**Dbpass** Vitcc$3499

**Section 1 : Course Introduction**

What are Databases?

Databases are systems that allow users to store and organize data

Why use Databases?

They are useful when dealing with large amounts of data

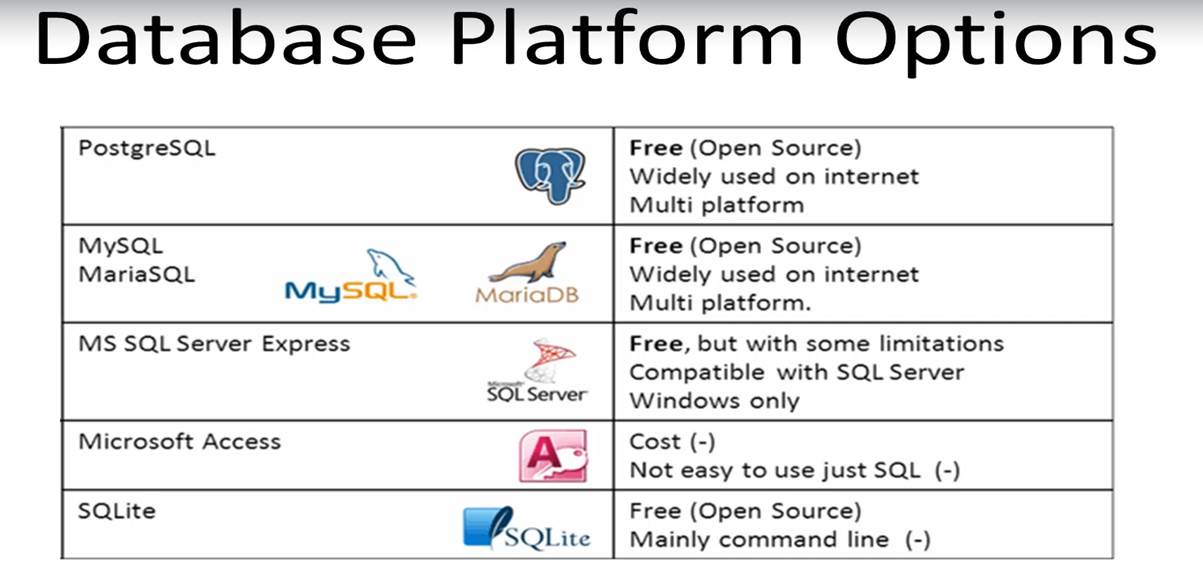
**From Spreadsheets to Databases**

|  |  |
| --- | --- |
| **Spreadsheets** | **Databases** |
| One time analysis | Data Integrity |
| Quickly need to chart something out | Can handle massive amounts of data |
| Reasonable data set size | Quickly combine different datasets |
| Ability for untrained people to work with data | Automate steps for re-use |
|  | Can support data for websites and applications |

Tabs = Tables

Rows = Rows

Columns = Columns



SQL is the programming language used to communicate with our Database



**PostgreSQL** - SQL Engine that stores data and read queries and returns information

**PgAdmin** - Graphical User Interface for connecting with PostgreSQL

**Section 2: SQL Statement Fundamentals:**

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

**Ex:** SELECT column\_name FROM Table;

**Ex:** SELECT c1,c2 FROM Table;

**Ex:** SELECT \* FROM Table;

* F5 is the shortcut for running the query
* The keywords Select or from are case insensitive(we can use both normal and capital letters)
* ; is to end the sql query for readability
* SELECT first\_name, last\_name, email FROM customer;(they may not give results in order)
* Sometimes a table contains a column that has duplicate values, and you find yourself in a situation where you only want to list the unique, distinct values. The **DISTINCT** keyword can be sued to return only the distinct values in a column

**Ex:** SELECT DISTINCT column FROM table;

**Ex:** SELECT DISTINCT(column) FROM table;

* The COUNT function returns the number of input rows that match a specific condition of a query. We can apply COUNT on a specific column or just pass COUNT(\*), we will soon see this should return the same result.

**Ex:** SELECT COUNT(name) FROM table;

**Ex:** SELECT COUNT(DISTINCT(name)) FROM table;

**Ex:** SELECT COUNT(\*) FROM payment;

**Result:** COUNT(bigint)

14596

* The **WHERE** statement allows us to specify conditions or filter on columns for the rows to be returned.

**Ex:** SELECT column1,column2 FROM table WHERE conditions;

Comparison Operators : =,>,<,>=,<=,<> or !=

Logical Operators : AND, OR , NOT

**Ex:** SELECT name FROM table WHERE name = 'David';

**Ex:** SELECT name FROM table WHERE name = 'David' AND choice = 'Red';

**Ex:** SELECT title FROM film WHERE rental\_rate>4 AND replacement\_cost >=19.99;

**Ex:** SELECT title FROM film WHERE rental\_rate>4 AND replacement\_cost >=19.99 AND rating = 'R';

**Ex:** SELECT title FROM film WHERE rating ='R' OR rating=' PG-13';

**Ex:** SELECT count(\*) FROM film WHERE rental\_rate>4 AND replacement\_cost >=19.99 AND rating = 'R';

* Use single quotes when comparing strings
* You may have noticed PostgreSQL sometimes returns the same request query results in a different order. You can use ORDER BY to sort rows based on a column value ,in either ascending or descending order.
* If you leave it blank, ORDER BY uses ASC by default.

**Ex:** SELECT c1,c2 FROM table ORDER BY c1 ASC/DESC;

* You an also ORDER BY multiple columns
* This makes sense when one column has duplicate entries.
* This makes sense when one column has duplicate entries.

**Ex**: SELECT company,name,sales FROM table ORDER BY company,sales;

**Ex**: SELECT \* FROM customer ORDER BY first\_name;

**Ex**: SELECT \* FROM customer ORDER BY first\_name DESC;

**Ex:** SELECT FIRST\_NAME,LAST\_NAME from customer ORDER BY store\_id DESC,first\_name ASC;

* The LIMIT command allows us to limit the number of rowa 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.

**Ex:** SELECT \* FROM payment WHERE amount!=0.0 ORDER BY payment\_date DESC LIMIT 5;

* The BETWEEN operator can be used to match value against a range of values: like values BETWEEN low AND high.(value>=low and value<=high)
* You can also combine BETWEEN with the NOT logical operator: like value NOT BETWEEN low AND high(value<low OR value>high)
* The BETWEEN operator can also be used with dates. Note that you need to format dates in the ISO 8601 standard format, which is YYYY-MM-DD(date BETWEEN '2007-01-01' AND '2007-02-01')
* When using BETWEEN operators with dates that also include timestamp information ,pay careful attention to using BETWEEN versus <=,>= comparison operators, due to the fact that a datetime starts at 0:00.

Later on we will study more specific methods for datetime information types.

**Ex:** SELECT \* FROM payment WHERE payment\_date BETWEEN '2007-02-01' AND '2007-02-15'

* In certain cases you want to check for multiple possible value options,for multiple value options,for example, if a user's name shows up IN a list of known names.
* We can use the IN operator to create a condition that checks to see if a value in included in a list of multiple options.

**Ex:** SELECT color FROM table WHERE color IN ('red','blue');

**Ex:** SELECT color FROM table WHERE color NOT IN ('red','blue');

* We've already been able to perform direct comparisons against strings ,such as
  + Where first\_name='John'

But what if we want to match against a general pattern in a string?

* All emails ending with '@gmail.com'
* All names that begin with an 'A'
* The LIKE operator 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
  + All names that begin with an 'A'
    - WHERE name LIKE 'A%'
  + All names that end with an 'a'
    - WHERE name LIKE '%a'

Notice that LIKE is case-sensitive, we can use ILIKE which is case-insensitive

**Ex:** SELECT \* FROM CUSTOMER WHERE first\_name ILIKE 'j%' AND last\_name ILIKE 'S%'

**Ex:** SELECT \* FROM CUSTOMER WHERE first\_name LIKE 'J%' AND last\_name NOT LIKE 'S%' ORDER BY last\_name

**Section 3: GROUP BY Statements**

* GROUP BY will allow us to aggregate data and apply functions to better understand how data is distributed per category.
* Section Overview
  + Aggregate Functions
  + GROUP BY - Part One -Theory
  + GROUP BY - Part Two - Implementaion
  + Challenge Tasks For GROUP BY
  + HAVING - filtering with a GROUP BY
  + Challenge Task for HAVING
* The main idea behind an aggregate function is to take multiple inputs and return a single output. Some of the Most Common Aggregate Functions:
  + AVG() - returns average value
  + COUNT() - returns number of values
  + MAX() - returns maximum value
  + MIN() - returns minimum value
  + SUM() - returns the sum of all values
* Aggregate function calls happen only in THE SELECT clause or the HAVING clause
* **Special Notes**
  + AVG() returns a floating point value many decimal places (e.g. 2.342418…)
    - You can use ROUND() to specify precision after the decimal.
  + COUNT() simply returns the number of rows ,which means by convention we just use COUNT(\*)

**Ex:**SELECT MAX(replacement\_cost),MIN(replacement\_cost) FROM film;

|  |  |
| --- | --- |
| Max | Min |
| 29.99 | 9.99 |

**Ex:** SELECT ROUND(AVG(replacement\_cost),2) FROM film;

In here 2 means precision like 1.232112312 will be converted to 1.23 due to precision

**Ex:** SELECT SUM(replacement\_cost) FROM film;

Sum

19984.00

* GROUP BY allows us to aggregate columns per some category

**Ex:** SELECT category\_col, AGG(data\_col) FROM table GROUP BY category\_col

* **Ex:** SELECT category\_col, AGG(data\_col) FROM table WHERE category\_col!='A' GROUP BY category\_col
* The GROUP BY clause must appear right after a FROM or WHERE statement
* In the SELECT statement ,columns must either have an aggregate function or be in the GROUP BY call

**Ex:** SELECT company,divison,SUM(sales) FROM finance\_table GROUP BY company,division

* WHERE statements should not refer to the aggregation result, later on we will learn to use HAVING to filter on those results

**Ex:** SELECT company,division,SUM(sales) FROM finance\_table WHERE division IN ('marketing','transport') GROUP BY company,division

* If you want to sort results based on the aggregate, make sure to reference the entire function

**Ex:** SELECT company,SUM(sales) FROM finance\_table GROUP BY company ORDER BY SUM(sales) LIMIT 5;

**Ex:** SELECT DATE(payment\_date),SUM(amount) FROM payment GROUP BY DATE(payment\_date) ORDER BY SUM(amount)

* DATE(payment\_date) will extract date.
* The HAVING clause allows us to filter after an aggregation has already taken place
* HAVING allows us to use the aggregate result as a filter along with a GROUP BY

**Ex:** SELECT comapany ,SUM(sales) FROM finance\_table WHERE company!='Google' GROUP BY company HAVING SUM(sales) >1000

**Section 5 : JOINS**

* Joins will allow us to combine information from multiple tables
* Section Overview
  + Creating an alias with the AS clause
  + Understanding different kinds of JOINs
    - INNER JOINS
    - OUTER JOINS
    - FULL JOINS
    - UNIONS
  + Challenge Tasks
* The AS clause which allows us to create an alias for a column or result
* **Ex:** SELECT column AS new\_name FROM table
* **Ex:** SELECT SUM(column) AS new\_name FROM table
* **Ex:** SELECT amount AS rental\_price FROM payment
* **Ex:** SELECT SUM(amount) AS net\_revenue FROM payment
* The AS operator gets executed at the very end of a query, meaning that we cannot use the ALIAS inside a WHERE or HAVING clause.
* **Ex:** SELECT customer\_id,amount AS new\_name FROM payment WHERE amount>2(**We can't use new\_name in the where clause**)
* **Ex:** SELECT customer\_id,SUM(amount) AS total\_spent FROM payment GROUP BY customer\_id HAVING SUM(amount)>100(**We can't use total\_spent in the HAVING clause**)
* JOINS allow us to combine multiple tables together.
* The main reason for the different JOIN types is to decide how to deal with information only present in one of the joined tables
* Machine generated alternative text:
  An INNER JOIN will 
  result with the set of 
  records that match in both tables. 
  REGISTRATIONS 
  LOGINS 
  reg_id 
  log_id 
  name 
  name 
  1 
  2 
  3 
  4 
  Andrew 
  Bob 
  Charlie 
  David 
  1 
  2 
  3 
  4 
  Xavier 
  Andrew 
  Yolanda 
  Bob 

Machine generated alternative text:
SELECT * FROM TableA 
INNER JOIN TableB 
ON TableA.col match = TableB.coI 
match 
TableA 
Table B 

Machine generated alternative text:
SQL 
, SELECT * FROM Registrations 
INNER JOIN Logins 
ON Registrations.name = Logins. 
REGISTRATIONS 
reg_id 
name 
RESULTS 
name 
LOGINS 
log_id 
name 
reg _ 
2 
id 
2 
3 
4 
Andrew 
Bob 
Charlie 
David 
name 
Andrew 
Bob 
log_id 
2 
4 
name 
Andrew 
Bob 
2 
3 
4 
Xavier 
Andrew 
Yolanda 
Bob 

Machine generated alternative text:
SELECT reg_id,Logins.name,Iog_id 
FROM Registrations 
INNER JOIN Logins 
ON Registrations.name = Logins.name 
reg_id 
2 
RESULTS 
name 
Andrew 
Bob 
log_id 
2 
4 

* Remember that table order won't matter in an INNER JOIN
* Also if you see JOIN without the INNER, PostgreSQL will treat it as an INNER JOIN
* They are few different types of OUTER JOINS. They will allow us to specify how to deal with values only present in one of the tables being joined.
* Types of Outer Joins
  + FULL OUTER JOIN
    - Clarifying WHERE null
  + LEFT OUTER JOIN
    - Clarifying WHERE null
  + RIGHT OUTER JOIN
    - Clarifying WHERE null
* Machine generated alternative text:
  SELECT * FROM TableA 
  FULL OUTER JOIN TableB 
  ON TableA.col match = TableB.coI 
  match 
  Table A 
  Table B 

* Machine generated alternative text:
  O 
  SQL 
  SELECT * FROM Registrations FULL OUTER JOIN Logins 
  ON Registrations.name = Logins.name 
  RESULTS 
  reg_id 
  REGISTRATIONS 
  reg_id 
  name 
  2 
  3 
  4 
  2 
  Andrew 
  3 
  Bob 
  4 
  Charlie 
  null 
  David 
  null 
  name 
  Andrew 
  Bob 
  Charlie 
  David 
  null 
  null 
  log_id 
  2 
  4 
  3 
  name 
  LOGINS 
  Andrew 
  log_id 
  name 
  Bob 
  Xavier 
  null 
  2 
  Andrew 
  null 
  Yolanda 
  3 
  Xavier 
  Bob 
  4 
  Yolanda 

Machine generated alternative text:
SELECT * FROM Registrations FULL OUTER JOIN Logins 
ON Registrations.name = Logins.name 
WHERE Registrations.reg_id IS null OR 
Logins.log_id IS null 
REGISTRATIONS 
reg_id 
name 
RESULTS 
2 
3 
4 
Andrew 
Bob 
Charlie 
David 
reg _ i 
3 
4 
null 
null 
name 
Charlie 
David 
null 
log_id 
3 
name 
null 
null 
Xavier 
Yolanda 
LOGINS 
log_id 
name 
Xavier 
2 
Andrew 
Yolanda 
3 
Bob 
4 

Machine generated alternative text:
SQL 
SELECT * FROM Registrations FULL OUTER 
JOIN Logins 
ON Registrations.name = Logins.name 
WHERE Registrations.reg_id IS null OR 
Logins.log_id IS null 
Registrations 
PIERIAN C DATA 
Logins 

* A LEFT OUTER JOIN results in the set of records that are in the left table, if there is no match with the right table ,the results are null
* Machine generated alternative text:
  SELECT * FROM TableA 
  LEFT OUTER JOIN TableB 
  ON TableA.coI match = TableB.coI 
  match 
  Table A 
  ' DATA 
  Table B 
  'IERIAN 
* We can call left outer join or left join(here order matters how we select from the table)
* Machine generated alternative text:
  , SELECT * FROM Registrations 
  LEFT OUTER JOIN Logins 
  ON Registrations.name = Logins. 
  REGISTRATIONS 
  reg_id 
  name 
  RESULTS 
  name 
  LOGINS 
  log_id 
  name 
  2 
  3 
  4 
  reg_id 
  Andrew 
  Bob 
  2 
  Charlie 
  3 
  David 
  4 
  name 
  Andrew 
  Bob 
  Charlie 
  David 
  log_id 
  2 
  4 
  null 
  null 
  name 
  Andrew 
  Bob 
  null 
  null 
  2 
  3 
  4 
  Xavier 
  Andrew 
  Yolanda 
  Bob 
  PIERIAN C DATA 

* Machine generated alternative text:
  SELECT * FROM Registrations 
  LEFT OUTER JOIN Logins 
  ON Registrations.name = Logins.name 
  WHERE Logins.log_id IS null 
  REGISTRATIONS 
  reg_id 
  name 
  LOGINS 
  log_id 
  name 
  RESULTS 
  2 
  3 
  4 
  Andrew 
  Bob 
  reg _ 
  Charlie 
  3 
  David 
  4 
  id 
  name 
  Charlie 
  David 
  log_id 
  null 
  null 
  name 
  null 
  null 
  2 
  3 
  4 
  Xavier 
  Andrew 
  Yolanda 
  Bob 
  PIERIAN C DATA 

* A **RIGHT JOIN** is essentially the same as a LEFT JOIN,except the tables are switched.This would be the same as switching the table order in a LEFT OUTER JOIN.

Machine generated alternative text:
SELECT * FROM TableA 
RICHT OUTER JOIN TableB 
ON TableA.col match = TableB.coI 
match 
Table A 
' DATA 
Table B 
PIERIAN 

* RIGHT JOIN and RIGHT OUTER JOIN are same.
* The UNION operator is used to combine the result-set of two or more SELECT statements
* It basically serves to directly concatenate two results together,essentially "pasting" them together.
* **Ex:** SELECT \*FROM Sales2021\_Q1

UNION

SELECT \* FROM Sales2021\_Q2

**Section 6: Advanced SQL Topics**

* Section Overview
  + Timestamps and EXTRACT
  + Math Functions
  + String Functions
  + Sub-query
  + Self-Join
* Types of Date and Time
  + TIME - Contains only time
  + DATE - Contains only date
  + TIMESTAMP - Contains date and time
  + TIMESTAMPZ - Contains date, time and timezone
* Lets explre functions and operations related to these specific data types
  + TIMEZONE
  + NOW
  + TIMEOFDAY
  + CURRENT\_TIME
  + CURRENT\_DATE
* SHOW TIMEZONE will give you "Asia/Calcutta"
* SELECT NOW() will give you time stamp "2024-09-19 17:25:46.504347+05:30"
* SELECT TIMEOFDAY() will give you timestamp "Thu Sep 19 17:27:25.176430 2024 IST" in more stringy understandable way
* SELECT CURRENT\_TIME will give you "17:28:33.858601+05:30"
* SELECT CURRENT\_DATE will give you "2024-09-19"
* Lets explore extracting information from a time based data type using:
  + EXTRACT()
  + AGE()
  + TO\_CHAR()
* EXTRACT()
  + Allows you to "extract" or obtain a sub-component of a date value
    - YEAR
    - MONTH
    - DAY
    - WEEK
    - QUARTER
* EXTRACT()
  + Allows you to "extract" or obtain a sub-component of a date value
    - EXTRACT(YEAR FROM date\_col)

**Ex:** SELECT EXTRACT(YEAR FROM payment\_date) FROM payment

* AGE()
  + Compares the timestamp with current timestamp.
  + Calculates and returns the current age given a timestamp
  + Usage
    - AGE(date\_col)
  + Returns
    - 13 years 1 mon 5 days 01:34:13.003423

**Ex:** SELECT AGE(payment\_date) FROM payment

* TO\_CHAR()
  + General function to convert data types to text
  + Useful for timestamp formatting
  + Usage
    - TO\_CHAR(date\_col,'mm-dd-yyyy')

**Ex:** SELECT TO\_CHAR(payment\_date,'dd-mm-YYYY') FROM payment

* SELECT COUNT(\*) FROM payment WHERE EXTRACT(dow FROM payment\_date)=1

Here dow returns a number from 1 to 7 representing the day of the week of a date, where Sunday=1, Monday=2, and so on.

**Mathematical Functions and Operators**

**Ex:** SELECT salary + bonus AS total\_compensation FROM employees;

**Ex:** SELECT total\_sales - expenses AS profit FROM financials;

**Ex:** SELECT LENGTH(first\_name) FROM customer;

**String Functions and Operators**

**Ex:** SELECT first\_name || ' ' || last\_name AS full\_name FROM customer

**Ex:** SELECT upper(first\_name) || ' ' || upper(last\_name) AS full\_name FROM customer

**Ex:** SELECT LOWER(LEFT(first\_name,1)) || LOWER(last\_name) || '@gmail.com' AS custom\_email FROM customer;

**SubQuery**

* A sub query allows you to construct complex queries, essentially performing a query on the results of another query
* The syntax is straight forward and involves two SELECT statements

**Ex:** This is where a subquery can help us get the result in a "single" query Request

SELECT student,grade FROM test\_scores WHERE grade > (SELECT AVG(grade) FROM test\_scores)

* The subquery is performed first since it is inside the parenthesis
* We can also use the IN operator in conjunction with a subquery to check against multiple results returned

**Ex:** SELECT student,grade FROM test\_scores WHERE student IN (('Zach','Chris','Karissa'))

* The EXISTS operator is used to test for existence of rows in a subquery
* Typically a subquery is passed in the EXISTS() function to check if any rows are returned with the subquery.
* **Ex:** SELECT column\_namw FROM table\_name WHERE EXISTS (SELECT column\_name FROM table\_name WHERE condition);
* A self-join is a query in which a table is joined to itself
* Self-joins are useful for comparing values in a column of rows within the same table.
* The self join can be viewed as a join of two copies of the same table

**Ex:** SELECT tableA.col,tableB.col FROM table AS tableA JOIN table AS tableB ON tableA.some\_col = tableB.other\_col