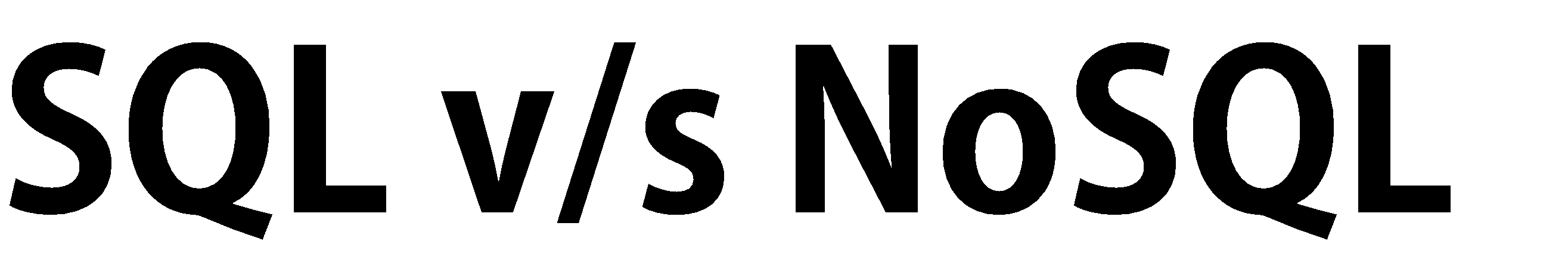
**SQL**

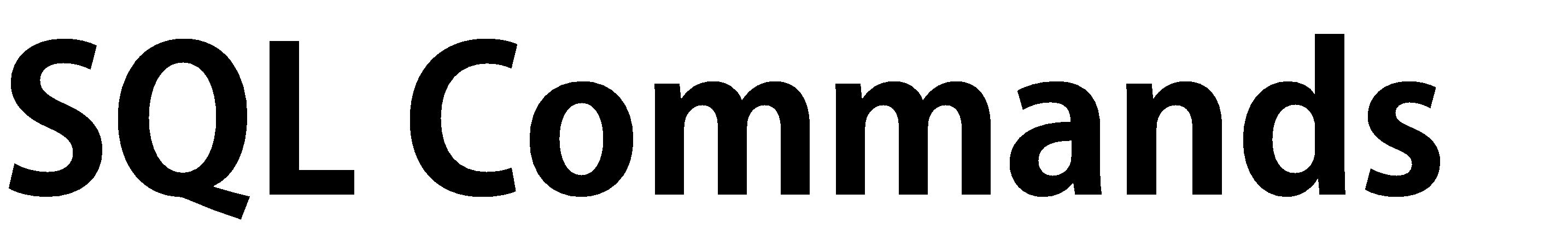
* + - 1. Introduction to SQL-What Is SQL & Database
      2. Data Types, Primary-Foreign Keys & Constraints
         1. Install postgresql and pgadmin4
      3. Create Table In SQL & Create Database
      4. INSERT UPDATE, DELETE & ALTER Table
      5. SELECT Statement & WHERE Clause with Example
      6. How To Import Excel File (CSV) to SQL
      7. Functions in SQL & String Function
      8. Aggregate Functions – Types & Syntax
      9. Group By and Having Clause
      10. Time Stamp and Extract Function, Date Time Function
      11. SQL JOINS – Types & Syntax
      12. SELF JOIN, UNION & UNION ALL
      13. Subquery

1. Window Function – Types & Syntax
2. Case Statement/Expression with examples
3. CTE- Common Table Expression with examples



|  |  |
| --- | --- |
| **Relational Database** | **Non-Relational Database** |
| SQL database | NoSQL database |
| Data stored in tables | Data stored are either key-value pairs, document-based, graph databases or wide- column stores |
| These databases have fixed or static or predefined schema | They have dynamic schema |
| Low performance with huge volumes of data | Easily work with huge volumes of data |
| Eg: PostgreSQL, MySQL, MS SQL Server | Eg: MongoDB, Cassandra, Hbase |

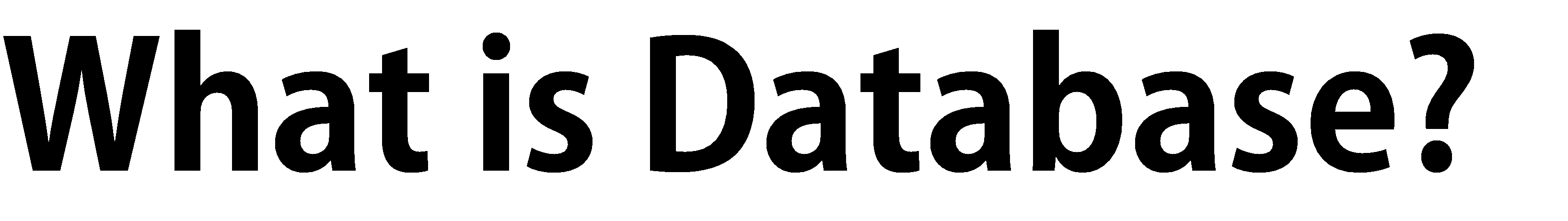
7

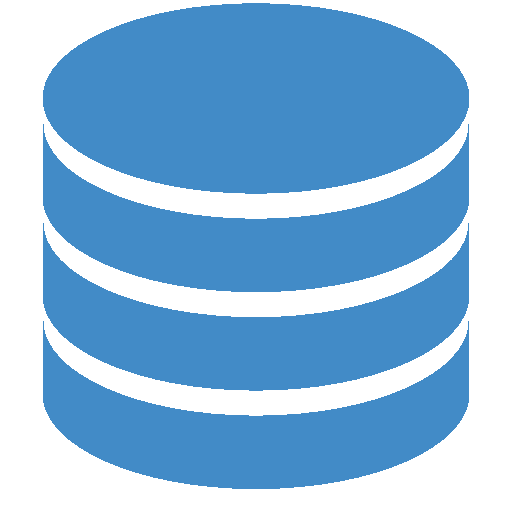


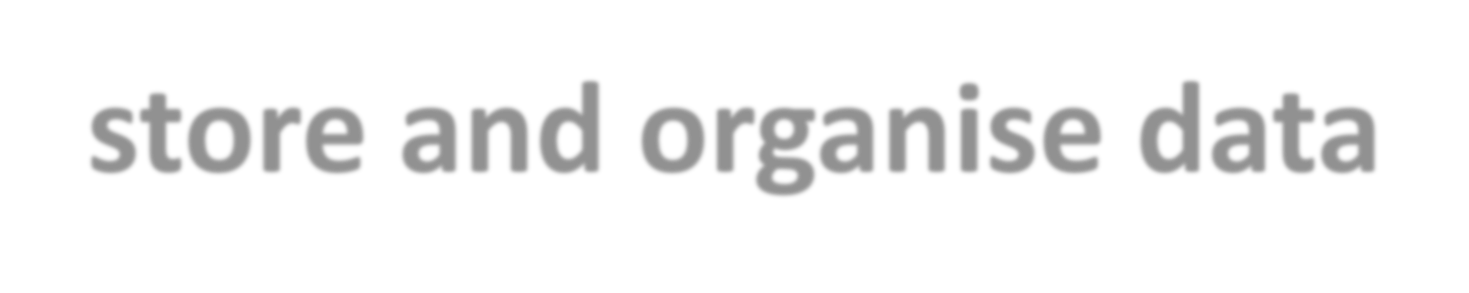
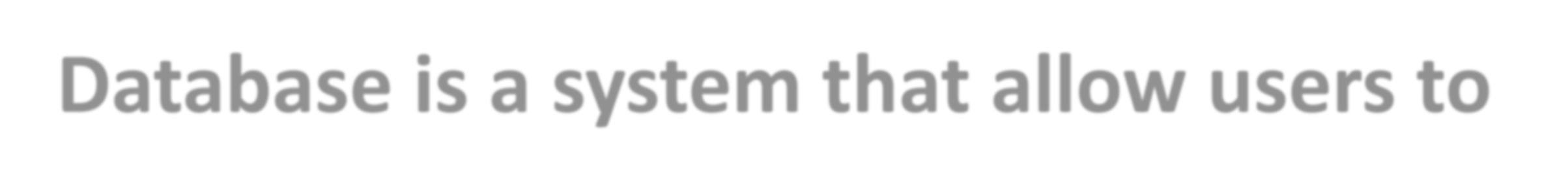
**There are mainly 3 types of SQL commands:**

* **DDL** (Data Definition Language): create, alter, and drop
* **DML** (Data Manipulation Language): select, insert, update and delete
* **DCL** (Data Control Language): grant and revoke permission to users

8

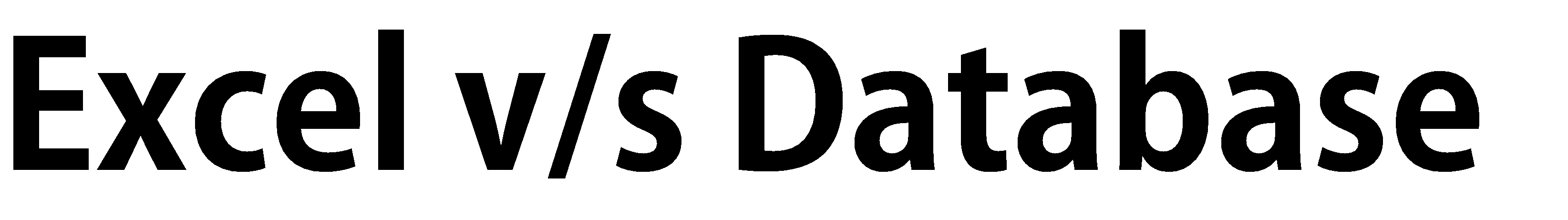






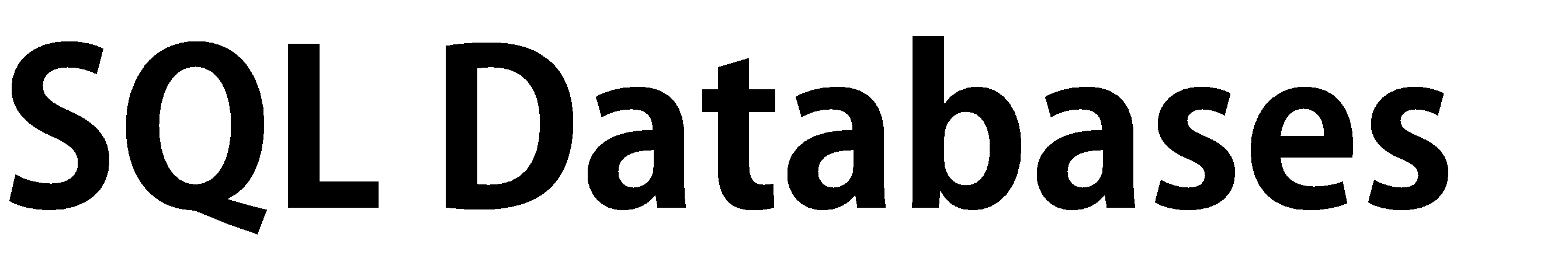
**Database is a system that allow users to store and organise data**

9



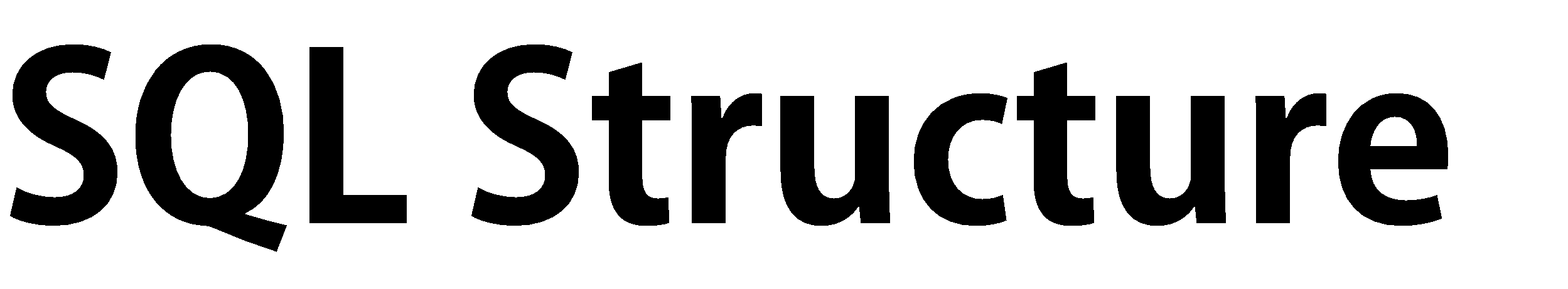
|  |  |
| --- | --- |
| **Excel** | **Database** |
| Easy to use- untrained person can work | Trained person can work |
| Data stored less data | Stores large amount of data |
| Good for one time analysis, quick charts | Can automate tasks |
| No data integrity due to manual operation | High data integrity |
| Low search/filter capabilities | High search/filter capabilities |

10





11

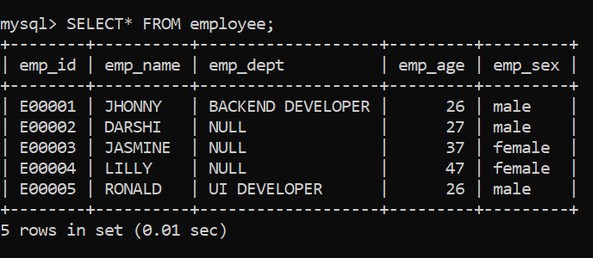


**Database**

**Tables**

**Data**

**(Rows & Columns)**



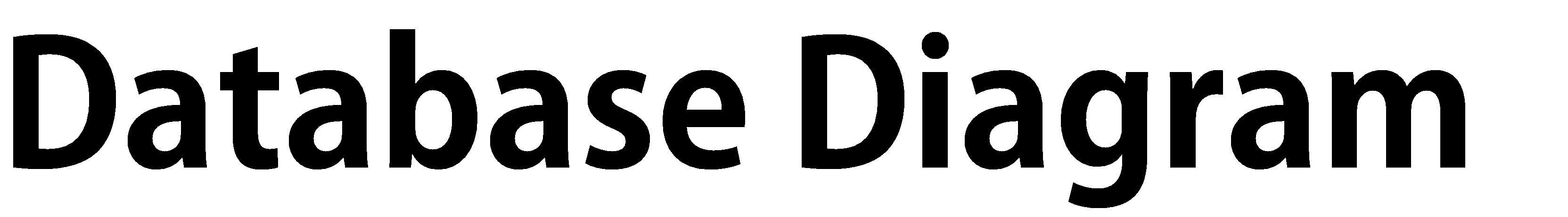
**Columns**

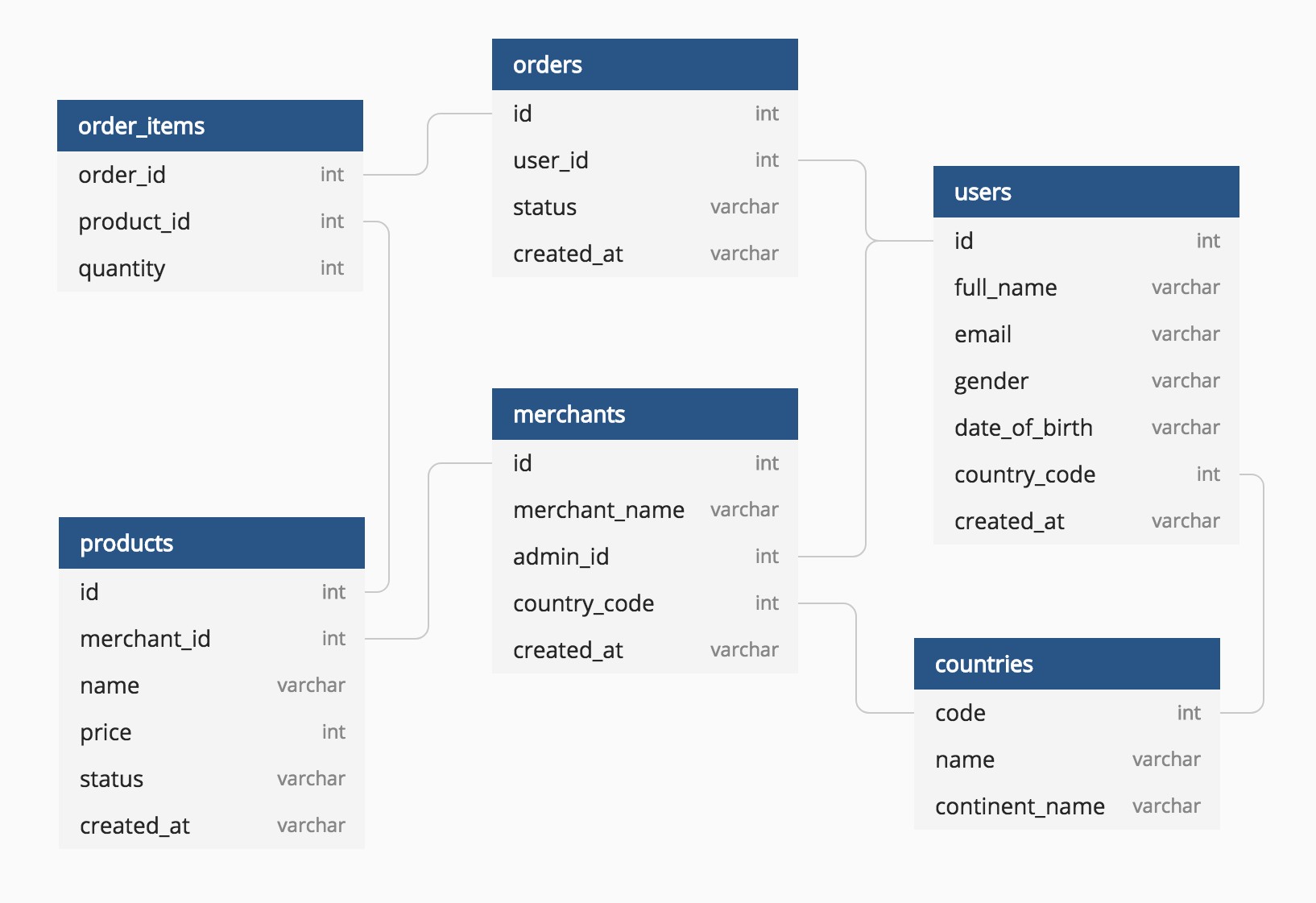
**Example**

**Rows**

**RDBMS**

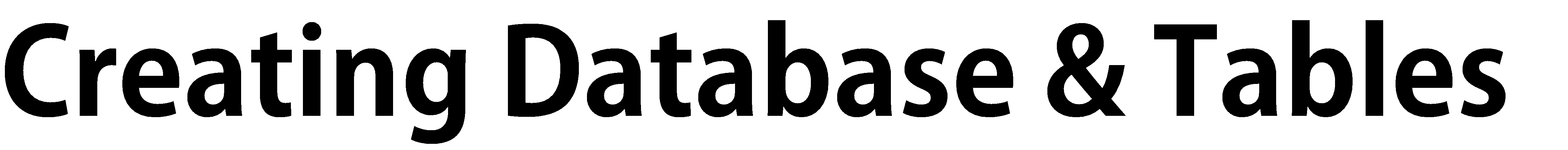
13



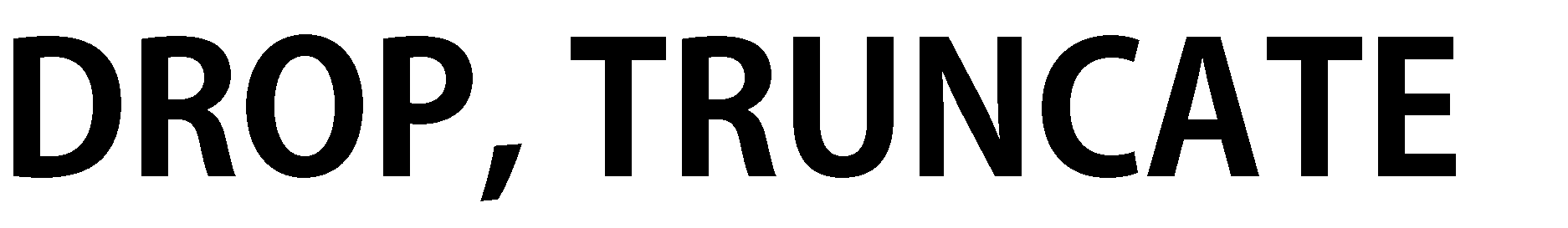
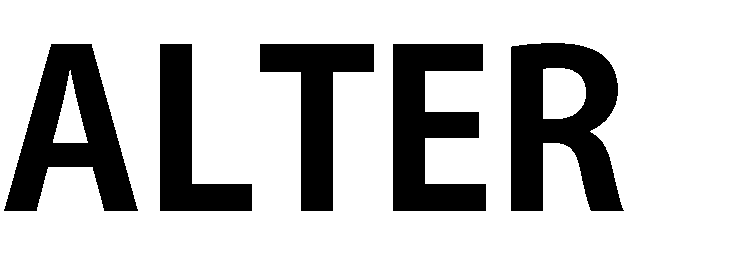
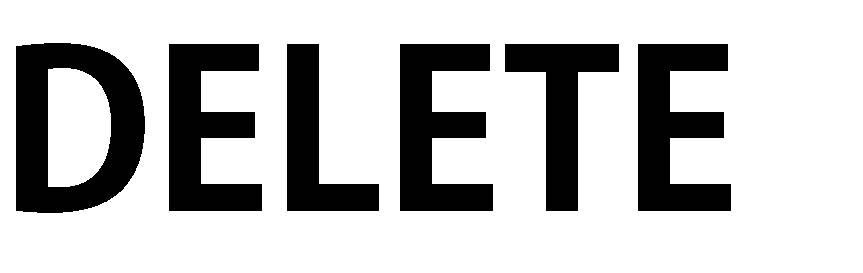
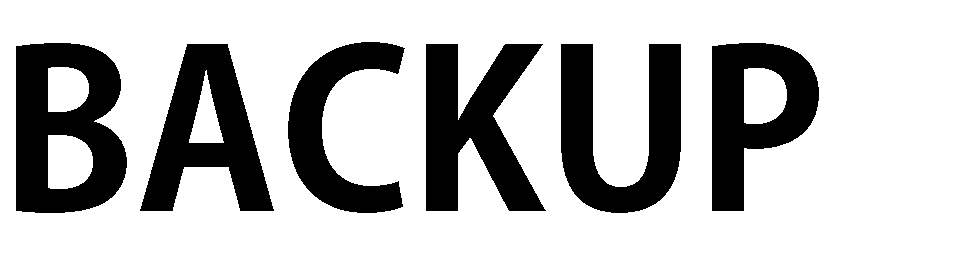
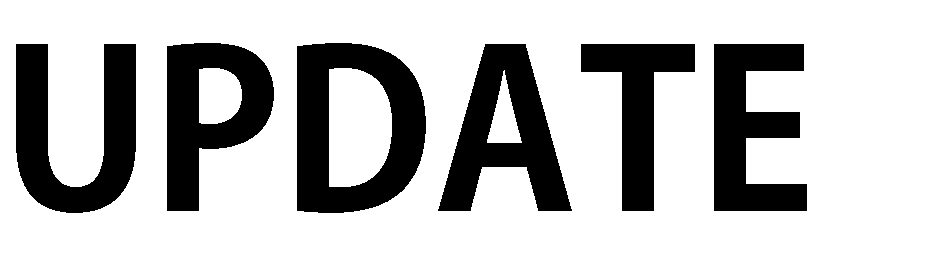
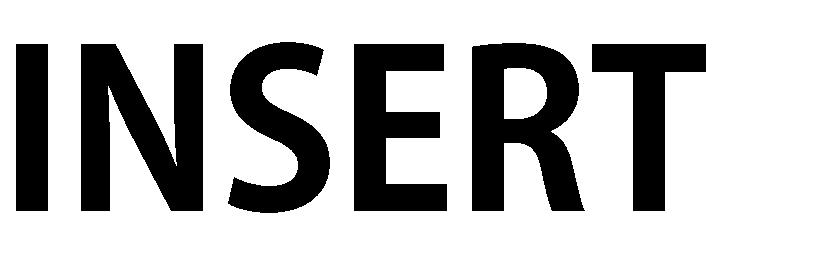
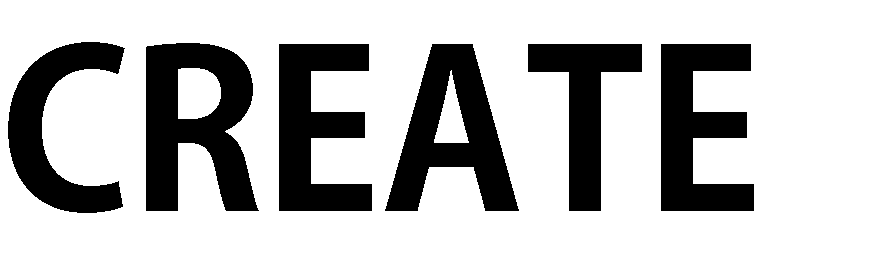
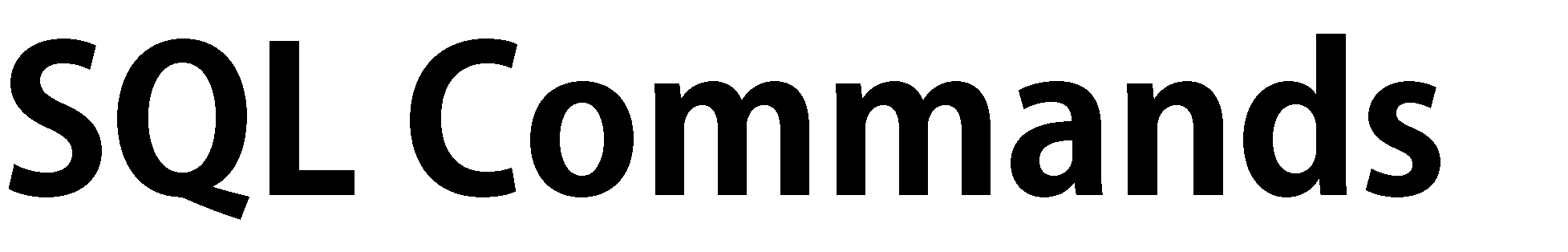
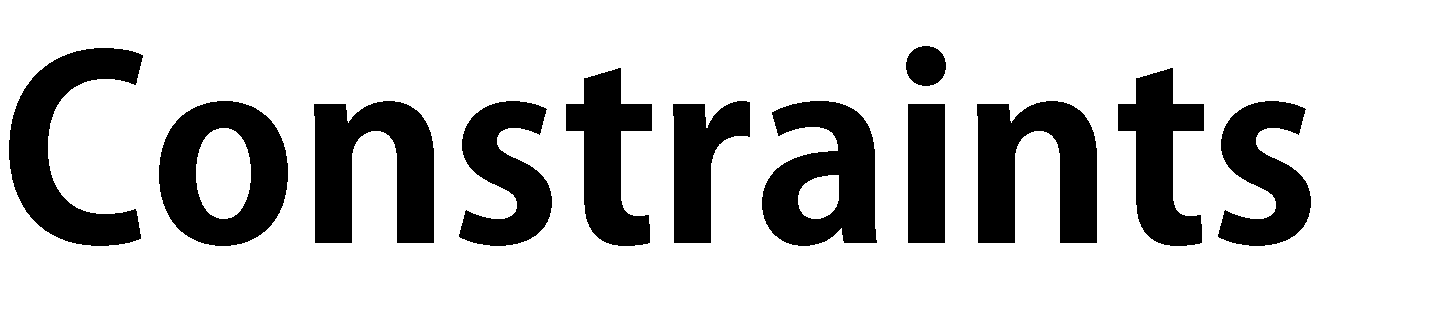
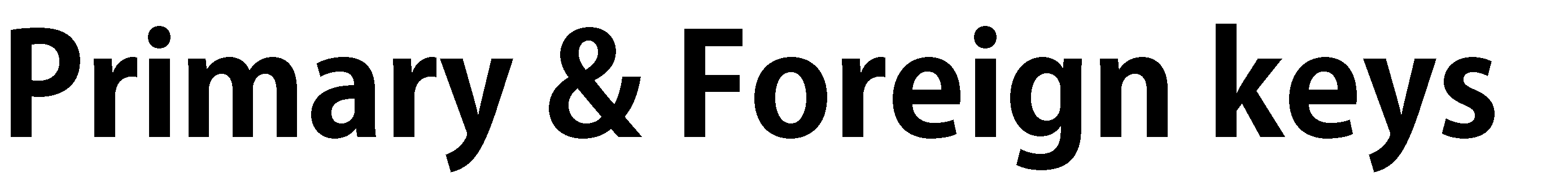
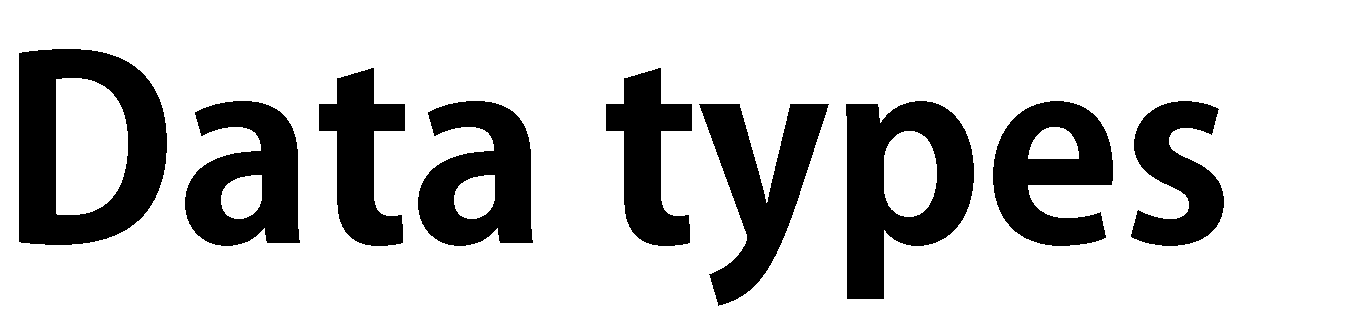


**Example**

14



•



•

•

•

•

•

•

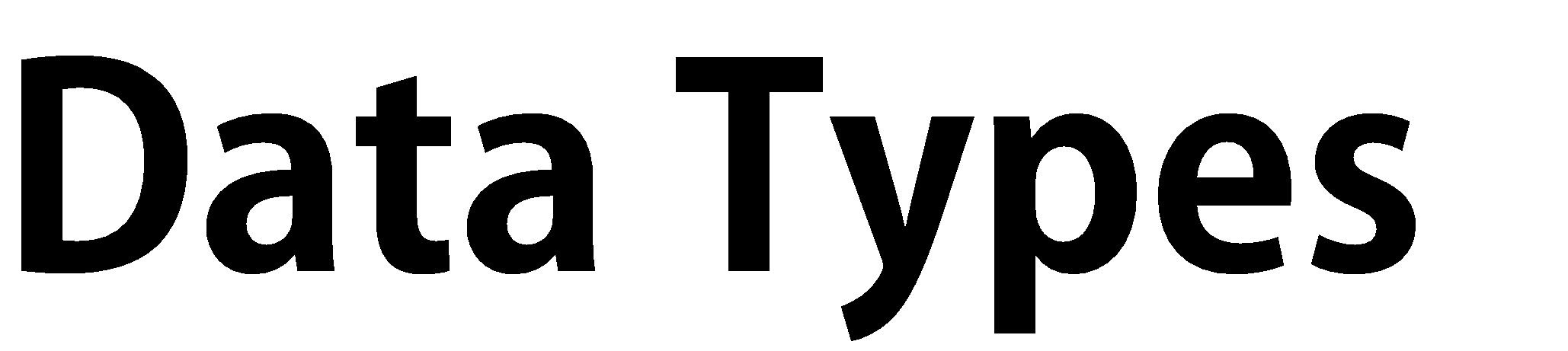
•

•

•

•

15

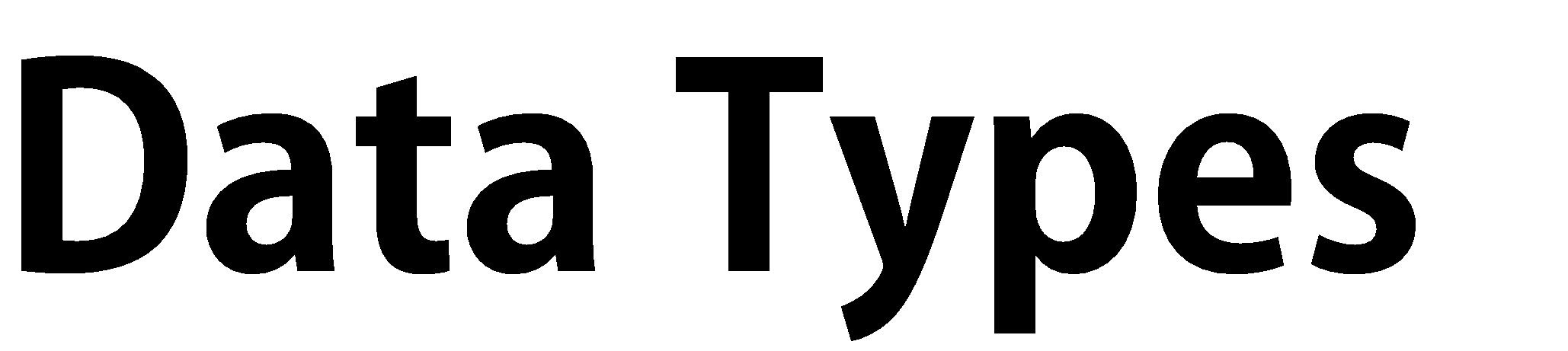


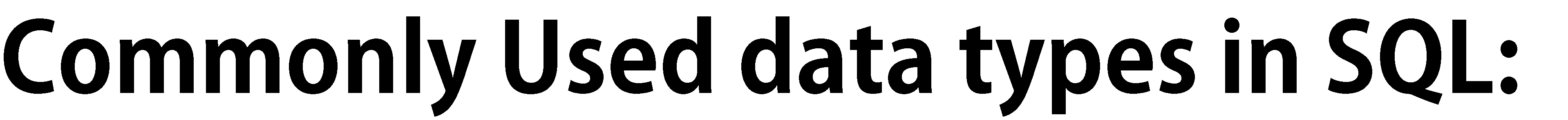
* Data type of a column defines what value the column can store in table
* Defined while creating tables in database

#### Data types mainly classified into three categories + most used

oString: char, varchar, etc oNumeric: int, float, bool, etc oDate and time: date, datetime, etc

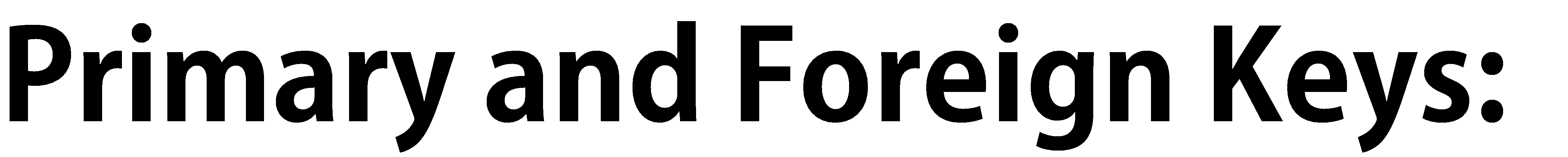
16

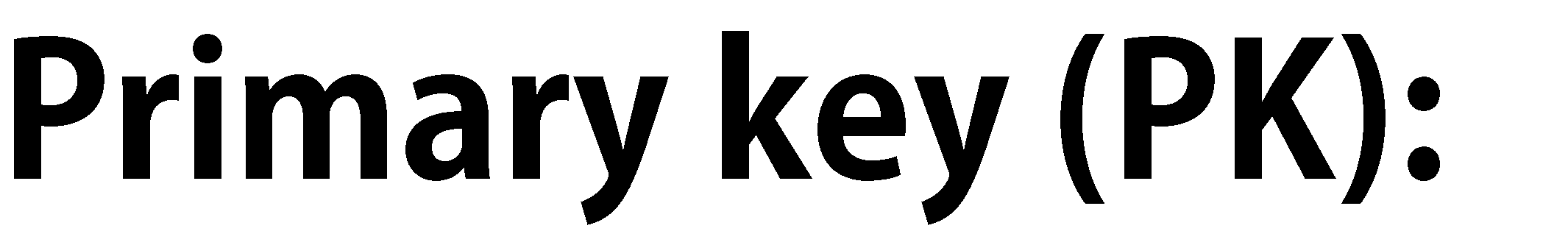




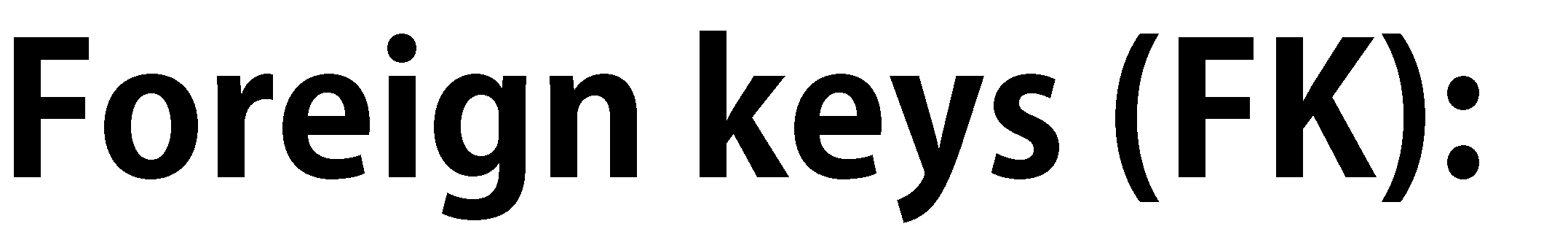
* **int:** used for the integer value
* **float:** used to specify a decimal point number
* **bool:** used to specify Boolean values true and false
* **char:** fixed length string that can contain numbers, letters, and special characters
* **varchar:** variable length string that can contain numbers, letters, and special characters
* **date:** date format YYYY-MM-DD
* **datetime:** date & time combination, format is YYYY-MM-DD hh:mm:ss

17



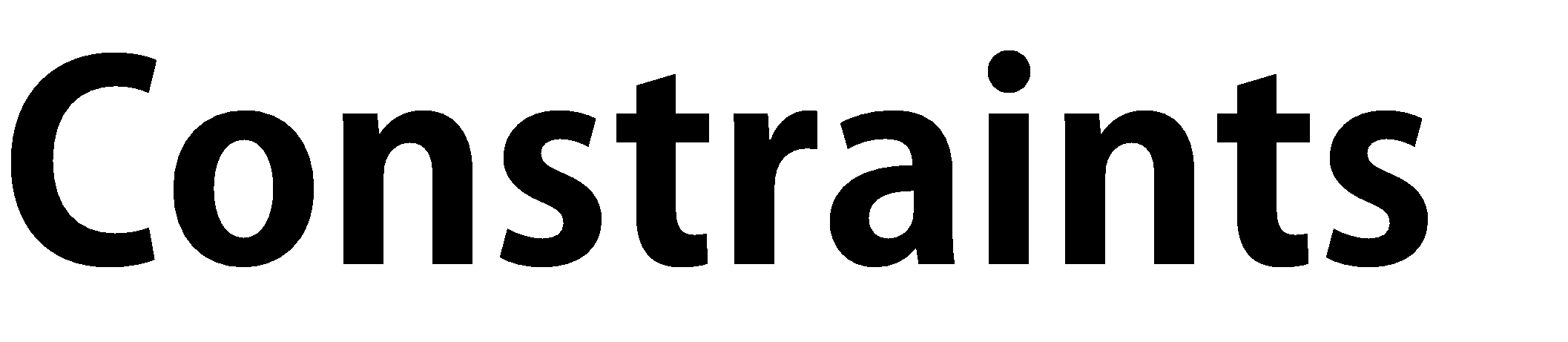


* A Primary key is a unique column we set in a table to easily identify and locate data in queries
* A table can have only one primary key, which should be unique and NOT NULL



* A Foreign key is a column used to link two or more tables together
* A table can have any number of foreign keys, can contain duplicate and NULL values

18



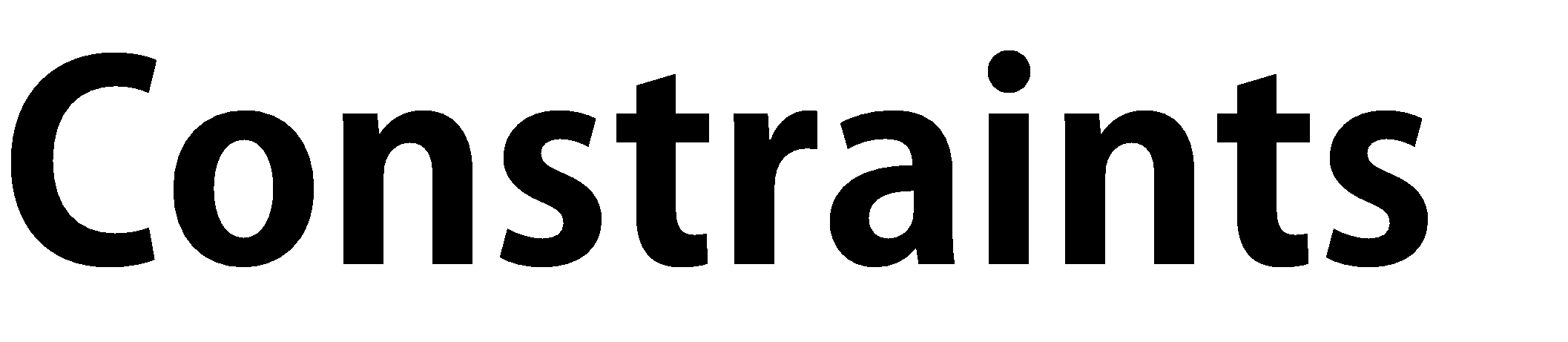
* Constraints are used to specify rules for data in a table
* This ensures the accuracy and reliability of the data in the table
* Constraints can be specified when the table is created with the CREATE TABLE statement, or
* after the table is created with the ALTER TABLE statement
* Syntax

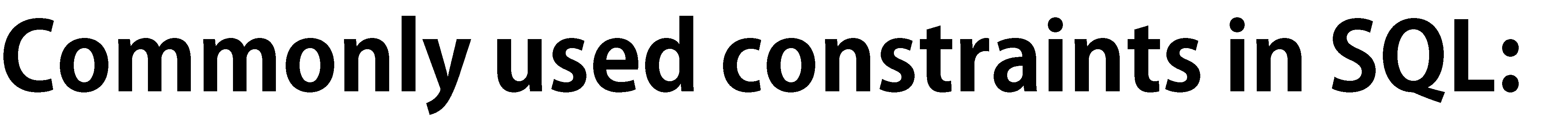
CREATE TABLE table\_name ( column1 datatype constraint, column2 datatype constraint, column3 datatype constraint,

....

);

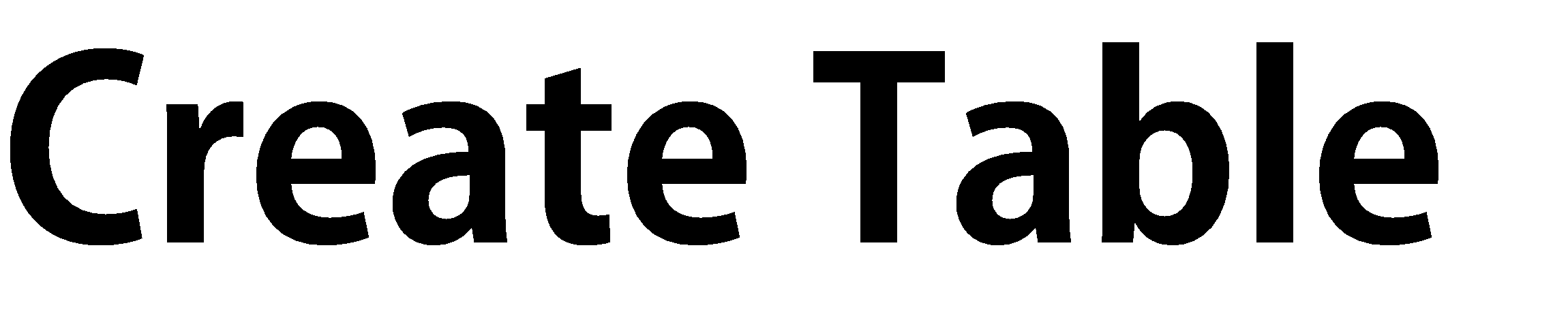
19





* NOT NULL - Ensures that a column cannot have a NULL value
* UNIQUE - Ensures that all values in a column are different
* PRIMARY KEY - A combination of a NOT NULL and UNIQUE
* FOREIGN KEY - Prevents actions that would destroy links between tables (used to link multiple tables together)
* CHECK - Ensures that the values in a column satisfies a specific condition
* DEFAULT - Sets a default value for a column if no value is specified
* CREATE INDEX - Used to create and retrieve data from the database very quickly

20



**The CREATE TABLE statement is used to create a new table in a database**

* **Syntax**

**CREATE TABLE** table\_name

(

column\_name1 **datatype** constraint,

column\_name2 **datatype** constraint, column\_name3 **datatype** constraint,

);

* **Example**

CREATE TABLE customer

(

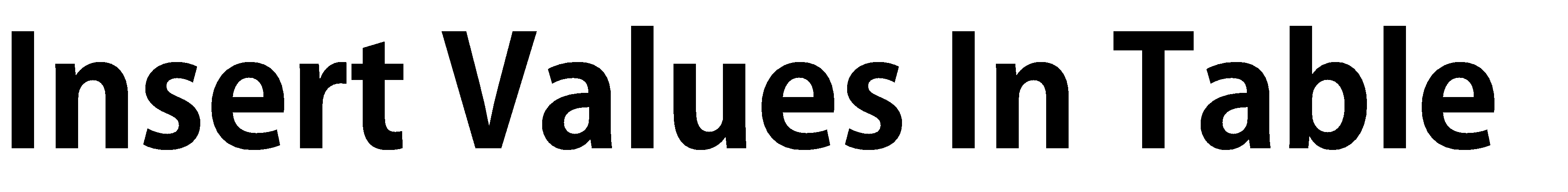
CustID int8 PRIMARY KEY,

CustName varchar(50) NOT NULL, Age int NOT NULL,

City char(50), Salary numeric

);

23



**The INSERT INTO statement is used to insert new records in a table**

* **Syntax**

**INSERT INTO** TABLE\_NAME

(column1, column2, column3,...columnN) VALUES

(value1, value2, value3,...valueN);

* **Example**

INSERT INTO customer

(CustID, CustName, Age, City, Salary)

VALUES

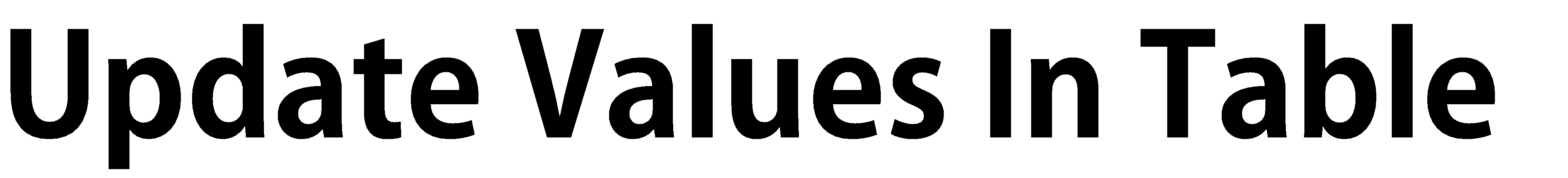
(1, ‘Sam’, 26, ‘Delhi’, 9000),

(2, ‘Ram’, 19, ‘Bangalore’, 11000),

(3, ‘Pam’, 31, ‘Mumbai’, 6000),

(4, ‘Jam’, 42, ‘Pune’, 10000);

25



The UPDATE command is used to update existing rows in a table

* **Syntax**

**UPDATE** TABLE\_NAME

SET “Column\_name1” = ‘value1’, “Column\_name2” = ‘value2’ WHERE “ID” = ‘value’

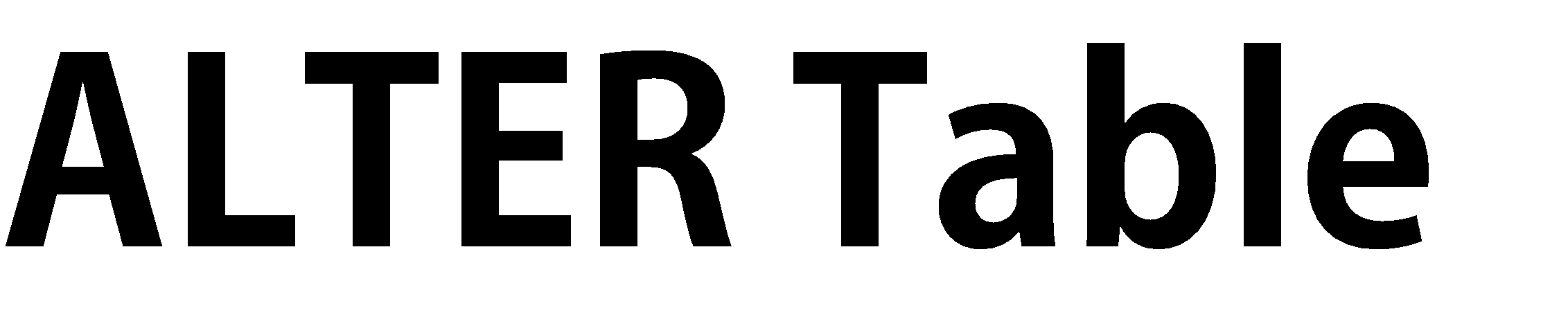
* Example

UPDATE customer

SET CustName = 'Xam’, Age= 32

WHERE CustID = 4;

26



The ALTER TABLE statement is used to add, delete, or modify columns in an existing table

* **ALTER TABLE - ADD Column Syntax**

**ALTER TABLE** table\_name

**ADD COLUMN** column\_name ;

* ALTER TABLE - DROP COLUMN Syntax

**ALTER TABLE** table\_name

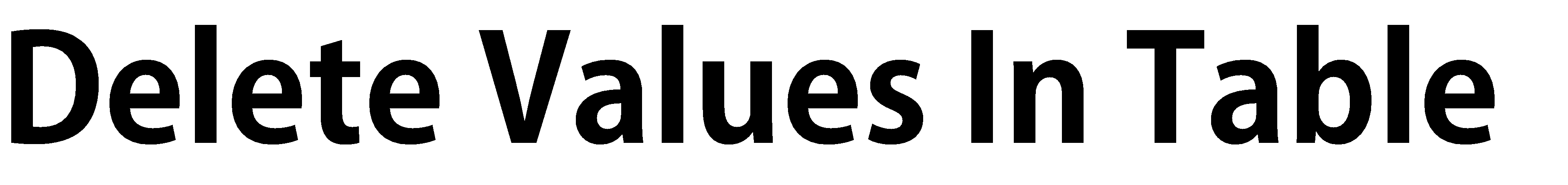
**DROP COLUMN** column\_name;

* ALTER TABLE - ALTER/MODIFY COLUMN Syntax

**ALTER TABLE** table\_name

**ALTER COLUMN** column\_name datatype;

27



The DELETE statement is used to delete existing records in a table

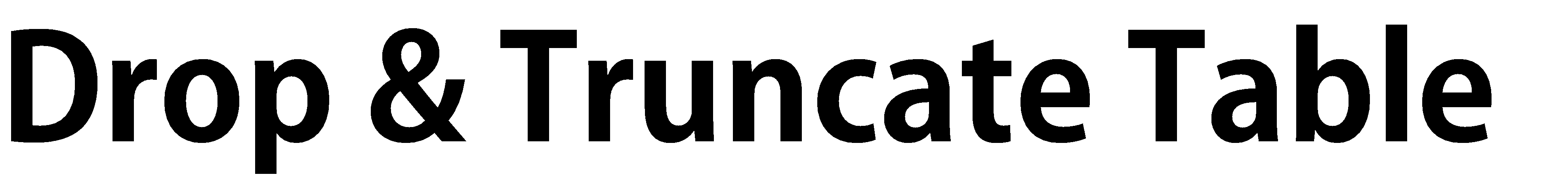
* **Syntax**

**DELETE** FROM table\_name WHERE condition;

* Example

DELETE FROM customer WHERE CustID = 3;

28



The DROP TABLE command deletes a table in the database

* **Syntax**

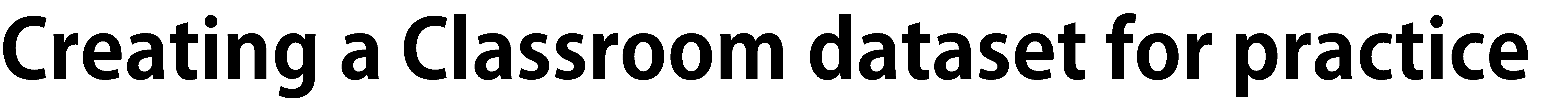
**DROP TABLE** table\_name;

The TRUNCATE TABLE command deletes the data inside a table, but not the table itself

* **Syntax**

**TRUNCATE TABLE** table\_name;

29



CREATE TABLE classroom

(

rollno int8 PRIMARY KEY, name varchar(50) NOT NULL, house char(12) NOT NULL, grade char(1)

);

INSERT INTO classroom (rollno, name, house, grade) VALUES

(1, ‘Sam’, ‘Akash’, ‘B’),

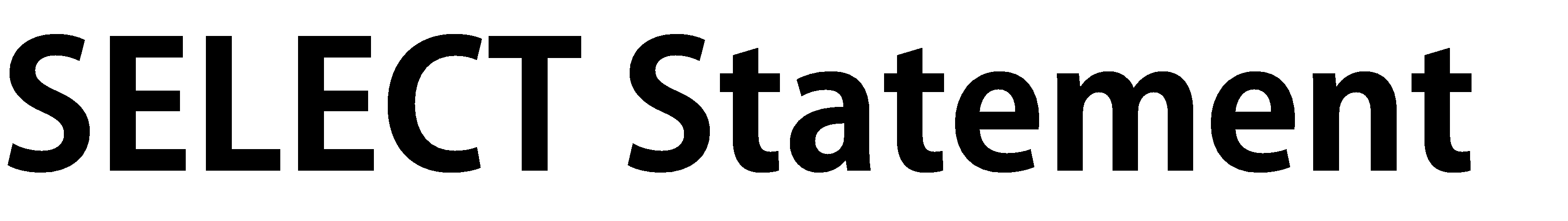
(2, ‘Ram’, ‘Agni’, ‘A’),

(3, ‘Shyam’, ‘Jal’, ’B’),

(4, ‘Sundar’, ‘Agni’, ’A’),

(5, ‘Ram’, ‘Yayu’, ‘B’);

31



The SELECT statement is used to select data from a database.

* **Syntax**

**SELECT** column\_name FROM table\_name;

To select all the fields available in the table

* **Syntax**

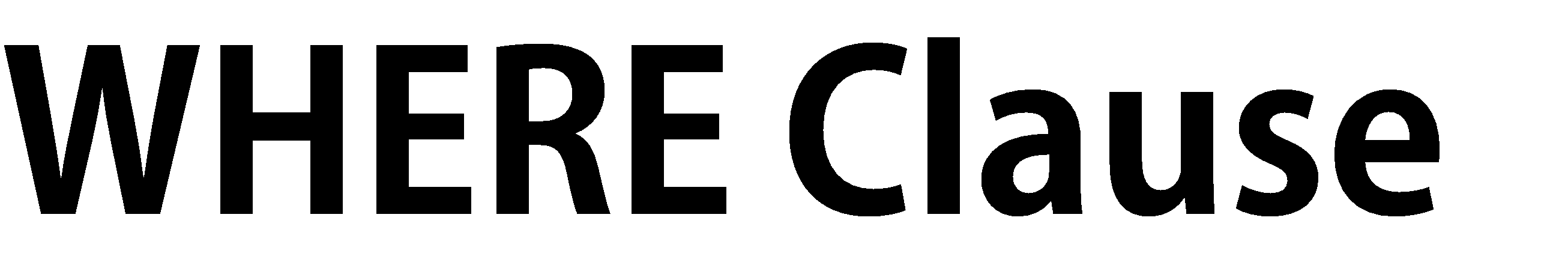
**SELECT** \* FROM table\_name;

To select distinct/unique fields available in the table

* **Syntax**

**SELECT DISTINCT** Column\_name FROM table\_name;

32



The WHERE clause is used to filter records.

**It is used to extract only those records that fulfill a specified condition**

* Syntax

**SELECT** column\_name FROM table\_name

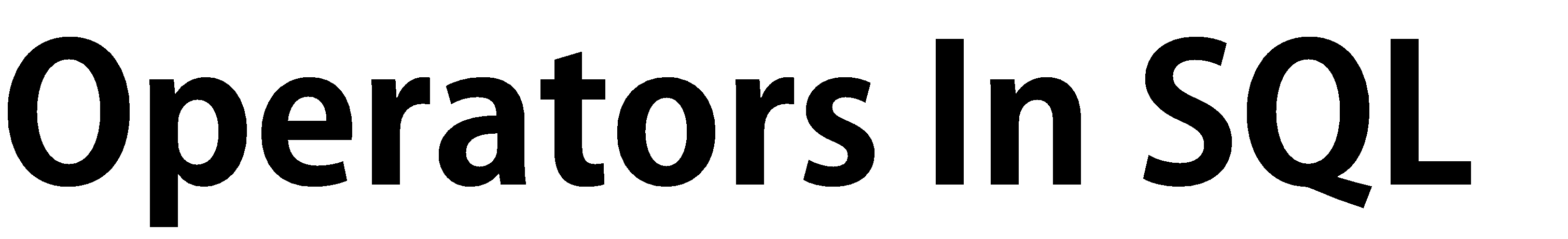
**WHERE** conditions;

* Example

**SELECT** name FROM classroom

**WHERE** grade=‘A’;

33



The SQL reserved words and characters are called operators, which are used with a WHERE clause in a SQL query

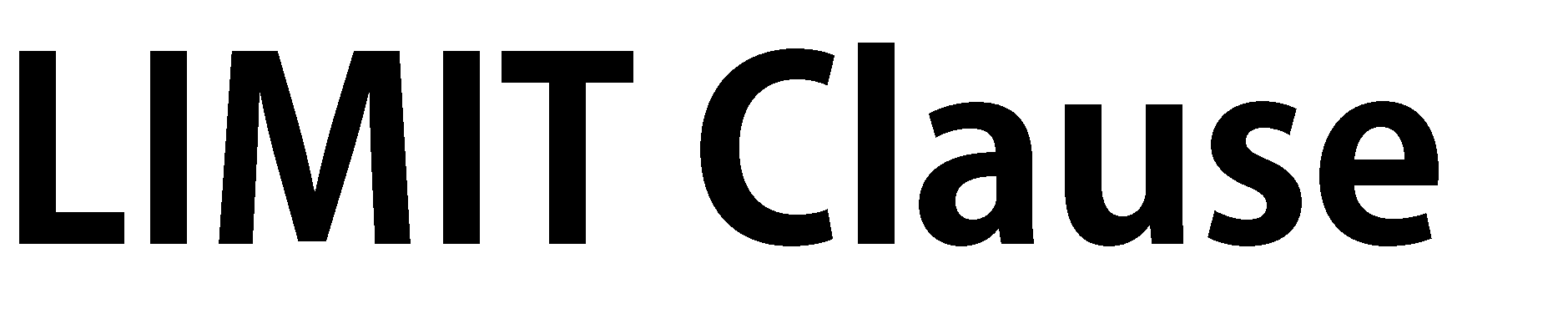
Most used operators:

1. **Arithmetic operators :** arithmetic operations on numeric values

Example: Addition (+), Subtraction (-), Multiplication (\*), Division (/), Modulus (%)

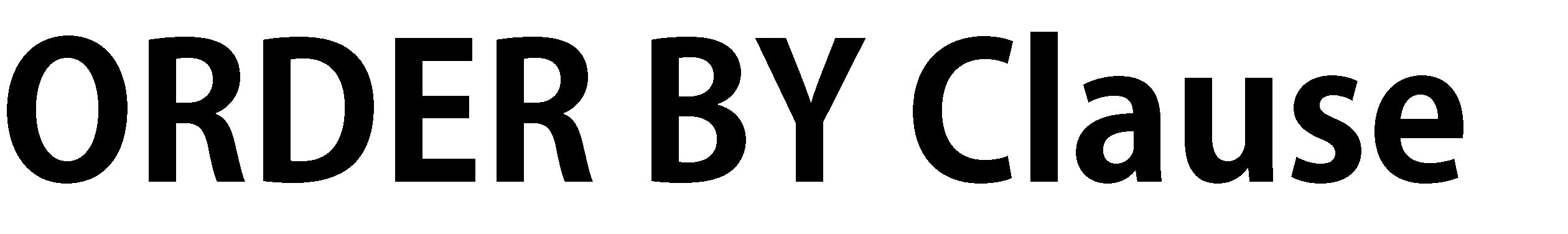
1. **Comparison operators:** compare two different data of SQL table
   * Example: Equal (=), Not Equal (!=), Greater Than (>), Greater Than Equals to (>=)
2. **Logical operators:** perform the Boolean operations
   * Example: ALL, IN, BETWEEN, LIKE, AND, OR, NOT, ANY
3. **Bitwise operators:** perform the bit operations on the Integer values
   * Example: Bitwise AND (&), Bitwise OR(|)

34



**The LIMIT clause is used to set an upper limit on the number of tuples returned by SQL.**

**Example:** below code will return 5 rows of data **SELECT** column\_name FROM table\_name **LIMIT** 5;



**The ORDER BY is used to sort the result-set in ascending (ASC) or descending**

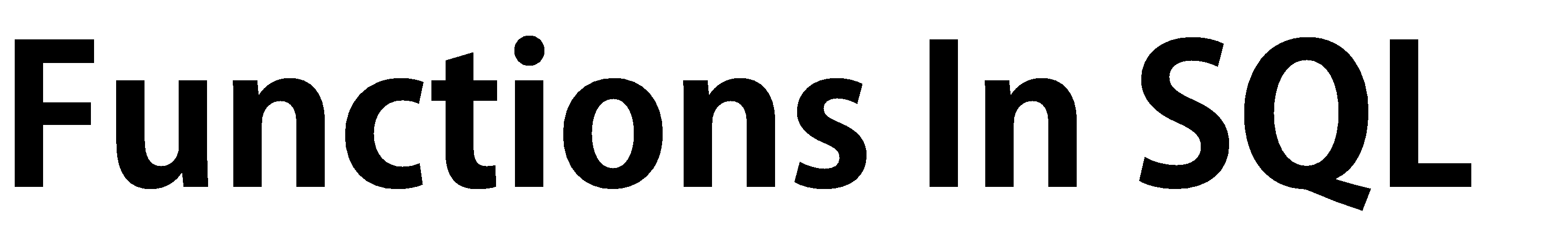
**order (DESC).**

**Example:** below code will sort the output data by column name in ascending order

**SELECT** column\_name FROM table\_name

**ORDER BY** column\_name e **ASC**;

35

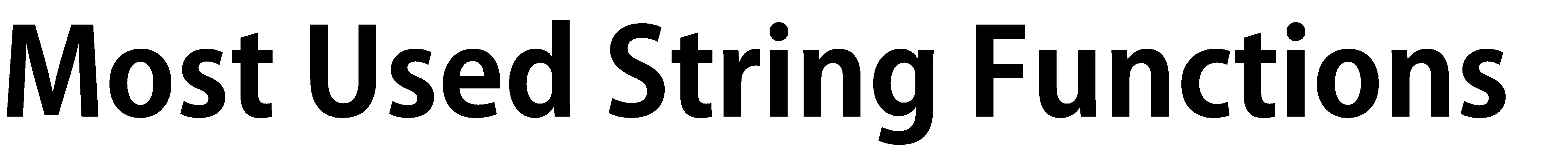


Functions in SQL are the database objects that contains a set of SQL statements to perform a specific task. A function accepts input parameters, perform actions, and then return the result.

**Types of Function:**

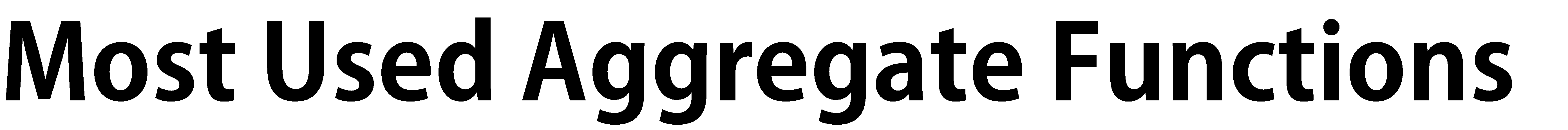
1. System Defined Function : these are built-in functions
   * Example: rand(), round(), upper(), lower(), count(), sum(), avg(), max(), etc
2. User-Defined Function : Once you define a function, you can call it in the same way as the built-in functions

39



String functions are used to perform an operation on input string and return an output string

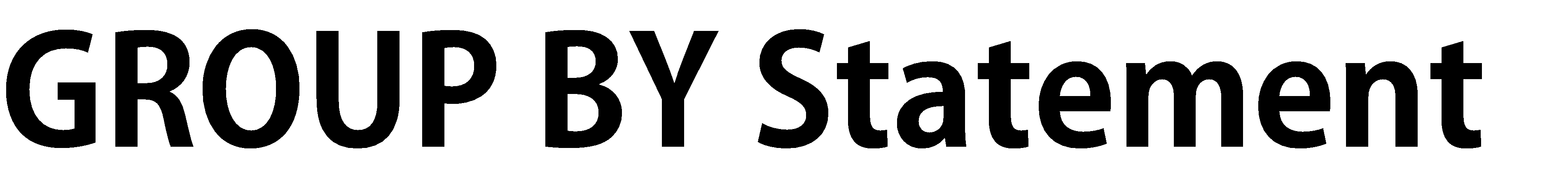
* **UPPER()** converts the value of a field to uppercase
* **LOWER()** converts the value of a field to lowercase
* **LENGTH()** returns the length of the value in a text field
* **SUBSTRING()** extracts a substring from a string---substring(name,1,3)
* **NOW()** returns the current system date and time
* **FORMAT()** used to set the format of a field
* **CONCAT()** adds two or more strings together---concat(firsname,lastname)
* **REPLACE()** Replaces all occurrences of a substring within a string, with a new substring
* **TRIM()** removes leading and trailing spaces (or other specified characters) from a string



Aggregate function performs a calculation on multiple values and returns a single value.

And Aggregate functiona are often used with GROUP BY & SELECT statement

* **COUNT()** returns number of values
* **SUM()** returns sum of all values
* **AVG()** returns average value
* **MAX()** returns maximum value
* **MIN()** returns minimum value
* **ROUND()** Rounds a number to a specified number of decimal places



The GROUP BY statement group rows that have the same values into summary rows.

It is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns

* **Syntax**

**SELECT** column\_name(s)

**FROM** table\_name

**GROUP BY** column\_name(s);

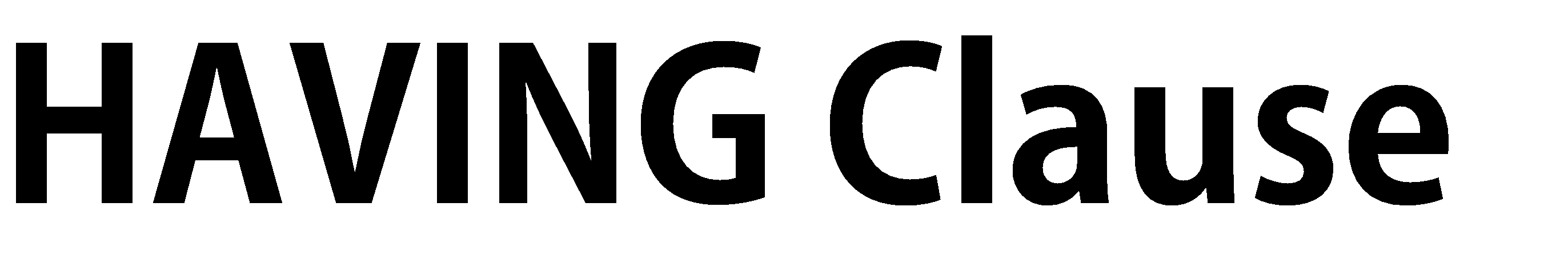
* **Example**

**SELECT** mode, SUM(amount) AS total

**FROM** payment

**GROUP BY** mode

44



The **HAVING** clause is used to apply a filter on the result of **GROUP BY** based on the

specified condition.

The **WHERE** clause places conditions on the selected columns, whereas the **HAVING**

clause places conditions on groups created by the **GROUP BY** clause

**Syntax**

SELECT column\_name(s)

FROM table\_name WHERE condition(s)

**GROUP BY** column\_name(s)

**HAVING** condition(s)

* **Example**

SELECT mode, COUNT(amount) AS total

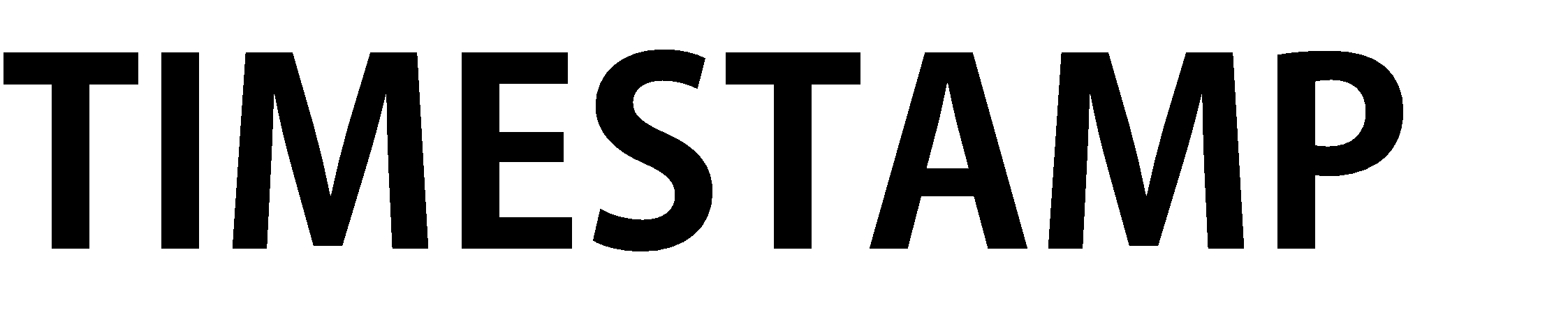
FROM payment

**GROUP BY** mode

**HAVING** COUNT(amount) >= 3

ORDER BY total DESC

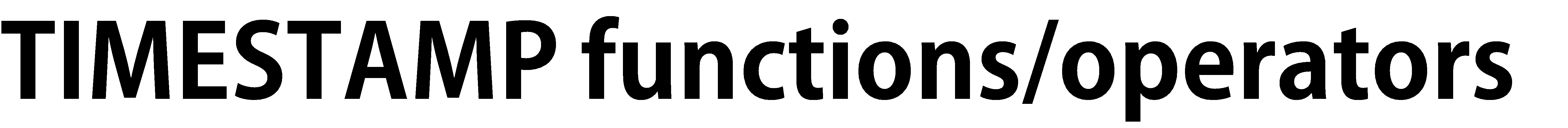
45



#### The **TIMESTAMP** data type is used for values that contain both date and time parts

* **TIME** contains only time, format HH:MI:SS
* **DATE** contains on date, format YYYY-MM-DD
* **YEAR** contains on year, format YYYY or YY
* **TIMESTAMP** contains date and time, format YYYY-MM-DD HH:MI:SS
* **TIMESTAMPTZ** contains date, time and time zone

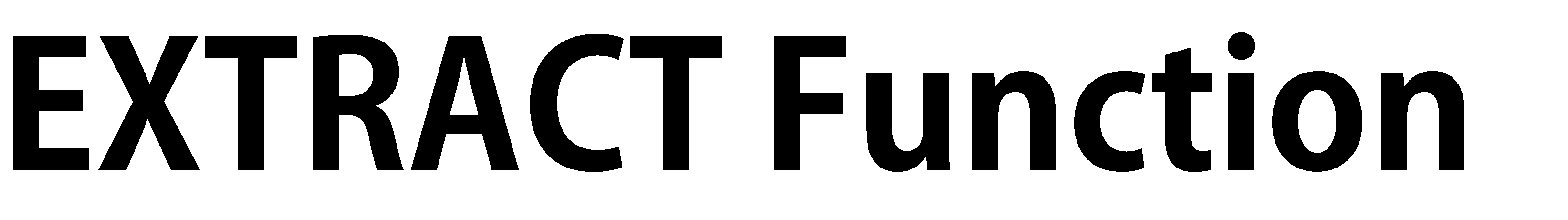
48



Below are the TIMESTAMP functions and operators in SQL:

* SHOW TIMEZONE
* SELECT NOW()
* SELECT TIMEOFDAY()
* SELECT CURRENT\_TIME
* SELECT CURRENT\_DATE

49

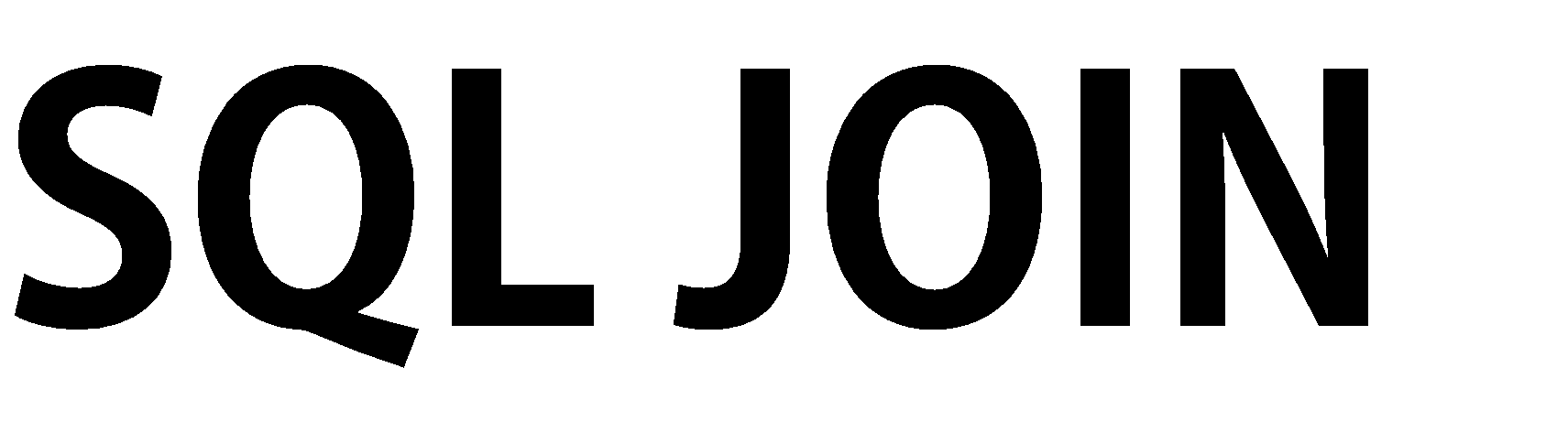


The **EXTRACT()** function extracts a part from a given date value.

**Syntax:** SELECT **EXTRACT**(MONTH FROM date\_field) FROM Table

* **YEAR**
* **QUARTER**
* **MONTH**
* **WEEK**
* **DAY**
* **HOUR**
* **MINUTE**
* **DOW** – day of week
* **DOY** – day of year

50

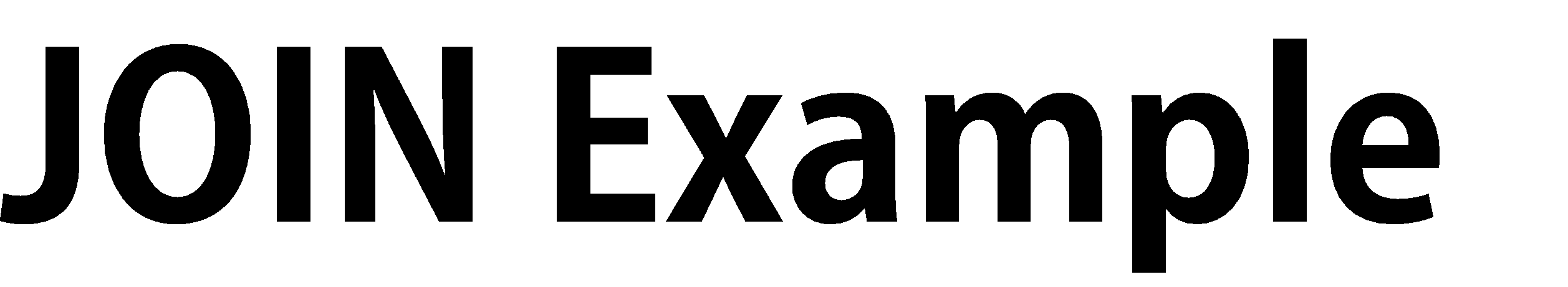


#### **JOIN** means to combine something.

* A **JOIN** clause is used to combine data from two or more tables, based on a related column between them

#### Let’s understand the joins through an example:

53



Question: How much amount was paid by customer ‘Madan’, what was

mode and payment date?

Database

54

payment

customer\_id

customer

customer\_id first\_name last\_name address\_id

amount

mode Payment\_date

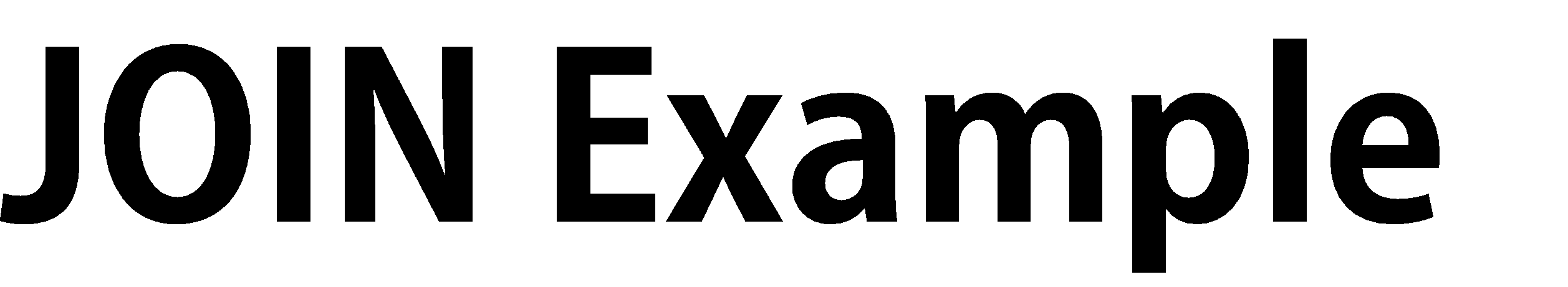
address

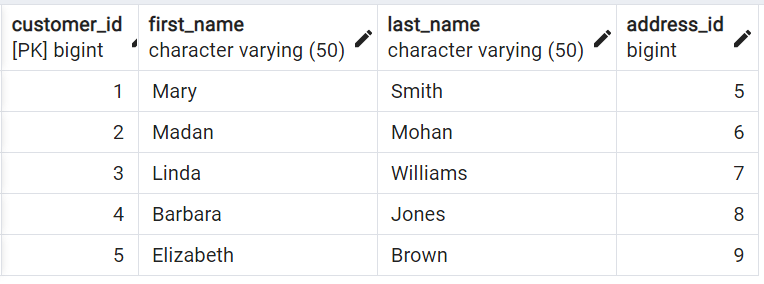
address\_id address city\_id postal\_code phone

country

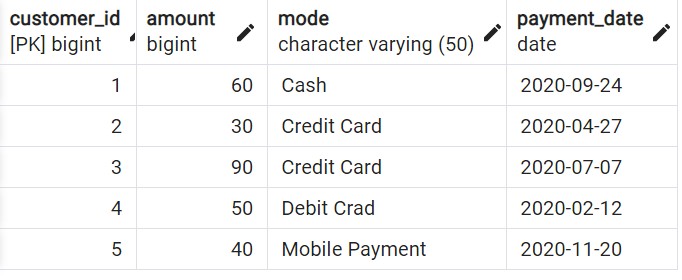
city\_id city country

SQL By a



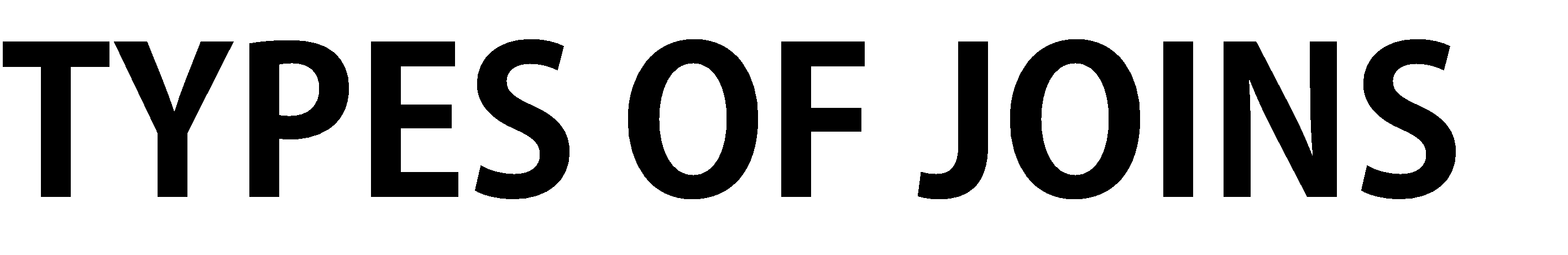
Question: How much amount was paid by customer ‘Madan’, what was mode and payment date?

|  |
| --- |
|  |
|  |
|  |

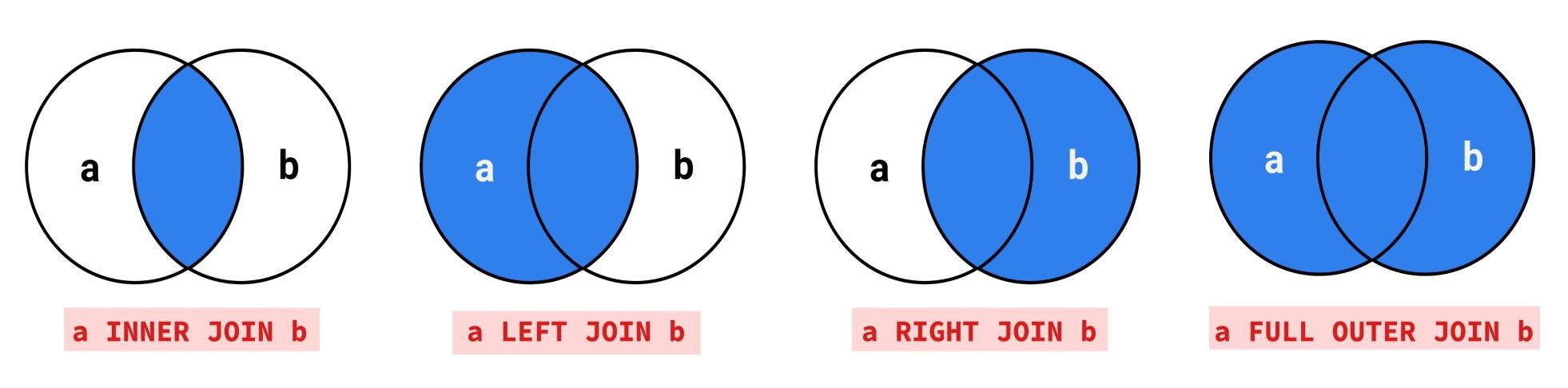
Answer: Amount = 30, Mode = Credit Card, Date = 2020-04-27

|  |
| --- |
|  |
|  |
|  |

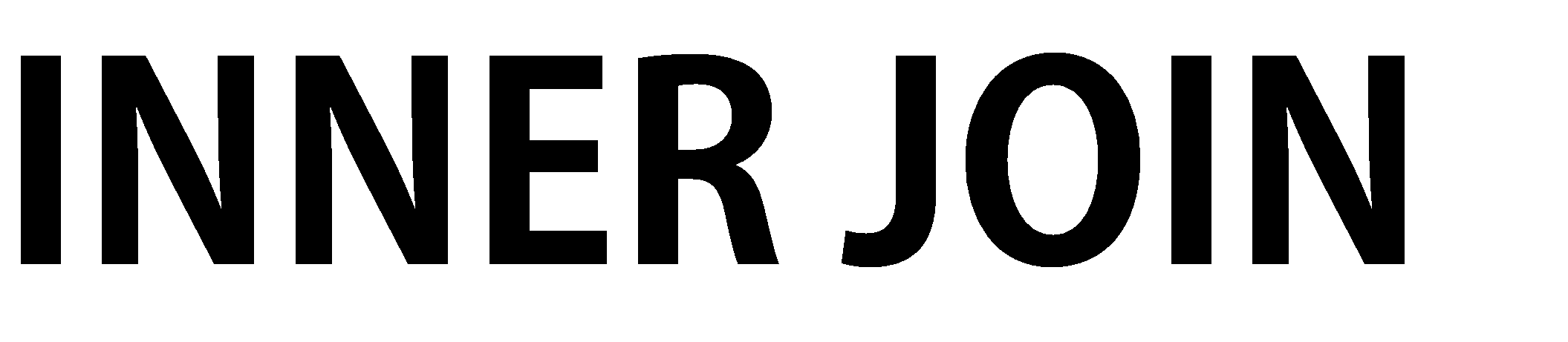
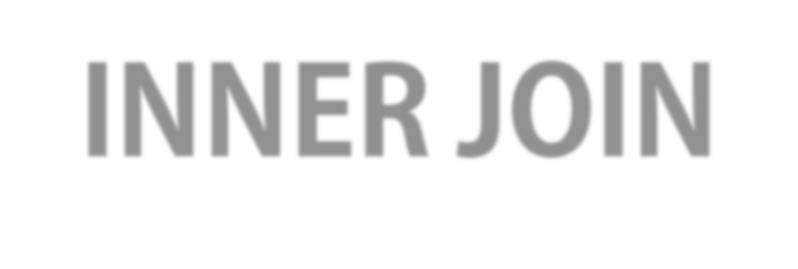
55



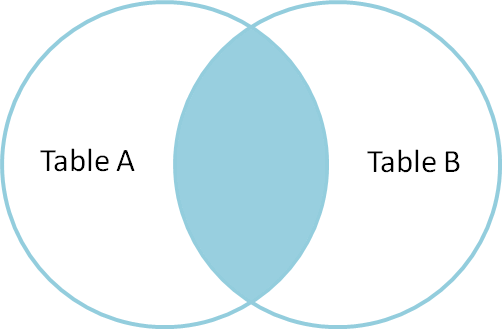
* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN



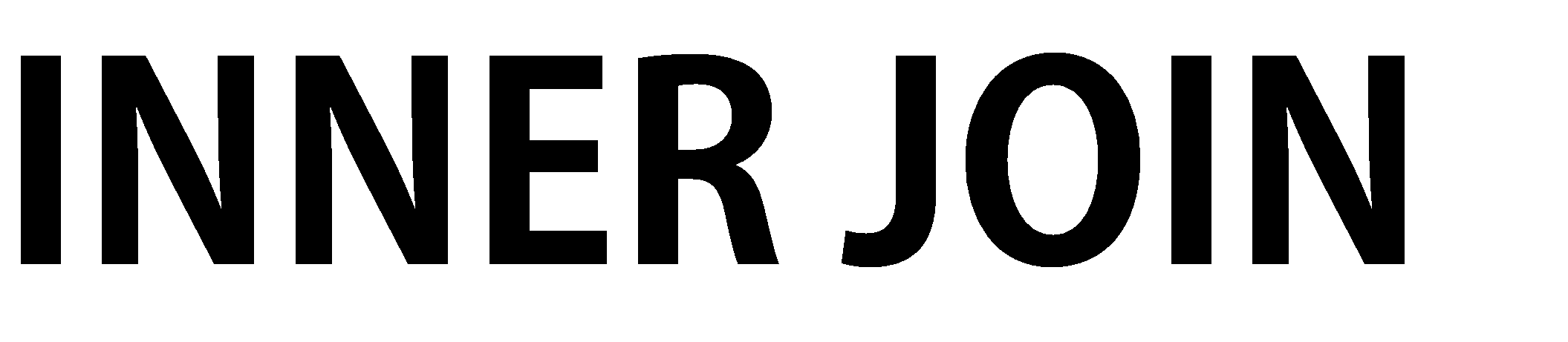
56



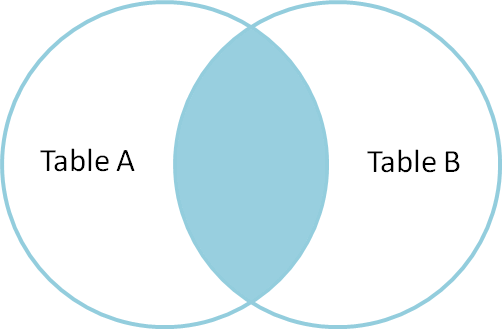
* Returns records that have matching values in both tables



57



* Syntax

SELECT column\_name(s) FROM **TableA**

INNER JOIN TableB

**ON TableA**.col\_name = **TableB**.col\_name

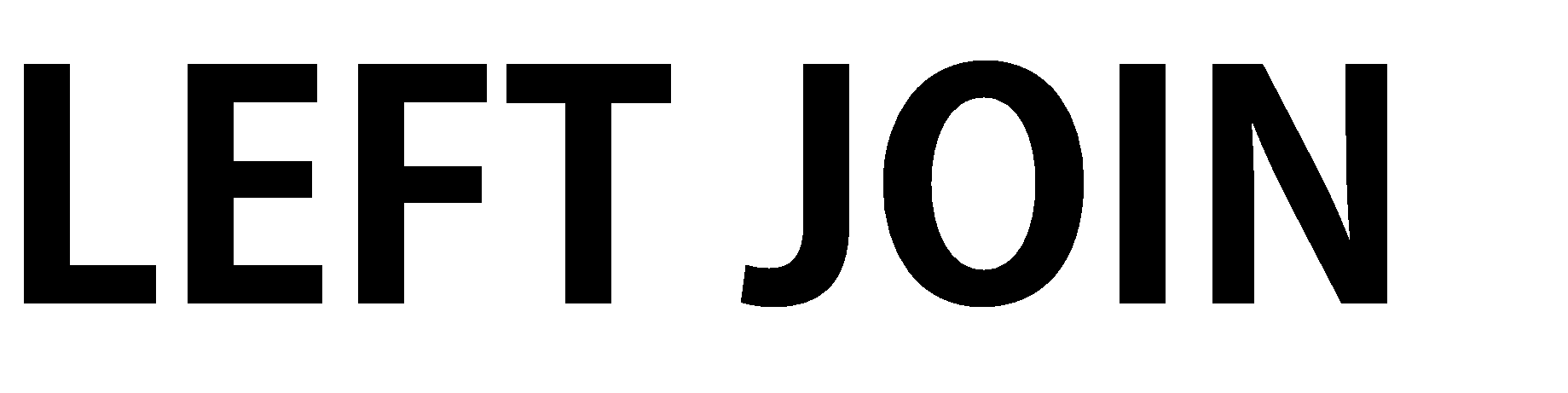
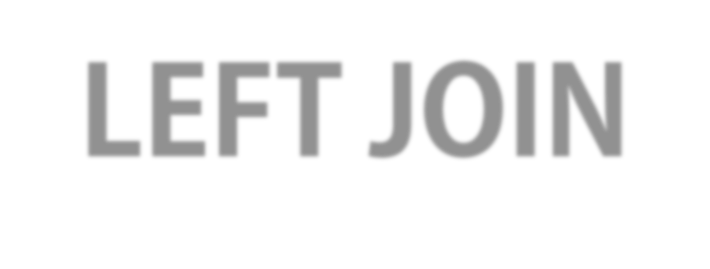
* Example

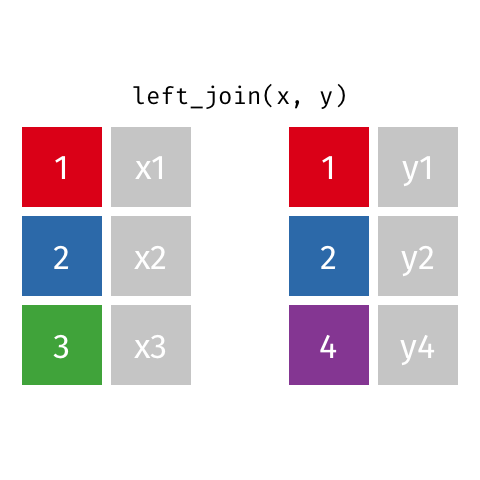
SELECT \*

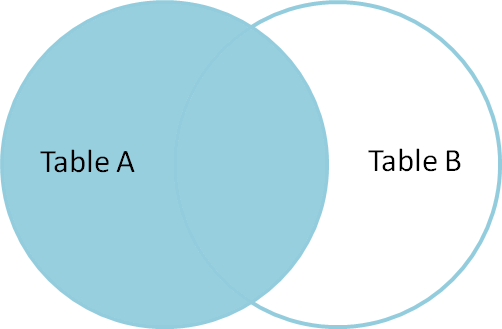
FROM customer AS c INNER JOIN payment AS p

ON c.customer\_id = p.customer\_id

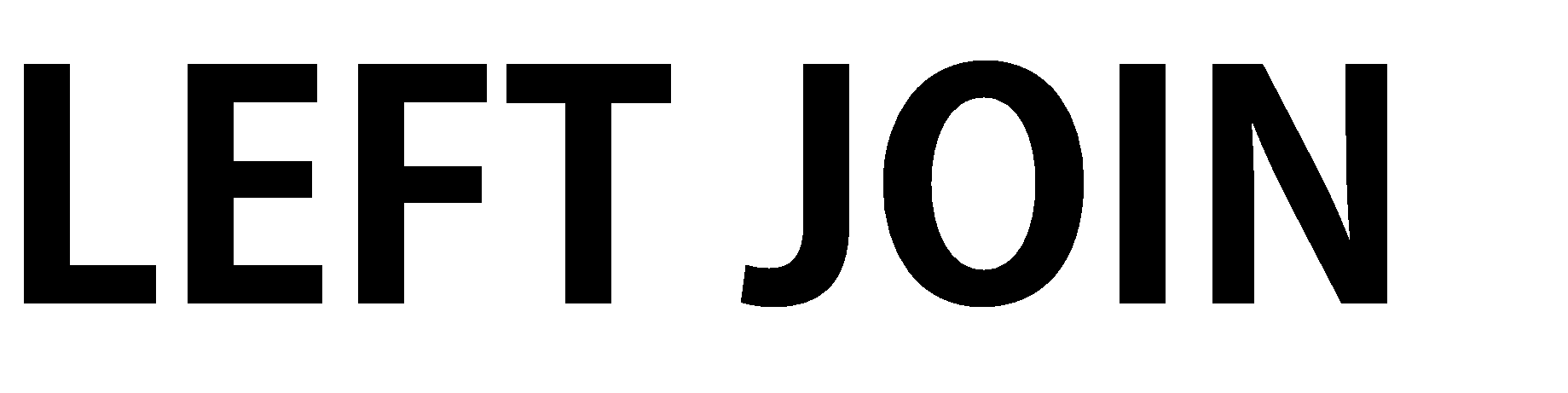
58



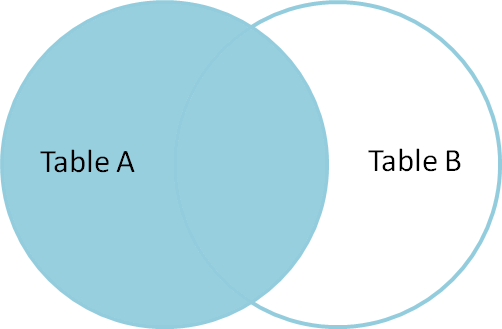
* Returns all records from the left table, and the matched records from the right table



59



* Syntax

SELECT column\_name(s) FROM **TableA**

LEFT JOIN TableB

**ON TableA**.col\_name = **TableB**.col\_name

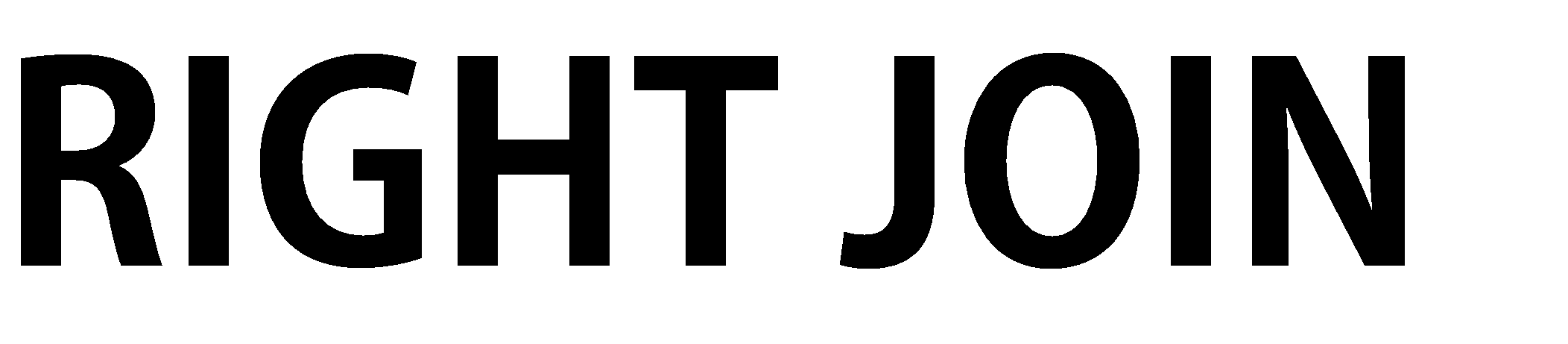
* Example

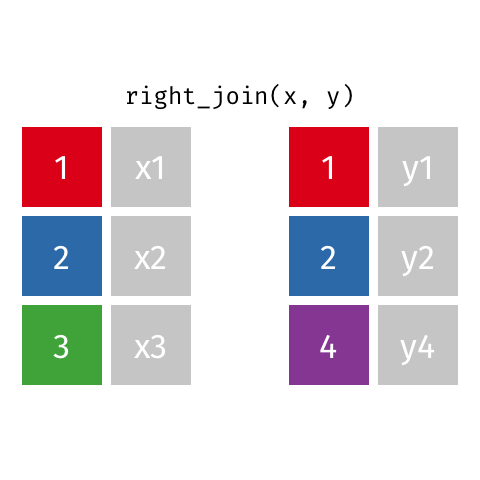
SELECT \*

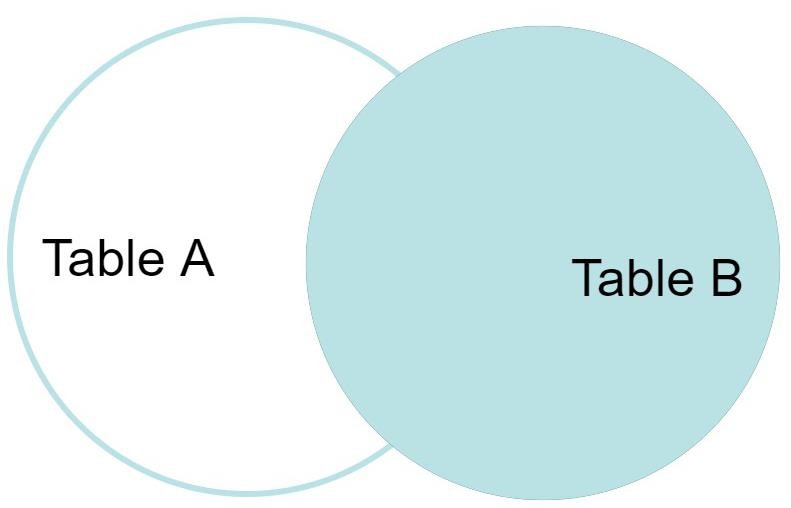
FROM customer AS c LEFT JOIN payment AS p

ON c.customer\_id = p.customer\_id

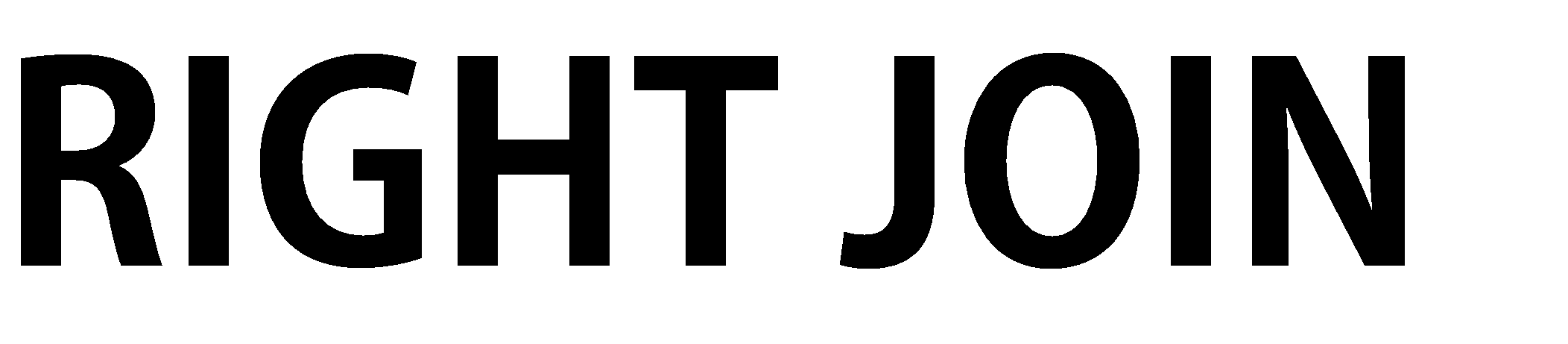
60



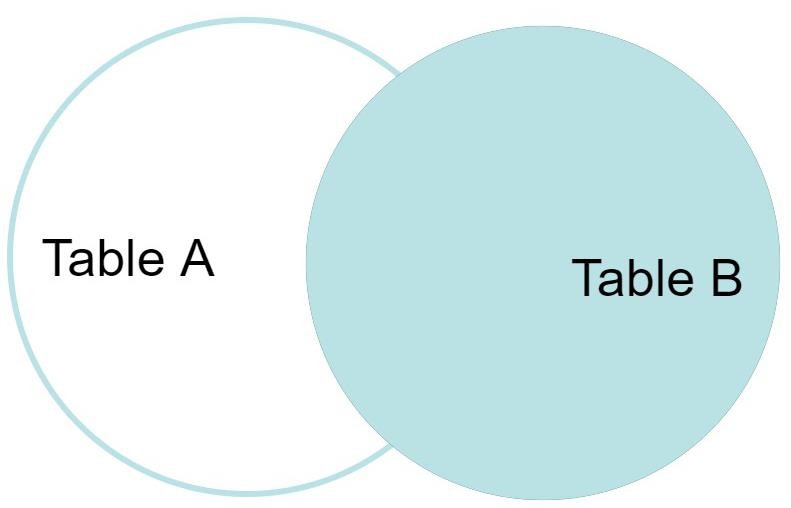
* Returns all records from the right table, and the matched records from the left table



61



* Syntax

SELECT column\_name(s) FROM **TableA**

RIGHT JOIN TableB

**ON TableA**.col\_name = **TableB**.col\_name

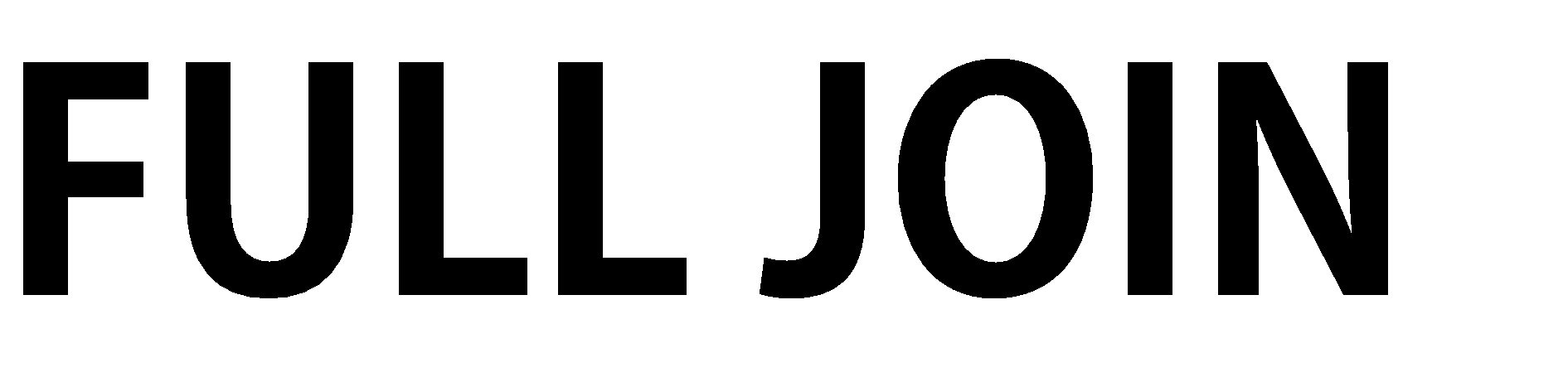
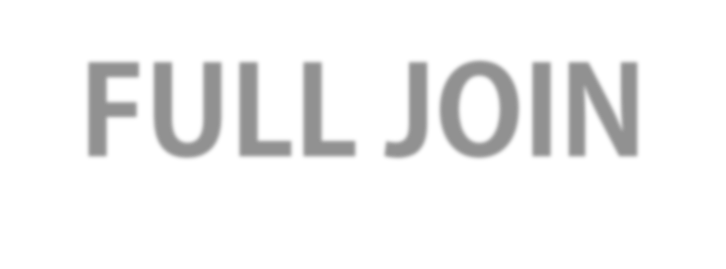
* Example

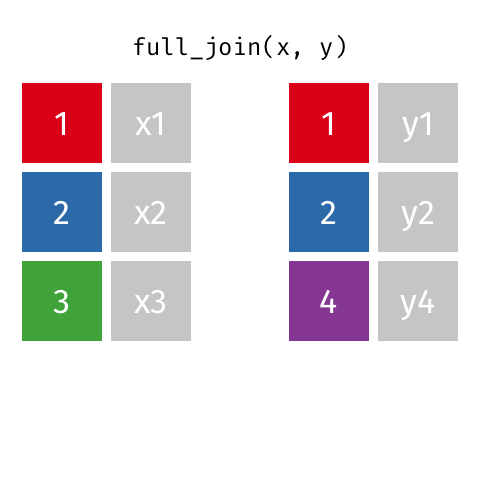
SELECT \*

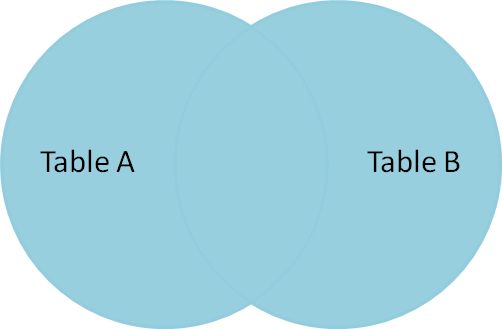
FROM customer AS c RIGHT JOIN payment AS p

ON c.customer\_id = p.customer\_id

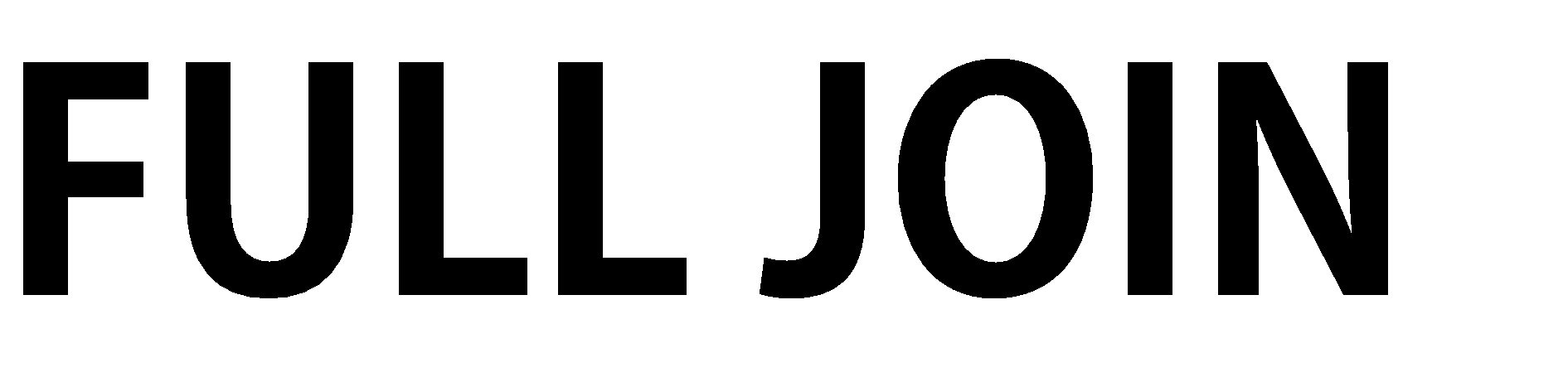
62



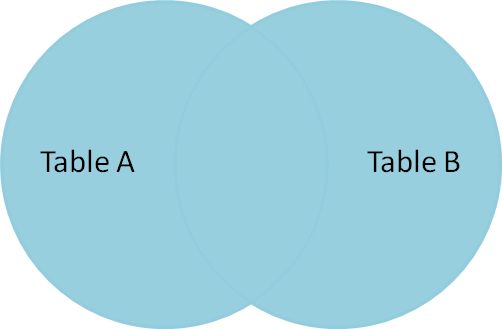
* Returns all records when there is a match in either left or right table



63



* Syntax

SELECT column\_name(s) FROM **TableA**

FULL OUTER JOIN TableB

**ON TableA**.col\_name = **TableB**.col\_name

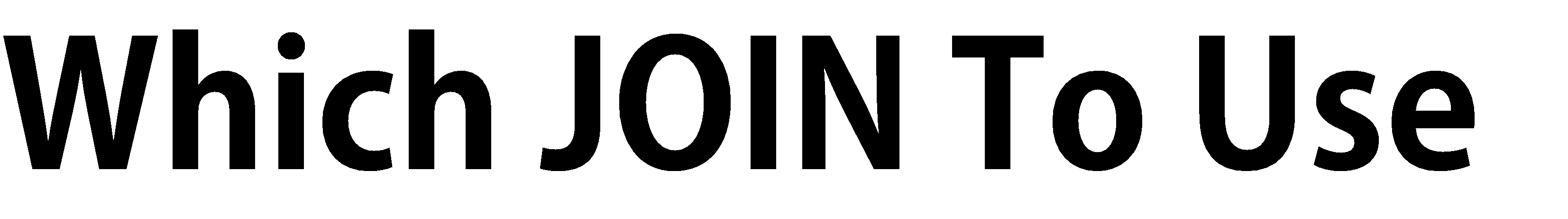
* Example

SELECT \*

FROM customer AS c

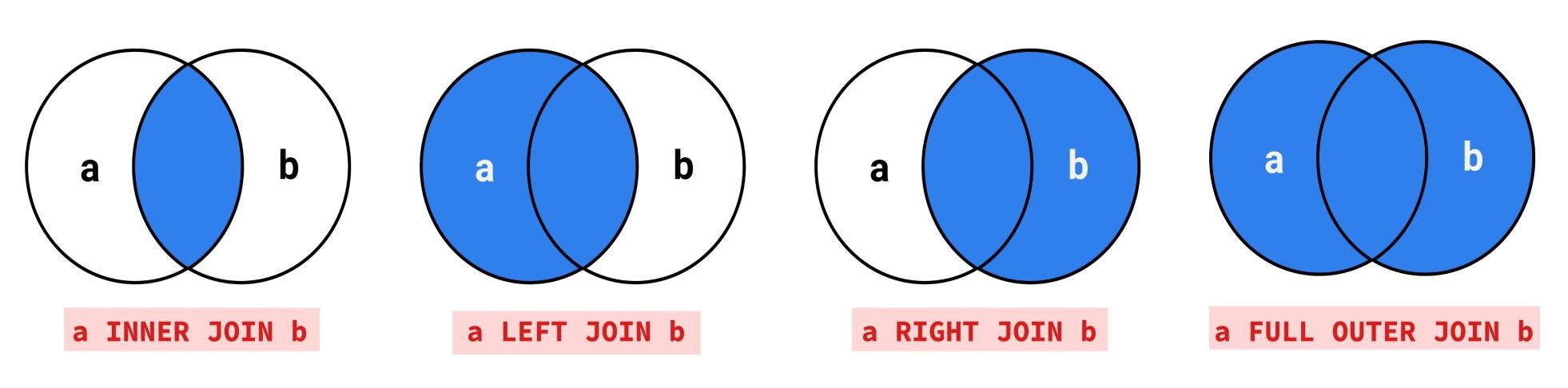
FULL OUTER JOIN payment AS p ON c.customer\_id = p.customer\_id

64

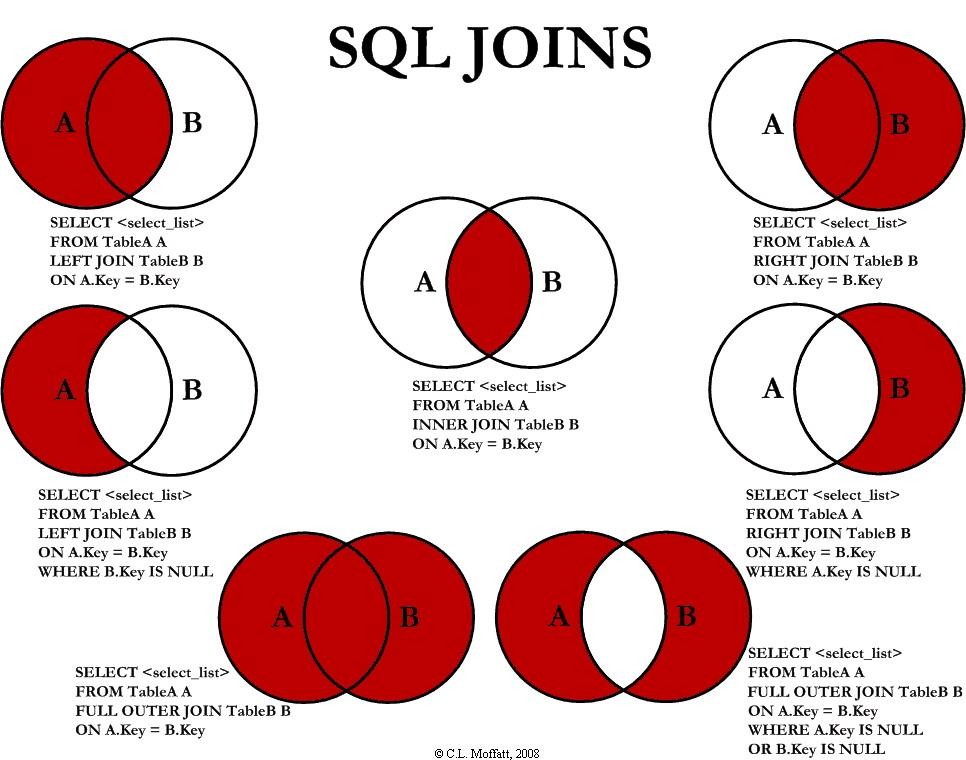
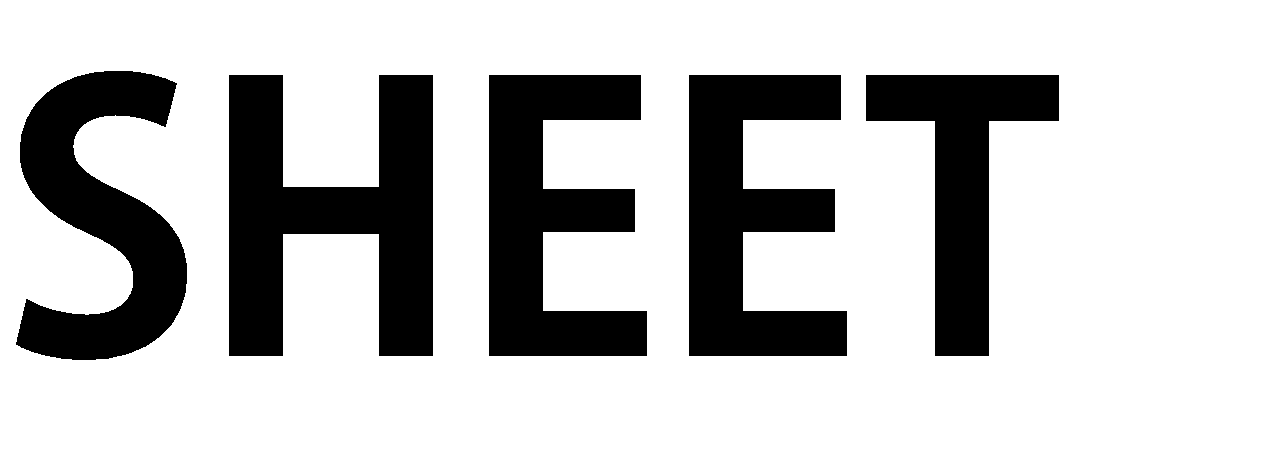
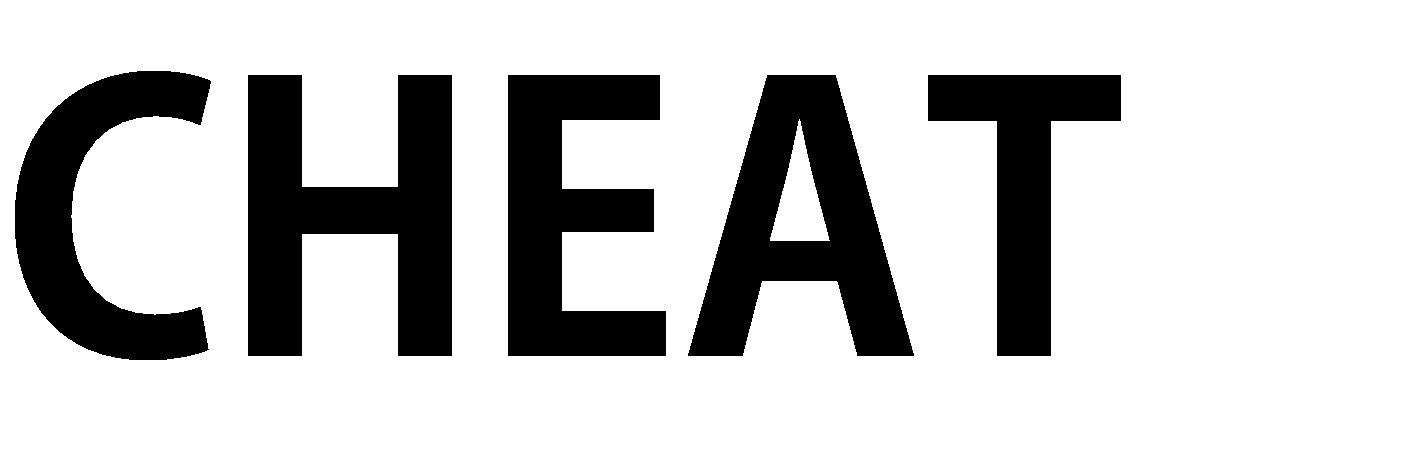
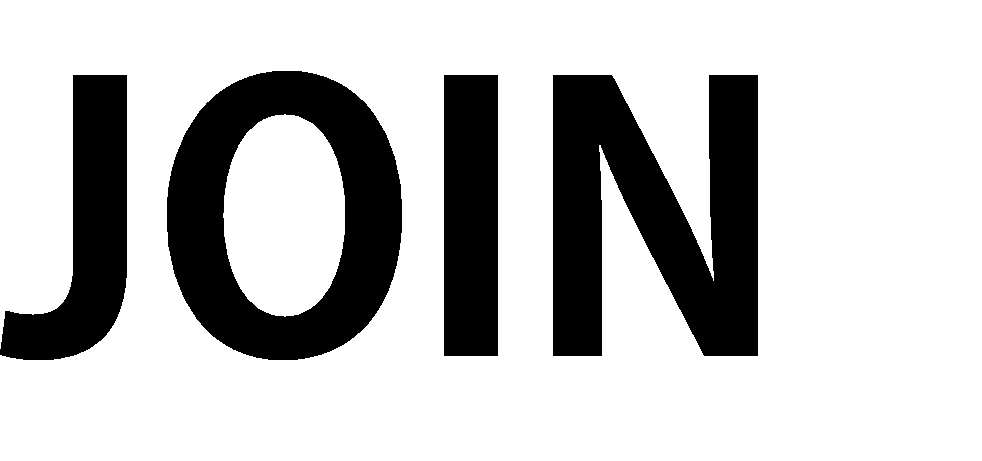


* **INNER JOIN:** Returns records that have matching values in both tables
* **LEFT JOIN:** Returns all records from the left table, and the matched records from the right table
* **RIGHT JOIN:** Returns all records from the right table, and the matched records from the left table
* **FULL JOIN:** Returns all records when there is a match in either left or right table

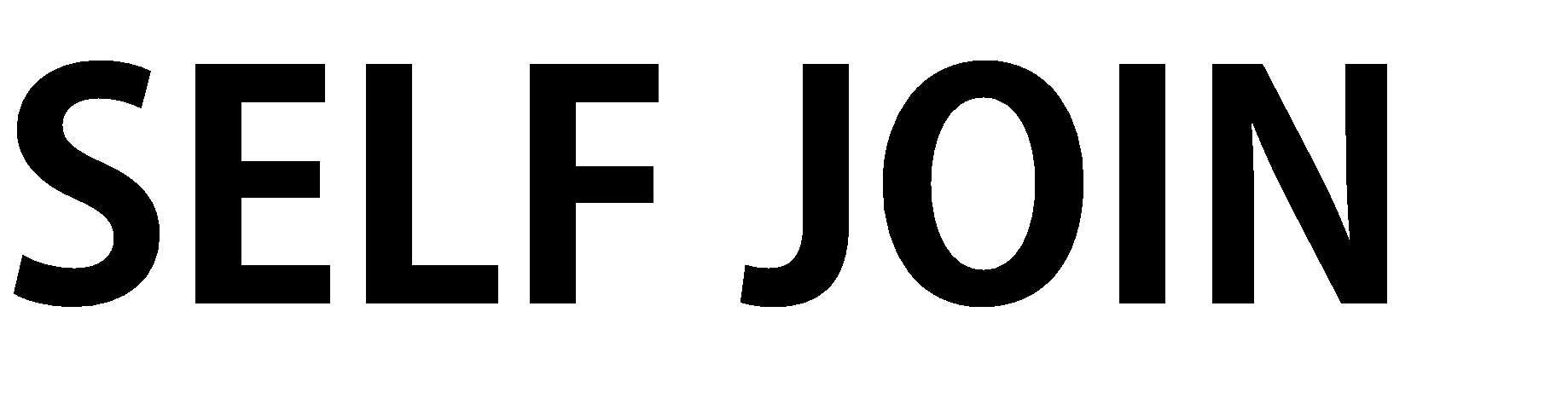
65



SQL By a



66



#### A **self join** is a regular join in which a table is joined to itself

* **SELF Joins** are powerful for comparing values in a column of rows with the same table

#### Syntax

SELECT column\_name(s) FROM **Table AS T1**

**JOIN Table AS T2**

**ON T1**.col\_name = **T2**.col\_name

68



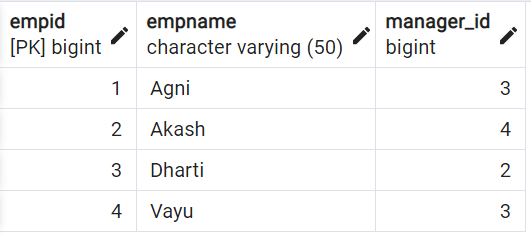
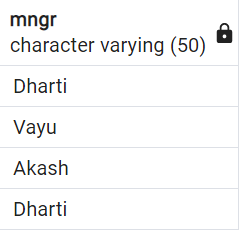
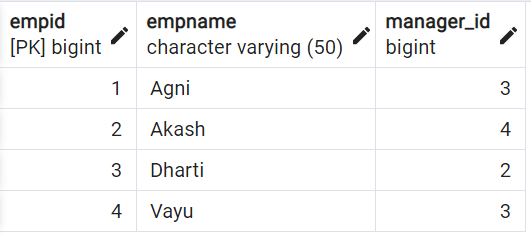


Table: emp

* Find the name of respective managers for each of the employees?

69

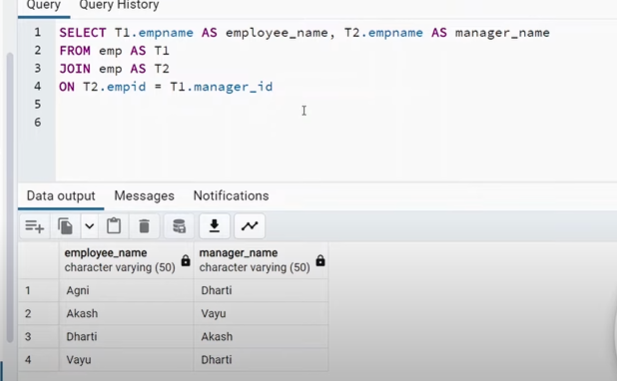




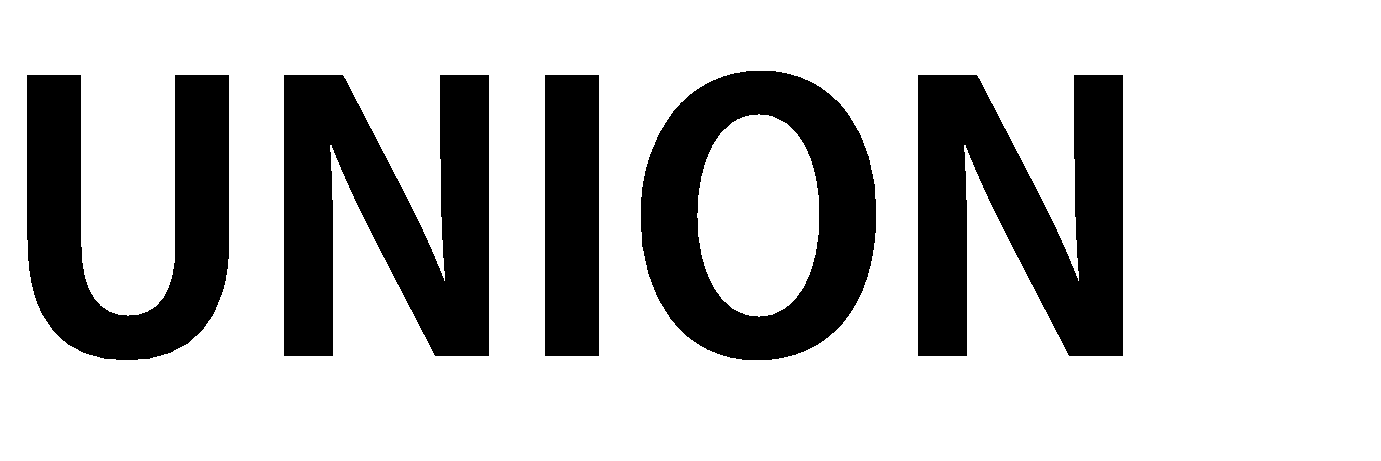
SELECT T2.empname, T1.empname

FROM emp AS T1 JOIN emp AS T2

ON T1.empid = T2.manager\_id



70



The SQL **UNION** clause/operator is used to combine/concatenate the results of two or more SELECT statements without returning any duplicate rows and keeps **unique records**

To use this UNION clause, each SELECT statement must have

* + The same number of columns selected and expressions
  + The same data type and
  + Have them in the same order
  + **Syntax**

SELECT column\_name(s) FROM **TableA UNION**

SELECT column\_name(s) FROM **TableB**

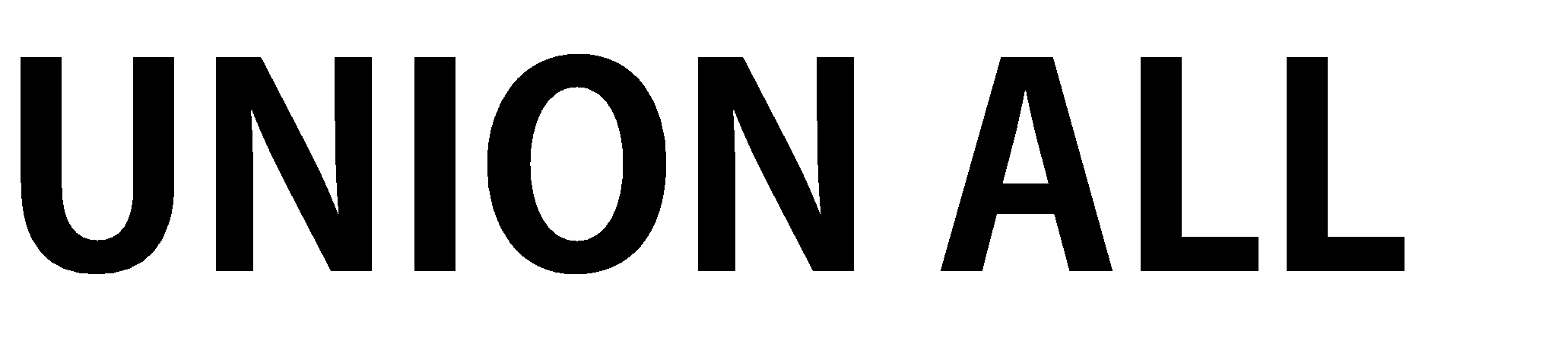
* + **Example**

SELECT cust\_name, cust\_amount from custA

**UNION**

SELECT cust\_name, cust\_amount from custB

71



In **UNION ALL** everything is same as **UNION**, it combines/concatenate two or more table but keeps all records, **including duplicates**

* + **Syntax**

SELECT column\_name(s) FROM **TableA UNION ALL**

SELECT column\_name(s) FROM **TableB**

* + **Example**

SELECT cust\_name, cust\_amount from custA

**UNION ALL**

SELECT cust\_name, cust\_amount from custB

72

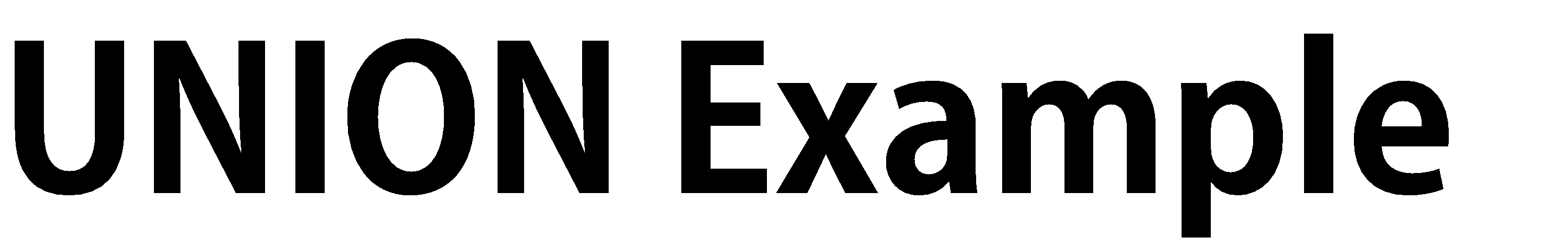
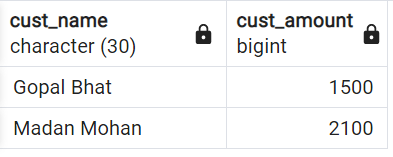
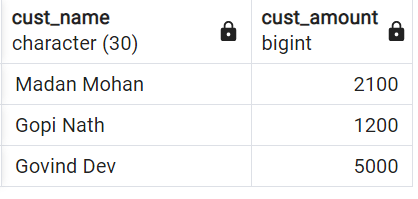
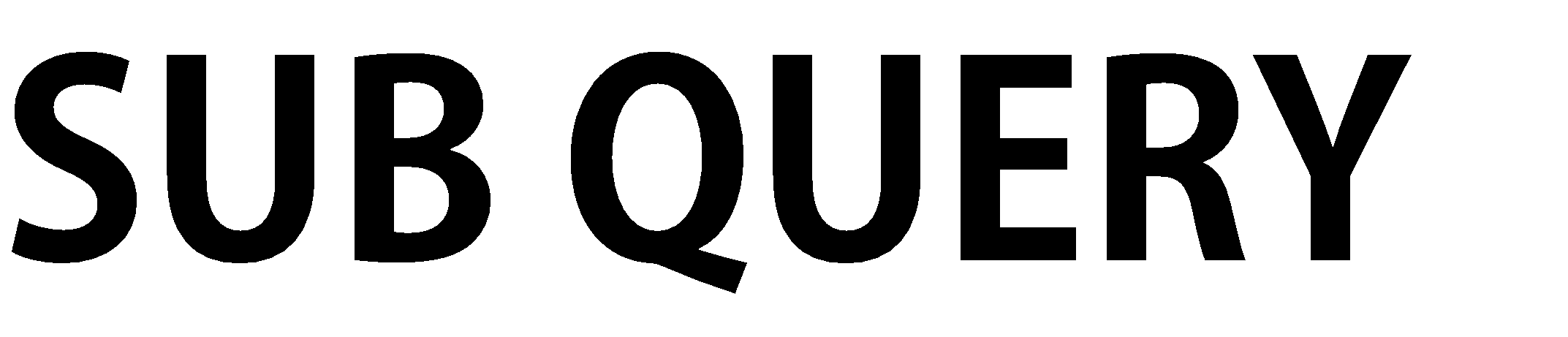


Table: custA

Table: custB



73



A **Subquery** or Inner query or a Nested query allows us to create complex query on the output of another query

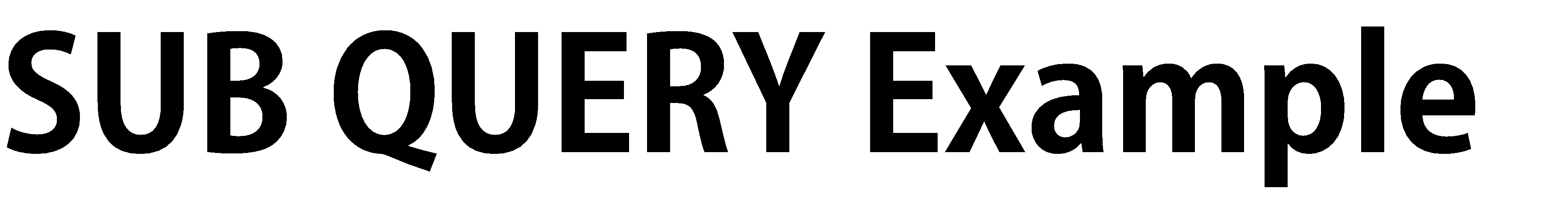
* + Sub query syntax involves two SELECT statements
  + **Syntax**

SELECT column\_name(s) FROM table\_name

WHERE column\_name ***operator***

###### ( SELECT column\_name FROM table\_name WHERE ... );

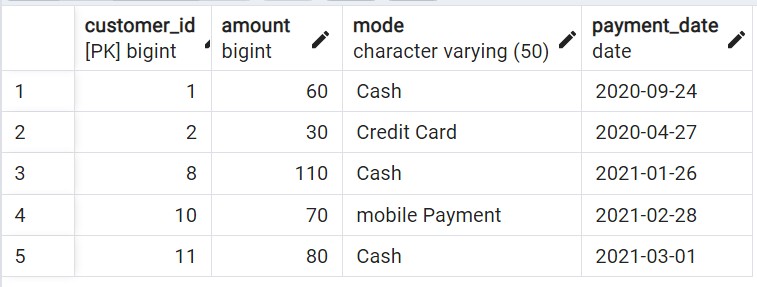
75

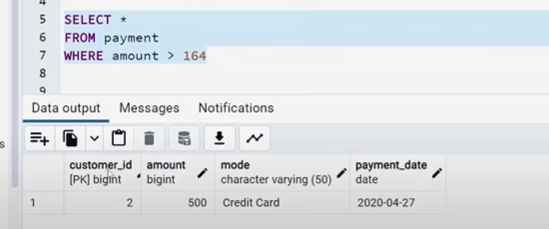
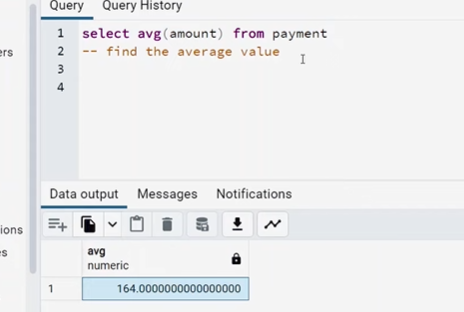


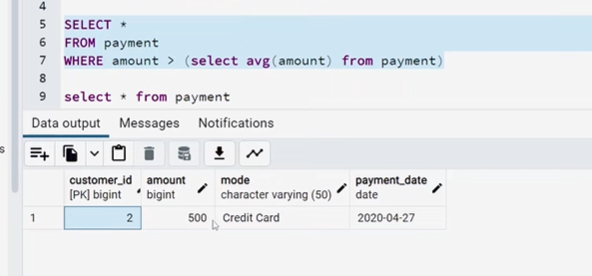
**Question:** Find the details of customers, whose payment amount is more than the average of total amount paid by all customers

Divide above question into two parts:

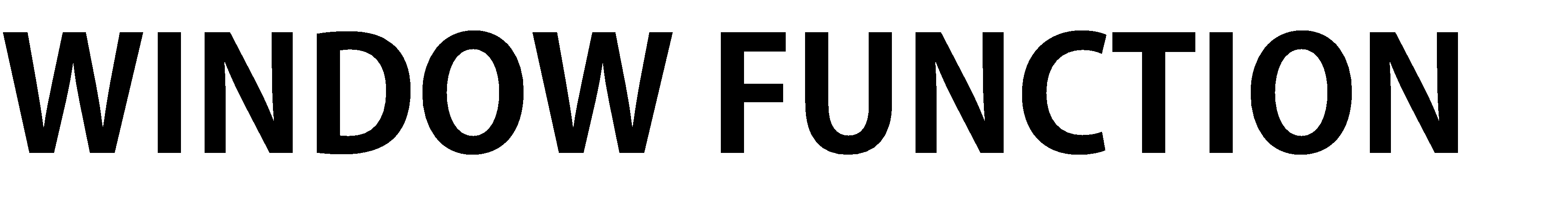
1. Find the average amount
2. Filter the customers whose amount **>** average amount



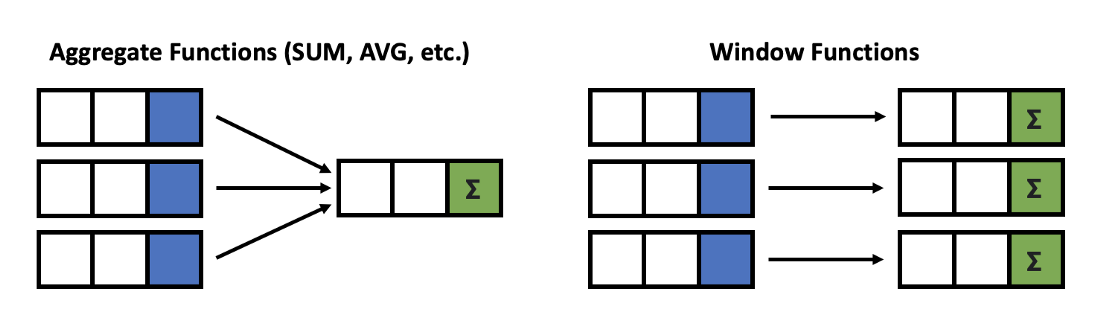
Subquery example 



Optimistic (dynamic) 76



* + **Window functions** applies aggregate, ranking and analytic functions over a particular window (set of rows).
  + And **OVER** clause is used with window functions to define that window.



Give output one row per aggregation The rows maintain their separate identities

79

SELECT column\_name(s),

**(** [ <PARTITION BY Clause> ] [ <ORDER BY Clause> ]

[ <ROW or RANGE Clause> ] **)**

FROM table\_name

**OVER**

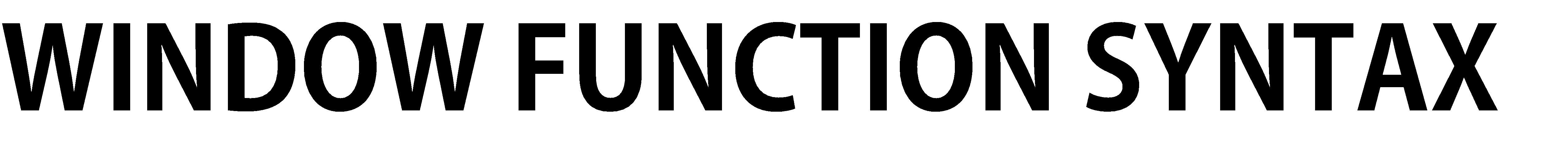
**fun**( )

**Select a function**

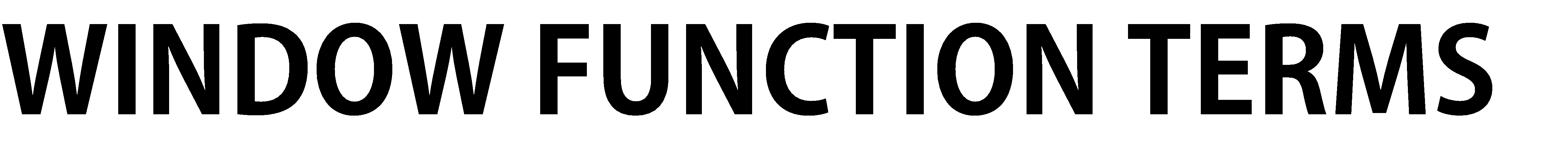
* + Aggregate functions
  + Ranking functions
  + Analytic functions

**Define a Window**

* + PARTITION BY
  + ORDER BY
  + ROWS



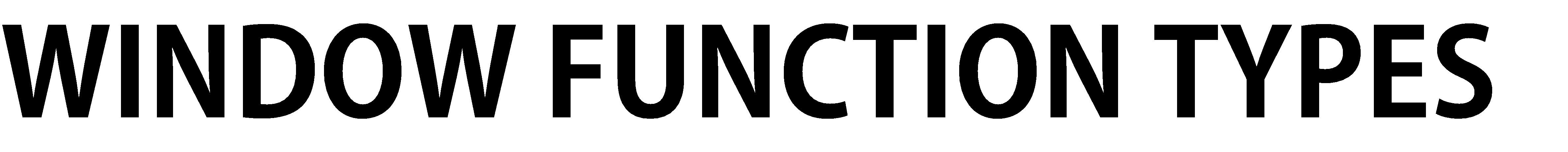
80



Let’s look at some definitions:

* + **Window function** applies aggregate, ranking and analytic functions over a particular window; for example, sum, avg, or row\_number
  + **Expression** is the name of the column that we want the window function operated on. This may not be necessary depending on what window function is used
  + **OVER** is just to signify that this is a window function
  + **PARTITION BY** divides the rows into partitions so we can specify which rows to use to compute the window function
  + **ORDER BY** is used so that we can order the rows within each partition. This is optional and does not have to be specified
  + **ROWS** can be used if we want to further limit the rows within our partition. This is optional and usually not used

81



There is no official division of the SQL window functions into categories but high level we can divide into three types

**Value/Analytic**

* LEAD
* LAG
* FIRST\_VALUE
* LAST\_VALUE

**Ranking**

* ROW\_NUMBER
* RANK
* DENSE\_RANK
* PERCENT\_RANK

**Aggregate**

* SUM
* AVG
* COUNT
* MIN
* MAX

**Window Functions**

82

**SELECT** new\_id, new\_cat,

**AGGREGATE FUNCTION**

**Example**

**SUM**(new\_id) **OVER**( PARTITION BY new\_cat ORDER BY new\_id ) AS "Total", **AVG**(new\_id) **OVER**( PARTITION BY new\_cat ORDER BY new\_id ) AS "Average", **COUNT**(new\_id) **OVER**( PARTITION BY new\_cat ORDER BY new\_id ) AS "Count", **MIN**(new\_id) **OVER**( PARTITION BY new\_cat ORDER BY new\_id ) AS "Min", **MAX**(new\_id) **OVER**( PARTITION BY new\_cat ORDER BY new\_id ) AS "Max" **FROM** test\_data

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **new\_id** | **new\_cat** | **Total** | **Average** | **Count** | **Min** | **Max** |
| 100 | Agni | 300 | 150 | 2 | 100 | 200 |
| 200 | Agni | 300 | 150 | 2 | 100 | 200 |
| 500 | Dharti | 1200 | 600 | 2 | 500 | 700 |
| 700 | Dharti | 1200 | 600 | 2 | 500 | 700 |
| 200 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |
| 300 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |
| 500 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |

83

**SELECT** new\_id, new\_cat,

**SUM**(new\_id) **OVER**( ORDER BY new\_id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Total", **AVG**(new\_id) **OVER**( ORDER BY new\_id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Average", **COUNT**(new\_id) **OVER**( ORDER BY new\_id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Count", **MIN**(new\_id) **OVER**( ORDER BY new\_id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Min", **MAX**(new\_id) **OVER**( ORDER BY new\_id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Max"

**FROM** test\_data

**AGGREGATE FUNCTION**

**Example**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **new\_id** | **new\_cat** | **Total** | **Average** | **Count** | **Min** | **Max** |
| 100 | Agni | 2500 | 357.14286 | 7 | 100 | 700 |
| 200 | Agni | 2500 | 357.14286 | 7 | 100 | 700 |
| 200 | Vayu | 2500 | 357.14286 | 7 | 100 | 700 |
| 300 | Vayu | 2500 | 357.14286 | 7 | 100 | 700 |
| 500 | Vayu | 2500 | 357.14286 | 7 | 100 | 700 |
| 500 | Dharti | 2500 | 357.14286 | 7 | 100 | 700 |
| 700 | Dharti | 2500 | 357.14286 | 7 | 100 | 700 |

84

**NOTE**: Above we have used: “**ROWS** BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING”

which will give a SINGLE output based on all INPUT VaSluQLeBsy/PRAishRaTbIhTMIOisNhra(if used)

SELECT new\_id,

**RANKING FUNCTION**

**Example**

ROW\_NUMBER() OVER(ORDER BY new\_id) AS "ROW\_NUMBER",

RANK() OVER(ORDER BY new\_id) AS "RANK",

DENSE\_RANK() OVER(ORDER BY new\_id) AS "DENSE\_RANK", PERCENT\_RANK() OVER(ORDER BY new\_id) AS "PERCENT\_RANK"

FROM test\_data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **new\_id** | **ROW\_NUMBER** | **RANK** | **DENSE\_RANK** | **PERCENT\_RANK** |
| 100 | 1 | 1 | 1 | 0 |
| 200 | 2 | 2 | 2 | 0.166 |
| 200 | 3 | 2 | 2 | 0.166 |
| 300 | 4 | 4 | 3 | 0.5 |
| 500 | 5 | 5 | 4 | 0.666 |
| 500 | 6 | 5 | 4 | 0.666 |
| 700 | 7 | 7 | 5 | 1 |

85

**SELECT** new\_id,

**ANALYTIC FUNCTION**

**Example**

**FIRST\_VALUE**(new\_id) **OVER**( ORDER BY new\_id) AS "FIRST\_VALUE", **LAST\_VALUE**(new\_id) **OVER**( ORDER BY new\_id) AS "LAST\_VALUE", **LEAD**(new\_id) **OVER**( ORDER BY new\_id) AS "LEAD",---it will escape first leading value

**LAG**(new\_id) **OVER**( ORDER BY new\_id) AS "LAG"—it will escape last value

**FROM** test\_data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **new\_id** | **FIRST\_VALUE** | **LAST\_VALUE** | **LEAD** | **LAG** | |
| 100 | 100 | 100 | 200 | null | |
| 200 | 100 | 200 | 200 | 100 | |
| 200 | 100 | 200 | 300 | 200 | |
| 300 | 100 | 300 | 500 | 200 | |
| 500 | 100 | 500 | 500 | 300 | |
| 500 | 100 | 500 | 700 | 500 | |
| 700 | 100 | 700 | null | 500 | |
| **NOTE**: If you just want the single last value from whole column, use: “**ROWS** BETWEEN  UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING” | | | | |  |

86



**Offset the LEAD and LAG values by 2 in the output columns ?**

**INPUT**

**OUTPUT**

|  |
| --- |
| **new\_id** |
| 100 |
| 200 |
| 200 |
| 300 |
| 500 |
| 500 |
| 700 |

|  |  |  |
| --- | --- | --- |
| **new\_id** | **LEAD** | **LAG** |
| 100 | 200 | NULL |
| 200 | 300 | NULL |
| 200 | 500 | 100 |
| 300 | 500 | 200 |
| 500 | 700 | 200 |
| 500 | NULL | 300 |
| 700 | NULL | 500 |

87

**SELECT** new\_id,

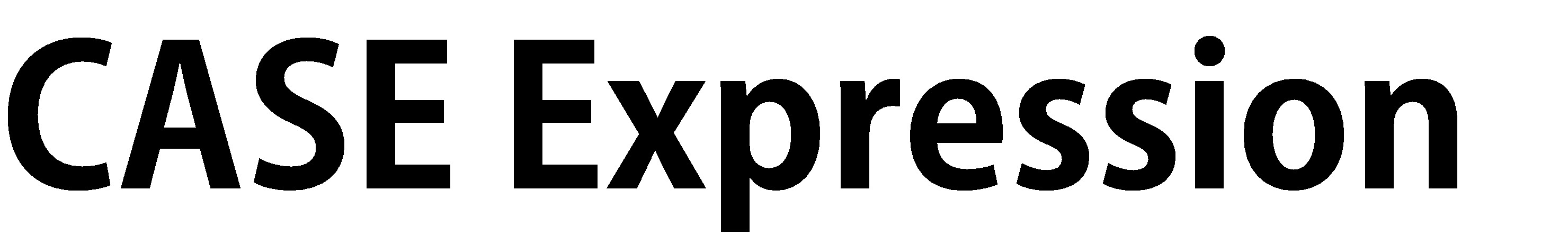
**ANALYTIC FUNCTION**

**Assignment**

**LEAD**(new\_id**, 2**) **OVER**( ORDER BY new\_id) AS "LEAD\_by2", **LAG**(new\_id**, 2**) **OVER**( ORDER BY new\_id) AS "LAG\_by2" **FROM** test\_data

|  |  |  |
| --- | --- | --- |
| **new\_id** | **LEAD\_by2** | **LAG\_by2** |
| 100 | 200 | null |
| 200 | 300 | null |
| 200 | 500 | 100 |
| 300 | 500 | 200 |
| 500 | 700 | 200 |
| 500 | null | 300 |
| 700 | null | 500 |

88

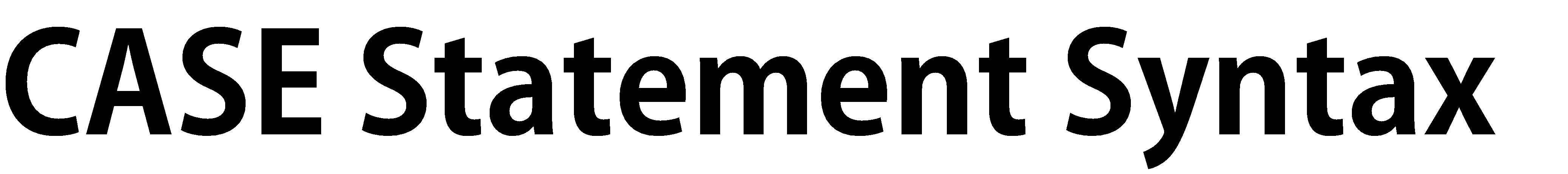


##### The CASE expression goes through conditions and returns a value when the first condition is met (like if-then-else statement). If no conditions are true, it returns the value in the ELSE clause.

* + If there is no ELSE part and no conditions are true, it returns NULL.

##### Also called CASE STATEMENT

90



* + - **Example:**
* **General CASE Syntax**

CASE

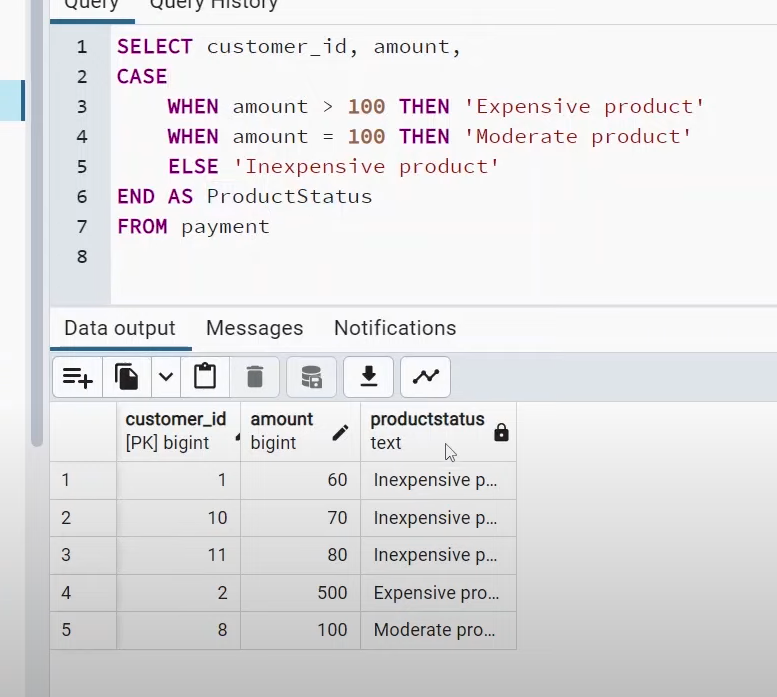
WHEN condition1 THEN result1 WHEN condition2 THEN result2 WHEN conditionN THEN resultN ELSE other\_result

END;

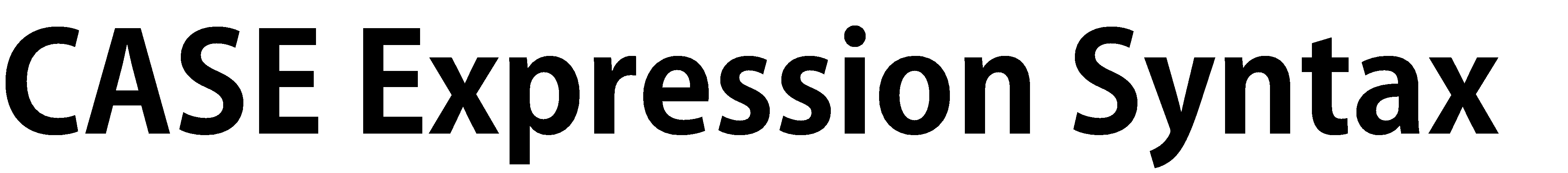
SELECT customer\_id, amount, CASE

WHEN amount > 100 THEN 'Expensive product' WHEN amount = 100 THEN 'Moderate product' ELSE 'Inexpensive product'

END AS ProductStatus FROM payment



92



* + - * **Example:**
* **CASE Expression Syntax**

CASE **Expression**

WHEN value1 THEN result1 WHEN value2 THEN result2 WHEN valueN THEN resultN ELSE other\_result

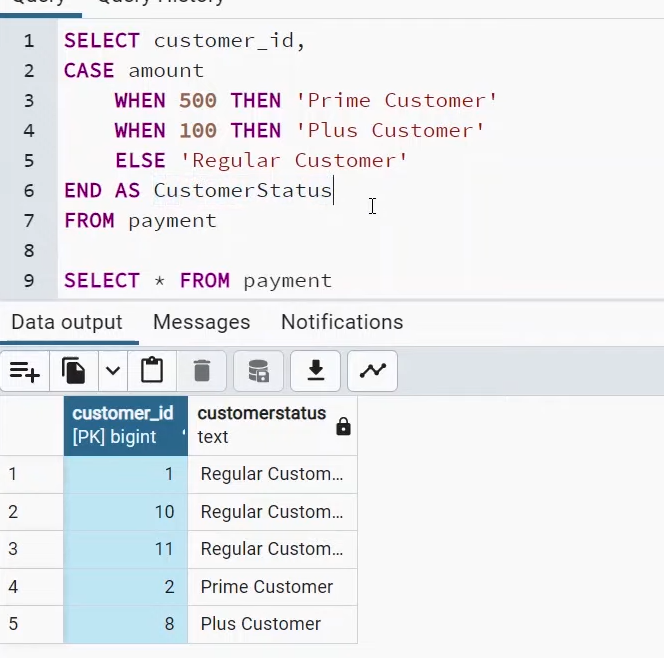
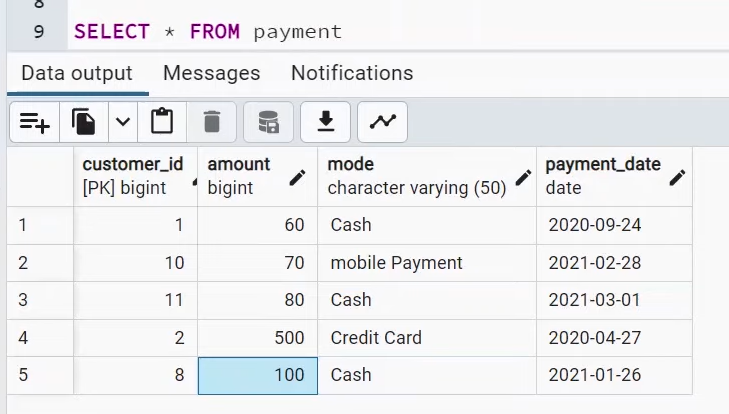
END;

SELECT customer\_id,

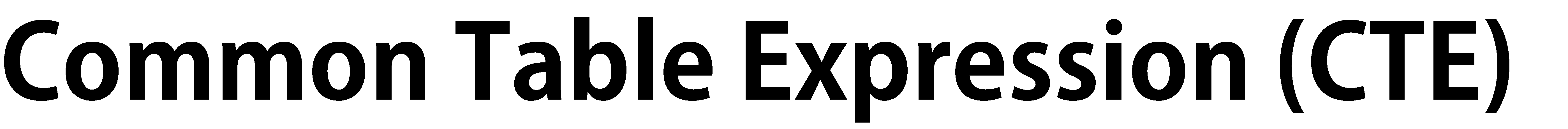
CASE **amount**

WHEN 500 THEN 'Prime Customer' WHEN 100 THEN 'Plus Customer' ELSE 'Regular Customer'

END AS CustomerStatus FROM payment



93

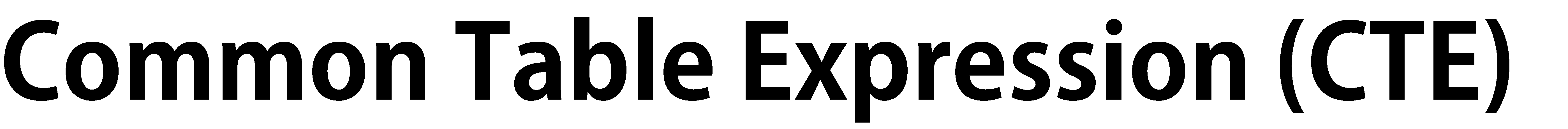


##### A common table expression, or CTE, is a temporary named result set created from a simple SELECT statement that can be used in a subsequent SELECT statement

* + We can define CTEs by adding a WITH clause directly before SELECT, INSERT, UPDATE, DELETE, or MERGE statement.

##### The **WITH** clause can include one or more CTEs separated by commas

##### (CTE is just like a function in programming language)



CTE query

The name of this CTE is my\_cte, and the CTE query is SELECT a,b,c FROM Table1. The CTE starts with the **WITH** keyword, after which you specify the name of your CTE, then the content of the query in parentheses. The main query comes after the closing parenthesis and refers to the CTE. Here, the main query (also known as the outer query) is SELECT a,c FROM my\_cte

* **Syntax**

**WITH my\_cte AS (**

SELECT a,b,c

FROM Table1 **)**

SELECT a,c

FROM **my\_cte**

Main query

97

**1. Example EASY**

WITH my\_cte AS (

SELECT \*, AVG(amount) OVER(ORDER BY

p.customer\_id) AS "Average\_Price",

COUNT(address\_id) OVER(ORDER BY c.customer\_id) AS "Count"

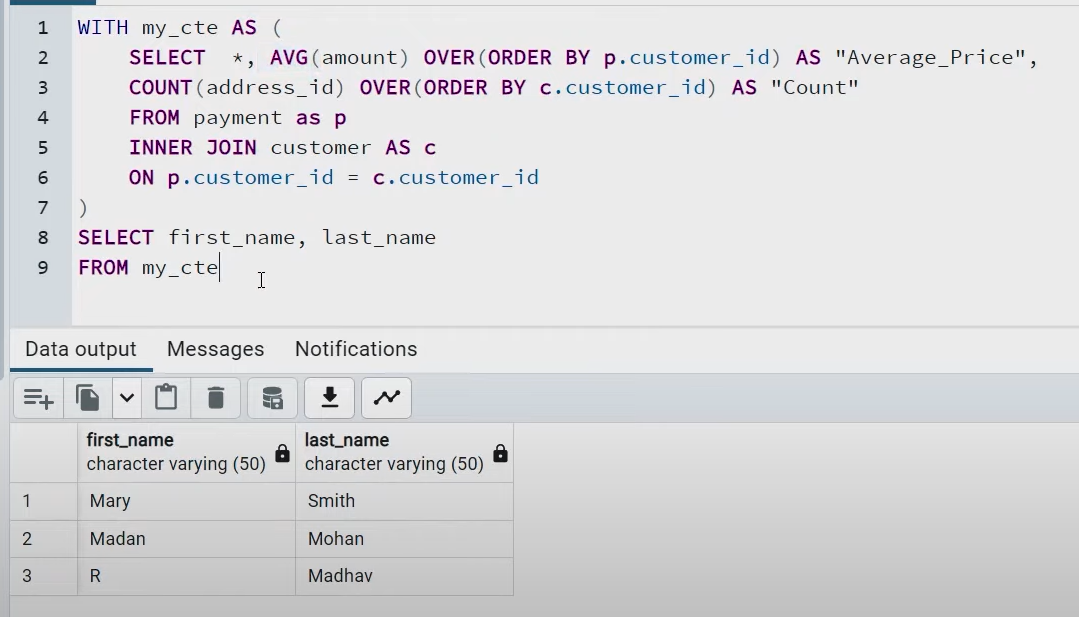
FROM payment as p

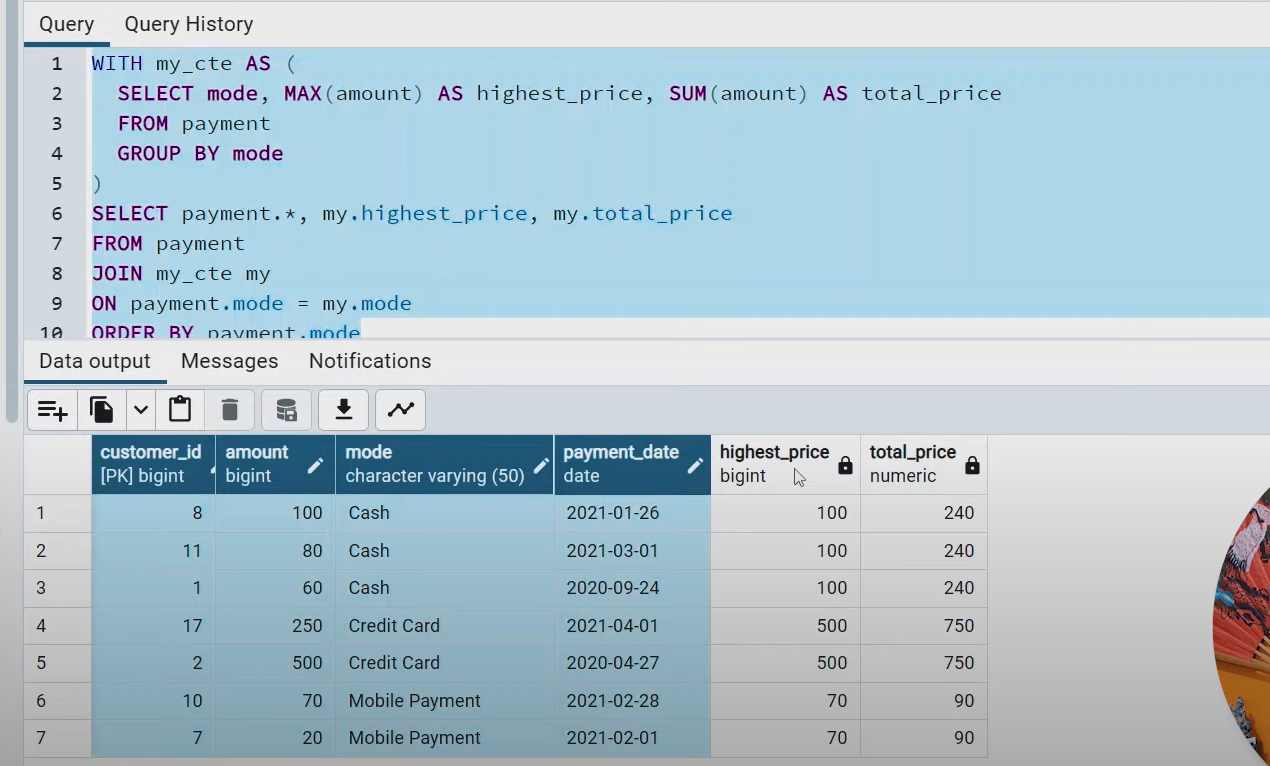
INNER JOIN customer AS c

ON p.customer\_id = c.customer\_id

)

SELECT first\_name, last\_name FROM my\_cte





**2. Example Advance**

WITH my\_cte AS (

SELECT mode, MAX(amount) AS highest\_price, SUM(amount) AS total\_price

FROM payment GROUP BY mode

)

SELECT payment.\*, my.highest\_price, my.total\_price

FROM payment JOIN my\_cte my

ON payment.mode = my.mode ORDER BY payment.mode

F

**1. Example Multiple CTEs**

WITH my\_cp AS (

SELECT \*, AVG(amount) OVER(ORDER BY p.customer\_id) AS "Average\_Price",

COUNT(address\_id) OVER(ORDER BY c.customer\_id) AS

"Count"

FROM payment as p INNER JOIN customer AS c

ON p.customer\_id = c.customer\_id

),

my\_ca AS (

SELECT \*

FROM customer as c INNER JOIN address AS a

ON a.address\_id = c.address\_id INNER JOIN country as cc

ON cc.city\_id = a.city\_id

)

SELECT cp.first\_name, cp.last\_name, ca.city, ca.country, cp.amount FROM my\_ca as ca , my\_cp as cp

Recursive CTE

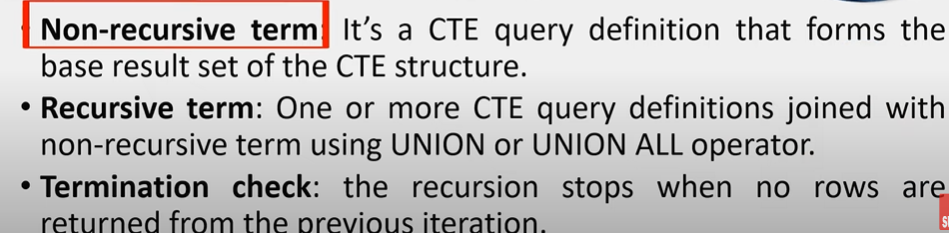
A recursive cte references itself , it return the results subset ,then it repeatedly(recursively) references itself ,and stops when it returns all the results.

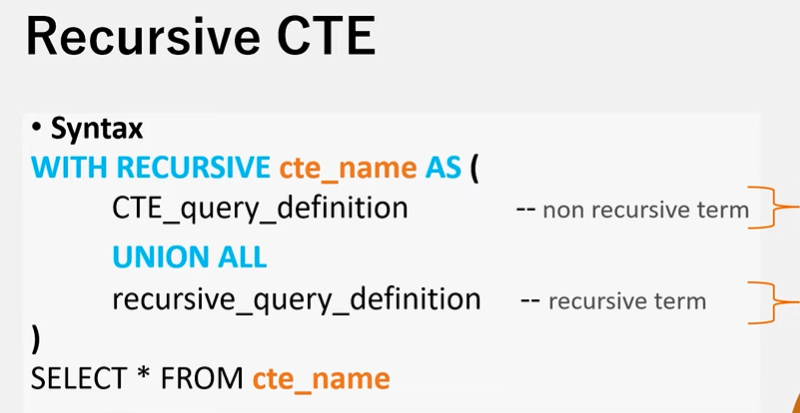
A recursive cte has three elements

Non recursive term

Recursive term

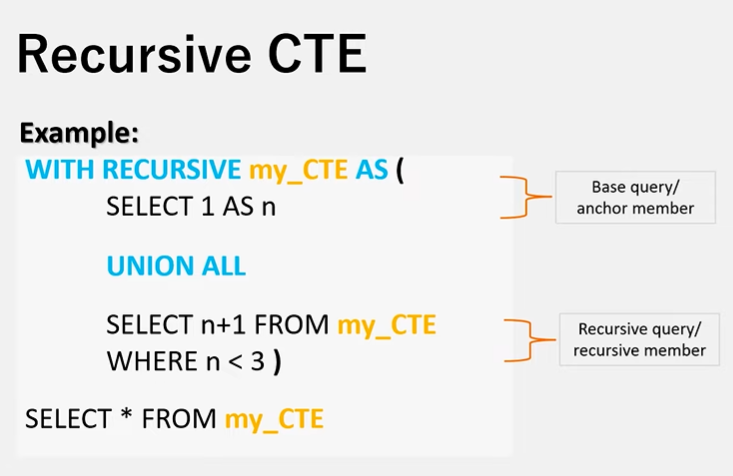
Termination check

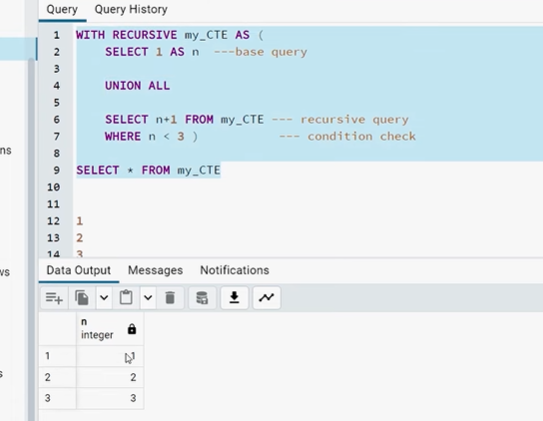


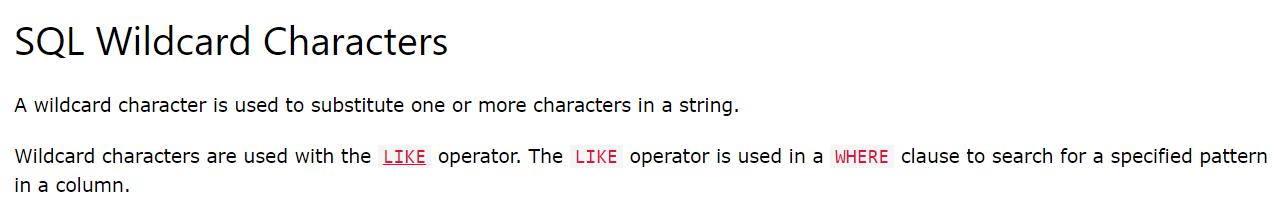


Recursive query

Base query





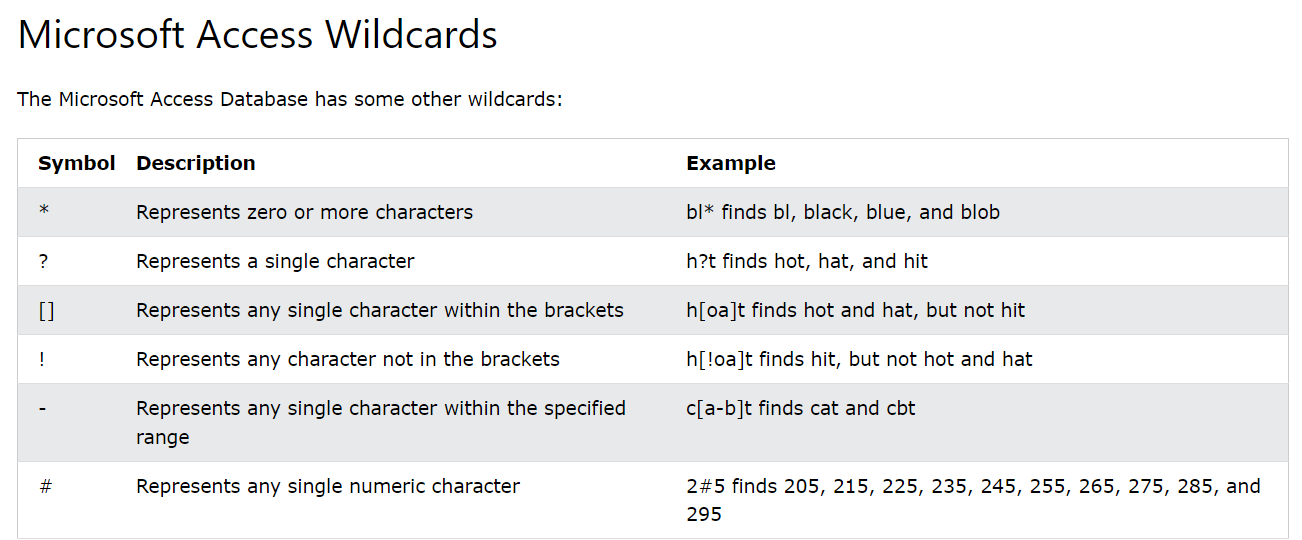


1.The % wildcard represents any number of characters, even zero characters.

**2.The \_ wildcard represents a single character.**

**It can be any character or number, but each \_ represents one, and only one, character**

SELECT \* FROM Customers  
WHERE City LIKE '\_ondon';

****