane.smith@.com	2022-09-20

Deletion of Tuples

Using DELETE

Row delete

```
DELETE FROM table_name
WHERE condition;
```

Example

```
DELETE FROM employees
WHERE id = 2;
```

Table delete

```
DELETE FROM table_name;
```

Clears data but keeps the table. Optionally, you can specify conditions to delete specific rows.

Using DROP

Column delete

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example

```
ALTER TABLE Employees
DROP COLUMN hire_date;
```

Table delete

```
DROP TABLE table_name;
```

Permanently deletes the table and all its data, along with its structure.

Adding a New Column

Syntax:

```
ALTER TABLE table_name
ADD COLUMN column_name datatype;
```

Example:

```
ALTER TABLE Employees
ADD COLUMN Email VARCHAR(255);
```

Modifying an Existing Column

Data type modify

Syntax:

```
ALTER TABLE table_name
ALTER COLUMN column_name new_datatype;
```

Example:

```
ALTER TABLE employees
ALTER COLUMN salary DECIMAL(10, 2);
```

What Does DECIMAL(10, 2) Mean?

1. The total number of digits (precision) is 10.
2. 2 digits are reserved for the fractional part(after the decimal point).
3. 8 digits are available for the integer part (before the decimal point).

Name modify

Syntax:

```
EXEC sp_rename 'table_name.old_column_name',
'new_column_name', 'COLUMN';
```

Example:

```
EXEC sp_rename 'employees.lastname', 'last_name',
'COLUMN';
```

We can aslo use this

```
SELECT FirstName AS First, LastName AS Last
FROM Employees;
```

Selecting Specific Columns

Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

Example:

```
SELECT FirstName, LastName
FROM Employees;
```

Concatenating Columns

Syntax:

```
SELECT data_1 + ' ' + data_2 AS merged_data
FROM table_name;
```

Example:

```
SELECT first_name + ' ' + last_name AS Full_name
FROM Employee;
```

Full_name
John Doe
John Smith

In this case we don't need to create a separate column called Full_name. This query will show us a table like the one on the left and will not have any effect on the main table.

Syntax:

```
UPDATE table_name
SET FullName = data_1 + ' ' + data_2;
```

Example:

```
UPDATE table_name
SET Full_name = first_name + ' ' + last_name;
```

```
ALTER TABLE Employees
ADD Full_name VARCHAR(101);

UPDATE table_name
SET Full_name = first_name + ' ' + last_name;
```

id	first_name	last_name	email	hire_date	Full_name
1	John	Doe	john.doe@.com	2023-01-15	John Doe
2	John	Smith	jane.smith@.com	2022-09-20	John Smith

Here another column called Full_name will be added to the main table

Filtering Results

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
SELECT email, hire_date
FROM Employee
WHERE last_name = 'Smith"
```

```
SELECT *
FROM Employees
WHERE first_name = 'Jhon';
```

This query retrieves all columns (*) for each row in the Employees table where the first_name column matches 'John'. Essentially, it returns every piece of information available in the Employees table for those employees whose first name is John.

Sorting Results

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

Example:

```
SELECT name, salary
FROM Employees
ORDER BY salary DESC, name ASC;
```

```
SELECT *
FROM Employees
ORDER BY last_name ASC;
```

```
SELECT first_name, last_name, hire_date
FROM employees
ORDER BY YEAR(hire_date) DESC;
```

String Matching

% represents zero or more characters or both.
_ represents a single character.

Example:

```
SELECT *
FROM Employees
WHERE last_name LIKE '%ith';
```

% is a wildcard character that matches any sequence of characters, including no characters at all.

```
SELECT *
FROM Employees
WHERE first_name LIKE 'John%';
```

```
SELECT *
FROM Employees
WHERE email LIKE '%example%';
```

```
SELECT first_name
FROM Employees
WHERE first_name LIKE '__n%';
```

_ (underscore) is a wildcard character that matches exactly one character.

```
SELECT *
FROM Employees
WHERE last_name LIKE 'S__h';
```

Set Operations

Overview of SQL set operations:

Combine Results : UNION, INTERSECT, and EXCEPT merge results from multiple SELECT queries.
Column Match : All SELECT queries must have the same number of columns with compatible data types.
Duplicates :
UNION removes duplicates; UNION ALL includes them.
INTERSECT and EXCEPT also remove duplicates.
Unordered : Results are unordered unless you use ORDER BY.
Distinct Rows : Results are distinct by default, except with UNION ALL.

table1

column1
1
2
3

table2

column1
3
4
5

UNION

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

column1
1
2
3
4
5

UNION ALL

```
SELECT column1 FROM table1
UNION ALL
SELECT column1 FROM table2;
```

column1
1
2
3
3
4
5

INTERSECT

```
SELECT column1 FROM table1
INTERSECT
SELECT column1 FROM table2;
```

column1
3

EXCEPT

```
SELECT column1 FROM table1
EXCEPT
SELECT column1 FROM table2;
```

column1
1
2

Aggregate Functions

AVG() - Average Price

```
SELECT AVG(Price) FROM Sales;
```

AVG(Price)
475

COUNT() - Total Sales

```
SELECT COUNT(*) FROM Sales;
```

COUNT(*)
5

MAX() - Max Price

```
SELECT MAX(Price) FROM Sales;
```

MAX(Price)
1000

MIN() - Min Price

```
SELECT MIN(Price) FROM Sales;
```

MIN(Price)
25

SUM() - Total Revenue

```
SELECT SUM(Price * Quantity) FROM Sales;
```

SUM(Price * Quantity)
2700

Table: Sales

SaleID	Product	Quantity	Price
1	A	2	1000
2	B	3	300
3	C	5	50
4	D	3	25
5	A	1	1000

Aggregate functions ignore NULL values,except for COUNT(*), which counts every row.

GROUP BY Clause

Aggregate functions are often used with the GROUP BY clause to group the result set by one or more columns. This is useful for performing calculations on each group separately.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Table: Sales

SaleID	Product	Quantity	Price
1	A	2	1000
2	B	3	300
3	C	5	50
4	D	3	25
5	A	1	1000

Examples:

```
SELECT Product, SUM(Quantity) AS TotalQuantity
FROM Sales
GROUP BY Product;
```

Product	TotalQuantity
A	3
B	3
C	5
D	3

```
SELECT Product, SUM(Quantity * Price) AS TotalSales
FROM Sales
GROUP BY Product;
```

Product	TotalSales
A	3000
B	900
C	250
D	75

```
SELECT Product, AVG(Price) AS AveragePrice
FROM Sales
GROUP BY Product;
```

Product	AveragePrice
A	1000
B	300
C	50
D	25

HAVING Clause

The HAVING clause is used to filter groups based on the result of an aggregate function. It is similar to the WHERE clause but is used after grouping the data.

Example:

```
SELECT Product, SUM(Quantity * Price) AS TotalRevenue
FROM Sales
GROUP BY Product
HAVING SUM(Quantity * Price) > 1500;
```

Product	TotalRevenue
A	3000

Null Values

To find rows where a column is NULL :

Syntax:

```
SELECT * FROM table_name WHERE column_name IS NULL;
```

These queries help in identifying rows that have missing data or ensuring that only rows with complete data are retrieved.

To find rows where a column is not NULL :

Syntax:

```
SELECT * FROM table_name WHERE column_name IS NOT NULL;
```

Inserting NULL Values :

Syntax:

```
INSERT INTO table_name (column1, column2) VALUES ('value1', NULL);
```

```
-- Ignores NULL values
SELECT AVG(column_name) FROM table_name;

-- Includes rows with NULL values
SELECT COUNT(*) FROM table_name;
```

Sub queries

Subqueries in SQL are queries nested within another SQL query. They can be used in various parts of a query, including the SELECT, FROM, and WHERE clauses. Subqueries can be particularly powerful when combined with IN and ALL constructs, allowing for complex data retrieval based on specific conditions.

Table: Sales

SaleID	Product	Quantity	Price
1	A	2	1000
2	B	3	300
3	C	5	50
4	D	3	25
5	A	1	1000

Price
1000
300
50
25
900

Consider this for Subquery with EXISTS

Subqueries with the IN Construct

if we want to find all sales of products that have prices matching the price of product "A"

```
SELECT *
FROM Sales
WHERE Price IN (SELECT Price FROM Sales WHERE Product = 'A');
```

SaleID	Product	Quantity	Price
1	A	2	1000
5	A	1	1000

Subquery with NOT IN

To find all sales where the product's price does not match the price of product "A"

```
SELECT *
FROM Sales
WHERE Price NOT IN (SELECT Price FROM Sales WHERE Product = 'A');
```

SaleID	Product	Quantity	Price
2	B	3	300
3	C	5	50
4	D	3	75

Subquery with ALL

To find sales where the price is greater than the price of all products in the table

```
SELECT *
FROM Sales
WHERE Price > ALL (SELECT Price FROM Sales );
```

SaleID	Product	Quantity	Price
1	A	2	1000
5	A	1	1000

Subquery with EXISTS

To find all sales where there exists another sale of the same product with a different price

```
SELECT *
FROM Sales S1
WHERE EXISTS (
   SELECT *
   FROM Sales S2
   WHERE S1.Product = S2.Product AND S1.Price <> S2.Price );
```

Since there's no difference in prices for Product "A", the query doesn't find any rows that match the condition. No rows will be returned for Product "A".

```
SELECT *
FROM Sales S1
WHERE EXISTS (
   SELECT *
   FROM Sales S2
   WHERE S1.Product = S2.Product AND S1.Price <> S2.Price );
```

SaleID	Product	Quantity	Price
1	A	2	1000
5	A	1	900

Table Aliases (S1 and S2):

1. S1 and S2 are both aliases for the Sales table. These aliases allow you to reference the same table twice within the query.

2. S1 represents the outer query, which is checking each row of the Sales table.

3. S2 represents the inner query, which is used to compare rows within the table.

UPDATE Statement

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Table: Sales

SaleID	Product	Quantity	Price
1	A	2	1000
2	B	3	300
3	C	5	50
4	D	3	25
5	A	1	1000

Example: update single column

```
UPDATE Sales
SET Quantity = 4
WHERE SaleID = 3;
```

Table: Sales

SaleID	Product	Quantity	Price
1	A	2	1000
2	B	3	300
3	C	4	60
4	D	3	25
5	A	1	1000

Example: update multiple columns

```
UPDATE Sales
SET Quantity = 4, Price = 60
WHERE SaleID = 3;
```

Chapter 02: KEYS

Definition

Database keys are attributes or sets of attributes in a relational database that are used to uniquely identify records within a table and establish relationships between tables. They are crucial for maintaining the integrity, consistency, and efficiency of the data in the database.

Different Types of keys in DBMS with example

1. Super Key

Definition: A super key is any set of one or more columns that can uniquely identify a record in a table. A super key can have extra columns that are not necessary for uniqueness.

Purpose: Represents all possible sets of columns that can uniquely identify a record.

Example:

Emp_id	Emp_Name	Email	Joining Date
1	Jhon	jhonuk@gmail.com	11/11/2019
2	Michle	Michleuk@gmail.com	16/08/2019
4	Jhon	Jhon22uk@gmail.com	16/11/2019
5	Sara	Sarauk@gmail.com	11/11/2019

For the above table the possible super keys are: {Emp_id}, {Email}, {Emp_id, Emp_Name, Email}, {Emp_Name, Email, JoiningDate} etc.

But EmpName & JoiningDate can not work as superkey because if you use these columnstoidentifyUnique values then you can not do it because there remains duplicate values.

SQL Queries for Super Key :

Using Emp_id and Email as a Super Key

```
SELECT * FROM Employee WHERE Emp_id = 1 AND Email = 'jhonuk@gmail.com';
```

Using Emp_id, Emp_Name, and Email as a Super Key

```
SELECT * FROM Employee WHERE Emp_id = 1 AND Emp_Name = 'Jhon' AND Email = 'jhonuk@gmail.com';
```

2. Candidate Key

Definition: A candidate key is a column or a group of columns that can potentially be used as a primary key. A table can have multiple candidate keys, but only one can be chosen as the primary key. A candidate key can never be NULL or empty. And its value should be unique.

Purpose: Represents a set of columns that uniquely identify a record.

Example:

Suppose, the super key is: {Emp_id, Emp_Name,Email}. We can use {Emp_id, Emp_Name} or {Emp_id,Email} as a candidate key
But candidate key is the minimal set of Super Key. {Emp_id, EmpName} from here we get {Emp_id} , {EmpName} but {EmpName} is not a candidate key. Again, {Emp_id,Email} from here we get {Emp_id} , {Email}.
Both can be called candidate key.

SQL Queries for Candidate Key :

Using Emp_id as a Candidate Key

```
SELECT * FROM Employee WHERE Emp_id = 1;
                    OR
SELECT DISTINCT Emp_id FROM Employee;
```

3. Alternate Key

Definition: Alternate keys is a column or group of columns in a table that uniquely identify every row in that table. A table can have multiple choices for a primary key but only one canbesetastheprimary key.
All the keys which are not primary key are called an Alternate Key.

Purpose: It serves as a unique identifier for records, similar to a primary key, but is not the primary key.

Example:

Emp_id	Emp_Name	Email	Joining Date
1	Jhon	jhonuk@gmail.com	11/11/2019
2	Michle	michleuk@gmail.com	16/08/2019
4	Jhon	jhon22uk@gmail.com	16/11/2019
5	Sara	sarauk@gmail.com	11/11/2019

In this table, Emp_id & Email are qualified to become a primary key. But since Emp_id is the primary key, so Email becomes the alternative key.

SQL Queries for Alternate Key :

Using Email as a Alternate Key

```
SELECT * FROM Employee WHERE Email = 'jhonuk@gmail.com';
                    OR
SELECT DISTINCT Email FROM Employee;
```

4. Primary Key

Definition: A primary key is a unique identifier for each record in a table. It cannot contain NULL values, and each value must be unique across the table.

Purpose: Ensures that each record can be uniquely identified.

Example:

Emp_id	Emp_Name	Email	Joining Date
1	Jhon	jhonuk@gmail.com	11/11/2019
2	Michle	michleuk@gmail.com	16/08/2019
4	Jhon	jhon22uk@gmail.com	16/11/2019
5	Sara	sarauk@gmail.com	11/11/2019

In the above table, Emp_id can be used as a primary key through which data/records can be uniquely identified.

SQL Queries for Primary Key :

1

```
CREATE TABLE Employee (
    Emp_id INT PRIMARY KEY,
    Emp_Name VARCHAR(50),
    Email VARCHAR(100),
    Joining_Date DATE
);
```

2

```
INSERT INTO Employee (Emp_id, Emp_Name, Email, Joining_Date)
VALUES
(1, 'Jhon', 'jhonuk@gmail.com', '2019-11-11'),
(2, 'Michle', 'michle@gmail.com', '2019-10-05'),
(3, 'Jhon', 'Jhon22@gmail.com', '2018-05-05'),
(4, 'Sara', 'sara@gmail.com', '2019-11-11');
```

3

```
SELECT * FROM Employee;
```

4

```
-- This will fail because Emp_id 1 already exists
INSERT INTO Employee (Emp_id, Emp_Name, Email, Joining_Date)
VALUES
(1, 'New Jhon', 'newjhon@gmail.com', '2020-01-01');
```

Adding a Primary Key to an Existing Table

```
ALTER TABLE Table_name
ADD PRIMARY KEY (column_name);
```

5. Foreign Key

Definition: A foreign key is a column or a group of columns in one table that creates a link between two tables. It refers to the primary key in another table.

Purpose: Enforces referential integrity by ensuring that a value in the foreign key column corresponds to a valid value in the related table's primary key.

The table containing the foreign key is called the child table, andthetablecontaining the candidate key is called the referenced or parent table.

Example:

S_id	S_Name	S_CGPA
13231003	x	3.70
13231004	y	3.50
13231005	z	3.99

TableName: Student_Info

S_id	Session	Dept
13231003	Summer-16	CSE
13231004	Summer-16	CSE
13231005	Summer-16	CSE

TableName: Dept_Info

```
Create table Student_info(
S_id int not null,
S_Name varchar(20),
S_CGPA float,
primary key(S_Id);
```

```
Create table Dept_info(
S_id int,
Session varchar(20),
Dept varchar(20),
Foreign key(S_id) references Student_info(S_id);
```

Set as primary key

S_id	S_Name	S_CGPA
13231003	x	3.70
13231004	y	3.50
13231005	z	3.99

Set as foreign key

S_id	Session	Dept
13231003	Summer-16	CSE
13231004	Summer-16	CSE
13231005	Summer-16	CSE

Chapter 03: Joins

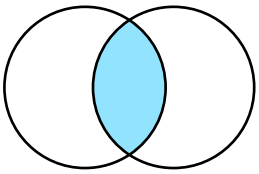
Join is used to combine rows from two or more tables, based on a related column between them.

Why we use join operation?

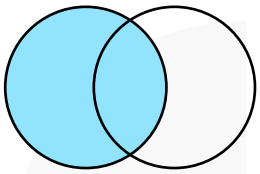
Join operations in SQL are used to combine rows from two or more tables based on a related column between them. They are necessary in relational databases, where data is normalized into multiple tables to maintain data integrity and reduce redundancy. Joins reconstruct the relationships between these tables, allowing efficient retrieval and combination of related data. Different types of joins (e.g., inner, outer) define how unmatched rows are treated, providing flexibility for filtering, reporting, and analysis.

In essence, join operations enable the practical use of relational databases by linking related data spread across different tables.

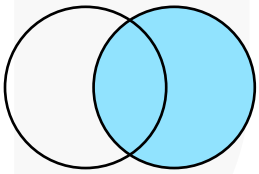
Types of Joins:



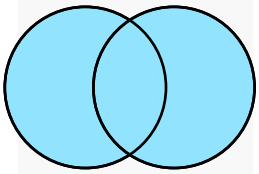
inner join



left join



right join



full join

Exceptional join:

1. self join
2. cross join

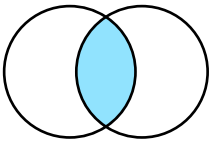
1. inner join

loan

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	loan_number
x	L-170
y	L-230
z	L-155



inner join

INTERSECT operation

```
SELECT *
FROM loan
INNER JOIN borrower
ON loan.loan_number = borrower.loan_number;
```

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	x	L-170
L-230	Redwood	4000	y	L-230

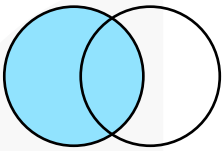
2. left join

loan

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	loan_number
x	L-170
y	L-230
z	L-155



left join

```
SELECT *
FROM loan
LEFT OUTER JOIN borrower
ON loan.loan_number = borrower.loan_number;
```

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	x	L-170
L-230	Redwood	4000	y	L-230
L-260	Perryridge	1700	NULL	NULL

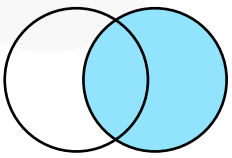
3. right join

loan

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	loan_number
x	L-170
y	L-230
z	L-155



right join

```
SELECT *
FROM loan
RIGHT OUTER JOIN borrower
ON loan.loan_number = borrower.loan_number;
```

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	x	L-170
L-230	Redwood	4000	y	L-230
NULL	NULL	NULL	z	L-155

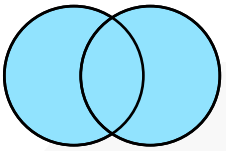
4. full join

loan

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	loan_number
x	L-170
y	L-230
z	L-155



full join

```
SELECT *
FROM loan
FULL OUTER JOIN borrower
ON loan.loan_number = borrower.loan_number;
```

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	x	L-170
L-230	Redwood	4000	y	L-230
L-260	Perryridge	1700	NULL	NULL
NULL	NULL	NULL	z	L-155

cross join

Student

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment

ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

```
SELECT *
FROM Student
CROSS JOIN Enrolment;
```

ID	Name	ID	Code
123	John	123	DBS
124	Mary	123	DBS
125	Mark	123	DBS
126	Jane	123	DBS
123	John	124	PRG
124	Mary	124	PRG
125	Mark	124	PRG
126	Jane	124	PRG
123	John	124	DBS
124	Mary	124	DBS
125	Mark	124	DBS
126	Jane	124	DBS
123	John	126	PRG
124	Mary	126	PRG
125	Mark	126	PRG
126	Jane	126	PRG

Cross join doesn't have any ON clause . As student table has 4 rows and Enrolment table has 4 rows so total 4*4=16 rows will be returned.

self join

Employees

employee_id	employee_name	manager_id
1	Alice	2
2	Bob	NULL
3	Charlie	2

Self Inner Join:

```
SELECT a.employee_id, a.employee_name, b.employee_name AS manager_name
FROM employees a
INNER JOIN employees b
ON a.manager_id = b.employee_id;
```

Employees

employee_id	employee_name	manager_name
1	Alice	Bob
3	Charlie	Bob

Self Left Join:

```
SELECT a.employee_id, a.employee_name, b.employee_name AS manager_name
FROM employees a
LEFT JOIN employees b
ON a.manager_id = b.employee_id;
```

Employees

employee_id	employee_name	manager_name
1	Alice	Bob
2	Bob	NULL
3	Charlie	Bob

Self Right Join:

```
SELECT a.employee_id, a.employee_name, b.employee_name AS manager_name
FROM employees a
RIGHT JOIN employees b
ON a.manager_id = b.employee_id;
```

Employees

employee_id	employee_name	manager_name
1	Alice	Bob
3	Charlie	Bob
NULL	NULL	Bob

Self Cross Join:

```
SELECT a.employee_id AS emp1_id, a.employee_name AS emp1_name,
    b.employee_id AS emp2_id, b.employee_name AS emp2_name
FROM employees a
CROSS JOIN employees b;
```

Employees

emp1_id	emp1_name	emp2_id	emp2_name
1	Alice	1	Alice
1	Alice	2	Bob
1	Alice	3	Charlie
2	Bob	1	Alice
2	Bob	2	Bob
2	Bob	3	Charlie
3	Charlie	1	Alice
3	Charlie	2	Bob
3	Charlie	3	Charlie

Chapter 04: Replacing values

ISNULL()

If you want to replace NULL with another value when querying, you can use the ISNULL() function.

```
SELECT Emp_Name, ISNULL(Manager_Name, 'No Manager')
AS Manager_Name
FROM Employees;
```

Employees

Emp_Name	Manager_Name
Alice	Charlie
Bob	NULL
Charlie	Alice

Employees

Emp_Name	Manager_Name
Alice	Charlie
Bob	No Manager
Charlie	Alice

Handling Multiple NULL Values:

Consistency: Each function processes NULL values row by row. If a column contains several NULL values, each one is replaced according to the function's logic.

Outcome: By applying these functions, you ensure that all NULL values in the column are replaced with a meaningful default value, making your data cleaner and easier to work with.

COALESCE

```
SELECT Emp_Name, COALESCE(Manager_Name, 'No Manager')
AS Manager_Name
FROM Employees;
```

Employees

Emp_Name	Manager_Name
Alice	Charlie
Bob	No Manager
Charlie	Alice

```
SELECT
   COALESCE(First_Name, 'no') AS FirstName,
   COALESCE(Middle_Name, 'no') AS MiddleName,
   COALESCE(Last_Name, 'no') AS LastName
FROM Employees;
```

Employees

First_Name	Middle_Name	Last_Name
A.P.J.	NULL	NULL
NULL	ABDUL	NULL
NULL	NULL	KALAM
A.P.J.	ABDUL	NULL

Employees

First_Name	Middle_Name	Last_Name
A.P.J.	no	no
no	ABDUL	no
no	no	KALAM
A.P.J.	ABDUL	no

COALESCE function

Employees

First_Name	Middle_Name	Last_Name
A.P.J.	NULL	NULL
NULL	ABDUL	NULL
NULL	NULL	KALAM
A.P.J.	ABDUL	NULL

```
SELECT COALESCE(First_Name, Middle_Name, Last_Name)
AS PreferredName
FROM Employee;
```

Employees

PreferredName
A.P.J.
ABDUL
KALAM
A.P.J.

First Name has 1st priority
Middle Name has 2nd priority
Last name has 3rd priority

First NOT NULL will show what it gets. Other NOT NULL values will not be displayed. If NULL value is found, check will continue until NOT NULL is found.

Chapter 05: Union & Union All

table1

column1
1
2
3

table2

column1
3
4
5

UNION

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

column1
1
2
3
4
5

UNION ALL

```
SELECT column1 FROM table1
UNION ALL
SELECT column1 FROM table2;
```

column1
1
2
3
3
4
5

SectionA

id	Name
1	Tania
2	Shohag
3	Arafat

SectionB

id	Name
3	Akib
4	Shohag
5	Siam

```
Select id, Name from SectionA
Union
Select id from SectionB
```

Has a column mismatch error, because the first part of the query selects two columns (id, Name) from SectionA, while the second part only selects one column (id) from SectionB.

All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists. This is true for both UNION & UNION all

Here's the corrected query

```
SELECT id, Name FROM SectionA
UNION
SELECT id, NULL AS Name FROM SectionB;
```

id	Name
1	Tania
2	Shohag
3	Arafat
3	NULL
4	NULL
5	NULL

SectionA

id	Name
1	Tania
2	Shohag
3	Arafat

SectionB

id	Name
1	Tania
2	Shohag
3	Arafat

```
Select id, Name from SectionA
Union
Select id, Name from SectionB
```

id	Name
1	Tania
2	Shohag
3	Arafat

```
Select id, Name from SectionA
Union ALL
Select id, Name from SectionB
```

id	Name
1	Tania
2	Shohag
3	Arafat
1	Tania
2	Shohag
3	Arafat

Comparison table for UNION vs UNION AL

Feature	UNION	UNION ALL
Duplicate Rows	Removes duplicate rows	Returns all rows, including duplicates
Sorting	Sorts the result set to remove duplicates	Does not sort the result set
Performance	Generally slower due to distinct sort operation	Generally faster as it does not sort or remove duplicates
Use Case	Use when you need distinct results	Use when you want to include all rows, and performance is a concern