DBA

RNAM:

1. used for backup

2. recovery files

3.update the corrupted files/column/index

4.Recovering data block

Database cloning:

1.DB cloning using the backup

2.DB cloning from active DB

Dataguard:

It’s basically a standby DB, needs extra server, extra DB, extra licensing, maintenance or recources. It is used for if the main db fails/corrupts then without downtime Dataguard (standby DB) can be used.

RAC: real application cluster

ARCHITECTURE:

If a command is executed

Select \* from emp:

And the execution fails check for

1.the syntax: weather the written syntax for the command is perfect or not.

2.permission to access

3.Semantic check: weather the table in the database exists or not, weather the columns inside the table exists or not.

4.Execution plan: implementing the steps(syntax/command) to consume less resources(cpu, memory). Here more resources = more cost, less resources = less cost.

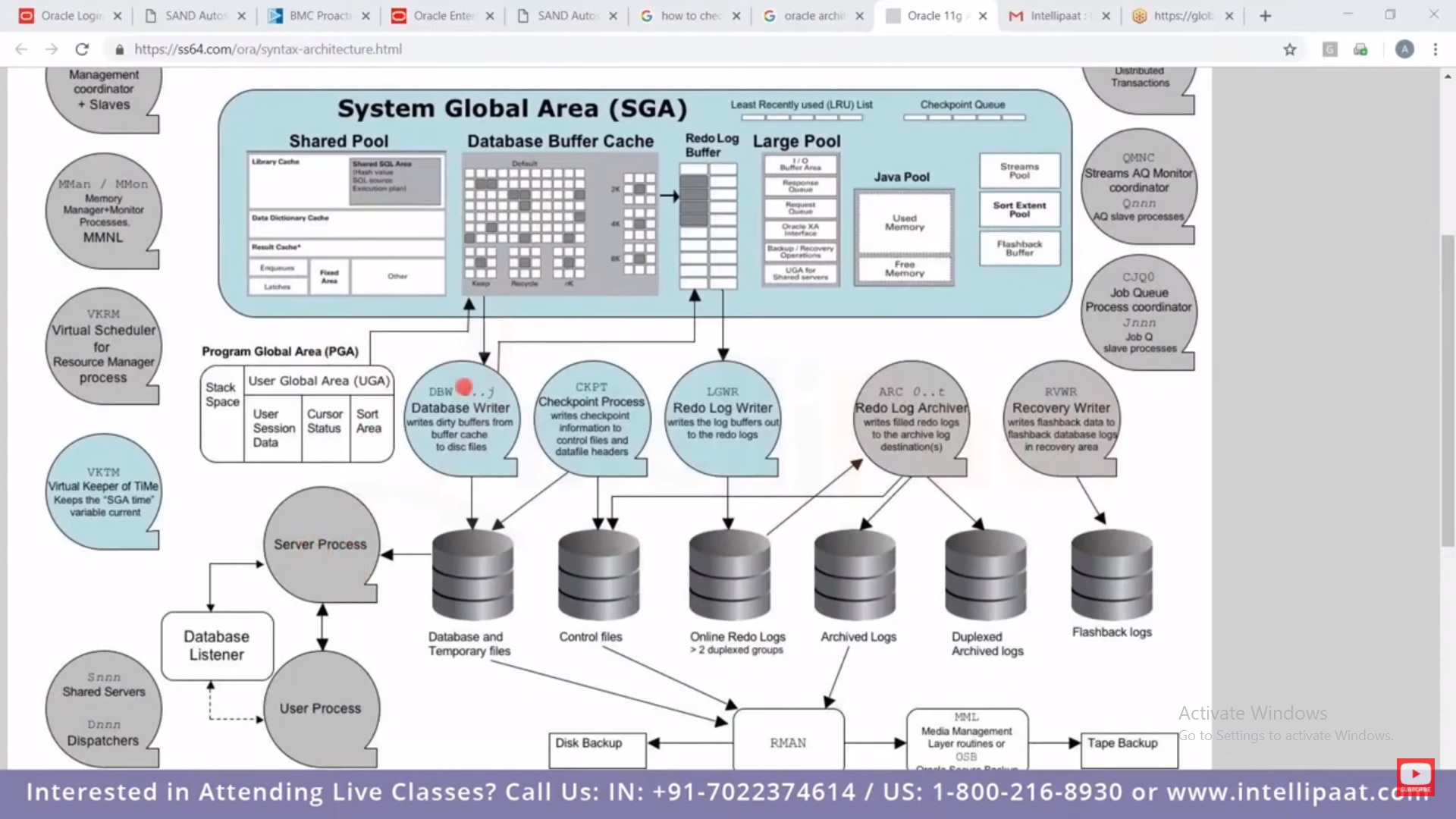
All this 1.syntax check 2. Semantic check and 3.Execution plan is stored in oracle memory

Library cache:

Parsing:

Soft Parsing: less time to query (get the data)

Hard Parsing: where the time taken for the query is more



Normalization:

Organizing of data in a database is normalization.

It is a process of reducing the redundancy (faltupan) of data.

It is a multistep process to put data into tabular form removing the duplicated data from its relational tables. This eliminates the repeated data. If the duplicated data is not removed the query process will take more time and cause trouble.

Normalization comes into existence due to the data in tables was not related/meaningful.

Data Anomalies:

If the data is not normalized and has a data redundancy then it will not only eat the space/memory but also make difficult to handle the database.

1.INSERTION ANOMALIES:

Inserting the specific row or specific column for many rows.

2.UPDATION:

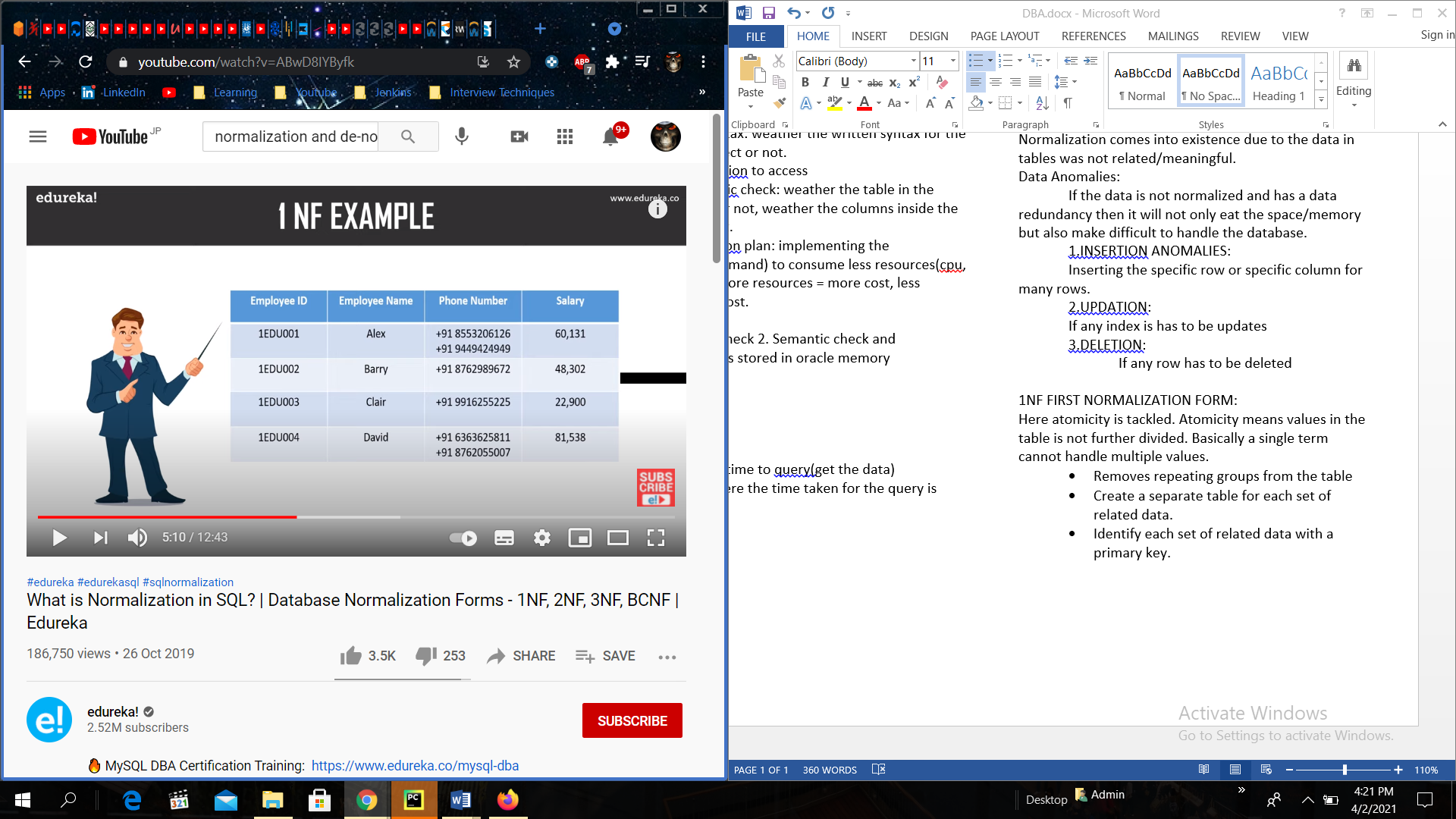
If any index is has to be updates

3.DELETION:

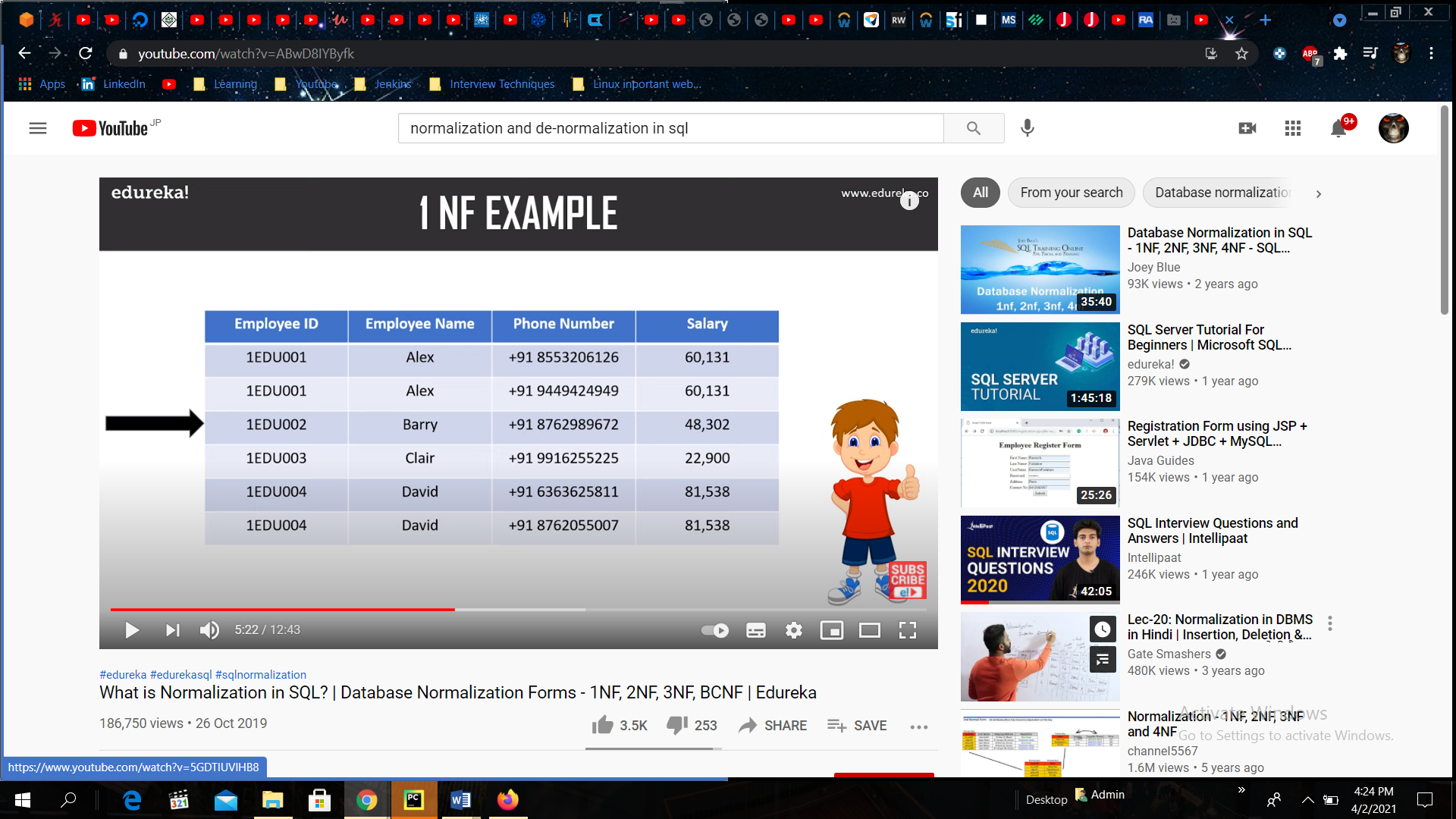
If any row has to be deleted

1NF FIRST NORMALIZATION FORM:

Here atomicity is tackled. Atomicity means values in the table is not further divided. Basically a single term cannot handle **multiple values.**

* Removes repeating groups from the table
* Create a separate table for each set of related data.
* Identify each set of related data with a primary key.
* 
* Here the phone no column has two phone nos in some rows which violets the rule.

If we apply a rule we will get



**SYNTAX:**

Some of The Most Important SQL Commands

SELECT - extracts data from a database

UPDATE - updates data in a database

DELETE - deletes data from a database

INSERT INTO - inserts new data into a database

CREATE DATABASE - creates a new database

ALTER DATABASE - modifies a database

CREATE TABLE - creates a new table

ALTER TABLE - modifies a table

DROP TABLE - deletes a table

CREATE INDEX - creates an index (search key)

DROP INDEX - deletes an index

**SELECT:**

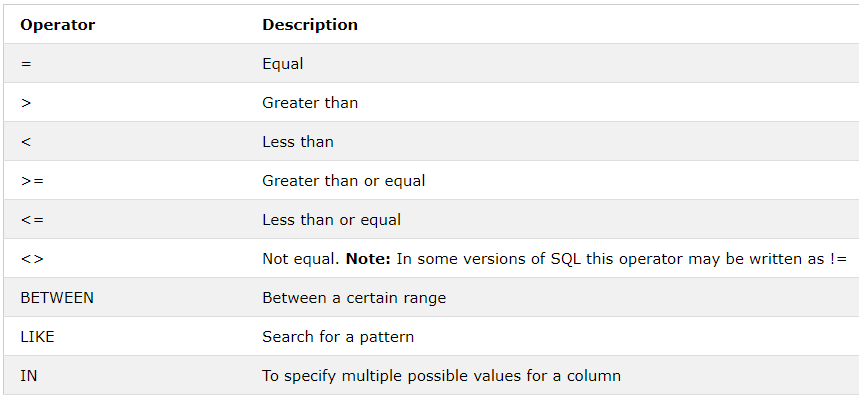
* SELECT column1, column2 FROM table\_name;

(SELECT CustomerName, City FROM Customers)

* SELECT \* FROM table\_name;

**SELECT DISTINCT:**

Is used to return only distinct(different or first unique) values.

* SELECT DISTINCT column1, column2, … FROM table\_name;
* SELECT COUNT(DISTINCT column\_name) FROM table\_name;

(returns **no**. of different values)

* SELECT COUNT(\*) AS DistinctCounries FROM (SELECT DISTINCT Country FROM Customers);

**WHERE:**

Where clause is used to filter records.

* SELECT column1, column2

FROM table\_name WHERE condition;

( SELECT \* FROM customers WHERE Country = ‘Mexico’; )

( SELECT \* FROM customers WHERE CustomerID = 1; )

(SELECT \* FROM customers WHERE NOT Country = ‘Berlin’;)

**AND, OR** and **NOT:**

**AND**

* SELECT column1, column2 FROM table\_name

WHERE condition AND condition2 AND condition3;

(SELECT \* FROM Customers WHERE Country = ‘Germany’ AND City = ‘Berlin’;)

**OR**

* SELECT column1, column2 FROM table\_name

WHERE condition1 OR condition2 OR condition3;

(SELECT \* FROM Customers WHERE Country = ‘Germany’ OR Country = ‘Spain’; )

**NOT**

SELECT column1, column2 FROM table\_name

WHERE NOT condition;

(SELECT \* FROM Customers WHERE NOT Country = ‘Germany’; )

**AND OR NOT:**

SELECT \* FROM Customers WHERE Country=’Germany’ AND (City = ‘Berlin’ OR City = ‘Munchen’; )

SELECT \* FROM Customers WHERE NOT Country = ‘Germany’ AND NOT Country = ‘USA’;

**ORDER BY: (ASC | DESC)**

SELECT column1, column2 FROM table\_name

ORDER BY column1, column2 ASC|DESC;

(SELECT \* FROM Customers ORDER BY Country DESC; )

(SELECT \* FROM Customers ORDER BY Country ASC, CustomerName DESC; )

**INSERT INTO:**

To insert new table in a table. 2 ways:

**1.**Specify column name and values to be inserted.

INSERT INTO table\_name (column1, column2, column3)

VALUES (value1, value2, value3);

**2.**Adding values for all the column w.r.to order of values.

INSERT INTO table\_name VALUES(value1, value2, value3);

Eg: **INSERT** **INTO** Customer(CustomerName, ContactName, Address, City, PostalCode, Country)

**VALUES**(‘Cardinal’, ‘Tom.B.Erichsen’, ‘Skagen 21’, ‘Stavanger’, ‘4006’, ‘Norway’)

**NULL VALUES:**

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

* SELECT Column\_name FROM table\_name

WHERE column\_name IS NULL;

* SELECT Column\_name FROM table\_name

WHERE column\_name IS NOT NULL;

**UPDATE:**

To modify existing records in a table.

* UPDATE table\_name

SET column1 = value1, column2 = value2

WHERE condition;

(UPDATE Customers SET ContactName = ‘Alfred Schmidt’, City = ‘Frankfurt’ WHERE CustomID = 1; )

(UPDATE Customers SET ContactName = ‘Juam’ WHERE Country = ‘Mexico’;)

**DELETE:**

* DELETE FROM table\_name WHERE condition;

(DELETE FROM Customers WHERE CustomerName = ‘Alter Futter’;)

Delete all records

(DELETE FROM table\_name;)

(Delete FROM Customers;)

**SELECT TOP:**

To specify no of records to return

* SELECT TOP number | percent column\_name(s) FROM table\_name WHERE condition;

(SELECT TOP 3 \* FROM Customers;) OR

(SELECT \* FROM Customers LIMIT 3;) OR

(SELECT TOP 3 \* FROM Customers WHERE Country=’Germany’;)

(SELECT \* FROM Customers WHERE Country = ‘Germany LIMIT = 3;)

**OFFSET:** to skip the no of rows

SELECT column\_name FROM table\_name OFFSET no\_of\_rows LIMIT no\_to\_limit;

( eg: SELECT id, name FROM customers LIMIT 3, 10; )

* Output🡪 row from 4th till 10 (mysql starts counting from zero, meaning that the offset of the first row is 0, not 1)

**MIN() and MAX():**

**MIN** returns smallest values of selected column.

**MAX** returns largest values of selected column.

* SELECT MIN(column\_name) FROM table\_name

WHERE condition;

(SELECT MIN(Price) AS SmallestPrice FROM Products;)

* SELECT MAX(column\_name) FROM table\_name

WHERE condition;

(SELECT MAX(Price) AS LargestPrice WHERE condition; )

**COUNT():**

Returns no of rows that matches a specific criterion.

* SELECT COUNT(column\_name) FROM table\_name

WHERE condition;

(SELECT COUNT(ProductID) FROM Products;)

**AVG():**

Returns average of numeric column.

* SELECT AVG(column\_name) FROM table\_name WHERE condition;

(SELECT AVG(Price) FROM Products;)

**SUM():**

Returns total sum of numeric columns.

* SELECT SUM(column\_name) FROM table\_name

WHERE condition;

(SELECT SUM(Quantity) FROM OrderDetails; )

**LIKE:**

**LIKE** operator is used with **WHERE** clause

* SELECT column1, column2 FROM table\_name

WHERE columnN LIKE pattern

Eg: abcdefghijklmnop

LIKE ‘a%’ = starts with a

LIKE ‘%p’ = ends with p

LIKE ‘%ij%’ = any position ij

LIKE ‘\_b%’ = second position must be b and then anyword

LIKE ‘a\_%’ = at least 2 character after a

LIKE ‘a\_\_%’ = at least 3 character after a

LIKE ‘a%p’ = starts with a and ends with p

LIKE ‘[acs]’ = starts with a or c or s (any of those)

LIKE ‘[!acf]’ = does not starts with any of those (acf)

(SELECT \* FROM Customers WHERE CustomerName LIKE ‘a%’ ;)

(Select all records where the second letter of the City is an "a". 🡪 SELECT \* FROM Customers WHERE City LIKE ‘\_a%’;)

(Select all records where the first letter of the City is an "a" or a "c" or an "s". 🡪 SELECT \* FROM Customers WHERE City LIKE ‘[acs]%’;)

(Select all records where the first letter of the City starts with anything from an "a" to an "f". 🡪 SELECT \* FROM Customers WHERE City LIKE ‘[a-f]%’;)

(Select all records where the first letter of the City is NOT an "a" or a "c" or an "f". 🡪 SELECT \* FROM Customers WHERE City LIKE ‘[!acf]%’;)

**WILDCARD Characters:**

\* = zero or more characters bl\* = bl, black, blue

? = single character h?t = hot, hat, hit

[] = single character with brackets h[oa]t = hot or hat

! = not in bracket h[!oa]t = hit

-= range of charac. c[a-c] = cat, cbt, cct

# = single numeric character 2#5 = 205,215,225…

**IN:**

Allows to specify multiple values in where clause

IN operator is shorthand for multiple OR

* SELECT column\_name(s) FROM table\_name WHERE column\_name IN (values1, values2);
* SELECT column\_name(S) FROM table\_name WHERE column\_name IN (SELECT STATEMENT);

(SELECT \* FROM Customers WHERE Country IN (‘Germany’, ‘France’, ‘UK’);)

(SELECT \* FROM Customers WHERE Country IN (SELECT Country FROM Suppliers);)

(Use the IN operator to select all the records where Country is NOT "Norway" and NOT "France". 🡪 SELECT \* FROM Customers WHERE Country NOT IN (‘Norway’, ‘France’);)

**BETWEEN:**

Values within range (date, numbers, text)

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

WHERE column\_name BETWEEN value1 AND value2;

(SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20;)

(SELECT \* FROM Products WHERE Price NOT BETWEEN 10 AND 20;)

(SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20  
AND CategoryID NOT IN (1,2,3);)

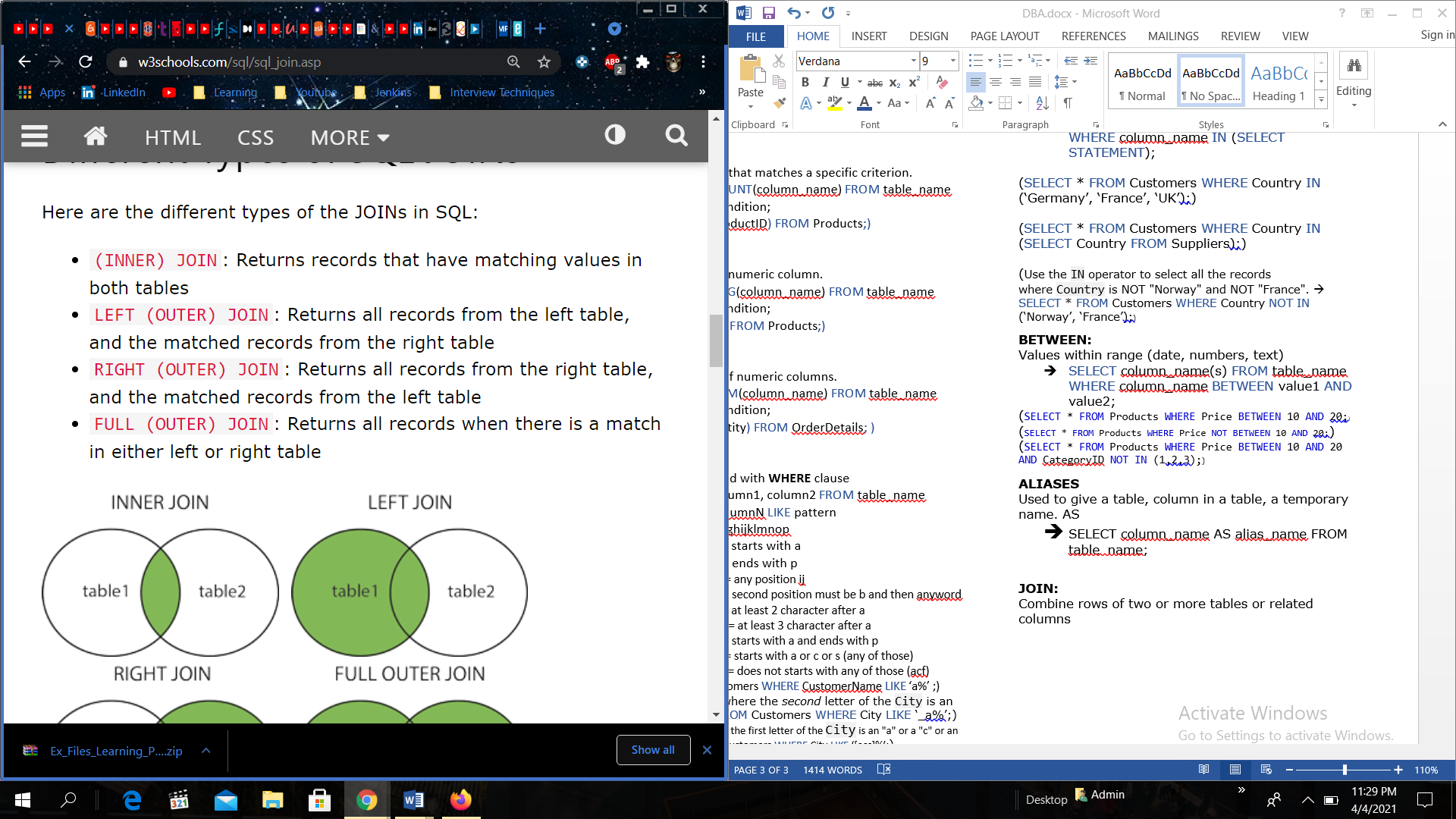
**ALIASES**

Used to give a table, column in a table, a temporary name. AS

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

**JOIN:**

Combine rows of two or more tables or related columns





* SELECT Order.OrderID, Customer.CustomerName, Order.OrderDate FROM Orders INNER JOIN Customers
* ON Order.CustomerID = Customers.CustomerID

**INNER JOIN:**

Matching values in both tables..

* SELECT column\_name(s) FROM table1 INNER JOIN table2 ON table.column\_name = table2.column\_name;

( SELECT Order.OrderID, Customers.CustomerName FROM Order INNER JOIN Customers ON Order.CustomerID = Customers.CustomerID )

**LEFT JOIN:**

* SELECT column\_name(s) FROM table1

LEFT JOIN table2 ON table1.column\_name = table2.column\_name;

**RIGHT JOIN:**

* SELECT column\_name(s) FROM table1 RIGHT JOIN table2 ON table1.column\_name = table2.column\_name;

**FULL OUTER JOIN:**

* SELECT column\_name(s) FROM table1 FULL OUTER JOIN table2 ON table1.column\_name = table2.column\_name WHERE condition;

**SELF JOIN:**

* SELECT column\_name(s) FROM table T1, Table T2 WHERE condition;

(SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City FROM Customers A, Customers B WHERE A.CustomerID <> B.CustomerID AND A.City = B.City ORDER BY A.City;)

**UNION:**

Union operator is used to combine the result-set of two or more SELECT statement.

* SELECT column\_name(s) FROM table1

UNION

SELECT column\_name(s) FROm table2;

**UNION ALL with WHERE:**

* SELECT City, Country FROM Customers

WHERE Country = ‘Germany’

UNION ALL

SELECT City, Country FROM Suppliers

WHERE Country = ‘Germany’

ORDER BY City;

**1.SUPER KEY: Super Set of Key.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Student\_ID | Student\_name | Phone | Address | DOB |
| 1 | Sita | 9987.. | Andheri | 17/02 |
| 2 | Rita | 98203 | Malad | 19/01 |
| 3 | Sita | 9978.. | Andheri | 2/03 |
| 4 | Nita | 6321.. | Malad | 7/07 |

Set of attributes that can uniquely identify each record.

**Super** key🡪{Student\_ID, Student\_name}

It may contain extra attributes.

**2.Candidate Key**: any key which can be unique like phone, ID but not name(name might be same)

🡪 minimal set of super key:

Candidate key🡪 {Student\_ID}

🡪{Phone}

**3. Primary Key:**

Most appropriate Candidate Key.

Here Student\_ID is most appropriate

**4. Foreign Key:**

Primary key in another table.

**5.Composite Key:**

**2 or more** attributes that uniquely identify any record in a table.

🡪{Student\_ID, Subject\_ID}

**DIFFERENCE BETWEEN:**

**WHERE VS HAVING:**

**WHERE:** is used for filtering rows not on aggregate data.

HAVING: works on aggregated data.

Aggrigate Functions:

To perform calculations on multiple rows of a single column. Returns single value. Used to summarize data (count, max, min, avg, sum)

NORMALIZATION:

Represent the way of organizing the structured data in database efficiently, includes creating table, defining rules and relationship in between them.

**DeNormalization:**

Inverse process of normalization, where normalized schema is converted into redundant information. The reason for performing the denormalization is the overheads produced in query processor by an over-normalized structure.

1NF 🡪 First Normal Form

Single valued attribute.

2NF 🡪 Second Normal Form

No partial dependency

**3rd Higgest Salary: table Emps**

|  |  |
| --- | --- |
| Id | Sal |
| 1 | 1000 |
| 2 | 2000 |
| 3 | 5000 |
| 4 | 3000 |
| 5 | 6000 |

|  |
| --- |
| 6000 |
| 5000 |
| 3000 |
| 2000 |
| 1000 |

(SELECT TOP 3 Sal FROM Emps ORDER BY DESC) 🡪

SELECT TOP 1 Sal FROM(SELECT TOP 3 Sal FROM Emps ORDER BY DESC) ORDER BY Sal ASC;

OR

**Nth Higgest**

**SELECT TOP 1 Column\_name FROM( SELECT )**

SELECT TOP 1 Sal FROM ( SELECT TOP N Sal FROM Emps ORDER BY DESC ) ORDER BY Sal ASC;

3Max salaries fromtable:

SELECT distinct Salary from worker a WHERE 3 >= (SELECT count(distinct Salary) from worker b WHERE a.Salary <= b.Salary) order by a.Salary desc;

**Increase Column by 5%** 🡪

UPDATE table\_name SET column\_name = column\_name + (column\_name \* 5.0/100.0);

**All starts with ‘A’** 🡪

SELECT column\_name FROM table\_name WHERE column\_name LIKE ‘A%’;

**No of rows in specific row values**

(no of employees working in department = ‘ABC’)🡪

SELECT COUNT (\*) FROM table\_name WHERE column\_name = ‘ABC’;

**Ends with ‘A’ and have 6 character**

SELECT \* FROM table\_name WHERE column\_name LIKE ‘\_ \_ \_ \_ \_ A’;

**DENSE RANK():** is an Analytical

JOIN 3 or More Tables:

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

**DENSE\_RANK Function:** is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that assigns a rank to each row within a partition or result set with no gaps in ranking values.

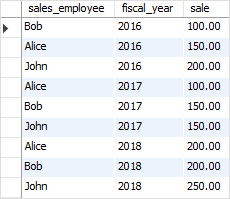
SYNTAX:

DENSE\_RANK() OVER (

PARTITION BY <expression>[{,<expression>...}]

ORDER BY <expression> [ASC|DESC], [{,<expression>...}]

)



**SELECT**

sales\_employee,

fiscal\_year,

sale,

**DENSE\_RANK**() **OVER** (**PARTITION** **BY**

fiscal\_year

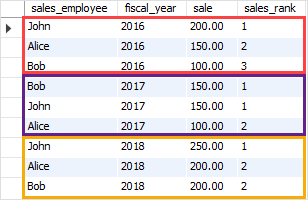
**ORDER** **BY**

sale **DESC**

) sales\_rank

**FROM**

sales;

OUTPUT: 

* First, the PARTITION BY clause divided the result sets into partitions using fiscal year.
* Second, the ORDER BY clause specified the order of the sales employees by sales in descending order.
* Third, the DENSE\_RANK() function is applied to each partition with the rows order specified by the ORDER BY clause.
* Find **DUPLICATE ROWS:**

**SELECT \* COUNT(**column\_name**) FROM** table\_name **GROUP BY** column\_name **HAVING COUNT(**column\_name**) > 1**

* **EVEN Records:**

SELECT \* FROM table\_name WHERE MOD (column\_name, 2) = 0;

* **ODD Records:**

SELECT \* FROM table\_name WHERE MOD (column\_name, 2) = 1;

* **Get all same Employee working on same department:**

SELECT DISTINCT E.eid, E.empname, E.dept FROM table1, Table2 WHERE E.eid = E

* **First 3 character** from First\_name from Worker table:

SELECT substring(First\_name, 1, 3) from Worker;

* **Position of “a” in the first name** column “Amitabh” from Worker Table:

SELECT INSTR(FIRST\_NAME, BINARY ‘a’) FROM Worker WHERE FIRST\_NAME = “Amitabh”;

* Removing **white space:**

SELECT RTRIM(column\_name) from table\_name;

* fetch worker names with **salaries >= 50000 and <= 100000.**

SELECT CONCAT(FIRST\_NAME, ' ', LAST\_NAME) As Worker\_Name, Salary

FROM worker WHERE WORKER\_ID IN

(SELECT WORKER\_ID FROM worker WHERE Salary BETWEEN 50000 AND 100000);

* **no. of workers for** each department in **the descending order**

SELECT DEPARTMENT, count(WORKER\_ID) No\_Of\_Workers FROM worker

GROUP BY DEPARTMENT ORDER BY No\_Of\_Workers DESC;

* **clone a new table** from another table.

CREATE TABLE WorkerClone LIKE Worker;

* show the **top n (say 10) records** of a table.

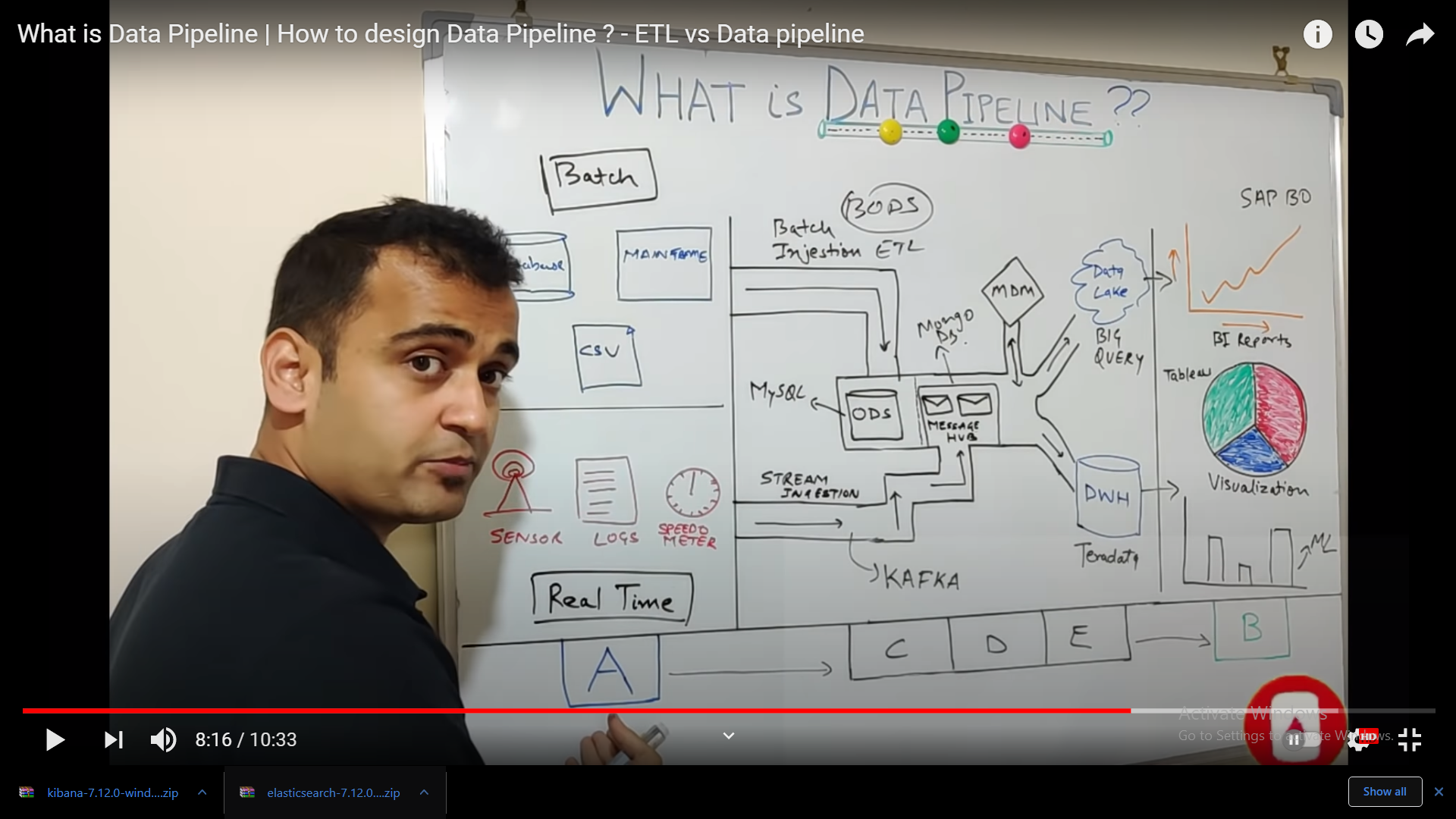
SELECT TOP 10 \* FROM Worker ORDER BY Salary DESC;

* list of employees with the same salary.

SELECT DISTINCT W.WORKER\_ID, W.FIRST\_NAME, W.Salary from Worker W, Worker W1

where W.Salary = W1.Salary and W.WORKER\_ID != W1.WORKER\_ID;

DATA PIPELINE



**ELASTICSEARCH**

**Db** stores, retrieves and manages document oriented and semi structured data. Store data in JSON formate.

ES used for:

Application search Website Search

Logging and Log Analytics

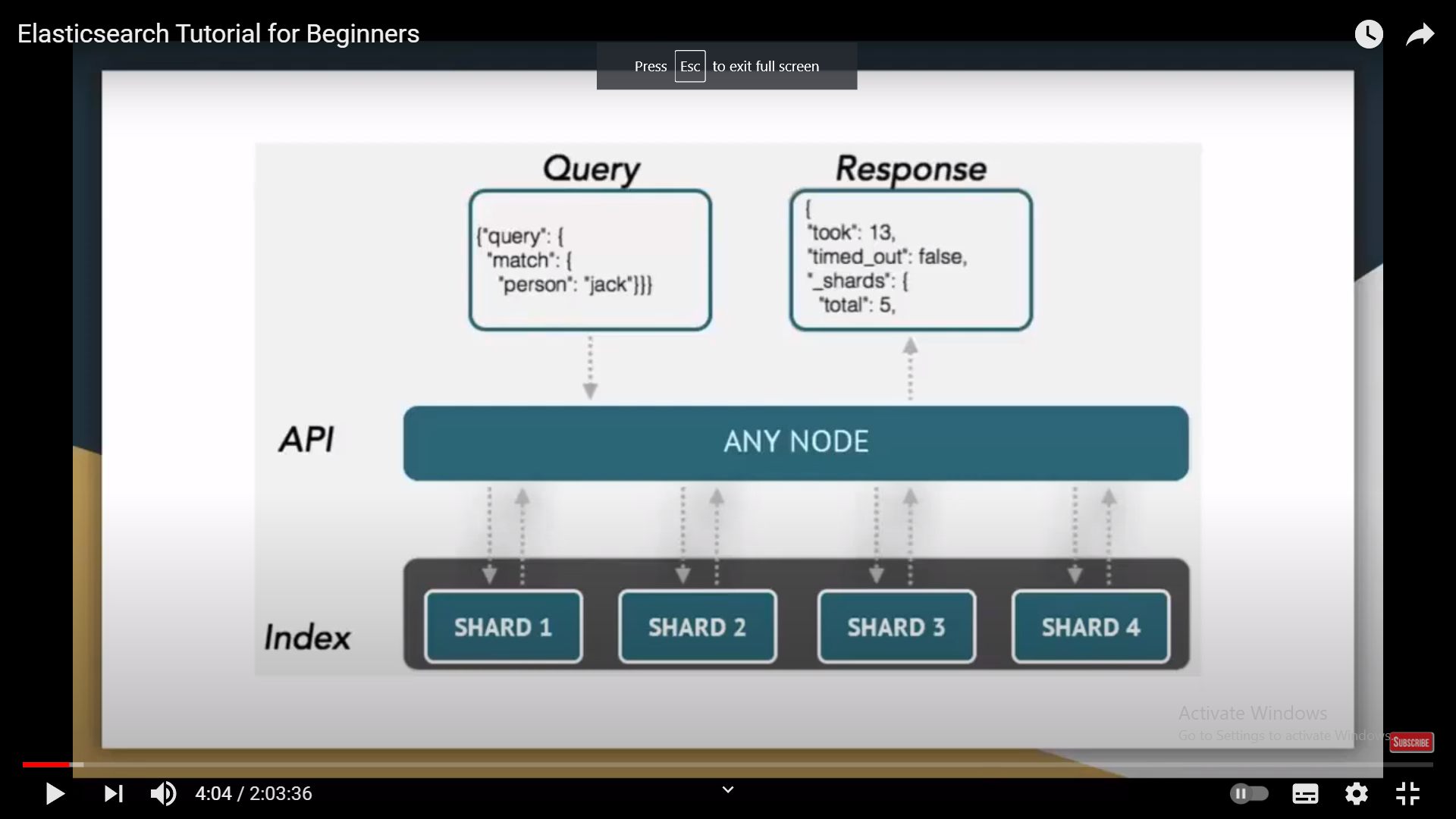
Geospatial data Analytics and visualization

Business Analytics

ES Working:

BY Data ingestion raw data is parsed, normalized and cleaned before it is indexed in Elasticsearch. Once indexed user can run complex queries against data and use complex queries to retrieve data summaries.

Kibana used for visualization of the data and share dashboards, manage the Elastic Stack



Query goes to 🡪 API and search 🡪 in an index 🡪 get results/response from same API

Response is also in JSON formate (in python useJSON normalize to parse the json Data)

ES Cluster: collection of 1 or more servers that holds the data and gives federated indexing and search capabilities.

(For RDB the node is DB instance. There can be N nodes with same cluster name.)

ES NODE: is a single server that holds data and participates in cluster’s indexing and querying.

Node can be configured to join a cluster and cluster can have many nodes.

A node is simply one ES instance.

ES SHARDS: is a subset of documents of an index

An index can be divided into many shards.

ES NEAR REAL TIME(NRT):

As soon as you create a index with data you can search for the real time data

INSTALL ES on windows:

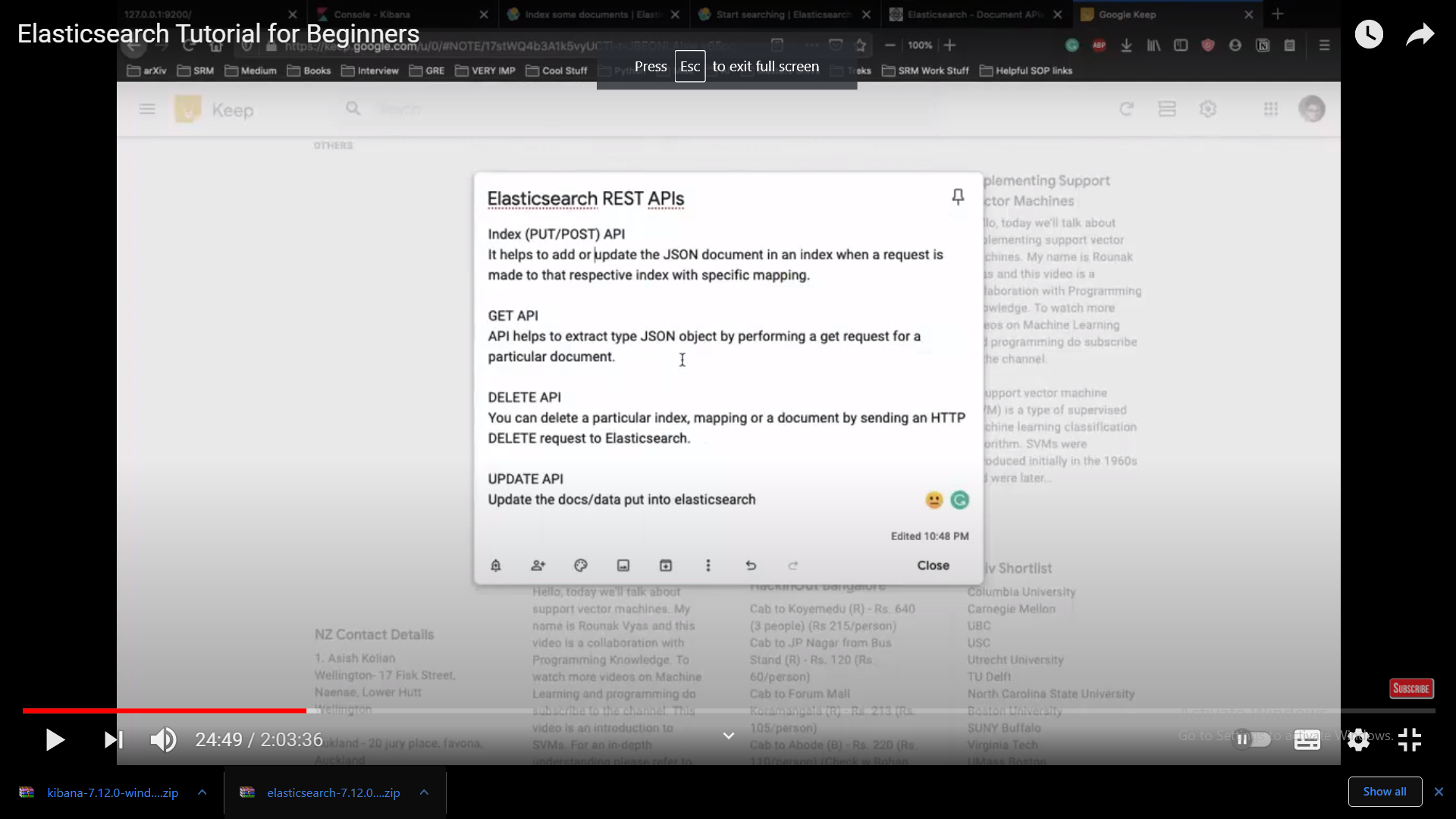
Download ES and Kibana from website and go to elasticsearch folder/bin and run “./elasticsearch” on cmd on windows.

Local port no. 9200

Install kibana: go to kibana folder then bin then run “./kibana” on cmd

Port No 5601

Kibana with Developer tool option we can curl CRUD options and there are some REST APIs for that.



Eg.

POST /customer/\_doc/7

{

“name”:”Elasticsearch again”

}

Or

GET /customers/\_search

{

“query”: {“match”: {“name”: “Elasticsearch” }}

}

Or

result = es.get(root\_url+"\_search", data={

"query": {

"range": { # expect this to return the one result on 2012-12-20

"datetime": {

"gte":"2012-12-01",

"lte":"2012-12-31",

}

}

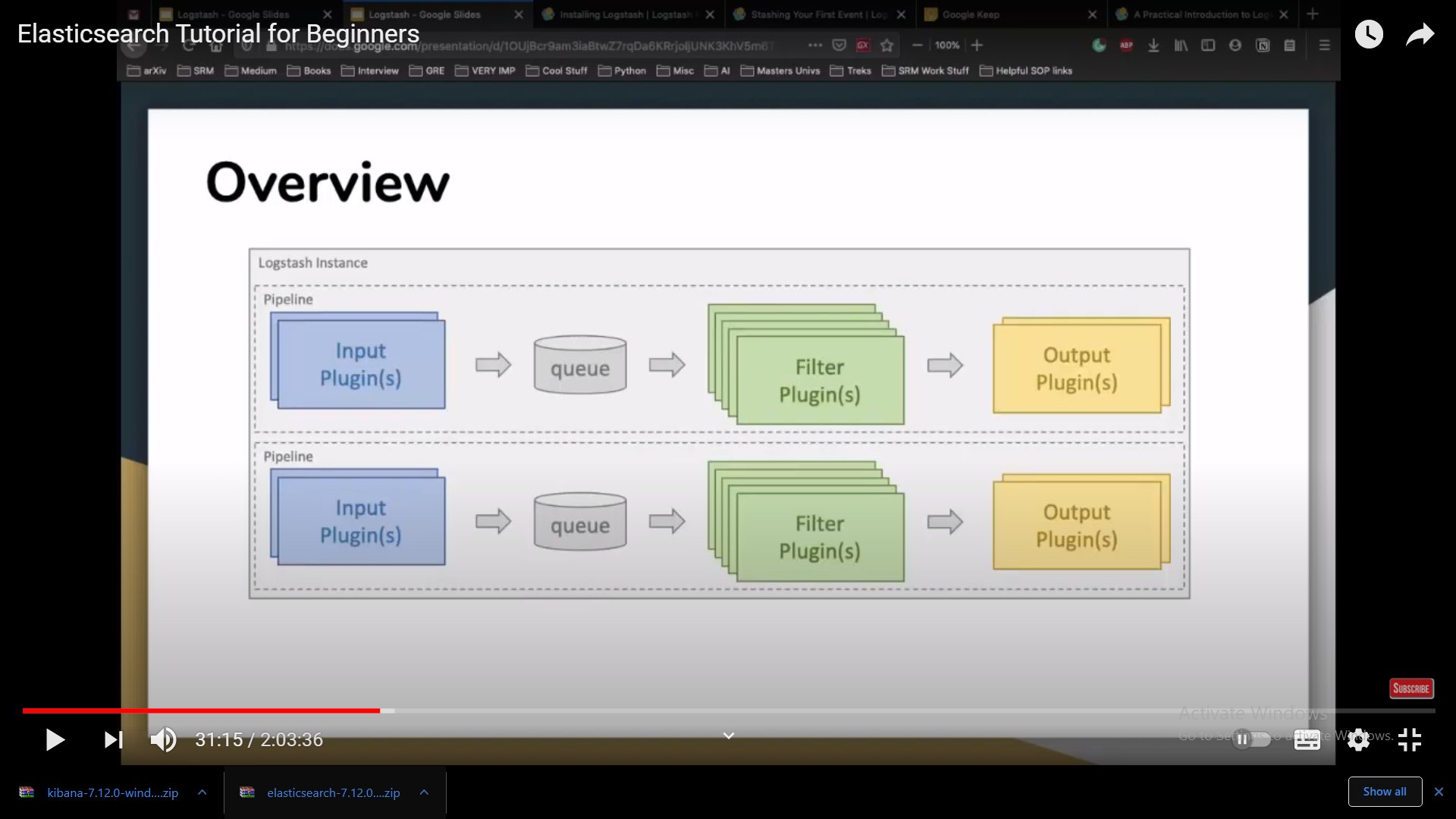
}

})

Logstash:

Is a tool based on filter/pipes pattern for gathering, processing and generating the logs or events. It helps in centralizing and making the real tme analysis of logs and events from different source.

Logstash is a plugin based data collection medium.



Processing is organized in one or more pipelines.

Input Plugin: every pipeline there are many input plugins which collects data and place in an internal queue.

Logstash can collect data from different source and send to multiple destinations.