

# SQL Fundamentals

M1W4D1

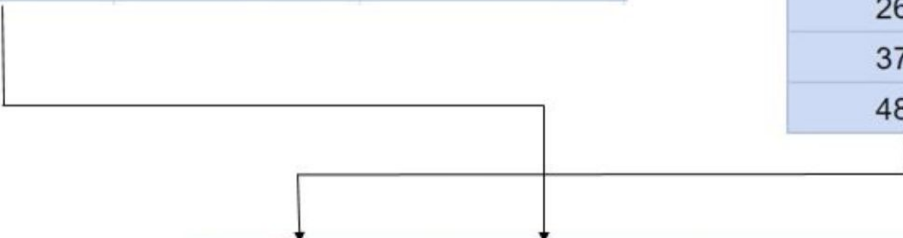
# Relational Databases

**A relational database is one that stores data in tables. The relationship between each data point is clear and searching through those relationships is relatively easy. The relationship between tables and field types is called a schema. For relational databases, the schema must be clearly defined.**

# Relational Databases

Name	Dry/Wet Food	Good Boy (Y/N)
Fido	Dry	Y
Rex	Wet	N
Bubbles	Dry	Y
Cujo	Wet	N

Tag #	Height (in)	Weight (lbs)
1573	15	21
2684	9	7
3795	27	130
4806	6	5



Tag #	Name	Breed	Color	Age
1573	Fido	Beagle	Brown/White	1.5
2684	Rex	Pekingese	White	9
3795	Bubbles	Rottweiler	Black	5
4806	Cujo	Chihuahua	Gold	4

# Non-Relational Databases

**A non-relational database is any database that does not use the tabular schema of rows and columns like in relational databases. Rather, its storage model is optimized for the type of data it's storing.**

# Non-Relational Databases

Key	Document
1001	{ "TagID": 1573 , "Dimensions": [ { "Height": 15, "Weight": 21 }], "Name:" Fido }
1002	{ "TagID": 2684 , "Dimensions": [ { "Height": 9, "Weight": 7 }], "Name:" Rex }

There are four different types of NoSQL databases.

**Document-oriented databases** – Also known as a document store, this database is designed for storing, retrieving and managing document-oriented information. Document databases usually pair each key with a complex data structure (called a document).

**Key-Value Stores** – This is a database that uses different keys where each key is associated with only one value in a collection. Think of it as a dictionary. This is one of the simplest database types among NoSQL databases.

**Wide-Column Stores** – This database uses tables, rows, and columns, but unlike a relational database, the names and format of the columns can vary from row to row in the same table.

**Graph Stores** – A graph database uses graph structures for semantic queries with nodes, edges, and properties to represent and store data.

# How to choose a

- 1) What type of data will you be analyzing?
- 1) How much data are you dealing with?
- 1) What kind of resources can you devote to the setup and maintenance of your database?
- 1) Do you need real-time data?

# Introduction to SQL

**SQL (Structured Query Language)** is a computer language aimed to store, manipulate and retrieve data stored in relational databases.

**SQL language has several parts:**

- 1) DDL - Data Definition Language**
- 2) DML - Data Manipulation Language**
- 3) View Definition**
- 4) Transaction Control**

# Data Definition Language

**DDL statements are used to define the database structure or schema. Examples :**

- 1) CREATE**
- 2) ALTER**
- 3) DROP**
- 4) RENAME**



# Data Manipulation Language

**DML statements are used for managing data within schema objects. Examples :**

- 1) SELECT**
- 2) INSERT**
- 3) UPDATE**
- 4) DELETE**

# Basic Structure of SQL Query

<b>General Structure</b>	SELECT, ALL/ DISTINCT, *, AS, FROM, WHERE
<b>Comparison</b>	IN, BETWEEN, LIKE, ILIKE
<b>Grouping</b>	GROUP BY, HAVING, COUNT(), SUM(), AVG(), MAX(), MIN()
<b>Display Order</b>	ORDER BY, ASC/ DESC
<b>Logical Operators</b>	AND, OR, NOT
<b>Output</b>	INTO TABLE/ CURSOR, TO SCREEN
<b>Union</b>	UNION

# SELECT & DISTINCT Statement

**SELECT** is the most common statement used, and it allows us to retrieve information from a table.

In order to select the entire table **SELECT \*** is used

Sometimes a table contains a column that has duplicate values and in order to return only the unique values, **DISTINCT** statement is used in combination with **SELECT**

# COUNT Statement

**COUNT** returns the number of input rows that match a specific condition of a query

**COUNT** can be applied on a specific column or just passed as **COUNT(\*)**, both giving same results

# WHERE Statement

**WHERE** statement allows us to specify the conditions on columns for the rows to be returned

OPERATOR	DESCRIPTION
=	EQUAL
>	GREATER THAN
<	LESS THAN
>=	GREATER THAN OR EQUAL TO
<=	LESS THAN OR EQUAL TO
<> OR !=	NOT EQUAL TO
AND	BOTH CONDITIONS TRUE
OR	EITHER OF THE CONDITIONS TRUE
NOT	OFFSET OF SPECIFIED CONDITION

# ORDER BY Statement

**SELECT** company, name, sales **FROM** table  
**ORDER BY** company, sales

COMPANY	NAME	SALES
Apple	Andrew	100
Apple	Zach	300
Google	Claire	200
Google	David	500
Xerox	Steven	100

# LIMIT Statement

The **LIMIT** command allows us to limit the number of rows returned for a query.

Useful for not wanting to return every single row in a table, but only view the top few rows to get an idea of the table layout

**LIMIT** also becomes useful in combination with **ORDER BY**

**LIMIT** goes at the very end of a query request and is the last command to be executed

# BETWEEN Statement

The **BETWEEN** operator can be used to match a value against a range of values

value **BETWEEN** low **AND** high

Can be combined with **NOT** operator

Can be also used with dates in the format : YYYY-MM-DD



# IN Statement

The **IN** operator can be used to create a condition that checks to see if a value is included in a list of multiple options for example if a user's name shows up in a list of known names

value **IN** ( option 1, option 2, option 3,....., option N )

# LIKE & ILIKE Statement

In order to match a string against a general pattern we use **LIKE** and **ILIKE** for example:

All emails ending with '@gmail.com'

**LIKE & ILIKE** allows us to perform pattern matching against string data with the use of **wildcard** characters

- **Percent %** - Matches any sequence of characters
- **Underscore \_** - Matches any single character

**LIKE** is case-sensitive whereas **ILIKE** is case-insensitive

# LIKE & ILIKE Statement

All names that begin with 'A'

**WHERE** name **LIKE** 'A%'

All names that end with 'a'

**WHERE** name **LIKE** '%a'

Get all Mission Impossible Films

**WHERE** name **LIKE** 'Mission Impossible \_ \_'

Combination of Wildcards

**WHERE** name **LIKE** '\_ her%'

- Cheryl
- Theresa
- Sherri

# Aggregate Functions

The main idea behind aggregate function is to take multiple inputs and return a single output

- **AVG()** - Returns floating point values. **ROUND()** can be used to specify precision after the decimal
- **COUNT()**
- **MAX()**
- **MIN()**
- **SUM()**

# GROUP BY Statement

**GROUP BY** allows us to aggregate data and apply functions to better understand how data is distributed per category

Category	Data
A	10
A	5
B	2
B	4
C	12
C	6



A	10
A	5

B	2
B	4

C	12
C	6

**AVG()**



Category	Result
A	7.5
B	3
C	9

# HAVING BY Statement

**HAVING** allows us to filter after an aggregation has already taken place.

We can use it along with a **GROUP BY**

# AS Statement

**AS** allows us to create an Alias for a column or a result

**AS** operator gets executed at the very end of the query meaning we can not use the Alias inside a **WHERE** or **HAVING** operator

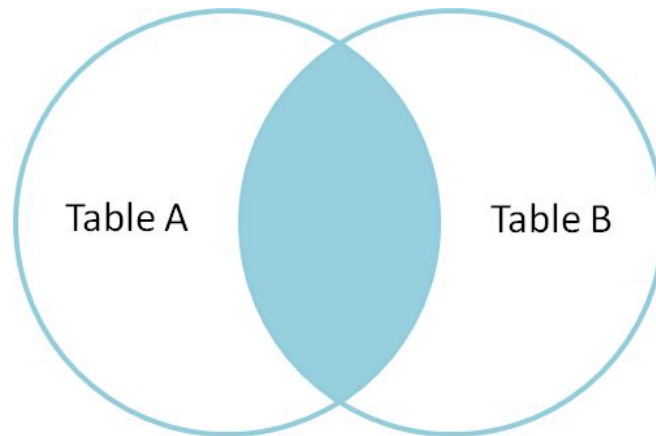
# INNER JOIN Statement

Reg_ID	Name
1	Andrew
2	Bob
3	Charlie
4	David

Log_ID	Name
1	Xavier
2	Andrew
3	Yauren
4	Bob



Reg_ID	Name	Log_ID	Name
1	Andrew	2	Andrew
2	Bob	4	Bob





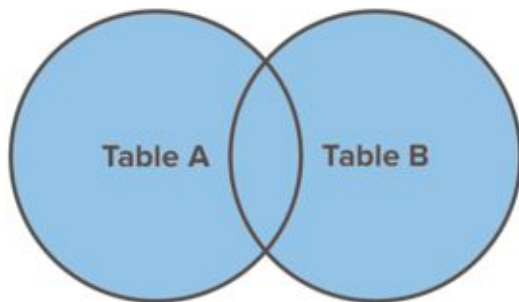
# FULL OUTER JOIN Statement

Reg_ID	Name
1	Andrew
2	Bob
3	Charlie
4	David

Log_ID	Name
1	Xavier
2	Andrew
3	Yauren
4	Bob



Red_ID	Name	Log_ID	Name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yauren



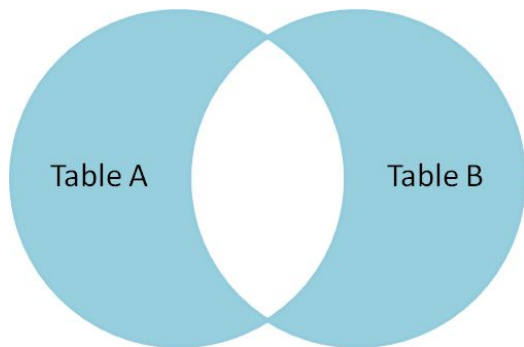
# FULL OUTER JOIN with WHERE

Reg_ID	Name
1	Andrew
2	Bob
3	Charlie
4	David

Log_ID	Name
1	Xavier
2	Andrew
3	Yauren
4	Bob



Red_ID	Name	Log_ID	Name
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yauren



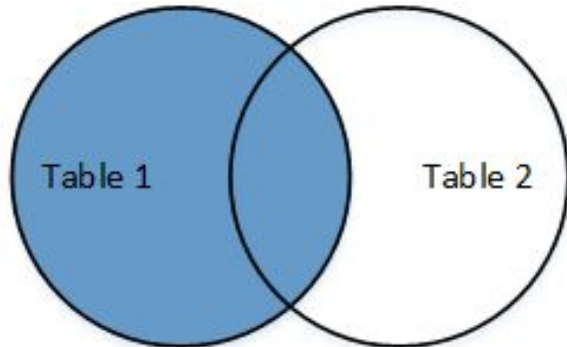
# LEFT OUTER JOIN Statement

Reg_ID	Name
1	Andrew
2	Bob
3	Charlie
4	David

Log_ID	Name
1	Xavier
2	Andrew
3	Yauren
4	Bob



Red_ID	Name	Log_ID	Name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null



# LEFT OUTER JOIN with WHERE

Reg_ID	Name
1	Andrew
2	Bob
3	Charlie
4	David

Log_ID	Name
1	Xavier
2	Andrew
3	Yauren
4	Bob



Red_ID	Name	Log_ID	Name
3	Charlie	null	null
4	David	null	null

