

Database:

- A database is organised collection of data.
- Databases are systems that allows user to store and organize data
- They are useful when dealing with a large amount of data
- Databases are useful for
 - Data integrity – hard to manipulate data
 - Handle massive amount of data
 - Combine different datasets
 - Automate the process/steps for re use
 - Can support data for websites or applications
- Databases can be thought as of being made from different tables corresponding to different sheets of the Excel. Then, those tables are made up of rows and columns.
 - Columns represent the different heads or characteristics of the data while the rows represent a particular dataset or instance corresponding to the characteristics of the spreadsheet.
 - Columns must have distinct column names in databases that is why they are more rigid than Excel.
 - Broadly, within databases, tables are organized in schemas.
- SQL (Structured Query Language) is a programming language used to communicate with our database while PostgreSQL and MYSQL are softwares that use SQL.
- Benefits of SQL:
 - No need to copy data
 - Easy to understand/ easier syntax
 - Compared to spreadsheet tools, data analysis done in SQL is easy to audit and replicate. For analysts, this means no more looking for the cell with the typo in the formula.

Points to note: -

- Don't use * very often as it will prolong the time taken to answer our query as it will search through all the database. For small databases that is not a problem but for a company level database command will take too long to respond.
- Column names were separated by a comma in the query. Whenever you select multiple columns, they must be separated by commas, but you should not include a comma after the last column name.
- By convention keep the keywords capitalize so that code is easy to understand for a later point of time or for others.
- SQL treats one space, multiple spaces, or a line break as being the same thing.
- All of the columns in the tables are generally named in lower case, and use underscores instead of spaces. Avoid putting spaces in column names because it's annoying to deal with spaces in SQL—if you want to have spaces in column names, you need to always refer to those columns in **double quotes**. The table name itself also uses underscores instead of spaces.

- In general, putting double quotes around a word or phrase will indicate that you are referring to that column name.
- If you'd like your results to look a bit more presentable, you can rename columns to include spaces using double quotes and AS as an alias. Without the double quotes, that query would read 'West' and 'Region' as separate objects and would return an error.
 - `SELECT west AS "West Region" FROM tutorial.us_housing_units`
- Start a new line for new keyword.

- **SELECT WHERE Operator**

- If you write a WHERE clause that filters based on values in one column, you'll limit the results in all columns to rows that satisfy the condition. The idea is that each row is one data point or observation, and all the information contained in that row belongs together.
- WHERE statement is generally used with comparison operators and logical operators.
- Comparison operators are

OPERATOR	DESCRIPTION
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal

- These comparison operators make the most sense when applied to numerical columns.
- All of the above operators work on non-numerical data as well. = and != make perfect sense—they allow you to select rows that match or don't match any value, respectively. If you're using an operator with values that are non-numeric, you need to put the value in single quotes: 'value'.
- **SQL uses single quotes to reference column values.**
- You can use >, <, and the rest of the comparison operators on non-numeric columns as well—they filter based on alphabetical order.
- The way SQL treats alphabetical ordering is a little bit tricky. We may have noticed in the query that selecting month name > 'J' will yield only rows in which month name starts with "j" or later in the alphabet. But, "January is included in the results—shouldn't we have to use month name >= 'J' to make that happen?" SQL considers 'Ja' to be greater than 'J' because it has

an extra letter. It's worth noting that most dictionaries would list 'Ja' after 'J' as well.

Arithmetic in SQL

- In SQL you can only perform arithmetic across columns on values in a given row. To clarify, you can only add values in multiple columns from the same row together using +. But, if you want to add values across multiple rows, you'll need to use aggregate functions.
 - `SELECT year, month, west, south, west + south - 4 * year AS nonsense_column FROM tutorial.us_housing_unit`
- As in Excel, you can use parentheses to manage the order of operations.

SQL Logical operators

- You'll likely also want to filter data using several conditions—possibly more often than you'll want to filter by only one condition. Logical operators allow you to use multiple comparison operators in one query.
 - LIKE allows you to match similar values, instead of exact values.
 - IN allows you to specify a list of values you'd like to include.
 - BETWEEN allows you to select only rows within a certain range.
 - IS NULL allows you to select rows that contain no data in a given column.
 - AND allows you to select only rows that satisfy two conditions.
 - OR allows you to select rows that satisfy either of two conditions.
 - NOT allows you to select rows that do not match a certain condition
-
- NOT logical operator in SQL that you can put before any conditional statement to select rows for which that statement is false.
 - `SELECT * FROM tutorial.billboard_top_100_year_end WHERE year = 2013 AND year_rank NOT BETWEEN 2 AND 3`
 - In the above case, you can see that results for which year_rank is equal to 2 or 3 are not included.
 - NOT is commonly used with LIKE.
 - `SELECT * FROM tutorial.billboard_top_100_year_end WHERE year = 2013 AND "group" NOT ILIKE '%macklemore%'`

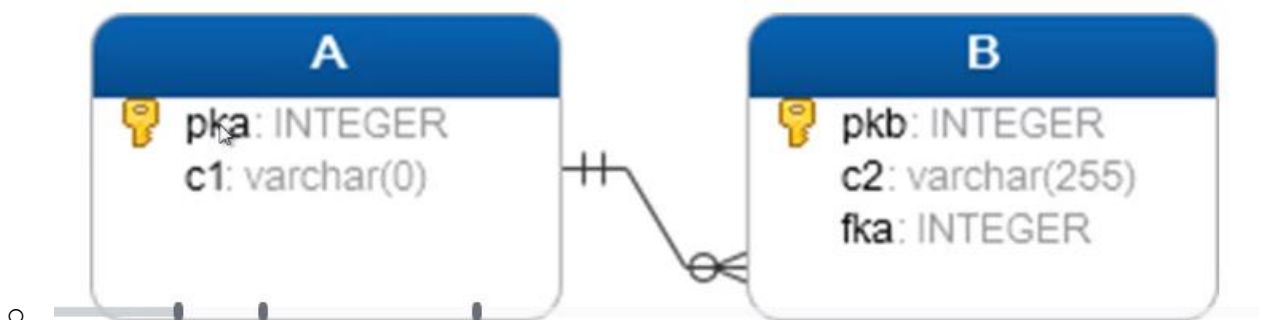
- NOT is also frequently used to identify non-null rows, but the syntax is somewhat special—you need to include IS beforehand.
 - `SELECT * FROM tutorial.billboard_top_100_year_end WHERE year = 2013 AND artist IS NOT NULL`
- DISTINCT function is used to find the distinct value in a column.
 - If we include two (or more) columns in a SELECT DISTINCT clause, your results will contain all of the unique pairs of those two columns
 - We only need to include DISTINCT once in your SELECT clause—you do not need to add it for each column name
- COUNT function returns the number of input rows that match a specific condition of a query.
 - `SELECT COUNT (*) FROM table;`
 - `SELECT COUNT (column) FROM table;`
 - `SELECT COUNT (DISTINCT column) FROM table;`
 - It does not consider the NULL value in the column.
 - This can also be used for columns containing non-numerical values.
- LIMIT function allows us to limit the number of rows we get back after a query
 - Useful when we want all the columns but not all the rows to check all the columns.
 - Many analysts use limits as a simple way to keep their queries from taking too long to return.
 - Used at the end of the query
 - `SELECT * FROM payment LIMIT 5;`
- ORDER BY clause will return the queries by a specific order either in ascending or descending order based on the criteria specified.
 - Without this clause it SELECT returns the queries in the order data is entered.
 - `SELECT column_1, column_2 FROM table ORDER BY column_1 DESC/ASC;`
 - If we want to sort by multiple columns, we use the comma to separate multiple columns.
 - If we leave blank ORDER BY will use ASC by default.
 - `SELECT column_1 FROM table ORDER BY column_2 ASC;` is a valid query in PostgreSQL but others like MYSQL may not allow this.
 - You can also order by multiple columns. This is particularly useful if your data falls into categories and you'd like to organize rows by date, for example, but keep all of the results within a given category together.
- BETWEEN statement is used to match value between/against a range of values.
 - Generally used with AND operator.
 - `VALUE BETWEEN low AND high;` which includes both low and high value
 - `VALUE >= low AND VALUE <= high;`
 - To check a value not in the range we can check NOT BETWEEN operator like
 - `VALUE NOT BETWEEN low AND HIGH;` which is equivalent of saying
 - `VALUE < low AND VALUE > high;`
 - `SELECT * FROM payment WHERE amount NOT BETWEEN 8 AND 9;`

- Commenting in SQL
 - We can use-- (two dashes) to comment out everything to the right of them on a given line
 - We can also leave comments across multiple lines using /* to begin the comment and */ to close it
- IN operator is used with WHERE to check if a value matches any value in a list of values
 - VALUE IN (value1, value2,.....)
 - Returns TRUE /FALSE
 - Not just limited to numbers and strings
 - As with comparison operators, you can use non-numerical values, but they need to go inside single quotes. Regardless of the data type, the values in the list must be separated by commas.
 - NOT IN can also be used.
 - SELECT customer_id,rental_id,return_date FROM rental WHERE customer_id IN (1,2);
 - SELECT customer_id, rental_id, return_date FROM rental WHERE customer_id NOT IN (1,2);
 - Helps in getting or searching for specific value.
 - IN process much faster than couple of the OR statement.
 - Use of subquery in the following query:
 - SELECT first_name, last_name, customer_id
FROM customer
WHERE customer_id IN (SELECT customer_id FROM rental
WHERE CAST (return_date AS DATE) = '2005-05-30');
- LIKE operator is used for pattern matching where we remember a part of the data
 - SELECT first_name, last_name FROM customer WHERE first_name LIKE 'Jen%';
 - JEN and Jen will be considered as two different strings
 - % and _ are called wildcard characters where % is used to match any sequence of characters and _ is used for matching any single character
 - % sign referred or represents a pattern
 - Wildcard characters can also be used at the beginning or at both ends.
 - SELECT first_name, last_name FROM customer WHERE first_name LIKE '_her%'
 - PostgreSQL also have ILIKE operator which is not case sensitive
- Aggregate Functions are generally used with GROUP BY operator
 - This includes MAX, MIN, COUNT, SUM, AVG
 - <https://www.postgresql.org/docs/9.5/functions-aggregate.html>
 - An important thing to remember: aggregators only aggregate vertically. If you want to perform a calculation across rows, you would do this with simple arithmetic.
- AVG- gives average of the column we asked for
 - SELECT AVG (amount) FROM payment;
 - It can only be used on numerical columns.

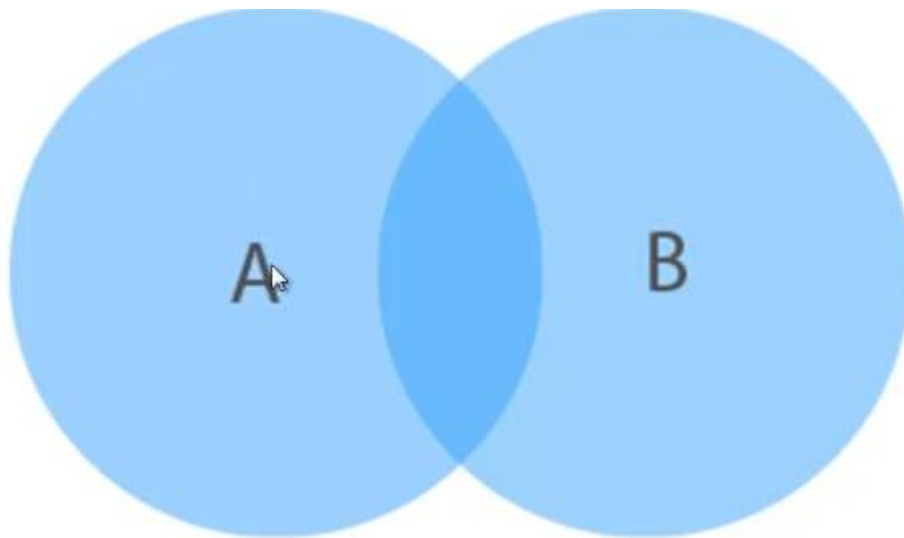
- It ignores nulls completely
 - To round it to two decimal places we can use
 - `SELECT ROUND (AVG (amount),2) FROM payment;`
- MIN-returns the minimum value of the column selected
 - `SELECT MIN (amount) FROM payment;`
 - They're similar to COUNT in that they can be used on non-numerical columns. Depending on the column type, MIN will return the lowest number, earliest date, or non-numerical value as close alphabetically to "A" as possible. As we might suspect, MAX does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to "Z."
- MAX-returns the maximum amount of the column selected
 - `SELECT MAX (amount) FROM payment;`
- SUM-returns the sum of the value of the column selected
 - `SELECT SUM (amount) FROM payment;`
 - Can be used only for numerical columns.
 - We don't need to worry as much about the presence of nulls with SUM as you would with COUNT, as SUM treats nulls as 0.
- IS NULL- logical operator in SQL that allows you to exclude rows with missing data from your results.
 - `WHERE artist = NULL` will not work because you can't perform arithmetic on null values.
- GROUP BY – divides the rows returned from the select statement into groups and for each group aggregate function can be applied.
 - GROUP BY allows we to separate data into groups, which can be aggregated independently of one another.
 - We can group by multiple columns, but we have to separate column names with commas—just as with ORDER BY.
 - `SELECT column_1, aggregate_function(column_2) FROM table_name GROUP BY column_1;`
 - So, all the data is grouped by distinct values of column_1
 - `SELECT column_1, column_2 FROM table_name GROUP BY column_1;`
 - `SELECT customer_id, FROM payment GROUP BY customer_id;`
 - Will show all the rows with distinct customer_id
 - `SELECT customer_id, SUM (amount) FROM payment GROUP BY customer_id`
 - Will show us the total amount paid by each customer_id
- HAVING-This clause is used in conjunction with GROUP BY clause to filter group rows that do not specify a given condition.
 - Same as WHERE clause only thing is that we are using it with a GROUP BY.
 - WHERE clause apply on individual rows before GROUP BY clause applies, whereas HAVING apply after the GROUP BY clause.

- SELECT column_1, aggregate_function(column_2) FROM table name GROUP BY column_1 HAVING condition;
- SELECT store_id, COUNT(customer_id) FROM customer GROUP BY store_id HAVING COUNT(customer_id) > 300
- SELECT rating, SUM(rental_rate) FROM film WHERE rating in ('R','G','PG') GROUP BY rating HAVING SUM(rental_rate) > 550;
- AS – allows us to rename columns or table selections with an alias.
 - SELECT payment_id AS my_payment_table FROM payment;
 - Will change the column name from payment_id to my_payment_table
 - Works not only on columns but also on GROUP BY functions, aggregate functions, table selections, etc.
 - SELECT customer_id, SUM(amount) AS total_spent FROM payment GROUP BY customer_id;
- JOINS – allows us to relate data from one table to another table
 - Three types of joins – Inner, Outer, Self-Join

- INNER JOIN –



- SELECT A.pka, A.c1, B.pkb, B.c2 FROM A INNER JOIN B ON A.pka = B.fka;
- INNER JOIN returns rows in table A that have corresponding rows in B table.

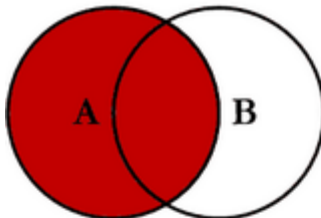


PostgreSQL INNER JOIN

- - INNER JOIN can also be written as JOIN i.e., JOIN by default is taken as INNER JOIN
 - `SELECT title, name AS movie_language FROM film JOIN language AS lan ON lan.language_id = film.language_id;`
 - Format for using alias for table name
- JOINS – can be viewed as VENN diagram

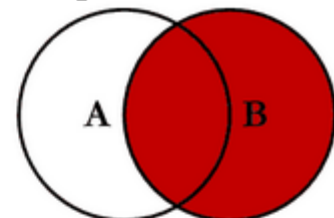
SQL JOINS

Left Outer Join



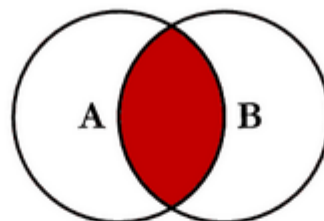
```
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON A.Key = B.Key
```

Right Outer Join



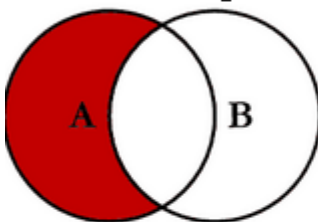
```
SELECT <select_list>
FROM Table_A A
RIGHT JOIN Table_B B
ON A.Key = B.Key
```

Inner Join



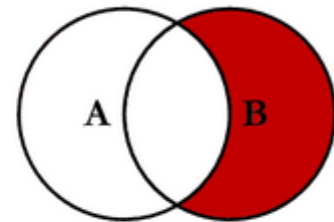
```
SELECT <select_list>
FROM Table_A A
INNER JOIN Table_B B
ON A.Key = B.Key
```

Left Excluding Join



```
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

Right Excluding Join

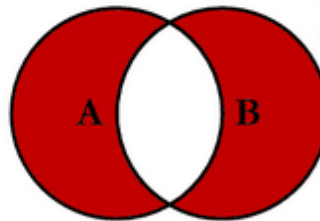
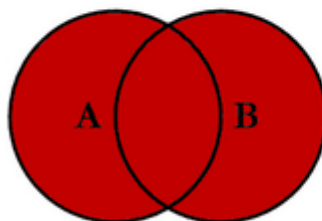


```
SELECT <select_list>
FROM Table_A A
RIGHT JOIN Table_B B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

Outer Join or
FULL OUTER JOIN
or FULL JOIN

Outer Excluding Join

```
SELECT
<select_list>
FROM Table_A A
FULL OUTER JOIN
Table_B B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM Table_A A
FULL OUTER JOIN Table_B B
ON A.Key = B.Key
WHERE A.Key IS NULL OR
B.Key IS NULL
```

'A' & 'B' are two sets.

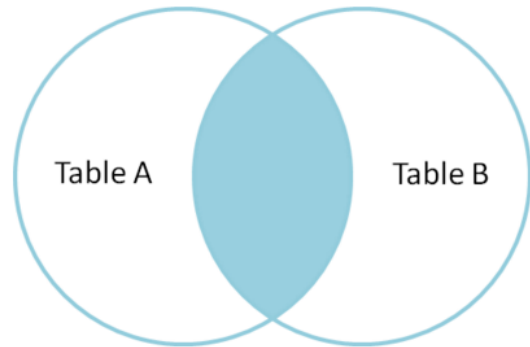
1. $A \cap B$ = Inner Join ('n' - intersection)
2. $A \cup (A \cap B)$ = Left Join ('u' - Union)
3. $(A \cap B) \cup B$ = Right Join
4. $A \cup B \cup (A \cap B)$ = Outer Join
5. $A - B$ = Left Join Excluding Inner Join or Relative Component
6. $B - A$ = Right Join Excluding Inner Join
7. $(A - B) \cup (B - A)$ = Outer Join Excluding Inner Join

id	name	id	name
--	----	--	----
1	Pirate	1	Rutabaga
2	Monkey	2	Pirate
3	Ninja	3	Darth Vader
4	Spaghetti	4	Ninja

- INNER JOIN produces only the set of records that match in both Table A and Table B.

```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name
```

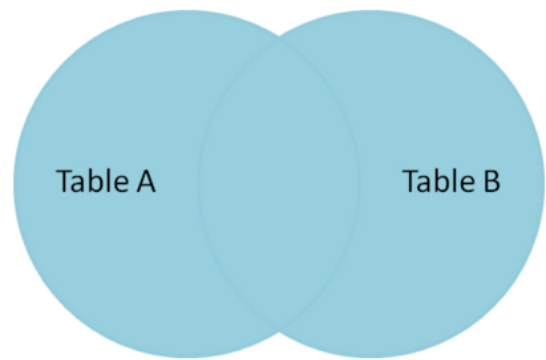
id	name	id	name
--	----	--	----
1	Pirate	2	Pirate
3	Ninja	4	Ninja



- FULL OUTER JOIN produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
```

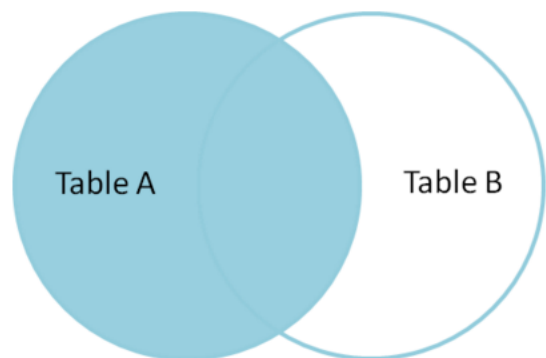
id	name	id	name
--	----	--	----
1	Pirate	2	Pirate
2	Monkey	null	null
3	Ninja	4	Ninja
4	Spaghetti	null	null
null	null	1	Rutabaga
null	null	3	Darth Vader



- LEFT OUTER JOIN - produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
```

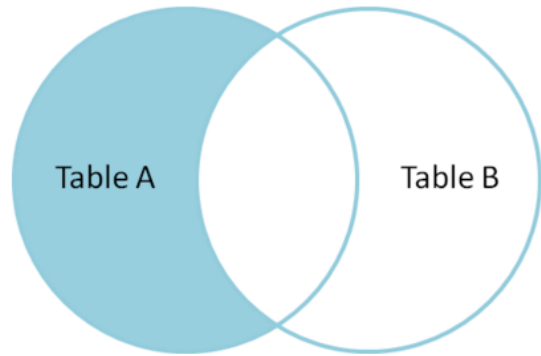
id	name	id	name
--	----	--	----
1	Pirate	2	Pirate
2	Monkey	null	null
3	Ninja	4	Ninja
4	Spaghetti	null	null



- SELECT film.film_id, film.title, inventory.inventory_id FROM film LEFT OUTER JOIN inventory ON inventory.film_id = film.film_id;
- LEFT OUTER JOIN with WHERE – to produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then exclude the records we don't want from the right side via a where clause.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null
```

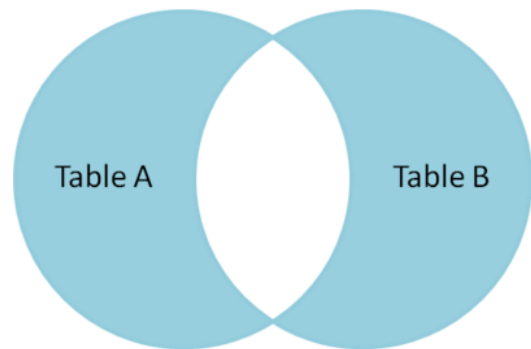
id	name	id	name
--	----	--	----
2	Monkey	null	null
4	Spaghetti	null	null



- SELECT film.film_id,film.title,inventory.inventory_id FROM film LEFT OUTER JOIN inventory ON inventory.film_id = film.film_id WHERE inventory.inventory_id IS NULL;
- FULL OUTER JOIN with WHERE - to produce the set of records unique to Table A and Table B, we perform the same full outer join, then exclude the records we don't want from both sides via a where clause.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null
```

id	name	id	name
--	----	--	----
2	Monkey	null	null
4	Spaghetti	null	null
null	null	1	Rutabaga
null	null	3	Darth Vader



- UNION – combines result set of two or more SELECT statements into single result set.
 - SELECT column_1, column_2 FROM table_1 UNION SELECT column_2, column_2 FROM table_2;
 - Rules while using the UNION operator: -
 - Both of the queries must return the same number of columns
 - Corresponding columns in the queries must have the compatible data type.
 - Removes all duplicate rows unless UNION ALL is used.
 - It places the rows in first query before, after or between the rows in the result set of second query
 - To sort the rows in the combined result set by a specified column, you can use the ORDER BY clause.
 - We often use the UNION operator to combine data from similar tables that are not perfectly normalised.
 - These tables are often found in the reporting or data warehousing system.

Imagine we have two tables:

- sales2007q1: stores sales data in Q1 2007
- sales2007q2: stores sales data in Q2 2007

sales2007q1

name	amount
Mike	150000.25
Jon	132000.75
Mary	100000

sales2007q2

name	amount
Mike	120000.25
Jon	142000.75
Mary	100000

**SELECT * FROM sales2007q1
UNION**

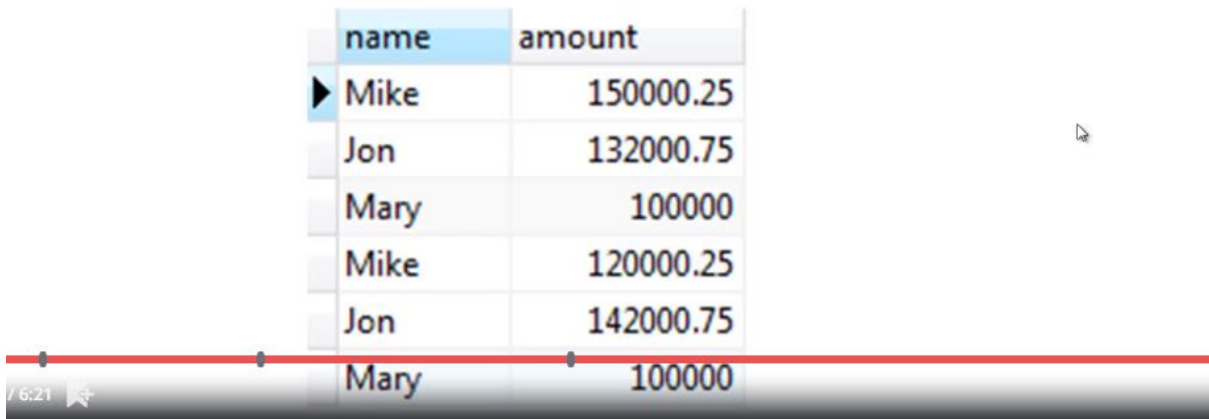
SELECT * FROM sales2007q2;

name	amount
Jon	132000.75
Jon	142000.75
Mary	100000
Mike	120000.25
Mike	150000.25

SELECT * FROM sales2007q1

UNION ALL

SELECT * FROM sales2007q2;



name	amount
Mike	150000.25
Jon	132000.75
Mary	100000
Mike	120000.25
Jon	142000.75
Mary	100000

- Timestamps – Different SQL engines (MYSQL, Oracle, etc.) use different syntax. So, see their documentation to get to know about their syntax.
- EXTRACT – to extract part from a date which can be many types of time-based information
 - EXTRACT (unit from date)
 - <https://www.postgresql.org/docs/9.0/functions-datetime.html>

Extract Function

Unit	Explanation
day	Day of the month (1 to 31)
dow	Day of the week (0=Sunday, 1=Monday, 2=Tuesday, ... 6=Saturday)
doy	Day of the year (1=first day of year, 365/366=last day of the year, depending if it is a leap year)
epoch	Number of seconds since '1970-01-01 00:00:00 UTC', if date value. Number of seconds in an interval, if interval value
hour	Hour (0 to 23)
microseconds	Seconds (and fractional seconds) multiplied by 1,000,000
millennium	Millennium value
milliseconds	Seconds (and fractional seconds) multiplied by 1,000
minute	Minute (0 to 59)
month	Number for the month (1 to 12), if date value. Number of months (0 to 11), if interval value
quarter	Quarter (1 to 4)
second	Seconds (and fractional seconds)
week	Number of the week of the year based on ISO 8601 (where the year begins on the Monday of the week that contains January 4th)
year	Year as 4-digits

- We can use arithmetic operators (+, *, etc.) with date time objects
- `SELECT EXTRACT(day from payment_date) FROM payment;`
- `SELECT SUM(amount) , EXTRACT(month from payment_date) AS month FROM payment GROUP BY month ORDER BY SUM(amount) DESC LIMIT 1;`
 - Above will return the highest monthly collection in a single month
- MATHEMATICAL FUNCTIONS - <https://www.postgresql.org/docs/9.5/functions-math.html>
 - May vary from one engine to another
 - `SELECT customer_id + rental_id AS new_id FROM payment;`
 - `SELECT customer_id / rental_id AS new_id FROM payment;`
 - Will show zero as result
 - `SELECT rental_id/ customer_id AS new_id FROM payment;`
 - Will show integer smaller than number as a result
- STRING FUNCTIONS - <https://www.postgresql.org/docs/9.5/functions-string.html>
 - `SELECT first_name || ' ' || last_name AS full_name FROM customer;`
 - `SELECT first_name,CHAR_LENGTH (first_name) FROM customer;`
 - Will return the number of characters in the first_name in a separate column
 - `SELECT UPPER (first_name) FROM customer;`
 - Will return everything in an upper case
- SUBQUERY – allows us to use multiple SELECT statements, where we basically have a query within a query
 - TASK – To find out the films whose rental rate is greater than the average rental rate. This can be done in two ways. First, we can find the average rental rate and use this average with a where statement to find out the required films. In the second case, we can use subquery to directly come at the result.
 - `SELECT AVG(rental_rate) FROM film;` will lend us 2.98 as a result.
 - `SELECT title, rental_rate FROM film WHERE rental_rate >= 2.98;`
 - Subquery is query nested inside another query.
 - We have to put the second query in brackets and use it in the where clause as an expression
 - `SELECT title, rental_rate FROM film WHERE rental_rate >= (SELECT AVG(rental_rate) FROM film);`
 - First engine will solve the subquery, then it will pass that result to the outer query and then it solves the outer query.

- TASK – to find the films that have been returned between May 29th and May 30th.
 - `SELECT inventory.inventory_id, film_id, return_date FROM inventory INNER JOIN rental ON rental.inventory_id = inventory.inventory_id WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30';`
- Also find the title of the above films
 - `SELECT title, film_id FROM film WHERE film_id IN (SELECT inventory.film_id FROM inventory INNER JOIN rental ON rental.inventory_id = inventory.inventory_id WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30') ORDER BY film_id ASC;`
- SELF JOIN – It is used to join a table with itself or to combine rows with other rows in the same table.
 - To perform this operation, we must use a table alias or AS statement to help SQL distinguish the left table from the right table of the same table.

Self Join

Suppose we want to find out which employees are from the same location as the employee named Joe.

employee_name	employee_location
Joe	New York
Sunil	India
Alex	Russia
Albert	Canada
Jack	New York

- Above problem can be solved by three methods:-
 - `SELECT employee_name FROM employee WHERE employee_location = 'New York';`
 - `SELECT employee_name FROM employee WHERE employee_location IN (SELECT employee_location FROM employee WHERE employee_name = 'Joe');`
 - `SELECT e1.employee_name FROM employee AS e1, employee AS e2 WHERE e1.employee_location = e2.employee_location AND e2.employee_name = 'Joe';`

Self Join

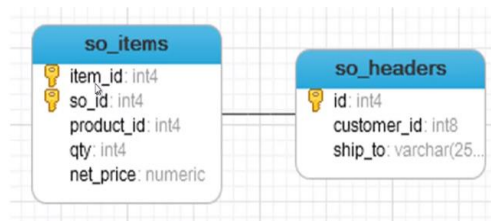
- And the final results of running the self join query above – the actual joined table – would look like this:

e1.employee_name	e1.employee_location	e2.employee_name	e2.employee_location
Joe	New York	Joe	New York
Jack	New York	Joe	New York

- Benefits of self-join:-
 - Less text than sub query
 - Performance benefit over sub-query
- `SELECT c1.first_name,c1.last_name,c2.first_name,c2.last_name FROM customer AS c1,customer AS c2 WHERE c1.first_name = c2.last_name;`
- `SELECT c1.customer_id, c1.first_name, c1.last_name, c2.customer_id, c2.first_name, c2.last_name FROM customer AS c1,customer AS c2 WHERE c1.first_name = c2.last_name;`
- `SELECT c1.customer_id, c1.first_name, c1.last_name, c2.customer_id, c2.first_name, c2.last_name FROM customer AS c1 JOIN customer AS c2 ON c1.first_name = c2.last_name;`
- LEFT JOIN can also be used on SELF JOIN as shown below:-
 - `SELECT c1.customer_id, c1.first_name, c1.last_name, c2.customer_id, c2.first_name, c2.last_name FROM customer AS c1 LEFT OUTER JOIN customer AS c2 ON c1.first_name = c2.last_name;`
- DATA TYPES – PostgreSQL supports following datatypes
 - Boolean
 - Character
 - Number
 - Temporal (time and date related data types)
 - Special types (like geometric data types)
 - Array
 - We should specify what kind of data type a column has, when we are creating a table.
- Boolean – can hold two possible values TRUE and False
 - In case value is unknown NULL value is used.
 - We can use Boolean or bool keyword to declare a column having Boolean datatype.
 - PostgreSQL takes and convert
 - T, true, 1, y and yes to true
 - 0, n, no, f, false to false

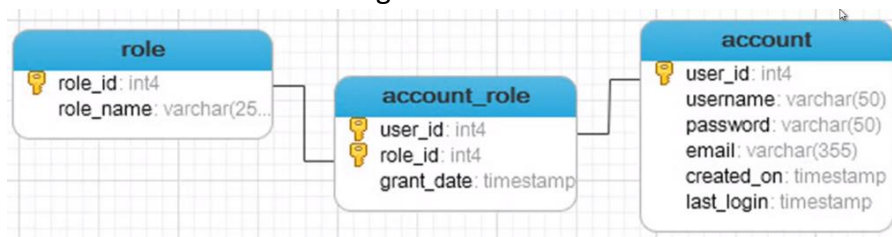
- PostgreSQL will return t for true, f for false and space character for NULL.
- Character – can be of three types
 - A single character – char
 - Fixed length character strings: char(n)
 - If entered string is short then it will add spaces
 - If entered string is longer then it will show error
 - Can be used for school ID etc.
 - Variable length character strings: varchar(n)
 - If entered string is short then it will not add spaces
 - If entered string is longer then it will show error
- Numbers – can be two types
 - Integer
 - Floating point numbers
- Integers – can be of three types
 - Small integers (smallint) – 2 byte signed integer with range of (-32768,32767)
 - Integer (int) – 4byte signed integer with range of (-214783648,214783647) or 2^{31}
 - Serial – same as integer except that PostgreSQL populate value into the column automatically.
 - Similar to AUTO_INCREMENT attribute in other DBMS
- Floating Point – can be of three types
 - float(n) – precision at least n upto a maximum of 8 bytes
 - real or float8 – double precision (8 byte) floating-point number
 - numeric or numeric(p,s)- real number with p digits with s number after decimal points
 - numeric(p,) is the exact number
- Temporal - stores date and time data
 - date stores date data
 - time stores time data
 - timestamp stores date and time
 - interval stores difference in timestamp
 - timestamptz stores both timestamp and timezone data
- Keys – can be of two types
 - Primary keys
 - Foreign keys
- Primary Keys: - column or group of columns that is used to identify a row uniquely in a table
 - Defined through primary key constraints
 - A table can have only one primary key
 - Good practice to add primary key to every table

- When we add a primary key to a table, PostgreSQL creates a unique index on the column or a group of columns used to define the primary key.
- We add the primary key to the table when we define the table's structure
- CREATE table_name (column_1 data_type PRIMARY KEY, column_2 data_type,...);
- Foreign Key: - Fields or a group of fields in a table that uniquely identifies a row in another table. In another words, a foreign key is defined in a table that refers to the primary key of the other table.
 - The table that contains the foreign key is called as child table or referencing table.
 - The table to which foreign key references is called referenced table or parent table.
 - A table can have multiple foreign key depending upon its relationships with another table.
 - In PostgreSQL, a foreign key is defined through a foreign key constraint.
 - A foreign key constraint indicates that values in a column or group of columns in a child table match with the values of a column or a group of columns of the parent table.
 - Foreign key constraint maintains the referential integrity between the child and the parent tables.



- In the so_items table item_id is the primary key of the table and so_id (Sales order id) is the foreign key, while id is the primary key of the so_headers table.
- CREATE TABLE: - to create a new table
 - CREATE TABLE table_name (column_name DATA_TYPE column_constraint, table_constraint) INHERITS existing table_name;
 - table_constraints are typed only after all new columns have been entered.
 - Inheriting from another table is optional
- PostgreSQL column constraints
 - NOT NULL: - the value of column cannot be NULL
 - UNIQUE: - the value of the column must be unique across the whole table
 - Column can have many NULL values as PostgreSQL treats each NULL value to be unique
 - SQL standard only allows one NULL value in the column that has UNIQUE constraint

- As websites generally tell us that this username is already taken.
- PRIMARY KEY: - this constraint is combination of UNIQUE and NOT NULL constraints.
 - We can define one column as PRIMARY KEY using column level constraint
 - In case, PRIMARY KEY contains multiple columns, you must use table level constraint.
- CHECK: - to check a condition when you insert or update data
 - Price of any good/item must be positive
- REFERENCES: - constrains the value of the column that exists in a column in another table.
 - We use it to define FOREIGN KEY constraint.
- PostgreSQL table constraints: - They are similar to column constraints with only difference being that they are applied to complete table rather than being applied to a single column
 - UNIQUE(column_list):- to force the value stored in the columns listed inside the parenthesis to be unique.
 - PRIMARY KEY(column_list): - to define the primary key that consists of multiple column
 - CHECK(column_list): - to check a condition when inserting or updating data
 - REFERENCES: - to constrain the value stored in the column that must exist in a column in another table.
 - Exercise: - to create following tables

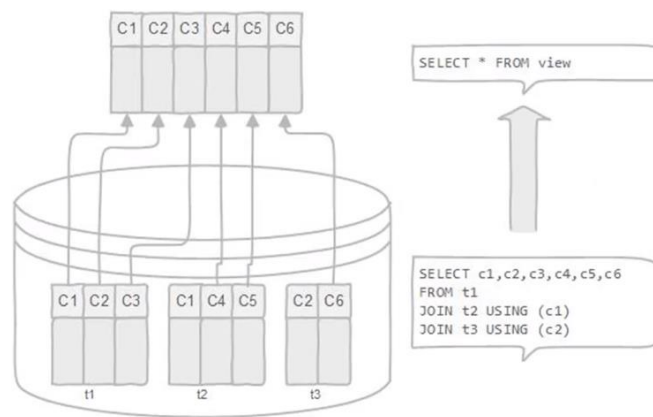


- INSERT – to enter the rows of data into the tables created earlier
 - INSERT INTO table_name (column_1, column_2,...) VALUES (value_1, Value_2);
 - The values list must be in the same order as the columns list specified after the table name.
 - To insert multiple rows we can use following syntax
 - INSERT INTO table_name (column_1,column_2,...) VALUES(value_1,value_2,...),(value_3, value_4,...),.....;
 - To insert data that comes from another table, we can use INSERT INTO SELECT statement
 - INSERT INTO table_name SELECT column1, column2,... FROM table_name2 WHERE condition;
 - CREATE TABLE link (ID SERIAL PRIMARY KEY, url VARCHAR(255) NOT NULL, name VARCHAR(255) NOT NULL, description VARCHAR(255), rel VARCHAR(50));
 - INSERT INTO link(url,name) VALUES ('www.google.com','Google'),('www.yahoo.com','Yahoo');
 - CREATE TABLE link_copy(LIKE link);

- INSERT INTO link_copy SELECT * FROM link WHERE name = 'Google';
- UPDATE – to update or to change the values of the existing table
 - UPDATE table_name SET column1 = value1, column2 = value2,... WHERE condition;
 - UPDATE link SET description = 'Empty Description';
 - Will set each value in description column to Empty Description
 - UPDATE link SET description = 'Starts with A' WHERE name LIKE '%A%';
 - We can update the data of a column from another column of the same table.
 - Always check for the data type. Only if, the data types match then we can update the column.
 - UPDATE link SET description = name;
 - UPDATE link SET description = 'New Description' WHERE id = 1 RETURNING id,url,name,description,rel;
 - Will return the column mentioned with the returning keyword and affected/updated by the query.
- DELETE – to delete the rows of the table
 - DELETE FROM table_name WHERE condition;
 - Where clause can be used to delete the specific row that we want to delete.
 - If we omit the where clause then all the rows in the table will be deleted.
 - DELETE statement returns the number of rows that are deleted.
 - DELETE FROM link WHERE name LIKE '%B%';
 - DELETE FROM link WHERE name LIKE '%Y%' RETURNING id, url, name, description, rel;
 - Will return the row that is deleted as all the columns are mentioned in the returning statement.
- ALTER – to change the existing table structure we can use ALTER TABLE statement.
 - ALTER TABLE table_name action;
 - PostgreSQL provides following actions: -
 - Add, remove or rename column
 - Set default value for the column
 - Add CHECK constraint to the column
 - Rename table
 - Keywords to be used are: -
 - ADD COLUMN
 - DROP COLUMN
 - RENAME COLUMN
 - ADD CONSTRAINT
 - RENAME TO
 - DROP TABLE if EXISTS link;
 - Will drop the table named link if it already exists
 - CREATE TABLE link (id serial PRIMARY KEY, title VARCHAR (512) NOT NULL, url VARCHAR (1024) NOT NULL UNIQUE);
 - ALTER TABLE link ADD COLUMN active BOOLEAN;

- ALTER TABLE link DROP COLUMN active;
- ALTER TABLE link RENAME COLUMN title to new_title_name;
- ALTER TABLE link RENAME to url_table;
 - This will rename the whole table not just a column
- DROP TABLE – to remove/drop a table from the database
 - DROP TABLE [IF EXISTS] table_name;
 - IF EXISTS is optional and is used to avoid errors
 - RESTRICT keyword if used after the DROP TABLE statement will refuse to drop a table if there is an object dependent on it.
 - CASCADE keyword can be used after the DROP TABLE statement to drop the table along with the objects dependent on it.
- CHECK - a constraint that allows us to specify if a value in a column must meet specific requirement.
 - The CHECK constraint uses a BOOLEAN expression to evaluate the values of a column.
 - If values of the column pass the check, PostgreSQL will insert or update the value.
 - CREATE TABLE new_user(id SERIAL PRIMARY KEY, first_name VARCHAR(50), birth_date DATE CHECK (birth_date >= '01-01-1900'), join_date DATE CHECK(join_date > birth_date), salary INTEGER CHECK (salary > 0));
 - INSERT INTO new_user (first_name, birth_date, join_date,salary) VALUES ('Joe','1980-02-02','1990-04-04',-10);
 - Will give error as salary can't be negative
 - CREATE TABLE checktest(sales INTEGER CONSTRAINT positive_check CHECK (sales>0));
 - Using the CONSTRAINT keyword will help us naming the error message.
 - Now error will positive_check as the error message.
- NULL/NOT NULL: - In database theory, NULL is unknown or missing information.
 - It is different from zero or empty
 - PostgreSQL provides the NOT NULL constraint to enforce a column must not accept NULL values.
 - CREATE TABLE learn_null(first_name VARCHAR(50), sales INTEGER NOT NULL);
 - INSERT INTO learn_null (first_name) VALUES ('John');
 - This will give following error message: -ERROR: null value in column "sales" violates not-null constraint DETAIL: Failing row contains (John, null).
- UNIQUE – This constraint makes sure that value in a column or a group of columns is unique across a whole table.
 - With UNIQUE constraint, every time we insert a new row, PostgreSQL checks if the value is already in the table.

- If it finds that the new value is already there, it will return the error message and reject the changes.
- The same process is carried out for the update existing data.
- CREATE TABLE people(id serial PRIMARY KEY, first_name VARCHAR(50), email VARCHAR (100) UNIQUE);
- INSERT INTO people(id, first_name, email) VALUES(1,'Joe','joe@joe.com');
- INSERT INTO people(id,first_name,email) VALUES(2,'Joseph','joe@joe.com');
 - This will give an error message like ERROR: duplicate key value violates unique constraint "people_email_key" DETAIL: Key (email)=(joe@joe.com) already exists.
- VIEW – Database object that is of a stored query
 - A view can be accessible as a virtual table in PostgreSQL
 - PostgreSQL view is a logical table that represents data of one or more underlying tables through a SELECT statement.



- Stack in above diagram represents databases.
- Where t1, t2, t3 represents table and c1, c2, c3, c4, c5, c6 represents column
- View does not store data physically
- A view helps simplify the complexity of a query because you can query a view, which is based on a complex query, using a simple SELECT statement.
- Like a table, you can grant permission to the user through a view that contains specific data that the users are authorised to see.
- A view provides a consistent layer even the column of underlying table changes.
- To create a view, we use CREATE VIEW statement.
- The simplest of the CREATE VIEW statement is as follows: -
 - CREATE VIEW view_name AS query;
- SELECT first_name,last_name,email,address,phone FROM customer INNER JOIN address ON customer.address_id = address.address_id;
 - Either we can save this query for repetitive view or we can create a view
- CREATE VIEW customer_info AS SELECT first_name, last_name, email, address, phone FROM customer INNER JOIN address ON customer.address_id = address.address_id;
 - Now we can just use SELECT * FROM customer_info; to view the join table.

- VIEW just creates a virtual table that means it is not going to change the way data is stored physically.
- ALTER VIEW – allows us to alter the name of the view.
 - ALTER VIEW name_of_view RENAME TO new_name;
 - ALTER VIEW customer_info RENAME TO customer_masterlist;
 - SELECT * FROM customer_masterlist;
- DROP VIEW: - to drop any existing view
 - DROP VIEW view_name;
 - DROP VIEW customer_masterlist;
 - We can also use IF EXISTS statement here
 - DROP VIEW IF EXISTS customer_masterlist;
- CREATE – can also be used to create a new database
 - CREATE DATABASE testdvd;
- Restoring a table schema only: - This method is used to restore the table schema and respective data types of a different database without copying the data of the database.
 - So, let us say if we have a new dvd rental store, we could just copy the table schema or structure and the respective data types from the previous dvdrental database and start putting the new transactions or data in that table schema to create a new database.
 - CREATE DATABASE new_database_name;
 - Right click on the new database and select restore option.
 - Restore the database from whom we want to copy the table schema while doing following things: -
 - Go to restore options
 - Select only schema option
 - This will copy only the schema and no data from dvdrental database file or dvdrental.tar
 - To restore a table schema in a database where data is there, we will follow the same process as above with one additional step.
 - While selecting only schema option, we will also select clean before restore option. This will clean the database from data available previously.
- To populate a database using command line we could use following method: -

- Download dvdrental.zip
- Extract .tar file from dvdrental.zip
- Open pgadmin III and go to run SQL Query and run:
CREATE DATABASE dvdrental;
- Open command prompt (search for **cmd**)
- Use **cd** to change directory to Postgres bin folder
- Use **pg_restore -U postgres -d dvdrental "C://path/to/tar"**

ORDER OF EXECUTION

Logical Processing Order of the SELECT statement

The following steps show the logical processing order, or binding order, for a SELECT statement. This order determines when the objects defined in one step are made available to the clauses in subsequent steps. For example, if the query processor can bind to (access) the tables or views defined in the FROM clause, these objects and their columns are made available to all subsequent steps. Conversely, because the SELECT clause is step 8, any column aliases or derived columns defined in that clause cannot be referenced by preceding clauses. However, they can be referenced by subsequent clauses such as the ORDER BY clause. **Note that the actual physical execution of the statement is determined by the query processor and the order may vary from this list.**

which gives the following order:

```
FROM
ON
JOIN
WHERE
GROUP BY
WITH CUBE or WITH ROLLUP
HAVING
SELECT
DISTINCT
ORDER BY
TOP
```

TOPICS TO BE COVERED

- CAST
- Manager employee self-join – Interview question

USEFUL LINKS

- <http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/>
- <http://www.sql-join.com/>
- [https://en.wikipedia.org/wiki/Join_\(SQL\)](https://en.wikipedia.org/wiki/Join_(SQL))
- https://wiki.postgresql.org/wiki/Sample_Databases

