SQL BOOTCAMP DOCUMENTATION

Databases are Good For:

- data integrity – where there are access modifiers so not anyone can access a database

- can handle massive amounts of data

- quickly combine different data sets

- automate steps for re-use

- can support data for websites and applications

DATA TYPES:

- Numeric data types such as int, tinyint, bigint, float, real etc.
- Date and Time data types such as Date, Time, Datetime etc.
- Character and String data types such as char, varchar, text etc.
- Unicode character string data types, for example nchar, nvarchar, ntext etc.
- Binary data types such as binary, varbinary etc.
- Miscellaneous data types – clob, blob, xml, cursor, table etc.

Data Types	From	To
Bit	0	1
tinyint	0	255
smallint	-32,768	32,767
int	-2,147,483,648	2,147,483,648
bigint	-9,233,372,036,854,775,808	9,233,372,036,854,775,808
decimal	-10^38+1	10^38-1
numeric	-10^38+1	-10^38-1
float	-1.79E+308	1.79E+308
real	-3.40E + 38	3.40E + 38

SQL CHARACTER AND STRING DATA TYPES

DATE	Stores date in the format YYYY-MM-DD
TIME	Stores time in the format HH:MI:SS
DATETIME	Stores date and time information in the format YYYY-MM-DD HH:MI:SS
TIMESTAMP	Stores number of seconds passed since the Unix epoch (‘1970-01-01 00:00:00’ UTC)
YEAR	Stores year in 2 digit or 4 digit format. Range 1901 to 2155 in 4-digit format. Range 70 to 69, representing 1970 to 2069.

SQL CHARACTER AND STRING DATATYPES

CHAR	Fixed length with maximum length of 8,000 characters
VARCHAR	Variable length storage with maximum length of 8,000 characters
VARCHAR(max)	Variable length storage with provided max characters, not supported in MySQL
TEXT	Variable length storage with maximum size of 2GB data

SQL UNICODE CHARACTER AND STRING DATA TYPES

NCHAR	Fixed length with maximum length of 4,000 characters
NVARCHAR	Variable length storage with maximum length of 4,000 characters
NVARCHAR(max)	Variable length storage with provided max characters
NTEXT	Variable length storage with maximum size of 1GB data

SQL BINARY DATA TYPES

BINARY	Fixed length with maximum length of 8,000 bytes
VARBINARY	Variable length storage with maximum length of 8,000 bytes
VARBINARY(max)	Variable length storage with provided max bytes
IMAGE	Variable length storage with maximum size of 2GB binary data

SQL MISCELLANEOUS DATA TYPES

CLOB	Character large objects that can hold up to 2GB
BLOB	For binary large objects
XML	For storing xml data
JSON	For storing JSON data

(Cont. =>)

SQL SYNTAX:

SELECT clause:
Example: SELECT column1, column2, … FROM table_name

first you specify a list of columns in the table from which you want to query data in the SELECT clause. You use a comma between each column in case you want to query data from multiple columns

If you want to query data from all columns, you can use an asterisk(*) as the shorthand for all columns.

Second, you indicate the table name after the FROM keyword.

SQL language is case insensitive, despite this, you use SQL keywords in uppercase to make code easier to read.

It is not good practice to use the asterisk (*) in the select statement.

Imagine that you have a large table with many columns, the SELECT statement with an asterisk(*) will query all the data from the entire columns, which may not be necessary.

It makes your database server work harder and increase the traffic between the database and server and applications.

Therefore, you should specify the column names in the SELECT clause whenever possible to get only necessary data from the table.

Examples:
SELECT * FROM actor; - returns all columns from table actor.

SELECT first_name, last_name FROM actor; - gets data from requested column

SELECT DISTINCT:
select distinct allows us to find the distinct types in a table. It can be used like so:

Examples:
SELECT DISTINCT rental_rate FROM film; - selects all the possible rental rates in our table

SELECT DISTINCT release_year FROM film; - selects and returns all the years from the year_release column

SELECT WHERE:
allows more specific selects based upon conditions

(Cont. =>)
Syntax Example:

SELECT column_1,column_2, … column_n
FROM table_name
WHERE conditions;

The WHERE clause appears right after the FROM clause of the SELECT statement.

The conditions are used to filter the rows returned from the SELECT statement.

PostgreSQL provides you with various standard operators to construct the conditions.

SELECT WHERE STATEMENT OPERATORS:

OPERATOR	DESCRIPTION
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal
AND	Logical operator AND
OR	Logical operator OR

Examples:
SELECT email FROM customer WHERE first_name = 'Jared'; - gathers the email from customer table where the first_name value is equal to the string Jared.

SELECT * FROM payment WHERE amount > 5; - can select values from payment where the amount is greater than 5 dollars. Same for less than, greater than or equal to, less than or equal to, and not equal to.

SELECT * FROM payment
where amount = 4.99 OR amount = 1.99; - checks for a or statement and pulls data where amount equals both of the values

SELECT * FROM customer
where store_id = 1 AND address_id > 5; - and logical operator that can check specifics on multiple columns.

COUNT:
The COUNT function returns the number of input rows that match a specific condition of a query.

(Cont. =>)
Example Syntax:

SELECT COUNT (*) FROM table;

The COUNT(*) function returns the number of rows returned by a SELECT clause.

When you apply the COUNT(*) to the entire table, PostgreSQL scans table sequentially.

COUNT statement can also specify a specific column count for readability.
Example:
SELECT COUNT(column) FROM table;

Similar to the COUNT(*) function, the COUNT(column) function returns the number of rows returned by a SELECT clause.

However, it does not consider NULL values in the column.

We can use COUNT with DISTINCT, for example

Examples:
SELECT COUNT(DISTINCT column)
FROM table; - returns a distinct column from the table with counted values

SELECT COUNT(*) FROM payment; - returns the number of entries quicker

SELECT COUNT(DISTINCT amount) FROM payment; - returns the total number of distinct amount types from our payment table.

LIMIT:
Limit allows you to limit the number of rows you get back from a query.
Example:
SELECT * FROM customer LIMIT 5; -returns first 5 rows

ORDER BY:
Order by allows specifying the column that you want to sort in the ORDER BY clause. If you sort the result set by multiple columns, use a comma to separate between two columns

Use ASC to sort the result set in ascending order and DESC to sort the result set in descending order.

If you leave it blank, the ORDER BY clause will use ASC by default.

Examples:
SELECT first_name, last_name FROM customer
ORDER BY first_name ASC; - allows us to sort by ascending order for first and last name and ordered by first name.

(Cont. =>)
SELECT first_name, last_name FROM customer

ORDER BY last_name DESC; - allows us to sort be descending order for first and last name. Ordered descending by last name.

SELECT first_name, last_name FROM customer
ORDER BY first_name ASC,
last_name DESC;   - this command allows us to sort by first name primarily in the ascending order but if there is name overlap (same name) we then sort the last name in descending order.

BETWEEN:

We use between operator to match a value against a range of values.

Example Syntax:
value BETWEEN low AND high

if the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, or vice versa.

We can rewrite the between operator by using the greater than or equal (>=) operator or less than equal (<=) operators as the following statement:

value >= low and value <= high

EXAMPLES:
SELECT customer_id, amount FROM payment
WHERE amount BETWEEN 8 AND 9; - selects customer id and amount from payment table. From here we set where from our amount and return values between 8 and 9 dollars.

SELECT customer_id, amount FROM payment
WHERE amount NOT BETWEEN 8 AND 9; - here we have the same syntax as above. The only difference is that we have NOT included. This inclusion returns all values as long as they are not between 8 and 9.

SELECT amount, payment_date FROM payment
WHERE payment_date BETWEEN '2007-02-07' AND '2007-5-19'; - this between statements returns all possible dates between the two dates (Syntax such as ‘YYYY-MM-DD’).

IN STATEMENT:
You can use the IN operator with the WHERE clause to check if a value matches any value in a list of values. The syntax of the IN operator is as follows:
EXAMPLE SYNTAX:

value IN(value1,value2,…)

(Cont. =>)

EXAMPLES:

SELECT customer_id, rental_id, return_date
FROM rental
WHERE customer_id IN (7,13,10)
ORDER BY return_date DESC; - this command takes customer id, rental id, and return date from rental table and returns where the customer id is (7,13,10) specifically.
Then it order them by return date in descending fashion.

SELECT *
FROM payment
WHERE amount IN (7.99,8.99)
ORDER BY amount ASC; - this command gets all columns from payment table where the amount is either 7.99 or 8.99. From here we order by the amount ascending.

LIKE STATEMENT:

like statement is a pattern matching statement. You can use this with special characters to find specific values or items based upon their values.

EXAMPLE SYNTAX:

SELECT first_name, last_name
FROM customer
WHERE first_name LIKE ‘Jen%’; - checks first and last name from customer where first name is like JEN. The like and % symbol is a character set when in SQL is referred to as a pattern.

You can construct a pattern by combining a string with wildcard characters and use the LIKE or NOT LIKE operator to find the matches.

-Percent (%) for matching any sequence of characters.
-Underscore (_) for matching any single character

EXAMPLES:

SELECT first_name, last_name
FROM customer
WHERE first_name LIKE '%y'; - returns all names that end with y from customer table and first name column

SELECT first_name, last_name
FROM customer
WHERE first_name LIKE '%er%'; - returns a list with any name that has er in the name.

(Cont. =>)

SELECT first_name, last_name
FROM customer
WHERE first_name ILIKE 'BAR%'; - ILIKE is a variant of like that does not care about case sensitivity.

<u>AGGREGATE FUNCTIONS</u>

<u>AVG:</u>
<u>EXAMPLE SYNTAX:</u>
SELECT AVG(amount) FROM payment; - AVGs passed value has parenthesis around the value as to get the value from that column from payment table.

<u>ROUND:</u>
<u>EXAMPLE SYNTAX:</u>
SELECT ROUND(AVG(amount),2) FROM payment; - this statement allows for the use of round. When used, the first value is the amount and the second value is the number of decimal places.

<u>MIN:</u>
<u>EXAMPLE SYNTAX:</u>
SELECT MIN(amount) FROM payment; - shows min amount of a specific value

<u>COUNT:</u>
<u>EXAMPLE SYNTAX:</u>
SELECT COUNT(amount) FROM payment WHERE amount = 0.00; - gets the total number of values where amount is equal to 0.00 from amount column.

<u>MAX:</u>
<u>EXAMPLE SYNTAX:</u>
SELECT MAX(amount) FROM payment; - gets the max value for amount from payment.

SUM:
<u>EXAMPLE SYNTAX:</u>
SELECT SUM(amount) FROM payment; - gets sum of amount from payment table.

<u>GROUP BY:</u>

The GROUP BY clause divides the rows returned from the SELECT statement into groups.

For each group, you can apply an aggregate function, for example:
-calculating the sum of items
-count the number of items in the groups.

<u>EXAMPLE SYNTAX:</u>
SELECT column_1, aggregate_function(column_2)
FROM table_name
GROUP BY column_1;

SELECT customer_id, SUM(amount)
FROM payment
GROUP BY customer_id; -sums the value of amount per customer and then groups by the customer id.

SELECT staff_id, COUNT(payment_id)
FROM payment
GROUP BY staff_id
ORDER BY staff_id DESC; - gets the staff ids and how many payments they have processed

SELECT rating, COUNT(rating)
FROM film
GROUP BY rating; - counts the occurrences of a specific rating.

SELECT rating, AVG(rental_rate)
FROM film
GROUP BY rating; - gets back average rental rate for each category of movie rating

<u>HAVING:</u>
we often use the HAVING clause in conjunction with the GROUP BY clause to filter group rows that do not satisfy a specified condition.

<u>EXAMPLE SYNTAX:</u>
SELECT column_1, aggregate_function(column_2)
FROM table_name
GROUP BY column_1
HAVING condition;

<u>EXAMPLES:</u>
SELECT customer_id, SUM(amount)
FROM payment
GROUP BY customer_id
HAVING SUM(amount) > 200; - this syntax returns a customer id and the amount they have paid. It is grouped by the customer id but the sum is only returned if the amount is greater than 200.
(Cont. =>)

Select store_id, COUNT(customer_id)
FROM customer
GROUP BY store_id
HAVING COUNT(customer_id) > 300 – returns a stored id column by the group by and shows what store has greater than 300 customers.

SELECT rating, AVG(rental_rate)
FROM film
WHERE rating IN(‘R’,’G’,’PG’)
GROUP BY rating
HAVING AVG(rental_rate) < 3;

ADVANCED CONCEPTS

AS STATMENT:
AS allows us to rename columns or table selections with an alias

EXAMPLES:
SELECT payment_id AS my_payment_column
FROM payment;
GROUP BY customer_id; -simply returns column name as the pseudonym.

JOINS:
JOINS allow us to match data columns from various tables into one query

There are several kinds of joins including INNER JOIN, OUTER JOIN, and self-join

INNER JOIN:
Inner join produces only the set of records that match in both table A and table B.

The INNER JOIN clause returns rows in A table that have the corresponding rows in the B table.

First you specify the column in both tables from which you want to select data in the SELECT clause.

Second you specify the main table, A in the FROM clause.

Third you specify the table that the main table joins to i.e., B in the INNER JOIN clause. In addition you put a join condition after the ON keyword I.e, A.pka =B.fka.
EXAMPLE SYNTAX:
TIP -“To get data from both tables, you use the INNE JOIN clause in the SELECT statement as follows”

SELECT A.pka, A.c1, B.pkb, B.c2
FROM A
INNER JOIN B ON A.pka = B.fka;

(Cont. =>)

EXAMPLES:

```
SELECT customer.customer_id,
customer.first_name,
customer.last_name,
customer.email,
payment.amount,
payment.payment_date
FROM customer
INNER JOIN payment ON payment.customer_id = customer.customer_id
WHERE customer.customer_id = 2
ORDER BY customer.first_name ASC;
```

- this entire example does a couple things. The select statement grabs our first key being customer_id from customer table. From that same table we grab first_name, last_name, email. From the payment table we grab payment amount, and payment date.

In the from clause, we specify customer and then call inner join.
In the INNER JOIN we then specify table b (payment) on payment.customer_id = customer.customer_id. This matches keys and creates a joined query of the two tables.

Finally we are looking for a specific customer so we check with a WHERE statement and then order by in case there are multiple with the same customer id.

```
SELECT payment.payment_id,
payment.amount,
staff.first_name,
staff.last_name
FROM payment
INNER JOIN staff ON payment.staff_id = staff.staff_id;
```

- this block gathers payment id, and amount from payments table
staff first name and last name from staff table and joins them on the staff_id of payment and staff tables.

FULL OUTER JOIN:

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.

Example Syntax:
```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name;
```

<u>LEFT OUTER JOIN:</u>
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.

<u>EXAMPLE SYNTAX:</u>

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

<u>EXAMPLES:</u>

```
SELECT film.film_id,
film.title,
inventory.inventory_id
FROM film
LEFT OUTER JOIN inventory
ON inventory.film_id = film.film_id;
```

- *left joins items from film.film_id,and film title and from inventory inventory id from inventory onto film_ID. Inventory.film_id is primary key and film.film_id is foreign key.*

```
select first_name, last_name, order_date, order_amount
from customers c
left join orders o
on c.customer_id = o.customer_id
```
 - *gets the first and last name and order date and amount. If there are no specific values in right join it nulls the values.*

<u>RIGHT JOIN:</u>
Right join is just left join but mirrored: Instead of removing columns not applicable to table b it removes the table a columns not applicable.

<u>EXAMPLES:</u>

```
select first_name, last_name, order_date, order_amount
from customers c
right join orders o
on c.customer_id = o.customer_id
```

<u>LEFT OUTER JOIN with WHERE:</u>
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.

<u>EXAMPLE SYNTAX:</u>

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

<u>EXAMPLES:</u>

```
SELECT film.film_id,
film.title,
inventory.inventory_id
FROM film
LEFT OUTER JOIN inventory ON inventory.film_id = film.film_id;
WHERE inventory.film_id IS NULL;
```

- same as before but removes null values where film_id is null. Only pulling new ids.

<u>FULL OUTER JOIN with WHERE:</u>

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.

<u>EXAMPLE SYNTAX:</u>

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

<u>EXAMPLES:</u>

```
select first_name, last_name, order_date, order_amount
from customers c
full join orders o
on c.customer_id = o.customer_id
```

-just lays ALL the columns out where the values are. Doesnt care if null or not.

UNION:
The union operator combines result sets of two or more SELECT statements into a single result set.
The following illustrates the syntax of the UNION operator that combines result sets from two queries

RULES:
both queries must return the same number of columns

The corresponding columns in the queries must have compatible data types.

The UNION operator removes all duplicate rows unless the UNION ALL is used.

The UNION operator may place the rows in the first query before, after or between the rows in the result set on the second query.

To sort the rows in the combined result set by a specified column you use the ORDER BY clause.

EXAMPLE SYNTAX:
SELECT column_1, column_2
FROM tbl_name_1
UNION
SELECT column_1,column_2
FROM tbl_name_2.

EXAMPLE:
SELECT * FROM sales2007q1
UNION ALL
SELECT * FROM sales207q2;

TIMESTAMPS:
SQL allows us to use the timestamp data type to retain time information.

The part to extract. Can be one of the following:
•MICROSECOND
•SECOND
•MINUTE
•HOUR
•DAY
•WEEK
•MONTH
•QUARTER
•YEAR
•SECOND_MICROSECOND
•MINUTE_MICROSECOND
•MINUTE_SECOND
•HOUR_MICROSECOND
•HOUR_SECOND
•HOUR_MINUTE
•DAY_MICROSECOND
•DAY_SECOND
•DAY_MINUTE
•DAY_HOUR
•YEAR_MONTH

EXAMPLES:
SELECT extract(day from payment_date) FROM payment; - extracts the day from payment date and displays that information.

SELECT customer_id, extract(day from payment_date) AS day FROM payment; - extracts payment date as day in column name and orders by customer id.

SELECT SUM(amount), extract(month from payment_date) AS MONTH FROM payment
GROUP BY month
ORDER BY SUM(amount) DESC
LIMIT 1; -
gets the sum amount from payment. Afterwards extracts the month from payment date.
We then group by the month and order by sum descending thus getting the highest amount in a specific month.

SUBQUERY:
A sub query allows us to use multiple SELECT statements, where you nest queries.

To do this we put the second query in brackets and use it in the WHERE clause as an expression:

EXAMPLES:
SELECT film_id,title,rental_rate FROM film
WHERE
rental_rate > (SELECT AVG(rental_rate) FROM film);

SELECT film_id, title
FROM film
WHERE film_id IN – this part simply grabs the film id and title from film. It then checks the film id in comparison to our sub query.

(SELECT inventory.film_id
FROM rental
INNER JOIN inventory ON inventory.inventory_id = rental.inventory_id
WHERE
return_date BETWEEN ‘2005-05-29’ AND ‘2005-05-30);

our sub query then does a inner join and matches dates in accordance with specific titles that came out between 2005-05-29 and 2005-05-30.
this compares and returns the film id as well as the films names that meet the criteria.

<u>SELF JOIN:</u>
you use self join when you want to combine rows with other rows in the same table.

To perform the self join operation you must use a table alias to help sql distinguish the left table from the right table alias.

<u>EXAMPLE SYNTAX:</u>

```
SELECT e1.employee_name
FROM employee AS e1, employee AS e2
WHERE
e1.emplyee_location = e2.employee_location
AND e2.emplyee_name=”joe”;
```

<u>EXAMPLE:</u>

```
SELECT a.customer_id, a.first_name,a.last_name,b.customer_id,b.first_name,b.last_name
FROM customer AS a
LEFT JOIN customer AS b
ON a.first_name = b.last_name
ORDER BY a.customer_id;
```

- *this self join syntax utilizes a left join to see if there is customer name overlap. We get the customer names and ids and then do a left join on the same table.*
We then return values based upon first = last name.
And order by their id. If the first and last name are the same of a person they overlap. Otherwise we get null values as the index spots.

<u>ACID PRINCIPLES:</u>

<u>ACID</u> (atomicity, consistency, isolation, and durability) is an acronym and mnemonic device for learning and remembering the four primary attributes ensured to any transaction by a transaction manager (which is also called a transaction monitor). These attributes are:

Atomicity. In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

Consistency. A transaction either creates a new and valid state of data, or, if any failure occurs, returns all data to its state before the transaction was started.

Isolation. A transaction in process and not yet committed must remain isolated from any other transaction.

Durability. Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

<u>CAP THEOREM:</u>

<u>CAP Theorem</u> is a concept that a distributed database system can only have 2 of the 3: **Consistency**, **Availability** and **Partition Tolerance**.

<u>Consistency:</u>
This condition states that all nodes see the same data at the same time. Simply put, performing a read operation will return the value of the most recent write operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process.

<u>Availability:</u>
This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times.

<u>Partition Tolerance:</u>
This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes. A system that is partition-tolerant can sustain any amount of network failure that doesn’t result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, Partition Tolerance is not an option. It’s a necessity. Hence, we have to trade between Consistency and Availability.