

What is SQL



- Computer Language used for
 - Storing
 - Manipulating
 - Retrieving Data
- Invented by **IBM**
- SQL stands for **Structured Query Language**

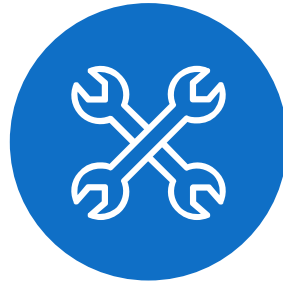
Why SQL



**Large Amount of
Data**



Controlled access

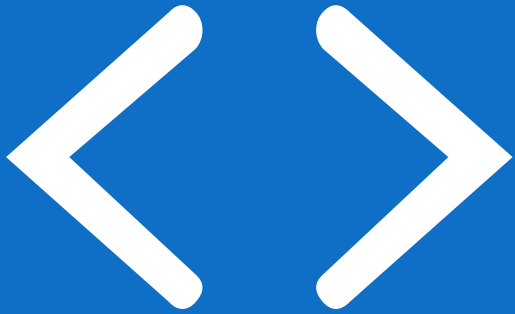


Data Manipulation



Business Insights

Who uses SQL



Software
developers

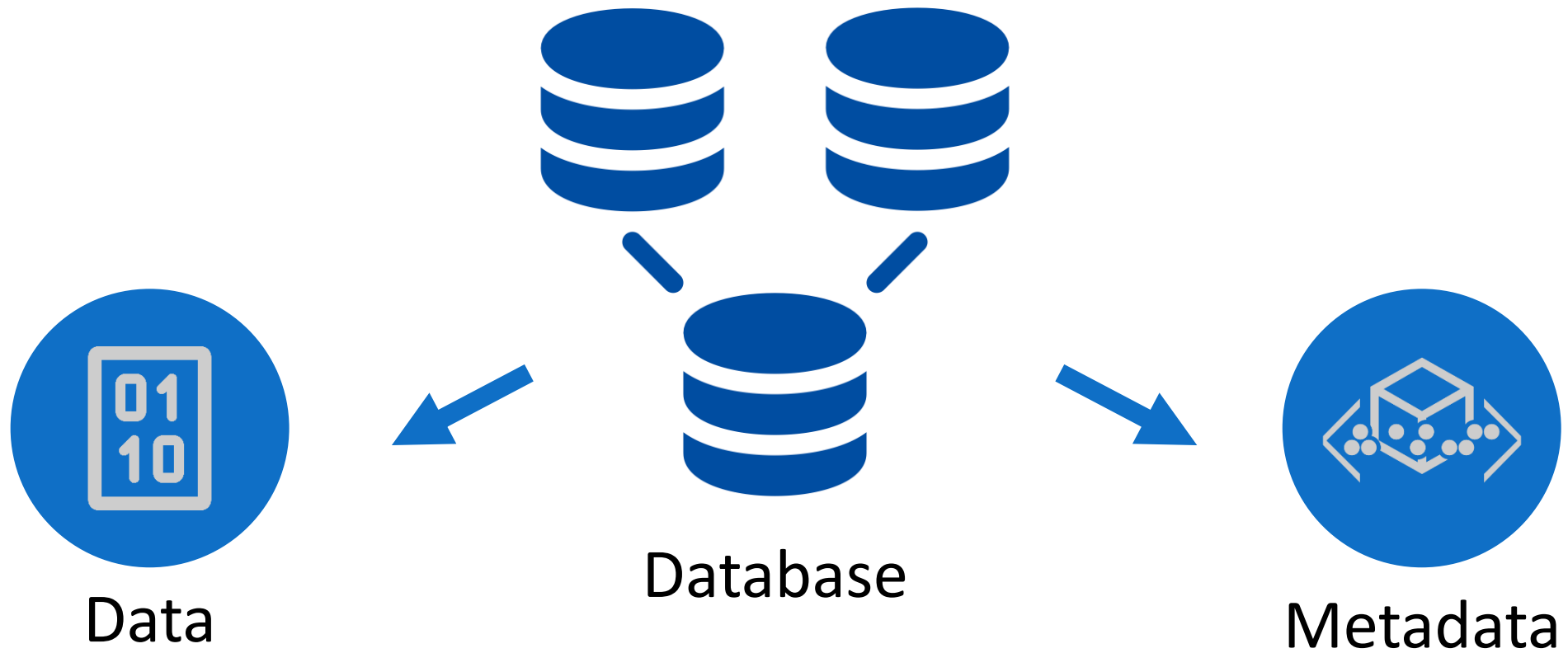


Database
Managers

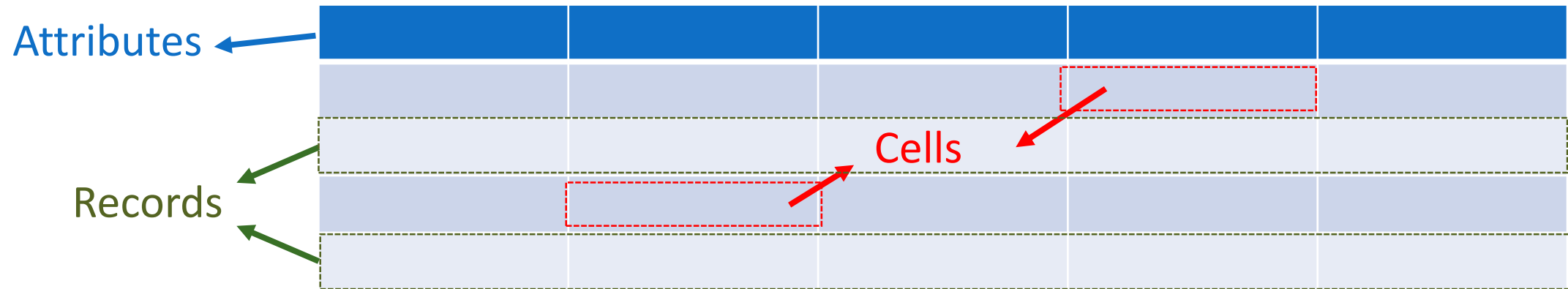


Business
Managers

Database



Tables



Example

Student	Age	Gender	Score	Rank
Student 1	18	M	89	2
Student 2	17	F	90	1
Student 3	19	M	72	3
Student 4	20	F	54	4

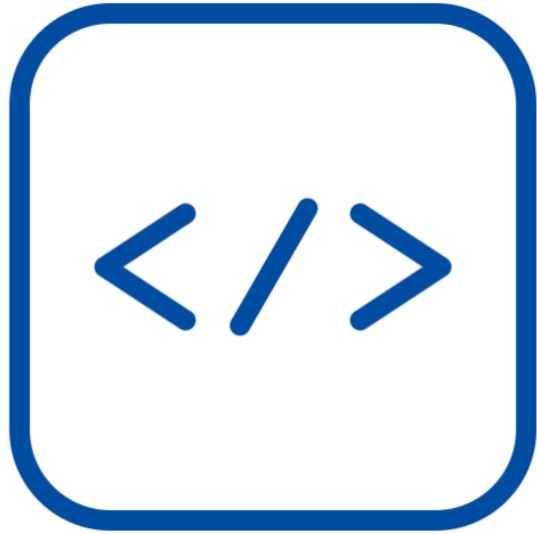
DBMS



DBMS Database Management Systems

- It allows creation of new DB and their data structures
- Allows modification of data
- Allows retrieval of Data
- Allows Storage over long period of time
- Enables recovery in times of failure
- Control access to users

SQL Queries



SQL Queries

1. DDL – Data Definition Language
CREATE, ALTER, DROP
2. DML – Data Manipulation Language
INSERT, UPDATE, DELETE
3. DQL – Data Query Language
SELECT, ORDER BY, GROUP BY
4. DCL – Data Control Language
GRANT, REVOKE
5. TCC – Transactional Control Commands
COMMIT, ROLLBACK

What is PostgreSQL

PostgreSQL

PostgreSQL is an advanced object-relational database management system that supports an extended subset of the SQL standard, including transactions, foreign keys, subqueries, triggers, user-defined types and functions.

Companies using PostgreSQL



Instagram



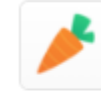
Spotify



Netflix



Uber Technologies



Instacart



reddit

Why PostgreSQL

Why PostgreSQL

- Completely Open source
- Complete ACID Compliance
- Comprehensive documentation and active discussion forums
- PostgreSQL performance is utilized best in systems requiring execution of complex queries
- PostgreSQL is best suited for Data Warehousing and data analysis applications that require fast read/write speeds
- Supported by all major cloud service providers, including Amazon, Google, & Microsoft

Create

Creating a basic table involves naming the table and defining its columns and each column's data type.

Syntax

```
CREATE TABLE      "table_name"(  
    "column 1"      "data type for column 1"      [column 1 constraint(s)],  
    "column 2"      "data type for column 2"      [column 2 constraint(s)],  
    ...  
    "column n "     [table constraint(s)] );
```

Create

Creating a basic table involves naming the table and defining its columns and each column's data type.

Constraints

- NOT NULL Constraint: Ensures that a column cannot have NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all values in a column are different.
- CHECK Constraint: Makes sure that all values in a column satisfy certain criteria.
- Primary Key Constraint: Used to uniquely identify a row in the table.
- Foreign Key Constraint: Used to ensure referential integrity of the data.

Create

Creating a basic table involves naming the table and defining its columns and each column's data type.

Keys

- A primary key is used to uniquely identify each row in a table.
- A primary key can consist of one or more columns on a table.
- When multiple columns are used as a primary key, they are called a composite key.
- A foreign key is a column (or columns) that references a column (most often the primary key) of another table.
- The purpose of the foreign key is to ensure referential integrity of the data.

Create

Creating a basic table involves naming the table and defining its columns and each column's data type.

Keys

Customer Table

Column Name	Characteristic
Cust_ID	Primary Key
Last_Name	
First_Name	

Order Table

Column Name	Characteristic
Order_ID	Primary Key
Order_Date	
Customer_SID	Foreign Key
Amount	

Create

Creating a basic table involves naming the table and defining its columns and each column's data type.

Data Types

- **Numeric** - This type of data stores numerical values. Following Data types fall in this category: Integer, Float, Real, Numeric, or Decimal.
- **Character String** - This type of data stores character values. The two common types are CHAR(n) and VARCHAR(n).
- **Date/Datetime** - This type of data allows us to store date or datetime in a database table.

Data Types

Numeric - This type of data stores numerical values.

Numeric

Name	Description	Storage Size	Range
smallint	Stores whole numbers, small range.	2 bytes	-32768 to +32767
integer	Stores whole numbers. Use this when you want to store typical integers.	4 bytes	-2147483648 to +2147483647
bigint	Stores whole numbers, large range.	8 bytes	-9223372036854775808 to 9223372036854775807
decimal	user-specified precision, exact	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point.

Data Types

Character String - This type of data stores character values. The two common types are CHAR(n) and VARCHAR(n).

Character

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

Data Types

Date/Datetime - This type of data allows us to store binary objects in a database table.

Date-Time

Name	Description	Storage Size	Low Value	High Value
timestamp	both date and time (no time zone)	8 bytes	4713 BC	294276 AD
date	date (no time of day)	4 bytes	4713 BC	5874897 AD
time	time of day (no date)	8 bytes	00:00:00	24:00:00
interval	12 bytes	time interval	-178000000 years	178000000 years

CREATE TYPE

Users can define data types as per requirements also

SYNTAX

- CREATE TYPE Dollar as DECIMAL(9,2);
- CREATE TYPE days AS ENUM ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday');

INSERT

The INSERT INTO statement is used to add new records into a database table

Syntax

```
INSERT INTO "table_name" ("column1", "column2", ...)  
VALUES ("value1", "value2", ...);
```

INSERT

The INSERT INTO statement is used to add new records into a database table

Example

- *Single row (without column names specified)*
`INSERT INTO customer_table
VALUES (1, 'bee', 'cee', 32, 'bc@xyz.com');`
- *Single row (with column names specified)*
`INSERT INTO customer_table (cust_id, first_name, age, email_id)
VALUES (2, 'dee', 22, 'd@xyz.com');`
- *Multiple rows*
`INSERT INTO customer_table
VALUES (1, 'ee', 'ef', 35, 'ef@xyz.com'),
(1, 'gee', 'eh', 42, 'gh@xyz.com'),
(1, 'eye', 'jay', 62, 'ij@xyz.com'),
(1, 'kay', 'el', , 'el@xyz.com');`

COPY

The basic syntax to import data from CSV file into a table using COPY statement is as below

Syntax

```
COPY "table_name" ("column1", "column2", ...)
FROM 'C:\tmp\persons.csv' DELIMITER ',' CSV HEADER;
```

Another option is to use PG Admin

SELECT

The SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called **result-sets**.

Syntax

```
SELECT "column_name1", "column_name2", "column_name3" FROM  
"table_name";
```

```
SELECT * FROM "table_name";
```

SELECT

The SELECT statement is used to fetch the data from a database table

Example

- Select one column
`SELECT first_name FROM customer_table;`
- Select multiple columns
`SELECT first_name, last_name FROM customer_table;`
- Select all columns
`SELECT * FROM customer_table;`

SELECT DISTINCT

The DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

Syntax

```
SELECT DISTINCT "column_name"  
FROM "table_name";
```


SELECT DISTINCT

The DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

Example

- Select one column
`SELECT DISTINCT customer_name FROM customer_table;`
- Select multiple columns
`SELECT DISTINCT customer_name, age FROM customer_table;`

WHERE

The SQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE "condition";
```

WHERE

The SQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table.

Example

- Equals to condition
`SELECT first_name FROM customer_table WHERE age = 25;`
- Less than/ Greater than condition
`SELECT first_name, age FROM customer_table WHERE age > 25;`
- Matching text condition
`SELECT * FROM customer_table WHERE first_name = "John";`

AND & OR

The SQL AND & OR operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE "simple condition"  
{ [AND|OR] "simple condition"}+;
```

AND & OR

The SQL AND & OR operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

Example

```
SELECT first_name, last_name, age  
FROM customer_table  
WHERE age>20  
AND age<30;
```

```
SELECT first_name, last_name, age  
FROM customer_table  
WHERE age<20  
OR age>30  
OR first_name = 'John';
```

NOT

NOT condition is used to negate a condition in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE NOT "simple condition"
```

NOT

NOT condition is used to negate a condition in a SELECT, INSERT, UPDATE, or DELETE statement.

Example

```
SELECT first_name,last_name, age  
FROM employee  
WHERE NOT age=25
```

```
SELECT first_name,last_name, age  
FROM employee  
WHERE NOT age=25  
AND NOT first_name = 'JAY';
```

UPDATE

The SQL UPDATE Query is used to modify the existing records in a table.

Syntax

```
UPDATE "table_name"  
SET column_1 = [value1], column_2 = [value2], ...  
WHERE "condition";
```


UPDATE

The SQL UPDATE Query is used to modify the existing records in a table.

Example

- *Single row (with column names specified)*

```
UPDATE Customer_table  
SET Age = 17, Last_name = 'Pe'  
WHERE Cust_id = 2;
```

- *Multiple rows*

```
UPDATE Customer_table  
SET email_id = 'gee@xyz.com'  
WHERE First_name = 'Gee' or First_name = 'gee';
```

DELETE

The DELETE Query is used to delete the existing records from a table.

Syntax

```
DELETE FROM "table_name"  
WHERE "condition";
```

DELETE

The DELETE Query is used to delete the existing records from a table.

Example

- *Single row*
DELETE FROM CUSTOMERS
WHERE ID = 6;
- *Multiple rows*
DELETE FROM CUSTOMERS
WHERE age>25;
- *All rows*
DELETE FROM CUSTOMERS;

ALTER

The ALTER TABLE statement is used to change the definition or structure of an existing table

Syntax

ALTER TABLE "table_name"
[Specify Actions];

Following actions can be performed

- Columns – Add, Delete (Drop), Modify or Rename
- Constraints – Add, Drop
- Index – Add, Drop

COLUMN – ADD & DROP

The basic syntax of an ALTER TABLE command to add/drop a **Column** in an existing table is as follows.

Syntax

```
ALTER TABLE "table_name"  
ADD "column_name" "Data Type";
```

```
ALTER TABLE "table_name"  
DROP "column_name";
```

COLUMN – MODIFY & RENAME

The basic syntax of an ALTER TABLE command to Modify/Rename a **Column** in an existing table is as follows.

Syntax

```
ALTER TABLE "table_name"
```

```
ALTER COLUMN "column_name" TYPE "New Data Type";
```

```
ALTER TABLE "table_name"
```

```
RENAME COLUMN "column 1" TO "column 2";
```

CONSTRAINT – ADD & DROP

The basic syntax of an ALTER TABLE command to add/drop a **Constraint** on a existing table is as follows.

Syntax

1. ALTER TABLE "table_name" ALTER COLUMN "column_name" SET NOT NULL;
2. ALTER TABLE "table_name" ALTER COLUMN "column_name" DROP NOT NULL;
3. ALTER TABLE "table_name" ADD CONSTRAINT "column_name" CHECK ("column_name">=100);
4. ALTER TABLE "table_name" ADD PRIMARY KEY ("column_name");
5. ALTER TABLE "child_table" ADD CONSTRAINT "child_column" FOREIGN KEY ("parent column") REFERENCES "parent table";

Issues while restoration

**pg_restore.exe
not found**



Most probable cause:

Location of pg_restore.exe file is different than the location available in pgAdmin

Possible solution:

Locate the file on your PC and update this address in pgAdmin

Issues while restoration

**Failed
(exit code: 1)**



Failed (exit code: 1).

1. Create a new DB for restoring the data and avoid using old Database for restoring.
2. Try out other TAR files.
3. Try using the “.BACKUP” extension file
4. Refresh DB once and check if tables are created even if it shows failure
5. Last option: Use the CSV files to create each individual

Issues while restoration

CSV files

Create customer table

```
create table customer (  
customer_id varchar primary key,  
customer_name varchar,  
segment varchar,  
age int,  
country varchar,  
city varchar,  
state varchar,  
postal_code bigint,  
region varchar);
```

Copy data from the customer CSV file

```
copy customer from 'C:\Program Files\PostgreSQL\12\data\dataset\Customer.csv'  
CSV header ;
```

Issues while restoration

CSV files

Create product table

```
create table product (  
product_id varchar primary key,  
category varchar,  
sub_category varchar,  
product_name varchar);
```

Copy data from product CSV file

```
copy product from 'C:\Program Files\PostgreSQL\12\data\dataset\Product.csv'  
CSV header ;
```

Issues while restoration

CSV files

Create Sales table

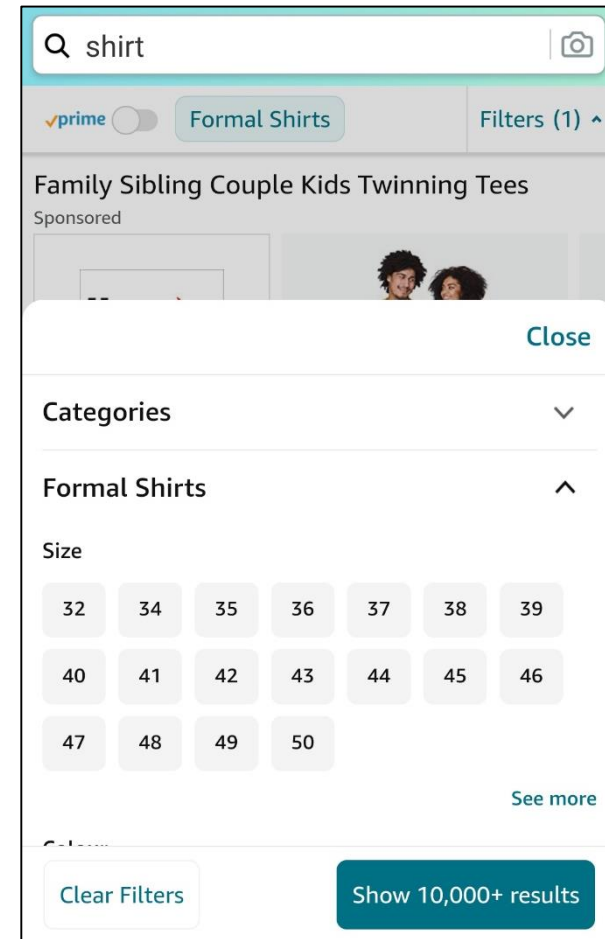
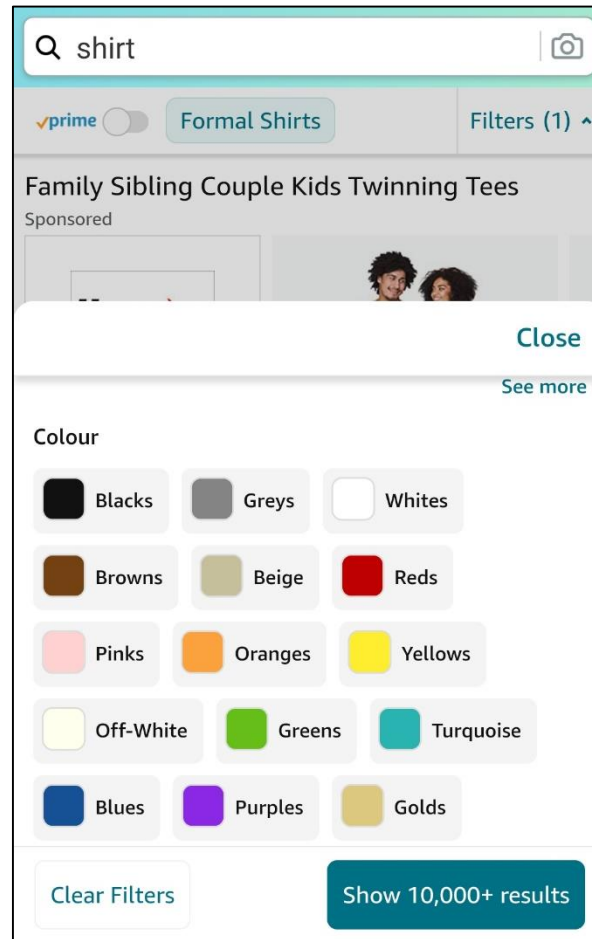
```
create table sales (  
order_line int primary key,  
order_id varchar,  
order_date date,  
ship_date date,  
ship_mode varchar,  
customer_id varchar,  
product_id varchar,  
sales numeric,  
quantity int,  
discount numeric,  
profit numeric);
```

Copy data from Sales CSV file

```
copy sales from 'C:\Program Files\PostgreSQL\12\data\dataset\Sales.csv' CSV header ;
```

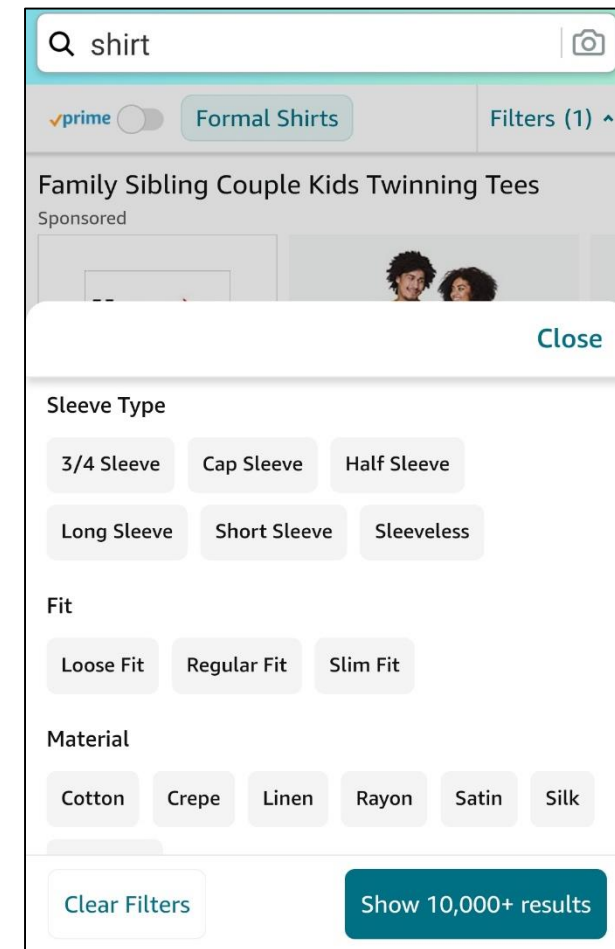
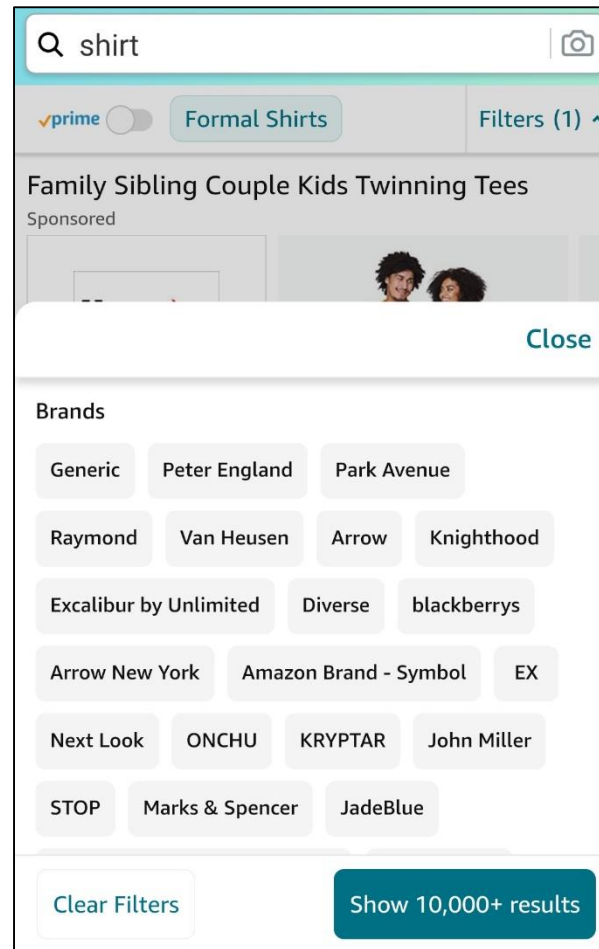
Filtering

Examples



Filtering


Examples




Filtering

Examples


Q refrig



Whirlpool 190 L 3 Star Direct-Cool Single Door Refrigerator (WDE 205 CLS 3S, Blue)
★★★★☆ 4,177
Limited time deal
₹12,190 ₹15,400 Save ₹3,210 (21%)
Save ₹500 with coupon
✓prime FREE delivery by Mon, 10 May, 7:00 am - 9:00 pm



Haier 195 L 4 Star Direct-Cool Single-Door Refrigerator (HED- 20CFDS, Dazzle Steel)
★★★★☆ 1,795
₹13,440 ₹18,400 Save ₹4,960 (27%)
...
✓prime FREE delivery by Mon, 10 May, 7:00 am - 9:00 pm



Samsung 244 L 2 Star Inverter Frost-Free Double Door Refrigerator...
★★★★☆ 15

Sorting

Examples

Customer Name	Age
Brosina Hoffman	20
Andrew Allen	50
Irene Maddox	66
Harold Pawlan	20
Pete Kriz	46
Alejandro Grove	18
Zuschuss Donatelli	66
Ken Black	67
Sandra Flanagan	41
Emily Burns	34

Customer Name	Age
Alejandro Grove	18
Andrew Allen	50
Brosina Hoffman	20
Emily Burns	34
Harold Pawlan	20
Irene Maddox	66
Ken Black	67
Pete Kriz	46
Sandra Flanagan	41
Zuschuss Donatelli	66

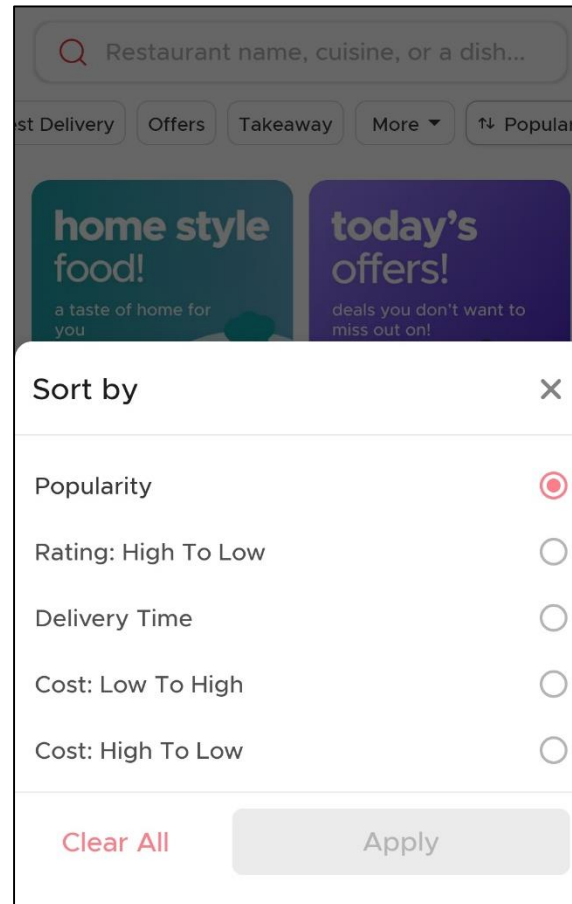
Customer Name	Age
Zuschuss Donatelli	66
Sandra Flanagan	41
Pete Kriz	46
Ken Black	67
Irene Maddox	66
Harold Pawlan	20
Emily Burns	34
Brosina Hoffman	20
Andrew Allen	50
Alejandro Grove	18

Customer Name	Age
Alejandro Grove	18
Harold Pawlan	20
Brosina Hoffman	20
Emily Burns	34
Sandra Flanagan	41
Pete Kriz	46
Andrew Allen	50
Zuschuss Donatelli	66
Irene Maddox	66
Ken Black	67

Customer Name	Age
Ken Black	67
Zuschuss Donatelli	66
Irene Maddox	66
Andrew Allen	50
Pete Kriz	46
Sandra Flanagan	41
Emily Burns	34
Harold Pawlan	20
Brosina Hoffman	20
Alejandro Grove	18

Sorting

Examples



Aggregation

Raw Data

Order Line	Order ID	Customer ID	Product ID	Sales	Quantity
1	CA-2016-152156	CG-12520	FUR-BO-10001798	261.96	2
2	CA-2016-152156	CG-12520	FUR-CH-10000454	731.94	3
3	CA-2016-138688	DV-13045	OFF-LA-10000240	14.62	2
4	US-2015-108966	SO-20335	FUR-TA-10000577	957.5775	5
5	US-2015-108966	SO-20335	OFF-ST-10000760	22.368	2
6	CA-2014-115812	BH-11710	FUR-FU-10001487	48.86	7
7	CA-2014-115812	BH-11710	OFF-AR-10002833	7.28	4
8	CA-2014-115812	BH-11710	TEC-PH-10002275	907.152	6
9	CA-2014-115812	BH-11710	OFF-BI-10003910	18.504	3
10	CA-2014-115812	BH-11710	OFF-AP-10002892	114.9	5

Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
CG-12520	Claire Gute	Consumer	67	United States	Henderson	Kentucky	42420	South
DV-13045	Darrin Van Huff	Corporate	31	United States	Los Angeles	California	90036	West
SO-20335	Sean O'Donnell	Consumer	65	United States	Fort Lauderdale	Florida	33311	South
BH-11710	Brosina Hoffman	Consumer	20	United States	Los Angeles	California	90032	West
AA-10480	Andrew Allen	Consumer	50	United States	Concord	North Carolina	28027	South
IM-15070	Irene Maddox	Consumer	66	United States	Seattle	Washington	98103	West
HP-14815	Harold Pawlan	Home Office	20	United States	Fort Worth	Texas	76106	Central
PK-19075	Pete Kriz	Consumer	46	United States	Madison	Wisconsin	53711	Central
AG-10270	Alejandro Grove	Consumer	18	United States	West Jordan	Utah	84084	West
ZD-21925	Zuschuss Donatelli	Consumer	66	United States	San Francisco	California	94109	West

Aggregation

Aggregated Data

Product ID	Number of Products sold
TEC-AC-10003832	75
OFF-PA-10001970	70
OFF-BI-10001524	67
OFF-BI-10002026	64
FUR-CH-10002647	64
FUR-TA-10001095	61
TEC-AC-10002049	60
OFF-BI-10004728	59
FUR-CH-10003774	59
TEC-AC-10003038	57

State Name	Number of customers
California	1998
New York	1110
Texas	917
Pennsylvania	641
Illinois	574
Ohio	480
Washington	460
North Carolina	343
Arizona	301
Florida	290

IN

IN condition is used to help reduce the need to use multiple OR conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE "column_name" IN ('value1', 'value2', ...);
```

IN

IN condition is used to help reduce the need to use multiple OR conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

Example

```
SELECT *  
FROM customer  
WHERE city IN ('Philadelphia', 'Seattle')
```

```
SELECT *  
FROM customer  
WHERE city = 'Philadelphia' OR city = 'Seattle';
```

BETWEEN

The BETWEEN condition is used to retrieve values within a range in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE "column_name" BETWEEN 'value1' AND 'value2';
```

BETWEEN

The BETWEEN condition is used to retrieve values within a range in a SELECT, INSERT, UPDATE, or DELETE statement.

Example

```
SELECT * FROM customer  
WHERE age BETWEEN 20 AND 30;
```

Which is same as

```
SELECT * FROM customer  
WHERE age >= 20 AND age <= 30;
```

```
SELECT * FROM customer  
WHERE age NOT BETWEEN 20 and 30;
```

```
SELECT * FROM sales  
WHERE ship_date BETWEEN '2015-04-01' AND '2016-04-01';
```

LIKE

The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
WHERE "column_name" LIKE {PATTERN};
```

{PATTERN} often consists of wildcards

WILDCARDS

The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

Example

Wildcard	Explanation
%	Allows you to match any string of any length (including zero length)
_	Allows you to match on a single character

A% means starts with A like ABC or ABCDE

%A means anything that ends with A

A%B means starts with A but ends with B

AB_C means string starts with AB, then there is one character, then there is C

LIKE

The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

Example

```
SELECT * FROM customer_table  
WHERE first_name LIKE 'Jo%';
```

```
SELECT * FROM customer_table  
WHERE first_name LIKE '%od%';
```

```
SELECT first_name, last_name FROM customer_table  
WHERE first_name LIKE 'Jas_n';
```

```
SELECT first_name, last_name FROM customer_table  
WHERE last_name NOT LIKE 'J%';
```

```
SELECT * FROM customer_table  
WHERE last_name LIKE 'G\%';
```

ORDER BY

The ORDER BY clause is used to sort the records in result set. It can only be used in SELECT statements.

Syntax

```
SELECT "column_name"  
FROM "table_name"  
[WHERE "condition"]  
ORDER BY "column_name" [ASC, DESC];
```

It is possible to order by more than one column.

```
ORDER BY "column_name1" [ASC, DESC], "column_name2" [ASC, DESC]
```

ORDER BY

The ORDER BY clause is used to sort the records in result set. It can only be used in SELECT statements.

Example

```
SELECT * FROM customer  
WHERE state = 'California' ORDER BY Customer_name;
```

Same as

```
SELECT * FROM customer  
WHERE state = 'California' ORDER BY Customer_name ASC;
```

```
SELECT * FROM customer  
ORDER BY 2 DESC;
```

```
SELECT * FROM customer  
WHERE age>25 ORDER BY City ASC, Customer_name DESC;
```

```
SELECT * FROM customer  
ORDER BY age;
```

LIMIT

LIMIT statement is used to limit the number of records returned based on a limit value.

Syntax

```
SELECT "column_names"  
FROM "table_name"  
[WHERE conditions]  
[ORDER BY expression [ ASC | DESC ]]  
LIMIT row_count;
```

LIMIT

LIMIT statement is used to limit the number of records returned based on a limit value.

Example

```
SELECT * FROM customer  
WHERE age >= 25  
ORDER BY age DESC  
LIMIT 8;
```

```
SELECT * FROM customer  
WHERE age >=25  
ORDER BY age ASC  
LIMIT 10;
```

AS

The keyword **AS** is used to assign an alias to the column or a table. It is inserted between the column name and the column alias or between the table name and the table alias.

Syntax

```
SELECT column_name" AS "column_alias"  
FROM "table_name";
```

AS

The keyword **AS** is used to assign an alias to the column or a table. It is inserted between the column name and the column alias or between the table name and the table alias.

Example

```
SELECT Cust_id AS "Serial number", Customer_name as name, Age as  
Customer_age  
FROM Customer ;
```


COUNT

Count function returns the count of an expression

Syntax

```
SELECT "column_name1", COUNT ("column_name2")  
FROM "table_name"
```

COUNT

Count function returns the count of an expression

Example

```
SELECT COUNT(*) FROM sales;
```

```
SELECT COUNT (order_line) as "Number of Products Ordered",  
COUNT (DISTINCT order_id) AS "Number of Orders"  
FROM sales WHERE customer_id = 'CG-12520';
```

SUM

Sum function returns the summed value of an expression

Syntax

```
SELECT sum(aggregate_expression)  
FROM tables  
[WHERE conditions];
```

SUM

Sum function returns the summed value of an expression

Example

```
SELECT sum(Profit) AS "Total Profit"  
FROM sales;
```

```
SELECT sum(quantity) AS "Total Quantity"  
FROM orders where product_id = 'FUR-TA-10000577';
```

AVERAGE

AVG function returns the average value of an expression.

Syntax

```
SELECT avg(aggregate_expression)
FROM tables
[WHERE conditions];
```

AVERAGE

AVG function returns the average value of an expression.

Example

```
SELECT avg(age) AS "Average Customer Age"  
FROM customer;
```

```
SELECT avg(sales * 0.10) AS "Average Commission Value"  
FROM sales;
```

MIN/MAX

MIN/MAX function returns the minimum/maximum value of an expression.

Syntax

```
SELECT min(aggregate_expression)
FROM tables
[WHERE conditions];
```

```
SELECT max(aggregate_expression)
FROM tables
[WHERE conditions];
```

MIN/MAX

MIN/MAX function returns the minimum/maximum value of an expression.

Example

```
SELECT MIN(sales) AS Min_sales_June15  
FROM sales  
WHERE order_date BETWEEN '2015-06-01' AND '2015-06-30';
```

```
SELECT MAX(sales) AS Min_sales_June15  
FROM sales  
WHERE order_date BETWEEN '2015-06-01' AND '2015-06-30';
```


GROUP BY

GROUP BY clause is used in a SELECT statement to group the results by one or more columns.

Syntax

```
SELECT "column_name1", "function type" ("column_name2")  
FROM "table_name"  
GROUP BY "column_name1";
```

GROUP BY

GROUP BY clause is used in a SELECT statement to group the results by one or more columns.

Example

```
SELECT region, COUNT (customer_id) AS customer_count  
FROM customer GROUP BY region;
```

```
SELECT product_id, SUM (quantity) AS quantity_sold FROM sales  
GROUP BY product_id ORDER BY quantity_sold DESC;
```

```
SELECT customer_id,  
MIN(sales) AS min_sales,  
MAX(sales) AS max_sales,  
AVG(sales) AS Average_sales,  
SUM(sales) AS Total_sales  
FROM sales  
GROUP BY customer_id  
ORDER BY total_sales DESC  
LIMIT 5;
```

HAVING

HAVING clause is used in combination with the GROUP BY clause to restrict the groups of returned rows to only those whose the condition is TRUE

Syntax

```
SELECT ColumnNames, aggregate_function (expression)
FROM tables
[WHERE conditions]
GROUP BY column1
HAVING condition;
```

HAVING

HAVING clause is used in combination with the GROUP BY clause to restrict the groups of returned rows to only those whose the condition is TRUE

Example

```
SELECT region, COUNT(customer_id) AS customer_count  
FROM customer  
GROUP BY region  
HAVING COUNT(customer_id) > 200 ;
```

CASE

The CASE expression is a conditional expression, similar to if/else statements

Syntax

```
CASE WHEN condition THEN result  
  [WHEN ...]  
  [ELSE result]  
END
```

```
CASE expression  
  WHEN value THEN result  
  [WHEN ...]  
  [ELSE result]  
END
```

CASE

The CASE expression is a conditional expression, similar to if/else statements

Example

```
SELECT *,  
    CASE WHEN age<30 THEN 'Young'  
        WHEN age>60 THEN 'Senior Citizen'  
        ELSE 'Middle aged'  
    END AS Age_category  
FROM customer;
```

Module – Joining Data

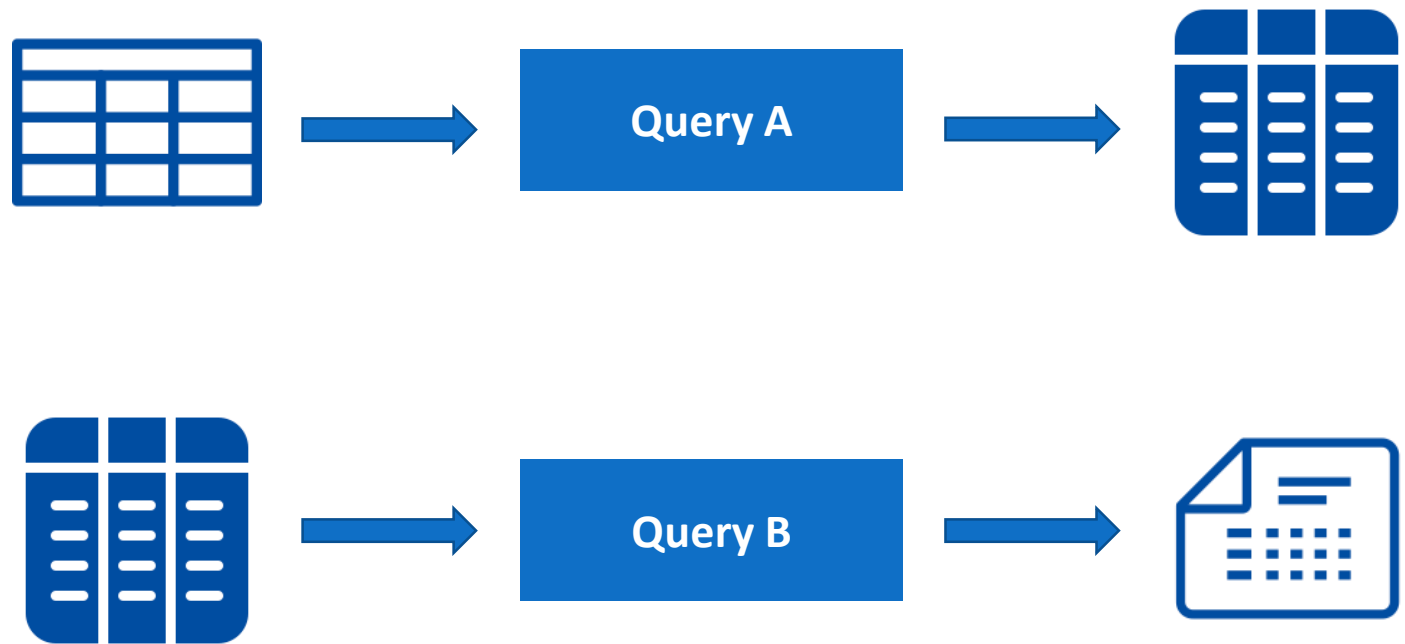
Agenda

Category	No. of products sold
Furniture	8028
Office Supplies	22906
Technology	6939



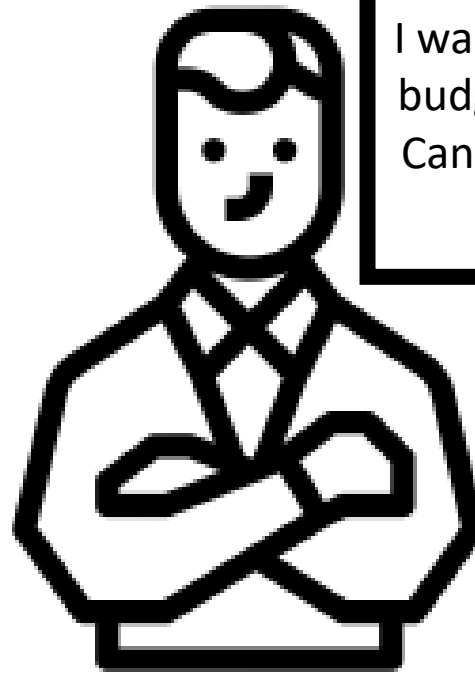
Subqueries

Agenda



Joins

Why



I want to allocate marketing budget to different regions.
Can you tell me the region-wise total sales?

Marketing Head

Joins

Why

Sales Table

Sales values

Customer Table

Region values

Order Line	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Product ID	Sales	Quantity	Discount	Profit
1	CA-2016-152156	08-11-2016	11-11-2016	Second Class	CG-12520	FUR-BO-10001798	261.96	2	0	41.9136
2	CA-2016-152156	08-11-2016	11-11-2016	Second Class	CG-12520	FUR-CH-10000454	731.94	3	0	219.582
3	CA-2016-138688	12-06-2016	16-06-2016	Second Class	DV-13045	OFF-LA-10000240	14.62	2	0	6.8714
4	US-2015-108966	11-10-2015	18-10-2015	Standard Class	SO-20335	FUR-TA-10000577	957.5775	5	0.45	-383.031
5	US-2015-108966	11-10-2015	18-10-2015	Standard Class	SO-20335	OFF-ST-10000760	22.368	2	0.2	2.5164

Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
CG-12520	Claire Gute	Consumer	67	United States	Henderson	Kentucky	42420	South
DV-13045	Darrin Van Huff	Corporate	31	United States	Los Angeles	California	90036	West
SO-20335	Sean O'Donnell	Consumer	65	United States	Fort Lauderdale	Florida	33311	South
BH-11710	Brosina Hoffman	Consumer	20	United States	Los Angeles	California	90032	West

Joins

To join tables we must know:

1. The names of the tables to be joined
2. The common column based on which we will join them
3. The list of columns from each table

What's needed

Order Line	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Product ID	Sales	Quantity	Discount	Profit
1	CA-2016-152156	08-11-2016	11-11-2016	Second Class	CG-12520	FUR-BO-10001798	261.96	2	0	41.9136
2	CA-2016-152156	08-11-2016	11-11-2016	Second Class	CG-12520	FUR-CH-10000454	731.94	3	0	219.582
3	CA-2016-138688	12-06-2016	16-06-2016	Second Class	DV-13045	OFF-LA-10000240	14.62	2	0	6.8714
4	US-2015-108966	11-10-2015	18-10-2015	Standard Class	SO-20335	FUR-TA-10000577	957.5775	5	0.45	-383.031
5	US-2015-108966	11-10-2015	18-10-2015	Standard Class	SO-20335	OFF-ST-10000760	22.368	2	0.2	2.5164

Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
CG-12520	Claire Gute	Consumer	67	United States	Henderson	Kentucky	42420	South
DV-13045	Darrin Van Huff	Corporate	31	United States	Los Angeles	California	90036	West
SO-20335	Sean O'Donnell	Consumer	65	United States	Fort Lauderdale	Florida	33311	South
BH-11710	Brosina Hoffman	Consumer	20	United States	Los Angeles	California	90032	West

Joins

Types of Joins

Order Line	Order ID	Order Date	Customer ID	Product ID	Sales
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96
88	CA-2017-155558	26-10-2017	PG-18895	OFF-LA-10000134	6.16
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62
5	US-2015-108966	11-10-2015	SO-20335	OFF-ST-10000760	22.368

Sales table

Customer ID	Customer Name	State	Region
CG-12520	Claire Gute	Kentucky	South
DV-13045	Darrin Van Huff	California	West
SO-20335	Sean O'Donnell	Florida	South
BH-11710	Brosina Hoffman	California	West

Customer table

Following points must be noted about customer ID

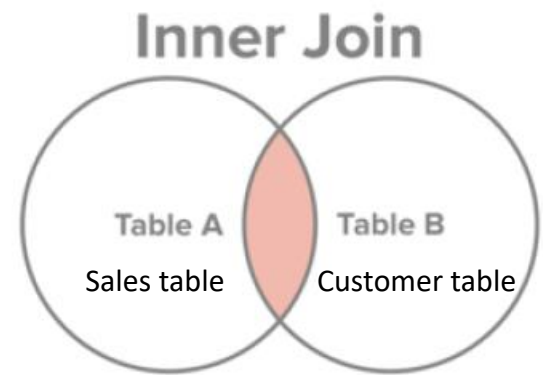
1. CG-12520, DV-13045 and SO-20335 are present in both tables
2. PG-18895 is present in sales table but not in customer table
3. BH-11710 is present in customer table and not in sales table

Joins

Types of Joins

Observations

1. CG-12520, DV-13045 and SO-20335 are present in both tables
2. PG-18895 is present in sales table but not in customer table
3. BH-11710 is present in customer table and not in sales table



Option 1 – Inner Join

- Result contains only those customer IDs which are present in both the tables

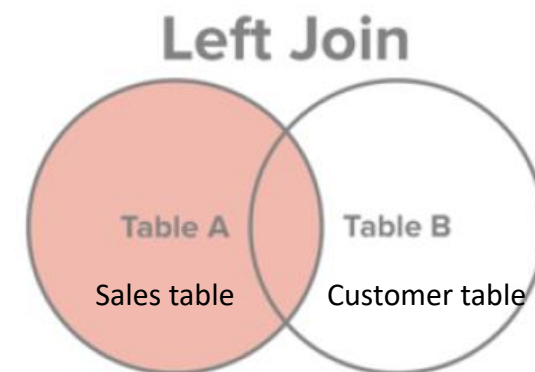
Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Customer Name	State	Region
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96	Claire Gute	Kentucky	South
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62	Darrin Van Huff	California	West
5	US-2015-108966	11-10-2015	SO-20335	OFF-ST-10000760	22.368	Sean O'Donnell	Florida	South

Joins

Types of Joins

Observations

1. CG-12520, DV-13045 and SO-20335 are present in both tables
2. PG-18895 is present in sales table but not in customer table
3. BH-11710 is present in customer table and not in sales table



Option 2 – Left Join

- Result contains all customer IDs which are present in the sales table

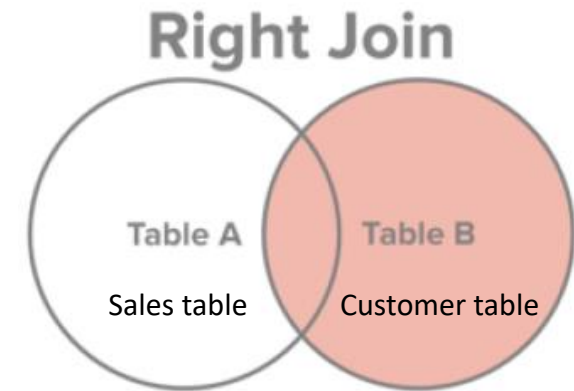
Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Customer Name	State	Region
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96	Claire Gute	Kentucky	South
88	CA-2017-155558	26-10-2017	PG-18895	OFF-LA-10000134	6.16	Null	Null	Null
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62	Darrin Van Huff	California	West
5	US-2015-108966	11-10-2015	SO-20335	OFF-ST-10000760	22.368	Sean O'Donnell	Florida	South

Joins

Types of Joins

Observations

1. CG-12520, DV-13045 and SO-20335 are present in both tables
2. PG-18895 is present in sales table but not in customer table
3. BH-11710 is present in customer table and not in sales table



Option 3 – Right Join

- Result contains all customer IDs which are present in the customer table

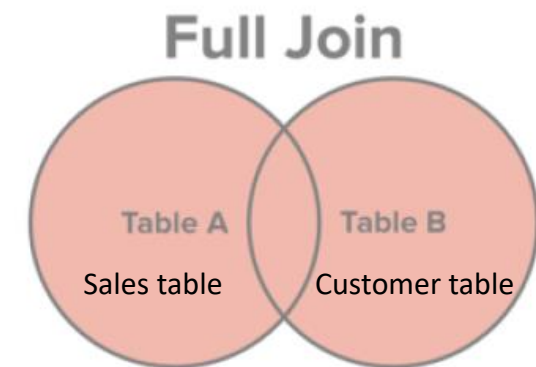
Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Customer Name	State	Region
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96	Claire Gute	Kentucky	South
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62	Darrin Van Huff	California	West
5	US-2015-108966	11-10-2015	SO-20335	OFF-ST-10000760	22.368	Sean O'Donnell	Florida	South
Null	Null	Null	BH-11710	Null	Null	Brosina Hoffman	California	West

Joins

Types of Joins

Observations

1. CG-12520, DV-13045 and SO-20335 are present in both tables
2. PG-18895 is present in sales table but not in customer table
3. BH-11710 is present in customer table and not in sales table



Option 4 – Full Outer Join

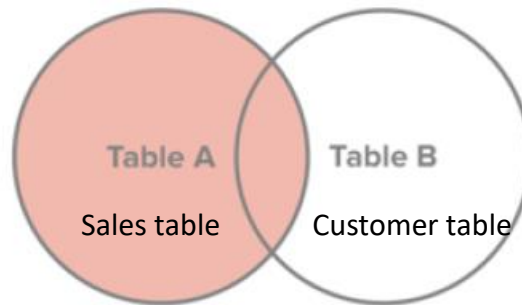
- Result contains all the customer IDs

Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Customer Name	State	Region
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96	Claire Gute	Kentucky	South
88	CA-2017-155558	26-10-2017	PG-18895	OFF-LA-10000134	6.16	Null	Null	Null
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62	Darrin Van Huff	California	West
5	US-2015-108966	11-10-2015	SO-20335	OFF-ST-10000760	22.368	Sean O'Donnell	Florida	South
Null	Null	Null	BH-11710	Null	Null	Brosina Hoffman	California	West

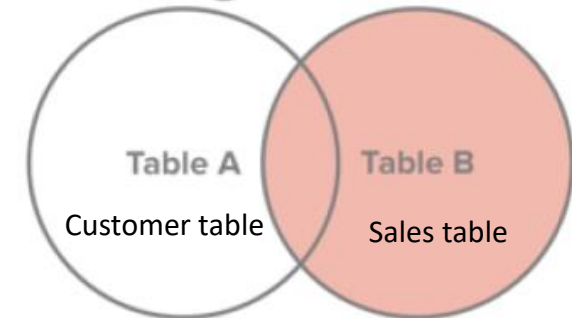
Joins

Types of Joins

Left Join



Right Join



Q. Are these two same?

A. Yes

Combining queries

What

Combining similar data using operators like Union, Intersect and Except

Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
EB-13870	Emily Burns	Consumer	34	United States	Orem	Utah	84057	West
EH-13945	Eric Hoffmann	Consumer	21	United States	Los Angeles	California	90049	West
TB-21520	Tracy Blumstein	Consumer	48	United States	Philadelphia	Pennsylvania	19140	East
MA-17560	Matt Abelman	Home Office	19	United States	Houston	Texas	77095	Central



Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
ON-18715	Odella Nelson	Corporate	27	United States	Eagan	Minnesota	55122	Central
PO-18865	Patrick O'Donnell	Consumer	64	United States	Westland	Michigan	48185	Central
LH-16900	Lena Hernandez	Consumer	66	United States	Dover	Delaware	19901	East



Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
EB-13870	Emily Burns	Consumer	34	United States	Orem	Utah	84057	West
EH-13945	Eric Hoffmann	Consumer	21	United States	Los Angeles	California	90049	West
TB-21520	Tracy Blumstein	Consumer	48	United States	Philadelphia	Pennsylvania	19140	East
MA-17560	Matt Abelman	Home Office	19	United States	Houston	Texas	77095	Central
ON-18715	Odella Nelson	Corporate	27	United States	Eagan	Minnesota	55122	Central
PO-18865	Patrick O'Donnell	Consumer	64	United States	Westland	Michigan	48185	Central
LH-16900	Lena Hernandez	Consumer	66	United States	Dover	Delaware	19901	East

JOINS

JOINS are used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement.

TYPES

- INNER JOIN (or sometimes called simple join)
- LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- FULL OUTER JOIN (or sometimes called FULL JOIN)
- CROSS JOIN (or sometimes called CARTESIAN JOIN)

JOINS

Query for creating dataset

```
/*Creating sales table of year 2015*/
```

```
Create table sales_2015 as select * from sales where ship_date between '2015-01-01' and '2015-12-31';
```

```
select count(*) from sales_2015; --2131
```

```
select count(distinct customer_id) from sales_2015;--578
```

```
/* Customers with age between 20 and 60 */
```

```
create table customer_20_60 as select * from customer where age between 20 and 60;
```

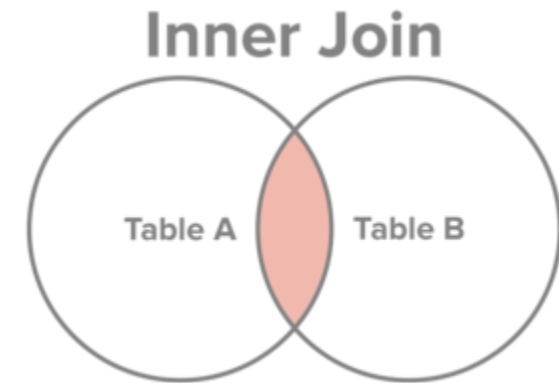
```
select count (*) from customer_20_60;--597
```

INNER JOIN

INNER JOIN compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

```
SELECT columns  
FROM table1  
INNER JOIN table2  
ON table1.column = table2.column;
```

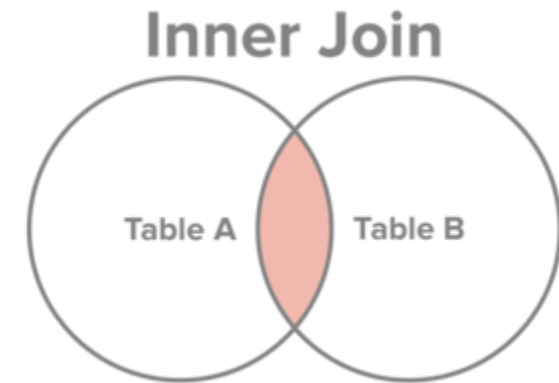


INNER JOIN

INNER JOIN compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age
FROM sales_2015 AS a
INNER JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```



JOINS

JOINS are used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement.

TYPES

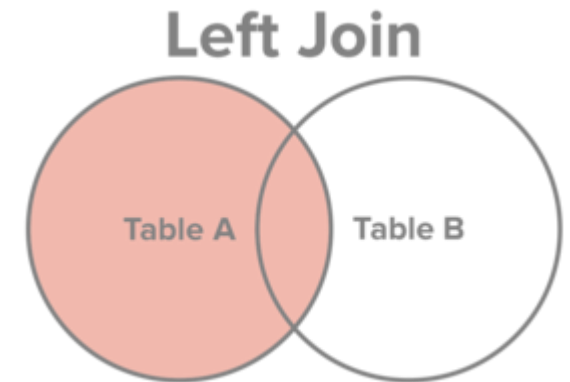
- INNER JOIN (or sometimes called simple join)
- LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- FULL OUTER JOIN (or sometimes called FULL JOIN)
- CROSS JOIN (or sometimes called CARTESIAN JOIN)

LEFT JOIN

The **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table.

Syntax

```
SELECT table1.column1, table2.column2...  
FROM table1  
LEFT JOIN table2  
ON table1.common_field = table2.common_field;
```

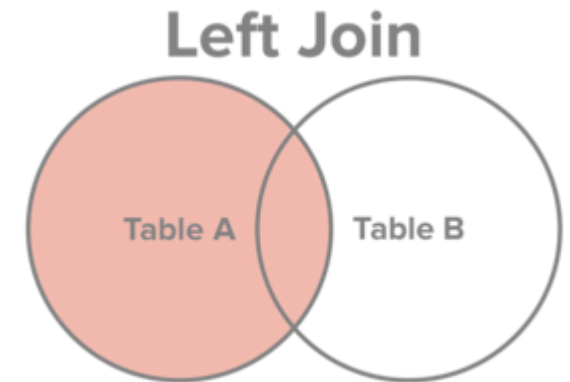


LEFT JOIN

The **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table.

Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age
FROM sales_2015 AS a
LEFT JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```

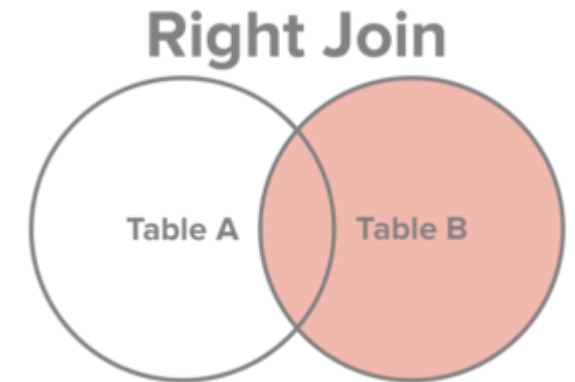


RIGHT JOIN

The **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table.

Syntax

```
SELECT table1.column1, table2.column2...  
FROM table1  
RIGHT JOIN table2  
ON table1.common_field = table2.common_field;
```

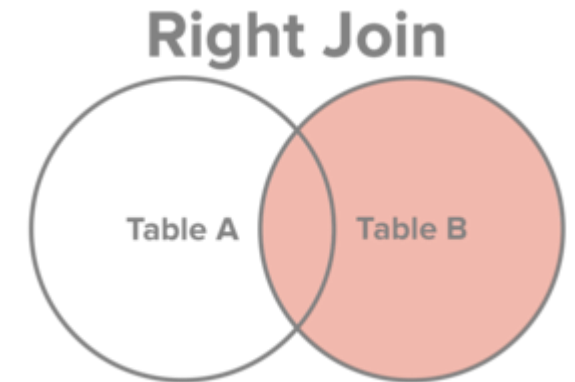


RIGHT JOIN

The **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table.

Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age
FROM sales_2015 AS a
RIGHT JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```

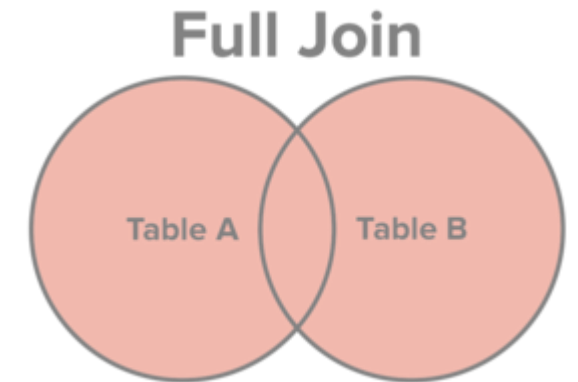


FULL OUTER JOIN

The **FULL JOIN** combines the results of both left and right outer joins

Syntax

```
SELECT table1.column1, table2.column2...  
FROM table1  
FULL JOIN table2  
ON table1.common_field = table2.common_field;
```

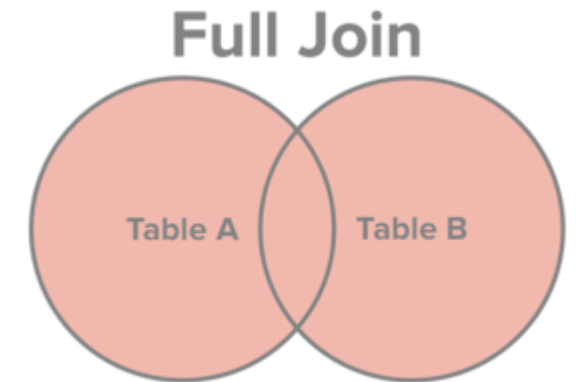


FULL OUTER JOIN

The **FULL JOIN** combines the results of both left and right outer joins

Example

```
SELECT
    a.order_line ,
    a.product_id,
    a.customer_id,
    a.sales,
    b.customer_name,
    b.age,
    b.customer_id
FROM sales_2015 AS a
FULL JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY a.customer_id , b.customer_id;
```



CROSS JOIN

The Cross Join creates a cartesian product between two sets of data.

Syntax

```
SELECT table1.column1, table2.column2...  
FROM table1, table2 [, table3 ]
```

CROSS JOIN

The Cross Join creates a cartesian product between two sets of data.

Example

```
SELECT a.YYYY, b.MM  
FROM year_values AS a, month_values AS b  
ORDER BY a.YYYY, b.MM;
```

Combining queries

What

Combining similar data using operators like Union, Intersect and Except

Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
EB-13870	Emily Burns	Consumer	34	United States	Orem	Utah	84057	West
EH-13945	Eric Hoffmann	Consumer	21	United States	Los Angeles	California	90049	West
TB-21520	Tracy Blumstein	Consumer	48	United States	Philadelphia	Pennsylvania	19140	East
MA-17560	Matt Abelman	Home Office	19	United States	Houston	Texas	77095	Central



Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
ON-18715	Odella Nelson	Corporate	27	United States	Eagan	Minnesota	55122	Central
PO-18865	Patrick O'Donnell	Consumer	64	United States	Westland	Michigan	48185	Central
LH-16900	Lena Hernandez	Consumer	66	United States	Dover	Delaware	19901	East



Customer ID	Customer Name	Segment	Age	Country	City	State	Postal Code	Region
EB-13870	Emily Burns	Consumer	34	United States	Orem	Utah	84057	West
EH-13945	Eric Hoffmann	Consumer	21	United States	Los Angeles	California	90049	West
TB-21520	Tracy Blumstein	Consumer	48	United States	Philadelphia	Pennsylvania	19140	East
MA-17560	Matt Abelman	Home Office	19	United States	Houston	Texas	77095	Central
ON-18715	Odella Nelson	Corporate	27	United States	Eagan	Minnesota	55122	Central
PO-18865	Patrick O'Donnell	Consumer	64	United States	Westland	Michigan	48185	Central
LH-16900	Lena Hernandez	Consumer	66	United States	Dover	Delaware	19901	East

Combining Queries

Combining queries are used to combine the results of two SELECT queries

What

SELECT column A, column B FROM Table X

SELECT column A, column B FROM Table Y

Note: Structure of the output from the two queries should be same

Combining Queries

Combining queries are used to combine the results of two SELECT queries

Example

Customer Name	Age
Claire Gute	67
Darrin Van Huff	31
Sean O'Donnell	65
Brosina Hoffman	20
Andrew Allen	50

Customer Name	Age
Harold Pawlan	20
Pete Kriz	46
Sean O'Donnell	65
Brosina Hoffman	20
Zuschuss Donatelli	66

Customer Name	Age
Sean O'Donnell	65
Brosina Hoffman	20

Customer Name	Age
Claire Gute	67
Darrin Van Huff	31
Andrew Allen	50

Customer Name	Age
Claire Gute	67
Darrin Van Huff	31
Sean O'Donnell	65
Brosina Hoffman	20
Andrew Allen	50
Harold Pawlan	20
Pete Kriz	46
Zuschuss Donatelli	66

Intersect

Intersect operator is used to find the common rows from the results of two SELECT queries

Syntax

```
SELECT column A, column B... FROM Table X
```

```
INTERSECT
```

```
SELECT column A, column B... FROM Table Y
```

Intersect

Intersect operator is used to find the common rows from the results of two SELECT queries

Example

```
SELECT customer_id FROM sales_2015
```

```
INTERSECT
```

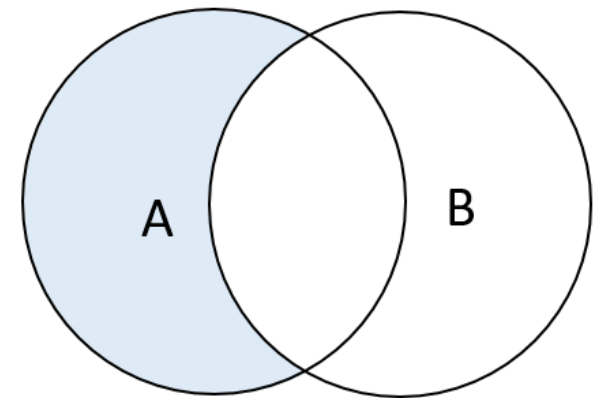
```
SELECT customer_id from customer_20_60
```

EXCEPT

EXCEPT operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.

Syntax

```
SELECT expression1, expression2, ...  
FROM tables  
[WHERE conditions]  
EXCEPT  
SELECT expression1, expression2, ...  
FROM tables  
[WHERE conditions];
```



EXCEPT

EXCEPT operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.

Example

```
SELECT customer_id  
FROM sales_2015  
EXCEPT  
SELECT customer_id  
FROM customer_20_60  
ORDER BY customer_id;
```

UNION

UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.

Syntax

Each SELECT statement within the UNION operator must have the same number of fields in the result sets with similar data types

```
SELECT expression1, expression2, ... expression_n  
FROM tables  
[WHERE conditions]  
UNION  
SELECT expression1, expression2, ... expression_n  
FROM tables  
[WHERE conditions];
```

UNION

UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.

Example

```
SELECT customer_id  
FROM sales_2015  
UNION  
SELECT customer_id  
FROM customer_20_60  
ORDER BY customer_id;
```


SUBQUERY

Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

Syntax

SYNTAX where subquery is in WHERE

```
SELECT "column_name1"  
FROM "table_name1"  
WHERE "column_name2" [Comparison Operator]  
(SELECT "column_name3"  
FROM "table_name2"  
WHERE "condition");
```

SUBQUERY

Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

Example

Subquery in WHERE

```
SELECT * FROM sales
WHERE customer_ID IN
    (SELECT DISTINCT customer_id
     FROM customer WHERE age >60);
```

SUBQUERY

Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

Example

Subquery in FROM

```
SELECT
    a.product_id ,
    a.product_name ,
    a.category,
    b.quantity
FROM product AS a
LEFT JOIN (SELECT product_id,
                  SUM(quantity) AS quantity
            FROM sales GROUP BY product_id) AS b
ON a.product_id = b.product_id
ORDER BY b.quantity desc;
```

SUBQUERY

Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

Example

```
SELECT  customer_id,  
        order_line,  
        (SELECT customer_name  
         FROM customer  
         WHERE sales.customer_id = customer.customer_id)  
FROM sales  
ORDER BY customer_id;
```

SUBQUERY

Subquery is a query within a query. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

RULES

There are a few rules that subqueries must follow –

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

VIEW

VIEW is not a physical table, it is a virtual table created by a query joining one or more tables.

Syntax

```
CREATE [OR REPLACE] VIEW view_name AS  
SELECT columns  
FROM tables  
[WHERE conditions];
```

CREATE VIEW

VIEW is not a physical table, it is a virtual table created by a query joining one or more tables.

Example

```
CREATE VIEW logistics AS
SELECT a.order_line,
       a.order_id,
       b.customer_name,
       b.city,
       b.state,
       b.country
FROM sales AS a
LEFT JOIN customer as b
ON a.customer_id = b.customer_id
ORDER BY a.order_line;
```

CREATE OR REPLACE VIEW can be used instead of just CREATE VIEW

DROP or UPDATE VIEW

VIEW is not a physical table, it is a virtual table created by a query joining one or more tables.

Example

DROP VIEW logistics;

UPDATE logistics

SET Country = US

WHERE Country = 'United States';

VIEW

NOTES

A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves. If data is changing in the underlying table, the same change is reflected in the view. A view can be built on top of a single table or multiple tables. It can also be built on top of another view. In the [SQL Create View](#) page, we will see how a view can be built.

Views offer the following advantages:

- 1. Ease of use:** A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.
- 2. Space savings:** Views takes very little space to store, since they do not store actual data.
- 3. Additional data security:** Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

VIEW

NOTES

VIEW can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

Syntax

```
CREATE [UNIQUE] INDEX index_name  
ON table_name  
  (index_col1 [ASC | DESC],  
   index_col2 [ASC | DESC],  
   ...  
   index_col_n [ASC | DESC]);
```

A simple index is an index on a single column, while a composite index is an index on two or more columns.

CREATE INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

Example

```
CREATE INDEX mon_idx  
ON month_values(MM);
```

DROP or RENAME INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

Syntax

```
DROP INDEX [IF EXISTS] index_name  
[ CASCADE | RESTRICT ];
```

```
ALTER INDEX [IF EXISTS] index_name,  
RENAME TO new_index_name;
```

DROP INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

Example

```
DROP INDEX mon_idx;
```

INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

GOOD PRACTICES

1. Build index on columns of integer type
2. Keep index as narrow as possible
3. Column order is important
4. Make sure the column you are building an index for is declared NOT NULL
5. Build an index only when necessary

INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

GOOD PRACTICES

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

SQL functions

Agenda

Functions are used to process data and improve its quality

Examples:

- Communication managers in companies want to find out the length of message they are sending to the customers

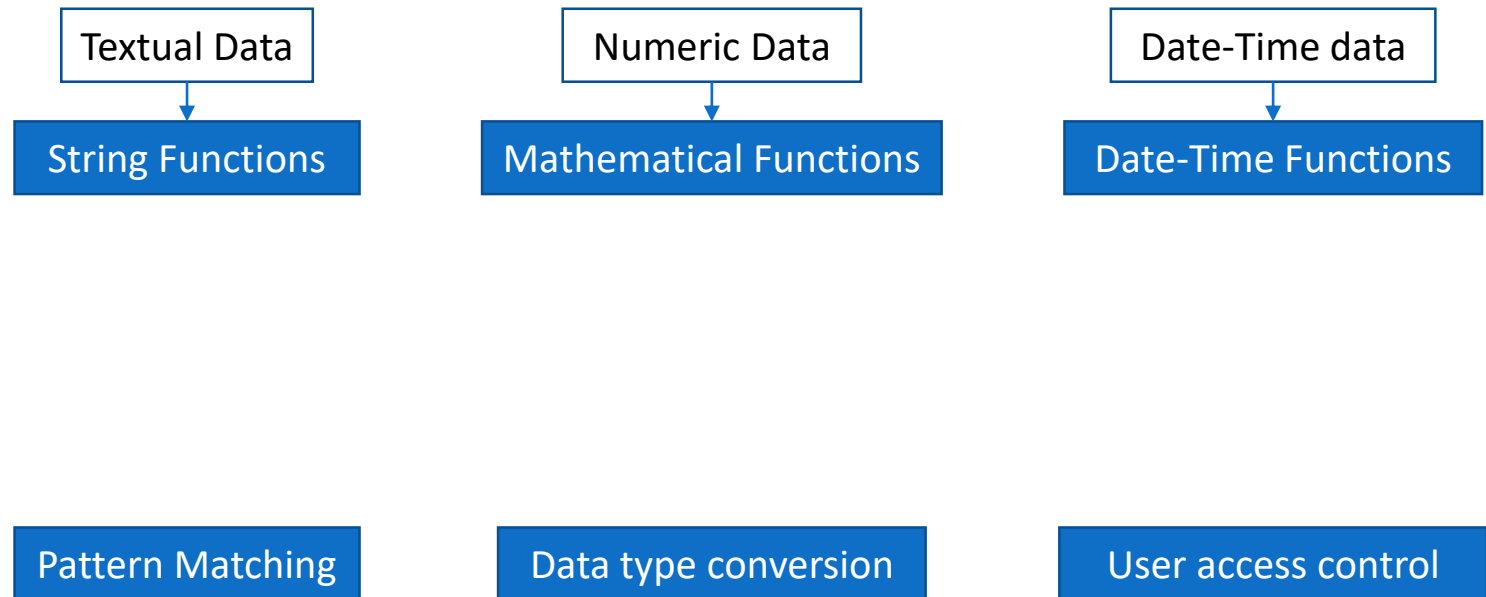
Length of “Hello There” = 11

- Supply chain manager is interested in the order processing time

Order processing time = **Age** between Order date and Ship Date

SQL functions

Classification



LENGTH

LENGTH function returns the length of the specified string, expressed as the number of characters.

Syntax

length(string)

LENGTH

LENGTH function returns the length of the specified string, expressed as the number of characters.

Example

```
SELECT Customer_name, Length (Customer_name) as characters  
FROM customer  
WHERE age >30 ;
```

UPPER & LOWER

UPPER/ LOWER function converts all characters in the specified string to uppercase/ lowercase.

Syntax

`upper(string)`

`lower(string)`

UPPER & LOWER

UPPER/ LOWER function converts all characters in the specified string to uppercase/ lowercase.

Example

```
SELECT upper('Start-Tech Academy');
```

```
SELECT lower('Start-Tech Academy');
```

REPLACE

REPLACE function replaces all occurrences of a specified string

Syntax

`replace(string, from_substring, to_substring)`

Replace function is case sensitive.

REPLACE

REPLACE function replaces all occurrences of a specified string

Example

```
SELECT
    Customer_name,
    country,
    Replace (country,'United States','US') AS country new
FROM customer;
```


TRIM,LTRIM & RTRIM

TRIM function removes all specified characters either from the beginning or the end of a string

RTRIM function removes all specified characters from the right-hand side of a string

LTRIM function removes all specified characters from the left-hand side of a string

Syntax

`trim([leading | trailing | both] [trim_character] from string)`

`rtrim(string, trim_character)`

`ltrim(string, trim_character)`

TRIM,LTRIM & RTRIM

TRIM function removes all specified characters either from the beginning or the end of a string

RTRIM function removes all specified characters from the right-hand side of a string

LTRIM function removes all specified characters from the left-hand side of a string

Example

```
SELECT trim(leading ' ' from ' Start-Tech Academy ');
```

```
SELECT trim(trailing ' ' from ' Start-Tech Academy ');
```

```
SELECT trim(both ' ' from ' Start-Tech Academy ');
```

```
SELECT trim(' Start-Tech Academy ');
```

```
SELECT rtrim(' Start-Tech Academy ', ' ');
```

```
SELECT ltrim(' Start-Tech Academy ', ' ');
```

CONCAT

|| operator allows you to concatenate 2 or more strings together

Syntax

string1 || string2 || string_n

CONCAT

|| operator allows you to concatenate 2 or more strings together

Example

```
SELECT
    Customer_name,
    city || ', ' || state || ', ' || country AS address
FROM customer;
```

SUBSTRING

SUBSTRING function allows you to extract a substring from a string

Syntax

substring(string [from start_position] [for length])

SUBSTRING

SUBSTRING function allows you to extract a substring from a string

Example

```
SELECT
    Customer_id,
    Customer_name,
    SUBSTRING (Customer_id FOR 2) AS cust_group
FROM customer
WHERE SUBSTRING(Customer_id FOR 2) = 'AB';

SELECT
    Customer_id,
    Customer_name,
    SUBSTRING (Customer_id FROM 4 FOR 5) AS cust_number
FROM customer
WHERE SUBSTRING(Customer_id FOR 2) = 'AB';
```

STRING AGGREGATOR

STRING_AGG concatenates input values into a string, separated by delimiter

Syntax

`string_agg (expression, delimiter)`

STRING AGGREGATOR

STRING_AGG concatenates input values into a string, separated by delimiter

Example

```
SELECT
    order_id ,
    STRING_AGG (product_id,', ')
FROM sales
GROUP BY order_id;
```


CEIL & FLOOR

CEIL function returns the smallest integer value that is greater than or equal to a number
FLOOR function returns the largest integer value that is equal to or less than a number.

Syntax

CEIL (number)

FLOOR (number)

CEIL & FLOOR

CEIL function returns the smallest integer value that is greater than or equal to a number
FLOOR function returns the largest integer value that is equal to or less than a number.

Example

```
SELECT order_line,  
       sales,  
       CEIL (sales),  
       FLOOR (sales) FROM sales  
WHERE discount>0;
```

RANDOM

RANDOM function can be used to return a random number between 0 and 1

Syntax

RANDOM()

The random function will return a value between 0 (inclusive) and 1 (exclusive), so value ≥ 0 and value < 1 .

RANDOM

RANDOM function can be used to return a random number between 0 and 1

Example

Random decimal between a range (a included and b excluded)

```
SELECT RANDOM()*(b-a)+a
```

Random Integer between a range (both boundaries included)

```
SELECT FLOOR(RANDOM()*(b-a+1))+a;
```

SETSEED

If we set the seed by calling the setseed function, then the random function will return a repeatable sequence of random numbers that is derived from the seed.

Syntax

SETSEED (seed)

Seed can have a value between 1.0 and -1.0, inclusive.

SETSEED

If we set the seed by calling the setseed function, then the random function will return a repeatable sequence of random numbers that is derived from the seed.

Example

```
SELECT SETSEED(0.5);  
SELECT RANDOM();  
SELECT RANDOM();
```

ROUND

ROUND function returns a number rounded to a certain number of decimal places

Syntax

ROUND (number)

ROUND

ROUND function returns a number rounded to a certain number of decimal places

Example

```
SELECT order_line,  
       sales,  
       ROUND (sales)  
FROM sales
```


POWER

POWER function returns m raised to the nth power

Syntax

POWER (m, n)

This will be equivalent to m raised to the power n.

POWER

POWER function returns m raised to the nth power

Example

```
SELECT POWER(6, 2);
```

```
SELECT age, power(age,2) FROM customer OrDER BY age;
```

CURRENT DATE & TIME

CURRENT_DATE function returns the current date.

CURRENT_TIME function returns the current time with the time zone.

CURRENT_TIMESTAMP function returns the current date and time with the time zone.

Syntax

CURRENT_DATE

CURRENT_TIME ([precision])

CURRENT_TIMESTAMP ([precision])

- The CURRENT_DATE function will return the current date as a 'YYYY-MM-DD' format.
- CURRENT_TIME function will return the current time of day as a 'HH:MM:SS.GMT+TZ' format.
- The CURRENT_TIMESTAMP function will return the current date as a 'YYYY-MM-DD HH:MM:SS.GMT+TZ' format.

CURRENT DATE & TIME

CURRENT_DATE function returns the current date.

CURRENT_TIME function returns the current time with the time zone.

CURRENT_TIMESTAMP function returns the current date and time with the time zone.

Example

```
SELECT CURRENT_DATE;
```

```
SELECT CURRENT_TIME;  
SELECT CURRENT_TIME(1);
```

```
SELECT CURRENT_TIMESTAMP;
```

AGE

AGE function returns the number of years, months, and days between two dates.

Syntax

`age([date1,] date2)`

If date1 is NOT provided, current date will be used

AGE

AGE function returns the number of years, months, and days between two dates.

Example

```
SELECT age('2014-04-25', '2014-01-01');
```

```
SELECT order_line, order_date, ship_date,  
       age(ship_date, order_date) as time_taken  
FROM sales  
ORDER BY time_taken DESC;
```

EXTRACT

EXTRACT function extracts parts from a date

Syntax

EXTRACT ('unit' from 'date')

EXTRACT

EXTRACT function extracts parts from a date

Units

Unit	Explanation
day	Day of the month (1 to 31)
decade	Year divided by 10
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)
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
second	Seconds (and fractional seconds)
year	Year as 4-digits

EXTRACT

EXTRACT function extracts parts from a date

Example

```
SELECT EXTRACT(day from '2014-04-25');
```

```
SELECT EXTRACT(day from '2014-04-25 08:44:21');
```

```
SELECT EXTRACT(minute from '08:44:21');
```

```
SELECT order_line, EXTRACT(EPOCH FROM (ship_date - order_date))  
FROM sales;
```

Pattern Matching

Methods

1. LIKE statements
2. SIMILAR TO statements
3. ~ (Regular Expressions)

LIKE WILDCARDS

The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

LIKE Wildcards

Wildcard	Explanation
%	Allows you to match any string of any length (including zero length)
_	Allows you to match on a single character

A% means starts with A like ABC or ABCDE

%A means anything that ends with A

A%B means starts with A but ends with B

AB_C means string starts with AB, then there is one character, then there is C

LIKE

The PostgreSQL LIKE condition allows you to perform pattern matching using Wildcards.

Example

```
SELECT * FROM customer_table  
WHERE first_name LIKE 'Jo%';
```

```
SELECT * FROM customer_table  
WHERE first_name LIKE '%od%';
```

```
SELECT first_name, last_name FROM customer_table  
WHERE first_name LIKE 'Jas_n';
```

```
SELECT first_name, last_name FROM customer_table  
WHERE last_name NOT LIKE 'J%';
```

```
SELECT * FROM customer_table  
WHERE last_name LIKE 'G\%';
```

WILDCARDS

REG-EX Wildcards

Wildcard	Explanation
	Denotes alternation (either of two alternatives).
*	Denotes repetition of the previous item zero or more times
+	Denotes repetition of the previous item one or more times.
?	Denotes repetition of the previous item zero or one time.
{m}	denotes repetition of the previous item exactly m times.
{m,}	denotes repetition of the previous item m or more times.
{m,n}	denotes repetition of the previous item at least m and not more than n times
^,\$	^ denotes start of the string, \$ denotes end of the string
[chars]	a <i>bracket expression</i> , matching any one of the chars
~*	~ means case sensitive and ~* means case insensitive

~ OPERATOR

Example

```
SELECT * FROM customer  
WHERE customer_name ~* '^a+[a-z\s]+$'
```

```
SELECT * FROM customer  
WHERE customer_name ~* '^(a|b|c|d)+[a-z\s]+$'
```

```
SELECT * FROM customer  
WHERE customer_name ~* '^(a|b|c|d)[a-z]{3}\s[a-z]{4}$' ;
```

```
SELECT * FROM users  
WHERE name ~* '[a-z0-9\.\-\_\_]+@[a-z0-9\-\_]+\.[a-z]{2,5}';
```

CONVERSION TO STRING

TO_CHAR function converts a number or date to a string

Syntax

TO_CHAR (value, format_mask)

CONVERSION TO STRING

TO_CHAR function converts a number or date to a string

Format Mask

Parameter	Explanation
9	Value (with no leading zeros)
0	Value (with leading zeros)
.	Decimal
,	Group separator
PR	Negative value in angle brackets
S	Sign
L	Currency symbol
MI	Minus sign (for negative numbers)
PL	Plus sign (for positive numbers)
SG	Plus/minus sign (for positive and negative numbers)
EEEE	Scientific notation

CONVERSION TO STRING

TO_CHAR function converts a number or date to a string

Format Mask

Parameter	Explanation
YYYY	4-digit year
MM	Month (01-12; JAN = 01).
Mon	Abbreviated name of month capitalized
Month	Name of month capitalized, padded with blanks to length of 9 characters
DAY	Name of day in all uppercase, padded with blanks to length of 9 characters
Day	Name of day capitalized, padded with blanks to length of 9 characters
DDD	Day of year (1-366)
DD	Day of month (01-31)
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
am, AM, pm, or PM	Meridian indicator

CONVERSION TO STRING

TO_CHAR function converts a number or date to a string

Example

```
SELECT sales, TO_CHAR(sales, '9999.99')  
FROM sales;
```

```
SELECT sales, TO_CHAR(sales, 'L9,999.99')  
FROM sales;
```

```
SELECT order_date, TO_CHAR(order_date, 'MMDDYY')  
FROM sales;
```

```
SELECT order_date, TO_CHAR(order_date, 'Month DD, YYYY')  
FROM sales;
```

CONVERSION TO DATE

TO_DATE function converts a string to a date.

Syntax

TO_DATE(string1, format_mask)

CONVERSION TO DATE

TO_DATE function converts a string to a date.

Format Mask

Parameter	Explanation
YYYY	4-digit year
MM	Month (01-12; JAN = 01).
Mon	Abbreviated name of month capitalized
Month	Name of month capitalized, padded with blanks to length of 9 characters
DAY	Name of day in all uppercase, padded with blanks to length of 9 characters
Day	Name of day capitalized, padded with blanks to length of 9 characters
DDD	Day of year (1-366)
DD	Day of month (01-31)
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
am, AM, pm, or PM	Meridian indicator

CONVERSION TO DATE

TO_DATE function converts a string to a date.

Example

```
SELECT TO_DATE('2014/04/25', 'YYYY/MM/DD');
```

```
SELECT TO_DATE('033114', 'MMDDYY');
```

CONVERSION TO NUMBER

TO_NUMBER function converts a string to a number

Syntax

TO_NUMBER(string1, format_mask)

CONVERSION TO NUMBER

TO_NUMBER function converts a string to a number

Format Mask

Parameter	Explanation
9	Value (with no leading zeros)
0	Value (with leading zeros)
.	Decimal
,	Group separator
PR	Negative value in angle brackets
S	Sign
L	Currency symbol
MI	Minus sign (for negative numbers)
PL	Plus sign (for positive numbers)
SG	Plus/minus sign (for positive and negative numbers)
EEEE	Scientific notation

CONVERSION TO NUMBER

TO_NUMBER function converts a string to a number

Example

```
SELECT TO_NUMBER ('1210.73', '9999.99');
```

```
SELECT TO_NUMBER ('$1,210.73', 'L9,999.99');
```


CREATE USER

CREATE USER statement creates a database account that allows you to log into the database

Syntax

```
CREATE USER user_name  
[WITH PASSWORD 'password_value' | VALID UNTIL 'expiration'];
```

CREATE USER

CREATE USER statement creates a database account that allows you to log into the database

Example

```
CREATE USER starttech  
WITH PASSWORD 'academy';
```

```
CREATE USER starttech  
WITH PASSWORD academy '  
VALID UNTIL 'Jan 1, 2020';
```

```
CREATE USER starttech  
WITH PASSWORD academy '  
VALID UNTIL 'infinity';
```

GRANT & REVOKE

Privileges to tables can be controlled using GRANT & REVOKE. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, INDEX, CREATE, ALTER, DROP, GRANT OPTION or ALL.

Syntax

GRANT privileges ON object TO user;

REVOKE privileges ON object FROM user;

GRANT & REVOKE

Privileges to tables can be controlled using GRANT & REVOKE. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, INDEX, CREATE, ALTER, DROP, GRANT OPTION or ALL.

Privileges

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
TRUNCATE	Ability to perform TRUNCATE statements on the table.
REFERENCES	Ability to create foreign keys (requires privileges on both parent and child tables).
TRIGGER	Ability to create triggers on the table.
CREATE	Ability to perform CREATE TABLE statements.
ALL	Grants all permissions.

GRANT & REVOKE

Privileges to tables can be controlled using GRANT & REVOKE. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, INDEX, CREATE, ALTER, DROP, GRANT OPTION or ALL.

Example

```
GRANT SELECT, INSERT, UPDATE, DELETE ON products TO starttech;
```

```
GRANT ALL ON products TO starttech;
```

```
GRANT SELECT ON products TO PUBLIC;
```

```
REVOKE ALL ON products FROM starttech;
```

DROP USER

DROP USER statement is used to remove a user from the database.

Syntax

`DROP USER user_name;`

If the user that you wish to delete owns a database, be sure to drop the database first and then drop the user.

DROP USER

DROP USER statement is used to remove a user from the database.

Example

```
DROP USER techonthenet;
```

RENAME USER

ALTER USER statement is used to rename a user in the database

Syntax

```
ALTER USER user_name  
RENAME TO new_name;
```


RENAME USER

ALTER USER statement is used to rename a user in the database

Example

```
ALTER USER starttech  
RENAME TO ST;
```

FIND ALL USERS

Run a query against **pg_user** table to retrieve information about Users

Syntax

```
SELECT username  
FROM pg_user;
```

FIND LOGGED-IN USERS

Run a query against **pg_stat_activity** table to retrieve information about Logged-in Users

Syntax

```
SELECT DISTINCT username  
FROM pg_stat_activity;
```

Interview & performance tuning tips

Agenda

Part 1

Theoretical concepts for interview preparation

Part 2

Performance tuning tips which help in faster data retrieval and maintains database integrity

TABLESPACES

Tablespaces allow database administrators to define locations in the file system where the files representing database objects can be stored

Syntax

```
CREATE TABLESPACE <tablespace name>  
LOCATION <location on drive>;
```

- Creation of the tablespace can only be done by database **superuser**
- Ordinary database users can be allowed to use it by granting them the CREATE privilege on the new tablespace

TABLESPACES

Tablespaces allow database administrators to define locations in the file system where the files representing database objects can be stored

Example

```
CREATE TABLESPACE newspace LOCATION '/mnt/sda1/postgresql/data';
```

```
CREATE TABLE first_table (test_column int) TABLESPACE newspace;
```

```
SET default_tablespace = newspace;
```

```
CREATE TABLE second_table(test_column int);
```

```
SELECT newspace FROM pg_tablespace;
```

TABLESPACES

Tablespaces allow database administrators to define locations in the file system where the files representing database objects can be stored

USES

- If the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.
- Tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

Primary Key

Primary Key

- **Primary key** consists of one or more columns
- Used to uniquely identify each row in the table
- No value in the columns can be blank or NULL

Primary Key

Customer Table

Customer ID	First Name	Last Name	Age
V_001	Jay	Pee	23
V_002	Kay	Qu	23
V_003	Ell	Ao	13
V_004	Emm	Qu	24
V_005	Emm	Es	67

Primary Key

Order Table

Order ID	Customer_ID	Product_key	Value
O_001	V_001	P_001_01	65
O_002	V_001	P_001_02	23
O_003	V_001	P_002_01	13
O_004	V_003	P_003_01	41
O_005	V_005	P_002_01	67

Primary Key

Product Table

Product ID	Variant	Combine_column	Name
P_001	01	P_001_01	Cola 500 ml
P_001	02	P_001_02	Coal 1 l
P_002	01	P_002_01	Coffee beans
P_003	01	P_003_01	Surf
P_004	01	P_002_01	Cookies

Primary Key

City Data

- **City Name**
- **Country Name**
- **State**
- **Population**
- **Mean Temperature**
- **Pin Code**
- **Longitude**
- **Latitude**

Foreign Key

Order Table

Order ID	Customer_ID	Product_key	Value
O_001	V_001	P_001_01	65
O_002	V_001	P_001_02	23
O_003	V_001	P_002_01	13

Customer Table

Customer ID	First Name	Last Name	Age
V_001	Jay	Pee	23
V_002	Kay	Qu	23
V_003	Ell	Ao	13

Product Table

Product ID	Variant	Combine_column	Name
P_001	01	P_001_01	Cola 500 ml
P_001	02	P_001_02	Coal 1 l
P_002	01	P_002_01	Coffee beans

ACID

ACID (an acronym for Atomicity, Consistency Isolation, Durability) is a concept that Database Professionals generally look for when evaluating databases and application architectures. For a reliable database all these four attributes should be achieved.

ACID

- ATOMICITY

Atomicity is an all-or-none proposition

- CONSISTENCY

Consistency ensures that a transaction can only bring the database from one valid state to another

- ISOLATION

Isolation keeps transactions separated from each other until they're finished.

- DURABILITY

Durability guarantees that the database will keep track of pending changes in such a way that the server can recover from an abnormal termination

TRUNCATE

The TRUNCATE TABLE statement is used to remove all records from a table or set of tables in PostgreSQL. It performs the same function as a DELETE statement without a WHERE clause.

Syntax

```
TRUNCATE [ONLY] table_name  
[ CASCADE | RESTRICT ] ;
```

TRUNCATE

The TRUNCATE TABLE statement is used to remove all records from a table or set of tables in PostgreSQL. It performs the same function as a DELETE statement without a WHERE clause.

Example

```
TRUNCATE TABLE Customer_20_60;
```

Same as

```
DELETE FROM Customer_20_60;
```


Normalization

What

Organizing the data to reduce data redundancy

Redundancy in table leads to three Anomalies:

1. Insert Anomaly
2. Update Anomaly
3. Delete Anomaly

Normalization

While inserting new data:

- If category value is not available, we cannot insert that sales transaction

If we have a separate product table and category values are not noted in sales table, then we can insert the sales transaction.

Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Category	Product Name
1	CA-2016-152156	08-11-2016	CG-12520	FUR-BO-10001798	261.96	Furniture	Bush Somerset Collection Bookcase
2	CA-2016-152156	08-11-2016	CG-12520	FUR-CH-10000454	731.94	Furniture	Hon Deluxe Fabric Upholstered Stacking Chairs Rounded Back
3	CA-2016-138688	12-06-2016	DV-13045	OFF-LA-10000240	14.62	Office Supplies	Self-Adhesive Address Labels for Typewriters by Universal
4	US-2015-108966	11-10-2015	SO-20335	FUR-TA-10000577	957.5775	Furniture	Bretford CR4500 Series Slim Rectangular Table

Insert Anomaly

Normalization

Update Anomaly

While updating data:

- If category of one product changes, we need to update several rows
- If the person updating the table is not aware of the redundancy, this may lead to inconsistency in data

If we have a separate product table, we need to update only one row of data.

Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Category	Product Name
1595	CA-2015-118423	24-03-2015	DP-13390	FUR-BO-10000362	359.058	Furniture	Sauder Inglewood Library Bookcases
1611	CA-2014-156349	26-05-2014	ML-17395	FUR-BO-10000362	290.666	Furniture	Sauder Inglewood Library Bookcases
2604	CA-2016-165848	04-06-2016	EN-13780	FUR-BO-10000362	136.784	Furniture	Sauder Inglewood Library Bookcases
5395	US-2014-123183	19-11-2014	GR-14560	FUR-BO-10000362	1025.88	Furniture	Sauder Inglewood Library Bookcases
7681	CA-2014-133592	31-12-2014	KM-16375	FUR-BO-10000362	341.96	Furniture	Sauder Inglewood Library Bookcases

Normalization

Deletion Anomaly

While deleting data:

- We may want to delete a particular sales transaction, but in doing so, we may unintentionally lose product data

If we have a separate product table, we can delete sales transactions without losing product data.

Order Line	Order ID	Order Date	Customer ID	Product ID	Sales	Category	Product Name
3513	CA-2017-140326	04-09-2017	HW-14935	FUR-BO-10000112	825.174	Furniture	Bush Birmingham Collection Bookcase Dark Cherry

Normalization

Forms of Normalization

- **First Normal Form or 1NF**
- **Second Normal Form or 2NF**
- **Third Normal Form or 3NF**
- **Boyle-Codd Normal Form or BCNF**
- **Fourth Normal Form or 4NF**
- **Fifth Normal Form or 5NF**

First Normal Form

Rules

1. Each attribute should contain a single value in each row
2. Maintain datatypes in a column
3. Each column should have unique column name

First Normal Form

Rules

1. Each attribute should contain a single value in each row

Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798, FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577, OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487



Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577
US-2015-108966	11-10-2015	18-10-2015	SO-20335	OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487

First Normal Form

Rules

- Maintain datatypes in a column

Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016 morning	11-11-2016	CG-12520	FUR-BO-10001798
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015 evening	18-10-2015	SO-20335	FUR-TA-10000577
US-2015-108966	11-10-2015	18-10-2015	SO-20335	OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487



Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577
US-2015-108966	11-10-2015	18-10-2015	SO-20335	OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487

First Normal Form

Rules

- Each column should have unique column name

Order_ID	Date	Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577
US-2015-108966	11-10-2015	18-10-2015	SO-20335	OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487



Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577
US-2015-108966	11-10-2015	18-10-2015	SO-20335	OFF-ST-10000760
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487

Second Normal Form

Rules

1. Table should be in first normal form
2. There should not be any partial dependency

Second Normal Form

Rules

- There should not be any partial dependency

Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID	Category	Product Name	Sales
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798	Furniture	Bush Somerset Collection Bookcase	261.96
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454	Furniture	Hon Deluxe Fabric Upholstered Stacking Chairs Rounded Back	731.94
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240	Office Supplies	Self-Adhesive Address Labels for Typewriters by Universal	14.62
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577	Furniture	Bretford CR4500 Series Slim Rectangular Table	957.58
CA-2017-125913	16-01-2017	16-01-2017	JO-15145	FUR-FU-10001487	Furniture	Eldon Expressions Wood and Plastic Desk Accessories Cherry Wood	27.92
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487	Furniture	Eldon Expressions Wood and Plastic Desk Accessories Cherry Wood	48.86

- Order_ID and Product_ID together form the composite primary key for this table
- Product name and Category only depend on product_ID and not on Order_ID
- This is partial dependency

Second Normal Form

Rules

- To remove partial dependency, split the table

Order_ID	Order_Date	Ship_Date	Customer_ID	Product_ID	Sales
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798	261.96
CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454	731.94
CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240	14.62
US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577	957.58
CA-2017-125913	16-01-2017	16-01-2017	JO-15145	FUR-FU-10001487	27.92
CA-2014-115812	09-06-2014	14-06-2014	BH-11710	FUR-FU-10001487	48.86

Product_ID	Category	Product Name
FUR-BO-10001798	Furniture	Bush Somerset Collection Bookcase
FUR-CH-10000454	Furniture	Hon Deluxe Fabric Upholstered Stacking Chairs Rounded Back
OFF-LA-10000240	Office Supplies	Self-Adhesive Address Labels for Typewriters by Universal
FUR-TA-10000577	Furniture	Bretford CR4500 Series Slim Rectangular Table
FUR-FU-10001487	Furniture	Eldon Expressions Wood and Plastic Desk Accessories Cherry Wood

Third Normal Form

Rules

1. Table should be in second normal form
2. There should not be any transitive dependency

Third Normal Form

Rules

- There should not be any transitive dependency

Order ID	Order Date	Customer ID	Sales	Customer Name	Age
CA-2016-152156	08-11-2016	CG-12520	261.96	Claire Gute	67
CA-2016-138688	12-06-2016	DV-13045	14.62	Darrin Van Huff	31
US-2015-108966	11-10-2015	SO-20335	957.5775	Sean O'Donnell	65

- Customer name and Age depend on Customer ID
- Customer ID depends on Order ID (primary key)
- This is transitive dependency

Third Normal Form

Rules

- To remove transitive dependency also, split the table

Order ID	Order Date	Customer ID	Sales
CA-2016-152156	08-11-2016	CG-12520	261.96
CA-2016-138688	12-06-2016	DV-13045	14.62
US-2015-108966	11-10-2015	SO-20335	957.5775

Customer ID	Customer Name	Age
CG-12520	Claire Gute	67
DV-13045	Darrin Van Huff	31
SO-20335	Sean O'Donnell	65

Boyce-Codd Normal Form

Rules

1. Table should be in third normal form

2. If B depends on A, A should be a superkey

Superkey is any such column or group of columns which can uniquely identify each row i.e. it can become a primary key

Order Line	Order ID	Order Date	Ship Date	Customer ID	Product ID	Sales
1	CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-BO-10001798	261.96
2	CA-2016-152156	08-11-2016	11-11-2016	CG-12520	FUR-CH-10000454	731.94
3	CA-2016-138688	12-06-2016	16-06-2016	DV-13045	OFF-LA-10000240	14.62
4	US-2015-108966	11-10-2015	18-10-2015	SO-20335	FUR-TA-10000577	957.5775

4th Normal Form

Rules

1. Table should be in Boyce-Codd normal form
2. There should not be any multivalued dependency

4th Normal Form

Rules

- There should not be any multivalued dependency

Scenario: John bought a Football and a baseball bat and received two coupons as a reward

Customer	Product	coupon
John	Football	10% off on next purchase
John	Baseball Bat	Free home delivery

Customer	Product	coupon
John	Football	Free home delivery
John	Baseball Bat	10% off on next purchase

- Product and coupon are not related

4th Normal Form

Rules

- There should not be any multivalued dependency

Scenario: John bought a Football and a baseball bat and received two coupons as a reward

Customer	Product	coupon
John	Football	10% off on next purchase
John	Baseball Bat	Free home delivery
John	Football	Free home delivery
John	Baseball Bat	10% off on next purchase

- For unrelated data we have to store all combinations of data

Customer	Product	coupon
John	Football	10% off on next purchase
John	Baseball Bat	Free home delivery

- This table suggests that the purchaser of football did not get the free delivery coupon

4th Normal Form

Rules

How to find multivalued dependency?

1. There should be three or more columns in the table
2. There should be no relation between values of two or more columns

4th Normal Form

Rules

- To remove multivalued dependency also, split the table

Separate product information and coupon information into separate tables

Customer	Product
John	Football
John	Baseball Bat

Customer	coupon
John	10% off on next purchase
John	Free home delivery

5th Normal Form

Rules

1. Table should be in 4th normal form
2. There should not be any join dependency

5th Normal Form

Rules

- There should not be any join dependency

Shop No.	Brand	Product
1	Nike	Sport shoes
1	Nike	Track suit
2	Nike	Sport shoes
2	Adidas	Track suit

Shop No.	Brand
1	Nike
2	Nike
2	Adidas

Brand	Product
Nike	Sport shoes
Nike	Track suit
Adidas	Track suit

Shop No.	Product
1	Sport shoes
1	Track suit
2	Sport shoes
2	Track suit

- Does shop 2 sell Nike Track suit?
 - Shop 2 sells Nike products, Nike has Track suit product, Shop 2 sells track suit
 - But NO, Shop 2 does not sell Nike track suit
 - This implies information was lost during the split

5th Normal Form

Rules

- There is join dependency if the shops are selling all the combinations of brand and product
- Take shop 2 brands and products -> cross join them
- Out of the result, whichever combination is actually offered, the shop has to sell it
- If this condition holds, the table is not in 5th Normal form and we can split the table

Shop No.	Brand
2	Nike
2	Adidas

Shop No.	Product
2	Sport shoes
2	Track suit

Brand	Product
Nike	Sport shoes
Nike	Track suit
Adidas	Track suit
Adidas	Sport shoes

Best Practices

Displays the execution plan for a query statement without running the query.

EXPLAIN (Syntax)

EXPLAIN [VERBOSE] *query*;

VERBOSE

Displays the full query plan instead of just a summary.

query

Query statement to explain.

Best Practices

SOFT DELETE VS HARD DELETE

SOFT DELETE

Soft deletion means you don't actually delete the record instead you are marking the record as deleted

HARD DELETE

Hard deletion means data is physically deleted from the database table.

Best Practices

UPDATE vs CASE

UPDATE

```
Update customer set customer_name = (trim(upper(customer_name)))  
where (trim(upper(customer_name)) <> customer_name)
```

Every updated row is actually a soft delete and an insert. So updating every row will increase the storage size of the table

CASE STATEMENT

Instead you can use the case statements while creating such tables

Best Practices

VACUUM

SYNTAX

VACUUM [*table*]

USE

- Reclaims disk space occupied by rows that were marked for deletion by previous UPDATE and DELETE operations.
 - Compacts the table to free up the consumed space
- Use it on tables which you are updating and deleting on a regular basis

Best Practices

TRUNCATE VS DELETE

- The TRUNCATE statement is typically far more efficient than using the DELETE statement with no WHERE clause to empty the table
 - TRUNCATE requires fewer resources and less logging overhead
- Instead of creating table each time try to use truncate as it will keep the table structure and properties intact
 - Truncate frees up space and impossible to rollback

Best Practices

STRING FUNCTIONS

Pattern Matching

- Whenever possible use LIKE statements in place of REGEX expressions
- Do not use 'Similar To' statements, instead use Like and Regex
- Avoid unnecessary string operations such as replace, upper, lower etc

String Operations

- Use trim instead of replace whenever possible
- Avoid unnecessary String columns. For eg. Use date formats instead of string for dates

JOINS

Syntax

```
SELECT a.order_line , a.product_id, b.customer_name, b.age
FROM sales_2015 AS a LEFT JOIN customer_20_60 AS b
ON a.customer_id = b.customer_id
ORDER BY customer_id;
```

Best Practices

- Use subqueries to select only the required fields from the tables
- Avoid one to many joins by mentioning Group by clause on the matching fields

Best Practices

A *schema* is a collection of database objects associated with one particular database.
You may have one or multiple schemas in a database.

SCHEMAS

1. To allow many users to use one database without interfering with each other.
2. To organize database objects into logical groups to make them more manageable.
3. Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Best Practices

A *schema* is a collection of database objects associated with one particular database.
You may have one or multiple schemas in a database.

SCHEMAS (Syntax)

```
CREATE SCHEMA testschema;
```

Final Project

About

Congratulations on reaching to the end of the Course.

In this Project, you will be using all the skills that you have acquired throughout this course.

Another application – Sports Analytics

Final Project



BEST CAREER ECONOMY RATE IN IPL

(MINIMUM 100 OVERS BOWLED)

Infographic showing the best career economy rate in the IPL for bowlers with a minimum of 100 overs bowled. The background features a photo of a bowler in a red jersey.

PLAYERS	MATCHES	MAIDENS	WICKETS	BBI	ECO	4WI	5WI
RASHID KHAN	46	3	55	3/19	6.55	0	0
ANIL KUMBLE	42	1	45	5/5	6.57	2	1
SUNIL NARINE	110	3	122	5/19	6.67	6	1
MUTTIAH MURALITHARAN	66	1	63	3/11	6.67	0	0
DALE STEYN	92	7	96	3/8	6.76	0	0

100MB MASTER BLASTER



Final Project

IPL dataset

- **Two sets of data:**
 - Ball-by-ball data
 - Match-wise data
- Create an SQL based database and perform the given tasks
- The queries and results of this project will help to answer questions in the final assessment