⇒ Cleaning Data :-

⇒ To clean data, we need column expressions. Column expressions are constructed from existing columns, operators, and built-in func.

Standard SQL query commands to express transformations :- DISTINCT, WHERE, GROUP BY

⇒ Counting NOT NULL values in a table :-

Suppose, we have this table named users_dirty

user_id	timestamp	email	updated

⇒ To get count of all the records in a table, we run the command :-

```
SELECT * FROM users_dirty
```

It will also count the rows, that has all the attributes as NULL.

⇒ Now, to get number of NOT NULL values in a specific column, we need to specify that column name in COUNT() :-

SELECT COUNT(email) FROM users_dirty

it will give the no. of records in which the email column contains some value (and is not null)

⇒ To get the no. of NULL email records :-

SELECT count(*) - Count(email) FROM users_dirty

✓ Same result can also be obtained as :-

SELECT COUNT_IF email IS (L) FROM users_dirty

gives no. of NULL email records.

⇒ COUNT_IF() function :-

⇒ Takes a condition as parameter and counts the no. of records that matches the passed condition.

Ex:-

SELECT count_if (len(email)>20) FROM users_dirty

⇒ Python code to count no. of

NULL values in a column :-

```python
%python
from pyspark.sql.functions import col
usersDF = spark.read.table("users_dirty")

usersDF.selectExpr("count_if(email IS NULL)")
usersDF.where(col("email").isNull()).count()
```

can also be written as :-

- usersDF.email
- usersDF["email"]

gives same result i.e, 848 in our case

Getting all the Distinct Rows :-

```
%sql
SELECT DISTINCT(*) FROM users_dirty
```

```
%python
usersDF.distinct().count()
```

distinct() dataframe method returns all the rows that has combination of values of all records unique (if no params passed)

Casting a column to diff. datatype in PySpark :-

For it, we use the .cast() method :-

Suppose, in a table (test), we have a column containing seconds from epoch.

sec_from_epoch
1690111010

This column can be converted to timestamp as follows :-

```
%python
from pyspark.sql.functions import col
myDF = spark.table("test")

myDF2 = myDF.select(col("sec_from_epoch").CAST("timestamp"))
```

col → Datatype to which the value should be casted

CAST → casting a column to any other datatype

≥ Another way of casting in SQL :-

≥ Suppose, user_id in a table is of INT datatype. It can be casted to string as :-

column to be casted.

```
SELECT string (user_id) FROM <table_name>
```

≥ If any column data is Binary Encoded casting it to string datatype with above method will succesfully find the string representation of the binary.

Kafka data is generally binary encoded, and can be converted to human readable string by the above method.

python

```
reqDF = (spark.table ("<table_name>")
         .select (col ("user_id").cast ("string"))
        )
```

⇒ date_format() :-

⇒ Can be used in Spark-SQL Ansi SQL, as well as in pySpark.

```
date_format (<time-stamp column>,
             "<format-string>")
```

"mmm d, yyyy" ⇒ Jul 4, 2020

"HH:mm:SS" ⇒ 15:38:40

... and so on.

⇒ Used to get a date in any custom-format, from timestamp type / datetime type.

①
```
SELECT date_format(created, "mmm d, yyyy")
FROM tbl
```

②
```
from pyspark.sql import functions as F
df.withColumn("ab",
    F.date_format("created", "HH:mm:SS")
    ).display()
```

regexp_extract()

- Uses regular expressions to search for and perform string manipulation.